Objekte und Klassen

(Delegation, Vererbung, Polymorphismus)

www.programmieraufgaben.ch

Philipp Gressly Freimann 19. Juni 2015 (Version 3.3.8 (TeX))

Impressum

19. Juni 2015

http://www.programmieraufgaben.ch

Dieses Buch kann in der neuesten Version frei heruntergeladen werden:

- http://www.programmieraufgaben.ch/uploads/oo.pdf
- http://www.gress.ly/oo/oo.pdf

Grafik: Philipp Gressly Freimann

Editor: GNU Emacs 23.3.1

Publisher: LaTeX: pdfTeX 3.1415926-1.40.10-2.2

Inhaltsverzeichnis

Präambel Eine geschlechtsneutrale Bezeichnung von Personen oder eine Bezeichnung verschiedener Geschlechter wurde weitgehend vermieden, um die Lesbarkeit des vorliegenden Dokuments zu erleichtern. Alle Leser sind selbstverständlich gleichermaßen angesprochen.

Vorwort

Noch ein Buch über die Objektorientierung mit JAVA, **ist das wirklich nötig**, Herr Gressly Freimann? Zugegeben, es gibt genügend Werke über die Syntax der JAVA-Programmiersprache, genügend Werke über die objektorientierte Programmierung und ja, sogar genügend Werke über die objektorientierte Programmierung **mit** JAVA. Doch wie «Programmieren lernen» [GFG11] (s. Literatur auf Seite 167) schließt auch dieses Buch eine Lücke: Die meisten Werke erläutern und vertiefen die Vererbung und den Polymorphismus[1] bzw. gehen stark auf syntaktische Details von JAVA ein. Für Umsteiger von rein strukturierten Sprachen bzw. für JAVA-Programmierer ist dies auch sinnvoll. Wer jedoch vor allem die Konzepte kennen lernen will, der braucht sich hingegen an einen `++`-Operator, ein `protected` oder gar ein `<<<` wirklich keinen einzigen Gedanken zu verlieren. Solche JAVA-Spezialitäten erhalten in diesem Band ganz bewusst nur marginale Präsenz.

Dieses Buch geht im zentralen Teil vorwiegend auf den **Methoden**aufruf und die **Delegation** ein. Ein Konzept, das meines Erachtens die notwendige Grundlage liefert, die Tragweite der Objektorientierung überhaupt zu begreifen und ein Konzept, dem sowohl in Lehrbüchern aber auch in der Aus- und Weiterbildung ein Schattendasein anhaftet. So sehr, dass oft sogar Lehrbuchbeispiele die Vererbung anwenden, wo eine Delegation einfacher, sinnvoller, weniger fehleranfällig und darüber hinaus wartbarer wäre!

Auch der `this`-Pointer ist für viele ein Buch mit sieben Siegeln. Wer jedoch die Syntax der Delegation in modernen Sprachen geübt hat, verwendet den `this`-Pointer auf natürliche Weise und hat in der Regel auch verstanden, wie dieser im Detail funktioniert. Um diese spezielle Referenzvariable kennen zu lernen, sollte genügend mit Methoden und allenfalls mit der Delegation geübt werden.

JAVA wird hier lediglich als Werkzeug verwendet. Es werden diejenigen Konzepte gezeigt, die auch in den meisten anderen OO-Sprachen verwendet werden können.

[1] S. Kapitel über die Delegation (S. 63), die Vererbung (S. 83) und den Polymorphismus (S. 105).

Inhalt

Dieses *Buch* setzt den Theorieteil fort, der in [GFG11] mit Kapitel 1-11 begonnen wurde. Der vorliegende Teil zeigt, welche Techniken und modernere Konzepte sich in der Softwareentwicklung durchgesetzt haben, namentlich die **objektorientierte Programmierung**.

Grundlegende neue Möglichkeiten zur Problemlösung mittels Computer sind hier nicht zu erwarten. Auch die objektorientierte Softwareentwicklung erzeugt immer noch Maschinencode mit den — unter anderen — bekannten Konstrukten der Sequenz und des *Jumps*.

Es hat jedoch in der Softwareentwicklung ein leichter Paradigmenwechsel stattgefunden, sodass nicht mehr der Algorithmus im Zentrum steht. Neu wird von Klassen des Fachgebietes (sog. Fachklassen[2]) ausgegangen und die Algorithmen werden diesen Objekten hinzugefügt.

Die Grundlagen aus [GFG11] sind weiterhin Basis für solide Programme, jedoch für große Projekte werden mehr und mehr vorgefertigte Klassen verwendet, die sinnvoll miteinander verknüpft werden. So programmiert heute wohl niemand mehr einen Algorithmus aus, der eine Liste von Personen nach Vornamen sortiert; hingegen wird eine Klassenbibliothek aufgerufen, die nur noch entsprechend konfiguriert werden muss.

Im Gegensatz zu [GFG11] werden die Konzepte und Aufgaben in diesem Band nicht unabhängig von einer Sprache formuliert. Alle Beispiele, Aufgaben und Lösungen sind hier in JAVA gehalten.

Dank

Insbesondere möchte ich mich bei den Applikationsentwicklerlehrlingen der Credit Suisse bedanken, die jahrelang diese Unterlagen – oft ohne ihr Wissen – verbessert hatten.

Ein weiterer Dank geht an all jene, welche sich die Mühe genommen haben, dieses Büchlein zu sichten und mich mit wertvollen Rückmeldungen eingedeckt haben: Milan Sismanovič (der mich schon im Band I [GFG11] unterstützt hatte), Andreas Arnold (fürs Durchlesen und die Sportchef-Idee), die Klasse CS-IL08 für die Kamele, Rémy Gressly fürs Korrekturlesen und kritische Fragen stellen, Herr Dave Brandstetter und Herr Cedric Annen fürs Lektorat, Joanna Pavel für wertvolle Verbesserungen in Stil und Verständlichkeit und weitere Personen.

[2] Fachklassen, engl. Domain-Classes; werden aus der Sicht der IBM (International **Business** Machines Corporation) im Zusammenhang mit der Wirtschaftsinformatik lediglich «Business-Klassen» genannt.

«Alter Wein in neuen Schläuchen»

Das Ziel dieses Kapitels ist das Kennenlernen der Geschichte und der grundlegenden Konzepte der Objektorientierung, namentlich der **Records**. Da diese Eigenschaft so zentral ist, habe ich ihr auch ein eigenes Kapitel gewidmet (s. Kap. 13 auf Seite 21).

Die folgenden objektorientierten Ideen sind auch auf Programmiersprachen anwendbar, die noch nicht als objektorientiert gelten. Lediglich das Konzept der Records muss in der Sprache implementiert sein.

Die Records machen jedoch nur einen Teil der Objektorientierung aus. Und da die Art der Programmierung entscheidend ist, ob eine Anwendung als objektorientiert zu betrachten ist oder nicht, beobachten wir die folgenden drei Hauptelemente, die ein Objektmodell auszeichnen:

- Abstraktion (Abstraction)
- Kapselung (Encapsulation) und Informationsschutz (Information Protection[3])
- Hierarchie (Hierarchy) und Vererbung (Inheritance)

Ein Modell ohne mindestens eines dieser Hauptelemente ist per Definition nicht objektorientiert.

[3]Der Informationsschutz (Information Protection) wird manchmal auch Geheimnisprinzip (*Information Hiding*) genannt. Doch um Geheimnisse geht es bei diesem Konzept nicht, und daher wird von mir lieber der Begriff «Protection» verwendet.

12.1 Objekte und Kapselung

Bei der Objektorientierung (wie in relationalen Datenbanken) werden alle in der realen Welt vorkommenden Gegenstände als Objekte (bzw. Entitäten) verstanden.

12.1.1 Objekte

Ein Objekt ist in der *objektorientierten Programmierung* ein eindeutig identifizierbares Element. Das Objekt enthält bestimmte Informationen und weiß, wie diese Informationen zu verarbeiten sind (oder welche Operationen damit ausgeführt werden können).

Kandidaten für Objekte und Klassen in der objektorientierten Analyse sind:

- Ein Ding, das sichtbar und beschreibbar ist
  - **Reale Dinge**: Buch, Kaffee, Kaffeemaschine, Kamel, Herde, Drucksensor, ...
- Etwas, das man sich vorstellen kann
  - **Rollen**: Angestellter, Kunde, Lieferant, ...
  - **Spezifikationen**: Automarke, PC-Typ, Teenamen, ...
- Etwas, das in Taten umgesetzt werden kann oder etwas **nicht** Gegenständliches
  - **Ereignisse / Vorgänge**: Laden, Bestellen, Einladung, Initialisierung, ...
  - **Interaktionen**: Zusammenarbeit, Treffen, ...

12.1.2 Kapselung

Ein Objekt kann in gewissem Sinne als Aussage bezüglich Verknüpfung von bestimmten Eigenschaften (Kenntnissen) und Operationen (Subroutinen, Methoden) angesehen werden, sodass eine Bündelung dieser beiden Aspekte sinnvoll ist. Diese Bündelung bezeichnet man als **Kapselung**.

Im ganzen Buch gehe ich nicht weiter auf den Begriff der Kapselung ein. Moderne Sprachen nehmen dies für uns ab und für eine Einführung in die Objektorientierung ist dies auch nicht zentral. Wer größere Applikationen schreibt, wird hingegen nicht darum herum kommen, seine Attribute und Methoden mehr oder weniger zu schützen (`public` , `private`).

12.1.3 Taxonomie (ist-ein)

Die alten Griechen (Platon/Aristoteles) haben mit ihrer *Erkenntnistheorie*[4] einen wichtigen Grundstein zum Verständnis von Objekten gelegt.

Beispiel 12.1. Das klassische Beispiel ist *Liese das Pferd*. Liese (das **Objekt**) wird geboren, wird alt und stirbt. Die Idee oder Form (oder eben die **Klasse**) *Pferd* bleibt hingegen ewig, auch dann, wenn es bereits keine Pferde mehr geben würde.[5]

Die Taxonomie wird durch die «ist ein»-Beziehung wiedergegeben:

- Herr Herbert Huber **ist ein** Bankkunde.
- Herbert **ist ein** Kassierer im lokalen Kochclub.
- Herbert Huber **ist ein** Patient.

Taxonomie bedeutet so viel wie «Klassifizierung» oder «Einteilung von Dingen». Im folgenden Beispiel werden die Menschen (Objekte) eingeteilt in *Kunden*, *Clubmitglieder*, *Patienten* usw. (die Arten, die Gattungen, die Familien, ... eben die Klassen).

Die Taxonomie von Dingen ist nicht von vornherein klar, oder «von der Natur gegeben». Erst unsere **Betrachtung** macht die Objekte der Realität zu Exemplaren von Klassen.[6]

Es ist also wesentlich, **wie** «Mensch» (hier unser Herr Huber) **abstrahiert** wird.

Beispiel 12.2. Herr Huber ist wie erwähnt ein Kunde. Das stimmt aus Sicht der Bank, wo Herr Huber sein Geld verwalten lässt. Als Kunde hat er bestimmte Merkmale, für welche sich die Bank interessiert. Herr Huber ist aber gleichzeitig auch Kassierer in einem lokalen Club und ebenso ist er ein Patient.

Ein Mensch als Bankkunde besteht aus «Name, Vorname, Adresse, Kundennummern, Konti, Arbeitgeber, Bonität, ...».

Als «Clubmitglied» hat er andere *Attribute*: «Name, Vorname, E-Mail, Adresse, Funktion, Eintrittsjahr, Clubbeitrag, ...»

Derselbe Mensch als Patient besteht aus «Name, Vorname, AHV-Versicherungs-Nummer, Krankenkasse, Krankheitsgeschichte, ...».

Obige Beispiele zeigen klar die Grenzen des Objektkonzepts aus der Sicht des Betrachters. Hier bildet die **Abstraktion** ein Verfahren, welches **aus einer bestimmten Sicht** die wesentlichen Merkmale eines Gegenstandes oder Begriffes zeigt.

[4] Englisch: *Theory of Forms* (Platon: Ideenlehre / Aristoteles: Metaphysik)

[5] In der Erkenntnistheorie könnte man das **Objekt** mit *dem Stoff* oder mit *der Materie* in Verbindung bringen. Die **Klasse** wäre dann in etwa die *Idee* oder *Form*.

[6] Hier ein stark vereinfachtes Beispiel: Der Stoff (= die Materie aus der Erkenntnistheorie) ist bereits vorhanden. Doch erst das Beschreiben, das Erkennen oder das Klassifizieren macht Materie zu **Objekten**. Beim Anfertigen von "Guetsli"(-Objekten) aus einem Stück Teig (Stoff) z. B. ist der Zimtstern bereits vor dem Ausstechen vorhanden. Doch erst das Anwenden der **Klasse** (Form) lässt uns den Stern erkennen ;-)

12.1.4 Das Objekt-Klassen Prinzip

Eine Klasse beschreibt die Struktur und das Verhalten einer Menge gleichartiger Objekte. Eine Klasse ist sozusagen der Bauplan.

Als Programmierer schreiben wir die (JAVA-)Klassen (das Kamel-Förmchen in der folgenden Grafik). Von diesen Klassen werden später (auch von Programmierern) neue Objekte erstellt.

Die obige Grafik kann im Zusammenhang mit objektorientierten Programmen wie folgt verstanden werden:

Weihnachtsgebäck	**Datenstruktur**
Teig	Hauptspeicher (Heap)
Keksform	Klasse (struct, record, Entitätstyp)
Ausgestochener Keks	Objekt, Instanz
Lücke im Teig	nicht mehr verwendbarer, also belegter Speicher
ausstechen	erzeugen von Objekten (`new` in JAVA, `malloc` in C)

Die folgende Tabelle zeigt den Unterschied zwischen Klassen und den dazugehörigen Objekten:

Klasse	**Objekte** (Exemplare, instances)
Ausstechform	Kekse
Form oder Idee	Stoff oder Materie
Entitätstyp, Tabelle (Relation in der Datenbank)	Entität (als Gegenstand, Rolle, Wesen oder Konzept), Tupel (= Zeile in der Datenbank)
struct, Datensequenz (record)	Variablenwert oder Exemplar des Recordtyps

12.2 Klassen

Eine Klasse ist die Definition ...

- ... der **Attribute**,
- ... der **Operationen** und
- ... der **Semantik**[7]

für eine Menge von Objekten. Alle Objekte einer Klasse entsprechen dieser Definition.

12.2.1 Attribute (Eigenschaften)

Attribute sind Eigenschaften der Objekte. Alle Objekte einer Klasse besitzen eine eigene Ausprägung dieser Eigenschaften. Natürlich können einige Objekte dieselbe Ausprägung haben, doch es ergibt keinen Sinn, wenn alle Objekte für eine Eigenschaft denselben Wert aufweisen müssen.

Beispiel: Kunde hat ein Attribut **Name**. In der Regel haben nicht alle Kunden denselben Namen, auch wenn das für einzelne Kunden zutreffen kann.

[7] Semantik = Lehre, die sich mit der **Bedeutung** sprachlicher Zeichen befasst.

12.3 Zustand, Verhalten, Identität

Ein Objekt hat einen definierten Zustand (State), ein klar beschriebenes Verhalten (Behaviour) und eine eindeutige Identität (Identity).

Zustand

- Der **Zustand** (State) umfasst sowohl die statischen als auch die dynamischen Merkmale:
  - statisches Merkmal ist die Objektbeschreibung,
  - dynamische Merkmale sind die aktuellen Werte, die sich im Objekt befinden und
  - der Zustand verändert sich immer dann, wenn Subroutinen (Methoden) die Informationen bearbeiten.
- Ein Zustand ist eine Bedingung bzw. Situation im Leben eines Objekts.
- Eine **Zustandsänderung** erfolgt, wenn Aktivitäten ausgeführt werden, eine Bedingung erfüllt oder auf ein Ereignis gewartet wird.
- Der Zustand wird in den Attributwerten festgehalten.

Verhalten

- Das **Verhalten** (Behaviour) zeigt, wie ein Objekt sich in Bezug auf Zustandsänderungen und Meldungsaustausch verhält.
- Das Objekt kann seinen Zustand nur im Rahmen der vordefinierten Nachrichten und Meldungen[8] verändern.

Identität

- Die **Identität** (Object identity) ist die eindeutige Beschreibung eines Objektes, welche ein Exemplar von allen anderen Objekten unterscheidet. Diese Unterscheidung ist auch dann noch gültig, wenn zwei Objekte möglicherweise die selben Attributswerte besitzen.
- Jedes Objekt ist unabhängig von seinen konkreten Attributwerten, und es ist von allen anderen Objekten auch bei gleichem Zustand und gleichem Verhalten eindeutig unterscheidbar.

[8] Unter Nachrichten und Meldungen ist das Aufrufen von Methoden zu verstehen (s. Kap. 16 auf Seite 49).

12.3.1 Beispiel zur Identität

Gegeben ist die folgende Klasse:

```
class Kamel {
  String name;
}
```

Betrachten wir nun folgendes Hauptprogramm:

```
main() {
  Kamel k1, k2, k3, k4;

  // Erzeuge zwei gleichartige Objekte/Instanzen/Exemplare
  // der Klasse  Kamel:
  k1 = new Kamel();
  k1.name = "Aisha";

  k2 = new Kamel();
  k2.name = "Aisha";

  // Erzeuge nun ein drittes Objekt, doch diesmal
  // mit anderem Attributwert:
  k3 = new Kamel();
  k3.name = "Pedro";

  k4 = k1;
  k4.name = "Ali";
}
```

Die Referenzvariablen `k1` und `k2` haben die gleichen Attributwerte bzw. Daten — in diesem Fall der Name; aber sie sind nicht identisch!

Hingegen ist `k4` identisch zu `k1`, weil diese Referenz-Variable auf dasselbe Objekt zeigen bzw. es sind nur verschiedene Zeiger (Pointer) für dasselbe Objekt. In der letzten Zeile wird der Name «Aisha» im ersten Objekt verändert. Da `k1` und `k4` auf dieses Objekt zeigen, könnte man nun glauben, dass sich zwei Objekte verändert haben.

Bemerkung 12.1. In objektorientierten Systemen braucht sich der Entwickler keine Gedanken mehr über eine Identifikationsnummer zu machen. Jedes Objekt ist sowieso einmalig.[9]

Bemerkung 12.2. Manchmal kommt es vor, dass man von einer Klasse nicht beliebig viele Objekte erzeugen lassen will. Ein typischer Vertreter eines derartigen globalen Objektes ist der **Singleton** (s. Kap. A.3 auf Seite 158).

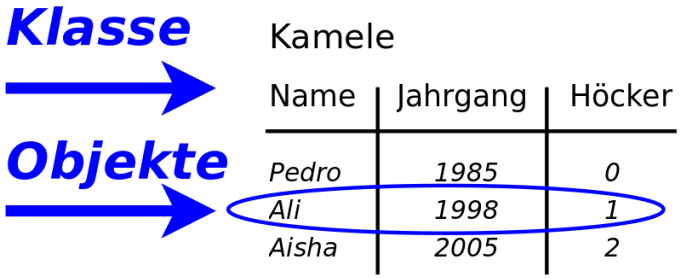
Beispiel 12.3. Jedes Objekt ist (in seiner Klassifizierung) einmalig. Herr Huber als Kunde bei der Bank existiert genau einmal. Natürlich kann es mehrere verschiedene Kunden mit dem Namen „Herbert Huber“ geben. Diese sind dann **gleichartig**, aber es sind **nicht dieselben**!

[9] Sofern notwendig, müssen Identifikationsnummern aus bestehenden (relationalen) Datenbanken übernommen werden, damit dort eine eindeutige Zuordnung stattfinden kann. Objektdatenbanken halten die ID für den Anwender / Entwickler transparent; d. h. in OO-Datenbanken braucht sich der Entwickler überhaupt nicht mehr um die IDs zu kümmern.

13 | Records (Strukturen)

«Klassen, Objekte und Attribute»

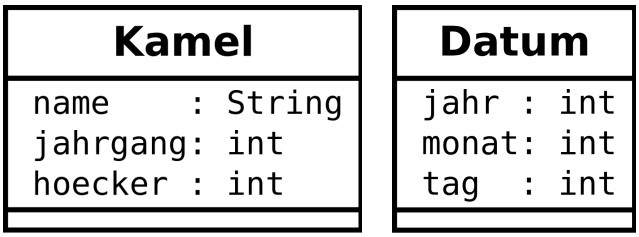
Records sind ein sehr mächtiges Konzept zur Analyse und zur Spezifikation von Software. Wer bereits in einer Tabellenkalkulation mit Spaltenüberschriften gearbeitet hat, kennt das Konzept bereits. Jede Zeile in einer Tabelle entspricht einem Objekt. Die Spaltenüberschrift entspricht der Klassendefinition, welche vorschreibt, was die einzelnen Objekte für Attribute, auch Datenfelder genannt, aufweisen dürfen. In der folgenden Grafik haben die Objekte die Attribute Name (vom Datentyp «Zeichenkette»), Jahrgang und Höcker (je vom Datentyp «Ganzzahl»):



Records sind aber keine neuen Konzepte. Diese existieren bereits in sehr alten Programmiersprachen. Dieses Kapitel dient der Repetition und liefert die Einsicht, dass moderne Objekte auf einfach verständlichen Prinzipien aufgebaut sind.

In der Literatur findet man diverse Begriffe für das Konzept der «Records»: Tupel, struct, Strukturen, Datensatz (Datenstruktur), Verbund-Datentypen, ...

Im UML-Diagramm sehen Records wie folgt aus:



13.1 Einführungsbeispiel

Das folgende Beispiel soll den Nutzen von Datenstrukturen aufzeigen. Das Beispiel speichert Geburtstage von vier Kamelen (Pedro, Ali, Aisha und Paola). Ein Geburtstag besteht aus Jahr, Monat und Tag. Das macht also insgesamt 12 Variablen. Zunächst aber einmal ohne Datenstrukturen:

```
int pedro_geb_jahr  = 1985;
int pedro_geb_monat =    4;
int pedro_geb_tag   =   22;

int ali_geb_jahr    = 1998;
int ali_geb_monat   =    6;
int ali_geb_tag     =   22;

int aisha_geb_jahr  = 2005;
int aisha_geb_monat =    6;
int aisha_geb_tag   =   15;

int paola_geb_jahr  = 1995;
int paola_geb_monat =    4;
int paola_geb_tag   =    1;

// alter(int jahr, int monat, int tag) {...}
alter(pedro_geb_jahr, pedro_geb_monat, pedro_geb_tag);
alter(  ali_geb_jahr,   ali_geb_monat,   ali_geb_tag);
alter(aisha_geb_jahr, aisha_geb_monat, aisha_geb_tag);
alter(paola_geb_jahr, paola_geb_monat, pedro_geb_tag);
```

Die obige Variante ist sehr fehleranfällig. Sowohl beim Tippen wie auch beim Kopieren der Codezeilen können sich Fehler einschleichen. Beim Kopieren geschehen Fehler meist dadurch, dass die Variablennamen nicht angepasst werden. (Haben Sie den Fehler in den obigen Zeilen gefunden?)

Geek Tipp 1

Kopierter Code fügt Ihnen und Ihrer Umwelt erheblichen Schaden zu.

Lösungsvorschlag

Nun aber gleich zur Lösung mit Hilfe von Datenstrukturen, um deren Vorteil zu zeigen. Zunächst definieren wir die Datenstruktur `Datum` (in JAVA mit dem Schlüsselwort `class`):

```
class Datum {
  int jahr, monat, tag;
}
```

Die Nutzung dieser Datenstruktur sieht dann wie folgt aus:

```
Datum pedro_geb;
Datum   ali_geb;
Datum aisha_geb;
Datum paola_geb;
/*
  void setzeDaten(Datum d, int jahr, int monat, int tag) {
    d.jahr = jahr; d.monat = monat; d.tag = tag;
  }
*/
setzeDaten(pedro_geb, 1985, 4, 22);
setzeDaten(  ali_geb, 1998, 6, 22);
setzeDaten(aisha_geb, 2005, 6, 15);
setzeDaten(paola_geb, 1995, 4,  1);

alter(pedro_geb);
alter(  ali_geb);
alter(aisha_geb);
alter(paola_geb);
```

Anfänglich denkt man, man könne damit nicht viel sparen (wir sparen gerade einmal 4 Codezeilen). Beachten Sie jedoch den Aufruf der Methoden `setzeDaten()` und `alter()`. Es wird nun übersichtlicher, bei jeder Verwendung Code eingespart und die Anzahl Kopier- bzw. Tippfehler verringert.

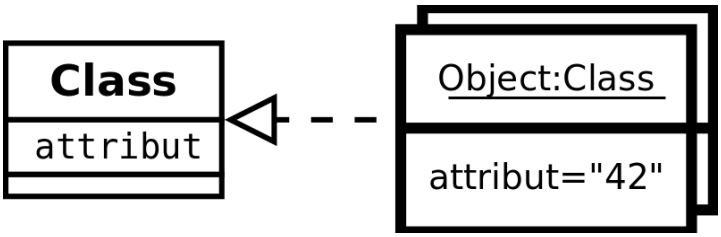
13.2 Datentypen der Attribute

Typisierte Sprachen wie JAVA erzwingen für jedes Attribut – also für jede Eigenschaft – einen Datentypen. JAVA kennt ganze und gebrochene Zahlen, Zeichen und Zeichenketten und den `boolean`, einen Ja/Nein (Wahr/Falsch) Datentypen. Ebenfalls können Referenzen (also Zeiger auf andere Objekte) als Datentypen verwendet werden.

13.3 Konstruktoren: 1. Objekterzeugung

Um Objekte in einem Programm entstehen zu lassen, müssen wir in erster Linie Speicher für die Attributwerte beschaffen. Diese Aufgabe nehmen die Konstruktoren ab. Sie verlangen dem Heap Speicher ab, damit danach mit den Objekten als eigenständige Elemente gearbeitet werden kann.[10]

Des Weiteren haben Konstruktoren keinen beliebigen Return-Wert, denn es wird immer ein neu erstelltes Objekt der entsprechenden Klasse zurückgegeben.



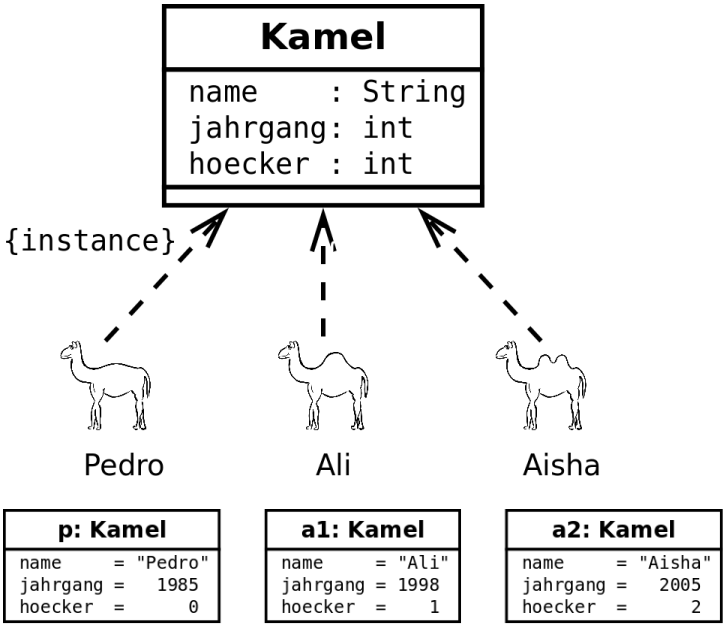
Beispiel 13.1. Konstruktor

```
class Kamel {
  ...
  int jahrgang;

}

...

Kamel k1 = new Kamel();
Kamel k2 = new Kamel();

k1.jahrgang = 2012;
k2.jahrgang = 1998;
```

[10] In der Sprache «C» z. B. geschieht diese Speicherbeschaffung mittels `malloc` (= Memory Allocation).

13.3.1 Beispiel: Objekte erstellen mit Java

Das folgende Beispiel zeigt, wie drei Objekte derselben Klasse *Kamel* erstellt werden.



Zunächst muss die Klasse in JAVA geschrieben werden, welche das Erstellen der Objekte erst ermöglicht. Das erste JAVA-Codebeispiel zeigt, wie obige Klassen im Programm umgesetzt werden könnten.

Das zweite Codebeispiel zeigt das Erstellen von Objekten in JAVA mit dem `new()`-Operator.[11]

```
/**
 * Diese Klasse beschreibt die Struktur und das Verhalten
 * eines jeden Kamels
 */
public class Kamel {
  String name    ;
  int    jahrgang;
  int    hoecker ;
}
```

```
// Hauptprogramm:

// Erzeuge drei gleichartige Objekte/Instanzen/Exemplare
// der Klasse Kamel:

Kamel p  = new Kamel();
Kamel a1 = new Kamel();
Kamel a2 = new Kamel();
...
```

[11] Bem.: Der Einfachheit halber habe ich das Zuweisen an die Attribute `name`, `jahrgang` und `hoecker` nicht abgedruckt.

Bemerkung 13.1. Wird kein Konstruktor explizit definiert und danach mit `new Kamel()` ein Objekt erstellt, so wird hier lediglich Speicher für die Attribute beschafft, aber keine weitere Initialisierung vorgenommen. Dies entspricht dem Erstellen eines «Records» in nicht objektorientierten Sprachen.

Bemerkung 13.2. «Instance»

Das Wort `{instance}` in obiger Zeichnung zeigt von den Exemplaren auf ihre Klasse. Instanz wurde aus dem Englischen falsch übersetzt. Ein deutscher korrekter Begriff wäre z. B. **Exemplar** oder eben ein Objekt. Wir sprechen auch in dieser Grafik von einer «ist ein»-Beziehung zwischen der Instanz (dem Objekt) und der zugehörigen Klasse.

13.4 Der «.»-Operator (1. Teil)

Auf Attribute von Objekten wird mit dem Punkt «.» - Operator zugegriffen. Der Punkt-Operator steht zwischen der Referenz auf ein Objekt und dessen Attribut.[12]

```
public class Person {
  String name, vorname;
}
```

Diese kann nun wie folgt verwendet werden:

```
public class Test {
  void printName(Person p) {
      System.out.println(p.name);
  }

  void haupt() {
    Person p1  = new Person();
    p1.name    = "Muster";
    p1.vorname = "Max";

    Person p2  = new Person();
    p2.name    = "Molina";
    p2.vorname = "Martina";

    printName(p1);
    printName(p2);
  }
}
```

[12]Der «.»-Operator für Methoden (`p1.printName();`) wird eingehend bei der Interaktion behandelt (s. Kap. 16.3 auf Seite 51).

13.5 Argumente und Parameter

Für viele Anwendungen werden den Subroutinen Argumente mitgegeben. Für «primitive» Datentypen (wie `int` oder `float`) ist dies bereits bekannt und soll hier deshalb nicht im Detail behandelt werden. Übergeben wir den Subroutinen Objekte, so ist es gut zu wissen, ob sich deren innerer Zustand beim Aufruf verändert. Wir unterscheiden «Call by Reference» und «Call by Value».

«Call by Reference» vs. «Call by Value»

- Es handelt sich bei einem Funktionsaufruf entweder um eine **Kopie der Daten** (= «**Call by Value**»), dabei hat die Subroutine keinen Zugriff auf das Original und kann lediglich die Kopie verändern (in JAVA geschieht dies automatisch mit allen Parametern), oder
- es handelt sich um eine **Kopie der Referenzen**. Eine Referenz ist ein Zeiger, Pointer, mit anderen Worten einfach die Speicheradresse auf die Datenstruktur. Dies nennt man «**Call by Reference**». JAVA kennt bei den Parametern **kein** «Call by Reference». Die Referenzvariablen werden kopiert und somit kann die Subroutine lediglich die Kopie der Referenz verändern. Da Referenzvariable aber auf Objekte zeigen, so wird dadurch dieses **referenzierte Objekt** tatsächlich via «Call by Reference» übergeben.

Auf der folgenden Seite sind drei Beispiele, um den Unterschied der Aufruf-Arten «Call by Reference», «Call by Value» zu demonstrieren:

Beispiel - «Call by Value» :

```
void fct(int x) {
  x = 4;
  print(x);
}
//Aufruf (Hauptprogramm):
{
  int a = 3;
  print(a); // -->   3   wird ausgegeben
  fct(a);   // --> **4** wird ausgegeben
  print(a); // --> **3** wird ausgegeben
}
```

Beispiel - «Call by Reference» [13]:

```
void fct(Kamel p) {
  p.name = "Paola";
  print(p.name);
}
//Aufruf (Hauptprogramm):
{
  Kamel q;
  q = kameldatenEinlesen();
  q.name = "Ali";
  print(q.name); // -->   Ali      wird ausgegeben
  fct(q);        // --> **Paola** wird ausgegeben
  print(q.name); // --> **Paola** wird ausgegeben
}
```

Beachten Sie den Unterschied in der Ausgabe. `a` und `x` sind zwei unabhängige Variablen mit je ihrem eigenen Inhalt. Ebenso sind `p` und `q` zwei unabhängige (Referenz)variablen, die hier jedoch beide auf ein und dasselbe Objekt **zeigen**.

JAVA kennt ausschließlich (für alle Datentypen, inklusive Referenzvariable) das «*Call by Value*»-Verfahren: Der Wert des Ausdrucks (hier 3, denn `a = 3`) wird dem aufgerufenen Parameter (hier `x`) übergeben. Die ursprüngliche Variable wird dabei nie verändert.

Für die (durch Referenzvariable) referenzierten Objekte kennt JAVA daher nur die oben beschriebene «*Call by Reference*»-Strategie.[14]

[13] In JAVA handelt es sich genau genommen um «Call Reference-Variable by Value, Call Object by Reference», denn JAVA kennt ausschließlich «Call by Value».

[14] In Programmiersprachen wie c, c++ u. ä. kann auch die Adresse einer Variablen übergeben werden. Somit ist dort auch für Variable ein echtes «*Call by Reference*» möglich.

Wollen wir eine Datenstruktur (Objekt) via «*Call by Value*» übergeben, so müssen wir in JAVA dieses Objekt selber kopieren (`clone()`):

Beispiel «Call Object by Value» :

```
void fct(Kamel p1) {
  p1 = p1.clone(); // Kopie der Daten
  // p1 zeigt nun auf eine Kopie des Originals.
  p1.name = "Paola"; // Arbeiten auf der Kopie
  print(p1.name);
}
//Aufruf (Hauptprogramm):
{
  Kamel q;
  q = kameldatenEinlesen();
  q.name = "Ali";
  print(q.name);  // -->   Ali    wird ausgegeben
  fct(q);         // -->   Paola  wird ausgegeben
  print(q.name);  // --> **Ali** wird ausgegeben
}
```

13.6 Garbage Collection

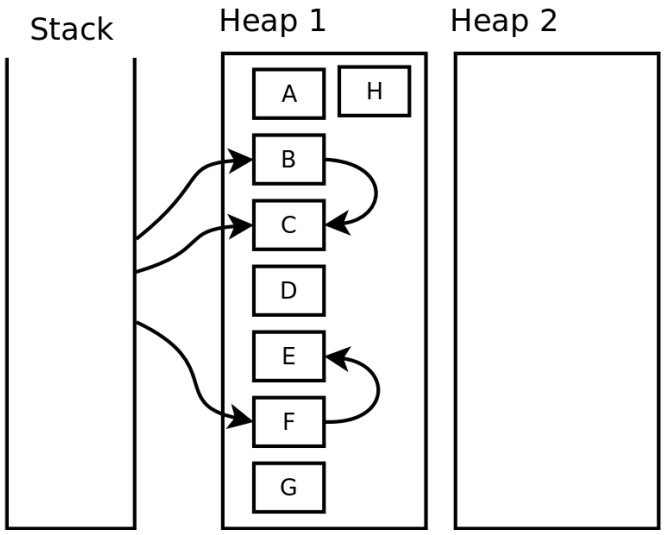
Die elektronische Müllabfuhr

Vor allem in älteren Programmiersprachen und teilweise in Echtzeitsystemen muss der durch ein Objekt allozierte Speicher nach der Verwendung durch die Applikationsentwickler rasch wieder freigegeben werden, damit andere Prozesse oder Programmteile dort wieder ihre Variablen anlegen können. In der Programmiersprache **C** z. B. geschieht das mit der Funktion `free`. Wenn unbenutzter Speicher nicht freigegeben wird – und dies wird leider häufig vergessen – kann ein Heap-Overflow auftreten, obschon noch genügend Speicher vorhanden wäre. Das ist auch bekannt unter dem Begriff *Memory Leak*.

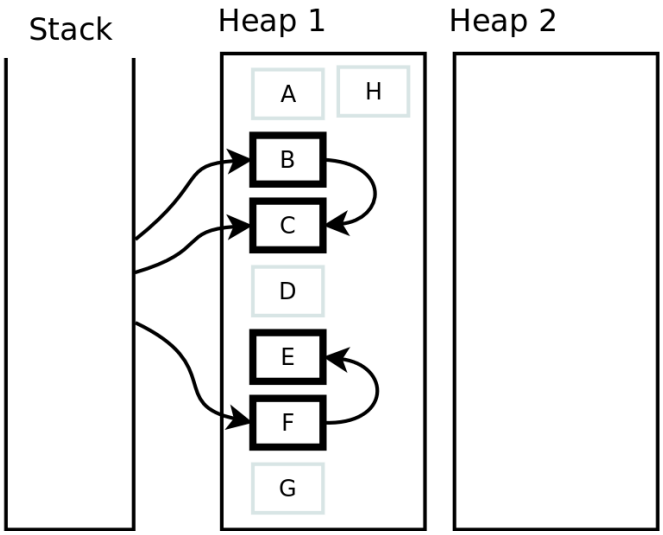
Moderne Sprachen kennen einen sog. **Garbage Collector**, der unbenutzte Objekte selbstständig wieder aus dem Speicher entfernt.

Vereinfacht könnten wir uns dieses Verhalten z. B. wie folgt vorstellen:

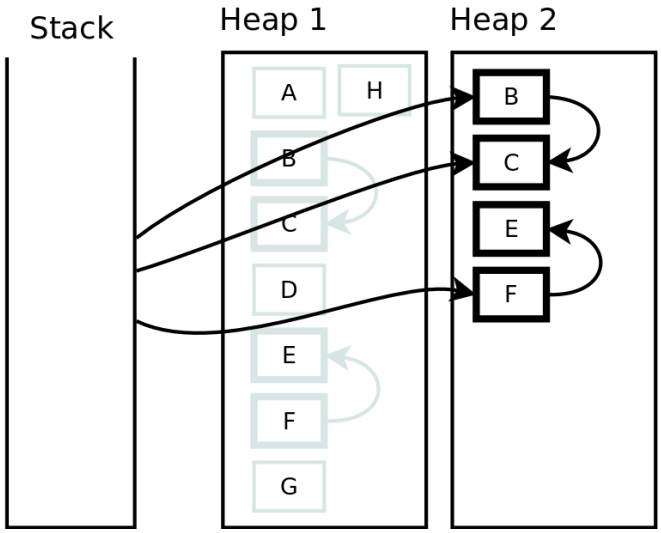
1. Initialisierung Alle Referenzen aus dem Stack (= Ausführungsstapel) und innerhalb des Heaps (= Objekt-Speicherbereich) werden gesucht (Referenzpfeile in folgender Grafik):



2. Markieren Alle verwendeten Objekte werden (zusammen mit ihren Referenzen) markiert:


Stack
Heap 1
Heap 2
A
H
B
C
D
E
F
G

3. Leeren Alle verwendeten Objekte werden in den 2. Heap kopiert und die Referenzen aktualisiert. Der Heap 1 wird «geleert» und steht für den nächsten Garbage Collector Schritt wieder zur Verfügung.



13.7 Aufgaben

Aufgabe 13.1 «Drei Kamele»

Schreiben Sie eine Subroutine `einlesenNeuesKamel()` welche die Daten für ein Kamel (Name, Jahrgang, Hoeckeranzahl) von der Tastatur einliest und eine Kamel-Struktur abfüllt. Prototyp:

```
Kamel einlesenNeuesKamel();
```

Schreiben Sie danach eine Funktion zur Ausgabe der Kameldetais. Prototyp:

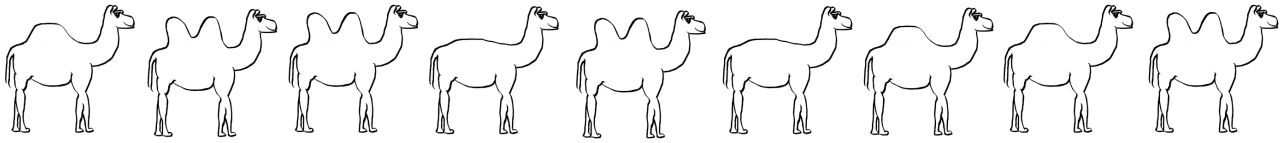
```
void kamelDetailsAusgeben(Kamel k);
```

Zu guter Letzt fügen Sie die beiden Methoden zu einem Hauptprogramm zusammen indem Sie zunächst drei Kamele einlesen (`k1` , `k2` und `k3`) und deren Details nach dem Einlesen der letzten Kameldaten wieder ausgeben.

Weitere Aufgaben zu Records finden Sie im Buch «Programmieren lernen» [GFG11] und im Internet unter `http://www.programmieraufgaben.ch` .

14 | Datensammlungen

«*Delegiere an eins, zwei, drei, ... viele*»



Oft müssen wir sehr viele gleichartige Daten zusammen abspeichern und mit geeigneten Suchalgorithmen wieder finden können. Wir kennen bereits die Arrays ([GFG11] Kapitel 6), mit denen wir die meisten der folgenden Konzepte auch simulieren können (speichern, suchen, sortieren, aufzählen, ...).

Betrachten wir dazu den folgenden Code:

```
Point[] points = new Point[100];
int i = 0;
while(this.hasMorePoints()) {
  points[i] = this.nextPoint();
  i = i + 1;
} // i >= **100** !?
```

Arrays haben einige, teilweise beträchtliche Nachteile in speziellen Anwendungen, sodass es von Vorteil ist, die folgenden abstrakten Datensammlungstypen (Collections wie Listen, Maps, Stapel, Bäume, ...) genauer zu kennen.

Nachteil von Feldern (Arrays[])	**gelöst in der Java-Klasse**
Beschränktheit: Verändern der Größe nach erstmaliger Initialisierung ist nicht möglich.	alle folgenden Konstruktionen. Meist wird die `ArrayList` verwendet.
Als Index sind nur Zahlen möglich.	`Maps`, insbesondere `HashMaps` erlauben auch andere Objekte als Indizes oder (Such)schlüssel.
Suchen kann nur bei sortiertem Array binär erfolgen. Ansonsten ist eine lineare Suche notwendig.	Binäre Bäume (`Tree`) bzw. (Hash)`Maps` erlauben ein sortiertes Einfügen bzw. ein Einfügen nach Schlüsselobjekt.
Einfügen in der Mitte und Löschen aus der Mitte sind umständlich.	Mit (Linked)`Lists` kann ein Objekt aus der Mitte sehr performant gelöscht (bzw. in die Mitte eingefügt) werden.
Einfügen und Löschen am Anfang (bei kleinen Indizes) ist rechenaufwändig.	`Queue`s und `Stack`s lösen dieses Problem meistens elegant mit einer `List` oder mit einem Ringpuffer.
Gleich sortiert einfügen ist umständlich.	Bäume (`Tree`) fügen Objekte direkt binär sortiert ein.

Geek Tipp 2

Verwenden Sie wenn immer möglich eine *Collection* anstelle eines Arrays. Im Zweifelsfalle ist die `ArrayList` zweckmäßig.

14.1 Initialisieren von Collections

Arrays können auf einfache Art initialisiert werden:

```
int[]          iarr = {4, 6, 9};
```

Da wir aber Arrays oft vermeiden wollen, bietet sich für Collections folgende Initialisierung an:

```
List<Integer> list = Arrays.asList(4, 6, 9);
```

... oder so ...

```
List<Integer> list = new ArrayList<Integer>();
list.add(4);
list.add(6);
list.add(9);
// ...
// z. B. Ausgabe
for(int z: list) {
  System.out.println(z);
}
```

14.2 Aufgaben zu Datensammlungen

Aufgabe 14.1 «Sortieren»

Schreiben Sie ein Programm, das vom Anwender so lange Zeichenketten (Strings) entgegen nimmt, bis das Wort ENDE eingegeben wird. Danach soll das Programm alle Zeichenketten in alphabetischer Reihenfolge wieder ausgeben (das Wort ENDE soll im Output natürlich nicht vorkommen).

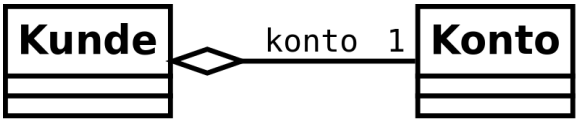
Aufgabe 14.2 «Viele Kamele (eine erste Karawane)»

Wie in Aufgabe «Drei Kamele» (S. 33) sollen wieder Kamele eingelesen werden. Diesmal jedoch nicht drei sondern eine beliebig lange, nicht leere Liste. Nach dem Einlesen jedes Kamels (Name, Jahrgang, Höcker) soll gefragt werden, ob noch weitere Kamele einzulesen sind. Nach dem letzten eingelesenen Kamel sollen alle Daten in der Reihenfolge ihrer Erfassung wieder ausgegeben werden.

Die Hilfsfunktionen (`einlesenNeuesKamel()` und `kamelDetailsAusgeben()`) aus der Aufgabe «Drei Kamele» können Sie natürlich wieder verwenden.

15 | Aggregation

«Datenstrukturen beliebig zusammensetzen»



Bei der Aggregation (auch Komposition genannt) werden zwei Objekte derart miteinander «verschmolzen», sodass das eine Objekt zu einem Teil des Anderen wird. Wir sprechen von einer

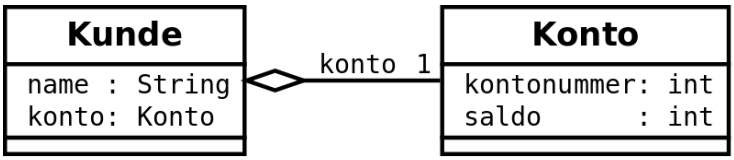
Hat-Ein-Beziehung

Solche Beziehungen kennen wir aus dem täglichen Leben zuhauf.[15]

- Kunde hat Konto
- Musiksammlung hat Musikstücke
- Auto hat Motor
- Lager hat Waren
- Mitglied hat Geburtsdatum

[15] Eingeschlossen sind natürlich Beziehungen zu mehreren Objekten: «Hat-Zwei», «Hat-Mehrere».

Beispiel zur Aggregation: «Kunde hat Konto»



Betrachten wir zunächst die beiden Klassen für sich:

```
class Kunde {                          class Konto {
  String name;                           int kontonummer;
  ...                                    int saldo;
  ...                                    ...
}                                      }
```

Durch folgende Konstruktion hat sich ein Kunde nun auch die Attribute des Kontos zu eigen gemacht.

```
class Kunde {                          class Konto {
  String name;                           int kontonummer;
  Konto  konto;                          int saldo;
  ...                                    ...
}                                      }
```

Referenzattribut Beim Attribut `konto` in der Klasse `Kunde` handelt es sich um eine **Referenzvariable**.

Auf die Kontodaten wird mit dem Punkt-Operator zugegriffen:

```
Kunde ku;
Konto ko;
ku                    = new Kunde();
ko                    = new Konto();
ku.name               = "Muster";
ku.konto              = ko;
ku.konto.saldo        = 2500;
ku.konto.kontonummer  = 223324;
```

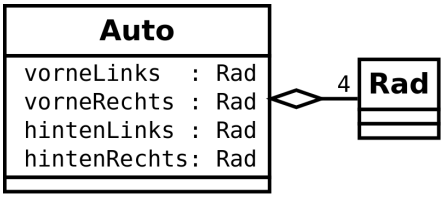
Geek Tipp 3

Benennen Sie das Attribut in einer Aggregation auch gleich wie die aggregierte Klasse – nur klein geschrieben. (Hier `konto`)

15.1 1:n - Beziehungen

15.1.1 Beispiel Auto-Räder

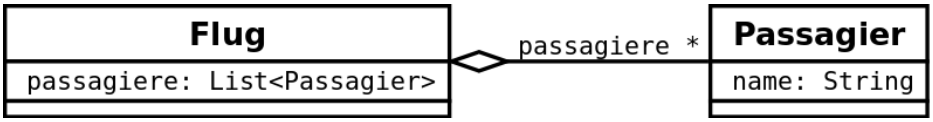
Oft hat ein Objekt viele gleichartige Teile. Ein Auto hat vier Räder, ein Zug hat mehrere Wagen, ein Haus besteht aus mehreren Wohnungen oder eine Karawane aus mehreren Kamelen. Handelt es sich um eine kleine und vorgegebene Anzahl (Auto aus vier Rädern), so kann durchaus für jedes Teil eine eigene Referenzvariable herangezogen werden:




```
class Rad{
}

class Auto{
  Rad vorneLinks;
  Rad vorneRechts;
  Rad hintenLinks;
  Rad hintenRechts;
}
```

15.1.2 Beispiel Flug-Passagiere

Handelt es sich hingegen um sehr viele gleichartige Teile oder ist die Anzahl dieser gleichartigen Teile nicht bekannt, so bieten sich Listen an. In JAVA wird dies meist mit einer `ArrayList` oder einer `LinkedList` implementiert:



Bemerkung 15.1. Hierbei (und auch im Bild auf Seite 40) handelt es sich um eine redundante Notation, die lediglich aufzeigt, wie diese Beziehung umgesetzt wird. In der Praxis wird **entweder** das Attribut angegeben **oder** der Pfeil beschriftet. Beides zu tun, ist unüblich.

Passagiere werden diesem Flug nun mittels den List-Funktionen angefügt und verarbeitet:

```
class Passagier {
  String name;
}

class Flug{
  List<Passagier> passagiere;
}
```

Anwendungsbeispiel:

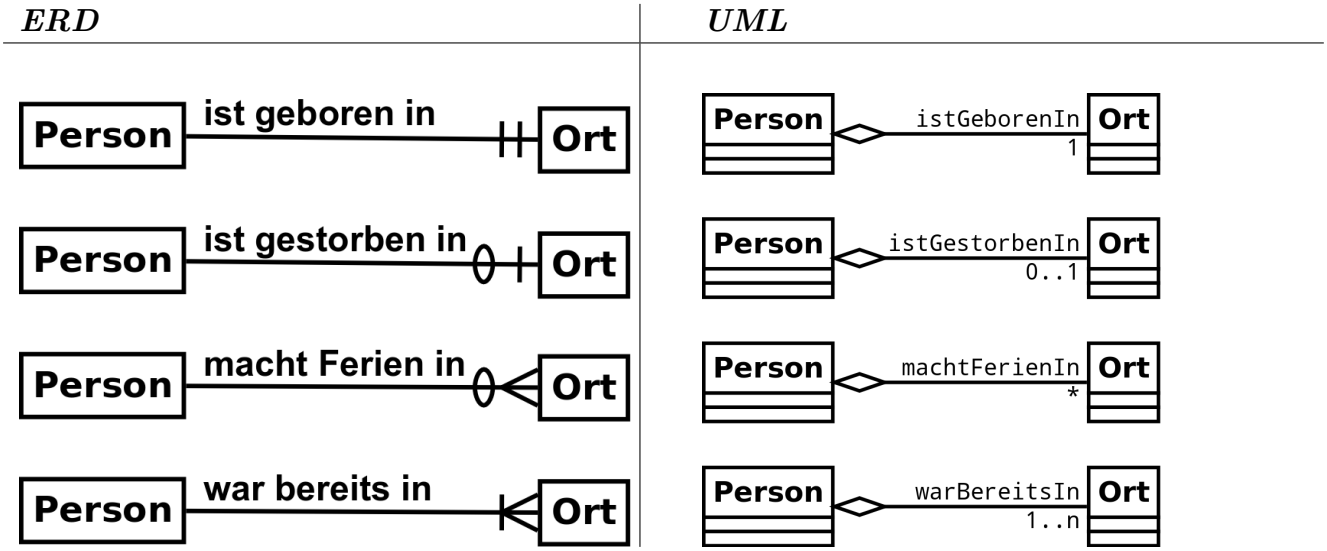
```
void passagierListeAusgeben(Flug flug) {
  for(Passagier p : flug.passagiere) {
    System.out.println(p.name);
  }
}

void main() {
  Flug flug       = new Flug();
  flug.passagiere = new ArrayList<Passagier>();

  Passagier p1    = new Passagier();
  Passagier p2    = new Passagier();
  p1.name         = "Gressly Freimann";
  p2.name         = "Guggisberg"      ;

  flug.passagiere.add(p1);
  flug.passagiere.add(p2);

  passagierListeAusgeben(flug);
}
```

Geek Tipp 4

Benennen Sie das 1:n-Attribut wie die aggregierte Klasse – nur diesmal in Mehrzahl. (Hier `passagiere`)

15.2 Zusammenfassung UML (Vergleich mit ERD)

Die beiden folgenden Grafiken zeigen die entsprechenden Notationen der Beziehung zwischen Entitäten; links die ERD (Entity Relation Diagram)-Notation[16], rechts die UML (Unified Modelling Language)-Notation[17].



Im objektorientierten Vorgehen wird meist die UML (rechts) verwendet. Die ausgesparten Zeilen in den Kästchen erlauben das Anfügen von Attributen und Funktionalitäten.

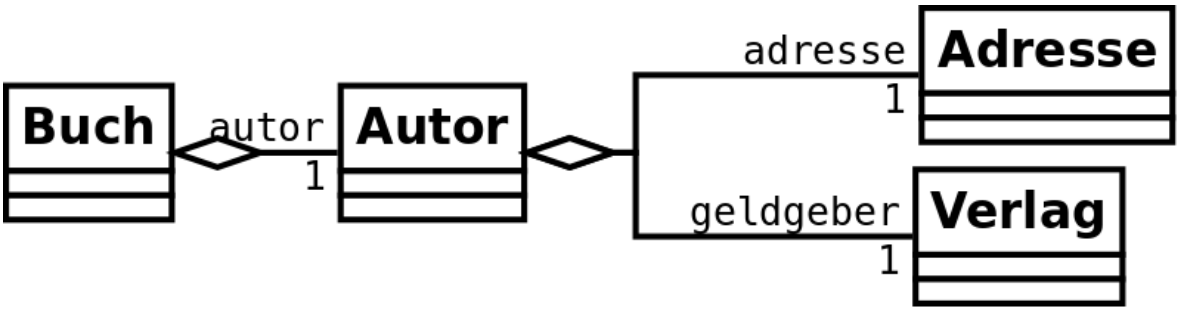
[16] Hierbei handelt es sich um die *Krähenfußnotation* nach Martin, Bachmann und Odell.
[17] Die UML Notation wurde von den *Amigos* Booch, Jacobson und Rumbaugh vorgeschlagen.

15.3 Aggregationsketten

«*Strukturen bestehend aus Strukturen bestehend aus Strukturen ...*»
Begriffe: Hierarchie, Rekursion
Datenstrukturen können hierarchisch aus weiteren Datenstrukturen bestehen. Beispiele:

- Computer - Motherboard - CPU
- Haus - Liftschacht - Liftsteuerung
- Buch - Autor - Adresse

Hier eine detailliertere Ausführung zunächst als UML, danach als JAVA-Code:




```
class Adresse {
  String str;
  String ort;
  ...
}

class Verlag {
  String name;
  ...
}

class Autor {
  Adresse adresse  ;
  Verlag  geldgeber;
}

class Buch {
  Autor autor;
  ...
}
```

15.4 Aggregationsketten und 1:n - Beziehungen

In der Praxis kommen meist beide Beziehungsarten 1:1 und 1:n gleichzeitig und mehrfach vor:

- So besteht z. B. ein Schachspiel aus Brett und Figuren. Ein Schachbrett besteht aus weißen und schwarzen Feldern. Felder haben eine Farbe und eine Koordinatenangabe.
- Eine Bestellung besteht aus Artikeln. Ein Artikel hat eine Bezeichnung, einen Preis, einen Lieferanten etc. Ein Lieferant hat eine Adresse, einen Namen, ...

Obiges Beispieldiagramm hier als JAVA-Programmcode:

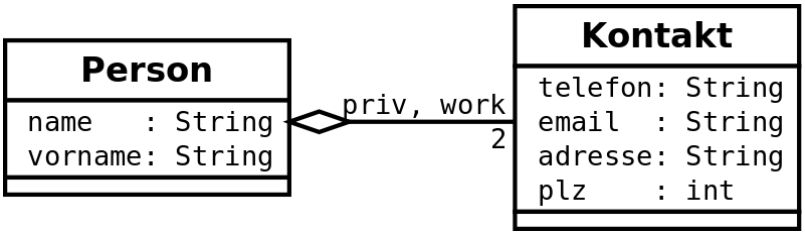
```
class Bestellung {
  List<Artikel> artikel;
  ...
}

class Artikel {
  Lieferant lieferant;
  ...
}

class Lieferant {
  ...
}
```

15.5 Aufgaben

Aufgabe 15.1 «Adressbuch»

Schreiben Sie ein Programm, das die folgenden Datenstrukturen enthält (s. Grafik). Schreiben Sie danach ein Hauptprogramm, das drei Personen mit je zwei Kontaktdaten abfüllt, und diese anschließend auf der Konsole wieder ausgibt.

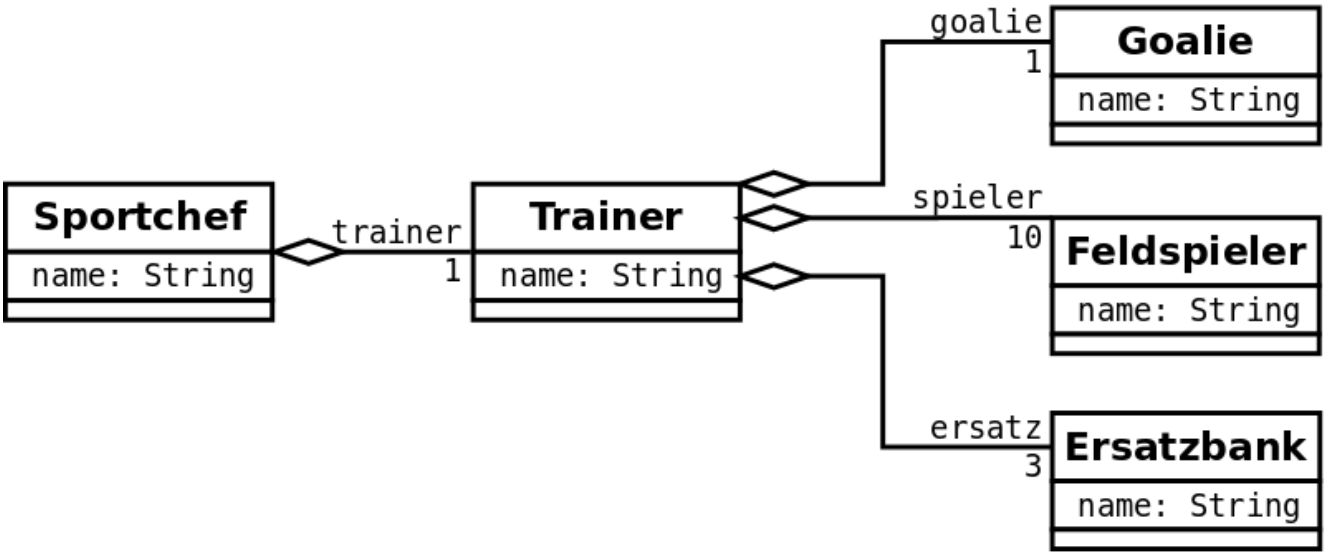
Beachten Sie auch, dass neben den explizit angegebenen Attributen innerhalb der Klassen noch weitere vorkommen müssen, um die Aufgabe zu lösen.



Aufgabe 15.2 «Sportchef»

Implementieren Sie alle Klassen des folgenden UML[18]-Diagramms. Schreiben Sie danach ein Hauptprogramm, das eine vollständige Mannschaft mit Trainer und Sportchef instanziert. Programmieren Sie eine `print()`-Subroutine, welche die komplette Mannschaft auf der Konsole ausgibt.

Zusatzaufgabe: Erzeugen Sie eine Schleife, in der die Namen aller Spieler erfasst werden können.



Aufgabe 15.3 «Flug und Passagiere»

Implementieren Sie das Flug-Passagier-Beispiel auf Seite 42

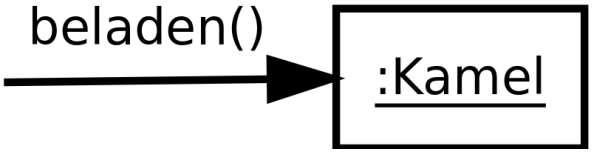
[18] UML = unified modelling language

16 | Methoden (Interaktion)

«*Subroutinen neu aufgerollt*»

Methoden erweitern die klassischen Subroutinen um eine interessante Sichtweise. Technisch ist die Erfindung von «Methoden» (im Gegensatz zu Modulen bestehend aus Funktionen) keine Höchstleistung in der Softwareentwicklung. Aus Sicht der Systemspezifikation und auch für eine einfachere Programmierung darf man durchaus von einer kleinen Revolution sprechen.

Vergleichen wir die beiden Codestücke, so wird uns klar, welche neuen Möglichkeiten mit Methoden offen stehen.



Klassisch:

```
beladen(kamel, ware);
```

Objektorientiert:

```
kamel.beladen(ware);
```

Bemerkung 16.1. Die objektorientierten Sprachen erlauben beide Notationen, wohingegen die klassische strukturierte Programmierung lediglich den ersten Code versteht.

16.1 Objekt, Subjekt, Prädikat

Wir kommen der natürlichen Sprache mit den Methoden schon näher:

«Peter kauft Käse»

Das Objekt Käse wird hier vom Subjekt Peter gekauft (Prädikat).

In OO-Sprachen könnte das nun so aussehen:

```
peter.kauf("Kaese");
subjekt.praedikat(objekte);
```

Diese am Objekt orientierte Sichtweise wird häufig als **der** Fortschritt in der Programmierung angesehen!

16.2 Innerer Zustand

Wir haben bereits gesehen, dass der innere Zustand von Objekten über die Attribute verändert werden kann. Im Hinblick auf die Kapselung des Zustandes zusammen mit der Funktionalität ist es jedoch unüblich, auf Attribute von außen zuzugreifen und diese direkt zu verändern. Hierzu dienen Methoden.

Um mit einem Objekt zu interagieren, wird man in den meisten Fällen eine Subroutine aufrufen, welche die Attributwerte verändert. Diese Art der Kommunikation wird auch Nachrichtenaustausch (oder «message passing») und diese Subroutinen werden **Methoden** genannt.

Das neue an der OO-Modellierung (Analyse **und im** Design) – gegenüber der rein strukturierten Vorgehensweise – ist es, die Objekte **zusammen mit ihren Interaktionen** als Einheit aufzufassen: Die Struktur (und das Diagramm) beim Übergang von der Analyse zum Design bleibt weitgehend erhalten (ein sog. **Strukturbruch** entfällt somit).

Objekte interagieren miteinander. Die inneren Zustände von Objekten sind normalerweise von außen nicht sichtbar. Objekte stellen aber Methoden (Funktionalitäten, Prozeduren, Routinen) zur Verfügung, mit denen der innere Zustand der Objekte betrachtet bzw. verändert werden kann.

Beispiel 16.1. Konto: `einzahlen()`, `abheben()`, `kontostandAbfragen()`, `zinsBerechnen()`, ...
Clubmitglied: `adresseÄndern()`, `clubBeitragFestsetzen()`, `funktionÄndern()`, ...

Methoden beschreiben somit das Verhalten der Objekte.

Geek Tipp 5

Verwenden Sie bei zusammengesetzten Wörtern für Attribute und Methoden die sog. «Kamelnotation». Dabei werden die Wörter ohne Zwischenraum zusammengeschrieben und jeder Wortanfang beginnt mit einem Großbuchstaben. Der erste Buchstabe der Namens ist dabei für Attribute und Methoden klein, jedoch für Klassennamen groß geschrieben.

16.3 Der «.»-Operator (2. Teil)

Auf Methoden wird ebenfalls[19] mit dem Punkt «.»–Operator zugegriffen.

Gegeben ist die folgende Klasse:

```
public class Person {
  String name, vorname;

  void printName() {
      System.out.println(name);
  }
}
```

Verwendet werden kann diese nun wie folgt:

```
public class Test {
  void haupt() {
    Person p1  = new Person();
    p1.name    = "Muster";
    p1.vorname = "Max";

    Person p2  = new Person();
    p2.name    = "Molina";
    p2.vorname = "Martina";

    p1.printName();
    p2.printName();
  }
}
```

Vergleichen Sie obigen Code nun mit dem Beispiel auf Seite 27. Die Subroutine `printName()` enthält nun keinen Parameter mehr, und somit ist beim Aufruf auch kein Attribut mitzugeben. Diese Art des Subroutinen-Aufrufs ist am Objekt orientiert: Solche Subroutinen nennen wir **Methoden**.

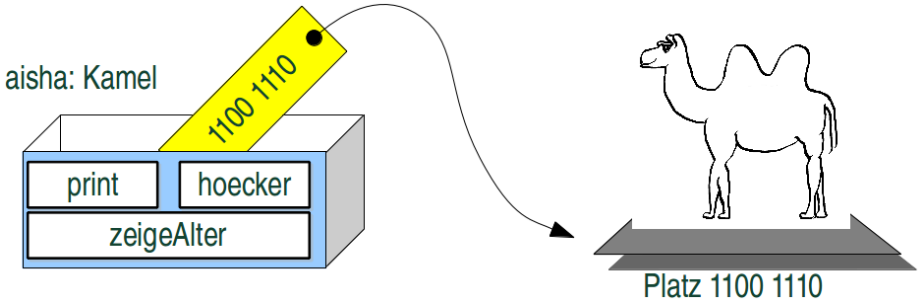
[19] Der «.»-Operator hat in JAVA verschiedene Bedeutungen: Wir kennen bereits das Zugreifen auf Attribute (s. Kap. 13.4 auf Seite 27); aber auch die Paket-Hierarchie (z. B. `import java.lang.String`) verwendet Punkte.

16.4 Referenzvariable

Um das nächste Kapitel (16.5) besser verstehen zu können, muss ich hier etwas «ausholen».

Referenzvariable sind *Zeiger* (*Pointer*) auf Objekte. Man kann sich diese wie eine Fernsteuerung vorstellen, um auf dem Objekt (auf der Datenstruktur) Änderungen vorzunehmen. Im Gegensatz zu primitiven Datentypen (Standardtypen wie `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`), bei denen die Variable direkt den Wert enthält, enthalten die Referenzen lediglich die Adresse des Objektes.

Betrachten wir dazu die Referenzvariable `aisha` in der folgenden Grafik:



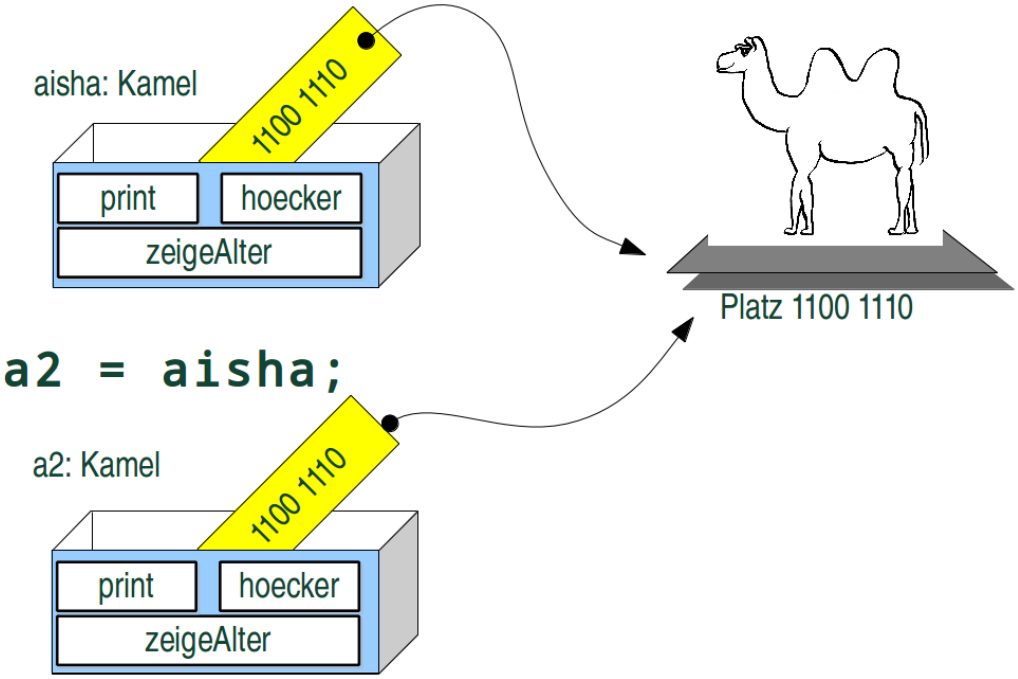
Bildlegende: *Eine Variable kann man sich wie eine Kiste vorstellen. Standardtypen wie ganze Zahlen (`int`) werden einfach in diese Kiste gepackt. Bei Referenzvariablen wird einfach die Speicheradresse des Objektes abgespeichert. So können mehrere Variablen auf dasselbe Objekt deuten.*

16.4.1 Mehrere Referenzen auf dasselbe Objekt

Betrachten wir nun den folgenden Code:

```
Kamel aisha;
Kamel a2   ;

aisha = new Kamel();

a2    = aisha;
```

Wird eine Referenzvariable einer anderen Referenzvariable zugewiesen, so wird das Objekt nicht verdoppelt. Aber beide Referenzvariablen zeigen nun auf dasselbe Objekt. Diese können also beide nun dasselbe Objekt via Methoden fernsteuern:



16.4.2 Aufruf von Funktionen mit Referenzen als Parameter

Den Funktionen können wir Attributwerte via Parameter mitgeben. Hierzu gibt es grundsätzlich drei Arten von Parametern:

Typ	Beschreibung
Primitiv	Dies sind Datentypen, die normalerweise in einem Register Platz finden: Zahlen, Zeichen und boolesche Werte.
Datenstruktur (oder Array)	Zusammengesetzte Datentypen oder Arrays mit einem Basistypen können in JAVA lediglich via Referenzvariable den Methoden übergeben werden. Sprachen wie C können beim Funktionsaufruf Kopien der kompletten Datenstruktur anlegen.
Referenz (Pointer)	Zeiger auf Datenstrukturen. Dies wird technisch meist durch Adressen oder Handles (= Adresse des Zeigers) implementiert. In der Variable wird die Speicherstelle der Datenstruktur festgehalten.

16.5 Der `this`-Pointer

Der `this`-Pointer in JAVA[20] ist eine temporäre Variable, die nur während des Aufrufs einer Methode lebt. Das aufrufende Programm gibt der Methode einen (versteckten) Parameter mit, der innerhalb der aufgerufenen Methode den Namen «this» trägt. Es handelt sich dabei um das Objekt, das beim Aufruf links vom Dereferenzierungspunkt (oder -pfeil) steht. Ein Beispiel zeigt den Sachverhalt am besten:

Statischer Code (**rein** strukturierte Programmierung)	**Dynamischer** Code (**Objektorientierte**, strukturierte Programmierung)
Deklaration / Definition	

`public class Person {`
`  String name, vorname;`
 	
`  static`
`  void printName(Person thisP) {`
`  print(thisP.name);`
`  }`
`}`
	`public class Person {`
	`  String name, vorname;`
	
	`  /* dynamisch */`
	`  void printName() {`
	`  print(this.name);`
	`  }`
	`}`

Aufruf	

`void callerStruct() {`
`  Person p1 = new Person();`
`  Person p2 = new Person();`
`  p1.name  = "Meier";`
`  p2.name  = "Keller";`
`  printName(p1);`
`  printName(p2);`
`}`
	`void callerOO() {`
	`  Person p1 = new Person();`
	`  Person p2 = new Person();`
	`  p1.name  = "Meier";`
	`  p2.name  = "Keller";`
	`  p1.printName();`
	`  p2.printName();`
	`}`

[20] In anderen Sprachen `self`, `me` oder ähnlich genannt.

OO–Sprachen **erweitern** die herkömmlichen Konzepte.[21] Somit funktionieren in JAVA **beide** obigen Vorgehensweisen, wohingegen z. B. die Sprache C nur den statischen Code (links) unterstützt.

Was tut JAVA? Im **dynamischen** Fall *versteckt* JAVA ein Attribut namens `this` in der Parameterliste. Der Aufruf `p1.printName()` übergibt die Referenz `p1` der Methode `printName()` im nicht sichtbaren Parameter `this`. Dieser Parameter muss in der Methodendefinition übrigens auch nicht mit angegeben werden. Sollten wir also `print(name)` anstelle von `print(this.name)` schreiben, so merkt das der JAVA-Compiler und fügt auch hier die versteckte Referenz wieder ein:

```
// Definition
void printName(/* Person this */) {
  print(/* this . */ name);
}
```

[21] Wir dürfen nicht vergessen, dass JAVA zwar eine OO-Sprache, jedoch hochgradig strukturiert ist: JAVA implementiert alle Konzepte strukturierter Programmiersprachen! Somit ist JAVA zusammen mit manchen anderen OO-Sprachen auch eine strukturierte Sprache!

16.6 Funktionsbibliotheken

Aus der klassischen Programmierung kennen wir Bibliotheken (Libraries) von Funktionen. Zum Beispiel kennt ein Astronom viele Subroutinen zur Zeitumrechnung. All diese Funktionen hat er bisher in einem Modul zusammengefasst, bindet diese Bibliothek bei Bedarf ein und verwendet die Methoden. Wenn er Korrekturen oder Geschwindigkeitsverbesserungen (Performance) an den Funktionen vornimmt, so werden alle Programme verbessert, welche diese Bibliotheken nutzen.

Solche Funktionsbibliotheken sind in objektorientierten Sprachen weiterhin möglich. Am einfachsten generieren wir eine Klasse mit all den geforderten Methoden. Der Aufrufer instanziert ein Objekt (`new`) und ruft anschließend die Methoden auf. Eine andere Möglichkeit in JAVA ist das Verwenden von `static`-Methoden. Diese verhalten sich exakt wie herkömmliche Funktionen. Beim Aufruf wird dann nicht eine Objektreferenz dem Funktionsaufruf vorangestellt, sondern der Klassenname:

```
  String hms = AstroRechnung.nachHMSUmrechnen(3.3455312);
```

Bei gegebener Bibliothek:

```
public class AstroRechnung {
  public static String nachHMSUmrechnen(double metrisch) {
    ...;
    return ...;
  }
}
```

PS: Natürlich ist es in JAVA auch möglich, bestehende *Libraries*, die in C oder Assembler geschrieben wurden, einzubinden. Dies detaillierter auszuführen, sprengt jedoch den Rahmen dieser Einführung.

16.7 Konstruktoren: 2. Initialisierung

Oft wollen wir bereits beim Erstellen von Objekten deren Attributwerte festlegen. Wir sprechen dabei von einer **Initialisierung** der Objekte. Dazu ist es nötig, auch eigene Konstruktoren schreiben zu können.

Beispiel 16.2. Konstruktor

```
public Person(String name, String vorname) {
  this.name    = name    ;
  this.vorname = vorname;
}
```

`this` referenziert hier im Konstruktor auf das eben erzeugte Objekt. Dieses Objekt wird dann mittels `return` zurückgegeben, ohne dass wir das im «Funktionsheader» oder mittels explizitem `return` angeben müssen.

Obiger Konstruktor sieht dann beim Erzeugen der Objekte wie folgt aus:

```
Person meier  = new Person("Meier"          , "Hans"              );
Person keller = new Person("Keller"         , "Petra"             );
Person p      = new Person(einlesen("Name"), einlesen("Vorname"));
```

Ein anderes Mal wollen wir, dass beim Erstellen von Objekten ein bestimmter Initialisierungscode jedes Mal ausgeführt wird. So ist es auch möglich, beliebige Code-Sequenzen in den Konstruktor zu verlegen, damit diese nicht nach oder vor jeder Objekterstellung aufgerufen werden müssen:

```
public Person() {
  this.name    = einlesen("Name");
  this.vorname = einlesen("Vorname");
}
```

Ein dritter Nutzen von selbst geschriebenen Konstruktoren sei hier auch noch erwähnt. Wir können Vorbedingungen abklären, bevor das Objekt erstellt wird:

```
public Person(String ahvNummer) throws NumberFormatException {
  this.name    = einlesen("Name");
  this.vorname = einlesen("Vorname");
  checkPruefziffer(ahvNummer);
  this.ahvNummer = ahvNummer;
}
```

16.8 Aufgaben

Aufgabe 16.1 «einfache Subroutine durch Methode ersetzen» Implementieren Sie die folgende Klasse `Kamel` und das darauf folgende Hauptprogramm Main mit den Methoden `top()` und `printKamelDetails()`. `printKamelDetails()` sei zunächst in der Klasse Main ausprogrammiert und nicht in der Klasse `Kamel`.

```
public class Kamel {
  String name;
}
```

Main:

```
void top() {
  Kamel k = new Kamel();
  k.name = "Aisha";
  printKamelDetails(k);
}

void printKamelDetails(Kamel k) {
  System.out.println("Kamel: " + k.name);
}
```

Verschieben Sie anschließend diese Subroutine (`printKamelDetails()`) aus dem aufrufenden Hauptprogramm in die entsprechende Klasse (hier `Kamel`) und rufen Sie diese neue Methode nun wie folgt auf:

```
k.printKamelDetails();
```

Beachten Sie, dass in diesem Beispiel die `Kamel`-Referenz sehr wohl weiterhin der Methode `printKamelDetails` mitgegeben wird; einzig nicht mehr im expliziten formalen Parameter, sondern als versteckte Referenzvariable mit dem Namen `this`.

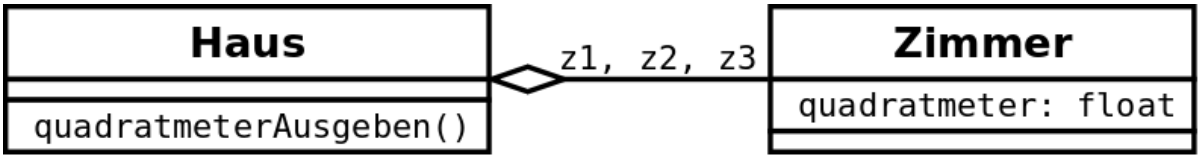
Aufgabe 16.2 «Konstruktor mit Parameter» Erstellen Sie in der Klasse `Kamel` einen Konstruktor (`Kamel(String name)`) und erzeugen Sie einige Kamel-Objekte:

```
Kamel k1 = new Kamel("Aisha");
Kamel k2 = new Kamel("Ali"  );
Kamel k3 = new Kamel("Paola");
```

Zusatz: Rufen Sie für alle Kamele dann die Methode `printKamelDetails()` aus Aufgabe 16.1 auf.

Aufgabe 16.3 «Methode benutzt Aggregation»

Schreiben Sie die beiden folgenden Klassen.

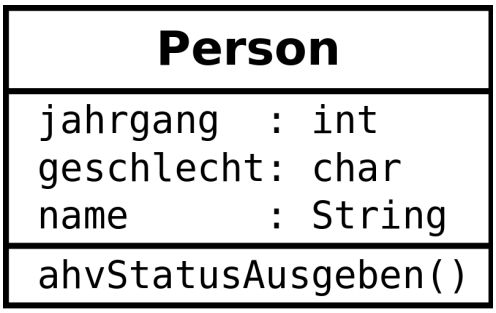


Schreiben Sie danach ein Hauptprogramm, worin Sie ein `Haus` und drei `Zimmer` erzeugen. Jedes Zimmer hat seine Quadratmeterzahl und dem Haus werden die drei Zimmer hinzugefügt.

Rufen Sie schließlich die Methode `quadratmeterAusgeben()` auf, welche die totale Anzahl der Quadratmeter im Haus ausgeben soll.

Aufgabe 16.4 «AHV Status»

Implementieren Sie zunächst die folgende Klasse:



Schreiben Sie nun ein Programm, das zuerst eine Person erfasst und danach ihre Methode `ahvStatusAusgeben()` aufruft.

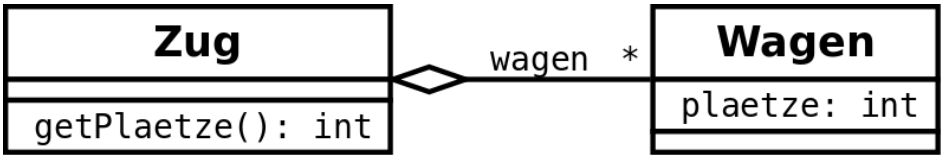
Der AHV-Status ist in der Schweiz wie folgt definiert:

Die Beitragspflicht für die AHV (Alters- und Hinterbliebenenversicherung) beginnt ab dem 1. Januar nach Vollenden des 17. Altersjahres und endet (bis zur vollständigen Umsetzung des Artikels 8 der neuen Schweizer Bundesverfassung) je nach Geschlecht unterschiedlich; nämlich im Alter von 64 Jahren bei Frauen bzw. 65 Jahren bei Männern.

Die Methode `ahvStatusAusgeben()` soll entweder «noch nicht AHV-beitragspflichtig», «AHV-beitragspflichtig» bzw. «bereits AHV-Empfänger» ausgeben.

Aufgabe 16.5 «Eisenbahn»

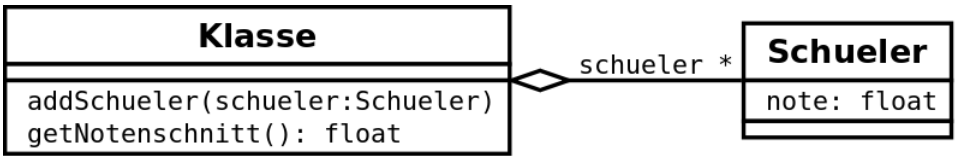
Schreiben Sie zunächst die beiden folgenden Klassen:



Erfassen Sie einige Wagen, welche Sie der Sammlung `wagen` im `Zug` hinzufügen. Jeder Wagen hat seine eigene Anzahl Plätze. Rufen Sie zuletzt die Methode `getPlaetze()` auf, um die gesamte Anzahl der Plätze im Zug zu errechnen. Die Methode `getPlaetze()` muss somit mit Hilfe einer Schleife alle Plätze der einzelnen Wagen zu einer Gesamtanzahl hinzufügen.

Aufgabe 16.6 «Noten-Durchschnitt»

Schreiben Sie wieder vorab die beiden folgenden Klassen:



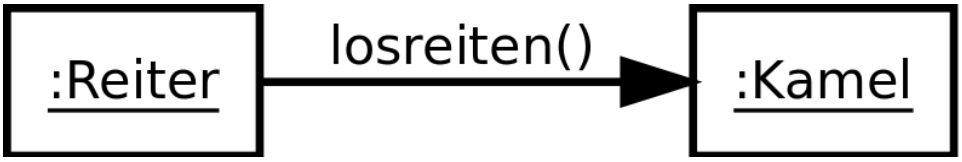
Erfassen Sie einige Schüler mit ihren eigenen Schulnoten und fügen Sie diese Objekte mit der Methode `addSchueler()` zur Klasse hinzu.

Rufen Sie nun die Methode `getNotenschnitt()` auf und geben Sie die Durchschnittsnote der Klasse aus.

Gegenüber der vorigen Aufgabe «Eisenbahn» hat diese Aufgabe zwei kleine Schwierigkeiten: Zunächst werden die Schüler mittels einer Methode der Klasse hinzugefügt und des Weiteren ist zur Berechnung des Durchschnittes sowohl die Summe, wie auch die Anzahl der Noten zu berücksichtigen.

17 | Delegation

«Was soll ich mich sorgen, mach Du Dir sorgen»



Der Begriff **Delegation** wird in der Softwareentwicklung unterschiedlich gehandhabt.

In diesem Kapitel verwende ich die **ursprüngliche Bedeutung**. Eine Klasse erweitert ihre Operationen um äquivalente Methoden aus einer anderen Klasse. So könnte ein Reiter seine `losreiten()`-Methode aus einer Klasse `Kamel` beziehen. Hierzu verwendet er eine Referenz auf sein Lasttier.

Allgemein: Eine Methode (`tuWas()`) in einem Objekt (`a`) wird derart implementiert, indem es die entsprechende Methode (`tuWasAehnliches()`) in einem bereits vorhandenen Objekt (`b`) aufruft.

17.1 Interaktion

Damit unser Objekt `a` mit einem Objekt `b` interagieren kann, muss Objekt `a` eine Referenz[22] auf das Objekt `b` kennen. Mit Hilfe dieser Referenz «refAufB» kann das Objekt `a` nun Methoden auf dem Objekt `b` aufrufen.

In JAVA wird dies mit dem Punkt «.»-Operator bewerkstelligt:

```
class B {
  public void tuWas() { ... }
}
```

Aufruf:

```
class A {
  // Attribut, Parameter oder lokale Variable:
   B refAufB = ...;

  public void tuWasAehnliches() {
    refAufB.tuWas();
  }
}
```

Diese Art der Interaktion nenne wir **Delegation**. Delegation könnte man auch als *Aggregation, um die Funktionalität zu erweitern*, betrachten. Die Delegation ist eigentlich kein neues Konzept, sondern eine Zusammensetzung von zwei Prinzipien: **Aggregation** (s. Kap. 15 auf Seite 39) & **Operation** (= Funktionsaufruf) (s. Kap. 16 auf Seite 49). Technisch könnte man sagen, die Delegation ist ein Funktionsaufruf in einem anderen Modul zu einem gegebenen Objekt.

Mit diesen Aussagen wird man der Delegation aber nur teilweise gerecht. Es geht auch darum, eine neue Datenstruktur um eine bestehende Struktur zu erweitern (aus bestehenden Datenstrukturen zu *komponieren*) und im zugehörigen bestehenden Modul Funktionen aufzurufen, die eng mit der bestehenden Struktur verbunden sind.

Programmiert wird das, indem zunächst jede Klasse einzeln programmiert wird, doch danach wird ein Objekt (mittels Referenzvariable) Teil des anderen Objektes.

[22] Eine «Referenz» wird auch Pointer, (lokale) Variable oder Attribut genannt.

17.2 Klassische Delegation

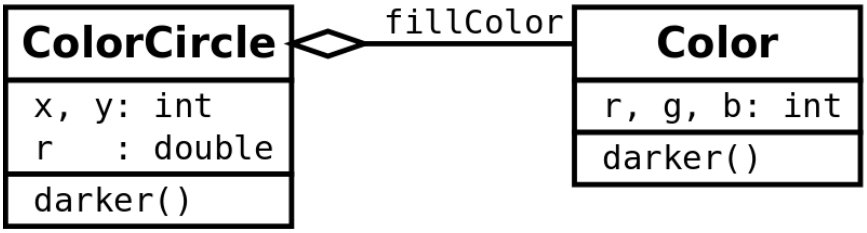
Die klassische Delegation kommt zur Code-Wiederverwendung noch ohne Vererbung aus.

Beispiel 17.1. Zeichnungsprogramm

In einem Zeichnungsprogramm gäbe es (neben anderem) die Möglichkeit, Kreise (Circle) zu zeichnen.

Ein Kreis bestehe aus Mittelpunkt (x, y) und einem Radius. Bis jetzt wird ein Kreis einfach mittels schwarzer Fläche gezeichnet. Ebenso gibt es bereits eine Klasse Color (Farbe) mit dem RGB-Farbwert und einigen Methoden zur Farbänderung, insbesondere `dunkler()`. Neu soll der Kreis (`Circle`) mit Funktionalitäten der „Farbe“ ergänzt werden.

Dazu wird `Circle` in `ColorCircle` umbenannt und um eine Farbe (`Color`) ergänzt:



Die Klassen sehen vor der Anwendung der Delegation wie folgt aus:

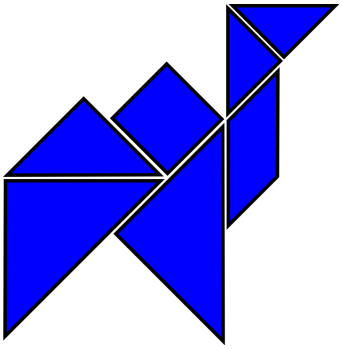
```
class ColorCircle{
  int x, y;
  double r;
}
```

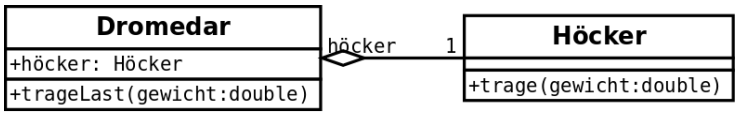
```
class Color {
  int r, g, b;

  void darker() {
    r = darker(r);
    g = darker(g);
    b = darker(b);
  }

  int darker(int v) {
    if(v < 10) {
      v = 10;
    }
    return v - 10;
  }
}
```

Mittels Delegation kann obiger Kreis nun ganz einfach um die Methoden der Farbe ergänzt werden:

```
class ColorCircle{
  int x, y;
  double r;
  Color fillColor;

  void darker() {
    fillColor.darker();
  }
}
```

Beispiel 17.2. Dromedar Das nächste Beispiel erweitert ein Lama um einen sog. Höcker[23]. Anstatt unser (neues) Dromedar die Last selbst tragen zu lassen, delegieren wir diese «schwere» Aufgabe an ebendiese Höcker.






```
class Dromedar {
  Hoecker hoecker;
  void trageLast(double gewicht) {
    hoecker.trage(gewicht);
  }
}
```

```
class Hoecker {
  void trage(double gewicht) {
    print("Trage␣" + gewicht + "␣kg.");
  }
}
```

[23]Einhöckrige Kamele werden Dromedare genannt.

17.3 Delegationsattribut

Um Aufgaben an andere Objekte zu delegieren, werden fast ausschließlich Attribute als Referenzvariable verwendet. Diese sog. **Delegationsattribute**[24] zeigen auf aggregierte Objekte, welche spezielle Teilaufgaben übernehmen.

Eine solche Referenzvariable muss mit einem zugehörigen Objekt initialisiert werden. Diese Eintragung kann im Konstruktor oder mit speziell dafür zur Verfügung gestellten Methoden [`setXYZ()` , `addXYZListener()` , `addXYZHandler()` , ...] vollzogen werden.

Auf den beiden folgenden Seiten sind einige Standardvorgehen für die Implementation der Delegation angegeben. Jeweils wird in der delegierenden Klasse ein Attribut geführt. An dieses Attribut werden die Ereignisse (Methoden) später weitergeleitet. In den folgenden vier Vorgehensweisen wird aufgezeigt, wie die Referenzvariablen registriert werden können, damit eine Delegation möglich wird:

A) Attribut-Zuweisung

B) Konstruktor

C) *setter*-Methoden

D) *Lazy Instantiation*

[24] Delegationsattribute hatten wir bisher auch **Aggregationsvariable** genannt: Siehe Kap. 15, 16.4 und 17.1

17.3.1 Registrieren von Objekten

A) Aus dem Hauptprogramm direkt dem Attribut angeben:

Dies ist die einfachste Möglichkeit ein Delegationsattribut mit einem sinnvollen Wert zu belegen; wenn wir uns zurück erinnern an das Kapitel über die Aggregation – wo wir die Methoden noch nicht kannten – war dies die einzige Variante.

```
ColorCircle circle1 = new ColorCircle();
Color       green   = new Color(0, 255, 63);

circle1.fillColor = green;
```

Diese Variante ist (manchmal zu unrecht) verpönt, denn es erlaubt unbeteiligtem Programmcode, die Attribute zu verändern, was dem Konzept des *Information-Protection* widerspricht.

B) Konstruktor:

- Das Objekt dem Konstruktor als Parameter mitzugeben, ist eine zweite Variante des Verankerns eines Delegations-Objektes:

```
Color       green   = new Color(0, 255, 63);
ColorCircle circle1 = new ColorCircle(green);
```

Dabei ist der folgende Konstruktor gegeben:

```
public ColorCircle(Color fillCol) {
  this.fillColor = fillCol;
}
```

- Im Konstruktor (oder via eigene `init()`-Methode) kann das Objekt auch mit Default-Wert erzeugen werden. Dies ist eine explizite Möglichkeit für das, was oben bereits via Parameter im Konstruktor vollzogen wurde.

```
public ColorCircle() {
  this.fillColor = new Color(0, 255, 63);
}
```

C) Mit setter- oder adding-Methoden:

- Die klassische Setter-Methode bei 1:1 - Beziehungen sieht wie folgt aus ...

```
public void setColor(Color newFillColor) {
  this.fillColor = newFillColor;
}
```

- ... oder mit `add()`-Registrierung bei 1:n - Beziehungen ...

```
public void addKreis(Kreis k) {
  this.kreise.add(k);
}
```

- ... oder mit einer `add()`-Registrierung bei 1:n - Beziehungen mittels *Lazy Instantiation*:

```
public void addXYZListener(XYZListener l) {
  if(null == this.listeners) {
    this.listeners = new ArrayList<...>();
  }
  this.listeners.add(l);
}
```

D) In der Methode bei Bedarf erzeugen (Lazy Instantiation):

- Neues Objekt erstellen:

```
public void darker() {
  if(null == this.fillColor) { // not yet defined
    this.fillColor = new Color(0, 255, 63);
  }
  this.fillColor.darker();
}
```

- Persistente Daten (z. B. via `java.util.prefs.Preferences`) einlesen:

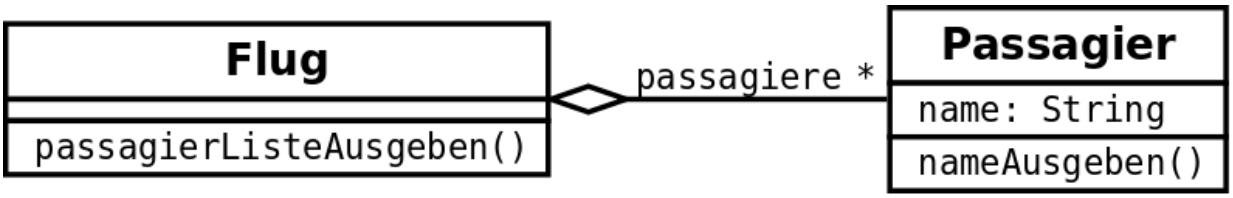
```
public void darker() {
  if(null == this.fillColor) { // not yet defined
    String colName = preferences.get("color.fill", "green");
    this.fillColor = makeColor(colName);
  }
  this.fillColor.darker();
}
```

17.4 Delegation in 1:n - Beziehungen

Betrachten wir wieder unser Flug-Passagier-Beispiel aus dem Kapitel über die Aggregation (s. Kap. 15.1.2 auf Seite 42). Dort gehörten zu einem Flug mehrere Passagiere. Nun wollen wir wieder die Passagierliste ausgeben, doch diesmal mit einer sinnvollen Delegation.

Dazu verschieben wir zunächst die Funktion `passagierListeAusgeben()` in die Klasse Flug. Jeder Passagier soll zusätzlich eine Methode `nameAusgeben()` enthalten.

Die Methode `passagierListeAusgeben()` des Flugs kann nun einfach via Delegation auf die Methode `nameAusgeben()` der Passagiere zurückgreifen.




```
public class Passagier {
  String name;

  void nameAusgeben() {
     System.out.println(name);
  }
}
```

```
public class Flug {
  List<Passagier> passagiere = new ArrayList<Passagier>();

  void passagierListeAusgeben() {
      for(Passagier p: passagiere) {
          p.nameAusgeben();
      }
  }
}
```

Das Hauptprogramm könnte nun so aussehen:

```
void main() {
  Flug flug    = new Flug();

  Passagier p1 = new Passagier();
  Passagier p2 = new Passagier();
  p1.name = "Gressly␣Freimann";
  p2.name = "Guggisberg"      ;

  flug.passagiere.add(p1);
  flug.passagiere.add(p2);

  flug.passagierListeAusgeben();
}
```

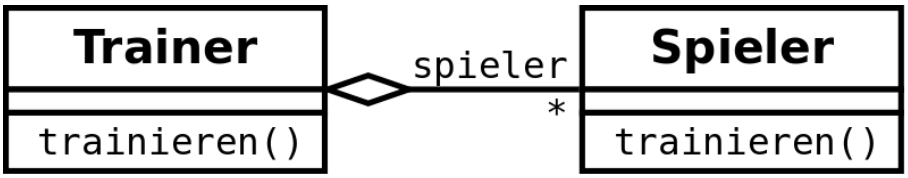
17.4.1 Arten der Schleifen in 1:n - Beziehungen

Bei 1:n - Beziehungen kann nicht direkt an ein Attribut delegiert werden. Stattdessen muss für jedes referenzierte Objekt der Sammlung (Array, `Collection`, ...) die entsprechende Methode aufgerufen werden. Dies geschieht typischerweise in der JAVA «for-each»-Schleife.

Erste Art: Methode delegieren

Beispiel 17.3. Trainer - Spieler: trainieren()

Im ersten Beispiel soll ein Trainer alle seine Spieler trainieren. Seine Methode heißt denn auch `trainieren()`; wie auch die Methode der Spieler.



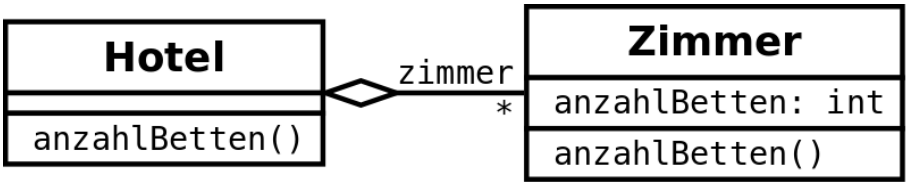

```
class Trainer{
  ArrayList<Spieler> spieler;
  ...
  void trainieren() {
    for(Spieler s: spieler) {
      s.trainieren();
    }
  }
}
```

```
class Spieler {
  void trainieren() {
    ...
  }
}
```

Zweite Art: Information sammeln

Beispiel 17.4. Hotel - Zimmer: `anzahlBetten()`

In diesem zweiten Beispiel soll eine Hotel-Software alle Betten im Hotel zählen. Dafür existiert aber nur eine «Collection» von Zimmern. Glücklicherweise enthalten diese Zimmer eine Methode `anzahlBetten()`.




```
class Hotel{
  ArrayList<Zimmer> zimmer;
  ...
  int anzahlBetten() {
    int total = 0;
    for(Zimmer z: zimmer) {
      total = total +
              z.anzahlBetten();
    }
    return total;
  }
}
```

```
class Zimmer {
  int anzahlBetten;
  ...
  int anzahlBetten() {
    return anzahlBetten;
  }
}
```

Die Methode `anzahlBetten()` ist zwar in den einzelnen Zimmern vorhanden, dennoch muss ich im Hotel diese Information zuerst sammeln, bevor ich diese sinnvoll nutzen kann.

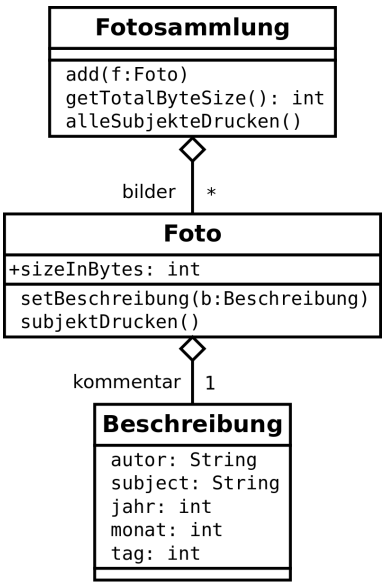
17.5 Aufgaben zur Delegation

Aufgabe 17.1 «Der Sportchef delegiert»

Lösen Sie die Aufgabe `http://www.programmieraufgaben.ch` (Web-Code: 3oxh 5vny).

Aufgabe 17.2 «Fotosammlung»

Implementieren Sie folgendes Klassendiagramm und testen Sie Ihren Code mit dem rechts gegebenen Code-Fragment:

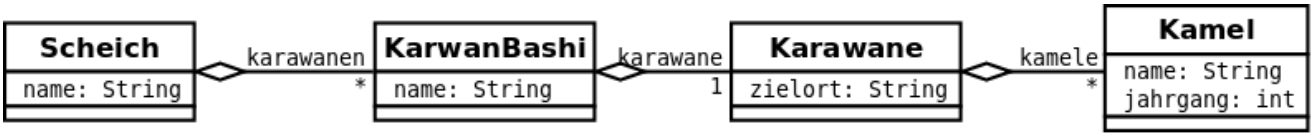



```
Fotosammlung fs;
Foto f;
fs = new Fotosammlung();

f = shoot("gressly", "baum", 510038);
fs . add(f);

f = shoot("meier", "haus", 733208);
fs . add(f);

fs . alleSubjekteDrucken();

// mit folgender Hilfsfunktion
Foto shoot(String autor, String subject,
           int size) {
  Beschreibung b;
  Foto f;
  b = new Beschreibung();
  f = new Foto();
  b . autor       = autor;
  b . subject     = subject;
  f . sizeInBytes = size;
  f . setBeschreibung(b);
  return f;
}
```

Aufgabe 17.3 «Karwan-Bashi»

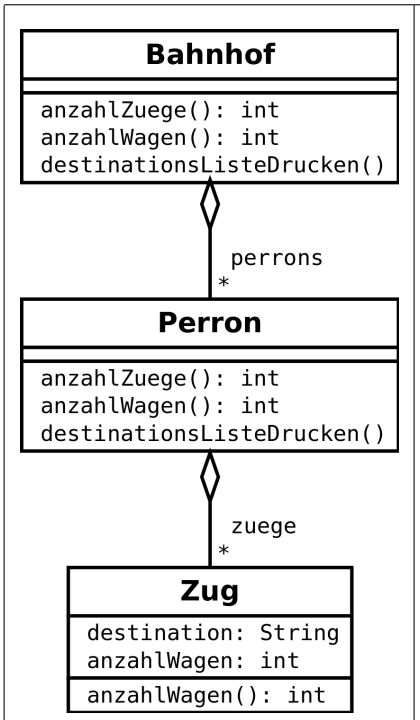
Mit *Karwan-Bashi* bezeichnet man den Führer einer Karawane. Ein *Scheich* kann mehrere Karawanen besitzen, die ihm Waren transportieren. Natürlich besteht jede Karawane aus mehreren Kamelen.

Implementieren Sie folgendes Klassendiagramm und instanzieren Sie einen Scheich, der im Besitz von drei Karawanen ist.



Aufgabe 17.4 «Eisenbahn»

Implementieren Sie folgendes Klassendiagramm und testen Sie Ihren Code mit dem rechts gegebenen Code-Fragment. Sie brauchen dazu in der Klasse `Zug` einen Konstruktor mit den beiden Parametern «Destination» und «Anzahl Wagen».




```
Zug z1 = new Zug("Bern"    ,  8);
Zug z2 = new Zug("Roma"    , 12);
Zug z3 = new Zug("Stambul",  4);
Zug z4 = new Zug("Paris"   ,  8);

Perron p1 = new Perron();
p1.zuege.add(z1);
p1.zuege.add(z2);

Perron p2 = new Perron();
p2.zuege.add(z3);

Perron p3 = new Perron();
p3.zuege.add(z4);

Bahnhof bh = new Bahnhof();
bh.perrons.add(p1);
bh.perrons.add(p2);
bh.perrons.add(p3);

bh.destinationsListeDrucken();
System.out.println("Total␣Zuege:␣" +
                   bh.anzahlZuege());
System.out.println("Total␣Wagen:␣" +
                   bh.anzahlWagen());
```

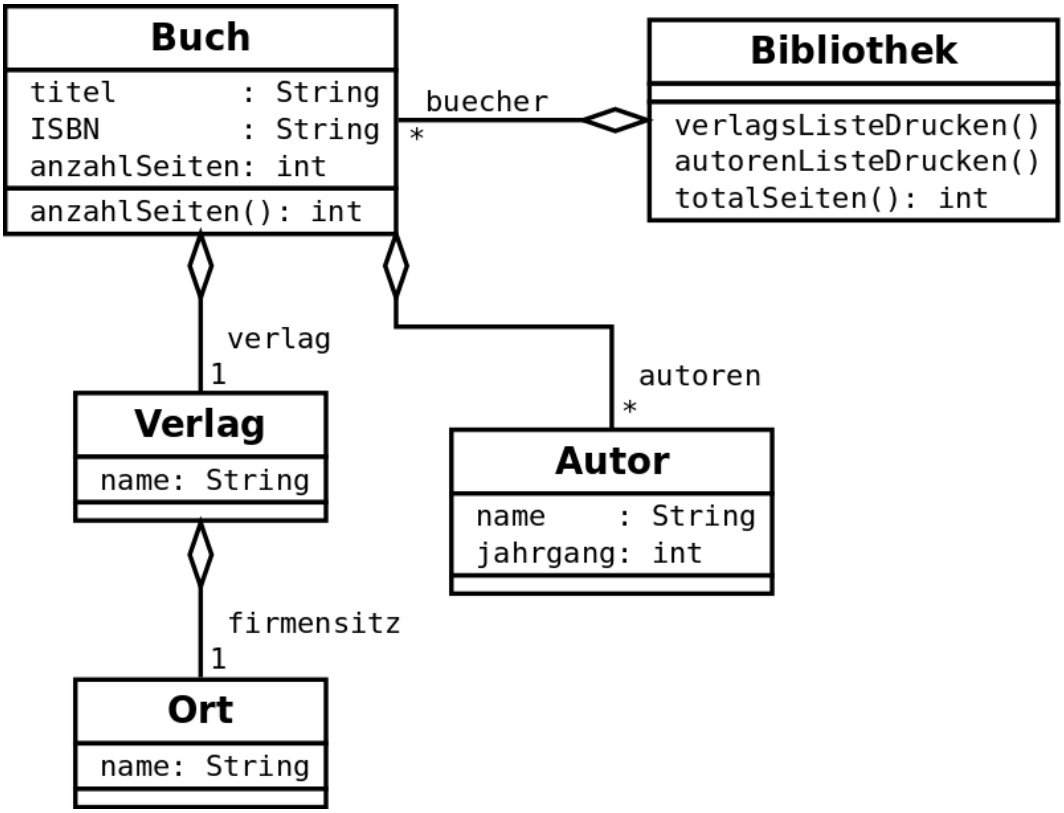
Aufgabe 17.5 «Hotel - Zimmer»

Implementieren Sie die beiden Klassen `Hotel` und `Zimmer` aus dem 2. Beispiel zu 1:n - Beziehungen (s. Kap. 17.4 auf Seite 74).

Wählen Sie eine akzeptable Möglichkeit, die *Delegations-Attribute* zu registrieren (s. Kap. 17.3 auf Seite 68) und implementieren Sie diese.

Aufgabe 17.6 «Bibliothek»

Implementieren Sie das folgende Klassendiagramm.

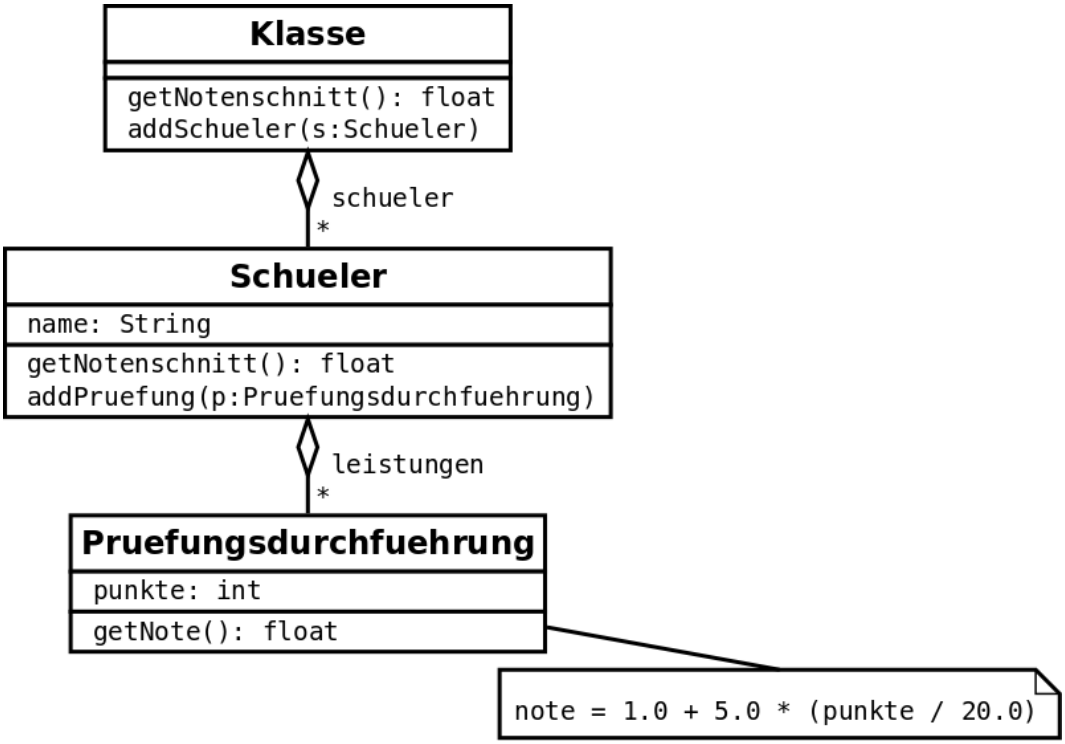


Verwenden Sie die Klassen in folgendem Hauptprogramm (Pseudo-Code). Fügen Sie bei Bedarf in den Klassen weitere Attribute und Methoden hinzu.

```
main() {
  // erstellen:
  eine Bibliothek
  drei Buecher
  zwei Verlage
  zwei Autoren
  zwei Orte

  // danach
  bibliothek.verlagsListeDrucken();
  bibliothek.autorenListeDrucken();
  System.out.println("Seiten:␣" + bibliothek.totalSeiten());
}
```

Aufgabe 17.7 «Notenschnitt»

Lehrer Sokrates will von seinen Sprösslingen möglichst automatisch die Noten und den Durchschnitt der Noten berechnen. Dazu hat er sich folgendes UML-Klassendiagramm ausgedacht:



Implementieren Sie zunächst das obige Diagramm, fügen Sie sinnvolle Konstruktoren ein und testen Sie Ihr Programm anschließend mit dem untenstehenden Code:

```
Klasse sKlasse = new Klasse();

Schueler s1    = new Schueler("Kasperle");
Schueler s2    = new Schueler("Seppel"  );

sKlasse.addSchueler(s1);
sKlasse.addSchueler(s2);;

Pruefungsdurchfuehrung p1_1 = new Pruefungsdurchfuehrung(17);
Pruefungsdurchfuehrung p1_2 = new Pruefungsdurchfuehrung(13);
Pruefungsdurchfuehrung p2_1 = new Pruefungsdurchfuehrung(12);

s1.addPruefung(p1_1);
s1.addPruefung(p1_2);
s2.addPruefung(p2_1);

System.out.println("Durchschnitt␣von␣" + s1.name + ":␣"
                   + s1.getNotenschnitt()              );
System.out.println("Durchschnitt␣der␣Klasse:␣"
                   + sKlasse.getNotenschnit()          );
```

Aufgabe 17.8 «Sand am Meer»

Programmieraufgaben zur Delegation gibt es wie *Sand am Meer*. Schreiben Sie die JAVA-Klassen der folgenden Beispiele, fügen Sie die angegebenen Methoden in jede Klasse ein und delegieren Sie das Verhalten von links nach rechts.

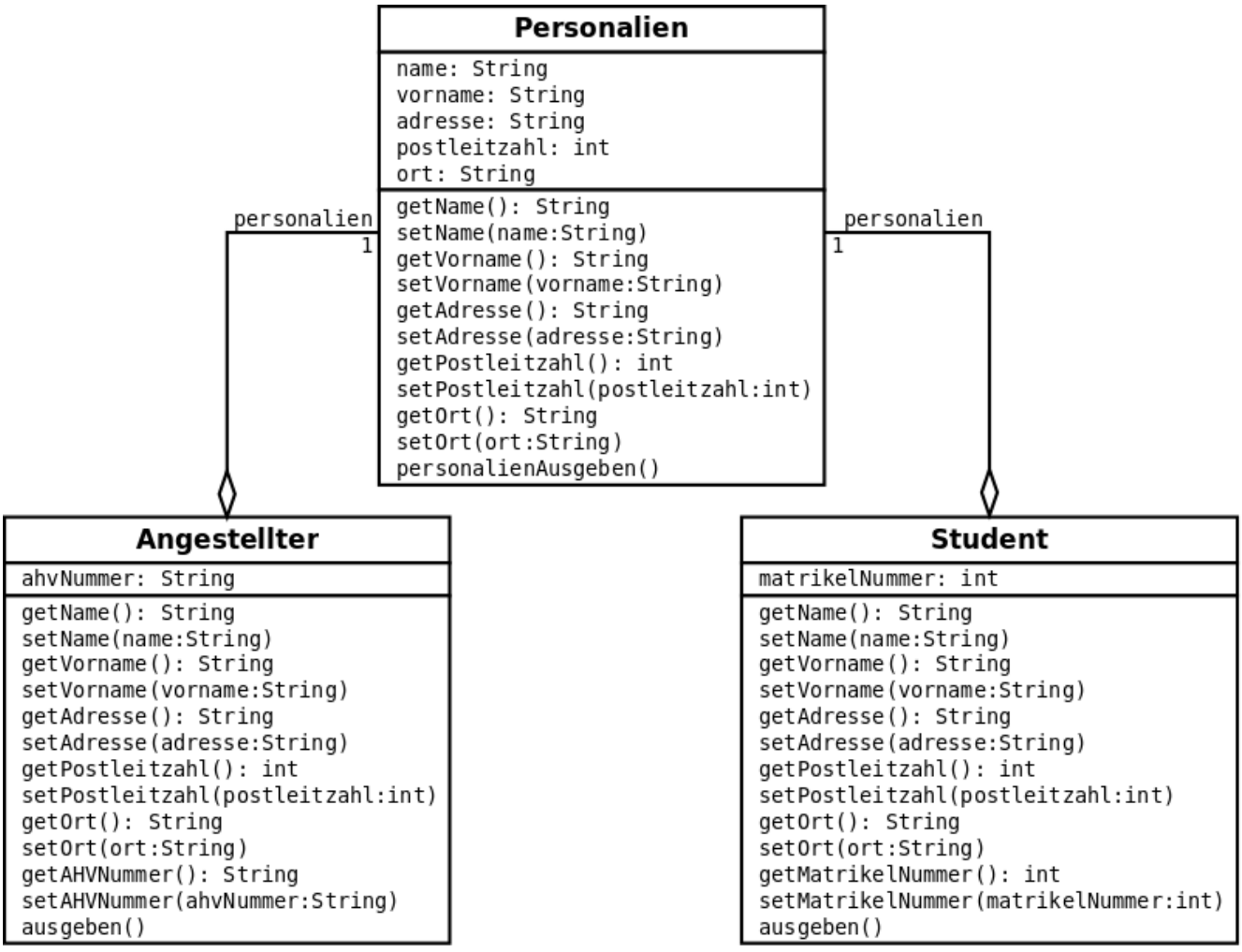
- `sand()`: Welt → Meer → Strand
- `trainieren()`: Sportchef → Trainer → Spieler
- `transport()`: Karwan Bashi → Treiber → Kamel
- `byteSumme()`: Fotosammlung → Foto → Pixeldata
- `anzahlZuege()`: Bahnhof → Perron → Gleis
- `abspielen()`: Film → Bild → Pixel
- `vermögen()`: Bank → Kunde → Konto

Aufgabe 17.9 «Personalien»

Das folgende Klassendiagramm enthält sehr viele sog. *getter*- und *setter*-Methoden. Zum Glück können diese mit modernen Entwicklungsumgebungen (z. B. *eclipse*[25]) allesamt automatisch generiert werden. Beachten Sie, dass sowohl ein `Angestellter`, wie auch ein `Student` die meisten *getter*- und *setter*-Methoden an die `Personalien`-Klasse delegieren darf.

Die Ausgabe (`ausgeben()` bzw. `personalienAusgeben()`) soll einfach auf der Konsole alle Attribute ausgeben (`System.out.println()`).

Beachten Sie bei der Umsetzung in JAVA-Code auch, dass die beiden `ausgeben()`-Methoden einfach «`personalienAusgeben()`» aufrufen können und dabei lediglich ein einziges weiteres Attribut ausgeben müssen.



[25] S. `www.eclipse.org`

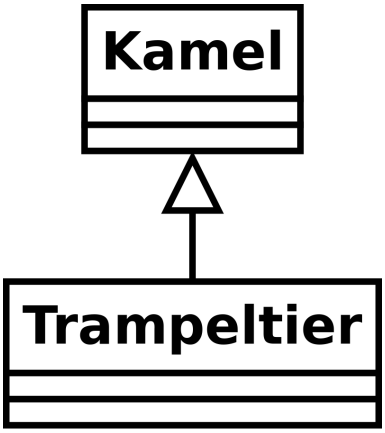
«*Darwin und andere Vorfahren*»

Um noch mehr Programmcode einzusparen, aber auch um Programmcode wieder zu verwenden, wurde das Konzept der Vererbung geschaffen. Eine bereits bestehende Klasse kann einfach um Attribute und Methoden erweitert werden, ohne den bestehenden Code verändern oder kopieren zu müssen.

Geek Tipp 6

Kopierter Code ist böse: Er verbreitet Sachschaden, Verderbnis und Tod!

Beispiel 18.1. Trampeltier



Jedes Trampeltier ist ein spezielles Kamel und beinhaltet somit alle Eigenschaften aller Kamele. Man sagt «das Trampeltier erbt vom Kamel».

Abstraktion I: Taxonomie

Objekt → Klasse

Die Abstraktion, wie wir sie bisher betrachtet hatten, zeigt die essentielle Charakteristik eines Objektes, welches sich von allen anderen Objekten unterscheidet. Diese Art von Abstraktion kennen wir bereits aus der Klassenbildung (s. Kap. 12.1.3 auf Seite 15). Sie hat noch nichts mit Vererbung zu tun. Bei der Taxonomie sucht der Analytiker Gemeinsamkeiten der in der realen Welt vorkommenden Objekte und fasst diese Objekte zu Klassen zusammen.

Bsp.: „Herr Huber **ist ein** Kunde“, „Max und Moritz **sind** Spitzbuben“.

18.1 Hierarchien und Vererbung

18.1.1 Definition Hierarchie

Hierarchie ist eine Rangordnung oder Reihenfolge von Klassen. Wir kennen bereits die Delegation (s. Kap. 17 auf Seite 63) als eine solche Reihenfolge. Hier lernen wir eine zweite Bedeutung kennen, bei der ein Objekt noch viel stärker an sein ranghöheres Objekt gebunden wird.

In der objektorientierten Softwareentwicklung (OOSE) gibt es eine zweite Art der Abstraktion:

Abstraktion II: Vererbung

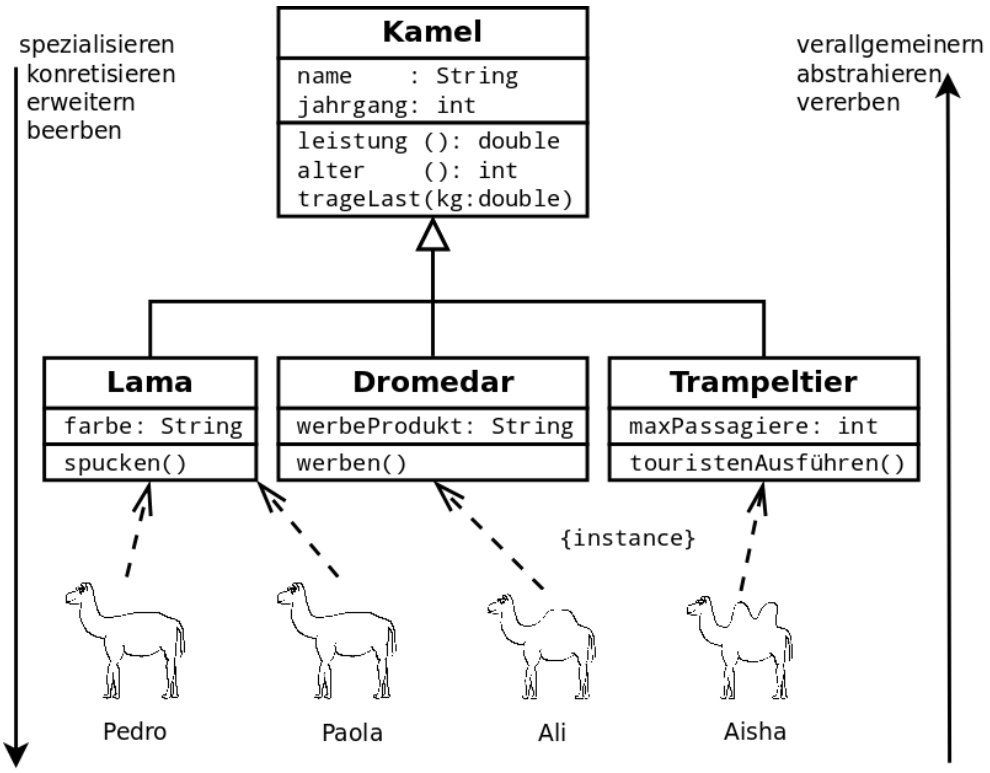
Klassen → Superklassen

In einem zweiten optionalen Schritt können vom Analytiker Gemeinsamkeiten in den Klassen gesucht werden. Anschließend werden diese Klassen zu abstrakteren Klassen, sogenannten Superklassen oder Überklassen, zusammengefasst.

Bsp.: «Eine Hauskatze **ist ein** Säugetier», «Tannen **sind** Nadelbäume».

In beiden Fällen (Taxonomie und Vererbung) sprechen wir von einer **ist ein**-Beziehung. Die Unterscheidung zwischen Instanz und Vererbung ist in der Umgangssprache meist nicht nötig, da es aus dem Zusammenhang in der Regel klar ist, worum es sich handelt.

18.1.2 Abstrahieren vs. Spezialisieren



Bemerkung zu obigem Bild: In der Umgangssprache wird meist nur von der unspezifischen Familie «Kamel» gesprochen, selbst wenn es sich dabei um die Art «Trampeltier» oder «Dromedar» handelt. In einigen Dialekten ist selbst der Name der Tierart Trampeltier nicht geläufig, und das Wort Kamel wird fälschlicherweise verwendet, um das Trampeltier vom Dromedar zu unterscheiden.

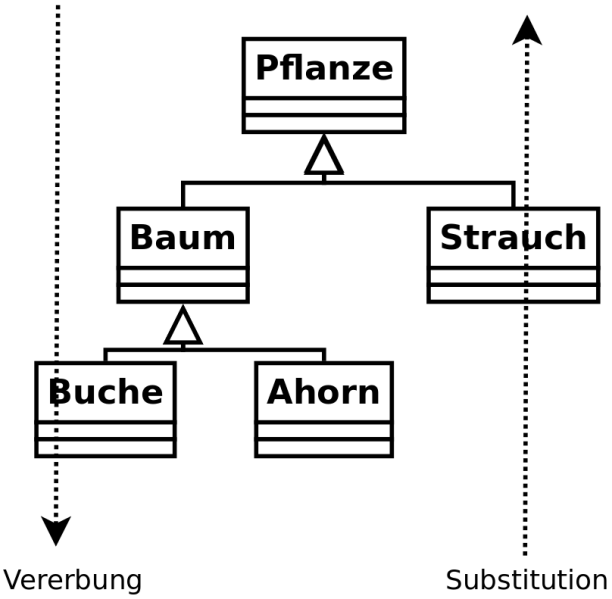
Eine Vererbung (inheritance) ist eine Hierarchie von Klassen. Die Vererbung erlaubt der Unterklasse (der vererbten Klasse) alle Eigenschaften (Attribute, Methoden) der Überklasse mitzuverwenden.

Elementare Vererbung ist

- das Erweitern um Attribute und
- das Erweitern um Methoden.

18.1.3 Begriffe

Synonyme für übergeordnete Klassen (Substitution):	Vaterklassen, Basisklassen, Superklassen, Oberklassen, parent class, base class, vererbende Klasse, Überklasse, Elternklasse, Verallgemeinerung, Abstraktion
Synonyme für untergeordnete Klassen (Vererbung):	Subklassen, Unterklassen, Kindklasse, abgeleitete Klassen, child class, sub class, derived class, erbende Klasse, Spezialisierung, Konkretisierung



Es handelt sich bei der elementaren Vererbung wie bei der klassischen Delegation um eine einfache Art der Code-Wiederverwendung.

Das Auffinden von Gemeinsamkeiten in verschiedenen Klassen wird **Generalisierung** genannt. Dabei wird eine neue Klasse erstellt, die die Gemeinsamkeiten zusammenfasst. Das Erben von diesen gemeinsamen Eigenschaften in den Subklassen wird **Vererbung** oder **Spezialisierung** genannt.

JAVA Beispiel Student:

```
public class Person {

  private String name;
  private String vorname;

  public void print() {
    System.out.println("Nachname:␣" + name);
    System.out.println("Vorname:␣␣" + vorname);
  }
}


Public class Student extends Person {

  private String matrikelnummer;

  public void printMatrikelnummer() {
    System.out.println("Matrikelnummer:␣" + matrikelnummer);
  }
}
```

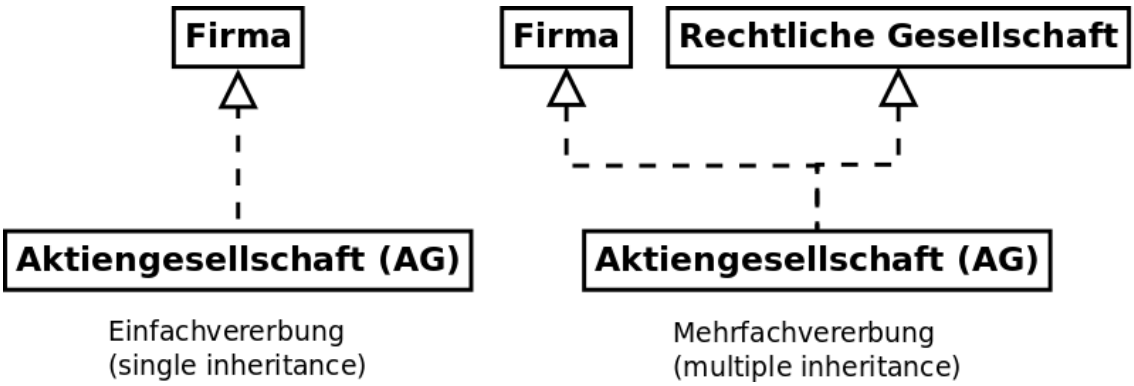
18.2 Vererbungswarnung

Vermeiden Sie Vererbung, wenn es Alternativen gibt. Es macht z. B. keinen Sinn, die Klasse «Mensch» von der Klasse «Kamel» abzuleiten, nur weil in der Klasse «Kamel» die Methode `ichHeisse()` bereits implementiert ist!

18.3 Mehrfachvererbung

Wir unterscheiden zwei Arten von Vererbungen:

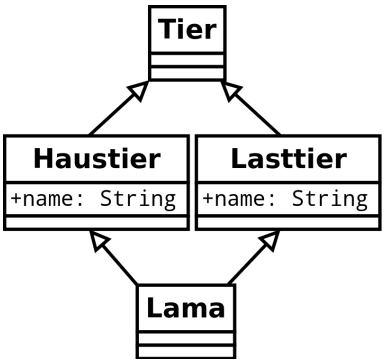
- **Single inheritance**: Die einfache Vererbung bedeutet eine Beziehung zwischen zwei Klassen (Subclass, Superclass).
- **Multiple inheritance**: Die mehrfache Vererbung bedeutet eine Beziehung von mehreren Superklassen zu einer Subklasse.



Einige Programmiersprachen kennen die Mehrfachvererbung (multiple inheritance). Dabei kann eine Klasse von mehreren Klassen erben. Das Lama könnte also erstens ein Lasttier sein, aber auch alle Attribute und Methoden des Haustiers erben.

Dieses Konzept hat aber einen Nachteil. Wenn nämlich eine Methode in beiden Superklassen überschrieben wurde, so ist oft nicht mehr klar, welche Methode denn bei der Kindklasse nun gültig ist. Da die Mehrfachvererbung häufig zu einem unverständlichen Code führte, hat JAVA diese bewusst nicht eingeführt. JAVA kennt einzig die Einfachvererbung (single inheritance).

Deadly Diamond of Death



Das Problem mit obiger Klasse `Lama` ist offensichtlich das Attribut `name`, das in beiden Oberklassen vorkommt. Es ist den Programmierern oft nicht klar, welches Attribut denn genau geerbt wurde. Das obige Diagramm macht auch klar, warum die Mehrfachvererbung manchmal auch Deadly Diamond of Death genannt wird.

18.4 Aufgaben zur Vererbung

Aufgabe 18.1 «Klassendiagramm»

Zeichnen Sie ein Klassendiagramm zu den folgenden Begriffen: Kamel, Dromedar, Trampeltier, Höcker, Gewicht und Alter. Verwenden Sie die Symbole der Vererbung (Dreieck für is-a) und der Aggregation (Raute für has-a). Überlegen Sie auch, wo ein einfaches Attribut sinnvoller ist als eine Delegation oder eine Vererbung.

Aufgabe 18.2 «ist ein»

Die folgenden Beziehungen sind allesamt ist-ein Beziehungen. Finden Sie heraus, ob es sich dabei um eine Instanz (Objekt / Klasse) oder um eine Vererbung (Klasse / Superklasse) handelt:

- Herr Hubert Müller - Kunde
- Frau Studer - Angestellte
- Trampeltier - Kamel
- Aisha, das Trampeltier - Kamel
- Das Buch, das Sie vor sich in der Hand halten - Medium
- Alkohol - Droge
- Eva (bzw. Adam) – Mensch

Aufgabe 18.3 «ist-ein/hat-ein»

Die folgenden Beziehungen sind allesamt ist-ein oder hat-ein Beziehungen. Finden Sie heraus, ob es sich dabei um eine Instanz (Objekt / Klasse), um eine Vererbung (Klasse / Superklasse) oder um eine Delegation (Objekt-Attribut) handelt:

- Buch - Seite
- Buch - Medium
- Motorola 68000 - Prozessorfamilie
- PC Bildschirm (Monitor) - Ausgabegerät
- Dackel - Hund
- J. S. Bach - Komponist
- Freiheitsstatue - Statue auf Liberty Island
- Mensch - Trockennasenaffe
- Tee - Getränk
- Tee – Aroma

Aufgabe 18.4 «Personalien vererbt»

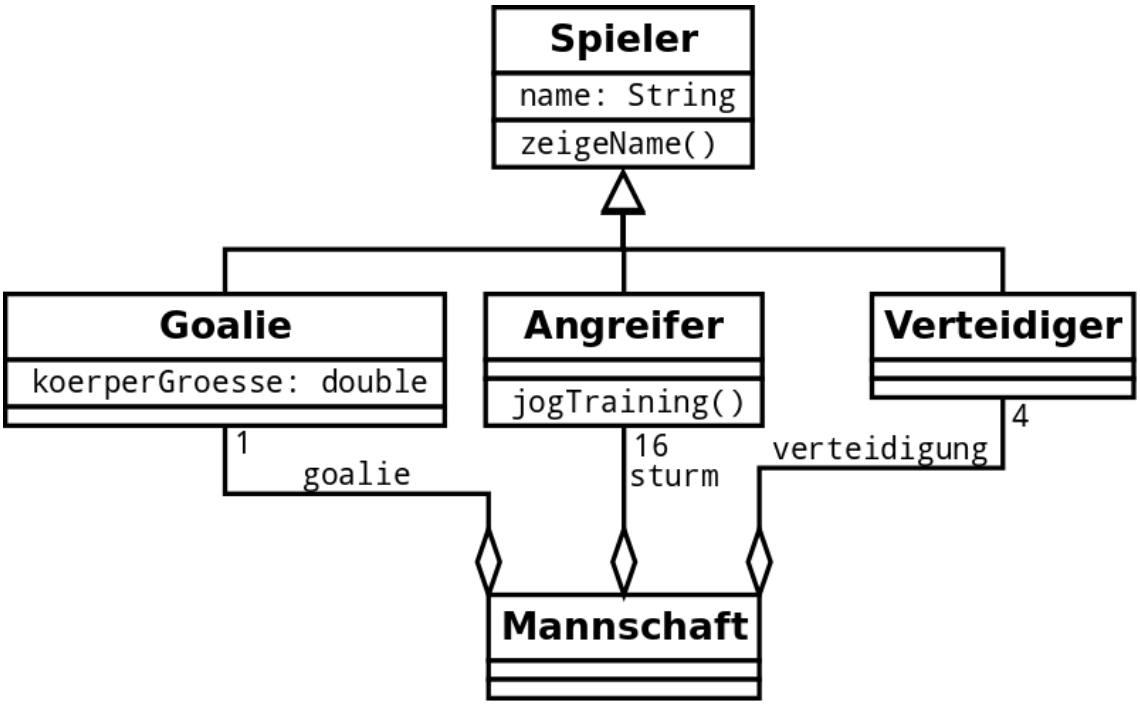
Verbessern Sie die Aufgabe «Personalien» (s Aufg. 17.9 auf Seite 80) indem Sie die Klasse `Personalien` durch eine Klasse `Person` ersetzen und danach die beiden Nutzklassen (`Student` bzw. `Angestellter`) davon erben lassen (`extends` in JAVA). Löschen Sie danach allen nicht mehr benötigten Code.

Aufgabe 18.5 «Fahrzeuge»

Zeichnen Sie ein Klassendiagramm zu den folgenden Begriffen: Fahrzeug, Fahrrad, Motorrad, Auto, Rad, Kilometerstand, Farbe. Verwenden Sie sowohl die Vererbung als auch die Aggregation.

Aufgabe 18.6 «Elementare Fußballmannschaft»

Implementieren Sie das folgende Klassendiagramm:

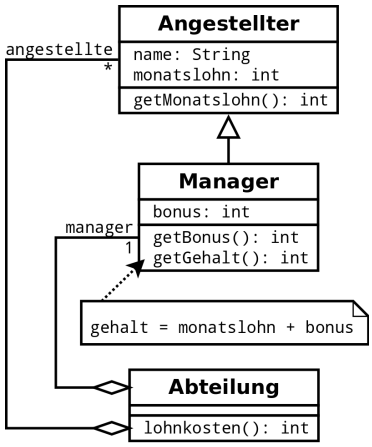


Danach sind ein «Mannschafts»-Objekt und die geforderten Spieler zu erzeugen (`new`). Zu guter Letzt sind die Spieler der Mannschaft hinzuzufügen. Alle gesuchten Methoden sollen einfach einen sinnvollen Text ausgeben.

Aufgabe 18.7 «Abteilung»

Implementieren Sie das folgende Klassendiagramm. Die Bemerkung «*gehalt = monatslohn + bonus*» steht hier lediglich als Notiz. Sie gibt an, wie die Logik innerhalb der Methode `getGehalt()` umzusetzen ist.

Testen Sie danach Ihre Klassen mit dem vorgegebenen Programmcode rechts.




```
void test() {
  Manager m;
  Angestellter a1, a2, a3;

  m  = erstelleManager("Meier", 4000, 2000);
  a1 = erstelleAngestellter("Schmid", 3000);
  a2 = erstelleAngestellter("Keller", 4000);
  a3 = erstelleAngestellter("Gerber", 5000);

  Abteilung abt = new Abteilung();
  abt.manager = m;
  abt.angestellte.add(a1);
  abt.angestellte.add(a2);
  abt.angestellte.add(a3);

  System.out.println("Total der Lohnkosten: "
                     + abt.lohnkosten());
}

Manager erstelleManager
  (String name, int monatslohn, int bonus){

  Manager m    = new Manager();
  m.name       = name;
  m.monatslohn = monatslohn;
  m.bonus      = bonus;
  return m;
}

Angestellter erstelleAngestellter
  (String name, int monatslohn) {

  Angestellter a = new Angestellter();
  a.name       = name;
  a.monatslohn = monatslohn;
  return a;
}
```

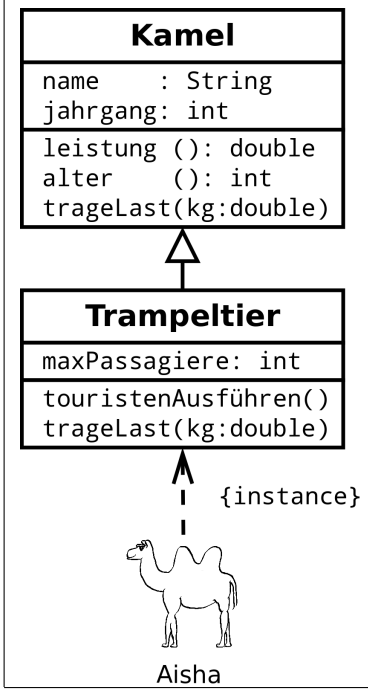
19 | Spezialisierung

«*Spezialisieren: Energie zuführen → fressen → essen → dinieren*»

Manchmal will man den Code nicht einfach erweitern (s. Kap. 18 auf Seite 83), sondern das Verhalten von bestehendem Code auch verändern.

Das **Überschreiben** (Overriding) von Methoden hat den Zweck, spezialisierteren Formen (Subklassen) auch spezialisierteres Verhalten zu verleihen. Oder anders ausgedrückt: Wenn sich eine Teilmenge anders verhält, so bietet sich das Überschreiben an.

Zum Beispiel könnte das Trampeltier die Methode `trageLast(double kg)` spezialisieren und dabei darauf beharren, das dieses Vorhaben nur zwischen den Höckern geschehen darf:

Klassen	Aufruf
	


```
Kamel       ka    = new Kamel();
Trampeltier aisha = new Trampeltier();

ka.trageLast(250);
  // -> "Uff, Uff, 250kg sind viel"

aisha.trageLast(120);
  // -> "Nur zwischen den Hoeckern!"
```

In ersten Objekt basierten Sprachen handelte es sich bei der Vererbung zunächst nur um eine einfache Erweiterung von Datenstrukturen. Anderen Sprachen wurden daraufhin erweitert, dass sie die Möglichkeit hatten, sich um Funktionalitäten zu Erweitern (Methoden). Danach kamen das Verdecken von Attributen und das Überschreiben von Operationen hinzu (z. B. JAVA); weiter hinzugekommen sind z. B. „mix-ins“ (z. B. Ruby).

Klassen können Spezialisierungen anderer Klassen darstellen. Dabei „erben“ bzw. übernehmen sie die Eigenschaften ihrer übergeordneten Klassen. Diese untergeordneten Klassen können neue Eigenschaften und Methoden aufweisen oder die Eigenschaften der Vaterklassen bedarfsweise spezialisieren (mittels Überschreiben und Erweitern), aber sie können solche Eigenschaften nicht eliminieren.

Beispiel 19.1. Student 2

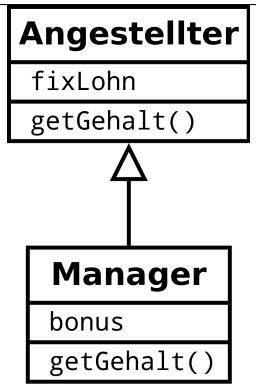
Auch unser Student aus dem Kapitel der elementaren Vererbung (s. Kap. 18.1.3 auf Seite 87) kann sich nun spezialisieren. Statt wie bisher eine `printMatrikelnummer()`-Funktion einzuführen, wird die bestehenden `print()`-Methode in der Subklasse verändert:

```
Public class Student extends Person {
  private String matrikelnummer;
  public void print() { // override print() from Person
    System.out.println("Nachname:␣␣␣␣␣␣␣" + name           );
    System.out.println("Vorname:␣␣␣␣␣␣␣␣" + vorname        );
    System.out.println("Matrikelnummer:␣" + matrikelnummer );
  }
}

// Aufruf
Person  p = new Person();
Student s = new Student();

p.print();
s.print();
```

Beispiel 19.2. `getGehalt()`

Ein weiteres Beispiel befasst sich mit dem Monatslohn von Angestellten. Auch wenn ein Manager in der Firmenhierarchie meistens höher gestellt ist als ein einfacher Angestellter, so ist doch jeder Manager selbst angestellt und steht im Sinne der Vererbung «unterhalb» des Angestellten:

Klassendiagramm	Rückgabe von `getGehalt()`	Spezialisierung

	`fixLohn`	`Angestellter a;`
		`a = new Angestellter();`
		`a.getGehalt();`
		`[fixLohn]`

	`fixLohn+bonus`	`Manager m;`
		`m = new Manager();`
		`m.getGehalt();`
		`[fixLohn+bonus]`

19.1 Aufruf der Superklassenmethoden

Um auf Methoden der Überklasse zuzugreifen, gibt es in JAVA eine Methode, die explizit die Methoden der Superklasse aufruft. Diese spezielle Referenzvariable heißt in JAVA `super`. Betrachten wir das folgende Beispiel:

Beispiel 19.3. Trampeltier

```
class Kamel {
  void essen() {
    print("mampf, mampf.");
  }
}
...
class Trampeltier extends Kamel {
  void essen() {
    super.essen();
    hoeckerZweiFuellen();
  }
}
```

Beispiel 19.4. Manager

Das funktioniert auch mit Funktionsresultaten, wie z. B. im Beispiel `getGehalt()`. Die folgende Implementation ist dem obigen Beispiel «Manager» vorzuziehen, denn es berücksichtigt spätere Änderungen in der Klasse Angestellter:

```
int getGehalt() {
  return super.getGehalt() + bonus;
}
```

19.2 Konstruktoren: 3. `super`

Ganz analog zum Aufruf von Superklassen-Methoden existiert die Möglichkeit innerhalb von **Konstruktoren** die Konstruktormethode der Superklasse explizit aufzurufen. In JAVA wird der Konstruktor bekanntlich nicht mit einem Funktionsnamen aufgerufen, sondern mit dem `new`-Operator. Daher existiert auch die spezielle Syntax `super();`. Damit wird der Konstruktor der Superklasse aufgerufen.

Beispiel 19.5. Person

```
public class Person {

  String name   ;
  String vorname;

  public Person(String name, String vorname) {
    this.name    = name;
    this.vorname = vorname;
  }

} // end of class Person
```

Dies wird dann in der Subklasse wiederverwendet:

```
public class Student extends Person{

  String matrikelnummer;

  public Student(String name, String vorname,
                 String matrikelnummer) {
    super(name, vorname);
    this.matrikelnummer = matrikelnummer;
  }

} // end of class Student
```

19.2.1 Default Konstruktor

Wenn ich als Programmierer keinen Konstruktor anfüge, so wird immer der folgende Default-Konstruktor erstellt:

```
public KlassName() {
   super();
}
```

Dabei bedeutet der `super()`-Aufruf, dass der (parameterlose) Konstruktor der Überklasse (Superclass) aufgerufen wird. So wird garantiert, dass der Konstruktor-Code der Klasse `Object` immer durchlaufen wird.

19.2.2 `super()`

Wenn ich selbst einen Konstruktor erstelle, den Aufruf zum Superkonstruktor aber nicht einfüge, so übernimmt dies der Compiler für mich. Im folgenden Konstruktor könnte `super()` auch weggelassen werden, denn der Compiler fügt diese Codezeile bei Konstruktoren automatisch ein.

```
public Angestellter(String ahvNr) throws NumberFormatException {
  super();
  checkPruefziffer(ahvNr);
  this.ahvNr = ahvNr;
}
```

Ein Aufruf von `super()` muss (wie auch `this()`) immer an erster Stelle im Konstruktor-Body stehen. So wird garantiert, dass der Konstruktor der Klasse `Object` stets aufgerufen wird.

19.3 Aufgaben zur Spezialisierung

Aufgabe 19.1 «Student»

Implementieren Sie die Subklasse `Student` aus dem Beispiel (s. Kap. 19.1 auf Seite 96) mit der überschriebenen Methode `print()` und dem überschriebenen Konstruktor. Verwenden Sie beide Male den `super()`-Aufruf, um den Code der Superklasse wieder zu verwenden. Vergleiche dazu die zugehörigen Kapitel: a) Aufruf von Methoden aus Superklassen (s. Kap. 19.1 auf Seite 97) und b) Aufruf des Konstruktors der Superklasse (s. Kap. 19.2 auf Seite 98).

Aufgabe 19.2 «get Gehalt»

Implementieren Sie die Klassen Angestellter und Manager aus dem Beispiel „Angestellter, Lohn und Manager“ (S. Bsp. 19.2 auf Seite 96). Richten Sie Ihr Augenmerk speziell auf die Methode `getGehalt()`.

Aufgabe 19.3 «Buch»

Schreiben Sie eine Klasse `Buch` mit einem Attribut `seitenzahl` und den zwei Methoden `zeigeAnzahlSeiten()` und `setzeAnzahlSeiten(int s)`.

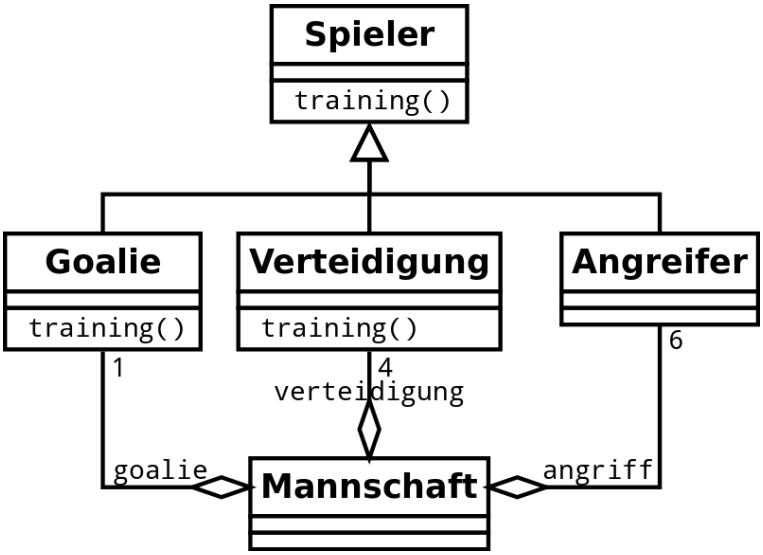
Schreiben Sie eine Subklasse `Nachschlagewerk` mit einem zusätzlichen Attribut für die Seitenzahlen in der Einleitung (`seitenzahlEinleitung`) und einer neuen Methode um diese einleitenden Seitenzahlen zu setzen: `setzteEinleitungsSeitenzahl()`.

Die Methode `zeigeAnzahlSeiten()` soll nun das Total aller Seiten zurückgeben.

Aufgabe 19.4 «Fußball»

Implementieren Sie die Klassen Spieler, Goalie, Angreifer und Verteidigung aus dem folgenden Diagramm. Dabei wird die Methode `training()` beim Goalie und beim Verteidiger so überschrieben, dass ein anderer Text als beim generellen Spieler ausgegeben wird. In der Klasse Mannschaft ist ein Attribut auf den Goalie zu setzen; zudem sind zwei Arrays mit 4 bzw. 6 Elementen auf die Verteidigung bzw. auf den Angreifer zu erstellen. Beim `training()` des Spielers soll `"Jogging"` auf der Konsole ausgegeben werden. Bei der Verteidigung bzw. beim Goalie soll `"Kniebeugen"` bzw. `"Hochsprung"` angezeigt werden.

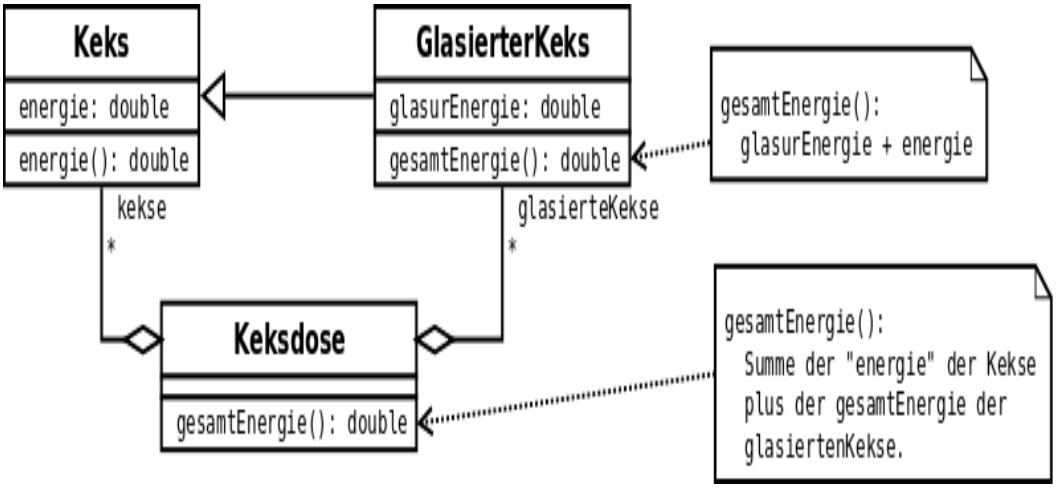
Diagramm:



Zusatzaufgabe: Implementieren Sie in der Klasse `Mannschaft` zusätzlich eine Methode Namens `training()`. Hierzu verwenden Sie einfach die Delegation auf die Klassen `Goalie`, `Verteidigung` und `Angreifer`.

Aufgabe 19.5 «Keksdose»

Implementieren Sie das folgende Klassendiagramm:



Testen Sie dieses mit dem folgenden Hauptprogramm `Konditorei`:

```
Keksdose dose = new Keksdose();
Keks            k1,  k2     ;
GlasierterKeks gk1, gk2, gk3;

k1  = new Keks();
k2  = new Keks();
gk1 = new GlasierterKeks();
gk2 = new GlasierterKeks();
gk3 = new GlasierterKeks();

 k1.energie        = 300;
 k2.energie        = 400;
gk1.energie        = 200;
gk1.glasurEnergie =  66;
gk2.energie        = 300;
gk2.glasurEnergie =  77;
gk3.energie        = 400;
gk3.glasurEnergie =  88;

dose.kekse.add(k1);
dose.kekse.add(k2);
dose.glasierteKekse.add(gk1);
dose.glasierteKekse.add(gk2);
dose.glasierteKekse.add(gk3);

System.out.println("Total␣Energie␣in␣der␣Dose:");
System.out.println("" + dose.gesamtEnergie());
```

«Keine Angst vor Fremdwörtern. Polymorphismus oder das Liskov[26]-Substitutionsprinzip»

Die elementare Vererbung für sich allein ist bereits ein sehr brauchbares Konstrukt. Es erlaubt einmal mehr, Programmcode mehrfach einzusetzen, ohne diesen zu kopieren und ist ein weiterer Schritt in Richtung Wartbarkeit. Aber erst im Zusammenhang mit der Möglichkeit des Polymorphismus wird die Vererbung zu einem mächtigen Werkzeug in der Softwareentwicklung.

Doch zuerst einmal nehmen wir die Angst vor Fremdwörtern:

- Poly-: Griechisch für «viel» oder «viele»
- -morphismus: Griechisch für «-gestaltig», «-förmig».

Somit bedeutet Polymorphismus (oder auch Polymorphie) nichts anderes als «Vielgestaltigkeit». Für (Referenz)variable bedeutet dies, dass sie Werte von verschiedenen Datentypen annehmen kann.

Das einfachste Beispiel kennen wir bereits von den Standardtypen, wo einer Variablen vom Typ `double` auch eine ganze Zahl (Typ `integer`) zugewiesen werden kann (`double x = 5;`).

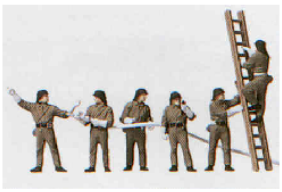
Polymorphie heißt auch, dass eine Operation sich in (unterschiedlichen Klassen) unterschiedlich verhalten kann (nicht muss). Wir unterscheiden...

[26]Barbara Liskov (* 1939) `http://www.pmg.csail.mit.edu/~liskov/`

20.1 ... drei Arten des Polymorphismus

- Polymorphie von Operationen in verschiedenen, nicht voneinander abgeleiteten Klassen. Dies stellt kein Problem dar, da die Klassen verschiedene Namensräume aufweisen.




- Statischer Polymorphismus (s. Kap. 20.2 auf Seite 107): Methoden derselben Klasse mit gleichem Namen, aber anderen Parametern können bereits bei der Kompilationszeit (also statisch) behandelt werden. Es sind für die (virtuelle) Maschine verschiedene Prozeduren. Unter der Signatur verstehen wir den Namen einer Methode **zusammen** mit seiner Parameterliste. In einem Namensraum (Klasse) dürfen zwei Methoden niemals dieselbe Signatur aufweisen. Haben jedoch zwei Methoden eine andere Parameterliste aber denselben Namen, so sind das für den Compiler zwei komplett verschiedene Methoden.

  Beispiel 20.1. Dreiecksfläche

  Für einen modernen Compiler bezeichnen die beiden folgenden Deklarationen zwei komplett unterschiedliche Methoden obschon sie denselben Namen tragen:

  - `dreiecksFlaeche(Point A, Point B, Point C)`
  - `dreiecksFlaeche(double seiteA, double seiteB, double seiteC)`

- Am spannendsten, aber auch am vielseitigsten, ist jedoch die dynamische Polymorphie von Methoden in Objekten derselben Vererbungshierarchie (s. Kap. 20.3 auf Seite 110): Methoden mit demselben Namen und derselben Parameterliste. Diese Methoden kennen wir bereits aus Kap. 19 auf Seite 95.

20.2 Statischer Polymorphismus

Unter statischem Polymorphismus (auch Überladen oder **Overloading** genannt) versteht man folgendes:

Wenn sich zwei Methoden derselben Klasse im Namen nicht unterscheiden, jedoch eine andere Parameterliste aufweisen, so sind es zwei verschiedene Methoden.

Haben zwei Methoden verschiedene Signaturen, so sind das für den Compiler (und auch für die virtuelle Maschine) zwei komplett verschiedene Methoden (als ob sie zwei verschiedene Namen hätten).

Beispiel 20.2. Statischer Polymorphismus

Die folgenden drei Methoden sind für den Compiler völlig verschiedene Methoden.

```
public double dreiecksFlaeche(double a, double b, double c) {...}
public float  dreiecksFlaeche(float  a, float  b, float  c) {...}
public double dreiecksFlaeche(Point pA, Point pB, Point pC) {...}
```

Einzig beim Aufruf muss sich der Compiler entscheiden, welche Methode er wählen soll. In der Regel ist das aber klar. Im Gegensatz dazu haben die folgenden Methoden dieselbe Signatur und dürfen somit in derselben Klasse **nicht** verwendet werden (bzw. wenn diese in Subklassen verwendet werden, spricht man von *Overriding*).

```
void x(int a);
void x(int b); // FEHLER
```

Bemerkung 20.1. In einigen Sprachen (wie z. B. `PHP`) gibt es die Möglichkeit, Parameter mit Standardwerten zu deklarieren. Im Zusammenhang mit einer variablen Anzahl von Parametern kann dies in gewissem Sinne auch als *statischer Polymorphismus* verwendet werden:

```
function transport($anzahlKamele, $ziel = "Bagdad")
{
    return "Es␣ziehen␣" . $anzahlKamele "␣nach␣" . $ziel . ".\n";
}

// Aufruf:
transport(100);
transport(42, "Memphis");
```

20.2.1 Konstruktoren: 4. Overloading (`this`)

So wie wir beliebige Methoden überlagern (overloading) können, so können wir dies auch mit den Konstruktoren anfangen. Dies nennen wir dann sinnvollerweise

«Constructor-Overloading».

Als Vorzeigebeispiel nehmen wir nochmals die Klasse `Color`. Ein solches Farb-Objekt kann auf verschiedene Arten initialisiert werden. Für die Weiterverwendung ist lediglich wichtig, dass die drei Attribute für «rot», «grün» und «blau» stets als `int`-Werte vorhanden und gültig sind.

Das Anbieten von mehreren Konstruktoren kann die Arbeit mit dem Objekt stark erleichtern.

Der wichtigste Konstruktor soll einfach die drei Werte im Bereich 0..255 als Integer (`int`) entgegennehmen. Mit der «Exception» wird gleich noch ein Konsistenztest durchgeführt. Die anderen Konstruktoren rufen jeweils mittels `this()` den Hauptkonstruktor auf.

```
public Color(int r, int g, int b) throws
java.lang.NumberFormatException {
   if(r < 0 || r > 255 || g < 0 || g > 255 || b < 0 || b > 255) {
       throw new NumberFormatException("rgb␣Werte:␣0..255");
   }
   this.r = r;
   this.g = g;
   this.b = b;
}

public Color(int rgb) throws NumberFormatException {
    this((rgb >> 16) & 0xFF, (rgb >> 8) & 0xFF, rgb & 0xFF);
}

public Color(String hexString) throws NumberFormatException {
   this(Integer.parseInt(hexString.substring(0, 2), 16),
        Integer.parseInt(hexString.substring(2, 4), 16),
        Integer.parseInt(hexString.substring(4, 6), 16));
}

public Color() {
   this(128, 128, 128);
}
```

Beispiel 20.3. Person

```
public Person() {
  this.name    = einlesen("Name"   );
  this.vorname = einlesen("Vorname");
}

public Person(String ahvNummer) throws NumberFormatException {
  this(); // Aufrufen des parameterlosen Konstruktors der Klasse
  Person.checkPruefziffer(ahvNummer);
  this.ahvNummer = ahvNummer;
}
```

Bemerkung: Ein Aufruf von `this()` muss immer an erster Stelle im Konstruktor-Body stehen!

20.3 Dynamischer Polymorphismus

Vereinfacht sprechen wir von dynamischem Polymorphismus, wenn wir einer Referenzvariable einer Klasse auch Objekte von Subklassen, also Spezialisierungen, dieser Klasse zuordnen können. So kann eine Variable `p` vom Typ `Person` problemlos ein Objekt vom Typ `Student` zugewiesen werden:

```
Person p;
p = new Student(); // Student extends Person
```

20.4 Substitutionsprinzip

Die präzise Forderung des Polymorphismus (also der Anwendung des Substitutionsprinzips) liest sich etwas holprig:

Definition 20.1 Wenn es für jedes Objekt o_S vom Typ S ein Objekt o_T vom Typ T gibt, sodass für alle Programme P, die auf der Basis des Typs T definiert wurden, das Verhalten von P unverändert bleibt, wenn o_S an Stelle von o_T eingesetzt wird, dann stellt S ein Subtyp von T dar.

Im Falle der Erweiterung kann ein Objekt einer abgeleiteten Klasse (`S extends T`) problemlos an die Stelle eines Objektes einer Basisklasse treten.

Was bedeutet das nun? Als Beispiel dient ein Programm, das mit Karawanen umgehen kann. Eine Karawane beinhaltet verschiedene Kamele. Diese können hinzugefügt, verkauft, beladen und entladen werden. Nun setze ich für diese Kamele hingegen im konkreten Fall Subtypen ein. Hier also Trampeltiere, Dromedare und Lamas. Wenn sich nun mein Programm immer noch korrekt verhält, wenn ich jedes «Kamel» durch «Trampeltier», «Dromedar» oder «Lama» ersetze, so sprechen wir von einem polymorphen Programm, das dem Substitutionsprinzip gerecht wird.

20.4.1 Typenbindung

Eine Variable (Attribut, Parameter, lokale Variable, ...) sollte wenn möglich – zur Stabilität, Robustheit, Einarbeitungszeit und Wartbarkeit des Codes – nur Werte aus einem bestimmten Datentyp zulassen. Wir nennen dieses Konzept das Konzept der *Variablen- oder der Typenbindung*. Variable werden an ihre Datentypen gebunden. Es kann nun durchaus sinnvoll sein, dieses Konzept in seiner Strenge etwas aufzuweichen: So kann es von Nutzen sein, einer Variable vom Datentyp `Person` andere Objekte zuzuweisen.

Anmerkung: Polymorphismus widerspricht im Grunde der strengen Typenprüfung von Variablen.

Natürlich sollten wir nicht irgendein Objekt zuweisen können! So macht es keinen Sinn, einer Variable vom Datentyp `Person` ein Objekt der Klasse `Bankkonto` oder der `Kundenliste` zuzuweisen.

Wo kann eine solche Zuweisung jedoch sinnvoll sein?

Sobald ein Objekt von der entsprechenden Klasse abgeleitet ist, also sogenannte *sub-Objekte* von `Person` wie z. B. `Kunde` oder `Angestellter`, sollte es sehr wohl einer Variable vom Typ `Person` zugewiesen werden können:

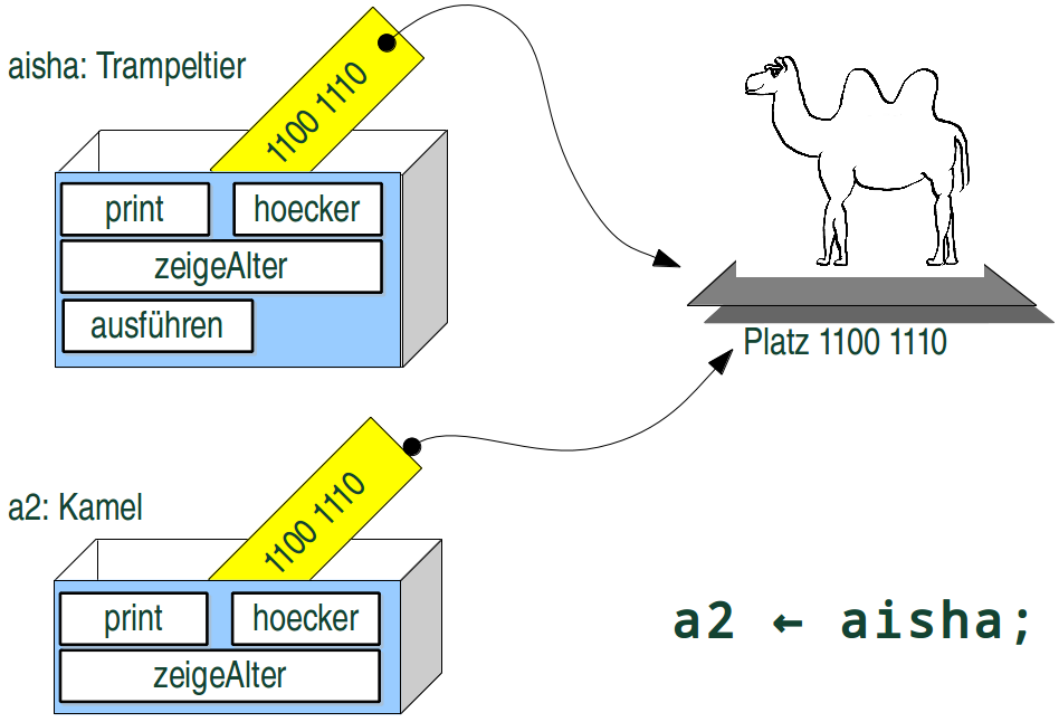
```
// Variable
Person pers;
pers = new Manager(); // Sofern "Manager" von Person abgeleitet
                      // wurde, also eine spezielle Person ist.
```

Somit kann die Variable `pers` aus obigem Beispiel viele Formen annehmen („viele Formen“ eben: Poly - Morphos).

20.5 Beispiel 1: Kamele

Zunächst etwas JAVA-Code:

```
Trampeltier aisha; // (Trampeltier extends Kamel)
Kamel       a2   ;

aisha = new Trampeltier();
a2    = aisha             ;

a2.zeigeAlter();
a2.print()     ;
```



Obiger Code (`a2 = aisha`) funktioniert, sofern `Trampeltier` von `Kamel` abgeleitet wurde, wenn also `Trampeltier` ein spezielles `Kamel` ist. Oder mit anderen Worten ausgedrückt: Ein Trampeltier (das ja auch ein Kamel ist), muss jederzeit jeder Variablen des Datentyps Kamel (genauer: jeder Überklasse von Trampeltier) zugewiesen werden können.

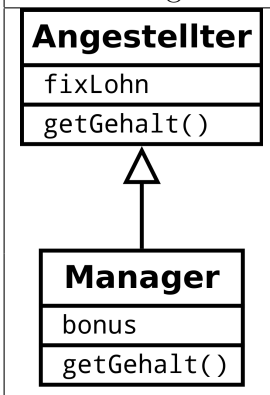
Im Falle der Spezialisierung (Erweiterung) kann ein Objekt einer abgeleiteten Klasse problemlos an die Stelle eines Objektes einer Basisklasse treten.

Im obigen Beispiel enthält die «Kiste» `a2:Kamel` eine Referenz auf dasselbe Objekt wie die «Kiste» `aisha:Trampeltier`. Beim Verwenden von `a2` werden die nicht benötigten Attribute und Methoden einfach ausgeblendet.

20.6 Beispiel 2: Angestellte

Die folgende Tabelle ergänzt das Beispiel aus Kap. 19.2 auf Seite 96. Beachtenswert ist der Code ganz rechts oben, wo der `Angestellte` `a` auch den Bonus erhält obschon es sich **nicht** um eine `Manager`-Variable handelt! Dies ist jedoch wichtig zum Verständnis: Es geht beim Polymorphismus nicht um die Referenzvariable, sondern um das darunterliegende Objekt; dieses ist aber in unserem Beispiel ein `Manager`.

Klassendiagramm	Spezialisierung	Polymorphismus

	`Angestellter a;`
	`a = new Angestellter();`
	`a.getGehalt()`
	`[fixLohn]`
		`Angestellter a;`
		`a = new Manager();`
		`a.getGehalt()`
		`  [fixLohn+bonus]`

	`Manager m;`
	`m = new Manager();`
	`m.getGehalt()`
	`[fixLohn+bonus]`
		`Manager m;`
		`m = new Angestellter();`
		`  FEHLER`
		`m.getGehalt()`
		`  FEHLER`

Einer Variablen vom Typ `Manager` kann hingegen kein allgemeines `Angestellter`-Objekt zugewiesen werden.

Streichen Sie bitte das letzte Kästchen (ganz rechts unten) rot durch.

20.7 Polymorphe Anwendungen

Sehr interessant ist der Polymorphismus im Zusammenhang mit Variablen, bei denen der Programmierer während der Entwicklung noch nicht weiß, welche Objekte schlussendlich verwendet werden.

20.7.1 Polymorphe Parameter

Polymorphe Parameter sind Methodenparameter, die bei der Laufzeit verschiedene Objekte annehmen können. Gleich ein Beispiel:

```
class Karawane {
  void addKamel(Kamel k) {
    ...
    kapazitaet = kapazitaet + k.getKapazitaet();
  }
}


Trampeltier  tt = new Trampeltier();
Dromedar     dr = new Dromedar()   ;

karawane.addKamel(tt);
karawane.addKamel(dr);
```

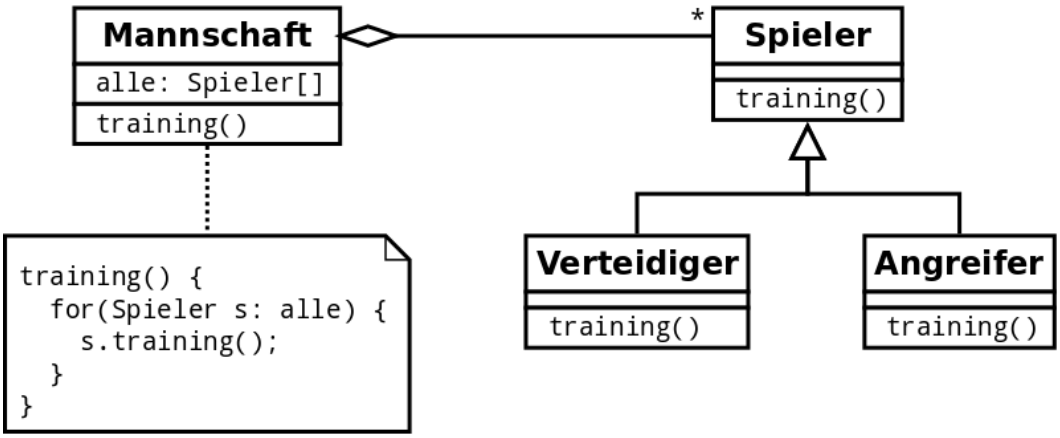
Was geschieht hier? Jedes Kamel hat eine Kapazität; mit anderen Worten eine Last, die es tragen kann. Die Karawane hat als Kapazität die Summe aller Kapazitäten der hinzugefügten Kamele. Wenn nun das Trampeltier und das Dromedar beide Kamele sind, so haben beide die Methode `getKapazitaet()`. Diese Methode wird nun in der Karawane auf der Variablen `k` aufgerufen und zwar unabhängig davon, ob es sich um ein Trampeltier, ein Dromedar oder ein unspezifisches Kamel handelt. Ebenso ist es unabhängig davon, ob das Dromedar oder das Trampeltier die Methode `getKapazitaet()` spezialisiert (sprich überschrieben) haben. Somit sprechen wir bei `k` von einem **polymorphen Parameter**.

20.7.2 Polymorphe Collections

Ebenso spannend wie die polymorphen Parameter sind polymorphe Collections. Gerade das Beispiel der Karawane kann wieder als Musterbeispiel herangezogen werden:

```
List<Kamel> karawane;

// Karawane zusammenstellen
karawane.add( new Trampeltier() );
karawane.add( new Dromedar()    );
karawane.add( new Dromedar()    );
karawane.add( new Lama()        );

// Karawane messen
double kapazitaet = 0;
for(Kamel k : karawane) {
  kapazitaet = kapazitaet + k.getKapazitaet();
}
```

Als zweites Beispiel einer polymorphen Collection dient wieder unsere Fußballmannschaft:



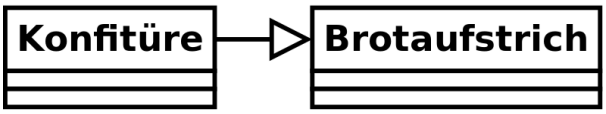
Die Mannschaft delegiert die Methode `training()` an alle Spieler der Sammlung[27] `alle`. Spannend ist nun, dass es sich bei den Spielern auch um Spezialisierungen wie Angreifer oder Verteidiger handeln kann, die sich ihrerseits in ihrer eigenen Methode `training()` speziell verhalten. So kann ein Angreifer sich im `dauerlauf()` üben, während ein Verteidiger eher mal ein paar `kniebeugen()` exerziert.

[27] Hierbei kann es sich z. B. um einen Array oder eine `List` handeln.

20.8 Aufgaben zum Polymorphismus

Aufgabe 20.1 «Brotaufstrich»

Gegeben ist die Klasse Brotaufstrich als Überklasse über die Subklasse Konfitüre.



Es sei `k` eine Referenzvariable des Typs Konfitüre und `b` eine Brotaufstrich-Referenzvariable. Welche der beiden folgenden Zuweisungen ist möglich? Erklären Sie, warum die andere Zuweisung problematisch ist bzw. einen Compilerfehler erzeugen sollte:

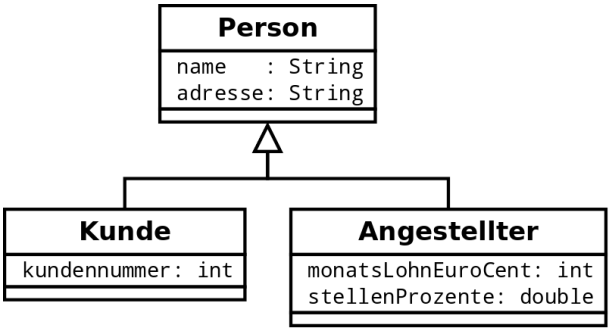
- `k = b`
- `b = k`

Machen Sie dieselben Überlegungen für die folgenden Referenzvariablen (der Basistyp ist jeweils in Klammern angegeben und die Vererbung sollte klar sein, ansonsten hilft Wikipedia): Welche der folgenden Zuweisungen sind nicht gestattet?

- `b` [Bürosoftware] := `t` [Textverarbeitung]
- `u` [Unix-Derivat] := `b` [Betriebssystem]
- `d` [Droge] := `w` [Wein]
- `k` [Kamel] := `d` [Dromedar]
- `fi` [Fischfilet] := `fl` [Fleischstück]
- `m` [Mensch] := `t` [Trockennasenaffe]

Aufgabe 20.2 «verschiedene Personen»

Gegeben ist folgendes Klassendiagramm.



Ergänzen Sie das Klassendiagramm (in jeder Klasse) um eine Methode `ausgeben()`, die alle Attribute einer Person auf die Konsole ausgeben kann. Implementieren Sie die drei Klassen. Schreiben Sie danach ein Programm, das in einem Array `personen` (des Basistyps Person) sechs Personen speichern kann. Füllen Sie zwei unspezifische Personen, zwei Kunden und zwei Angestellte dem Array hinzu. Geben Sie danach in der folgenden Schleife alle Personen des Arrays aus:

```
for(Person p: personen) {
  p.ausgeben();
}
```

Beachten Sie den Nutzen des Polymorphismus, der uns eine Selektion nach Datentypen komplett erspart. Somit wird die Kontrollstruktur `switch/case` häufig überflüssig.

Bemerkung 20.2. Anstelle eines Arrays (`Person[]`) kann alternativ eine Liste verwendet werden (`List<Person>`). Die Schleife ändert sich dadurch nicht.

Aufgabe 20.3 «Fußball»

Implementieren Sie die Fußballmannschaft mit polymorpher Collection von Seite 115.

Aufgabe 20.4 «Touristengruppe»

Modellieren Sie eine Touristengruppe mit sechs Kamelen; diese sind ein Leittier (Lama), zwei Lasttiere (Dromedare) und drei Touristentiere (Trampeltiere).

Eine Methode `trageLast()` in der Überklasse `Kamel` soll einfach den Text «Keuch, Keuch!» ausgeben.

Die Trampeltiere laden vor dem Keuchen die Last auf. Überschreiben Sie dazu die Methode `trageLast()`, indem Sie zunächst den Text «Last aufladen» ausgeben und danach die Methode in der Klasse Kamel aufrufen (`super.trageLast()`).

Das Lama spuckt vor und nach dem Keuchen (geben Sie das mit dem Text «flatsch» aus).

Das Dromedar keucht hingegen nicht und meldet sich auch sonst nicht zu Wort.

Zeichnen Sie zunächst das UML-Klassendiagramm.

Befüllen Sie sodann in der Klasse «Touristengruppe» eine `ArrayList` mit den sechs Kamelen und lassen Sie die sechs Kamele in einer Programmschleife je ihre Last tragen.

21 | Abstrakte Klassen und Schnittstellen

«Das gibt's doch gar nicht!»

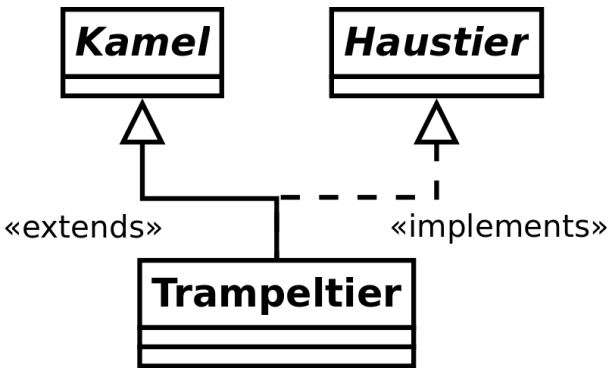
Frei nach P. Picasso

21.1 Abstrakte Klassen

Klassen, von denen keine konkreten Exemplare erzeugt werden, d. h. von denen es grundsätzlich keine Objekte geben wird, bezeichnet man als abstrakte Klassen (oder «building blocks»[28]).

Notation: Im UML-Diagramm werden abstrakte Klassen und Schnittstellen mit kursiver Schrift dargestellt.

UML-Notation von abstrakten Klassen (hier Kamel) und Schnittstellen (hier Haustier):



Eigenschaften abstrakter Klassen:

- Eine abstrakte Klasse ist eine Klasse, die einen Allgemeinbegriff darstellt (Fahrzeug bzw. Pflanze), der für die Implementierung einer Spezifikation bedarf (Fahrrad, Auto, .. bzw. Baum, Strauch, Zimmerpflanze, ...).
- Abstrakte Klassen sind immer Oberklassen (Superklassen).
- Abstrakte Klassen ohne potentielle Unterklassen ergeben keinen Sinn.

[28] Das Konzept «Building Blocks» stammt von Bjarne Stroustrup, dem Entwickler der Programmiersprache C++.

Wann verwenden wir abstrakte Klassen?

- Abstrakte Klassen werden dann eingesetzt, wenn verschiedene Kategorien sich im konkreten nicht überschneiden. Beispiele:
  - abstrakte Klassifizierung **Himmelskörper**, konkrete Kategorien Stern, Planet, Mond, ...
  - abstrakte Klassifizierung **Zootier**, konkrete Kategorien Elefant, Tiger, Trampeltier
  - abstrakte Klassifizierung **GUI-Komponente**, konkrete Kategorie Push-Button, Scrollbar, Eingabefeld, ...
- Gewisse Attribute oder Methoden können in ähnlichen Klassen gemeinsam vorhanden sein. Zeigt es sich, dass es eine abstrakte Überkategorie gibt, welche auch diese Attribute und Methoden haben muss und dass es von dieser Überklasse jedoch nie Objekte geben wird, so kann eine abstrakte Überklasse gebildet werden. Beispiel:

  Gegeben sind in einer Bibliothek «Buch», «DVD», «CD» und «Zeitschrift». Man erkennt, dass alle Objekte eine Identifikationsnummer und einen Ausleihestatus haben. Gesucht ist nun eine abstrakte Klasse, die als Überklasse hergenommen werden könnte. Hinhalten kann z. B. das «Medium».

Beispiel 21.1. Abstraktes Kamel

In einem Zoo werden Kamele gehalten. Als da sind: Aisha, das Trampeltier, Pedro, das Lama und Ali, das Dromedar. Für die Verwaltung werden dann auch die Klassen Trampeltier, Lama und Dromedar unterhalten. Zur Vereinfachung wird eine abstrakte Überklasse (Superklasse) gesucht, die alle Gemeinsamkeiten der drei genannten Klassen zusammenfasst. Die Klasse Kamel bietet sich an.

Direkt von der Klasse Kamel wird es keine Objekte geben. Jedoch immer von einer der drei genannten Subklassen. Um zu verhindern, dass von der abstrakten Klasse jemals Instanzen (Objekte) erzeugt werden, kann in JAVA eine Klasse als `abstract` markiert werden. Dennoch können alle konkreten Attribute und gemeinsamen Methoden in einer solchen abstrakten Superklasse stehen.

Abstrakte Klassen sind also Klassen, die Konzepte vertreten, für welche es keine unspezifischen Objekte geben kann. Jede Klasse ist entweder abstrakt oder konkret. Von abstrakten Klassen kann es keine Instanzen geben. Mit anderen Worten: Eine Objekt-Instanzierung mit `new` ist nicht möglich. Nur für konkrete Klassen gibt es Exemplare.

21.2 Anwendungsbeispiele abstrakter Klassen

21.2.1 Beispiel: Grafikprogrammierung

Das folgende Beispiel zeigt eine abstrakte Überklasse über geometrische Figuren in einem Zeichnungsprogramm. Es gibt hier alternativ Rechtecke und Kreise. Eine Figur ist immer entweder Rechteck oder Kreis, nie aber beides gemeinsam. Somit haben wir ausschließende Kategorien und die abstrakte Überklasse nennen wir sinnigerweise `Figur`.

```
abstract public class Figur {
  Point pos;
  public void moveTo(Point newPos) {
      this.pos = newPos;
  }
  public abstract void paint(Graphics g);
}
```

```
public class Rechteck extends Figur {
    int width, height;
    public void paint(Graphics g) {
       g.drawRect(pos.x, pos.y, width, height);
    }
}
```

```
public class Kreis extends Figur {
    float radius;
    public void paint(Graphics g) {
      g.drawOval((int) (pos.x - radius),
                 (int) (pos.y - radius),
                 (int) (2*radius),
                 (int) (2*radius));
    }
}
```

```
public class Main {
  public static void main(String[] args) {
    Figur f1 = new Kreis();    // i. O: Polymorphismus
    Figur f2 = new Rechteck();
    f1.moveTo(new Point(30, 50));
    f1.paint(System.getGraphicsContext());
    f2.moveTo(new Point(30,70));
    f2.paint(System.getGraphicsContext());
 }
}
```

Beachten Sie, dass im Hauptprogramm keine Referenzvariable vom Datentyp `Kreis` bzw. `Rechteck` existieren. Als Referenzvariable dient lediglich der Datentyp der Schnittstelle (hier `Figur`).

Merke

a) Nutzklassen nutzen nur Referenzen auf abstrakte Klassen[29]. Zum Beispiel: `JButton` benutzt ein `java.awt.event.ActionListener`.

b) Gemeinsames Verhalten wird in der Superklasse zusammengefasst. Diese für sich alleine macht jedoch keinen Sinn: `Figur`, `java.lang.Number`, ...

c) Kombination aus a) und b): So enthält z. B. `java.awt.Component` generalisierten Code von Buttons und Panels (z. B. die Methode `contains(Point p)`), kann aber generell eingesetzt werden:
`panel.add(component)`.

[29] Beziehungsweise auf Schnittstellen (s. Kap. 21.3 auf Seite 127)

21.2.2 Beispiel: Code einsparen mit abstrakten Klassen

Das folgende Beispiel stammt aus einer Implementation für eine grafische Benutzerschnittstelle (GUI = graphical user interface). Dabei wurden die Fenstergrößen dynamisch gehalten und die zweitunterste Komponente wird in der Höhe gestreckt, wenn der Benutzer das Fenster in der Höhe verändert. Da die zweitunterste Komponente nicht immer auf derselben Zeile (von oben gezählt) war, wurden zwei Methoden für das Layout bereitgestellt. Jede Methode war dabei in ihrer eigenen Klasse; einmal fürs Editieren und einmal für den Suchdialog:

Editierdialog-Klasse

```
public void setComponent(
  GridBagConstraints c,
  int gridx,
  int gridy,
  JComponent comp)
{
  c.gridx = gridx;
  c.gridy = gridy;
  if(gridy < 3) {
    c.weighty = 0.0;
  } else {
    c.weighty = 1.0;
  }
  add(comp, c);
}
```

Suchdialog Klasse

```
public void setComponent(
  GridBagConstraints c,
  int gridx,
  int gridy,
  JComponent comp)
{
  c.gridx = gridx;
  c.gridy = gridy;
  if(gridy < 7) {
    c.weighty = 0.0;
  } else {
    c.weighty = 1.0;
  }
  add(comp, c);
}
```

Entdecken Sie den Unterschied? Richtig! Einzig die Zahl in der Bedingung hat sich von 3 nach 7 geändert. Diese Methode gehört ausgelagert. Dazu könnte man die obige Methode `setComponent()` mit einem weiteren Parameter (`cntOfStaticLines`) schreiben und diese von beiden Klassen aufrufen. Nichts Neues also.

In obigem Code bietet sich aber eine weitere Möglichkeit mittels Polymorphismus an. In der Basisklasse (Superklasse) wird die Methode implementiert und somit von beiden Klassen geerbt. Dazu würde aber der Parameter fix in der Superklasse integriert sein. Die Lösung bedient sich daher eines gängigen Tricks über eine abstrakte Methode `cntOfStaticLines()`.

Lösung in der **Basisklasse**:

```
public void setComponent(
  GridBagConstraints c     ,
  int                gridx,
  int                gridy,
  JComponent         comp)
{
  c.gridx = gridx;
  c.gridy = gridy;
  if(gridy < cntOfStaticLines()) {
    c.weighty = 0.0;
  } else {
    c.weighty = 1.0;
  }
  add(comp, c);
}
```

Dabei muss in den Subklassen lediglich die Methode `cntOfStaticLines()` überschrieben werden, sofern sich diese vom Standard in der Basisklasse unterscheidet:

Editierdialog-Klasse	Suchdialog-Klasse
`public int cntOfStaticLines() { return 3; }`	`public int cntOfStaticLines() { return 7; }`

Das Ganze hat nun unmittelbar auch mit Polymorphismus zu tun:

Ein Aufruf der Methode `setComponent()` wird die Methode `cntOfStaticLines()` aufrufen. Wenn diese Methode überschrieben wurde, so wird automatisch die überschriebene Methode aufgerufen.

Ein weiterer Vorteil: Die Zahlen 3 bzw. 7 im ursprünglichen Code rufen nach einem Kommentar. Dieser ist durch den sinnvollen Methodennamen vorweggenommen und kann nun mit gutem Gewissen weggelassen werden.

Wenn nun von der neu erstellten Superklasse keine Objekte erstellt werden, so kann diese Klasse als abstrakte Klasse erstellt werden. Somit wird garantiert, dass immer die Editier- oder die Suchdialog-Klasse verwendet wird.

21.3 Schnittstellen (Interfaces)

Schnittstellen sind auch unter den verwandten Begriffen **Vertrag** und **Protokoll** anzutreffen.

Klassen können, wie bereits erwähnt, auch abstrakt sein. Von abstrakten Klassen können keine Objekte erzeugt werden. Abstrakte Klassen können aber Methodendeklarationen beinhalten, die dann von allen Subklassen (zwecks Polymorphismus) implementiert werden müssen. Eine Klasse, die ausschließlich Methodendeklarationen, aber keinerlei Methodendefinitionen (sprich Implementierungen) aufweist, nennt man auch **Schnittstelle**. Schnittstellen können in JAVA mittels eigenen klassenähnlichen Definitionen (`interface`) umgesetzt werden.

Häufige Anwendungen von Schnittstellen sind

- Das Entwickeln im Team
- Verwenden von Callback-Funktionen (sog. Handlern, Hooks oder Listeners)
- Trennung von Spezifikation und Implementierung
- Marker-Interfaces, also Schnittstellen ohne Methoden und ohne Konstanten, die lediglich dazu dienen, zu prüfen, ob ein Objekt in eine Kategorie passt: Beispiele in JAVA: `Cloneable`, `Serializable`
- Verwenden von bestehenden Schnittstellen des API

Schnittstellen werden in JAVA 1:1 in sogenannten Interface-Klassen umgesetzt. Typischerweise sind 3 Klassen im Spiel, nämlich

- Die Schnittstellenklasse (`interface`),
- der Schnittstellenanbieter (`implements ...`) und
- der Schnittstellennutzer (`import ...`).

21.3.1 Schnittstellen sind abstrakte Klassen

Hier eine zweite Definition der Schnittstelle: Eine Schnittstelle in JAVA ist eine total abstrakte Klasse: alle Methoden sind abstrakt; zudem können Schnittstellen nur noch konstante (final) Attribute enthalten.

Abstrakte Klassen können (im Gegensatz zu Interfaces) auch vollständig definierte Methoden — also mit Code — enthalten. Beispiel einer abstrakten Klasse: `GeometrischeFigur`. Diese hat eine abstrakte Methode `draw()` und zugleich eine ganz konkrete Methode `moveTo()`. Eine Schnittstelle darf dies nicht!

Es gibt somit generell zwei Arten von abstrakten Klassen: Schnittstellen (Interfaces) und Building Blocks (= abstract class). Interfaces haben keinerlei Implementation und können so Schnittstellen zu Klassen definieren. Ein Interface ist ein Vertrag zwischen Nutzer (Nutzklasse) und Implementation einer Funktionalität. Interfaces stehen häufig zuoberst in der Klassenhierarchie.

Abstrakte Klassen mit einigen implementierten Methoden stehen häufig in der Mitte in Klassenhierarchien. Sie enthalten gemeinsame Funktionalitäten der Subklassen, sind aber immer noch nicht komplett: Es können keine Objekte generiert werden.

Zusammenfassung

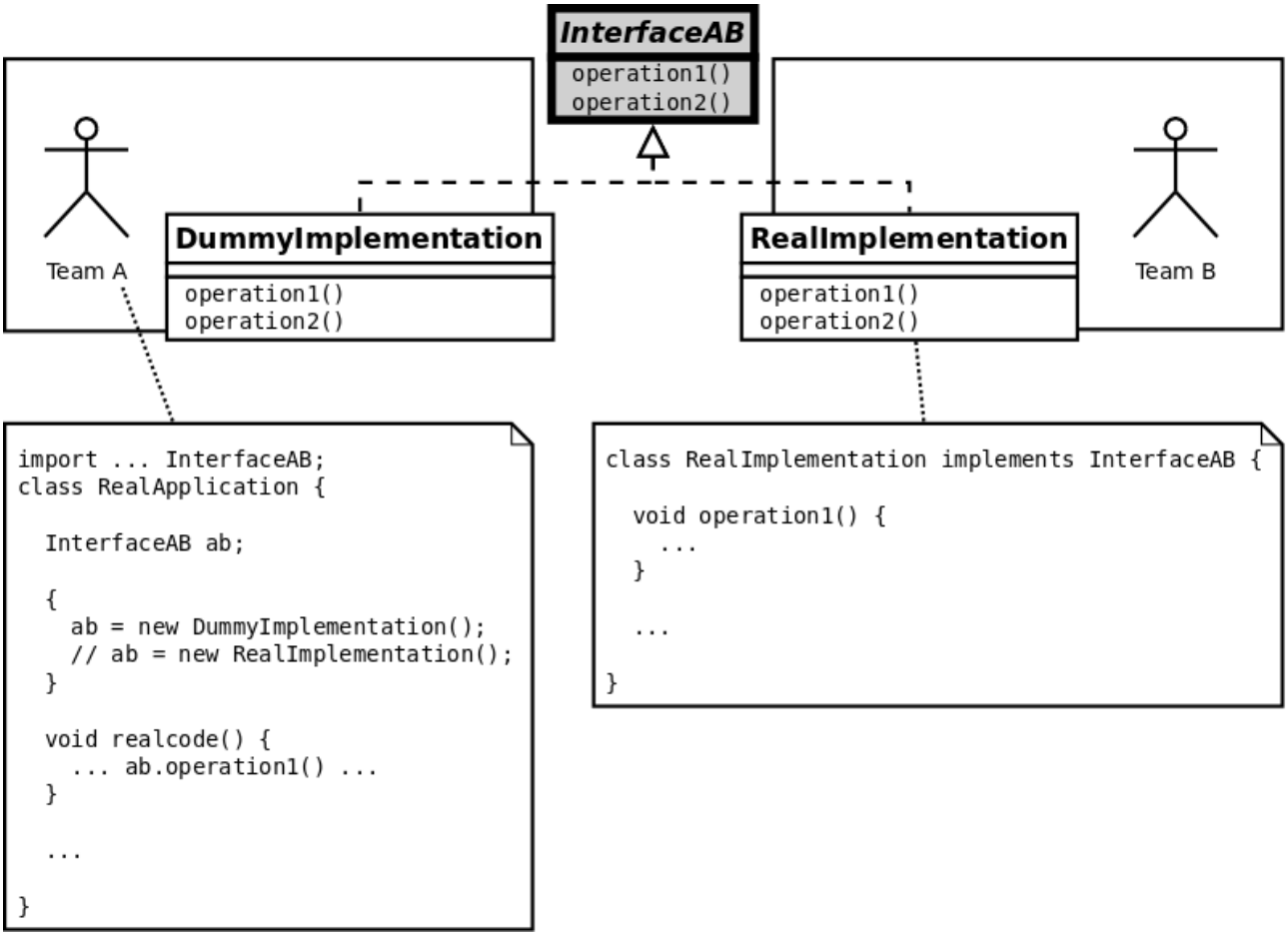
Name	Definition	Alternativer Name	Eigenschaft
interface	keine Funktionalität: Alle Methoden sind nur deklariert, nicht aber definiert.	Schnittstellen	**keine** Objekte erzeugbar
abstract class	einige Funktionalitäten: Einige Methoden enthalten Programmcode (Definition).	Building Blocks	**keine** Objekte erzeugbar
class	alle Funktionen sind definiert.	normale, voll funktionsfähige Klassen	Objekte erzeugbar

Achtung: In JAVA muss eine Klasse bereits mit `abstract` deklariert werden, wenn nur schon eine einzige Methode mit `abstract` deklariert wird. Eine abstrakte Methode in JAVA hat keinen Block nach der Deklaration, auch keinen leeren.

21.4 Anwendungsbeispiele von Schnittstellen

21.4.1 Beispiel: Softwareentwicklung in Teams

Wenn im Team entwickelt wird, so stellt sich häufig das Problem, dass mehrere Entwickler an denselben Programmteilen arbeiten wollen. Damit es hierbei nicht zu allzu großen Problemen kommt, spricht man sich vorher ab und definiert, welche Schnittstellen Team A benötigt und Team B zur Verfügung stellen kann. Team A kann somit mit der Implementation beginnen, bevor Team B die Schnittstelle komplett implementiert hat.



Zuerst werden sich Team A und Team B auf ein Interface (`InterfaceAB`) einigen. Sobald dies geschehen ist, werden die beiden Teams unabhängig voneinander mit der Entwicklung beginnen. Team B muss im Wesentlichen dafür sorgen, dass eine reale funktionsfähige Implementation (`RealImplementation`) vorhanden ist. Team A kann mit der realen Applikation (`RealApplication`) beginnen, und das, noch bevor Team B mit dem Implementieren des Interfaces fertig ist. Sollte Team A nun Code zur Verfügung haben, um erste Tests durchführen zu können, so kann sich Team A selbst eine Attrappe (dummy) zur Verfügung stellen (in der Grafik: `DummyImplementation`). Diese Attrappe implementiert nur die für die Tests wesentlichen Methoden.

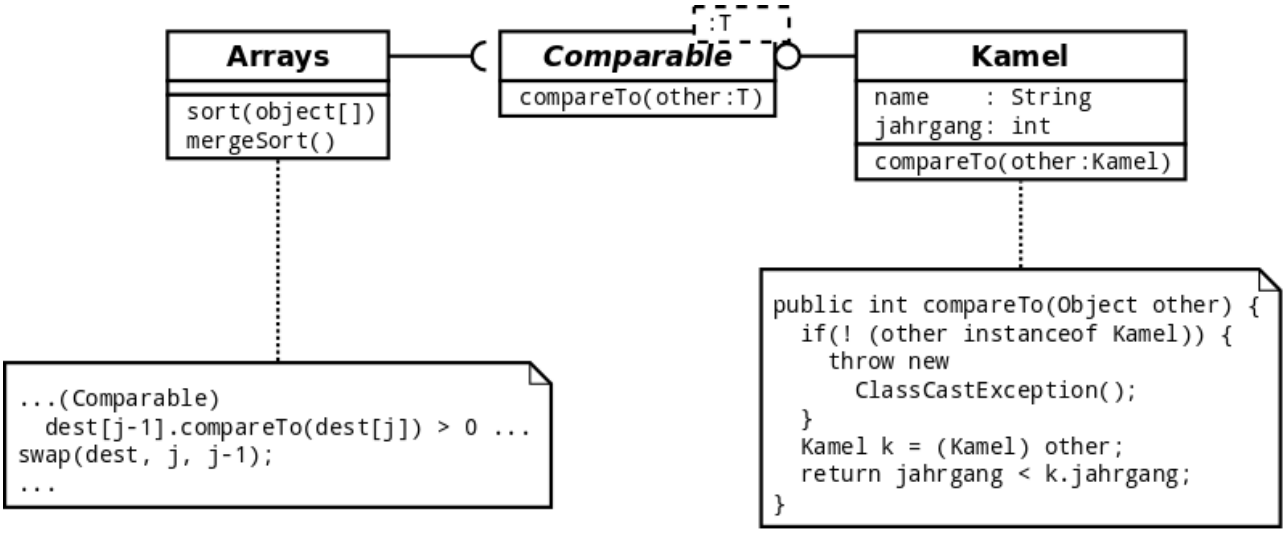
Sobald Team B mit der Entwicklung fertig ist, so wird Team A einzig `DummyImplementation` durch `RealImplementation` ersetzen.[30]

[30] Der Autor hat dies selbst durchgeführt; am Tag der Projektabnahme wurde die Attrappe erfolgreich durch die reale Implementation ersetzt!

21.4.2 Beispiel: `mergeSort()` aus der Java-API

Im folgenden Beispiel sollen Kamele sortiert werden. Es fungieren

- `Comparable`[31] als Schnittstellenklasse,
- `Kamel` als Schnittstellenanbieter (implementiert die Schnittstelle) und
- `java.util.Arrays` als Schnittstellennutzer (`mergeSort()`).



Beachten Sie, dass die Klassen `Arrays` und `Comparable` vollständig unabhängig von der Implementation `compareTo()` in der Klasse `Kamel` sind. Ebenso ist die Klasse `Arrays` (Schnittstellennutzer) vollständig von `Kamel` unabhängig.

Ein Interface kann somit als **Vertrag** zwischen Nutzer (Nutzklasse) und Implementation einer Funktionalität angesehen werden.

[31] In der Praxis wird der `java.util.Comparator` dort eingesetzt, wo mehrere Sortierkriterien möglich sind. Ist wie hier eine kanonische Sortierreihenfolge vorgegeben, so wird üblicherweise `java.lang.Comparable` verwendet.

21.4.3 Beispiel: Funktionspointer

Funktionspointer sind Variable vom Datentyp «Subroutine» (Funktion). Dabei muss natürlich angegeben werden, welche Parameter die Routine entgegen nimmt und welches Resultat die Funktion zurückgibt.

Die folgende Deklaration in der Sprache C[32]

```
double (*f) (double);
```

erzeugt eine Variable `f`, die als Wert jede Funktion annehmen kann, die einen `double` Wert entgegen nimmt und einen `double` zurück gibt. Eine Anwendung ist die Integration (Flächenbestimmung mittels Funktionen). Die Integration sieht dann z. B. wie folgt aus, und zwar für alle Werte, die `f` annehmen kann (Beispiel aus [ML90] Seite 122):

```
double integral(double a, double b, int n, double (*f) (double))
{
  double s;
  int i;
  s = 0.5 * (f(a) + f(b));
  i = 1;
  while(i < n) {
    s = s + f(((n-i) * a + i * b) / n);
    i = i + 1;
  }
  return s * (b-a) / n;
}
```

Beachten Sie, dass `f` aufgerufen wird, obschon nicht bekannt sein muss, wie die Funktion `f` genau aussieht. Wir wissen nur, dass sie einen `double` entgegen nimmt und einen `double` zurück liefert. In der Praxis ist `f` überhaupt nicht bekannt, während obige Integrationsfunktion (`integral`) geschrieben wird.

[32] denn JAVA unterstützt keine Funktionspointer

21.4.4 Callback-Funktionen (mit Schnittstellen)

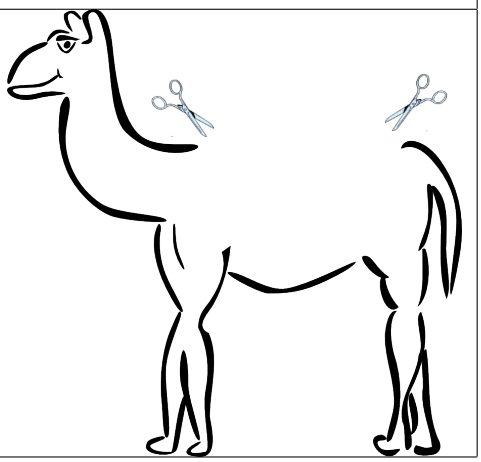
JAVA kennt keine Funktionspointer und implementiert dies mittels Polymorphismus. Die noch nicht implementierte Funktion (`f`) wird als Funktionsheader in einer Überklasse (von Vorteil als `abstract` oder `interface`) deklariert. Nun kann der Algorithmus darauf zugreifen, als ob die Funktion bereits existieren würde. Später kann jeder Programmierer die Funktion nach seinem Gutdünken programmieren und den Algorithmus einfach aufrufen, ohne diesen noch einmal neu kompilieren oder verändern zu müssen. Diese Umsetzung der Funktionspointer mittels Schnittstellen nennt man die Callback-Funktionen.

Ein weiterer Vorteil der Funktionspointer oder Callback-Funktionen ist die Möglichkeit, später den Algorithmus zu verändern, ohne die aufrufenden Funktionen neu programmieren zu müssen!

21.4.5 Beispiel: Karawane

Gleich noch ein Codebeispiel. Eine Karawane werde programmiert und zwar bevor wir die einzelnen Kameltypen (Lama, Dromedar und Trampeltier) im Detail kennen. Wir schreiben einen Algorithmus, der von einer Karawane bestimmen kann, wie viele Höcker insgesamt vorhanden sind (`totalHoeckerAnzahl()`). Dieser Algorithmus verwendet pro Kamel eine Funktion `hoeckerAnzahl()`, die erst zur Verfügung stehen wird, wenn wir Klassen wie Lama, Dromedar und Kamel implementieren werden. Sobald die Gentechnologie weiter fortgeschritten ist und es drei- oder mehrhöckrige Kamele geben sollte, so brauchen wir an unserer Karawane nichts mehr zu ändern ;-)

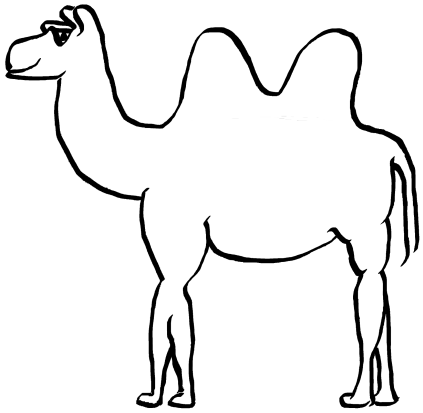
Zunächst das abstrakte Kamel als Schnittstelle:

Programmcode	Grafik
`interface Kamel { int hoeckerAnzahl(); }`	

Der Algorithmus zur Berechnung der totalen Anzahl Höcker sieht nun wie folgt aus:

```
class Karawane {
  Collection kamele;
  int totalHoeckerAnzahl() {
    int summe = 0;
    for(Kamel k : kamele) {
      summe = summe + k.hoeckerAnzahl();
    }
    return summe;
  }
}
```

Nun als Beispiel das Trampeltier als konkretes Kamel:

Programmcode	Grafik
`class Trampeltier implements Kamel{ int hoeckerAnzahl() { return 2; } }`	

21.4.6 Beispiel: Java Event-Handling

Event-Handling Ein *Event* ist nichts Anderes als ein «Ereignis». Auch das Betriebssystem verarbeitet Ereignisse wie Tastatur- oder Mauseingaben.

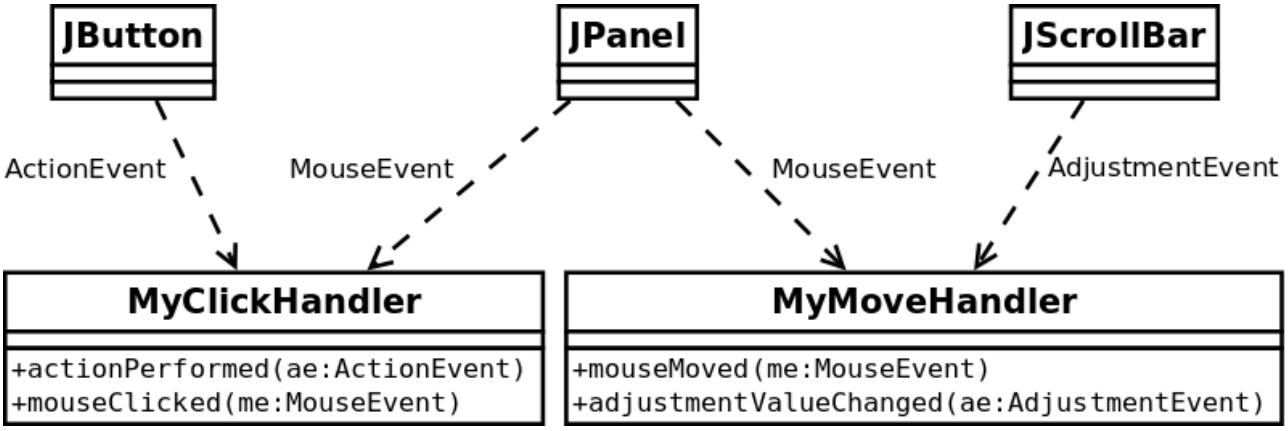
Beispiel 21.2. Schnittstelle

```
public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent e);
}
```

HW-Events , Window-Event-Queue, ... Hardware-technisch oder auf Betriebssystemebene werden solche Ereignisse in der Regel mit Interrupts umgesetzt. Diese Hardwareinterrupts sollen uns hier nicht interessieren.

Betriebssysteme wie z. B. Ms-Windows verwenden oft eine Event-Queue. Das ist eine verkettete Liste, in der die von der Hardware eingetroffenen Events chronologisch abgelegt sind. Programme werden beim Eintreffen von Nachrichten vom Betriebssystem benachrichtigt.

Javas Delegation Event Handling In JAVA kann jede Event-Quelle mehrere Listener registrieren (Siehe Variante C im Kapitel 17.3.1 ab Seite 69). Tritt ein Ereignis ein, so «verschickt» die Quelle dieses an jeden registrierten Listener indem es dessen Event-Methode (z. B. `actionPerformed()`, `mouseMoved()`, ...) mit dem Delegationsprinzip aufruft.


JButton
JPanel
JScrollBar
ActionEvent
MouseEvent
MouseEvent
AdjustmentEvent
MyClickHandler
+actionPerformed(ae:ActionEvent)
+mouseClicked(me:MouseEvent)
MyMoveHandler
+mouseMoved(me:MouseEvent)
+adjustmentValueChanged(ae:AdjustmentEvent)

Vorgehen zur Nutzung von Java-Events

1. Schreiben der Listener-Klasse (inkl. auszuführendem Code bei Eintreffen des Events)	`MyClickListener implements ActionEvent`
2. Instanzieren eines Listener-Objektes	`myClickListener = new MyClickListener();`
3. Erstellen des Quell-Objektes	`myButton = new JButton("Klick Mich");`
4. Registrieren des Listeners bei der Event-Quelle (Source)	`myButton.addActionListener(myClickListener);`

Üblicherweise wird für den Listener eine eigene Klasse verwendet. Ebenso existiert eine Hauptklasse, die sowohl die Quelle (hier einen `JButton`) wie auch ein Listener-Objekt erzeugt und den Listener bei der Quelle registriert
(`myJButton.addActionListener(myEventListener);`).

Natürlich kann zu Demonstrationszwecken alles in derselben Klasse geschehen. Der folgende `JButton` registriert sich selbst als «Listener» und schickt sich danach die Events selbst!

```
import java.awt.event.*;
import javax.swing.*;
                                                          // 1.
public class TestEvent extends JButton implements ActionListener {
  int count; // Effekt sichtbar machen mit Zahl auf dem Button.

  public TestEvent() {
    super("Klick␣mich");                                  // 3.
    // Dieses Objekt ist gleichzeitig Quelle und Listener:
    this.addActionListener(this);                         // 4.
    JFrame jf = new JFrame("Event␣Test␣Frame");
    jf.setContentPane(this);
    jf.setSize(250, 100); jf.setVisible(true);
  }

  public void actionPerformed(ActionEvent ae) {           // 1.
    this.setText("Klick␣mich␣nochmals␣(" + (++ count) + ")" );
  }

  public static void main(String args[]) {
    new TestEvent();                                      // 2.
  }
}
```

21.4.7 Beispiel: MVC und Observer/Observable

Theorie MVC: Das Original MVC Design Pattern[33] aus der Programmiersprache «Smalltalk» sieht drei Komponenten (Pakete / Klassen) vor:

- Das **Modell** (M), das die modellierten **Daten** aus dem Problembereich beinhaltet,
- den **Controller** (C), der die **Interaktion** mit dem Anwender steuert und
- den **View** (V), der die Daten **darzustellen** hat.

Das Original-Pattern wurde seither vielfach abgeändert. Die Idee der Trennung der drei Komponenten blieb sich aber grundlegend gleich. So gibt es Implementationen, bei denen das Modell ausschließlich via Controller auf den View zugreift; andererseits gibt es Theorien, bei denen jede Änderung im Modell vom Modell selbst dem View mitgeteilt werden müssen.

Teilweise (wie bei vielen JAVA-Swing Klassen) werden View und Controller in derselben Komponente zusammengefasst (Observer / Observable) oder es wird eine weitere Komponente entwickelt, die die Interaktion zwischen diesen beiden Komponenten steuert (UI-Delegate).

Codebeispiel eines Swing-Modells:

```
public class MyModel {
  private int myData;

  List<ActionListener> myListeners =
      new ArrayList<ActionListener>();

  public void setMyData(int newValue) {
    this.myData = newValue;
    fireEvent();
  }

  public int getMyData() {
    return this.myData;
  }

  public void addListener(ActionListener ml) {
    myListeners.add(ml);
  }

  private void fireEvent() {
    ActionEvent ae =
        new ActionEvent("New␣Value" + this.myData, this);
    for (ActionListener al : myListeners) {
      al.action(ae);
    }
  }
}
```

[33] = Entwurfsmuster

In der JAVA-Praxis bieten sich für MVC die Klassen `Observer/Observable` (Vererbung) oder `PropertyChangeSupport/PropertyChangeListener` (Delegation) an.

Bei beiden APIs ist die `fireEvent()`-Methode bereits implementiert und muss nur noch aufgerufen werden.

21.5 Aufgaben zu Schnittstellen und abstrakten Klassen

Aufgabe 21.1 «Sortieren von Datenstrukturen»

Ihre Lösung dieser Aufgabe sortiert Objekte beliebiger Art. Diesmal sollen (anstelle von Zahlen oder `Strings`) Kamel-Objekte sortiert werden. Das bedingt aber, dass wir uns Gedanken machen, wie die Tiere dann sortiert werden sollen. Dazu haben wir in JAVA die Möglichkeiten, das Comparable-Interface zu implementieren oder einen eigens zum Sortieren vorgesehenen Comparator (`java.util.Comparator`) zu schreiben. Diskutieren Sie die beiden Methoden und sortieren Sie die Tiere einmal nach Name und das andere Mal nach dem Alter (gespeichert ist jedoch jeweils nur das Geburtsjahr).

Aufgabe 21.2 «Gang, Zimmer, Tür»

Für ein Alarmsystem wird ein Modell eines Gebäudes benötigt. Relevante Modellelemente sind Gang, Zimmer und Tür. Eine Tür innerhalb eines Gebäudes verbindet entweder zwei Zimmer, zwei Gänge oder einen Gang mit einem Zimmer. Die bestehenden Klassen sind Gang, Tür und Zimmer. Um sicherzugehen, dass eine Tür nichts Anderes als das erwähnte miteinander verbindet, könnte man die Tür mit vier Attributen versehen: `zi1`, `zi2`, `ga1`, `ga2`: Zwei vom Datentyp Zimmer und zwei vom Datentyp Gang. Mit der folgenden etwas komplizierten Zusicherung innerhalb der Klasse Tür kann erreicht werden, dass genau zwei Räume miteinander verbunden sind:

```
boolean istTuereOK() {
  summe = 0;
  if(null != zi1) summe = summe + 1;
  if(null != zi2) summe = summe + 1;
  if(null != ga1) summe = summe + 1;
  if(null != ga2) summe = summe + 1;
  return (2 == summe) && (zi1 != zi2) && (ga1 != ga2);
}
```

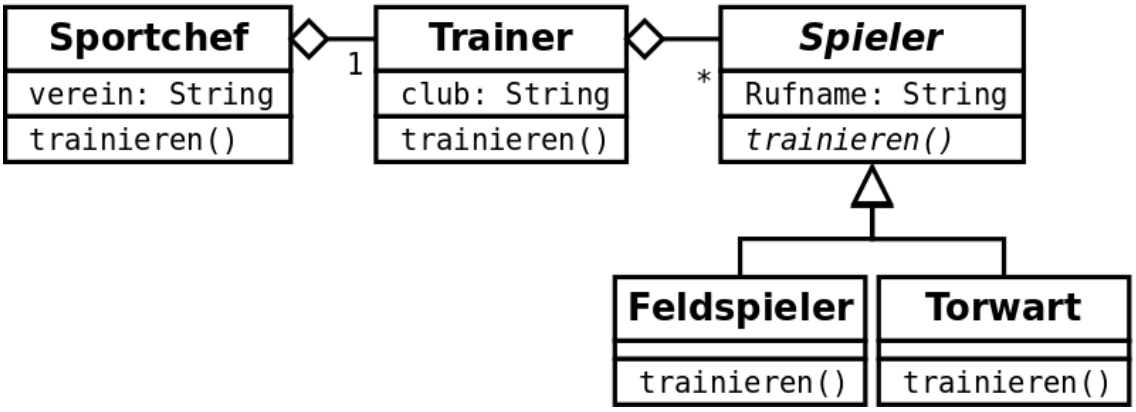
Die obige Zusicherung wird mittels Polymorphismus massiv einfacher. Wir lassen Zimmer und Gang von einer Klasse «Raumeinheit» erben. Dann benötigen wir nur noch die folgenden Tatsachen zu Berücksichtigen:

- Eine Tür grenzt an zwei Räume (`r1` und `r2`). Die Tür benötigt somit nur noch zwei Attribute.
- Die beiden angrenzenden Räume einer Tür sind nicht identisch.
- Beide Variablen, welche die Räume bezeichnen, sind nicht leer (`null`).

Fügen Sie eine abstrakte Klasse «Raumeinheit» als Überklasse von Gang und Zimmer dazu. Implementieren Sie schließlich die Klassen und die neue, einfacher zu programmierende Zusicherung `istTuereOK()` in der Klasse `Tür`.

Aufgabe 21.3 «Sportchef»

Implementieren Sie folgendes Klassendiagramm.



Dabei sind für die Aggregationssymbole die entsprechenden Attribute anzufügen. Der Sportchef soll die Methode `trainieren()` an den Trainer delegieren. Der Trainer soll die Anfrage weitergeben an alle Spieler:

Trainer:

```
void trainieren() {
  for(Spieler s : spieler) {
    s.trainieren();
  }
}
```

Die Klasse Spieler benötigt selbst keine Implementation der `trainieren()`-Methode: Die Methode hat demzufolge einen Deklarationskopf jedoch keine Definition (Body); die Klasse kann auch als abstrakte Klasse oder als Schnittstelle (Interface) programmiert werden. Die konkreten Methoden des Trainings stehen in den Subklassen Feldspieler bzw. Torwart. Geben Sie beim Feldspieler den Text «jogging» bzw. beim Torwart den Text «Liegestützen» aus.

Aufgabe 21.4 «Kochrezepte»

Modellieren Sie ein Kochrezept (das Zeichnen der Diagramme reicht). Ein solches besteht aus 3 Teilen. Im ersten Teil steht die Anzahl der Portionen und eine Kurzbeschreibung mit Menüvorschlag. Im zweiten Teil sind die Zutaten vermerkt. Jede Zutat hat einen Namen und eine Menge. Die Menge ist entweder eine einfache Anzahl oder aber es handelt sich um eine Maßangabe zusammen mit einer Maßeinheit (z. B. «2.5 kg»). Im dritten Teil folgt eine Aufzählung der Zubereitungsschritte. Leiten Sie die Maßangabe von der Menge ab und verwenden Sie dadurch das Prinzip des Polymorphismus.

Aufgabe 21.5 «Java-API»

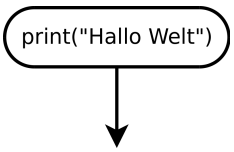
Suchen Sie in der Java API Dokumentation nach Klassen und kategorisieren Sie diese in a) Schnittstellen, b) allgemeine abstrakte Klassen und c) konkrete Klassen. Machen Sie je mindestens drei Beispiele.

Aufgabe 21.6 « `JButton` »

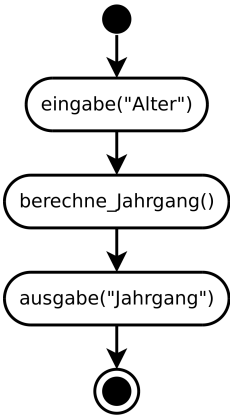
Schreiben Sie ein Programm mit zwei `JButton`-Objekten und fügen Sie zunächst via den Methoden `getModel()` und `setModel()` das `ButtonModel` des einen Buttons dem anderen hinzu. Was geschieht nun, wenn Sie einen der beiden Buttons anklicken? Erklären Sie, warum Sie dem Button 1 einen `ActionListener` hinzufügen können und dieser auch ausgeführt wird, wenn der Button 2 geklickt wird.

«*Zusammenfassung der graphischen Notationselemente*»

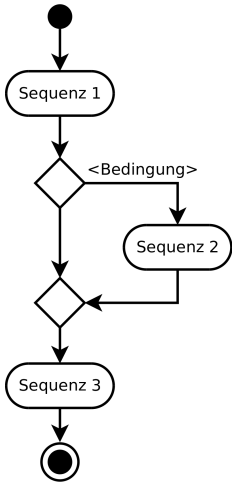
22.1 Anweisung

 Siehe [GFG11] Kap. 2	Eine Anweisung ist typischerweise entweder eine Zuweisung oder der Aufruf einer bestehenden Subroutine.

22.2 Sequenz

 Siehe [GFG11] Kap. 2	Mehrere Anweisungen werden hintereinander geschrieben. Eine Anweisung beginnt erst, wenn die vorangehende Anweisung komplett zu Ende geführt wurde. Das Start-Symbol ist ein Kreis und das End-Symbol ein umrandeter Kreis.

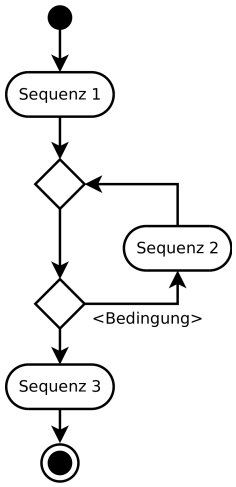
22.3 Selektion (Verzweigung)



Siehe [GFG11] Kap. 3

Bei einer Selektion wird eine Anweisung oder eine Sequenz nur unter einer gegebenen Bedingung ausgeführt.

22.4 Iteration (Schleife)



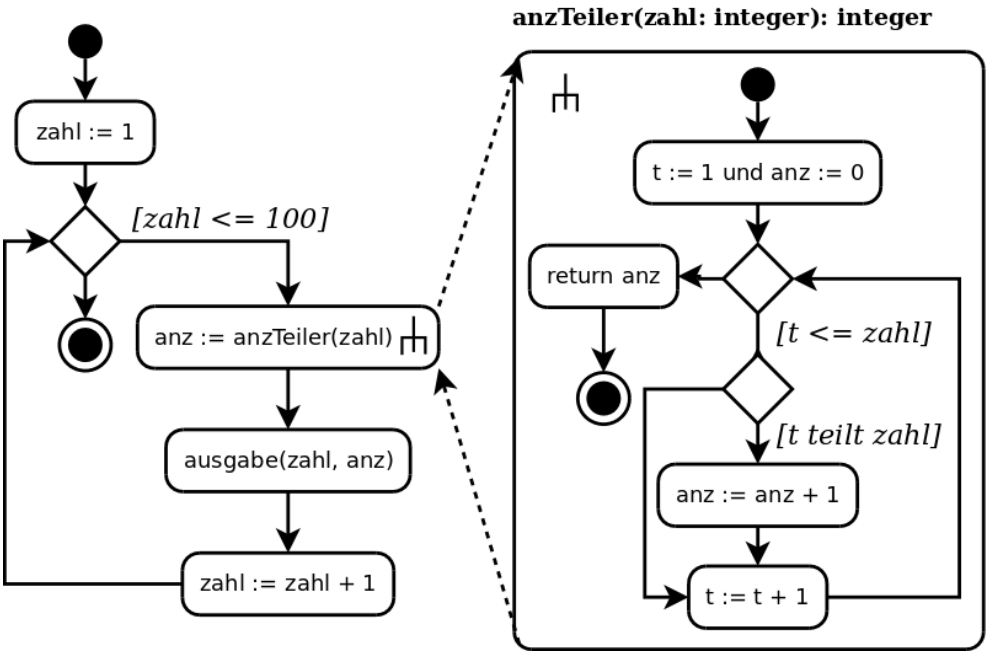
Siehe [GFG11] Kap. 4

Eine Anweisung oder Sequenz wird solange ausgeführt bis eine Bedingung nicht mehr erfüllt ist.

22.5 Unterprogramm

 Siehe [GFG11] Kap. 5	Mittels Subroutinen (auch Unterprogramm, Prozedur, Funktion oder Methode) wird eine Abfolge in eine einzige Anweisung zusammengefasst. Da es in der Regel klar ist, dass es sich beim Aufruf um eine Subroutine handelt, wird das «Rechen»-Symbol meist weggelassen.

Beispiel 22.1. Unterprogramm



22.6 Record

Typ attributName: attributTyp (s. Kap. 13 auf Seite 21 und [GFG11], dort Kap. 8)	Records (auch Datenstrukturen, Entitäten, (Verbund)typen, o. ä. genannt) enthalten Attribute mit unterschiedlichen Datentypen. Sie ergänzen das Konzept der Felder (Arrays) und kommen in der Entwurfsphase den realen Fachklassen schon sehr nahe.

Beispiel 22.2. Record

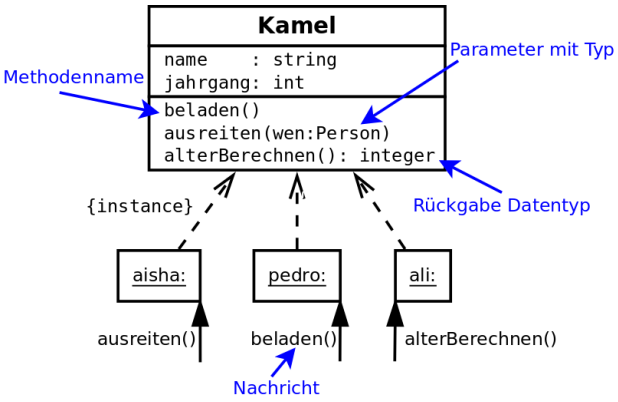
Clubmitglied

```
name          : string
vorname       : string
adresse       : string
ort           : string
postleitzahl  : integer
eintrittsjahr: integer
geburtsjahr   : integer
ehrenmitglied: boolean
```

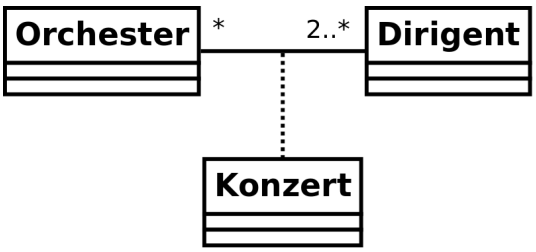
22.7 Klasse

Name Attribute Operationen()	Jede Klasse wird mit drei Teilen dargestellt. Zuoberst steht der Klassenname (also die Typenbezeichnung), in der Mitte stehen die Attribute und zuunterst sind die Methoden (also die möglichen Nachrichten) aufgeführt.

22.8 Methode

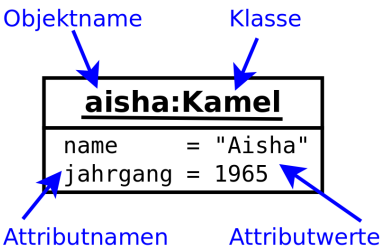
 (s. Kap. 16 auf Seite 49)	Der dritte Block einer Klasse ist für die Methoden vorgesehen. Angegeben werden die Methodennamen und optional ihre Parameter und Rückgabewerte. Der Aufruf einer Methode auf ein Objekt wird auch als «Senden einer Nachricht» verstanden. Im Gegensatz zu einem einfachen Unterprogramm beinhalten Methoden eine Angabe an welches Objekt die Nachricht gesendet wird, damit nur dessen Attribute verändert werden.

22.9 Assoziationsklasse



Stehen zwei Klassen in einer m:n-Beziehung, so gibt es generell zwei Arten dies umzusetzen. Einerseits kann jede der Klassen eine Liste der Aggregate (also der Teilklassen) führen. Dies führt zu zwei Listen.
Anderseits kann man auch eine Zwischenklasse (analog zu den SQL-Zwischentabellen) einführen. Eine solche Klasse hat dann je eine Referenz auf die beiden m:n-Klassen. Im Beispiel links braucht man nur noch eine Liste des Basistyps `Konzert` zu führen.

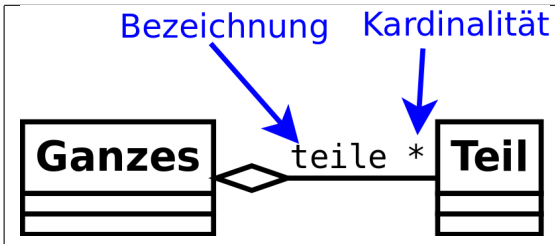
22.10 Objekte



(s. Kap. 13 auf Seite 21 und [GFG11], dort Kap. 8)

Neben der Deklaration von Datentypen (Records) existiert auch eine Notation für die Instanzen. Alternativ ist der Objektname oder die Nennung der Klasse optional, sofern es aus dem Zusammenhang klar ist, worum es sich handelt. Im Objekt-Diagramm werden weder Methoden noch die Datentypen der Attribute angegeben.

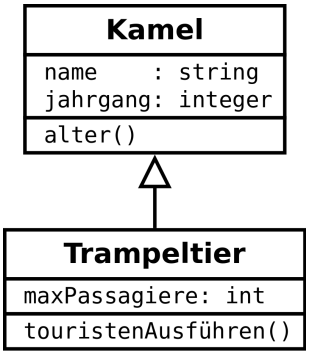
22.11 Aggregation



(s. Kap. 15 auf Seite 39)

Aggregation ist eine Ganzes-Teil Beziehung. Im Klassendiagramm wird dies mit einer Raute beim «Ganzen» angegeben. Ob es sich um eine 1:1- oder 1:n-Beziehung handelt, wird mit der Kardinalität angegeben. Der Aggregations-«Pfeil» wird beschriftet, sodass die Bedeutung des Teils klar wird. Typischerweise handelt es sich dann auch gleich um den Attributnamen in der Klasse.

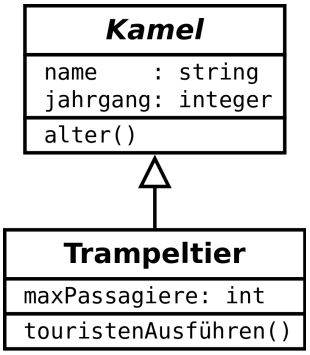
22.12 Vererbung



(s. Kap. 18 auf Seite 83)

Wird eine Klasse (hier Kamel) **erweitert** (hier Trampeltier), so wird das mit einem leeren Dreieck angegeben.

22.13 Abstrakte Klassen



(s. Kap. 21.1 auf Seite 121)

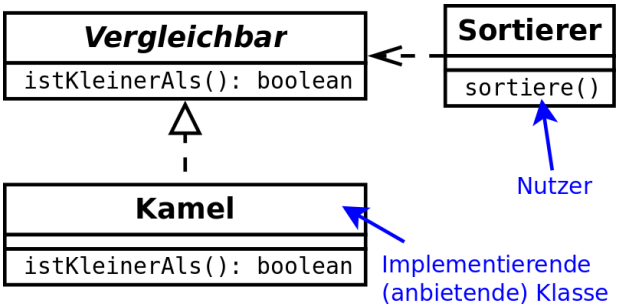
Von abstrakten Klassen kann es im Wesentlichen keine Objekte geben. Dies wird dadurch angegeben, indem der Klassenname **kursiv** geschrieben wird.

22.14 Schnittstellen

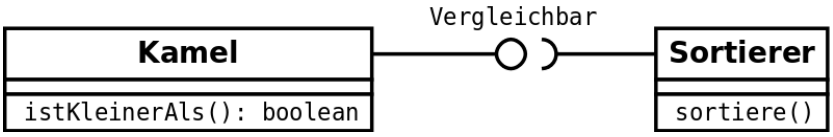
 (s. Kap. 21.3 auf Seite 127)	Schnittstellen werden ohne Attribute angegeben (also nur mit zwei Bereichen im Gegensatz zur Klasse, die drei Bereiche aufweist). Wie abstrakte Klassen werden auch Schnittstellen kursiv geschrieben.

Bei der Schnittstelle gibt es im Wesentlichen einen Anbieter, der die geforderten Methoden zur Verfügung stellt und es gibt einen Nutzer, der – wie könnte es anders sein – die Methoden nutzen will.

Hierzu gibt es zwei alternative Notationen. Eine erste Notation, falls die Schnittstellenklasse explizit angegeben wird. Die Implementation wird wie die Subklasse mit leerem Dreieck dargestellt, **aber** der Pfeil ist gestrichelt. Die Nutzklasse wird ebenfalls mit gestricheltem Pfeil, jedoch mit normaler Spitze (statt Dreieck) dargestellt.



Die alternative Notation wird benutzt, falls die Schnittstelle wohl bekannt ist. Sie verwendet eine «Stecker-Buchsen»-Notation:



A | Anhang

«Was ich sonst noch alles sagen wollte.»

A.1 Metasprache zur Darstellung der Grammatiken

Ich verwende hier eine vereinfachte EBNF[34]-Notation, um die JAVA-Syntax darzulegen. Die hier benutzte Notation ist zwar nicht mehr ganz korrekt, dafür jedoch etwas einfacher.

Die folgenden Symbole werden verwendet und beziehen sich auf das Zeichen oder den eben verwendeten Klammerausdruck:

*	Das Feld darf mehrfach verwendet werden, darf aber auch weggelassen werden.
+	Das Feld darf mehrfach verwendet werden, muss aber mindestens einmal aufgeführt werden.
[...]	Der Ausdruck innerhalb der eckigen Klammern ist optional.
?	Alternative Schreibweise für *optional*: Feld kommt maximal einmal vor, kann aber auch wegfallen.
|	Entweder das Symbol links oder das Symbol rechts des Striches wird eingesetzt.
<>	Der Textblock innerhalb der spitzen Klammern muss durch den effektiven Wert ersetzt werden. Beispiel Person :== «<Name>» « <Vorname>» wird zu «Hans» «Meyer».
(...)	Klammerausdruck. Die obigen Symbole werden rechts des Ausdrucks geschrieben und gelten dann für den gesamten Klammerausdruck.
fett	Muss exakt so abgeschrieben werden (z. B. <MODIFIER>+ `class` ...).

Beispiele

```
a*     : "", "a", "aa", "aaa", ...
(xy)+  : "xy", "xyxy", "xyxyxy", ...
x*y?   : y, xx, xxxxy, ...
```

[34] Erweiterte Backus-Naur-Form (s. Wikipedia)

A.2 Java-Klasse (Kompilationseinheit)

Jede Kompilationseinheit in JAVA beinhaltet in der Regel genau eine Klasse. Diese ist (abgesehen von Code-Kommentaren) nach dem folgenden Grundmuster aufgebaut:

[package]	genau eine Paketdeklaration, die aussagt, wo die Datei aufzufinden ist
[imports]	beliebig viele Angaben zu externen Referenzklassen und Paketen
[class]	normalerweise eine Klassendefinition

Dabei ist die Klasse (`class`) wie folgt aufgebaut[35]:

```
[<MODIFIER>]* class      <Klassenname>
              [extends    <Klassenname>]
              [implements <Klassenname>+]
{
  [member]*
}
```

[35] Für die EBNF Notation (s. Kap. A.1 auf Seite 153)

A.2.1 Members einer Klasse

Innerhalb einer Klasse (`class`) können die folgenden *Klassenelemente* (Members) auftreten (wobei nur die ersten drei für ein Verständnis der Objektorientierung nötig sind):

Member	Beschreibung	Beispiel	Bemerkung
Attribut	Objekt-Eigenschaft	`String name;`	Jedes Objekt hat seinen eigenen internen Zustand.
Methode	Ereignisse (Events), die auf die Objekte wirken können	`void add(float smd) { summe += smd; }`	Ereignisse verändern typischerweise den internen Zustand (Attribute) der Objekte.
Konstruktoren	Spezielle Methode, um Objekte zu erzeugen	`public Person () {...}`	Jedes Objekt wird durch einen Konstruktor (mittels `new`) erzeugt.
Statisches Attribut	Klasseneigenschaft	`static int obCount;`	Jede Klasse kann auch globale Variable aufweisen, die für alle Objekte den Selben Speicherbereich teilen.
Statische Initialisierer	Code, der bei Klassenerzeugung ausgeführt werden soll	`static {...}`	Dieser Code wird beim Laden der Klasse ausgeführt. Ohne das Schlüsselwort `static` wird der Code bei der Objekterzeugung ausgeführt.
Dynamischer Initialisierer	Code, der nach `super()` , aber vor dem Rest des Konstruktors ausgeführt wird.	`{ ...}`	Dynamische Initialisierer werden selten gebraucht, da den Entwicklern oft nicht klar ist, wann diese ausgeführt werden. Finger davon lassen!
Statische Methoden	Klassenereignisse	`static int random()`	Werden meist für klassische Funktionen, die nicht von einem Objektzustand abhängig sind, verwendet.
Innere Klassen	Klassen innerhalb bestehender Klassen	`class Xyz {...}`	Beispiel: AHV Nummer als innere Klasse einer Person.

1. Beispiel einer Kompilationseinheit, die lediglich aus einer Klasse besteht:

```
public class Trampeltier extends Kamel implements Haustier
{
  float   gewicht;
  String  name   ;

  public Trampeltier(String name, Date geburt, float gewicht) {
      super(name, geburt);
      this.gewicht = gewicht;
  }

  float getGewicht() {
      return this.gewicht;
  }
}
```

2. Beispiel einer Kompilationseinheit:

```
/* package */
package ch.santis.training;

/* Klassendefinition der Klasse Kunde */
public class Kunde {
  String name, vorname;
  int    kundennummer ;

  public Kunde(String name, String vorname, int id) {
      this.name          = name   ;
      this.vorname       = vorname;
      this.kundennummer  = id     ;
  }

  public String getName() {
      return name;
  }
}
```

Bemerkung A.1. Neben `class` sind auch `interface` oder `enum` als Inhalt einer Kompilationseinheit möglich.

A.3 Entwurfsmuster Singleton

Viele Programme benutzen immer und immer wieder dieselben Muster. Diese Muster wurden gesammelt und soweit abstrahiert, dass ihre Lösung nun auf viele Problemstellungen angewendet werden kann. Eine solche abstrahierte Lösung nennt sich **Entwurfsmuster** oder **Design Pattern**[36].

Ein häufig verwendetes Muster ist der **Singleton**.

Manchmal will man nämlich von einer Klasse garantieren, dass nur ein einziges Objekt erzeugt wird. Dies wird einerseits für globale Variable verwendet, kann aber auch für Zugriffe auf Ressourcen (Drucker, Datenbank, ...) eingesetzt werden. Es reicht einen Druckertreiber im ganzen System zu haben, damit der Zugriff auf das physische Gerät besser gesteuert werden kann.

Ein Singleton darf also nicht mit `new` von überall her erzeugt werden. Der Konstruktor muss `private`, also eingeschränkt sein. So eben, dass nur der Singleton selbst sein Objekt erzeugen kann. In JAVA darf der Konstruktor nicht einfach weggelassen werden, denn sonst würde der Compiler einen `public` Konstruktor ohne Parameter erzeugen!

Das Objekt muss von außen abgerufen werden können. Daher werden wir eine «getter»-Methode erstellen. Diese Methode muss jedoch `public` und statisch (`static`) sein, damit sie auch aufgerufen werden kann, wenn noch kein Objekt vorhanden ist (analog `main()`).

Innerhalb der Klasse wird ein statisches privates Attribut geführt, das nur vom Singleton gesteuert wird. Typischerweise nennen wir dies gleich `singleton`.

In der «getter»-Methode soll immer eine Referenz auf dasselbe Objekt zurückgegeben werden. Beim ersten Aufruf soll dieses Objekt erstellt und in der Variable `singleton` abgelegt werden.

Die «getter»-Methode soll auch synchronisiert werden, damit zwei Zugriffe nicht gleichzeitig das Objekt erzeugen.

Nächste Seite: Ein Prototyp einer Singleton-Klasse.

[36] Eine der Bekanntesten Sammlungen von Entwurfsmustern ist das Buch «Design Patterns» der «Gang of Four»[GJHV94]

```java
package ly.gress.demos.singleton;

/**
 * Demonstration eines 'Singleton', also einer Klasse, von
 * der es lediglich eine Instanz (also ein Exemplar) geben kann.
 * @author Philipp Gressly (phi AT gress DOT ly)
 */
public class SingletonDemo {

    private static SingletonDemo singleton;

    private int irgendeinAttribut;

    private SingletonDemo() {
    }

    public static synchronized SingletonDemo getSingletonDemo() {
        if(null == singleton) {
            singleton = new SingletonDemo();
        }
        return singleton;
    }

    public String irgendeinePublicMethode() {
       return "Hallo du einsame Welt";
    }

    public void setIrgendeinAttribut(int iea) {
        this.irgendeinAttribut = iea;
    }

    public int getIrgendeinAttribut() {
        return this.irgendeinAttribut;
    }

} // end of class SingletonDemo
```

Bemerkung A.2. Siehe auch Wikipedia über Entwurfsmuster und den Singleton:

`http://de.wikipedia.org/wiki/Singleton_(Entwurfsmuster)`

`http://de.wikipedia.org/wiki/Entwurfsmuster`

A.4 Gegenüberstellung: strukturiert vs. objektorientiert

Hier versuche ich eine Gegenüberstellung den **Vorgehensweisen** «strukturiert» vs. «objektorientiert» (OO) anzugeben. Wichtig ist, dass dies keine Gegenüberstellung der strukturierten und den objektorientierten **Programmiersprachen** ist: Jede OO-Sprache ist hochgradig strukturiert und verwendet **alle** Konzepte der strukturierten Programmiersprachen.

- Der wesentliche Unterschied der beiden Vorgehensweisen liegt in der Philosophie der Programmierung, der Analyse und der Planung. Programmiersprachen unterstützen das OO-Vorgehen mehr oder weniger. Der strukturierte Ansatz stellt den Algorithmus in den Vordergrund (Beispiele aus [GFG11]); wohingegen der OO-Ansatz die Datenstrukturen ins Zentrum stellt und die Methoden und Algorithmen den Datenstrukturen hinzufügt.
- Die OO-Programmiersprachen ermöglichen die 3 fundamentalen Forderungen der Objektorientierung zu erfüllen:
  - Abstraktion
  - Kapselung
  - Hierarchie (Vererbung)

Das strukturierte Vorgehen beinhaltet das Denken in Aufgaben und Strukturen...

- Begriffe der Strukturierung sind:
  - Modulare Verarbeitung (Executable Modules)
  - Wiederverwendbarkeit (Reusability)
  - Geheimnisprinzip[37] (Information Protection)
  - Datenhierarchie (Data Hierarchy)
- Programmiersprachen, die sich für die strukturierte Programmierung eignen sind: BASIC, COBOL, PL/1, C, aber auch JAVA, Python, Ruby, C++, Smalltalk, PHP, ...

Das OO-Vorgehen ergänzt das strukturierte Vorgehen um...

- ... das Denken in Objekten und um die
- typischen Konzepte von OO-Sprachen (Smalltalk, C++, JAVA, Python, Ruby, PHP, ...). Diese sind die
  - Delegation
  - die Vererbung und
  - der Polymorphismus.

[37] auch Information Hiding genannt

A.4.1 Zusammenfassung der Konzepte von Programmiersprachen

1. Teil: Herkömmliche Konzepte der strukturierten Programmiersprachen

- Variable und Datentyp und Typenbindung (s. [GFG11] Kap. 1)
- Ausdrücke (Terme) (s. [GFG11] Kap. 1)
- Sequenz (Reihung / Folge) (s. [GFG11] Kap. 2)
- Selektion (Verzweigung) (s. [GFG11] Kap. 3)
- Iteration (Schleifen) (s. [GFG11] Kap. 4)
- Subroutinen (Methoden, Prozeduren, Funktionen) = Prinzip der *schrittweisen Verfeinerung* auch *funktionale Dekomposition genannt* (s. Kap. 16 auf Seite 49 und [GFG11], dort Kap. 5)
- Arrays (Felder, Tabellen, Matrizen) (s. [GFG11] Kap. 6)
- Strings (Zeichenketten) (s. [GFG11] Kap. 7)
- Records (Datenstrukturen und Sammelobjekte) auch Datenabstraktion genannt[38]. (s. Kap. 13 auf Seite 21 und [GFG11], dort Kap. 8)
- Algorithmen (s. [GFG11] Kap. 2, 3, 4, 9 und 10)
- Rekursion (s. [GFG11] Kap. 11)
- Modularisierung (`uses` in Pascal, Modula; `PACKAGE` in PL/I; header Files in C/C++, Pakete in JAVA
- Funktionspointer (Zeiger auf Funktionen / Callback) (s. Kap. 21.4.4 auf Seite 133)
- Persistenz

[38] In OO-Sprachen können hierfür die Klassen verwendet werden, auch wenn diese viel mächtiger sind.

2. Teil: Konzepte objektorientierter strukturierter[39] Programmiersprachen

- **Delegation** (s. Kap. 17 auf Seite 63)
- **Erweitern von Datenstrukturen** (s. Kap. 18 auf Seite 83)
- **Überschreiben und Verändern von Methoden** (s. Kap. 19 auf Seite 95)
- **Kapselung** von Daten (Lokalität) (Information Protection)
- **Polymorphismus** (Substitutionsprinzip und Überschreiben) (s. Kap. 20 auf Seite 105)
- Abstrakte Klassen (s. Kap. 21.1 auf Seite 121)
- Schnittstellen (Interfaces) (s. Kap. 21.3 auf Seite 127)
- Garbage Collection (s. Kap. 13.6 auf Seite 31)
- Introspection und Reflection

Weist eine Programmiersprache alle strukturierten Konzepte auf und darüber hinaus auch die fett-gedruckten Konzepte der Objektorientierung, so wird die Sprache **objektorientiert** genannt.

Daneben gibt es natürlich noch viele weitere Konzepte; teilweise auch von sehr spezialisierten Programmiersprachen. Beispiele:

- Abfrage(sprachen): Wichtige Abfragearten an Datenbestände sind vorprogrammiert (Bsp. SQL-SELECT)
- Selbständige Pfadsuche in Regelsystemen (PROLOG)
- Reguläre Ausdrücke (REGEX)
- Exception Handling
- Multithreading

Es gibt auch Konzepte der **Algorithmen**, die unabhängig von Programmiersprachen sind. Je nachdem wird man von Klassenbibliotheken bei den folgenden Konzepten unterstützt:

- Komponentenentwicklung
- Genetische Algorithmen
- Neuronale Netze

[39] Nochmals zur Erinnerung: Objektorientierte Sprachen sind immer auch strukturierte Sprachen!

B | Link-Verzeichnis

Verzeichnis aller verwendeten Links aus den vorangehenden Kapiteln.

- `http://www.gress.ly/oo/oo.pdf` : Download zu diesem Buch
- `http://www.pmg.csail.mit.edu/~liskov/` : Polymorphismus: Das Substitutionsprinzip
- `http://www.programmieraufgaben.ch` : Aufgabenbuch zu diesem Skript
- `http://www.programmieraufgaben.ch/uploads/oo.pdf` : Download zu diesem Buch
- `http://de.wikipedia.org/wiki/Singleton_(Entwurfsmuster)` : *Singleton*-Pattern
- `http://de.wikipedia.org/wiki/Entwurfsmuster` : Entwurfsmuster (*Design-Pattern*)

Literatur

[Blo08] Bloch, Juhua: *Effective Java*. Addison-Wesley; 2 edition (May 28, 2008), 2008. – ISBN 978–0321356680

[GFG11] Gressly-Freimann, Philipp ; Guggisberg, Martin: *Programmieren lernen*. Orell Füssli, 2011 `http://www.programmieraufgaben.ch`. – ISBN 978–3–280–04066–9

[GJHV94] Gamma, E. ; Johnson, R. ; Helm, R. ; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. – ISBN 0–201–63361–2

[HSSV12] Hubwieser, Peter ; Spohrer, Matthias ; Steinert, Markus ; Voss, Siglinde: *Informatik*. Klett, 2012. – ISBN 978–3–12–731768–8

[Knu97] Knuth, Donald E.: *The Art of Computer Programming*. Addison Wesley, 1997 `http://www-cs-faculty.stanford.edu/ uno/`. – ISBN 0–201–89683–4

[ML90] Martin Lowes, Augustin P.: *Programmieren mit C*. Vieweg+Teubner, 1990. – ISBN 978–3519122869

C | Stichwortverzeichnis