Ein paar wichtige Grundbegriffe der Objektorientierung am Beispiel von Java

Michael Dienert [1]

13. Oktober 2003

[1]Vielen Dank an: **Donald Knuth** und **Leslie Lamport** für dieses *sichere, geniale* Text-Satzsystem und an **Hàn Thê Thành**, dank dessen Arbeit TEX inzwischen direkt PDF-Dateien erzeugt.

Vorbemerkung

Sucht man im Internet nach Büchern und Kursen über Java, wird man von der Anzahl der Suchergebnisse fast erschlagen. Hier ein paar Beispiele: *Bruce Eckels, Thinking in Java* [2], *Hubert Partl, Java eine Einführung* [6], *Christian Ullenboom, Java ist auch eine Insel* [8] und *Guido Krüger, Goto Java 2* [9].

Meine Lieblingsbücher und persönliche Empfehlung sind *David Flanagan, Java Examples in a Nutshell* [3] und das sehr kompakte, preiswerte aber dennoch vollständige Taschenbuch *Florian Hawlitzek, Java2 Nitty Gritty* [5].

Die wichtigste Beschreibung von Java liefert aber der Erfinder SUN, selbst: es ist die Dokumentation der Java-Klassenbibliothek, die sog. Java-API (Application Programmers Interface). Diese gibt es kostenlos und ohne Registrierung im html-Format direkt bei `http://java.sun.com`. Während des Entwickelns eines Java-Programms ist es *absolut* ratsam, in einem Fenster einen html-Browser mit der API-Dokumentation von SUN zu starten.

Bei soviel geballtem Fachwissen ist es eigentlich überflüssig, noch eine Einführung zu Java zu schreiben. Der folgende Text enthält deshalb nur die allerwichtigsten Grundlagen, um einen schnellen Einstieg in die *objektorientierte Programmierung mit Java* und einen winzigen Teil der Software-Entwurfssprache **UML** (Unified Modelling Language) zu finden.

Inhaltsverzeichnis

1 Compiler und virtuelle Maschine

1.1 Maschinensprache

Ein Mikroprozessor kann nur sehr einfache, elementare Befehle ausführen. Dabei besteht ein ausführbares Programm (executable) aus einer Folge dieser sogenannten *Maschinenbefehle*. Um das Programm abarbeiten zu können, muss eine Sequenz dieser Befehle im *Arbeitsspeicher* stehen.

Dabei wird jeder Befehl durch einen Binärcode, also Folge von Nullen und Einsen, dargestellt. Während der Laufzeit des Programms, werden die Befehle nacheinander vom Speicher in den Prozessor geladen und dort entsprechend abgearbeitet.

Das Schreiben eines umfangreicheren Programms direkt mit den Maschinenbefehlen ist sehr mühsam, da die Art der Befehle nicht den typischen Aufgabenstellungen von

Software, sondern den Anforderungen der Prozessorhardware angepasst ist.

Ein weiterer Nachteil von Maschinenprogrammen ist, dass sie nur auf der zugehörigen Maschine, sprich dem zugehörigen Prozessor, laufen.

Ein gravierender Vorteil von Maschinencode soll jedoch nicht verschwiegen werden: Maschinenprogramme sind sehr kompakt, effizient und rasend schnell.

1.2 Höhere Programmiersprachen

Eine Programmiersprache stellt einen sehr begrenzten Ausschnitt der (zumeist) englischen Sprache dar. Dieser Ausschnitt ist so gewählt, dass die damit formulierten Befehle eindeutig sind.

Die höheren Programmiersprachen wurden entwickelt, da mit ihnen eine bessere Anpassung an die gegebenen Softwareaufgaben erreicht wurde.

Allerdings kann man die Befehle einer höheren Programmiersprache *nicht* direkt von einem Mikroprozessor abarbeiten lassen, sondern muss diese vorher in die Maschinenbefehle übersetzen lassen. Dieser Vorgang wird von speziellen Programmen durchgeführt, die im nächsten Kapitel beschrieben werden.

1.3 Compiler und Linker

Das Wort *Compiler* bedeutet einfach nur *Übersetzer*. Ein Compiler übersetzt ein Programm, das in einer höheren Programmiersprache wie z.B. C, Pascal oder Java geschrieben wurde, in eine Folge von Maschinenbefehlen.

Programmteile, die z.B. die Bildschirmausgabe oder das Einlesen von Zeichen von der Tastatur oder Zugriffe auf das Dateisystem usw. erledigen, werden als vorcompilierte Module aus einer Bibliothek entnommen.

Der Linker (Verbinder) ist ein Softwarewerkzeug, das diese Bibliotheks-Module und die selbstgeschriebenen Programmteile zu einer lauffähigen Anwendung zusammenbaut.

Da die Bibliotheks-Module auch wieder von der Hardwareumgebung und vom Betriebsystem abhängig sind, ist das Erstellen von Anwendungen für verschiedene Rechnerplattformen ziemlich aufwändig: möchte man das Programm auf verschiedenen Prozessoren laufen lassen, muss es für jeden Prozessortyp von einem eigenen Compiler übersetzt werden und man muss für jede Rechnerplattform die passenden Bibliotheken dazubinden (linken).

1.4 Virtuelle Maschine

Ziel bei der Entwicklung von Java war es, so weit wie möglich von der Hardware unabhängig zu werden. Um das Problem mit verschiedenen Betriebssystemen und Rechnerplattformen zu lösen, hat Sun, die Firma, die Java entwickelt hat, folgenden Trick angewandt:

Der Java-Quellcode wird nicht in die Maschinensprache eines bestimmten Prozessors übersetzt, sondern in die Befehle eines einfachen, virtuellen Prozessors (Virtual Machine VM). Dieser virtuelle Prozessor wird während der Programmlaufzeit von einer Software, der virtuellen Maschine **java** simuliert. Gleichzeitig werden die Bibliotheken (s.o.) *während* der Laufzeit und je nach Bedarf dazugebunden.

2 javac und java

Die beiden wichtigsten Anwendungen des Java-Entwicklungspakets (JDK - Java Development Kit) sind:

- der **J**ava-**C**ompiler `javac`
- die virtuelle Maschine `java`

2.1 javac

Nach der Installation des JDK unter Windows oder GNU/Linuxsteht der Java-Compiler im Verzeichnis:

`...\jdkx.x.x\bin` (Sinnlos)
bzw.
`/usr/lib/jd2sdk/bin` (GNU/Linux)

Die Pfadangabe für GNU/Linux-Systeme kann etwas variieren, als Beispiel habe ich den Pfad auf meinem Rechner angegeben.

Selbstverständlich gibt es auch für Macintosh-Anwender eine kostenlose Entwicklungsumgebung, die man bei http://developer.apple.com/java/ erhält. Allerdings sind die neueren JDKs (ab jdk1.2) nur für das neue Mac OSX erhältlich. Wer noch mit OS 9 arbeitet, muss sich mit dem jdk1.1.2 begnügen. Allerdings ist OSX soviel besser als alles bisher dagewesene, dass man unbedingt wechseln sollte.

Bei der Installation des JDK sollte die Systemvariable PATH *automatisch* auf die jeweiligen Pfade gesetzt werden, so dass das Betriebssystem den Compiler findet. Ist das nicht der Fall, muss man die entsprechenden Init-Scripte (autoexec.bat, .bashrc) von Hand editieren.

Aufruf des Compilers:

`javac` *QuellDateiName*`.java`

- Bei der Angabe der Quelldatei muss man *genau auf Gross- und Kleinschreibung* achten!
- Der Compiler erzeugt aus der Quelldatei mit der Endung .java eine Klassendatei mit der Endung .class.
- Der Dateiname dieser Klassendatei ist dabei *nicht* der QuellDateiName, sondern der Name, der im Quelltext unmittelbar auf das Schlüsselwort **class** folgt!
- Damit das nicht zur Verwirrung führt, sollte man unbedingt die Quelldatei unter dem *gleichen Namen wie die Klasse* abspeichern!

Aufgerufen wird der Compiler unter GNU/Linuxund Mac OSX in einem Terminal-Fenster, unter Windows in einer DOS-Box und auf dem Mac mit Classic OS gibt es eine grafische `javac`-Anwendung.

2.2 java

Die virtuelle Maschine steht ebenfalls im Verzeichnis

```
..\jdk1.x.x\bin
```

bzw.

```
/usr/lib/j2sdk/bin
```

Aufruf der virtuellen Maschine:

`java` *Klassenname*

- Achtung: Den Klassennamen *ohne* die Endung .class angeben!

2.2.1 Klassenpfad

Wie schon erwähnt, lädt die virtuelle Maschine während der Laufzeit benötigte Module nach. Dabei “weiss” die VM wo die Java-Standard-Klassenbibliothek mit diesen Modulen liegt. Schreibt man nun aber eigene Klassen, muss die VM diese natürlich auch finden. Hierzu wird die Systemvariable `CLASSPATH` verwendt: in `CLASSPATH` stehen alle zusätzlichen Verzeichnisse, in denen die VM nach Klassen suchen soll.

Ein typisches Beispiel für einen Eintrag in `CLASSPATH` ist der Pfad zu den Treiberklassen der JDBC-Schnittstelle. Auf einem GNU/Linux- Rechner kann der Pfad z.B. so aussehen:

```
.:/usr/local/lib/mm.mysql.jdbc-1.2c
```

Das aktuelle Verzeichnis (.) und das Startverzeichnis der JDK-Klassenbibliothek werden von der VM *standardmässig* abgesucht. Diese Verzeichnisse sollen also eigentlich *nicht* in `CLASSPATH` eingetragen werden [1] . Sagt SUN.

Da die VM per default auch das aktuelle Verzeichnis (.) absucht, ist es am einfachsten, *alle* Klassendateien, die zu einem Programm gehören (s.u.), in *einem gemeinsamen* Verzeichnis zu speichern und die VM von dort aus aufzurufen.
Ansonsten muss mit sog. *packages* gearbeitet werden. Das ist aber ein Thema, für das es eine extra Anleitung gibt und das man als Anfänger ersteinmal auf später verschieben sollte.

Noch besser als die JDBC-Klassen irgenwohin zu installieren und sie dann der VM über den `CLASSPATH` sichtbar zumachen ist es, alle zusätzlichen Klassen ins Verzeichnis

```
JAVA_HOME/j2re/lib/ext
```

zu instalieren. Dieses Verzeichnis ist extra für Erweiterungen (Extensions) vorgesehen und wird von der VM auch ohne `CLASSPATH`-Eintrag nach Klassen abgesucht. `JAVA_HOME` steht hier für das Wurzelverzeichnis der Java-Installation auf dem jeweiligen System.

2.3 Erster Test von javac und java

Bild 1 zeigt ein einfaches Java-Programm. Es hat sich als Tradition eingebürgert, eine neue Programmiersprache und die zugehörigen Werkzeuge mit diesem *Hello world!* zu testen.

```
class Hello{
   public static void main(String[] args){
      System.out.println("Hello world!");
   }
}
```

Abbildung 1: Das unvermeidliche Hello-world-Programm

Um “Hello world!” zu testen, geht man so vor:

1. erzeugen der Datei Hello.java mit einem Texteditor: **(x)emacs**, **vi, vim** unter GNU/Linux und Apple-OSX oder **UltraEdit, JOE** unter Windows.

[1] Im obigen Klassenpfad zum `mm.mysql`-Treiber musste ich aber das aktuelle Verzeichnis mit aufnehmen, da es nicht mehr abgesucht wurde, sobald die Systemvariable `CLASSPATH` nicht mehr leer war.

2. Aufruf des Compilers: `javac Hello.java`; erzeugt wird die Datei `Hello.class`. Diese Datei ist unabhängig von der verwendeten Rechnerplattform. Der Inhalt dieser Klassen-Datei wird als *Bytecode* bezeichnet.

3. Start der virtuellen Maschine: `java Hello`: es wird hello.class ausgeführt. Die Software, die die VM emuliert, eben das Programm `java` ist von der Rechnerplattform abhängig.

Abb. 2 veranschaulicht das Ganze nochmal.

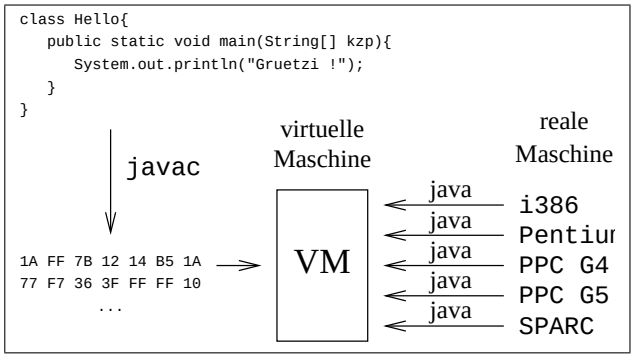

Abbildung 2: Modell der virtuellen Maschine

3 Begriffe

Das Hello-world-Programm (Abb.1) ist zwar in Java geschrieben, es ist aber noch kein richtiges, objektorientiertes Programm. Eine objektorientierte Variante würde wie in Abb.3 gezeigt aussehen. Wo steckt da nun die Objektorientierung?

Betrachten wir die drittletzte Zeile:

```
text.ausgabe();
```

Diese Syntax bedeutet: Führe die Methode `ausgabe()` des Objekts `text` aus.

Jemand der mit Programmiersprachen wie Pascal oder C vertraut ist (prozedurale Sprachen), wundert sich vielleicht, warum man das Programm nicht so formuliert hat, dass zur Ausgabe der Funktionsaufruf

```
ausgabe(text);
```

verwendet wird.

Bei der objektorientierten Programmierung steht das Objekt im Vordergrund, nicht der Funktionsaufruf:

Beispiel: `text.ausgabe()`
allgemein: *objekt.methode()*

```
class HelloOO{

   //eigenschaften
   private String grussText;

   //konstruktor
   HelloOO(String grussTextVar){
      grussText = grussTextVar;
   }

   //methoden
   public void ausgabe(){
      System.out.println(grussText);
   }

   public static void main(String[] args){
      HelloOO text =
         new HelloOO("Gruetzi wohl!");
      text.ausgabe();
    }
}
```

Abbildung 3: Objektorientiertes Hello-Programm

Um zu verstehen, was es nun im Detail mit diesem objektorientierten Hello-world-Programm auf sich hat, sollten wir ersteinmal ein paar Begriffe klären:

- **Objekt**
- **Klasse**
- **Eigenschaft** ($\equiv$ Attribut)
- **Methode** ($\equiv$ Operation)

Dabei ist anzumerken, dass die Begriffe *Eigenschaft* und *Methode* nur im direkten Zusammenhang mit Java verwendet werden. Hält man sich exakt an die Regeln der weiter unten noch ausführlich behandelten **UML**-Notation, sollte man die entsprechenden Begriffe *Attribut* und *Operation* verwenden.

3.1 Objekt

Ein *Objekt* ist die Abbildung realer Dinge wie z.B.:

- Gegenstand
- Person
- Prozess, Vorgang
- Konto, Datei, Vertrag, Schreiben, usw.

in einer "*computergerechten*" Form. Da man nicht alle Eigenschaften eines realen Objekts abbilden kann, beschränkt man sich auf diejenigen, die für die gestellte Aufgabe wichtig sind (Abb. 4).

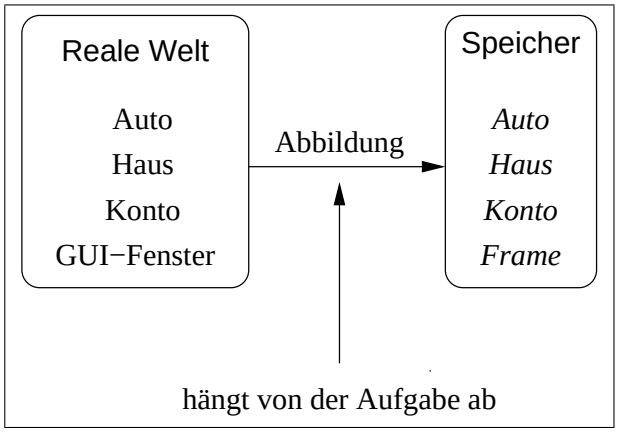

Abbildung 4: Von der realen Welt zum objektorientierten Programm

Für Objekt werden auch die synonymen Begriffe *Instanz* und *Exemplar* verwendet. Diese beiden verdeutlichen sehr schön die Beziehung zwischen Objekt und seiner *Klasse*:

3.2 Klasse

Eine Klasse ist die *Beschreibung* gleichartiger Objekte. Anschaulich kann man sich eine Klasse als **Bauplan** oder Schablone für die Objekte der Klasse vorstellen.

Daraus folgt zunächst, dass sich die Objekte einer Klasse *gleich* verhalten.

3.3 Eigenschaft

Was unter den Eigenschaften eines realen Objekts verstanden wird, ist sehr anschaulich: die Farbe eines Autos, sein Hubraum, das Baujahr ...

Ebenso besitzen aber auch die Objekte in Java Eigenschaften: das sind reservierte Plätze im Speicher, auf die mit den Variablennamen *farbe*, *hubraum* und *baujahr* zugegriffen werden kann, und die in diesem Fall die konkreten Werte `"rot", 1500, 1964` enthalten.

Auch hier werden leider teilweise abweichende Synonyme verwendet: *Attribut* und *Datenfeld*.

3.3.1 Konstanten

Eine Konstante ist eine Eigenschaft, deren Wert einmal festgelegt wird und der anschliessend nicht mehr verändert werden kann. D.h. auf eine Konstante kann nur *lesend* zugegriffen werden. Gemäss der Namenskonventionen von Java werden die Namen von Konstanten mit *Grossbuchstaben* geschrieben.

Deklariert werden Konstanten in Java mit dem Schlüsselwort **final**. Hier das unvermeidliche π-Beispiel:

```
public final double PI = 3.14159;
```

Wie alle Eigenschaften, sind in Java auch die Konstanten streng typisiert. PI in diesem Beispiel ist vom Datentyp **double**.

3.4 Methode

Wie bereits erwähnt, wird der Begriff *Methode* nur bei den Programmiersprachen Smalltalk und Java verwendet. Synonym dazu sind *Operation*, *Funktion* und *Prozedur*.

Mit den Methoden wird beschrieben, was mit dem Objekt *geschehen* kann

Im Beispiel unseres Autoobjekts könnte es folgende Methoden geben:

- *anmelden*
- *abmelden*
- *betanken*
- *lackieren*
- *starten*
- ...

4 Klassen- und Objektdiagramme in UML

Mit Diagrammen lassen sich Klassen und Objekte anschaulich und übersichtlich beschreiben. Abbildung 5 zeigt die UML-Notation für Klassen und Objekte.

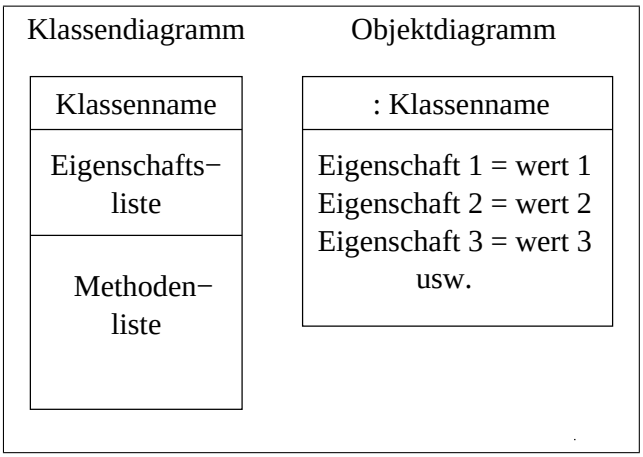

Abbildung 5: Beschreibung von Klassen und Objekten

Das Klassendiagramm besteht aus drei Teilen:

- dem Namensfeld für die Klasse
- der Eigenschaftsliste
- der Methodenliste

Der **Klassenname** steht fett gedruckt im Namensfeld. Als Klassennamen sollen *Substantive im Singular* verwendet werden.

In der Kopfzeile des Objektdiagramms steht nach einem Doppelpunkt der Name der Klasse, von der das Objekt abgeleitet wurde. Vor dem Doppelpunkt sollte eigentlich der *Objektname* stehen, aber in Java sind die Objekte *namenlos*!

4.1 Objektreferenz

Im konkreten Fall würden die Diagramme wie in Abbildung 6 dargestellt aussehen. Da ein Java-Objekt namenlos ist, kann auf das Objekt nur über die *Objektreferenz* zugegriffen werden. Die Objektreferenz ist ein *Zeiger* auf den Speicherbereich, in dem das Objekt abgelegt ist.

Das Anlegen einer Ojektreferenz in Java ist dem Deklarieren einer Eigenschaft sehr ähnlich:

```
Auto meinAuto;
```

Hiermit wird die Objektreferenz **meinAuto**, die auf ein Objekt der Klasse **Auto** zeigt, angelegt.

Erzeugt wird das Objekt dann mit folgendem Java-Code:

```
meinAuto =  new Auto("rot", 1500, 1964);
```

Erst durch das Erzeugen mit dem **new**-Operator wird das Objekt tatsächlich im Speicher angelegt. **new** ruft auch gleichzeitig den sog. Konstruktor auf, der das Objekt initialisiert. Der Konstruktor ist eine spezielle Methode, die denselben Namen wie die Klasse besitzt: **Auto**. Konstruktoren werden in Kap. 6.1 ausführlicher behandelt.

Die Deklaration und das Erzeugen des Objekts kann man auch in einer Zeile zusammenfassen:

```
Auto meinAuto = new Auto("rot", 1500, 1964);
```

Erzeugt man mehrere, verschiedene Objekte der gleichen Klasse, besitzen diese alle die gleichen Methoden, die deshalb auch nur einmal, zusammen mit der Klassendefinition im Speicher stehen.

Damit die Objekte auf ihre Methoden zugreifen können, müssen sie also ihre Klasse kennen. Aus diesem Grund enthält jedes Objekt eine automatisch erzeugte Eigenschaft (≡ Attribut), die den Verweis auf die zugeordnete Klasse enthält.

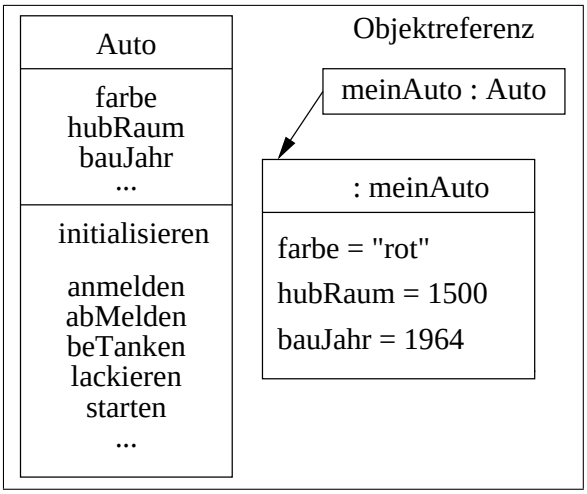

Abbildung 6: Beispiel für Klassen- und Objektdiagramm

4.2 Objekte für altgediente Pascal- und C-Entwickler

Für jemanden der jahrelang strukturiert mit Pascal oder reinem C programmiert hat, gibt es noch eine andere Sichtweise auf den Begriff Objekt:

In C und Pascal gibt es Variablen. Eine Variable ist eine recht einfache Datenstruktur, mit der man z.B. nur schlecht Datensätze erfassen kann. Eine Variable speichert nunmal nur einen Wert.

Von der einfachen Variablen gelangt man zum Variablenfeld, dem Array. Schon besser für das Bearbeiten von Daten, aber ein Array enthält nur Daten gleichen Typs. Und da

haben sich Brian und Dennis bzw. Nikolaus die Datentypen `struct` bzw. `record` ausgedacht.

In einem Record können z.B. sehr einfach die Daten einer Person (Name, Vorname, Tel.Nr., ...) gespeichert werden.

Um jetzt z.B. die Personen einer Kartei nach Namen sortiert auszugeben, schreibt man sich noch schnell eine Sortier-Prozedur.

Und dann schreibt man noch viele weitere Prozeduren, um Daten zu erfassen, zu löschen, etc. . Und je mehr Prozeduren man schreibt, umso schwieriger wird deren Verwaltung: Datentypen, Parameteranzahl, Namenskonflikte, etc. : all dies muss man berücksichtigen und im Auge behalten.

Und da wird es endlich Zeit, zur nächsthöheren Datenstruktur überzugehen: eben dem Objekt.

Ein Objekt ist nichts anderes als ein Record (=eine Struktur), der gleichzeitig alle Methoden zur Manipulation und Verwaltung seiner Datenfelder enthält.

So stellt sich ein Objekt zumindest aus der Sicht des Entwicklers dar. In Wirklichkeit werden die Methoden nicht für jedes Objekt extra gespeichert, sondern stehen nur einmal im Speicher. Die ganze Arheit mit dem Verwalten der Methoden/Prozeduren nimmt uns aber der Compiler ab.

4.3 Was Variablen und Objekte gemeinsam haben

Im vorhergehenden Kapitel (Kap. 4.2) wurde gezeigt, dass ein Objekt eine *Datenstruktur* ist, die im Speicher steht.

Mit Hilfe eines *Schubladenmodells* kann man sich den Übergang von einfachen Datentypen zu Objekten veranschaulichen. Zuersteinmal stellt man sich den Hauptspeicher als *Stapel* von *Schubladen* vor. Jede Schublade entspricht hierbei einer Variablen. Die Schublade ist vorne mit dem Variablennamen beschriftet, während der Variableninhalt dem Schubladeninhalt entspricht. Abb. 7 zeigt ein solches Speichermodell.

Einfache Datentypen kann man nun als Schubladen ohne weitere interne Unterteilung darstellen. Abb. 8 zeigt das Modell der Variablen `pi` mit dem Wert `3.1415`.

Fasst man mehrere *gleiche* Datentypen zusammen, erhält man eine Schublade, die im Inneren regelmäßig, wie ein Setzkasten, unterteilt ist. Dieser Datentyp wird *Array* oder *Liste* genannt. Abb. 9 zeigt das zugehörige Modell am Beispiel eines Arrays von 3 Zeichenketten (`String[]`). Das ganze Array hat den Namen `args`, während die einzelnen Elemente `args[0]`, `args[1]` und `args[2]` heissen [2].

Lässt man nun innerhalb der Schublade *verschiedene* Datentypen zu, kommt man zu einer Datenstruktur, die man `record` (in Pascal) oder `struct` (in C) oder einfach *Datensatz* nennt. Abb. 10 zeigt dieses Modell. Der Schubladenname entspricht hier dem Namen des Datensatzes, während die Namen der internen Fächer den Zugriff auf deren Inhalt ermöglichen.

[2] Auch in Java gibt es Arrays. Im Unterschied zu den hier dargestellten Arrays haben die Schubladen von Java-Arrays noch ein Fach mit dem Namen `length`, in dem immer die Anzahl der Array-Elemente steht.

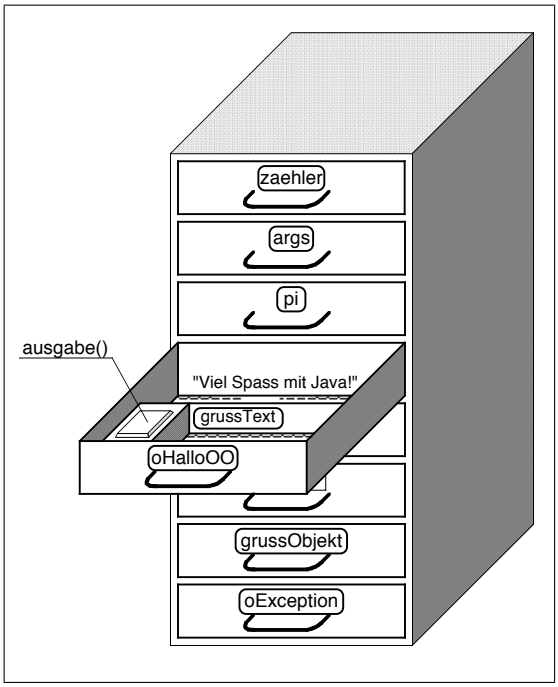

Abbildung 7: Der Speicher als Schubladenstapel

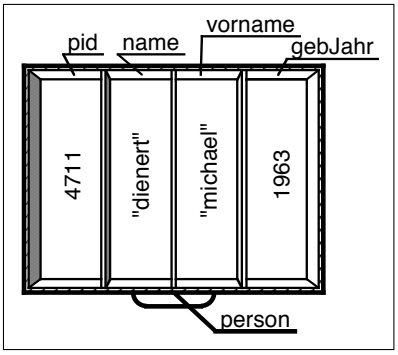

Abbildung 10: Modell eines Datensatzes

Jetzt wird die Schublade nochmals erweitert: zusätzlich zu den Unterfächern wird ein Feld mit *Funktionstasten* eingebaut. Mit diesen Tasten kann man Funktionen starten, die mit den Daten in den Unterfächern arbeiten. In Abb. 7 ist die Schublade `oHalloOO` als ein Objekt der Klasse `hallooo` (vergl. Kap. 3) dargestellt. Dies ist ein recht einfaches Objekt, es besitzt ein *Datenfeld* (Schubladenfach) für einen Grusstext und eine Methode (Funktionstaste), die diesen Grusstext auf dem Bildschirm ausgibt.

Die Sprache Java kennt nur Objekte aber keine Records oder Structures. Dies ist auch nicht nötig, denn ein Record ist ja nichts anderes als ein Objekt ohne Funktionen.

Anhand des Schubladenmodells kann man nun auch leicht erklären, wie man auf die Datenfelder (Schubladenfächer) und Funktionen eines Objekts zugreifen muss: zuerst muss man die entsprechende Schublade aufziehen, d.h. ihren Namen angeben und dann erst kann man eine darin enthaltene Funktionstaste betätigen oder auf ein Datenfeld zugreifen.

In Java-Syntax wird das mit der Punkt-Notation geschrieben:

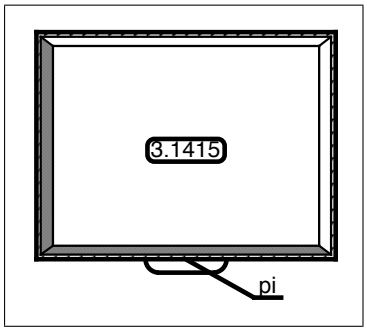

Abbildung 8: Modell einer einfachen Variablen

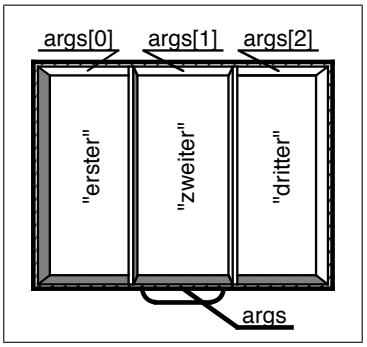

Abbildung 9: Modell eines Arrays

Zugriff auf Eigenschaft: `oHalloOO.grusstext ="Viel Spass mit Java"`

Funktionsaufruf: `oHalloOO.ausgabe()`

4.4 Unterschiede zwischen Java-Code und UML

Die beiden Diagramme aus Abb. 6 würden in Java codiert etwa wie Abb. 30 aussehen.

Im Klassendiagramm Abb. 6 sind zunächst keine Datentypen für die Eigenschaften und Rückgabewerte der Methoden mit aufgenommen. Ein ausführlicheres Diagramm der Klasse **Auto** zeigt Abb. 11 .

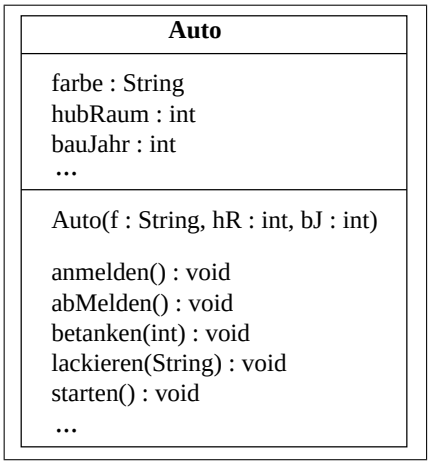

Abbildung 11: Ausführliches Klassendiagramm

In UML werden die Datentypen durch Doppelpunkt getrennt *hinter* die Eigenschaftsnamen geschrieben. Ebenso steht der Klassenname, wiederum durch einen Doppelpunkt getrennt, hinter dem Namen der Objektreferenz.

Betrachtet man nun aber den Java-Quellcode (Abb. 30) fällt auf, dass hier die Reihenfolge von Datentyp und Eigenschaftsname bzw. Klasse und Objektreferenz *vertauscht* ist und der Doppelpunkt *entfällt* [3].

5 Bestandteile eines Java-Programms

5.1 Aufbau eines Java-Programms

Ein Java-Programm besteht aus einzelnen Klassendateien (Endung .class). Diese werden von `javac` aus den Quellen (Endung `.java`) ezeugt.

Solange man keine sog. *inneren Klassen* verwendet, sollte man jede dieser Quellen in eine eigene Quelldatei schreiben (siehe auch Kap. 2.2.1).

Jede dieser Quelldateien beginnt nach evtl. import-Anweisungen mit dem Schlüsselwort `class`, gefolgt vom Klassennamen, zum Beispiel:

```
import java.io.*;
import java.awt.*;

class Beispielklasse{ .... }
```

Dieser Quellcode sollte dann also in die Datei `Beispielklasse.java` geschrieben werden.

Es gilt die Konvention:

- Klassennamen beginnen mit einem Großbuchstaben und bestehen aus Klein- und Grossbuchstaben!

5.2 Aufbau einer Java-Klasse

Bild 3 zeigt den typischen Aufbau einer Java-Klassendefinition. Nach dem Schlüsselwort **class** folgt der Klassenname und die Definition der Klasse mit ihren Eigenschaften und Methoden. Eine Sonderrolle spielt dabei die Methode `main`:

5.3 Die `main`-Methode

Wer schon mal in C programmiert hat, kennt bereits die `main`-Funktion. Die `main`-*Methode* ist bis auf winzige Unterschiede identisch. *Methode* ist nur ein anderes Wort für *Funktion*. Für ein Java-Programm muss nun folgendes gelten:

[3] Beim guten, alten *Pascal* und seinem objektorientierten Nachfolger *Oberon* von Nikolaus Wirth stimmen die Schreibweisen überein.

In einem Java-Programm *muss* es genau eine Klasse geben, die die main-Methode enthält.
Diese Klasse wird ab jetzt als *Startklasse* bezeichnet.

Besteht das Programm nur aus einer Klasse, muss eben diese Klasse die main-Methode enthalten.

Abb. 12 zeigt die main-Methode.

```
public static void main(String[] kzp){
   anweisung;
        ...
}
```

Abbildung 12: Die `main`-Methode

5.4 Funktionen und Methoden

Funktionen sind Programmteile, mit denen man immer wiederkehrende Aufgaben erfüllt. Man schreibt den Code nur an einer Stelle ins Programm, kann ihn aber über den *Funktionsnamen* von beliebigen Stellen aus aufrufen.

Eine Funktion kann vom aufrufenden Programmteil verschiedene Daten übergeben bekommen. Diese Daten nennt man *Parameter*.

Eine Funktion kann einen Wert an den Programmteil, der sie aufgerufen hat, zurückgeben. Das ist der *Rückgabewert*.

In Java wird statt des Begriffs Funktion der Begriff *Methode* verwendet.

Methoden schreibt man so:

- Der Methodenname beginnt mit einem Kleinbuchstaben und besteht aus Gross- und Kleinbuchstaben!
- Direkt hinter dem Methodennamen steht eine runde Klammer. Kein Leerzeichen zwischen Namen und Klammer!
- In der runden Klammern kann eine *Parameterliste* stehen.
- Vor dem Methodennamen steht der *Datentyp* des Rückgabewerts.
- Das optionale Schlüsselwort **static** besagt, dass die Methode sofort in den Speicher geladen wird.
- Das optionale Schlüsselwort **public** besagt, dass die Methode von allen anderen Klassen aus sichtbar ist.

Parameterliste

Die Parameterliste einer Methode sieht so aus:

- Werden gar keine Parameter benötigt, steht eine *leere* runde Klammer:
  `testMethode();`
- Ein einzelner Parameter wird so definiert: *datentyp variablenname*:
  `testMethode(double dezimalZahl);`
- Hat man mehrere Parameter, werden diese durch Kommas getrennt:
  testMethode(double dezimalZahl, String vorName);

Datentyp des Rückgabewerts

Für den Typ des Rückgabewerts gilt:

- Wird kein Wert zurückgegeben, *muss* das Schlüsselwort `void` stehen:
  `void testMethode();`
- Wird ein Wert zurückgegeben, steht vor dem Methodennamen der Typ des Rückgabewerts:
  `int testMethode();`

5.5 Datentypen und Eigenschaften

Vorneweg gleich wieder eine Namenskonvention:

Es gilt die Konvention:

- Eigenschaftsnamen beginnen mit einem Großbuchstaben und bestehen aus Klein- und Grossbuchstaben!

Java ist eine sehr streng typisierte Sprache. Das heisst, man muss dem Compiler *immer* mitteilen, welchen Datentyp eine Eigenschaft haben soll. Java lässt dabei folgende Datentypen zu:

- `char` (16bit)
- `boolean` (true / false)
- ganze Zahlen, *alle* mit Vorzeichen: `byte, short, int, long` (8, 16, 32, 64 bit)
- reelle Zahlen: `float, double` (32, 64 bit)
- Zeichenketten: `String`

Da die heutigen Rechner üppig mit Speicher ausgestattet sind, sollte man für reelle Zahlen der Einfachheit halber immer `double` nehmen. Der Standardtyp für ganze Zahlen ist int.

Wie bereits erwähnt, beginnen nur Klassennamen mit einem Grossbuchstaben. *String* als Typ für Zeichenketten ist also eine vordefinierte Klasse.

Einer Eigenschaft darf man nur Werte die vom gleichen Typ sind zuweisen. Soll dabei eine Typumwandlung stattfinden, muss man den neuen Datentyp in Klammern vor den Wert schreiben:

```
int ganzzahl = 100;
short kurzzahl = 10;
kurzzahl = (short)ganzzahl;
```

6 Weitere Begriffe

Leider kommt man mit den vier Grundbegriffen *Eigenschaft*, *Methode*, *Klasse* und *Objekt* noch nicht ganz aus. Deshalb geht der Definitionsmarathon zunächst noch ein bisschen weiter:

6.1 Konstruktor

Einen Konstruktor haben wir im HelloOO-Programm bereits kennengelernt (zumindest diejenigen, die den Quelltext aus Abb. 3 durchgelesen und evtl. ausprobiert haben.): Es ist die Methode `HelloOO(String grussTextVar)`.

Ein Konstuktor ist also nichts anderes als eine Methode (das erkennt man an der *runden* Klammer nach dem Namen), die allerdings folgende, besondere Eigenschaften besitzt:

- der Konstruktor wird automatisch immer dann aufgerufen, wenn ein neues Objekt mit **new** erzeugt wird
- der Konstruktor hat *immer* denselben Namen wie die Klasse, in der er definiert ist; sein Name beginnt deshalb auch mit einem Grossbuchstaben

- der Konstruktor liefert *keinen* Wert zurück; es darf daher auch kein Rückgabetyp angegeben werden, auch nicht **void**
- man kann dem Konstruktor allerdings Parameter mitgeben, mit denen das Objekt initialisiert wird
- wird der Konstruktor nicht definiert, verwendet Java einen Default-Konstruktor
- es kann pro Klasse mehrere Konstruktoren geben, die sich aber in Typ oder Anzahl der Parameter unterscheiden müssen

Da mir selbst der Begriff Konstruktor und sein Sinn und Zweck auf Anhieb nicht klar waren, habe ich im Anhang eine Übungsaufgabe untergebracht, die vielleicht dabei helfen kann zu verstehen, was ein Konstruktor ist.

6.2 Überladen von Methoden

Eine Klasse kann mehrere Methoden mit *gleichem* Namen enthalten, die sich aber in Typ oder Anzahl der Parameter unterscheiden müssen. Zum Beispiel:

```
// erste methode
void lackieren( String farbe ) {
   ...
}

// zweite methode
void lackieren( String farbe,
             boolean metallic) {
   ...
}
```

Man sagt nun *lackieren* ist eine *überladene* Methode.

Nun kann die Klasse AutoTest ein Auto normal lackieren oder mit Metallic-Lack versehen. In beiden Fällen ist der Methodenname *lackieren(...)*:

```
//normale Lackierung
meinAuto.lackieren("rot");

//Metallic-Lack
meinAuto.lackieren("silber", true);
```

Das Java-Laufzeitsystem sucht automatisch die Methode, die zu den übergebenen Parametern passt.

Da der Konstruktor ebenfalls eine Methode ist, kann man ihn auch überladen.

6.3 Botschaften

Der Begriff *Botschaft* steht synonym für das Aufrufen einer Methode. Man sendet also z.B. dem Objekt *meinAuto* die Botschaft:

```
meinAuto.lackieren("rot");
```

Dabei besteht die eigentliche Botschaft aus dem Methodenaufruf: `lackieren("rot")` . Natürlich muss man den Adressaten der Botschaft auch angeben, im Beispiel also: `meinAuto`. Adresse und Botschaft werden durch einen Punkt getrennt.

6.4 Klassenattribut und Klassenmethode

Bisher haben wir Eigenschaften ($\equiv Attribute$) und Methoden nur für Objekte definiert. Da aber auch Klassen durchaus Eigenschaften besitzen können auf die man auch mit Methoden zugreifen kann, macht es Sinn auch Klassenattribute und -methoden zu definieren.

6.4.1 Klassenattribut

Die Definition des Begriffs *Klassenattribut* ist eigentlich ganz einfach: da Attribut synonym zu Eigenschaft ist, beschreibt ein Klassenattribut eine Eigenschaft *der Klasse*.

Wie die Methoden auch, ist das Klassenattribut nur einmal im Speicher vorhanden, und es kann ebenso wie die Methoden von allen Objekten der Klasse genutzt werden.

Im UML-Klassendiagramm werden die Klassenattribute *unterstrichen* dargestellt (siehe Abb. 13).

Daraus folgt natürlich auch, dass der Wert des Klassenattributs für alle Objekte der Klasse gleich ist. Aus diesem Grund werden Klassenattribute auch als *statische* Eigenschaften bezeichnet und in Java mit dem Schlüsselwort **static** deklariert.

Abbildung 31 zeigt ein Beispiel in Java.

Die Anzahl der gemeldeten Autos -`anzahlGemeldet`- ist in diesem Beispiel eine Eigenschaft der Klasse *Auto*. Der Datentyp ist **int** und mit dem Schlüsselwort **static** wird `anzahlGemeldet` als Klassenattribut deklariert.

Natürlich kann man auch Klassenattribute mit **final** als unveränderlich deklarieren:

```
public static final double E = 2.718282;
```

Solch eine **static final** - Konstante entspricht der `#define name ersatztext` - Deklaration von C (symbolische Konstante).

6.4.2 Klassenmethode

Klassenmethoden oder auch *statische Methoden* sind Methoden, die *ausschliesslich* auf Klassenattribute zugreifen können. In Java werden sie wiederum mit dem Schlüsselwort **static** deklariert, während in der UML-Notation der Methodenname unterstrichen wird (siehe Abb. 13).

In Abbildung 31 ist die statische Methode `getAnzahlGemeldet()` definiert. Sie wird in der Klasse *AutoTest* mit `meinAuto.getAnzahlGemeldet()` bzw. `deinAuto.getAnzahlGemeldet()` aufgerufen.

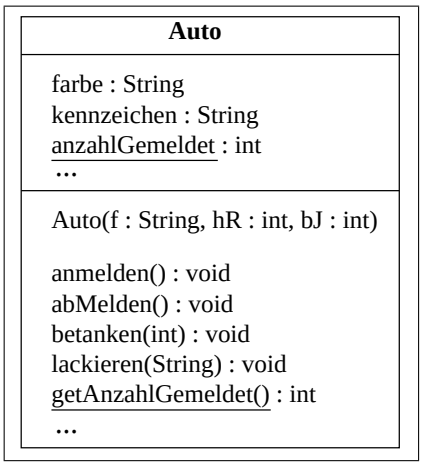

Abbildung 13: Eine Klasse mit statischer Methode

Statische Methoden können auch aufgerufen werden, bevor man Ojekte von der Klasse abgeleitet hat. Das heisst, man kann auch einer Klasse Botschaften schicken:

```
Auto.getAnzahlGemeldet()
```

Wie bei den Botschaften an Objekte, steht vor dem Methodennamen der Adressat, in diesem Fall also der Klassenname, getrennt durch einen Punkt [4].

Bitte beachten Sie, dass ein solcher Aufruf nur bei statischen Methoden (eben den Klassenmethoden) funktioniert:
Die nicht-statischen Methoden greifen auf Werte von Objekteigenschaften zu. Diese Eigenschaften werden aber erst zusammen mit dem Objekt im Speicher angelegt. Bevor ein Objekt existiert, existieren also gar keine Eigenschaftswerte, auf die eine Methode zugreifen könnte. Sind dann aber Objekte angelegt, spielt es keine Rolle mehr, ob ich über das Objekt auf die statische Methode zugreife oder direkt über die Klasse. Alle drei vorangehenden Beispielbotschaften liefern dann dasselbe Ergebnis.

Warum das so ist, ist klar: die Klassenmethode steht ja auch nur an einer Stelle im Speicher. Unabhängig vom Adressaten der Botschaft wird also jedesmal der selbe Code ausgeführt. Auch Objektmethoden stehen unabhängig von der Anzahl der erzeugten Objekte nur einmal im Speicher. Der Unterschied zwischen Objektmethoden und statischen Methoden ist folgender:

- eine Objektmethode, man sagt auch Instanzmethode, wird erst dann in den Speicher geladen, wenn ein Objekt erzeugt wird
- eine statische Methode wird sofort, beim Start der virtuellen Maschine, in den Speicher geladen

[4] Die Botschaft im obigen Beispiel würde als Antwort selbstverständlich den Wert 0 zurückgeben, da ja zunächst noch keine Auto-Objekte existieren, also auch keine Autos angemeldet seien können

Aus Gründen der besseren Lesbarkeit von Programmen, sollte man allerdings Klassenmethoden **nicht** über ein Objekt, sondern *nur über die Klasse* aufrufen, schliesslich handelt es sich ja um eine *Klassen*methode.

6.5 Kapselung

Autos werden in erster Linie über das Aussehen ihrer Karosserie vermarktet. Ein gutes Karosseriedesign soll zunächst die Eigenschaften von Motor, Fahrwerk usw. verbergen.

Beim Entwurf von Klassen gilt dasselbe: die Eigenschaften der Objekte sollten von aussen *nicht* sichtbar sein (Geheimnisprinzip). Der Zugriff auf die Eigenschaften darf nur über von aussen *sichtbare* Methoden erfolgen, die man **set**- und **get**-Methoden nennt:

In der UML-Notation gehört zu *jeder* Eigenschaft *automatisch* eine set- und eine get-Methode. Dies ist nur eine Konvention, keine Java-Sprachregel. D.h. es wird vom Compiler nicht erzwungen.
Die Sichtbarkeit wird dabei mit folgenden Symbolen gekennzeichnet:

- **-** für **private**
- **+** für **public**

Alle Eigenschaften und Methoden die **private** definiert werden, sind nur innerhalb der selben Klasse sichtbar und können damit nur von innerhalb dieser Klasse aus angesprochen werden, aber **nicht** von deren Unterklassen! Alles was mit **public** definiert wird, ist von *überall* sichtbar.

Public und **private** sind die beiden wichtigsten Schlüsselwörter zum Steuern der Sichtbarkeit. Daneben gibt es noch das Schlüsselwort **protected**, das in UML mit einem # symbolisiert wird. Elemente, die als **protected** deklariert sind, sind in der Klasse *und* deren Unterklassen sichtbar. Die Anwendung von **protected** sollte aber vermieden werden, da damit das Geheimnisprinzip aufgeweicht wird.

Das Gebot der Kapselung ist zwar eine sehr restriktive Einschränkung, es stellt aber sicher, dass die Eigenschaften nur in wohldefinierter Weise verändert werden können:

Eine Änderung oder Abfrage der Eigenschaften eines Objekts ist nur über seine **set**- und **get**-Methoden möglich. Ein direkter Zugriff wird durch die “Unsichtbarmachung” der Eigenschaften verhindert.

Abb. 14 zeigt zur Abrundung noch ein UML-Beispiel.

7 Vererbung

Und noch ein Begriff!

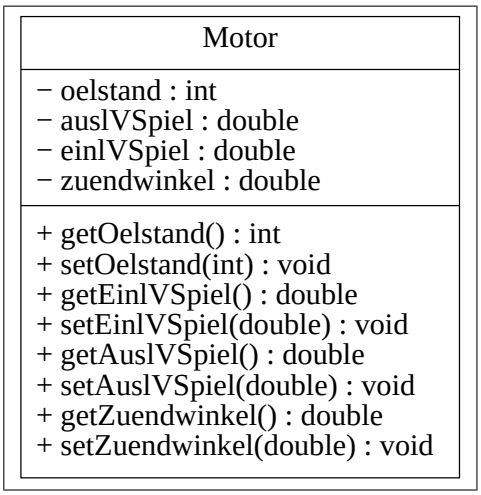

Abbildung 14: set- und get-Methoden in UML-Notation

Und zwar ein so wichtiger, dass ich ihn nicht bei den anderen Begriffen aufgeführt habe, sondern dafür ein eigenes Kapitel eröffne.

Vererbung heisst einfach, dass man Klassen durch weitere Präzisierung in *Unterklassen* einteilt. Man gelangt auf diese Weise zu einer *Baumstruktur*, die man *Klassenhierarchie* bzw. *Vererbungsstruktur* nennt. In Abb. 15 ist dies am Beispiel der Klasse **Kraftfahrzeug** demonstriert. Diese Oberklasse besitzt nur diejenigen Eigenschaften und Methoden, die allen Kraftfahrzeugen gemeinsam sind.

Die beiden Unterklassen **PKW** und **LKW** besitzen dagegen nur noch die für sie charakteristischen Eigenschaften und Methoden. Alle gemeinsamen Eigenschaften und Methoden *erben* sie von ihrer Ober- oder **Super**klasse. Die Unterklassen nennt man auch *abgeleitete* Klassen.

7.1 Endgültige Klassen

Möchte man verhindern, dass eine Klasse Erben bekommt, muss man sie mit dem Schlüsselwort **final** als endgültige Klasse deklarieren.

7.2 Klassenhierarchie

Leider hat die Vererbung nicht nur Vorteile. Zum Beispiel muss man um eine abgeleitete Klasse verstehen zu können, auch deren Superklasse kennen und durch die Vererbung sind die Methoden der Superklasse für die abgeleitete Klasse sichtbar, d.h. das Geheimnisprinzip wird aufgeweicht.

Aus diesen Gründen sollte die Vererbung nur so sparsam wie möglich angewendet werden. Sinnvoll ist sie nur dann, wenn man damit eine Klassenhierarchie erzeugt: die abgeleitete Klasse *muss* eine *Spezialisierung* der Superklasse darstellen. Dieser Fall ist immer dann gegeben, wenn man sagen kann:

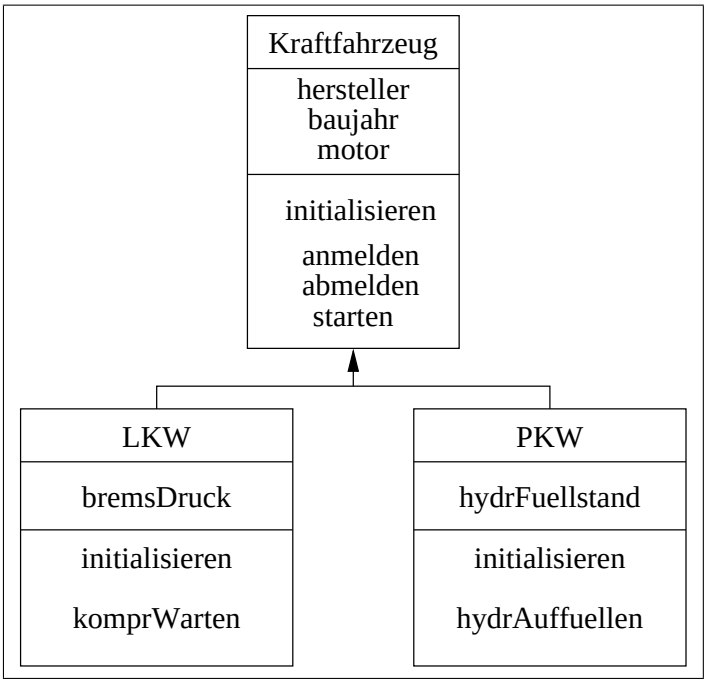

Abbildung 15: Baumstruktur bei der Vererbung

Ein Objekt der abgeleiteten Klasse *ist ein* Objekt der Superklasse. Beipiel: Ein PKW *ist ein* Kraftfahrzeug, ein LKW *ist ein* Kraftfahrzeug.

7.3 Abstrakte Klassen

Bevor wir nun dieses Diagramm (Abb. 15) in Java-Quellcode übersetzen können, fehlt uns noch der Begriff *abstrakte* Klasse.

Im Bild 15 fällt auf, dass die Klasse von Kraftfahrzeugen gar keine Bremse hat. Jedes Kraftfahrzeug ist aber entweder ein PKW oder ein LKW und diese haben jeweils ein Bremssystem.

Ein Objekt direkt von der Klasse **Kraftfahrzeug** zu bilden macht deshalb keinen Sinn: es gibt eben nur PKW oder LKW und natürlich noch Krafträder, die sind in Abb. 15 aber nicht dargestellt.

Eine Klasse, von der keine Objekte gebildet werden können, wird *abstrakte Klasse* genannt. Im UML-Klassendiagramm wird der Name abstrakter Klassen *kursiv* geschrieben.

Damit sieht der Quellcode des Beispiels nun wie in Abb. 32 und 33 aus. Die Superklasse **Kraftfahrzeug** wird mit dem Schlüsselwort **abstract** als abstrakte Klasse definiert.

In Abb. 33 ist der Quellcode für die Klassen **PKW** und **LKW** gezeigt. Beide *erben* von ihrer Superklasse. Die Java-Syntax mit dem Schlüsselwort **extends** hierzu ist:

```
class klassenname extends superklasse
```

7.4 Einfach- und Mehrfachvererbung

Bei der Vererbung wird nochmals zwischen Einfach- und Mehrfachvererbung unterschieden. Die Begriffe sind so definiert:

- Bei der Einfachvererbung hat jede abgeleitete Klasse *genau eine* Superklasse. Die Klassenhierarchie hat eine exakte Baumstruktur.
- Bei der Mehrfachvererbung wird zugelassen, dass eine abgeleitete Klasse Eigenschaften und Methoden von **mehreren** Superklassen erbt. Die Klassenhierarchie hat keine Baumstruktur mehr, es entstehen geschlossenen Maschen.

Abbildung 16 zeigt ein Beispiel für Mehrfachvererbung. Ein Wohnmobil erbt Eigenschaften und Methoden von einem Kraftfahrzeug und von einer Behausung. Bei der Mehrfachvererbung kann es zu Namenskonflikten kommen, wenn die Superklassen Eigenschaften oder Methoden gleichen Namens enthalten. Es muss daher festgelegt werden, wie diese Mehrdeutigkeiten wieder aufgelöst werden.

In Java haben Sie dieses Problem jedoch nicht: Mehrfachvererbungen sind in Java *nicht* möglich.

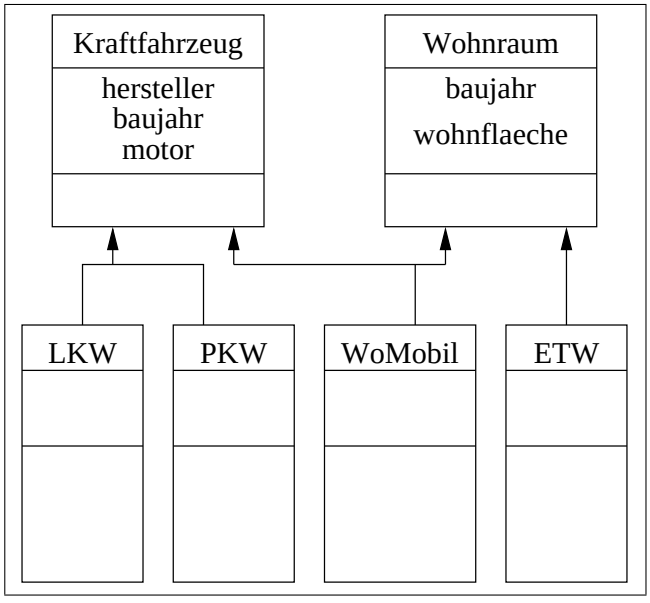

Abbildung 16: Beispiel für Mehrfachvererbung

7.5 Überschreiben von Methoden

Bei unserem KFZ-Beispiel haben wir in der Superklasse die Methode **starten** definiert. Nun ist es aber so, dass PKW heutzutage immer durch drehen des Zündschlüssels gestartet werden, während manche LKW noch einen extra Startknopf haben.

D.h. bei LKWs bräuchte man eigentlich eine andere start-Methode. Bild 17 zeigt die Lösung dieses Problems: in der abgeleiteten Klasse **LKW** wird nochmals die Methode **starten** definiert. Und diese nochmalige Definition von **starten** *überschreibt* die Methode der Superklasse.

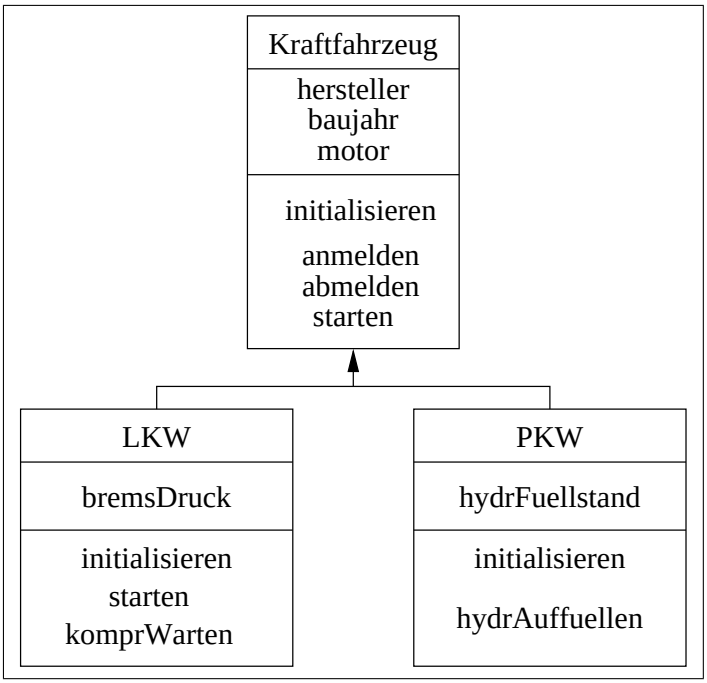

Abbildung 17: Überschriebene Methode **starten**

Das heisst:

- In der Klasse **PKW** braucht die Methode starten nicht definiert zu werden; die Methode ist schon wie gewünscht in der Superklasse definiert und die abgeleitete Klasse erbt diese Methode.
- In der Klasse **LKW** wird die Methode **starten** durch eine neue Methodendefinition *gleichen Namens* überschrieben.

Hat man nun einen LKW, der auch durch Schlüsseldrehen gestartet wird, wäre es doch praktisch, wenn man wieder an die überschriebene Methode rankäme: dies wird durch das Schlüsselwort **super** möglich gemacht. Der Aufruf

```
super.starten();
```

ruft direkt die Methode der Superklasse auf.

7.5.1 Endgültige Methoden

Wenn man verhindern will, dass eine Methode überschrieben werden kann, muss man sie mit dem Schlüsselwort **final** als endgültige Methode deklarieren.

7.6 Verbergen von Eigenschaften

Das Verbergen von Eigenschaften funktioniert analog zum Überschreiben von Methoden: ist in der abgeleiteten Klasse eine neue Eigenschaft definiert, die **denselben** Namen wie eine Eigenschaft der Superklasse hat, wird die Eigenschaft der Superklasse *verborgen*.

Mit dem Schlüsselwort **super** gelangt man auch hier wieder an die Eigenschaft der Superklasse.

7.7 Abstrakte Methoden

Das Problem mit dem unterschiedlichen Startverhalten von KFZ lässt sich auch mit Hilfe einer *abstrakten Methode* lösen.

Abstrakte Methoden sind Methoden, die nur aus dem Deklarationsteil bestehen. Ihr Methodenrumpf ist leer. **Sie dürfen nur in abstrakten Klassen auftreten** (!) und dienen einfach als Platzhalter für konkrete, gleichnamige Methoden, die in den abgeleiteten Klassen definiert werden.

Anders ausgedrückt: die abstrakten Methoden *müssen* in den abgeleiteten Klassen von gleichnamigen, konkreten Methoden überschrieben werden.

Mit Hilfe der abstrakten Klassen kann man erzwingen, dass alle abgeleitete Klassen bestimmte Methoden enthalten müssen, die *alle den gleichen Namen haben, deren Quellcodes aber verschieden sein können*.

Zurück zum KFZ-Start-Beispiel: die Superklasse enthält die abstrakte Methode **starten**, in den abgeleiteten Klassen wird festgelegt, wie gestartet wird. (Bild 18, damit die Motorräder nicht zu kurz kommen)

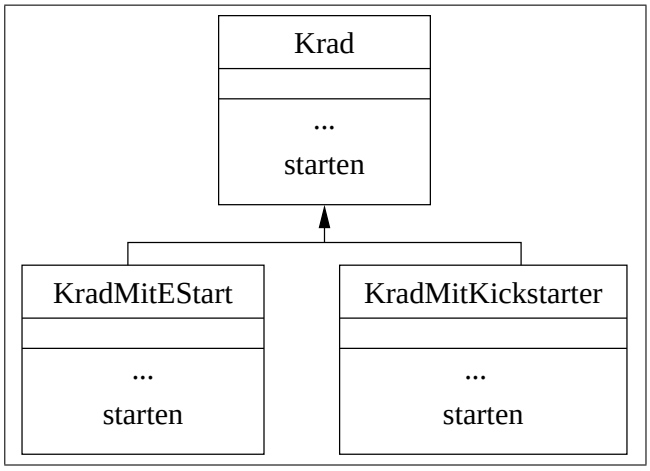

Abbildung 18: Abstrakte Methode **starten**

8 Polymorphismus

Zunächst betrachten wir die Klasse der Motorradfahrer (Abb. 19).

Abbildung 19: Motorradfahrer-Klasse

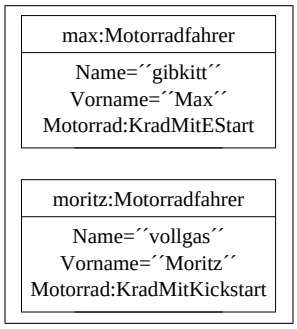

Abbildung 20: Zwei Motorradfahrer-Objekte

In diesem Klassendiagramm taucht als Eigenschaft **Motorrad** auf. **Motorrad** wiederum ist ein Objekt der Klasse **Krad**(vgl. Abb. 18). [5] Da **Krad** eine abstrakte Klasse ist, heisst das, dass ein Motorradfahrer ein **KradMitEStart** oder ein **KradMitKickstart**er besitzen muss.

Nun werden zwei Motorradfahrer-Objekte erzeugt (Abb. 20). Anders als in der Klassendefinition ist das Motorrad nun nicht vom Typ Krad, sondern es werden die abgeleiteten Klassen verwendet. Das ist erlaubt! Generell gilt:

[5] Streng genommen ist Motorrad nicht das Objekt, sondern nur die Referenz auf ein Objekt, da Objekte in Java namenlos sind.

Ein Objekt einer abgeleiteten Klasse kann überall da verwendet werden, wo ein Objekt der Superklasse erlaubt ist.

Jetzt sollen Max und Moritz ihre Motorräder starten. Die Botschaften an die Objekte sehen dann so aus:

```
max.motorrad.starten();
moritz.motorrad.starten();
```

Es wird also dieselbe Botschaft (`starten()`) an Objekte *verschiedener Klassen einer Vererbungshierarchie* gesendet: das Objekt **motorrad** von **max** und **moritz** hat eben im einen Fall einen Kickstarter und im anderen Fall einen E-Starter.

Damit die Motorräder auch wirklich anspringen, muss die Botschaft von den Objekten jeweils richtig interpretiert werden, denn die Startmethode hat zwar immer den gleichen Namen, unterscheidet sich aber von Objekt zu Objekt: Das heisst, sie hat verschiedene "Erscheinungsformen", ist also polymorph:

Polymorphismus bedeutet, dass derselbe Methodenaufruf ($\equiv$ dieselbe Botschaft) an Objekte verschiedener Klassen einer Klassenhierarchie gesendet werden kann und dass die Adressaten die Botschaft richtig interpretieren.
Er ermöglicht den gleichen Namen für ähnliche Methoden von Objekten verschiedener Klassen zu verwenden.

9 Schnittstellen

Eine Schnittstelle ist eine Klasse, deren Methoden alle abstrakt sind, die also nur aus dem Deklarationsteil bestehen. Eine Schnittstelle darf auch keine Eigenschaften besitzen [6] , weshalb in der UML-Notation der Eigenschaftsteil wegfällt (Abb. 22). Der Schnittstellenname wird *nicht* fett geschrieben und zur weiteren Abgrenzung gegen UML-Klassendiagramme, wird vor den Schnittstellennamen $\ll$interface$\gg$ gesetzt.

In Java wird eine Schnittstelle als **interface** bezeichnet. Deklariert wird eine Schnittstelle ähnlich wie eine Klasse, aber anstelle des Schlüsselworts **class** steht das Schlüselwort **interface** (Abb. 21):

```
public interface Wohnraum {
   public void setRaumtemperatur();
   public void betaetigeToilettenspuelung();
               ...
}
```

Abbildung 21: Java-Syntax zu `Interface`

Ein Interface allein hat keinerlei Funktionalität. Alle Methoden sind leer. Erst wenn eine Klasse alle diese Methoden *implementiert*, ($\equiv$verwirklicht), kann man die Metho-

[6] In Java darf ein Interface konstante Klassenattribute (static final ...) enthalten.

den verwenden. Dieses *Implementieren* einer Schnittstelle wird in UML-Diagrammen mit einem **gestrichelten** Vererbungspfeil dargestellt (Abb. 22).

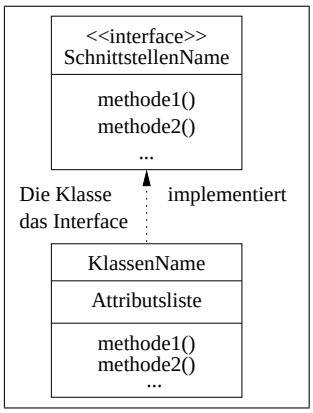

Abbildung 22: UML-Notation einer Schnittstelle

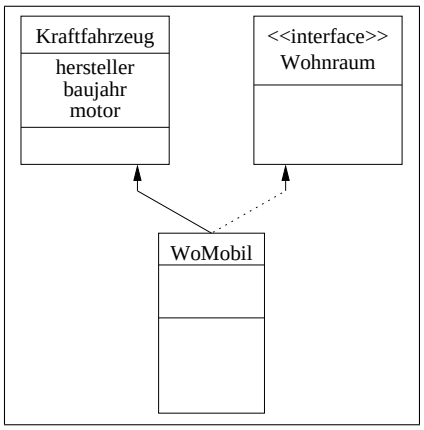

Abbildung 23: Ersatz-Mehrfachvererbung mit Interface

Das Implementieren ist also nichts anderes, als eine spezielle Form der Vererbung. Die Java-Syntax hierzu lautet:

```
class ETW implements Wohnraum
```

Nun kann eine Klasse auch *mehrere* Interfaces implementieren. Oder sie kann eine Superklasse erweitern *und* ein Interface implementieren. Hiermit wird in Java eine Ersatzlösung für die ansonsten nicht erlaubte Mehrfachvererbung möglich.

Unsere Wohnmobilklasse liesse sich demnach erzeugen wie in Abb. 24 angedeutet. Das zugehörige UML-Diagramm zeigt Bild 23.

```
class WoMobil extends Kraftfahrzeug implements Wohnraum {
  // WoMobil erbt die Methoden von Kraftfahrzeug muss
  // aber die Methoden von Wohnraum implementieren }
```

Abbildung 24: : Ersatz-“Mehrfachvererbung”

Literatur

[1] Helmut Balzert. *Lehrbuch Grundlagen der Informatik.* Spektrum Akademischer Verlag GmbH, Heidelberg . Berlin, 1999.

[2] Bruce Eckels. *Thinking in Java.* Prentice Hall 1998. Diverse Postscript und html-Versionen sind kostenlos im Internet erhältlich.

[3] David Flanagan. *JAVA in a Nutshell, deutsche Ausgabe.* O´Reilly Verlag, Köln 1998.

[4] Kalle Dallheimer. *Jetzt mach ich´s selber. Ein Programmierkurs für Einsteiger.* Zeitschrift c´t, 1999 Heft 11 bis 16. Heise Verlag, Hannover, 1999.

[5] Florian Hawlitzek. *Java2 Nitty Gritty*. Addison Wesley Verlag, 2000.

[6] Hubert Partl. *JAVA eine Einführung.* Postscriptversion ftp://ftp.boku.ac.at/www/javakurs.ps, html-Version http://www.boku.ac.at/javaeinf/ .

[7] R. Rössler, G. Hartmann. *Grundlagen der Objektorientierten Programmierung.* URL: //parallel.fh-bielefeld.de/pv/vorlesung /sp/begleitmaterial/Java.html .

[8] Christian Ullenboom. *Java ist auch eine Insel.* Galileo Verlag, zweite Auflage, Sep. 2002.

[9] Guido Krüger. *Goto Java 2.* Addison-Wesley, ISBN 38273-1710-X.

A Online Dokumentation zu Java

Auf der Java-Seite von SUN befindet sich eine Riesenmenge an Online-Dokumentaion im html-Format. Diese ist in zwei Pakete aufgeteilt. Leider sind die Pakete recht gross (entpackt 107MByte/38MByte), so dass man schon eine schnelle Internetanbindung braucht, um vernünftige Übertragungszeiten zu erhalten.

SUN erlaubt es meines Wissens nach nicht, dass die Dateien auf CDROMs veröffentlicht werden, aber gelegentlich verirren sie sich doch auf eine CD...

A.1 Dokumentation zur Java 2 Standard Edition

Der wichtigste Teil dieses Pakets ist die Beschreibung der Java API (Application Programming Interface). Sie enthält einen alfabetischen Index aller Java Standard Klassen, Methoden und Konstanten und eine ausführliche Beschreibung *aller* Klassen.

Gerade diese Klassendokumentation ist beim Entwickeln von Java-Programmen eine grosse Hilfe. Beim Schreiben eines Programms kann man die API-Dokumentation in einem eigenen Fenster geöffnet halten und hat dann ein bequemes Nachschlagewerk aller Java-Standard Klassen.

A.2 Java Tutorial

Dieses Paket enthält, ebenfalls im html-Format, eine Menge an Einführungen, die fast alles abdecken, was mit Java zu tun hat. Sehr interessant ist zum Beispiel die Lektion über Objekte und Klassen: Eigentlich hätte ich mir diesen Text hier sparen können...

B Einige Regeln zu Java-Quelldateien

Beim Übersetzen von UML-Diagrammen in Java-Quellcode sind folgende Regeln zu beachten:

- jede Klasse wird in eine *eigene* Datei geschrieben
- der Dateiname ist identisch mit dem Klassennamen und hat die Endung **. java**
- es muss eine sog. Startklasse geben, die eine Methode mit Namen **main** enthält

Die **main** - Methode erhält vom Betriebssystem des jeweiligen Rechners die erste Botschaft und versendet anschliessend Botschaften an die Objekte der Klassen, die zur jeweiligen Java-Anwendung gehören.

Der Ablauf von hin- und hergesendeten Botschaften zwischen den Objekten wird also über die **main** - Methode vom Betriebsystem *angestossen*.

C Aufgaben

Hier einige Übungsaufgaben:

1. (a) Erzeugen Sie mit einem Editor die Quelldatei `Hello.java` aus Abbildung 1. Compilieren Sie die Datei mit *javac*. Welche Datei erzeugt *javac*? Starten Sie die virtuelle Maschine mit der Hello-Klassendatei.

   (b) Ändern Sie den Klassennamen von `Hello.java` in einen beliebigen, anderen Wert. Beliebig sei aber dadurch eingeschränkt, daß Sie keine Umlaute und Sonderzeichen verwenden sollten. Welche Klassendatei erzeugt der Compiler jetzt?

   (c) Erweitern Sie den Begrüssungstext um eine weitere Zeile. Welchen Unterschied in der Ausgabe bewirken `println` bzw. `print`?

2. Die Ausgabe einer Textzeile erfolgt mit der Anweisung

   ```
   System.out.println("ausgabetext").
   ```

   Erklären Sie mit Hilfe der Online API-Dokumentation von SUN (auf CD), was die drei Begriffe

   - `System`
   - `out`

- `println`

bedeuten. Achten Sie dabei auf die Groß- und Kleinschreibung! Die Bedeutung der Punkte ist in Kapitel 6.3 erklärt.

3. (a) Testen Sie das Programm HelloOO aus Abbildung 3.
   (b) Erweitern Sie das Programm so, daß verschiedene Grußtext-Objekte in verschiedenen Sprachen erzeugt werden.
   (c) Falls Sie mit dem *xemacs* unter GNU/Linuxarbeiten, können Sie ein `makefile` zum HelloOO-Programm schreiben. Das `makefile` soll zuerst die Datei `HelloOO.java` compilieren und anschließend die Klassendatei ausführen.

4. Arbeiten Sie das Konstruktor-Beispiel aus Kap. D durch.

5. Erweitern Sie die Klasse `Auto` aus Kap. D um einen zweiten, überladenen Konstruktor, bei dem die Parameter in einer anderen Reihenfolge übergeben werden können.

6. Erweitern Sie `Auto` um die statische Eigenschaft `anzahlAutos` und die statische Methode `getAnzahlAutos`. An welcher Stelle der Klasse muss `anzahlAutos` hochgezählt werden, damit es die Anzahl der erzeugten Autoobjekte wiedergibt? Passen sie auch die Methode `beschreibung` an und geben sie auf diese Weise die Anzahl der Autoobjekte aus.

7. Als Java-Neuling wundert man sich immer wieder über die unmöglich lange Zeile

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

   Erklären Sie die Bedeutung der Schlüsselwörter

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

   Hinweis: Die eckigen Klammern hinter `String` deklarieren eine Liste, ein *Array*. In SUNs Java-Tutorial finden Sie unter `/java/data/arraybasics.html` die zugehörige Beschreibung.

8. In Abb. 32 und 33 ist ein Beispiel für Vererbung in Java dargestellt. Schreiben Sie dazu die noch fehlende Klasse `KFZTest`. Erzeugen Sie in dieser Testklasse PKW- und LKW-Objekte und testen sie die ererbten und die Instanzmethoden.

9. Auf der CD finden Sie im Ordner `/dienert/development/java/fourier2` ein Java-Programm, das aus mehreren Klassen besteht. Zeichnen Sie für jede dieser Klassen ein ausführliches Klassendiagramm.

D Was ist ein Konstruktor

Folgende Übung zeigt die Verwandschaft zwischen einer normalen Methode und einem Konstruktor auf. Sie besteht aus vier Beispielen, in denen man schrittweise einen Konstruktor konstruieren soll.

Stichworte: Klasse, Objekt, Eigenschaft, Methode, Membervariablen, Kapselung, private, public, set-Methode, Konstruktor

D.1 Direkter Zugriff auf die member-Variablen

Im ersten Beispiel sollen Sie eine Klasse *Auto* schreiben, die vier Eigenschaften und eine Methode enthält. Bild 25 zeigt das entsprechende Klassendiagramm.

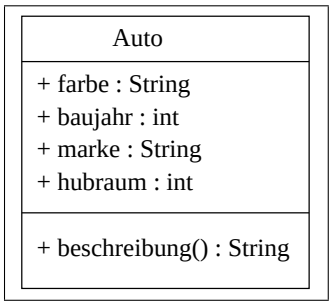

Abbildung 25: UML-Klassendiagramm

Um diese Klasse zu testen, müssen Sie noch eine weitere Klasse, die Klasse *AutoTest* schreiben.

AutoTest soll nur aus der Methode `main` bestehen, weshalb ein UML-Diagramm hier wenig aussagt. Als Hilfe gibt es stattdessen ein Code-Gerüst:

```
class AutoTest {
   public static void main(
               String[] comLinPar){
                    ...
   }
}
```

Abbildung 26: Code-Gerüst der *AutoTest*-Klasse

Da in der Klasse *Auto* zwar Eigenschaften definiert werden, diese Eigenschaften zunächst aber noch keine Werte haben, müssen Sie den Eigenschaften in der Klasse *AutoTest* konkrete Werte zuweisen.

Wie kann man nun aber auf die Eigenschaften zugreifen? Nehmen wir als Beispiel die `farbe` eines Autos. Wenn man sich die Klasse als *Bauplan für Objekte* vorstellt, darf man der `farbe` in der Klasse noch keine Werte zuweisen. Schliesslich werden

Autos in der Realität auch erst lackiert, nachdem die Karosserie zusammengeschweisst wurde.

D.h. Sie müssen *erst* ein Objekt erzeugen und können es dann färben, indem sie der Eigenschaft `farbe` einen Wert, z.B. `"rot"` zuweisen. Und auf die Eigenschaft eines bestimmten Objekts - die sog. *member-Variablen* - greift man mit der Punkt-Notation zu. Angenommen man hat das Auto-Objekt `fiat` gebildet, dann kann man den Fiat so färben:

```
fiat.farbe = "rot";
```

Das funktioniert aber nur, wenn diese Eigenschaften von aussen überhaupt sichtbar sind!

Aus diesem Grund werden alle Eigenschaften der Klasse *Auto* im UML-Diagramm mit einem vorangestellten **Pluszeichen** geschrieben. Dieses Pluszeichen wird im Java-Quelltext durch das Schlüsselwort `public` ersetzt. Eine mit `public` deklarierte Eigenschaft ist auch von ausserhalb ihrer Klasse sichtbar. Und genau das wollten wir ja erreichen.

Das gleiche gilt auch für die Methode `beschreibung()`, die wir auch von einer anderen Klasse aus aufrufen.

Es soll allerdings nicht verschwiegen werden, dass die direkte Manipulation von Eigenschaften ein *sehr schlechter* Java-Programmierstil ist! Im nächsten Beispiel werden Sie erfahren, wie man es besser macht.

Aufgaben zum ersten Beispiel:

1. Einzige Eigenschaft dieser Klasse soll die Methode `beschreibung` sein. Sie soll einen String zurückliefern, der das Autoobjekt beschreibt. Beispiel:

   ```
   Dieses Auto ist rot, hat 1500 ccm Hubraum und wurde 1964 von VW gebaut.
   ```

   Wie sieht diese Methode aus?

   Hilfen:
   Benötigt die Methode `beschreibung` irgendwelche Parameter?
   Welchen Datentyp liefert `beschreibung` zurück?
   Der Beschreibungs-String wird aus mehreren Teilstrings zusammengesetzt. Mit welchem Operator-Zeichen werden in Java Strings verkettet (Nachschlagen unter `String` in der API-Doc)?

2. Vervollständigen Sie die Klasse *AutoTest*:

   - Erzeugen Sie ein *Auto*-Objekt!
   - Weisen Sie den Eigenschaften dieses Objekts Werte zu! Um auf die Eigenschaften des Objekts zugreifen zu können, müssen Sie die Punkt-Notation verwenden.
   - Geben Sie den `String`, den die Methode `beschreibung()` zurückliefert mit der Methode `println` auf dem Bildschirm aus!
   - Starten Sie die Klasse *AutoTest*!

D.2 Klasse Auto mit Set-Methoden

Im zweiten Beispiel haben alle Eigenschaften im UML-Diagramm nun ein vorangestelltes **Minus**-Zeichen. Das Minus steht für `private`. Eigenschaften, die als `private` deklariert sind, sind von ausserhalb der Klasse unsichtbar und man kann deshalb auch nicht auf sie zugreifen.

Um sie dennoch verändern zu können, gibt es zu jeder Eigenschaft eine **set**-Methode. Die set-Methoden sind alle `public`, können also von ausserhalb der Klasse aufgerufen werden. Jede dieser set-Methoden enthält einen Parameter, dem dann beim Aufruf der Methode die entsprechende Eigenschaft zugewiesen wird. Als Name für diesen Parameter kann man z.B. den entsprechenden Eigenschaftsnamen verwenden, dem man die Endung `Par` anhängt: `farbePar`, `baujahrPar`, ...

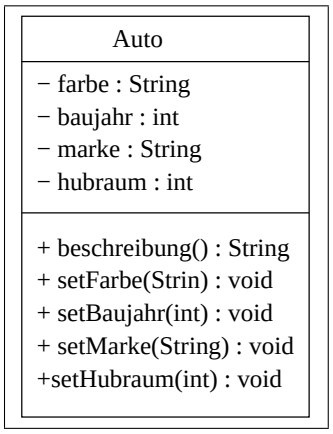

Abbildung 27: UML-Klassendiagramm

Aufgaben zum zweiten Beispiel:

1. Schreiben Sie die vier benötigten set-Methoden der Klasse *Auto*!
2. Passen Sie die Klasse *AutoTest* an! Statt die member-Variablen direkt zu verändern, müssen Sie jetzt die set-Methoden aufrufen.
3. Testen Sie Ihre Klassen!

D.3 Klasse Auto mit einer kombinierten Set-Methode

Abbildung 28 zeigt das UML-Diagramm der Auto-Klasse, bei dem alle vier Eigenschaften mit *einer* set-Methode gesetzt werden. Diese set-Methode `setAlleEigenschaften(...)` ist noch nicht vollständig dargestellt: es fehlen noch die Parameter, die Sie selbst festlegen sollen.

Aufgaben zum dritten Beispiel:

1. Wieviele Parameter muss man `setAlleEigenschaften` übergeben und welche Datentypen haben diese jeweils?

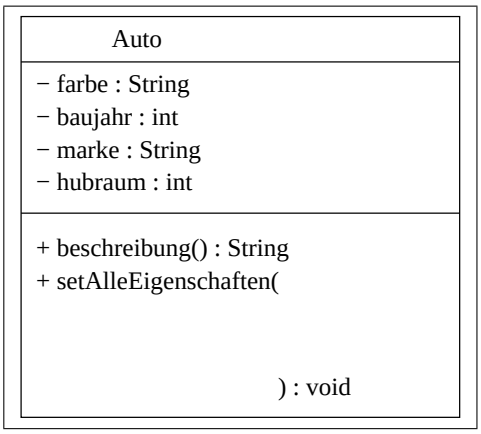

Abbildung 28: UML-Klassendiagramm mit einer set-Methode

2. Tragen Sie die Parameter in das UML-Diagramm ein!
3. Ändern Sie die Klasse *Auto*: Ersetzen Sie die vier einzelnen set-Methoden durch `setAlleEigenschaften`!
4. Nach dem Anpassen von *AutoTest*, sollen wieder alle Klassen getestet werden.

D.4 Klasse *Auto* mit Konstruktor

Der krönende Abschluss dieser Übung ist schliesslich die *Auto*-Klasse mit Konstruktor: Ein Konstruktor ist nichts Anderes als eine Methode, die den gleichen Namen wie die Klasse hat und die *nichts* zurückliefert, auch nicht `void`!

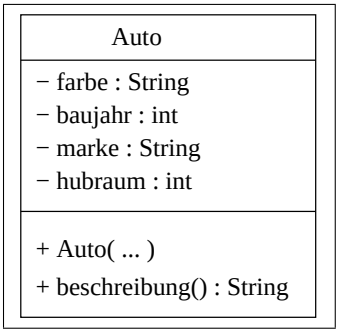

Abbildung 29: UML-Klassendiagramm mit Konstruktor

Aufgaben zum vierten Beispiel:

1. Vervollständigen Sie das Klassendiagramm!
2. Schreiben Sie den Konstruktor in Java, führen evtl. nötige Anpassungen an *AutoTest* durch und testen alles!

3. Wann wird der Konstruktor aufgerufen? Wann kann ich eine gewöhnliche Methode aufrufen?

E Kostenlose Entwicklungsumgebungen.

Am einfachsten lässt sich Software innerhalb einer integrierten Entwicklungsumgebung (Integrated Development Environment, IDE) schreiben.

Zur Zeit streiten sich die Firmen SUN und IBM darum, wer die bessere Java-IDE entwickelt hat: **Eclipse** von IBM und **NetBeans** von SUN. Beide Produkte sind Open Source und damit kostenlos. Der Vorteil von Eclipse ist die hohe Geschwindigkeit, die aber mit einer plattformabhängigen Grafikbibliothek erkauft wird. NetBeans ist dagegen völlig plattformunabhängig.

Meine bevorzugte Entwicklungsumgebung ist allerdings der Editor *xemacs* unter GNU/Linux oder unter X11 auf dem Mac mit dem *Java Development Environment*, JDE. Das JDE erweitert den Editor um einige Menuepunkte mit Untermenues zum Compilieren, Starten der VM, Debuggen, einem Klassen-Browser, etc.

Wichtig bei der Wahl der Entwicklungsumgebung ist bei grösseren Projekten nur, das sie *Ant* (siehe E.2) unterstützt. Bei den drei zuvor genannten IDEs ist das natürlich der Fall. Durch die Verwendung von Ant lässt sich auch ein grosses Projekt problemlos auf beliebigen Plattformen compilieren.

E.1 Java Oriented Editing JOE

Unter Windows gibt es einen sehr einfach zu bedienenden Editor mit direkter JDK-Anbindung: JOE, das steht für Java Oriented Editing. Das Paket gibt es bei `www.javaeditor.de`. Leider wird JOE inzwischen nicht mehr weiterentwickelt.

E.2 Another Neat Tool

Das *Another Neat Tool*, **Ant** ist keine Entwicklungsumgebung sondern ein Java-Programm, mit dem man das Compilieren beliebig komplexer Java-Projekte automatisieren kann.

Dazu schreibt man eine `build.xml` - Datei, die aus mehreren sog. *Targets* besteht. Ein Target kann man sich wie eine Funktion vorstellen, die über einen Namen gestartet werden kann. Die Teilaufgaben, aus denen wiederum das Target besteht, nennt man *Tasks*.

Ant wird bereits mit einer grossen Zahl von Standard-Tasks ausgeliefert. Diese erledigen z.B. das Anlegen von Unterverzeichnissen für den erzeugten Bytecode, das Compilieren oder das Packen der Klassen in Jar(Java Archiv)-Dateien.

Was Ant unschlagbar flexibel macht ist die Möglichkeit, eigene Tasks in Form von Java-Klassen zu schreiben und dem Ant-System hinzuzufügen. Im Anhang E.2 ist als Beispiel eine einfache `build.xml`-Datei enthalten.

F Java-Beispiele

Abbildung 30: Klassendefinition und Objekterzeugung in Java

```
class Auto {

   // eigenschaften
   String farbe;
   int hubraum;
   int baujahr;
       ...

   // methoden
   Auto(String farbePar, int hubraumPar, int baujahrPar) {
      farbe = farbePar;
      hubraum = hubraumPar;
      baujahr = baujahrPar;
   }

   void anmelden(){
        ...
   }
   void abmelden(){
        ...
   }
   void betanken(int füllung){
        ...
   }
   ...
}   //ende der klassendefinition

****************ende der datei Auto.java******************


class Autotest {

   //main-methode
   public static void main( String[] args ){

      Auto meinAuto;   //erzeugen der objektreferenz
      meinAuto = new Auto("rot", 1500, 1964);   //anlegen und initialisieren des objekts
      ...
      meinAuto.anmelden();   //aufruf der methode anmelden()
      meinAuto.betanken(50);    //aufruf der methode betanken mit parameter 50
      ...
   }
}   //ende der klassendefinition

***************ende der datei AutoTest.java***************
```

Abbildung 31: Klassenattribut in Java

```
   class Auto {

   // eigenschaften
   String farbe;
   String kennzeichen;
       ...

   // statische eigenschaft = klassenattribut

    static int anzahlGemeldet = 0;

   // methoden
   Auto(String farbePar, int hubraumPar, int baujahrPar) {
      farbe = farbePar;
      hubraum = hubraumPar;
      baujahr = baujahrPar;
   }

   void anmelden(String kennzeichenPar){
      kennzeichen = kennzeichenPar;                    //kennzeichen zuteilen
      anzahlGemeldet = anzahlGemeldet + 1;             //anmeldungen zaehlen
   }

   // statische Methode = klassenmethode

    public static int getAnzahlGemeldet() {
      return anzahlGemeldet;
   }
      ...

}   //ende der klassendefinition

****************ende der datei Auto.java******************


class Autotest {

   //main-methode
   public static void main( String[] args ){

      Auto meinAuto = new Auto("rot", 1500, 1964);    //anzahlGemeldet = 0
      Auto deinAuto = new Auto("weiss", 1300, 1971);  //anzahlGemeldet = 0

      ...
      meinAuto.anmelden("EM-MD 1964");
      System.out.println("Sie haben das"
                         + meinAuto.getAnzahlGemeldet() //aufruf der klassenmethode
                         + "-ste angemeldete Auto");
      //anzahlGemeldet = 1

      deinAuto.anmelden("EM-A 7571");
      System.out.println("Sie haben das"
                         + deinAuto.getAnzahlGemeldet() //aufruf der klassenmethode
                         + "-ste angemeldete Auto");
      //anzahlGemeldet = 2
      ...
   }
}   //ende der klassendefinition

***************ende der datei AutoTest.java***************
```

Abbildung 32: Vererbung in Java: Superklasse

```
public abstract class Kraftfahrzeug {

   // eigenschaften
   String hersteller;
   int baujahr;
   String motor;
   String kennzeichen;
   // methoden
      //an dieser stelle steht zunaechst kein konstruktor, es koennen ja keine
      //kraftfahrzeugobjekte gebildet werden.

   public void setHersteller(String herstellerPar){
      hersteller = herstellerPar;
   }

   public void setBaujahr(int baujahrPar){
      baujahr = baujahrPar;
   }

   public void setMotor(String motorPar){
      motor = motorPar;
   }

   void anmelden(String kennzeichenPar){
          kennzeichen = kennzeichenPar;
          System.out.println("zulassung:" + kennzeichen);
   }

   void abmelden(String kennzeichenPar){
          kennzeichen = "";
          System.out.println("nu isser wech");
   }

   void starten(){
      System.out.println("leerlaufdrehzahl ist 800 u/min");
   }

}   //ende der klassendefinition


***************ende der datei Kraftfahrzeug.java******************
```

Abbildung 33: Vererbung in Java: abgeleitete Klassen

```
public class LKW extends Kraftfahrzeug {

   //eigenschaften
   int bremsDruck = 10;

   //methoden
   LKW(String herstellerPar, int baujahrPar, String motorPar){
      setHersteller(herstellerPar);   //aufruf der ererbten methoden
      setBaujahr(baujahrPar);         //           - ´´-
      setMotor(motorPar);             //           - ´´-
   }

   void wartenKompr(String schlossername){
      //kompressor fuer bremspneumatik warten
      System.out.println(schlossername + " hat den kompressor gewartet");
      System.out.println("der bremsdruck beträgt " + bremsDruck + "bar.");
   }

}   //ende der klassendefinition

****************ende der datei LKW.java***************


public class PKW extends Kraftfahrzeug {

   //eigenschaften
   int hydrFuellstand = 50;

   //methoden
   PKW(String herstellerPar, int baujahrPar, String motorPar){
      setHersteller(herstellerPar);   //aufruf der ererbten methoden
      setBaujahr(baujahrPar);         //           - ´´-
      setMotor(motorPar);             //           - ´´-
   }

   void auffuellenHydr(String schlossername){
      //bremsfluessigkeit nachfuellen
      System.out.println(schlossername + " hat bremsflüssigkeit aufgefüllt");
      System.out.println("der füllstand beträgt " + hydrFuellstand + "mm.");
   }
}   //ende der klassendefinition

****************ende der datei LKW.java***************
```

G Eine einfache build.xml-Datei

E.2

```
<?xml version="1.0"?>
<project name="adressbuch" default="dist" basedir=".">
    <description>
        eine einfache build.xml datei
    </description>
  <!-- properties entsprechen variablen: -->
  <property name="userName" value="micha"/>

  <target name="all" depends="clean, dist"
        description="alles loeschen und neu compilieren" >
  </target>

  <target name="init"
        description="verzeichnisse ./build und ./dist anlegen" >
     <!-- build-verzeichnis anlegen -->
     <!-- wird nur ausgefuehrt, wenn build noch nicht vorhanden -->
     <mkdir dir="build"/>
     <!-- dist-verzeichnis anlegen -->
     <mkdir dir="dist"/>
  </target>

  <target name="compile" depends="init"
        description="quellen uebersetzen" >
    <!-- Compile java-code von src nach build compilieren -->
    <javac srcdir="src" destdir="build"/>
  </target>

  <target name="dist" depends="compile"
        description="die distribution erzeugen" >

    <!-- Die manifestdatei muss folgende eintraege enthalten:
         Main-Class: schule.ist.schoen.Paket  -->

    <jar destfile="dist/auto.jar" basedir="build">
       <include name="**/*.class"/>
       <manifest>
         <attribute name="Built-By" value="${userName}"/>
         <attribute name="Main-Class" value="schule.ist.schoen.Paket"/>
       </manifest>
    </jar>

    <!-- das archiv im dist-verzeichnis wird gestartet: -->

    <java jar="dist/auto.jar"
       fork="true"
       failonerror="true"
       maxmemory="128m"
    >
    </java>

  </target>

  <target name="clean"
        description="aufraeumen" >
    <!-- build und dist verzeichnisse loeschen -->
    <delete dir="build"/>
    <delete dir="dist"/>
  </target>
</project>
```

Abbildung 34: **??** Eine einfache build.xml-Datei