Inhaltsverzeichnis

1. Erläuterung der Begriffe Objekt, Klasse, Attribut und Methode

Durch die Verwendung von Objekten versucht man, natürliche Sachverhalte bestmöglich in einem Programm nachzubilden. Unter einem Objekt versteht man ein Exemplar einer bestimmten Klasse, man spricht auch von Instanzen einer Klasse. Klassen beschreiben den Aufbau von Objekten, sie sind sozusagen deren Bauplan. Softwaretechnisch gesehen ist z. B. jeder Mitarbeiter einer Firma ein Objekt der Klasse Mitarbeiter. Objekte enthalten i. d. R. Attribute und Methoden. Mit Attributen sind die Eigenschaften gemeint, welche den Objekten zugeordnet werden können. Alle Objekte einer Klasse besitzen dieselben Attribute und Methoden, können sich aber in ihren Attributwerten unterscheiden. Schließlich habe alle Mitarbeiter einer Firma zwar einen Vornamen (Attribut), aber nicht alle heißen Horst (Attributwert). Methoden beschreiben das Verhalten bzw. die Fähigkeiten eines Objekts. Dabei greifen diese Methoden häufig lesend oder schreibend auf Attribute zu. Wenn z. B. der Mitarbeiter Horst Voll seine Freundin Gisela Leer heiratet und dabei ihren Nachnamen annimmt, so sollte das entsprechende Objekt eine Methode zum Ändern des Attributs "Nachname" besitzen.

2. Beispielhafte Erläuterung der objektorientierten Programmierung (OOP)

Bevor sich die OOP durchgesetzt hatte, wurde vorzugsweise das Prinzip der prozeduralen Programmierung (strukturierte Programmierung) genutzt. Dieses Prinzip orientiert sich an der Arbeitsweise von Rechnern. Der Code besteht dabei aus einer Folge von Anweisungen. Inhaltlich zusammengehörende Anweisungen werden zu Prozeduren bzw. Funktionen zusammengefasst. Im Gegensatz dazu orientiert sich die OOP an der realen Welt. Man versucht hier, Objekte der realen Welt möglichst 1:1 im Programmcode nachzubilden. Beispiel:

In einem Getränkemarkt findet man jede Menge Objekte, z. B. das Mineralwasser "Ruhrsprudel" der Firma "Ruhr-Brunnen-GmbH" in der 1,5-Liter-Flasche für 0,12€. All diese Objekte besitzen die gleichen Attribute, nämlich einen Namen, einen Hersteller, eine Füllmenge und einen Preis. Deshalb kann man diese Objekte zu einer Klasse zusammenfassen, welche man sinnvollerweise "Getränk" nennen sollte. Die folgende Abbildung zeigt eine verkürzte Version dieser Klasse, welche nur die beiden Attribute `name` und `preis` enthält. Um den Inhalt eines Attributs (Attributwert) verändern zu können, verwendet man set-Methoden. Für das Auslesen eines Attributs (Rückgabe des Attributwerts) dienen get-Methoden. Hier sind beide Methoden exemplarisch nur für das Attribut `name` implementiert.

```
2  public class Getraenk
3  {
4      //Die Attribute name und preis werden definiert:
5      private String name;
6      private double preis;
7
8      //set-Methode zum Setzen des Attributs name wird definiert:
9      public void setName(String neuerName)
10     {
11         name = neuerName;
12     }
13
14     //get-Methode zum Auslesen des Attributs name wird definiert:
15     public String getName()
16     {
17         return name;
18     }
19 }
```

Zeile 2: Hier beginnt die Definition der öffentlichen (`public`) Klasse `Getraenk`.

Zeile 5: Jedes Getränk hat einen Namen, welcher als String gespeichert wird. Deshalb wird hier das Attribut `name` mit dem Datentyp `String` definiert. Das Schlüsselwort `private` steht für "nicht öffentlich". Damit wird gewährleistet, dass nur innerhalb der Klasse `Getraenk` auf dieses Attribut direkt zugegriffen werden kann. (Anmerkung: Attribute werden manchmal auch Membervariablen genannt.)

Zeile 6: Jedes Getränk hat einen Preis. Hier wird ein entsprechendes Attribut vom Datentyp `double` definiert.

Zeile 9: Hier wird eine Methode definiert, welche zum Setzen (verändern, überschreiben) des Attributs `name` dient. Diese Art Methoden werden set-Methoden genannt, ihre Namen sollten immer mit dem Wort "set" beginnen und den Namen des zu setzenden Attributs enthalten. Der Methode wird beim Aufruf ein Wert vom Typ `String` übergeben. Innerhalb der Methode ist dieser Wert in der lokalen Variablen

`neuerName` gespeichert. Innerhalb des geschweiften Klammerpaares (Zeilen 10 und 12) wird auf Zeile 11 festgelegt, dass das Attribut `name` mit dem in `neuerName` gespeicherten Wert überschrieben werden soll. Durch das Schlüsselwort `public` wird festgelegt, dass diese Methode öffentlich ist. Das bedeutet, dass von jeder anderen Klasse heraus diese Methode aufgerufen werden kann. Beim Aufruf gibt diese Methode keinen Wert zurück, was durch das Schlüsselwort `void` festgelegt wird.
Die Definition der Methode `setName` endet mit der geschweiften Klammer auf Zeile 12.

Zeile 15: Hier wird eine öffentliche (`public`) Methode definiert, welche zum Auslesen des Attributs `name` dient. Diese Art Methoden werden get-Methoden genannt, ihre Namen sollten immer mit dem Wort "get" beginnen und den Namen des auszulesenden Attributs enthalten. Dieser Methode wird beim Aufruf nichts übergeben, deshalb ist innerhalb des runden Klammerpaares auch keinerlei Inhalt. Durch die Angabe `String` vor dem Methodennamen `getName` wird festgelegt, dass diese Methode beim Aufruf einen Wert vom Typ `String` zurückgeben soll. Innerhalb des geschweiften Klammerpaares (Zeilen 16 und 18) wird auf Zeile 17 durch das Schlüsselwort `return` festgelegt, dass der Inhalt des Attributs `name` zurück gegeben werden soll. Die Definition der Methode `getName` endet mit der geschweiften Klammer auf Zeile 18.

Zeile 19: Hier endet die Definition der Klasse Getraenk.

Hinweis:
Die Methode `setName` kann auch so definiert werden:
Die in der Übergabeklammer definierte (lokale) Variable `name` besitzt den gleichen Bezeichner wie das Attribut, welches innerhalb der Methode gesetzt werden soll.

```
public void setName(String name)
{
    this.name = name;
}
```

Damit der Compiler zwischen Attribut (globale Variable für die gesamte Klasse) und lokaler Variable unterscheiden kann, muss in diesem Fall der Operator this verwendet werden. Dieser Operator ist ein Verweis auf das aktuelle Objekt, von welchem er aufgerufen wird.

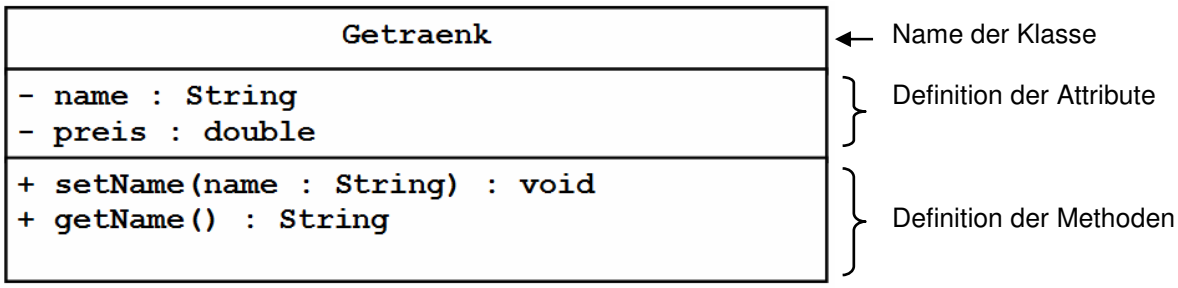
Hinweis:
In einer Klasse können auch Attribute definiert werden, welche selbst wieder Objekte einer anderen Klasse sind. So kann beispielsweise eine Klasse "Konto" ein Attribut besitzen, welches ein Objekt der Klasse "Kunde" ist. Denn jedes Konto einer Bank hat schließlich einen Besitzer, welcher Kunde dieser Bank ist. Und zu einem Kunden gehören zum Beispiel ein Name, ein Geburtsdatum und eine Adresse.

Die Klasse Getraenk wird als Datei Getraenk.java (Dateiname = Klassenname!) gespeichert. Jetzt wird das "eigentliche" Programm geschrieben, in dem Objekte der Klasse Getraenk erzeugt und verwendet werden. Dieses "eigentliche" Programm ist eine Main-Methode, welche von "oben" nach "unten" ausgeführt wird. Also so, wie es bereits von der strukturierten Programmierung her bekannt ist. Auch wenn es nicht zwingend erforderlich ist, sollte die Main-Methode immer in einer separaten Klasse geschrieben und als Datei (Dateiname = Klassenname!) gespeichert werden. Diese Klasse kann als "Startklasse" betrachtet werden. Die folgende Abbildung zeigt die Startklasse `Getraenkeverwaltung`. Hier wird ein Objekt der Klasse `Getraenk` erzeugt. Anschließend wird dem Attribut `name` ein Wert zugewiesen. Am Ende des Programms wird der in `name` gespeicherte Wert zur Kontrolle ausgegeben.

```
1  public class Getraenkeverwaltung
2  {
3      public static void main (String args[])
4      {
5          Getraenk g1 = new Getraenk (); //Das Objekt g1 der Klasse Getraenk wird erzeugt.
6
7          /* Dem Attribut name des Objekts g1 wird jetzt ein Wert zugewiesen.
8             Dies erfolgt durch den Aufruf der zugehörigen set-Methode:                */
9
10         g1.setName("Ruhrsprudel");
11
12         /* Der Inhalt des Attributs name wird durch den Aufruf der zugehörigen
13            get-Methode ausgelesen und in der Variable nameGetraenk zwischengespeichert.
14            Anschließend wird der Inhalt dieser Variable auf der Konsole ausgegeben:   */
15
16         String nameGetraenk = g1.getName();
17         System.out.println(nameGetraenk);
18     }
19 }
```

3. Unified Modeling Language (vereinheitlichte Modellierungssprache) - UML

Während der Entwurfsphase einer objektorientierten Software spielt die UML eine sehr wichtige Rolle. Sie dient zur programmiersprachenunabhängigen Darstellung von statischen Strukturen und dynamischen Abläufen. Dabei sind unbedingt die Standards der UML-2 einzuhalten. In der UML sind verschiedene Diagrammtypen definiert, von denen einige im Rahmen dieses Skripts näher erläutern werden. Jetzt soll zunächst nur (ansatzweise) das Klassendiagramm betrachtet werden. In einem Klassendiagramm wird u. a. der Aufbau einer Klasse beschrieben. Später werden dann auch die Beziehungen zwischen mehreren Klassen einer Software dargestellt. Die in diesem Skript bereits als Javacode vorgestellte Klasse `Getraenk` wird nach UML-2 als Klassendiagramm so dargestellt:

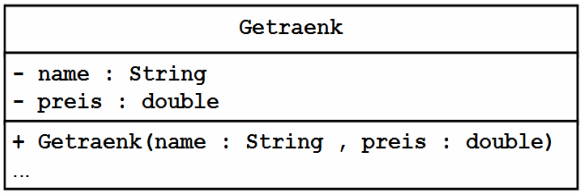


Hinweis: Das Zeichen "-" steht für "private". Das Zeichen "+" steht für "public".

4. Konstruktor

Soll bereits bei der Erzeugung eines Objekts allen oder einigen Attributen ein Wert zugewiesen werden, so muss in der Klasse ein Konstruktor definiert sein. Dieser Konstruktor ähnelt vom Aufbau her einer Methode. Sein Name muss identisch mit dem Klassennamen sein. Er gibt niemals einen Wert zurück, deshalb entfällt auch das Schlüsselwort `void`. Wurde ein Konstruktor definiert, so müssen alle entsprechenden Werte unter Beachtung der Reihenfolge und des Datentyps bei der Objekterzeugung als Parameter übergeben werden.

Die bereits bekannte Klasse `Getraenk` wird jetzt mit einem Konstruktor versehen, welcher die Attribute `name` und `preis` bei der Erzeugung setzt (initialisiert).



Konstruktor im Klassendiagramm

Konstruktor im Javacode

```
2  public class Getraenk
3  {
4      //Die Attribute name und preis werden definiert:
5      private String name;
6      private double preis;
7
8      //Ein Konstruktor wird definiert:
9      public Getraenk(String name, double preis)
10     {
11         this.name = name;
12         this.preis = preis;
13     }
14
15     // Get- und Set-Methoden werden definiert ...
```

Die folgende Abbildung zeigt beispielhaft, wie die Objekterzeugung in der Main-Methode realisiert werden muss:

```
2  public class Getraenkeverwaltung
3  {
4      public static void main (String args[])
5      {
6          /* Das Objekt g1 der Klasse Getraenk wird erzeugt.
7             Dabei werden dem Konstruktor alle erforderlichen Werte übergeben. */
8
9          Getraenk g1 = new Getraenk("Ruhrsprudel", 0.12);
10
11         // Die Inhalte der gesetzten Attribute werden ausgegeben:
12
13         System.out.println("Im Angebot: " + g1.getName() + " nur " + g1.getPreis());
14     }
15 }
```

5. Überladen von Methoden und Konstruktoren

Das von der strukturierten (prozeduralen) Programmierung her bekannte Prinzip des Überladens von Funktionen kann auch für Methoden und Konstruktoren verwendet werden. Es können also innerhalb einer Klasse mehrere Methoden bzw. Konstruktoren definiert werden, welche alle denselben Namen besitzen. Sie müssen sich aber durch die Anzahl der Parameter oder/und unterschiedliche Parameter-Datentypen unterscheiden. Definiert man in einer Klasse mehrere Konstruktoren, so findet zwangsläufig eine Überladung statt. Denn diese Konstruktoren müssen immer denselben Namen haben, nämlich den Namen der Klasse.

Das folgende Beispiel zeigt das Überladen des Konstruktors der Klasse `IrgendeineKlasse`.
Bei der Objekterzeugung in der Main-Methode (Startklasse `Startklasse`) werden dann alle Varianten ausprobiert. Weitere Erläuterungen sind den Kommentaren der beiden folgenden Klassen zu entnehmen:

```
public class IrgendeineKlasse
{
    int a, b;
    char c;

    // 1. Konstruktor wird definiert:
    public IrgendeineKlasse(int a)
    {
        this.a = a;
    }

    // 2. Konstruktor wird definiert:
    public IrgendeineKlasse(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    // 3. Konstruktor wird definiert:
    public IrgendeineKlasse(int a, char c)
    {
        this.a = a;
        this.b = b;
    }
```

```
2  public class Startklasse
3  {
4      public static void main (String args[])
5      {
6          IrgendeineKlasse objekt1 = new IrgendeineKlasse(7, 9);    // 2.Konstruktor wird aufgerufen
7          IrgendeineKlasse objekt2 = new IrgendeineKlasse(4);       // 1.Konstruktor wird aufgerufen
8          IrgendeineKlasse objekt3 = new IrgendeineKlasse(4, 'X'); // 3.Konstruktor wird aufgerufen
9      }
10 }
```

6. Auswahl geeigneter Bezeichner (Konventionen)

Bei der Erstellung von Javacode oder UML-Diagrammen existieren Konventionen, welche einzuhalten sind. Neben den bereits von der strukturierten Programmierung her bekannten Konventionen gelten in der OOP folgende Ergänzungen bzgl. der Bezeichner:

Großschreibung: Klassen, Konstruktoren
Kleinschreibung: Objekte, Attribute, Methoden

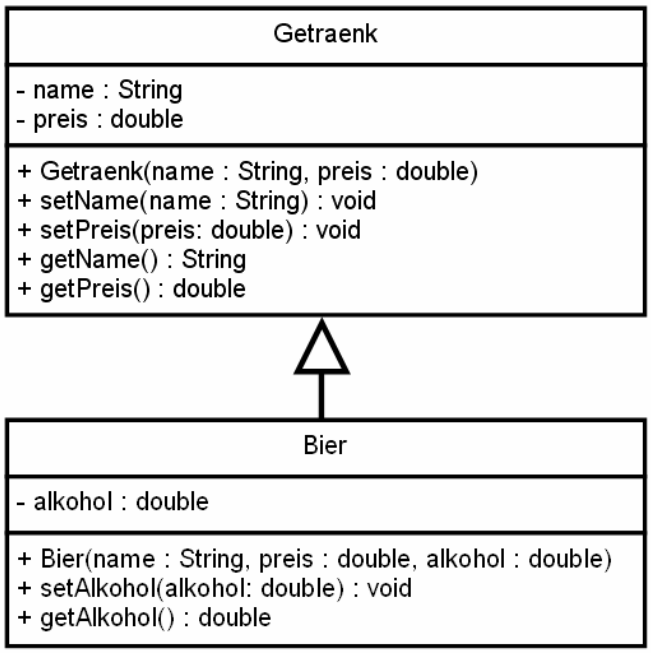
Die Bezeichner der oben aufgeführten Elemente sollten sinnvoll gewählt und möglichst selbsterklärend sein. Soll z. B. in einem Attribut der Vorname einer Person gespeichert werden, so wäre der Bezeichner `vorname` perfekt. Natürlich könnte man stattdessen auch `qwertz0815` verwenden, was allerdings äußerst sinnfrei wäre und den Konventionen widersprechen würde.

7. Vererbung

Eines der wichtigsten Ziele der OOP ist es, bereits erstellten Programmcode möglichst oft wiederzuverwenden. Dies wird u. a. durch das Prinzip der Vererbung erreicht, welches ein grundlegendes Konzept der OOP ist. Die Vererbung dient dazu, aufbauend auf existierenden Klassen neue zu schaffen. Beispiel:

Ein Getränkemarkt verwaltet seine Produkte mittels einer objektorientierten Software. Die dafür notwendige Klasse `Getraenk` ist Ihnen bereits bekannt. Um die Anzahl der Kunden kräftig zu erhöhen, werden verschiedene Biersorten in das Sortiment aufgenommen. Bier besitzt die gleichen Attribute wie jedes andere Getränk, allerdings spielt hier zusätzlich der Alkoholgehalt (z. B. 4,9 %) eine Rolle. Man könnte jetzt eine neue Klasse `Bier` erstellen und sie mit allen Attributen und Methoden füllen. Sinnvoller wäre es aber, das Prinzip der Vererbung zu nutzen. Dabei erbt die Klasse `Bier` alle Attribute und Methoden der Klasse `Getraenk`. Diese Attribute und Methoden stehen dann automatisch auch in der erbenden Klasse `Bier` zur Verfügung. Es müssen nur diejenigen Attribute und Methoden neu geschrieben werden, welche speziell zur Klasse `Bier` gehören. Der Vorgang des Erbens wird deshalb auch Spezialisierung und in umgekehrter Richtung Generalisierung genannt.

Die folgende Abbildung zeigt die Vererbung im Klassendiagramm:



← **Superklasse** `Getraenk`
(Oberklasse, Basisklasse, Vaterklasse, vererbende Klasse)

← **Vererbungspfeil** nach UML2-Standard: Die Klasse `Bier` erbt von der Klasse `Getraenk`. Man kann auch sagen: Die Klasse `Bier` erweitert die Klasse `Getraenk`.

← **Subklasse** `Bier`
(Unterklasse, abgeleitete Klasse, Kindklasse, erbende Klasse)

Die Klasse `Bier` besitzt jetzt die geerbten Attribute `name` und `preis` sowie das zusätzliche Attribut `alkohol`. Sie besitzt ebenfalls alle Methoden der Superklasse `Getraenk` sowie die zusätzlichen Methoden `setAlkohol` und `getAlkohol`. Weiterhin wurde ein Konstruktor vorgesehen, welche alle Attribute setzt.

Für Klassendiagramme und auch für den daraus zu erstellenden Javacode gilt: In der Subklasse werden geerbte Attribute und Methoden nicht aufgeführt.

Bei der Umsetzung des Klassendiagramms in den Javacode ist zu beachten, dass die Superklasse und die Subklasse(n) in einem gemeinsamen Ordner liegen. Natürlich gibt es auch andere Möglichkeiten, auf welche hier aber nicht näher eingegangen wird.

Es folgt jetzt der Javacode der Klassen `Getraenk` und `Bier`:

Hier ist die Klasse `Getraenk` abgebildet. Diese ist bereits bekannt und enthält keine neuen Elemente. Auf eine Erläuterung des Codes wird deshalb verzichtet.

```
 2  public class Getraenk
 3  {
 4      private String name;
 5      private double preis;
 6
 7      public Getraenk (String name, double preis)
 8      {
 9          this.name = name;
10          this.preis = preis;
11      }
12
13      public double getPreis()
14      {
15          return preis;
16      }
17
18      public void setPreis(double neuer_preis)
19      {
20          preis = neuer_preis;
21      }
22
23      public String getName()
24      {
25          return name;
26      }
27
28      public void setName(String neuer_name)
29      {
30          name = neuer_name;
31      }
32  }
```

Hier ist die Klasse `Bier` abgebildet, welche von der Klasse `Getraenk` erbt.

```
 2  public class Bier extends Getraenk
 3  {
 4      private double alkohol;
 5
 6      public Bier (String name, double preis, double alkohol)
 7      {
 8          super(name, preis);
 9          this.alkohol = alkohol;
10      }
11
12      public double getAlkohol()
13      {
14          return alkohol;
15      }
16
17      public void setAlkohol(double alkohol)
18      {
19          this.alkohol = alkohol;
20      }
21  }
```

Erläuterungen zur Klasse `Bier`:

Zeile 2: Das Schlüsselwort `extends` gibt dem Compiler bekannt, dass die Klasse `Bier` die Klasse `Getraenk` erweitert. Oder anders gesagt: Die Klasse `Bier` erbt von der Klasse `Getraenk`.

Zeile 4: Das zusätzliche (also nicht geerbte) Attribut `alkohol` wird definiert.

Zeilen 6 - 10: Es wird ein Konstruktor definiert, welcher alle Attribute (geerbte und nicht geerbte) initialisiert. Auf der Zeile 8 wird mit dem Schlüsselwort `super` festgelegt, dass diejenigen Werte, mit denen die geerbten Attribute `name` und `preis` gesetzt werden sollen, dem Konstruktor der Superklasse übergeben werden. Beim Schreiben dieser Zeile muss also die Definition des Konstruktors der Superklasse (Anzahl, Datentypen und Reihenfolge der Übergabeparameter) bekannt sein. Auf der Zeile 9 wird dann das zusätzliche Attribut `alkohol` gesetzt.

Zeilen 12 - 20: Hier werden diejenigen Methoden definiert, welche nicht geerbt wurden.

Diese Bilder zeigen typische Vererbungs-strukturen, welche in der OOP genutzt werden.

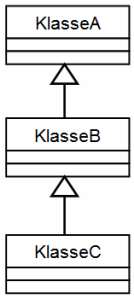


Bild 1

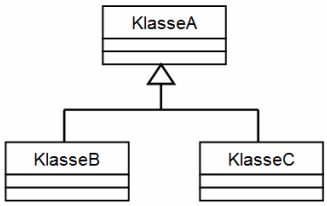


Bild 2

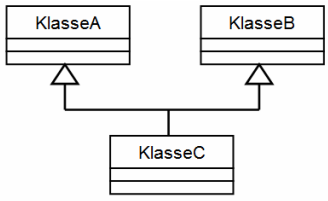


Bild 3

Bild 1: Hier erbt die Klasse C von der Klasse B, welche wiederum von der Klasse A erbt. Diese "Kette" kann beliebig fortgesetzt werden.

Bild 2: Die Klassen B und C erben direkt von der Klasse A. Selbstverständlich können auch mehr als zwei Klasse von einer Klasse direkt erben.

Bild 3: Hier erbt die Klasse C direkt von zwei (oder mehr) Klassen. Diese Struktur (und nur diese!) wird in der OOP als Mehrfachvererbung bezeichnet und ist z. B. in der Sprache C++ anwendbar.
Hinweis: In Java ist die Mehrfachvererbung (absichtlich) nicht möglich!

In Java können die in den Bildern 1 und 2 dargestellten Strukturen beliebig kombiniert werden. Die Klassen des Java-Development-Kits (JDK) sind Bestandteile einer riesigen Vererbungsstruktur, an deren Spitze die Klasse `Object` steht. Damit erben also alle JDK-Klassen entweder direkt oder indirekt von der Klasse `Object`. Wird in Java eine Klasse ohne explizite Angabe einer Vererbung (Schlüsselwort `extends`) erstellt, so erbt sie trotzdem automatisch und direkt von `Object`.

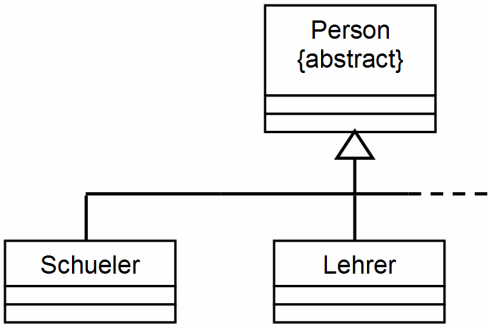
8. Abstrakte Klassen

In einer Schule werden mittels Software verschiedene Personengruppen verwaltet. Es gibt Schüler, Lehrer und Verwaltungsangestellte (Hausmeister, Sekretärinnen). Weil für diese Gruppen unterschiedliche Attribute gespeichert werden müssen, ist es sinnvoll, für jede dieser Gruppen eine Klasse zu erstellen. Allerdings haben diese Gruppen auch viele gemeinsame Eigenschaften (Name, Vorname, Geburtsdatum, Wohnort,...), damit müssten viele Attribute und zugehörige Methoden mehrfach geschrieben werden. Um dies zu vermeiden, fasst man alle gemeinsamen Attribute/Methoden in einer Superklasse zusammen, welche man z. B. `Person` nennen kann. Dieses Prinzip wird Generalisierung (Verallgemeinerung) genannt. Von dieser Klasse lässt man dann die anderen Klassen erben. Prinzipiell könnte man auch Objekte von der Klasse `Person` erzeugen, was aber überflüssig und sogar unsinnig wäre. Deshalb gehört es zu einem sauberen Programmierstil, wenn man in diesem Fall `Person` als abstrakte Klasse definiert. Der Compiler reagiert dann mit einer Fehlermeldung, falls irgendwo im Programmcode versucht wird, ein Objekt der Klasse `Person` zu erzeugen.

Von abstrakten Klassen können keine Objekte erzeugt werden.
Sie dienen nur als Superklassen für Vererbungen.

Die folgenden Abbildungen zeigen, wie die Klasse `Person` in Java bzw. im Klassendiagramm abstrakt "gemacht" wird. Anstelle des Zusatzes {abstrakt} im Klassendiagramm ist es auch möglich, den Namen von abstrakten Klassen *kursiv* zu schreiben. Weil dies aber nicht immer eindeutig erkennbar ist, besonders bei handschriftlich erstellten Diagrammen, sollte auf diese Möglichkeit verzichtet werden.

```
2  public abstract class Person
3  {
4
```



9. Objekte in Listen speichern

Gleichartige primitive Datentypen (z. B. Integer) kann man in einem Array speichern. Für gleichartige Objekte ist dies auch möglich. Weil die Realisierung von Schreib- und Lesezugriffen auf das Array sehr aufwendig ist, sollte man besser die JDK-Klasse `ArrayList` verwenden. Diese gehört zu den sogenannten Containerklassen, von denen im JDK eine Vielzahl vorhanden ist. Die Klasse nutzt ein Array zum Speichern der Objekte, auf das aber nur mittels Methoden zugegriffen werden kann. Eine Beschreibung der JDK-Klassen und ihrer Methoden und Konstruktoren findet man in der sogenannten API-Dokumentation (API: Application Programming Interface). Diese API-Dokumentation, kurz API-Doc (manchmal auch nur API), kann man bei Oracle herunterladen. Die aktuelle Version ist auch online verfügbar.

Das abgebildete Beispiel demonstriert die Verwendung der Klasse `ArrayList`. Hier sollen Objekte der Klasse `Getraenk` in einer Liste gespeichert werden.

```
 1  import java.util.*;
 2
 3  public class Getraenkeverwaltung
 4  {
 5      public static void main (String args[])
 6      {
 7          Getraenk g;
 8          String name;
 9          double preis;
10          ArrayList<Getraenk> getraenkeListe = new ArrayList<Getraenk>(8);
11
12          while(true)
13          {
14              name = Tastatur.stringInput();
15
16              if(name.equals("ende"))
17              {
18                  break;
19              }
20
21              preis = Tastatur.doubleInput();
22
23              g = new Getraenk(name, preis);
24              getraenkeListe.add(g);
25          }
```

Zeile 1: Das Package `java.util` muss importiert werden, weil es die Klasse `ArrayList` enthält.

Zeile 7: Ein Objekt `g` der Klasse `Getraenk` wird deklariert, aber noch nicht erzeugt.

Zeilen 8,9: Zwei Hilfsvariablen zur Zwischenspeicherung von Attributwerten werden deklariert.

Zeile 10: Das Objekt `getraenkeListe` der Klasse `ArrayList` wird erzeugt. Der in den spitzen Klammern angegebene Parameter `Getraenk` legt fest, dass in der Liste ausschließlich Objekte der Klasse `Getraenk` gespeichert werden können. Der Parameter `8` innerhalb der runden Klammer des Konstruktors initialisiert die maximale Anzahl an gespeicherten Objekten.

Zeilen 12-25: Innerhalb der Schleife kann der Anwender mehrmals ein Objekt der Klasse `Getraenk` erzeugen, die jeweiligen Attributwerte sind per Tastatur einzugeben. Wird für den Namen des Getränks die Zeichenkette "ende" eingegeben, so wird die Schleife verlassen.

Zeile 23: Das Objekt `g` der Klasse `Getraenk` wird erzeugt.

Zeile 24: Der Methode `add` des Objekts `getraenkeListe` wird ein Verweis (Referenz) auf das Objekt `g` übergeben. Innerhalb der Methode wird eine Kopie des Objekts an der ersten freien Position der Liste gespeichert. (Anmerkung: Das Objekt `g` kann in der Schleife beliebig oft überschrieben werden, nachdem eine Kopie in der Liste gespeichert wurde.)

Das folgende Beispiel zeigt, wie man das erste Objekt in der Liste (Index 0) herausholt. Das Objekt wird dabei in der Liste nicht gelöscht. Es wird eine Kopie des Objekts erzeugt, welche dann als `getr` für weitere Verwendungen zur Verfügung steht.

```
Getraenk getr = getraenkeListe.get(0);
```

10. Objektdiagramm

Das Objektdiagramm ist (neben dem bereits bekannten Klassendiagramm) eines der 14 UML-Diagrammtypen. Es zeigt den Zustand von Objekten, also die Belegung der Attribute, zu einem bestimmten Zeitpunkt. Auch sind Beziehungen zwischen Objekten dargestellbar, darauf wird aber erst später eingegangen.

Beispiel:
Bild 1 zeigt ein Programm, welches ein Objekt der Klasse `Getraenk` erzeugt und verwendet. Die Struktur der Klasse `Getraenk` ist in Bild 2 dargestellt. Bild 3 zeigt die Ausgaben, welche das Programm erzeugt.

Bild 1

```
2  public class Getraenkeverwaltung
3  {
4      public static void main (String args[])
5      {
6          Getraenk getr1 = new Getraenk("Blue Horse", 2.49);
7          System.out.println("Name: " + getr1.getName());
8          System.out.println("Preis: " + getr1.getPreis());
9          getr1.setPreis(3.19);
10         System.out.println("\n* Preiserhoehung! *");
11         System.out.println("Name: " + getr1.getName());
12         System.out.println("Preis: " + getr1.getPreis());
13     }
14 }
```

Bild 2

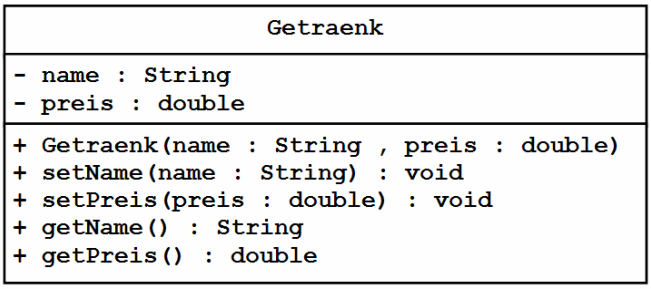
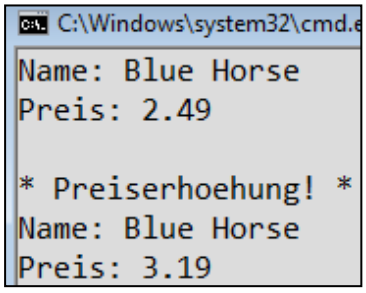


Bild 3



Auf der Programmzeile 6 wird das Objekt `getr1` erzeugt, dabei werden allen Attributen Werte zugewiesen. Bild 4 zeigt das Objektdiagramm für den Zeitpunkt unmittelbar nach Ausführung der Programmzeile 6. Bild 5 zeigt das Objektdiagramm nach Ausführung der Programmzeile 9, hier wurde der Wert des Attributs `preis` verändert.

Bild 4

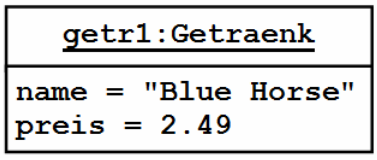
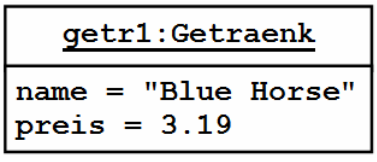
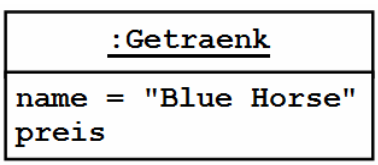


Bild 5



Objektdiagramme enthalten die Bezeichner (Namen) von Objekten, Klassen und Attributen sowie Attributwerte. Dabei ist die in den Bildern 4 und 5 erkennbare Notation einzuhalten. Sollten allerdings Elemente unbekannt sein, so dürfen sie entfallen. Beispiel: Angenommen, der Objektbezeichner und auch der Wert des Attributes `preis` von einem Objekt der Klasse `Getraenk` sei unbekannt. Damit würde sich das in Bild 6 dargestellte Objektdiagramm ergeben.

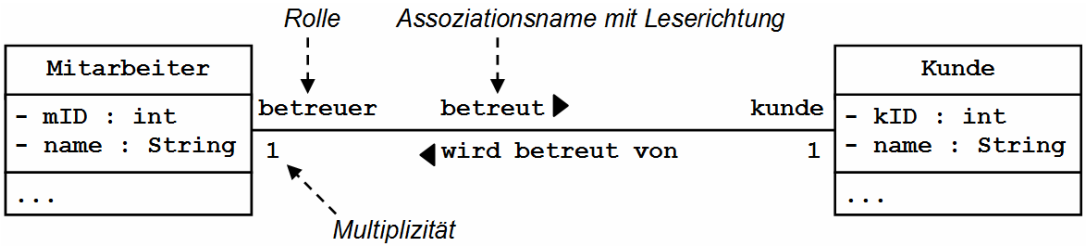
Bild 6



11. Assoziationen in der UML und deren Realisierung in Java

11.1 Darstellung einer allgemeinen Assoziation im Klassendiagramm

Assoziationen sind Beziehungen zwischen Objekten. Da Objekte immer zu einer Klasse gehören, bestehen auch Assoziationen zwischen Klassen. Im einfachsten Fall wird im Klassendiagramm eine Assoziation durch eine Linie dargestellt, welche zwei Klassen verbindet. Beispiel: Die Software einer Firma soll deren Mitarbeiter und Kunden verwalten. Jeder Mitarbeiter besitzt eine Nummer (Attribut `mID`, das Kürzel ID bedeutet Identifikator und steht immer für eine eindeutige Kennung) und einen Namen (Attribut `name`). Kunden besitzen ebenfalls eine Nummer (`kID`) und einen Namen. Klassendiagramm:



Erläuterung der Assoziation: Die Mitarbeiter besitzen hier die Rolle eines Betreuers. Die Kunden treten hier in der Rolle des Kunden auf. Die jeweilige Rolle erscheint an den Enden der Verbindungslinie neben der zugehörigen Klasse. Sie ist ein Substantiv und beginnt mit einem Kleinbuchstaben. Assoziationsnamen sollten möglichst Verben sein und beginnen ebenfalls mit einem Kleinbuchstaben. Zu einem Assoziationsnamen gehört immer eine Leserichtung. Für Rollen und Assoziationsnamen sind aussagekräftige Bezeichner zu verwenden. Rollen und/oder Assoziationsnamen dürfen entfallen, wenn die Bedeutung der Assoziation dabei offensichtlich bleibt. Die Multiplizität (auch Wertigkeit genannt) gibt an, wieviel Beziehungen ein bestimmtes Objekt zu anderen Objekten haben kann bzw. darf. Hier gilt: Ein Mitarbeiter betreut genau einen Kunden. Und ein Kunde wird von genau einem Mitarbeiter betreut.

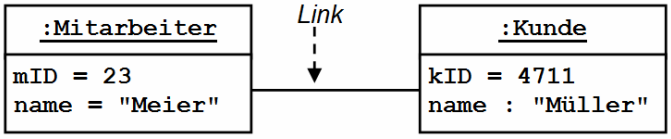
Multiplizitäten bestehen immer aus genau einem zusammenhängendes Zahlen-Intervall mit einer unteren und einer oberen Schranke an, Beispiele:

`0..5`	von einschließlich 0 bis einschließlich 5
`1..*`	von einschließlich 1 bis zu einem beliebigen Wert, welcher nicht unter 1 liegt
`*`	Abkürzung für `0..*`, also eine beliebige Zahl
`1`	Abkürzung für `1..1`, also genau 1

11.2 Darstellung einer allgemeinen Assoziation im Objektdiagramm

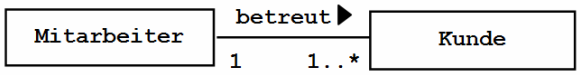
Beziehungen zwischen Objekten können in einem Objektdiagramm dargestellt werden. Dabei wird zwischen den Objekten eine Verbindungslinie gezeichnet, welche hier als Link bezeichnet wird. Weil an den beiden Enden eines Links immer genau ein Objekt steht, entfällt die Angabe von Multiplizitäten. Für Rollen und Assoziationsnamen gelten die bereits für das Klassendiagramm gemachten Aussagen.

Das Bild zeigt ein mögliches Objektdiagramm, welches zum oben dargestellten Klassendiagramm passt.

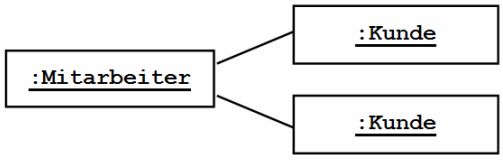


Jetzt werden die Multiplizitäten am Klassendiagramm geändert:

Klassendiagramm (vereinfacht)

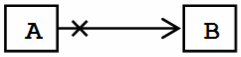


... und ein mögliches Objektdiagramm (vereinfacht)

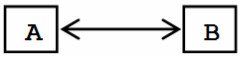


11.3 Navigierbarkeit

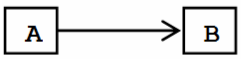
Für die vollständige Beschreibung einer Assoziation ist die eindeutige Angabe der Navigierbarkeit erforderlich. Was damit gemeint ist, sollen die folgenden Beispiele (vereinfachte Klassendiagramme) verdeutlichen. Nur die ersten beiden Beispiele zeigen eine eindeutige Navigierbarkeit.



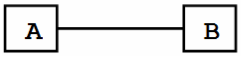
Von einem Objekt der Klasse A muss ein direkter Zugriff auf ein Objekt der Klasse B möglich sein. Von einem Objekt der Klasse B darf kein direkter Zugriff auf ein Objekt der Klasse A möglich sein.



Von einem Objekt der Klasse A muss ein direkter Zugriff auf ein Objekt der Klasse B möglich sein. Von einem Objekt der Klasse B muss ein direkter Zugriff auf ein Objekt der Klasse A möglich sein.



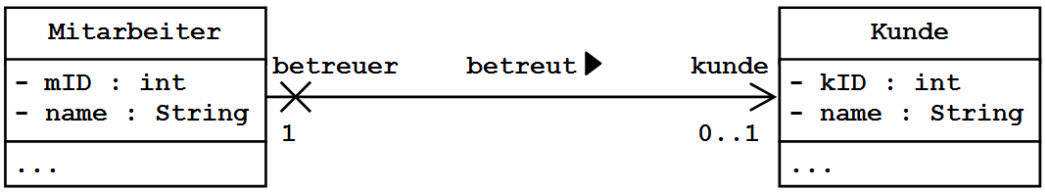
Von einem Objekt der Klasse A muss ein direkter Zugriff auf ein Objekt der Klasse B möglich sein. Für die umgekehrte Richtung gibt es keine Festlegung.



Hier existieren für beide Richtungen keine Festlegungen.

11.4 Realisierung von 1 zu 1 - Assoziationen im Javacode

Eine 1 zu 1 (oder 1:1) - Assoziation liegt vor, wenn ein Objekt maximal 1 anderes Objekt "kennt" und umgekehrt. Nachdem die Datenstruktur einer Software als Klassendiagramm entworfen wurde, erfolgt die entsprechende Umsetzung in Java. Zunächst werden die einzelnen Klassen erstellt. Dabei sind, wie bereits bekannt, alle Vorgaben bzgl. Attribute, Methoden und Konstruktoren zu berücksichtigen. Aus den Assoziationen zwischen den Klassen ergeben sich dann Ergänzungen im Code, was jetzt mittels Beispiel erläutert wird.



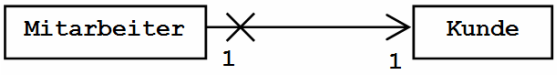
Weil die Objekte der Klasse `Mitarbeiter` die Objekte der Klasse `Kunde` "kennen" müssen, ergeben sich Ergänzungen im Code, welche hier durch Kommentare erläutern werden:

```
 2  public class Mitarbeiter
 3  {
 4      // "normale" Attribute definieren
 5      private int mID;
 6      private String name;
 7
 8      /* Weil das Mitarbeiter-Objekt das Kunden-Objekt (Objekt der Klasse Kunde) "kennen" muss,
 9      wird eine Referenz zum Kunden-Objekt erzeugt (Referenzattribut). Der Bezeichner für diese
10      Referenz lautet "kunde" und muss mit dem im Klassendiagramm angegebenen Rollennamen
11      übereinstimmen. (Falls der Rollenname fehlt, ist ein sinnvoller Bezeichner zu wählen. */
12
13      private Kunde kunde;
14
15      //Zur Verwaltung der Referenz kunde müssen set- und get-Methoden implementiert werden:
16
17      public void setKunde(Kunde kunde)
18      {
19          this.kunde = kunde;
20      }
21
22      public Kunde getKunde()
23      {
24          return kunde;
25      }
26
27      // Jetzt werden noch set- und get-Methoden für die "normalen" Attribute implementiert...
```

Laut Klassendiagramm darf ein Kunden-Objekt das Mitarbeiter-Objekt nicht "kennen". Damit entfallen das entsprechende Referenzattribut und alle zugehörigen Methoden. Die Klasse `Kunde` enthält damit keinerlei Ergänzungen bzgl. der Assoziation zur Klasse `Mitarbeiter`.

Die gerade beschriebene Klasse `Mitarbeiter` muss laut Klassendiagramm keine Konstruktoren besitzen, sie wurden deshalb auch nicht implementiert.

Falls jeder Mitarbeiter genau einen Kunden haben muss, ändern sich die Multiplizitäten: (Klassendiagramm verkürzt dargestellt)



In diesem Fall muss beim Erzeugen eines Mitarbeiter-Objekts das zugehörige Kunden-Objekt dem Konstruktor übergeben werden. Die folgende Abbildung zeigt einen passenden Konstruktor für die Klasse `Mitarbeiter`, welcher neben dem Referenzattribut auch die "normalen" Attribute setzt:

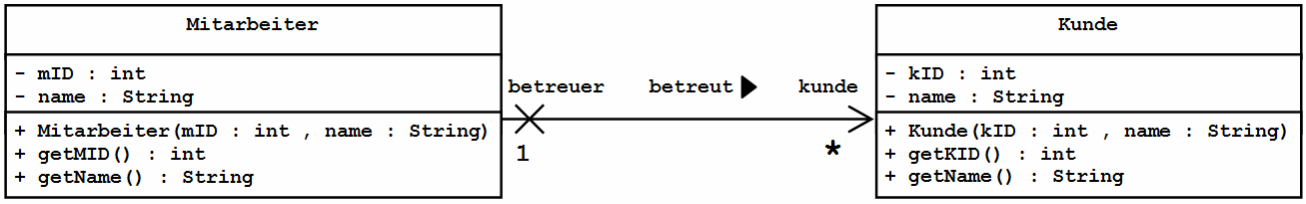
```
public Mitarbeiter(int mID, String name, Kunde kunde)
{
    this.mID = mID;
    this.name = name;
    this.kunde = kunde;
}
```

Die folgende Abbildung zeigt eine passende Startklasse mit der main-Methode. Hier wird vorausgesetzt, dass die Klasse `Mitarbeiter` den oben abgebildeten Konstruktor besitzt. Ebenfalls wird vorausgesetzt, dass auch die Klasse `Kunde` einen Konstruktor besitzt, welcher beide Attribute setzt (siehe Zeile 13).

```
 2  public class Verwaltung
 3  {
 4      public static void main (String args[])
 5      {
 6          /* Ein neuer Mitarbeiter wird angelegt. Weil sein zu betreuender Kunde noch
 7          nicht bekannt ist, wird dem Konstruktor als Referenz zum Kunden-Objekt
 8          zunächst null übergeben. */
 9
10          Mitarbeiter betreuer = new Mitarbeiter(101, "Hoppenstedt", null);
11
12          // Ein neuer Kunde wird angelegt:
13          Kunde kunde = new Kunde(1005, "Meier");
14
15          // Die Objektreferenz zum neuen Kunden wird dem Mitarbeiter-Objekt übergeben:
16          betreuer.setKunde(kunde);
17
18          System.out.println("Der Mitarbeiter " + betreuer.getName() + " betreut ");
19          System.out.println("diesen Kunden: " + betreuer.getKunde().getName());
20      }
21  }
```

11.5 Realisierung von 1 zu n - Assoziationen im Javacode

Eine 1 zu n (oder 1:n) - Assoziation liegt vor, wenn z. B. ein Objekt der Klasse `Mitarbeiter` mehr als ein Objekt der Klasse `Kunde` "kennen" kann. Das folgende Beispiel zeigt eine solche Beziehung: Der Mitarbeiter einer Firma betreut eine beliebige Anzahl von Kunden:



Weil jetzt im Objekt der Klasse `Mitarbeiter` mehrere Referenzen zu den Objekten der Klasse `Kunde` gespeichert werden müssen, wird als Referenzattribut in der Klasse `Mitarbeiter` eine Liste benötigt. Hier kann z. B. die JDK-Klasse `ArrayList` genutzt werden. Um die Liste verwalten zu können, sind Methoden zum Hinzufügen, Löschen und Auslesen von Referenzen zu implementieren.

Hinweise:
Das Referenzattribut (Liste) muss nicht zwingend im Klassendiagramm enthalten sein, weil es sich aus der Assoziation ergibt. Die zugehörigen Methoden sollten aber im Klassendiagramm und/oder einer zusätzlichen Beschreibung enthalten sein. Schließlich gibt es eine Vielzahl von möglichen Methoden zur Verwaltung einer Liste, von denen dann möglicherweise nur sehr wenige tatsächlich genutzt werden.

Es folgt der Javacode der Klasse `Mitarbeiter`. Als Referenzattribut wird auf Zeile 8 ein Listen-Objekt der JDK-Klasse `ArrayList` erzeugt. Zur Verwaltung dieses Attributs werden drei Methoden implementiert, welche folgende Operationen ermöglichen:

- Kunden-Objekt zur Liste hinzufügen (Zeilen 26-29)
- Kunden-Objekt in der Liste löschen (Zeilen 31-34)
- komplette Liste zurückgeben (Zeilen 36-39)

```
 2  import java.util.ArrayList;
 3
 4  public class Mitarbeiter
 5  {
 6      private int mID;
 7      private String name;
 8      private ArrayList<Kunde> kunden = new ArrayList<Kunde>();
 9
10      public Mitarbeiter(int mID, String name)
11      {
12          this.mID = mID;
13          this.name = name;
14      }
15
16      public int getMID()
17      {
18          return mID;
19      }
20
21      public String getName()
22      {
23          return name;
24      }
25
26      public void hinzufuegenKunde(Kunde einKunde)
27      {
28          kunden.add(einKunde);
29      }
30
31      public void loeschenKunde(Kunde einKunde)
32      {
33          kunden.remove(einKunde);
34      }
35
36      public ArrayList<Kunde> getKundenliste()
37      {
38          return kunden;
39      }
40  }
```

Anmerkung: Die Methode `add` der Klasse `ArrayList` gibt einen `boolean`-Wert zurück. Wenn dieser im Programm genutzt werden soll, so muss die Methode `hinzufuegenKunde` entsprechend geändert werden:

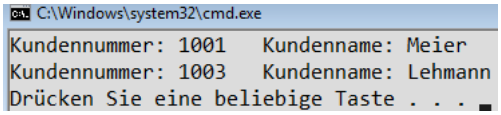
```
public boolean hinzufuegenKunde(Kunde einKunde)
{
    return kunden.add(einKunde);
}
```

Laut Klassendiagramm darf ein Kunden-Objekt das Mitarbeiter-Objekt nicht "kennen". Damit entfallen das entsprechende Referenzattribut und alle zugehörigen Methoden. Die Klasse `Kunde` wird also 1:1 vom Klassendiagramm in den Javacode umgesetzt, welcher hier nicht dargestellt wird.

Die folgende Abbildung zeigt eine passende Startklasse mit der main-Methode:

```
 2  import java.util.ArrayList;
 3
 4  public class Verwaltung
 5  {
 6      public static void main (String args[])
 7      {
 8          // Ein neues Mitarbeiter-Objekt wird erzeugt:
 9          Mitarbeiter betreuer = new Mitarbeiter(101, "Hoppenstedt");
10
11          // Mehrere Kunden-Objekte werden erzeugt:
12          Kunde k1 = new Kunde(1001, "Meier");
13          Kunde k2 = new Kunde(1002, "Schulze");
14          Kunde k3 = new Kunde(1003, "Lehmann");
15
16          // Referenzen zu Kunden-Objekten im Mitarbeiter-Objekt speichern:
17          betreuer.hinzufuegenKunde(k1);
18          betreuer.hinzufuegenKunde(k2);
19          betreuer.hinzufuegenKunde(k3);
20
21          // Referenz zu einem Kunden-Objekt im Mitarbeiter-Objekt löschen:
22          betreuer.loeschenKunde(k2);
23
24          // komplette Liste mit allen Referenzen zu Kunden-Objekten auslesen:
25          ArrayList<Kunde> liste = betreuer.getKundenliste();
26
27          // Kundendaten zur Kontrolle auf Konsole ausgeben:
28          for(int i = 0; i < liste.size(); i++)
29          {
30              System.out.print("Kundennummer: " + liste.get(i).getKID());
31              System.out.println("   Kundenname: " + liste.get(i).getName());
32          }
33      }
34  }
```

Die main-Methode erzeugt diese Ausgaben:



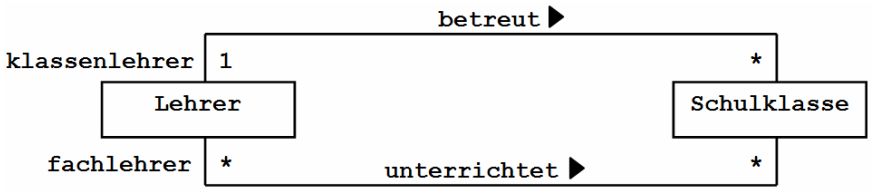
Ersetzt man in der oben abgebildeten main-Methode die Zeilen 9 - 19 durch den folgenden Code, so können die Kundennamen per Tastatur eingegeben werden. Die Kundennummern (Attribut `kID` der Klasse `Kunde`) werden automatisch (beginnend bei 1001) erzeugt.

```
Mitarbeiter betreuer = new Mitarbeiter(101, "Hoppenstedt");
Kunde kunde;
String name;

for(int i = 0; i < 3; i++)
{
    //Kundenname eingeben:
    name = Tastatur.stringInput();

    //Kunden-Objekt erzeugen, Kundennummer wird dabei automatisch erzeugt:
    kunde = new Kunde(1001+i, name);

    //Kunden-Objekt-Referenz übergeben:
    betreuer. hinzufuegenKunde(kunde);
}
```

11.6 Spezielle Assoziationen

a) Es gibt mehr als eine Beziehung zwischen Objekten zweier Klassen

Beispiel:



Aus dem Klassendiagramm ergibt sich zwangsläufig folgende Situation: In einer Schule unterrichten viele Lehrer in vielen Schulklassen. Jede Schulklasse wird von genau einem Klassenlehrer betreut.

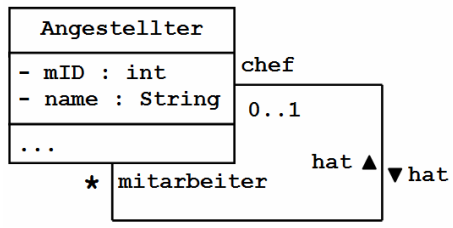
Weil hier zwei Beziehungen zwischen den Klassen `Lehrer` und `Schulklasse` existieren, sind die Beziehungsnamen im Klassendiagramm zwingend erforderlich. Und weil hier der Lehrer in zwei verschiedenen Rollen auftritt (er kann sowohl betreuender Klassenlehrer als auch unterrichtender Fachlehrer sein), müssen im Klassendiagramm diese Rollen unbedingt angegeben werden. Die Rolle der Schulklasse ist für beide Beziehungen gleich und eindeutig, entsprechende Bezeichner dürfen damit entfallen.

Für die Umsetzung des Klassendiagramms in den Programmcode gelten die bereits bekannten Regeln, dabei muss jede Beziehung für sich realisiert werden.

b) Reflexive Assoziation

Gibt es Beziehungen zwischen Objekten, welche zur selben Klasse gehören, so spricht man von einer reflexiven Assoziation. Ein Beispiel ist hier als Klassendiagramm dargestellt:

In einer Behörde gibt es viele Angestellte, welche alle (also auch Führungskräfte) in einer Software als Objekte der Klasse `Angestellter` verwaltet werden. Alle "einfachen" Mitarbeiter haben genau einen Chef, z. B. den Gruppenleiter. Alle Gruppenleiter ("höher gestellte Mitarbeiter") haben ebenfalls genau einen Chef, z. B. den Abteilungsleiter. Deren Chef ist der Behördenleiter, welcher als Einziger keinen Chef innerhalb der Behörde hat.

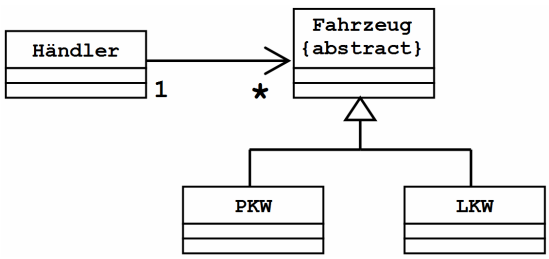


Für die eindeutige Beschreibung der Beziehung sind Angaben der Rollen und Beziehungsnamen erforderlich.

c) Assoziationen mit abstrakten Klassen

Ein Beispiel ist hier als Klassendiagramm dargestellt:

Ein Händler besitzt beliebig viele Fahrzeuge, dies sind entweder PKW oder LKW.
Hier existiert eine Beziehung zwischen der Klasse `Händler` und der abstrakten Klasse `Fahrzeug`, von welcher die Klassen `PKW` und `LKW` erben.



Aus der 1:n - Beziehung ergibt sich ein Listen-Attribut, welches z. B. so realisiert werden kann:

```
ArrayList<Fahrzeug> liste = new ArrayList<Fahrzeug>();
```

Das Objekt `liste` der Klasse `ArrayList` kann jetzt Referenzen zu Objekten der Klassen `PKW` und `LKW` speichern, weil diese von der Klasse `Fahrzeug` erben.

Wenn allerdings PKW und LKW strikt getrennt verwaltet werden sollen, so müssen im Klassendiagramm entsprechende Beziehungen zwischen beiden Subklassen (`PKW` und `LKW`) und der Klasse `Händler` dargestellt werden. Die Beziehung zwischen `Händler` und `Fahrzeug` muss dann entfallen.