Einführung in die Objektorientierte Programmierung

Thomas Letschert

Einführung in die Objektorientierte Programmierung

Thomas Letschert

Version vom 4. Mai 2012

Der Autor dankt allen die mit Hinweisen und Verbesserungsvorschlägen zur jetzigen Gestalt des Skripts beigetragen haben. Besondere Erwähnung verdienen

Werner Lauwerth,
Berthold Franzen,
Oliver Correll.

Weitere Hinweise auf orthographische oder inhaltliche Fehler, Ungenauigkeiten, Auslassungen, Missverständlichkeiten sowie Verbesserungsvorschläge sind willkommen.

Dr. Th Letschert
Fachbereich MNI

Technische Hochschule Mittelhessen
D-35390 Gießen

Inhaltsverzeichnis

Kapitel 1

Algorithmen und Programme

1.1 Hardware und Software

1.1.1 Programme und Computer

Software – Maschinen aus Ideen

Lokomotiven, Toaster und Videorecorder werden von Maschinenbau– und Elektroingenieuren hergestellt. Bauingenieure bauen Brücken und Häuser. Informatiker – *Software–Ingenieure*, wie sie sich oft selbst nennen – stellen Software her. Software ist anders als Brücken, Toaster, Lokomotiven oder alles andere sonst, das von anderen (richtigen ?) Ingenieuren gebaut wird. Wird Software gebaut? Was ist Software?

Software begegnet uns meist als Programm, das auf einem PC installiert ist oder das man in Form einer CD oder DVD kauft und dann selbst installiert. Ein *installiertes Programm* erscheint meist als kleines Bildchen oder Symbol (oft *Icon* genannt) auf dem Bildschirm. Klickt man es mit der Maus an, beginnt der PC sich in einer besonderen Art und anders als zuvor zu verhalten. Vielleicht ertönt Musik, vielleicht hört man Motorengeräusche und man sieht das Innere eines Flugzeug–Cockpits auf dem Bildschirm, vielleicht zeigt der PC aber nur stumm einem Bildschirmausschnitt – ein "Fenster" – als langweilige weiße Fläche in der die Zeichen erscheinen, die man auf der Tastatur tippt.

Richtige Maschinen tun etwas Richtiges. Toaster toasten, Lokomotiven ziehen Züge. Entweder sind die richtigen Maschinen so schwer, dass man sie nicht heben kann, oder, wenn doch, dann kann man sie fallen lassen und sie sind kaputt, oder die Füße tun weh, oder beides. Programme sind anders. Sie sind nicht schwer oder leicht und sie tun auch nicht wirklich etwas. Zumindest tun sie es nicht selbst. Wie ein Traum oder ein Albtraum den Geist, oder wie ein Virus eine Zelle, so besetzen Programme einen Computer und bringen ihn dazu sich in einer bestimmten Art zu verhalten. Meist ist dies erwünscht und mit dem Programm auch mehr oder weniger teuer bezahlt. Manche werden aber gegen unseren Willen installiert und aktiviert und bringen den Rechner dazu, sich in einer Art zu verhalten, die uns nicht gefällt – dann nennen wir sie auch manchmal tatsächlich "Virus".

Computer ohne Programme sind nur *Hardware* – nutzlose Energie– und Platzfresser. Programme ohne Computer sind nur *Software* – wirkungslose Ideen von etwas, das geschehen könnte. Die geniale Idee der Informatik besteht darin, die Idee des Tuns – das Programm – und sein Ausführungsorgan – die Maschine – zu trennen. Mit den Computern hat sie Maschinen geschaffen, die keinen eigenen Willen, keinen eigenen Zweck, kein eigenes Verhalten haben, sondern sich jederzeit, wie Zombies, bereitwillig einem beliebigem fremden Willen, dem Programm, unterwerfen. Beide, Programme und Computer, können damit unabhängig voneinander weiterentwickelt und nahezu beliebig kombiniert werden. Diese Idee, so seltsam, exotisch und wenig anwendbar sie vor einigen Jahren einmal erschien, hat in kurzer Zeit die Welt erobert und ihre Zombies kommen manchem von uns inzwischen recht selbstbewusst vor.

Grundbausteine eines Computers

Computer, Informatiker nennen sie oft auch "Rechner", sind dazu da, einen fremden Willen auszuführen, der ihnen in Form eines Programms aufgenötigt wird. Zu diesem Zweck sind sie in einer bestimmten Weise aufgebaut. Bei aller Verschiedenheit im Detail folgt dieser Aufbau einem Grundmuster, das sich der Großcomputer eines Rechenzentrums, der PC auf dem Schreibtisch, das Innere eines Handys und das Steuerungselement in einer Waschmaschine teilen. Dieses Grundmuster nennt man etwas hochtrabend "Architektur" der Rechner. Sie beruht auf folgenden Gundbestandteilen:

- *Prozessor*, auch: CPU *(Central Processing Unit)*,
- *Hauptspeicher*, auch: *Arbeitsspeicher*, oder *RAM (Random Access Memory)*,
- *Plattenspeicher* und
- *Ein–/Ausgabegeräte* wie Monitor, Tastatur, Maus, etc.

Eine Waschmaschine hat (noch) keinen Monitor und ein Handy (noch) keinen Plattenspeicher, aber der grundsätzliche Aufbau der Rechner in allen Geräten ist stets der gleiche. Wir stellen die Dinge hier nur so dar, wie sie sich prinzipiell verhalten. Tatsächlich ist alles etwas komplexer. So beginnt eine moderne CPU ihre Arbeit damit, die aktuellen Softwareupdates einzuspielen. – Heute ist eben alles ein Computer.

Programm

Der Prozessor verarbeitet die Daten, die sich im Hauptspeicher befinden. Er befolgt dabei die Anweisungen eines *Programms*. Ein Programm besteht aus einer Folge von Anweisungen. *Anweisungen* nennt man auch Befehle. Eine Anweisung ist eine Aktion, die vom Prozessor ausgeführt werden kann. Das Programm befindet sich im Hauptspeicher. Es sagt mit all seinen

einzelnen Anweisungen dem Prozessor, und damit dem Computer insgesamt, was er tun soll. Wenn der Computer läuft, holt sich also der Prozessor die Befehle des Programms aus dem Hauptspeicher und führt sie aus – Befehl für Befehl.

Im Plattenspeicher befinden sich die Programme und Daten des Systems und seiner Benutzer. Sollen sie verarbeitet werden, müssen sie zuerst in den Hauptspeicher transportiert (geladen) werden. Die Ein–/Ausgabegeräte dienen dazu mit dem Computer zu kommunizieren. Der Plattenspeicher ist sehr groß. Er enthält alle Programme und Daten. Der Hauptspeicher ist kleiner und enthält nur die Daten und Programme, die gerade verarbeitet werden. Der Prozessor ist noch kleiner. Er enthält nur den einzelnen Befehl der gerade verarbeitet wird und dazu noch einige wenige Daten die dazu gebraucht werden.

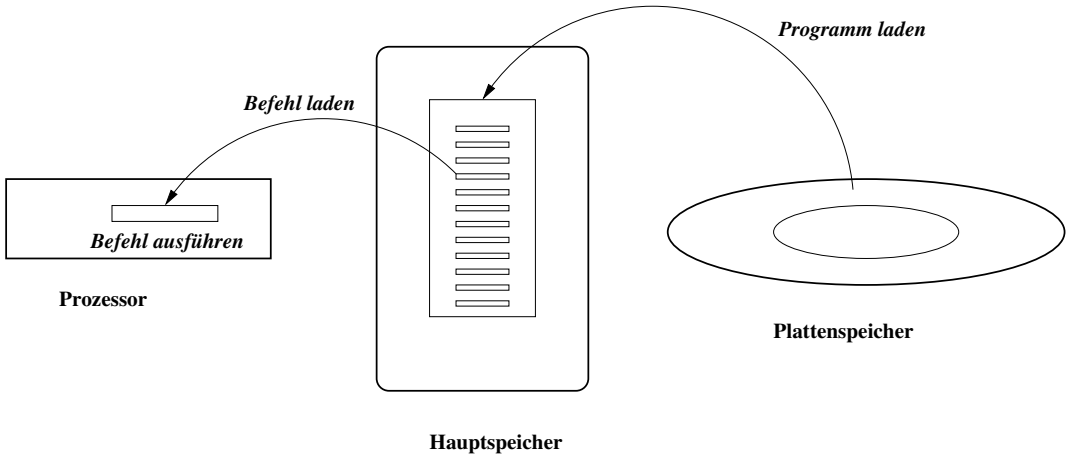


Abbildung 1.1: Programme und Befehle laden und ausführen

Prozessor

Der *Prozessor* (CPU) bildet zusammen mit dem Hauptspeicher den Kern des Rechners. Er enthält einige wenige Speicherzellen, *Register* genannt. In diese Register kann er aus dem Hauptspeicher Daten laden, sie durch Rechenoperationen verändern und dann wieder in den Hauptspeicher schreiben. Die Aktivität des Prozessors wird durch ein Programm gesteuert. Bei einem Programm handelt es sich ebenfalls um Daten. Jeder einzelne Befehl des Programms besteht aus ein paar Nullen und Einsen und ein Programm besteht typischerweise aus Millionen von Befehlen. Sie befinden sich auf der Platte, einer CD oder einem anderen Speichermedium. Von dort holt das Betriebssystem sie in in einem größen Happen in den Hauptspeicher. Sie stehen dann dort zur Verfügung und der Prozessor lädt sie häppchenweise, Befehl für Befehl, in ein spezielles Register (eine Speicherstelle) und führt sie dann aus.

Manchmal passieren dabei Fehler. So kommt es öfter vor, dass der Prozessor irrtümlich eine Folge von Nullen und Einsen als Befehl laden und ausführen will, die gar nicht als Befehl gemeint ist, sondern eine Zahl darstellen die verarbeitet werden soll. Oder der Befehl sagt, dass ein Wert von einer Speicherstelle geholt werden soll, die nicht existiert oder die gesperrt ist. Solche Fehler beruhen darauf, dass das gerade ausgeführte Programm fehlerhaft ist. Der Rechner reagiert darauf mit dem sofortigem Abbruch des Programms. In den Urzeiten hat der Rechner dazu einfach seine Aktivität insgesamt eingestellt. Heute wird die Situation (in aller Regel) dadurch bereinigt, dass ein anders Programm ausgeführt wird.

Hauptspeicher

Der *Hauptspeicher* enthält viele Speicherzellen: Sie sind direkt adressierbar und ihr Inhalt kann von der CPU (dem Prozessor) sehr schnell gelesen und geschrieben werden kann. Eine Speicherzelle enthält typischerweise ein *Byte*. Ein Byte sind 8 Bit (8 Binärzeichen, 0 oder 1). “Direkt adressierbar” bedeutet, dass jede Zelle, d.h. jedes Byte, eine Adresse hat und der Prozessor jederzeit auf jede beliebige Speicherzelle zugreifen kann. Dieser wahlfreie Zugriff (engl. *Random Access*) ist sehr wichtig. Ohne ihn könnten die Programme nicht vernünftig ausgeführt werden. Daten müssen an beliebigen, nicht vorhersehbaren Stellen geholt und gespeichert werden. Und von einer Anweisung muss je nach Bedarf zu einer beliebigen anderen verzweigt werden können.

Plattenspeicher

Die Schnelligkeit und die direkte Adressierbarkeit machen den Hauptspeicher sehr teuer. Zur Speicherung von Massendaten wird darum ein billigerer Hintergrundspeicher eingesetzt. Typischerweise ist das ein *Plattenspeicher*. Der Plattenspeicher – die "Festplatte" – kann sehr viele Daten aufnehmen. Er ist aber im Vergleich zum Hauptspeicher langsam und nicht direkt adressierbar. Daten können nur in großen Einheiten und nicht byteweise gelesen und geschrieben werden. Daten im Plattenspeicher müssen darum immer zuerst in den Hauptspeicher geladen werden, bevor der Prozessor sie verarbeiten kann.

Externe Geräte

Den Plattenspeicher und die Ein-/Ausgabegeräte wie Tastatur, Graphikkarte (steckt im Computer und steuert den Monitor) und Maus, CD–Laufwerk, etc. bezeichnet man als *externe Geräte*. Sie liegen ausserhalb von Prozessor und Hauptspeicher, die den innersten Kern des Computers bilden. Die externen Geräte werden auch durch den Prozessor gesteuert. Informationen, die über die externen Geräte eintreffen, werden vom Prozessor angenommen und im Hauptspeicher und dann eventuell auf der Platte gespeichert.

Aus der Sicht des Rechners sind also Dinge wie Maus oder das CD–Laufwerk am alleräussersten Ende der Welt. Sie sind "extern" und liefern oder schlucken Daten. Dinge wie Auge, Hand oder auch die CD, von denen Daten tatsächlich kommen oder zu denen sie letztlich gehen, sind für den Rechner nicht einmal mehr extern. Sie spielen in der Welt der Programme keine Rolle.

Dateien und Programme

Die Masse der Daten liegt in Dateien auf der Festplatte. Dateien sind in Verzeichnissen (auch "Ordner", oder engl. *Directory*) organisiert. Eine Datei kann Daten beliebiger Art enthalten: Texte, Musik, Graphiken, etc. Eine Datei kann auch ein Programm enthalten. Programme können in den Hauptspeicher geladen und dann ausgeführt werden. Das nennt man "Aktivieren des Programms". Oft werden Programme durch kleine Symbole (Bildchen) auf dem Bildschirm dargestellt. Klickt man sie an, werden sie aktiviert, also von der Platte in den Hauptspeicher geladen und dann ausgeführt. Beim Ausführen liest der Prozessor Befehl für Befehl des Programms und befolgt ihn. Ein Programm besteht aus einer – meist sehr langen – Sequenz von Bits, die vom Prozessor häppchenweise als Befehle verstanden werden. Ein *Programm* ist also eine Datei deren Inhalt vom Prozessor verstanden wird.

Dateien, die Befehle für den Prozessor enthalten, nennt man ausführbare Dateien. Dateien mit einem anderen Inhalt können nicht aktiviert werden. Eine Textdatei beispielsweise kann aber gedruckt werden, man kann sie mit Hilfe eines Editors betrachten und verändern. Oft identifiziert man eine ausführbare Datei mit ihrem Inhalt und sagt auch "Programm" zu der Datei die das Programm enthält.

Das Betriebssystem startet Programme

Ein Programm wird also gestartet, indem der Inhalt der Datei, die es enthält, in den Hauptspeicher kopiert wird und der Prozessor dann von dort den ersten Befehl des Programms lädt und ausführt. Das Kopieren in den Hauptspeicher und die Umorientierung der CPU (des Prozessors) auf die neue Aufgabe wird vom *Betriebssystem* erledigt. Das Betriebssystem ist selbst ein Programm, das ständig aktiv ist und dem Benutzer und seinen Programmen dabei hilft, die Hardware des Systems zu nutzen.

Meist startet das Betriebssystem ein Programm nicht aus eigenem Entschluss, sondern nachdem der Benutzer es dazu aufgefordert hat. Heutzutage klickt man dazu meist ein kleines Bildchen ("Icon") an, das das Programm symbolisiert. Das System kennt die Koordinaten des Bildchens und weiß welches Programm (d.h. welche Datei) damit gemeint ist. Klickt man in diesem Bereich, dann startet das (Betriebs–) System das Programm. Das Programm und das zugehörige Bild müssen dem Betriebssystem dazu natürlich vorher bekannt gemacht werden, man sagt, das Programm wird registriert.

Wenn das Betriebssystem ein Programm startet, bedeutet das, dass es ihm die CPU zur Ausführung seiner Befehle überlässt. Natürlich ist das Programm nicht wirklich etwas Aktives. Der Prozessor ist der aktive Teil. Er führt einen Befehl des Betriebssystems aus, dieser veranlasst ihn, den ersten Befehl des Programms zu laden und dessen Ausführung zieht das Laden und Ausführen der anderen Befehle des Programms nach sich. Am Ende oder bei einem Fehler wird dafür gesorgt, dass es wieder mit Befehlen des Systems weitergeht.

Eingabe von Kommandos

Unsere einfachen Übungsprogramme funktionieren oft nicht richtig, und wenn doch, dann aktivieren wir sie ein einziges Mal, nur um zu sehen, ob sie korrekt sind. Für diese Fingerübungen wäre eine Registrierung beim Betriebssystem viel zu aufwändig. Wir benutzen darum oft eine andere, einfachere Methode, um Programme zu starten: die *Kommandoeingabe*.

Ähnlich wie der Prozessor seine Maschinenbefehle interpretiert (= versteht und ausführt), hat das Betriebssystem eine Menge von Befehlen, die es direkt ausführen kann. Die Befehle nennt man "Kommandos". Die Befehle, die von der CPU verstanden werden, und aus denen ein ausführbares Programm besteht sind kryptische Folgen von Nullen und Einsen. Die Kommandos dagegen sind Texte, die man verstehen kann. Man tippt sie an der Tastatur ein und sie werden dann vom System befolgt. Beispielsweise kann man

```
dir
```

eintippen und das System liefert eine Liste aller Dateien und Unterverzeichnisse im aktuellen Verzeichnis. Die Kommandos werden von einem Teilprogramm des Betriebssystems ausgeführt, das dazu selbst erst gestartet werden muss. Man nennt es allgemein "Kommandointerpretierer". In einem Windowssystem wird der Kommandointerpretierer "DOS–Fenster", "Kommandoeingabe" oder ähnlich genannt. Bei einem Unix–System (Mac oder Linux) nennt man ihn meist "Terminal".

Ein Kommando besteht oft einfach aus dem Namen eines Programms. Genau genommen ist es der Name der Datei die ein Maschinenprogramm enthält. Tippt man ihn ein und schickt ein *Return* hinterher, dann weiß das System, dass es das Programm ausführen soll.

1.1.2 Programme und Algorithmen

Algorithmus: Definition einer zielgerichteten Aktion

Ein *Algorithmus* beschreibt, wie eine komplexe Aufgabe als Folge von einfacheren Aktionen gelöst werden kann. Es ist eine Handlungsanweisung, eine Aussage darüber "wie etwas gemacht wird". Ein Algorithmus beschreibt eine Problemlösung in Form von Einzelschritten. Ein Backrezept ist ein gutes Beispiel für einen Algorithmus. In ihm wird Schritt für Schritt beschrieben, wie aus den Backzutaten ein Kuchen gemacht wird. Ein anderes Beispiel sind die Verfahren zur Addition, Subtraktion und Multiplikation mehrstelliger Zahlen, die man, zumindest früher, in der Grundschule lernte.

"Algorithmus" ist ein informaler und allgemeiner Begriff. Ein Algorithmus sagt, was getan werden muss, um ein Ziel zu erreichen. Das kann richtig oder falsch sein, völlig unabhängig davon, in welcher Form und für wen es aufgeschrieben wurde. Ein Backrezept, in dem der Kuchen 15 Stunden bei 450 Grad gebacken wird, ist höchst wahrscheinlich falsch. Dabei ist es egal in welcher Sprache es aufgeschrieben wurde. Das heißt natürlich nicht, dass die Sprache, in der es verfasst wurde, gleichgültig ist. Algorithmen in einer Formulierung, die niemand versteht, sind nutzlos.

Ein Algorithmus muss nicht nur auf die sprachlichen Fähigkeiten dessen Rücksicht nehmen, der ihn ausführen soll. Auch seine sonstigen Fähigkeiten sind von Belang. So ist der folgende Algorithmus zur Erlangung einer größeren Geldmenge sowohl verständlich, als auch korrekt:

1. Reise in die Zukunft.
2. Stelle fest, welche Lottozahlen gezogen werden.
3. Kehre zurück und fülle einen Lottoschein mit diesen Zahlen.

Die meisten von uns sind aber wohl nicht in der Lage diesen Algorithmus auszuführen. Die Frage, ob er prinzipiell ausführbar ist oder nicht, ist noch offen. Im Allgemeinem setzt man immer dann, wenn informal von einem Algorithmus die Rede ist, voraus, dass er sowohl verständlich ist, als auch ausgeführt werden kann. Neben dem *Was ist zu tun* spielt also auch das *Was kann es tun* eine entscheidende Rolle.

Programme

Für praktische Zwecke muss der Begriff des Algorithmus' präzisiert werden. Wir müssen dabei das Ausführungsorgan des Algorithmus genauer in Betracht ziehen. Es muss die Anweisungen sowohl verstehen als auch ausführen können. Speziell dann, wenn das Ausführungsorgan eine Maschine ist, muss genau festgelegt sein, welche Formulierung welche Aktion genau auslösen soll. So präzise dargelegte Algorithmen nennen wir "Programme".

Programme sind Algorithmen, die in einer bestimmten festen Form aufgeschrieben sind. Die Regeln, nach denen ein Programm aufgeschrieben wird, werden als *Programmiersprache* definiert. Die feste Form und die strengen Regeln sind notwendig, da die

Anweisungen des Programms von einem Rechner (= Computer) ausgeführt werden sollen. Die Programmiersprache legt zum einen die Notation fest und zum anderen sagt sie, was die Formulierungen in dieser Notation bedeuten oder bewirken sollen.

Programmiersprachen bestimmen damit zwei Aspekte eines Programms:

1. *Syntax* (Form): Die exakte Notation in der das Programm als Text aufgeschrieben werden muss.
2. *Semantik* (Inhalt): Was bedeutet ein solcher (Programm–) Text, welche Anweisungen sind möglich und was bewirken (bedeuten) sie genau.

Eine Programmiersprache ist damit eine *formale Sprache*: es ist zweifelsfrei und exakt festgelegt, welche Texte als korrekte Programme dieser Sprache gelten und was sie bedeuten, d.h. was bei ihrer Ausführung zu passieren hat.

Programme unterscheiden sich von Algorithmen durch den höheren Grad an Präzision und Formalität. Sie sind in einer, bis auf das letzte Komma festgelegten Notation zu verfassen und wenden sich an ein Ausführungsorgan, das ein ganz genau definiertes Repertoire an Fähigkeiten hat. Jedes Programm ist ein Algorithmus aber viele Algorithmen sind zu informal um Programme zu sein.

Syntax und Semantik

In der Informatik ist es wichtig zwischen einem Text und dem was er bedeutet zu unterscheiden. Beispielsweise ist "`Kuh`" keine Kuh und "`123`" ist keine Zahl. Beides sind kurze Texte, die aus einer Folge von drei Zeichen bestehen. Die Zeichen des Textes "`123`" stehen für eine Zahl, sie *bezeichnen* oder *bedeuten* eine Zahl. Normalerweise macht es keinen Sinn pedantisch zwischen Texten und dem was sie bedeuten zu unterscheiden. Manchmal ist es jedoch notwendig und in der Informatik ist dieses "manchmal" häufiger als im Alltagsleben.

Die Bedeutung des Textes "`123`" hängt von dem Zahlsystem ab, in dem wir uns bewegen. Im Zehnersystem ist die Zahl hundert–drei–und–zwanzig (123_{10}) gemeint. Im Vierer–System bedeutet er sieben–und–zwanzig ($27_{10} = 123_4$). Im Dreier– oder Zweiersystem ist "`123`" kein gültiger Ausdruck.

Zahlsysteme kann man als sehr einfache Art von formaler Sprache betrachten, die ihre eigene Syntax und Semantik haben. Die Syntax legt fest, welche Texte als korrekte Texte gelten. Wir nennen sie gültige (korrekte) *Zahlausdrücke*. Die Semantik sagt, welche Zahl sie darstellen. Die Syntax des Vierersystems legt beispielsweise fest, dass jede Folge der Ziffern `0`, `1`, `2`, `3` die nicht mit `0` beginnt, ein korrekter Zahlausdruck ist. Die Semantik des Vierersystems legt fest, dass mit einem Zahlausdruck die Zahl gemeint ist, die sich aus der Summe der Zifferwerte, multipliziert mit der jeweiligen Viererpotenz, ergibt ($123_4 = 1*4^2+2*4^1+3*4^0 = 27_{10}$).

Maschinensprache und Maschinenprogramme

Die interessantesten Programme sind die, die von der *Hardware* eines Rechners verstanden und ausgeführt werden. Genauer gesagt ist es der Prozessor, der die Programme ausführt. Wer sonst will schon Programme ausführen. Um zum Ausdruck zu bringen, dass eine Maschine die Programme versteht, nennen wir sie genauer *Maschinenprogramme*.

Die Anweisungen eines Maschinenprogramms werden also vom Prozessor verstanden. Sie sind darum in einer ihm angenehmen Form verfasst, als Folgen aus 0–en und 1–en: Befehle in Form von Bitmustern, von denen jedes eine Bedeutung hat, die auf die Fähigkeiten des Prozessortyps zugeschnitten ist. Die Fähigkeiten eines Prozessors darf man dabei nicht überschätzen. Viel mehr als einfache arithmetische Operationen, das Verschieben von Daten (auch wieder Bitmuster) von einer Speicherstelle zur anderen und das Laden neuer Anweisungen von bestimmten Speicherstellen im Hauptspeicher, ist nicht möglich.

Menschen sind kaum in der Lage Maschinenprogramme zu lesen, geschweige denn korrekte Maschinenprogramme zu schreiben. Die Programme der frühen Pioniere der Informatik wurden zwar in Maschinensprache verfasst. Sehr schnell hat man dann aber eingesehen, dass Bitmuster, die das Verändern und Verschieben von Bitmustern beschreiben, nicht unbedingt eine für Menschen geeignete, geschweige denn angenehme Art sind, einen Algorithmus zu beschreiben.

Für Menschen ist es einfach eine Zumutung, sich auf die Ebene eines Prozessors zu begeben. Prozessoren, die in der Lage sind Anweisungen auf dem Niveau von Menschen zu bearbeiten, sind dagegen zumindest vorerst technisch und ökonomisch nicht realisierbar. Die Lösung des Problems besteht darin, Menschen Programme in "menschlicher" Form schreiben zu lassen und sie dann in Maschinensprache zu übersetzen. Das Übersetzen sollte dabei am besten von einem Computer übernommen werden.

Höhere Programmiersprache

Programme in einer *höheren Programmiersprache* enthalten Anweisungen, die sich in Form und Inhalt an den Fähigkeiten von Menschen orientieren. Beispiele für solche Sprachen sind *C, C++, Visual Basic* und eben die Sprache *Java* mit der wir uns hier

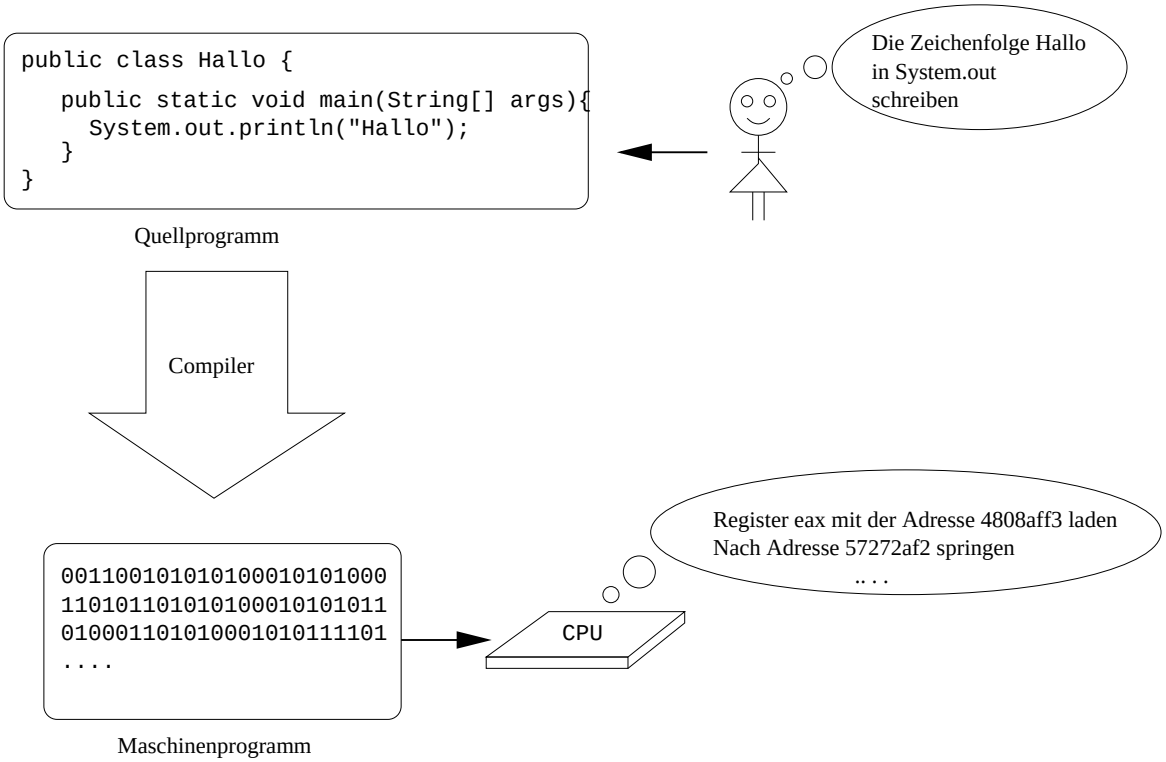


Abbildung 1.2: Quellprogramm, Compiler, Maschinenprogramm

näher beschäftigen wollen. Programme in höheren Programmiersprachen bestehen aus Anweisungen die Menschen verstehen und schreiben können. Beispielsweise ist

```
public class HalloWelt {
  public static void main(String[] args) {
    System.out.println("Hallo Welt");
  }
}
```

ein Programm in der höheren Programmiersprache Java. Man sieht es vielleicht nicht auf den ersten Blick, aber es soll den Computer dazu bringen "`Hallo Welt!`" auf dem Bildschirm auszugeben. Solche Programme sollen Menschen leicht schreiben oder verstehen können – zumindest wenn sie Programmierer sind. Zumindest ist es leichter zu verstehen als ein Maschinenprogramm in der Form:

```
48
00 00
00 90 88 04 08 04
00 00
00 21
00 0e
00 51 01
00 00
90
9a 04 08 8c 00 00 00
... noch viele derartige Zeilen ...
```

Der Nachteil der Programme in höheren Programmiersprachen ist, dass es keinen Prozessor – also keine Maschine – gibt, die ihre Anweisungen versteht. Nicht nur die Notation ist dabei für den Prozessor unverständlich, es werden auch Aktionen von ihm verlangt, die er in dieser Form nicht ausführen kann.

Die Programme der höheren Programmiersprachen enthalten Anweisungen an eine virtuelle Maschine, also eine nur gedachte Maschine. Im Java–Programm steht mit

```
...
System.out.println("Hallo Welt");
...
```

so etwas wie “Gib Hallo Welt aus!”. Das ist für den Prozessor so unverständlich und unlösbar, wie die Aufforderung “Backe eine köstliche Sahnetorte!” für den Autor dieser Zeilen. So wie ich *im Prinzip* eine Sahnetorte backen kann, wenn man mir haarklein jeden einzelnen Schritt erklärt, so kann der Prozessor “Hallo” ausgeben. Man muss ihm nur jeden Schritt dazu genau erklären. Diese Erklärung steht aber nicht in diesem Programm. Der “Befehl” `System.out.println` wendet sich darum an jemanden, der nicht exisitiert – an eine gedachte Maschine eben.

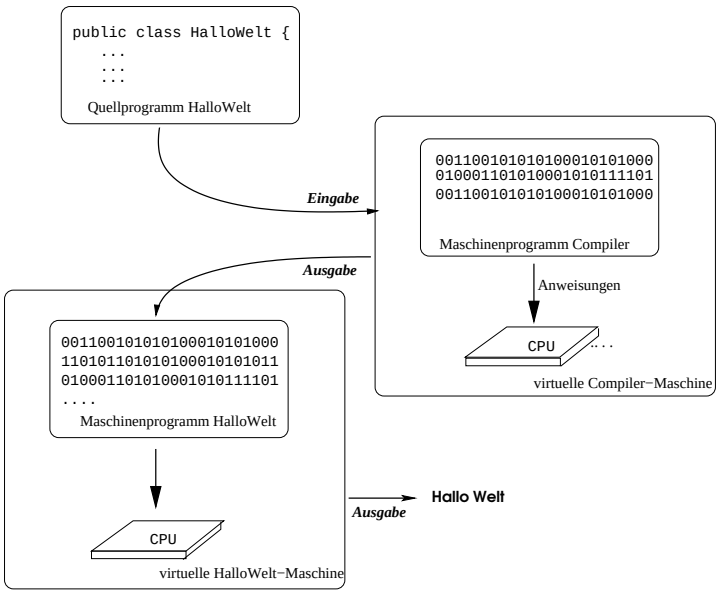


Abbildung 1.3: Ausführung via Compiler

Compiler: Programm in Maschinenprogramm übersetzen

Die Lücke zwischen einem Programm, das Menschen konzipieren und schreiben können und dem Prozessor, der nur mit 0–en und 1–en hantieren kann, wird vom Compiler gefüllt.

Ein *Compiler* ist ein Programm, das ein Programm in einer höheren Programmiersprache, das *Quellprogramm*, in ein äquivalentes Maschinenprogramm übersetzt. (Siehe Abbildung 1.2). Die für Menschen verstehbaren Anweisungen an eine gedachte (virtuelle) Maschine werden dabei in maschinenlesbare Anweisungen an den Prozessor umgesetzt. Das sollte er natürlich so tun, dass das Quellprogramm und das erzeugte Maschinenprogramm im Endeffekt die gleiche Wirkung haben. In der Regel ist der Compiler korrekt und tut was man von ihm erwartet. Das Quell–Programm von oben wird vom Compiler in ein Maschinenprogramm – eine lange unverständliche Folge von Nullen und Einsen – übersetzt, die der Prozessor ausführen kann und dabei das Gewollte tut, nämlich `Hallo Welt!` auf dem Bildschirm ausgeben.

Übersetzen: Quellcode $\overset{\text{Compiler}}{\longrightarrow}$ Maschinencode

Ausführen: Eingabedaten $\overset{\text{Maschinencode}}{\longrightarrow}$ Ausgabedaten

Interpreter und virtuelle Maschinen

Ein Progamm einer höheren Programmiersprache richtet sich in Form und Inhalt – wir Informatiker sagen *Syntax* und *Semantik* – mehr nach den Menschen als nach den Maschinen. Will man ein solches Programm ausführen, dann kann man es durch einen Compiler in ein Maschinenprogramm übersetzen. Dabei müssen wir im Auge behalten, dass weder der Compiler, noch das übersetzte Programm selbst wirklich etwas tun. Es handelt sich lediglich um Software. Erst in Kombination mit der Hardware, dem Prozessor, wird daraus etwas das sich bewegen kann. In dem einen Fall ist die Bewegung gering: es wird lediglich “Hallo Welt” ausgegeben. Grundsätzlich können natürlich beliebig komplexe Berechnungen ausgeführt werden. In der Zeit, in der das (Maschinen–) Programm ausgeführt wird, bilden die Software des Programms und die Hardware des Rechners eine Einheit:

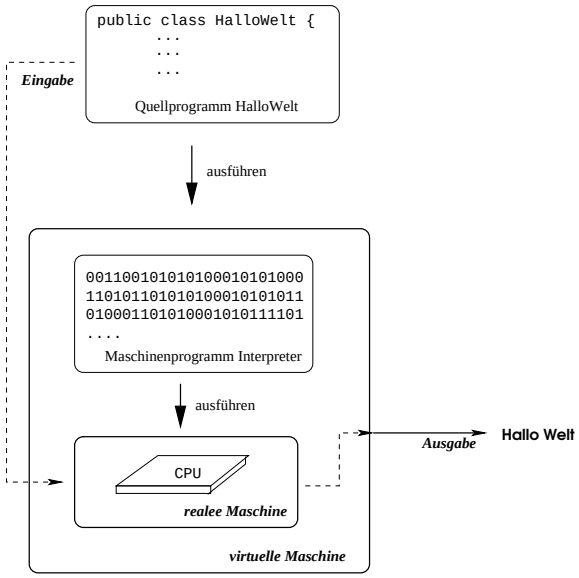


Abbildung 1.4: Ausführung durch Interpreter

eine spezielle Maschine zur Ausgabe von “Hallo Welt”. Diese Maschine besteht nicht nur aus physischer Materie und exisitiert auch nur kurze Zeit. Man nennt darum die Kombination Hardware plus Software oft *virtuelle Maschine*.

Virtuelle Maschine = Maschinencode eines Programms + Hardware eines Rechners

Wie Informatiker so sind, vergessen sie oft die Hardware und nennen schon das Programm allein *virtuelle Maschine*. Im Falle unseres Hallo–Welt–Programms handelt es sich dann um eine *virtuelle Hallo–Welt–Maschine*. (Siehe Abbildung 1.3.) Natürlich käme niemand auf die Idee bei einem solchen Programm eine so hochtrabende Bezeichnung wie *virtuelle Maschine* zu verwenden.

Neben der Methode des Übersetzens durch einen Compiler gibt es noch eine alternative Methode um Programme höherer Programmiersprachen auszuführen: den *Interpreter*. Ein Interpreter (auch *Interpretierer*) ist wie der Compiler ein (Maschinen–) Programm. Statt aber wie der Compiler ein vorgegebenes Programm in äquivalente Anweisungen einer anderen Sprache zu übersetzen, führt der Interpreter diese Anweisungen gleich selbst aus. Der Compiler verhält sich wie ein Übersetzer, der Anweisungen von Englisch nach Deutsch übersetzt. Der Interpreter dagegen würde sie gleich selbst ausführen (ausführen = interpretieren). Genauer gesagt, agiert der Interpretierer wie ein Simultan–Dolmetscher, dessen übersetzte Anweisungen sofort von der Hardware ausgeführt werden.

Interpretieren: Quellcode + Eingabedaten $\overset{\text{Interpreter}}{\longrightarrow}$ Ausgabedaten

Ein Interpreter ist natürlich zunächst einmal nur Software, eine aneinander gereihte Folgen von Nullen und Einsen. Diese Software wird erst in Gegenwart von Hardware lebendig: Die Kombination Hardware plus Interpreter–Software ist dann in der Lage die Quellprogramme direkt auszuführen. In dem Fall ist es sinnvoll und auch üblich von einer virtuellen Maschine zu sprechen. Die Software des Interpreters macht aus einer Maschine, die Programme in Maschinensprache versteht und ausführt, eine *virtuelle Maschine*, die Programme in der höheren Sprache versteht und ausführt. (Siehe Abbildung 1.4.)

Java und die JVM

Bei dem Konzept der virtuellen Maschinen handelt es sich nicht, wie man vielleicht zunächst denken mag, um sinnlose Gedankenspielereien. Nun, vielleicht sind es Gedankenspielereien, aber sie sind nicht sinnlos.[1] Bei Java[2] wird das Konzept der virtuellen Maschine in seiner speziellen Variante eingesetzt. (Siehe Abbildung 1.5.) Das Quellprogramm wird zwar übersetzt, aber nicht in Maschinencode, sondern in den Code einer hypothetischen, ausgedachten, *virtuellen Maschine*, der *Java Virtual Machine – JVM*. Bei der JVM handelt es um ein (Maschinen–) Programm, das den vom Java–Compiler erzeugten Code interpretiert. Zur Ausführung von Java–Programmen wird also eine Kombination aus Compiler und Interpreter eingesetzt.

[1] Anfänger haben oft Probleme solche Gedankenspiele ernst zu nehmen und sich auf sie einzulassen. Die Informatik ist aber nun mal die Wissenschaft der Gedankenspielereien. Es ist darum nicht so verwunderlich, dass sie eine wichtige Rolle spielen.

[2] und anderen modernen Programmiersprachen wie etwa C#

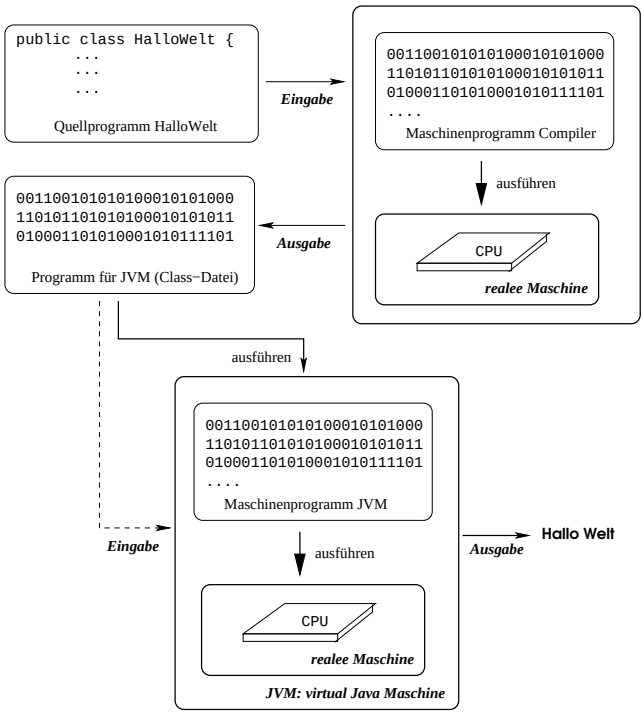


Abbildung 1.5: Konzept der Ausführung von Java–Programmen

Sowohl der Compiler, als auch die JVM liegen in dem eben geschilderten Szenario als Maschinenprogramm vor. Selbstverständlich schreibt niemand Maschinenprogramme. Beide werden also als Quellprogramme verfasst worden sein, die dann von einem Compiler übersetzt wurden. Es könnte auch sein, dass JVM und Compiler als Quellprogramm von einem Interpreter abgearbeitet werden. Vielleicht werden sie auf weiteren virtuellen Maschinen ausgeführt. Deren Quellcode wurde eventuell wieder von einem Compiler übersetzt, oder er wird von einem Interpretierer interpretiert, oder ... Sie sehen, ganz so trivial sind die Gedankenspiele der Informatik nicht.

Bytecode und virtuelle Maschine

Java–Programme werden, wie oben angesprochen, übersetzt. Der dabei erzeugte Maschinencode wird oft *Bytecode* (oder gelegentlich auch *Class-Code*) genannt, wird aber typischerweise nicht von einer Maschine ausgeführt. Es ist ein weiteres Programm, ein Interpretierer, ist notwendig um ihn auszuführen. Diesen Bytecode–Interpretierer nennt man *virtuelle Maschine*.

Die Verarbeitung von Daten mit einem Javaprogramm läuft darum immer in dieser Form ab:

Übersetzen: Java–Code $\overset{\text{Compiler}}{\longrightarrow}$ Bytecode

Interpretieren: Byte–Code + Eingabedaten $\overset{\text{JVM}}{\longrightarrow}$ Ausgabedaten

Nun mag man sich fragen, warum so umständlich? Warum wird nicht entweder direkt in Maschinencode übersetzt? Oder, wenn denn schon interpretiert werden soll, warum wird nicht gleich der Quellcode interpretiert? Die Antwort ist recht einfach: Das Interpretieren des Quellcodes ist zu langsam und “richtiger Maschinencode” ist auf richtige, ganz bestimmte richtige, Maschinen festgelegt. Er kann nicht auf einer Maschine ausgeführt werden, für die er nicht erzeugt wurde. Bytecode kann dagegen überall dort ausgeführt werden, wo es eine virtuelle Maschine gibt. Diese Art der Mobilität war ein wichtiges Entwurfsziel von Java. Sie ermöglicht es, dass ein Rechner (fast) lauffähigen Code von einem anderen Rechner lädt, ohne dass dieser irgendetwas über die Hardware des Zielrechners weiß. Die Methode kombiniert die Flexibilität des Interpretierens mit der Geschwindigkeit des Übersetzens.

Programme und Software

Die scheinbare Leichtigkeit, mit der Programme entwickelt werden können, und die fehlende materielle Komponente verleiten dazu, ihren Wert zu unterschätzen. Tatsächlich hat Software heutzutage einen beträchtlichen Anteil am Wert und den Kosten

vieler Produkte. Der PC spielt dabei eine Rolle, aber nicht die einzige. Praktisch jedes technische Produkt enthält inzwischen einen erheblichen Anteil an Software, deren Zuverlässigkeit über Wohl und Wehe von Weltfirmen mitentscheiden kann. Die Effizienz mit der administrative Vorgänge in Banken, Behörden und Wirtschaftsunternehmen abgewickelt werden können entscheidet mit über deren Rentabilität und damit deren langfristige Existenz.

Die Fähigkeit, Software von guter Qualität zu günstigen Preisen herstellen zu können, wird in Zukunft von entscheidender Bedeutung für den Wohlstand ganzer Regionen und Volkswirtschaften sein. Trotzdem herrscht oft die Vorstellung, dass Software nichts anderes ist, als ein etwas größeres Programm an dem ein paar Hacker mehr herum gehackt haben. Software ist die Steuerung komplexer Systeme und Vorgänge. Sie wird letztlich in Form von Programmen realisiert, ist aber viel mehr. Sie umfasst alle Aspekte der Analyse, der Konzeption und Planung von Steuerungsvorgängen sowie der langfristigen Wartung der erstellten Programme. Dazu gehört viel mehr als nur die Beherrschung einer Programmiersprache und die Fähigkeit ein Programm von ein paar Hundert Zeilen schreiben zu können. Umgekehrt kann aber ohne diese Fähigkeiten auch keine Software erstellt werden.

Ein Architekt muss mehr können als mauern, aber ohne mauern zu können, kann man kein Haus bauen. Genauso muss ein Software–Ingenieur mehr können als programmieren, aber ohne das ist Sie oder Er Nichts. Beginnen wir also damit Stein auf Stein zu setzen.

Zusammenfassung

Fassen wir noch einmal kurz die wichtigsten Begriffe zusammen:

- *Algorithmus*: Ein Algorithmus ist ein Verfahren nach dem eine Aufgabe erfüllt werden kann.
- *Programm*: Ein Programm ist die Beschreibung eines Algorithmus' in einer bestimmten fest definierten Notation (Syntax) und klar definierter Bedeutung (Semantik) das sich an ein Ausführungsorgan mit exakt definierten Fähigkeiten wendet. Es ist entweder ein Maschinenprogramm oder ein Programm in einer höheren Programmiersprache.
- *Programmiersprache*: Festlegung der Syntax und Semantik von Programmen (Programm–Texten). Man unterscheidet Maschinensprachen und höhere Programmier–Sprachen. Programmiersprachen sind formale Sprachen.
- *Maschinenprogramm*: Ein Programm, das vom Prozessor eines Rechners verstanden und ausgeführt werden kann.
- *Programm in einer höheren Programmiersprache*: Ein Programm in einer höheren Programmiersprache kann von Menschen geschrieben und verstanden werden. Es kann nur von virtuellen (gedachten) Maschinen direkt ausgeführt werden. Um es auf realen Maschinen auszuführen benötigt man einen Compiler.
- *Compiler*: Ein Programm das ein Quellprogramm in ein äquivalentes Maschinen– oder Bytecodeprogramm übersetzt.
- *Quellprogramm*: Ein Programm in einer höheren Programmiersprache. Es wird vom Compiler in ein Maschinenprogramm übersetzt. Es ist die "Quelle" des Übersetzungsvorgangs.

1.2 Java–Programme schreiben und ausführen

1.2.1 Hallo Welt

Das Hallo Welt Programm

Ein häufiges und, auf den ersten Blick, ziemlich nutzloses Programm ist das sogenannte *Hallo Welt Programm*: Ein Programm, das nichts anderes tut, als den Text *Hallo Welt* auszugeben. Zwar ist es ohne jeden Nutzen, von seinem eigenen Programm freundlich begrüßt zu werden, es ist aber nicht sinnlos ein solches Programm zu schreiben. Mit ihm kann man testen, ob und wie es möglich ist, ein Programm "zum Laufen" zu bringen. Da Programmieren mindestens so viel Können wie Wissen ist, und Können auch mehr Spaß macht als nur zu wissen, sollte alles an Erkenntnisgewinn auch gleich ausprobiert werden.

Also hier ist das *Hallo Welt Programm* in Java:

```
package hallo;

/**
 * Diese Klasse implementiert eine Anwendung die
 * "Hallo Welt" auf der Konsole ausgibt.
 */
public final class HalloWelt {
  private HalloWelt(){}
  public static void main(String[] args) {
    System.out.println("Hallo Welt");
 }
}
```

Es besteht aus einer öffentlichen (*public*), finalen (*final*) Klasse (*class*) mit dem Namen `HalloWelt` als Bestandteil des Pakets *hallo*.

Die Bestandteile des Textes sind im Einzelnen:

- Eine Paket–Deklaration **package** `hallo;` Mit ihr wird gesagt, dass die Klasse zum Paket *hallo* gehört.
- Ein Kommentar zwischen `/*` und `*/`, der die Aufgabe der Klasse `HalloWelt` beschreibt. Kommentare sind für den menschlichen Leser gedacht, sie beeinflussen das Verhalten des Programms nicht.
- Eine private (*private*) Methode mit dem Namen `HalloWelt`, die Konstruktor genannt wird, sowie
- Eine öffentliche statische (*static*) Methode (Funktion) mit Namen `main`. In `main` wiederum finden wir
- Eine Ausgabe-Anweisung `System.out.println("Hallo Welt");`
  mit der der Text "`Hallo Welt`" auf der Konsole ausgegeben wird.

Java ist eine Sprache für professionelle Software–Entwickler. Ihre Programme wirken manchmal etwas geschwätzig, denn sie enthalten Elemente, die für große Programmsysteme wichtig, für winzige Spielprogramme aber unnötig sind und die Anfänger gelegentlich verwirren. Wir nehmen diese Elemente vorerst einmal einfach so hin. Später kommen wir natürlich darauf zurück: wir wollen ja keine Anfänger bleiben, sondern möglichst schnell professionell werden.

Kommentare Außer der Klasse und der in ihr enthaltenen Methode enthält das Programm noch einen *Kommentar*. Kommentare sind die Textbereiche zwischen `/*` und `*/`. Kommentare sind Bestandteile eines Programms, die bei der Ausführung vollständig ignoriert werden. Sie haben keinen Einfluss auf das Verhalten des Programms sondern dienen nur dazu, es für menschliche Leser besser verständlich zu machen.

Kommentare können noch in einer zweiten Form als Zeilenende–Kommentare auftreten. Man schreibt zwei Schrägstriche und dahinter ist alles bis zum Zeilenende ein Kommentar. Beispiel:

```
package hallo;                    // Paketdeklaration

public class HalloWelt {          // Beginn der Klassendefinition

  private HalloWelt(){}           // Konstruktor (unterdrueckt)

  public static void main(String[] args) { // Beginn der Methodendefinition
    System.out.println("Hallo Welt"); // Anweisung
  }                               // Ende der Methodendefinition
}                                 // Ende der Klassendefinition
```

Struktur Das kleine Java–Programm ist also ein Text mit einer geschachtelten Struktur:

- Eine **Anweisung** zum Drucken einer Zeile (`println` = *Print Line*)
- **in** der Definition einer **Methode** (oder auch **Funktion**) mit Namen `main`
- **in** der Definition einer **Klasse** mit Namen `HalloWelt`
- **in** einem **Paket** mit Namen `hallo`.

Ein Programm hat also Elemente wie *Anweisungen*, *Methoden*, *Klassen* und *Pakete*. Es hat eine Struktur, die man am Aufbau des Textes erkennt. Wir wollen uns aber vorerst nicht weiter damit aufhalten, was das alles im Detail bedeutet, wir wollen unser Programm "laufen" sehen: Es ist nur ein Programm, Text der für sich allein nichts tut, außer zu sagen, was getan werden soll. Jemand oder etwas muss es ausführen.

Programm mit einem Texteditor erstellen und ausführen

Um ein Java–Programm zu erstellen benötigt man nichts weiter als einen Texteditor.[3] Der Text einer Klasse wird in eine Datei geschrieben und abgespeichert. Die Datei muss den Namen der Klasse entsprechen und die Erweiterung `.java` haben. Der Name sollte stes mit einem Großbuchstaben beginnen. Der Quelltext der Klasse `HalloWelt` beispielsweise wird in der Datei `HalloWelt.java` gespeichert. Klassen werden also in Dateien gespeichert. Im Regelfall enthält eine Datei eine Klasse und jede Klasse hat ihre Datei.

Pakete finden ihre Entsprechung in Verzeichnissen.[4] Zu jedem Paket muss es ein Verzeichnis geben, das den gleichen Namen wie das Paket hat. Die Datei, in der sich der Text einer Klasse befindet, muss in dem Verzeichnis abgespeichert werden, das dem Paket entspricht, zu dem die Klasse gehört. In unserem Beispiel gehört die Klasse `HalloWelt` zum Paket `hallo`. Die Datei `HalloWelt` wird darum im Verzeichnis `hallo` gespeichert.

Klasse	`HalloWelt`	→	Datei	*HalloWelt.java*
Paket	`hallo`	→	Verzeichnis	*hallo*
Klasse	`HalloWelt` im Paket `hallo`	→	Datei	*HalloWelt.java* im Verzeichnis *hallo*

Man beachte auch die Konventionen für Namen: Der Name einer Klasse beginnt immer mit einem Großbuchstaben. Der Name eines Pakets beginnt immer mit einem Kleinbuchstaben.

Um das Programm ausführen zu können muss es zunächst übersetzt werden. Dazu benötigt man eine Konsole.[5] Man öffnet eine Konsole und macht das Verzeichnis, in dem sich das Verzeichnis *hallo* befindet, zum aktuellen Verzeichnis.[6] Mit der Eingabe eines Kommandos wie beispielsweise

```
javac hallo/HalloWelt.java[7]
```

wird dann `javac`, der Java–Compiler, aktiviert. Er übersetzt den Quellcode in der Datei `HalloWelt.java` im Verzeichnis `hallo` und erzeugt eine Datei mit dem Namen `HalloWelt.class` im Verzeichnis `hallo`. Diese Datei kann mit dem Kommando

```
java hallo.HalloWelt
```

ausgeführt werden. Mit `java` wird die virtuelle Maschine aktiviert und

```
hallo.HalloWelt
```

ist der *vollständige Name* der Klasse. Der vollständige Name einer Klasse besteht also aus dem Paketnamen und dem Klassennamen. Voraussetzung ist hierbei dass das aktuelle Verzeichnis der Konsole das Verzeichnis oberhalb von `hallo` ist. Die Klasse kann von einem beliebigen Punkt aus aktiviert werden, wenn dieses Verzeichnis mit angegeben wird. Man nennt dies den Klassenpfad: den Pfad zu dem oder den Verzeichnisse(n) mit den Klassen–Dateien:

`java -cp` *<Wurzel der Paketverzeichnisse>* *<vollständiger Klassenname>*

`cp` steht für *class path* (Klassenpfad) und *<Wurzel der Paketverzeichnisse>* steht für einen Pfadausdruck mit dem man ein Verzeichnis angibt. Das Kommando könnte konkret

```
java -cp /meineDaten/javaVerzeichnis/ hallo.HalloWelt
```

[3] Es muss ein einfacher Texteditor sein, nicht *Word* oder ein anderes Programm, das es erlaubt, Texte mit einer bestimmten Formatierung (fett, kursiv, unterschiedliche Schriftgrößen, etc.) zu erstellen.

[4] Statt von *Verzeichnissen* sprechen viele auch von *Ordnern*. Ein anderes Wort mit der gleichen Bedeutung ist *Directory*.

[5] Eine Konsole erlaubt die Eingabe von Kommandos, also mit der Tastatur getippten Anweisungen. Statt "Konsole" findet man auch die Bezeichnungen "Eingabeaufforderung" oder "Terminal".

[6] Mit dem Kommando `cd` wechselt man in ein Verzeichnis (macht es zum aktuellen Verzeichnis).

[7] Je nach Betriebssystem ist "/" durch "\" zu ersetzen.

lauten. (Beachten Sie die Leerzeichen!) Im Verzeichnis `/meineDaten/javaVerzeichnis/` müsste sich dann das Verzeichnis `hallo` und in diesem die Datei `HalloWelt.class` finden.

Kurz zusammengefasst: Mit den folgenden Schritten kann eine sehr einfache Java-Anwendung, das Hallo-Welt-Programm, erstellt und ausgeführt werden:

1. Installiere ein aktuelles Java JDK. (Herunterladen und Installationsanweisungen beachten.)
2. Erzeuge eine Verzeichnis, z.B. `JavaBeispiele`.
3. Erzeuge eine Unterverzeichnis `JavaBeispiele\hallo`.
4. Erzeuge in `hallo` eine Textdatei `HalloWelt.java`.
5. Fülle die Datei `HalloWelt.java` mit dem Text des Javaprogramms.
6. Öffne eine Konsole und mache `JavaBeispiele` zum aktuellen Verzeichnis: Tippe `cd JavaBeispiele`
7. Übersetze das Quellprogramm: Tippe `javac hallo\HalloWelt.java`.
8. Führe das übersetzte Programm aus: Tippe `java hallo.HalloWelt`.

Linux– und Mac–User verwenden dabei / statt \.

Ein Programm mit Eclipse erstellen

Die Organisation der Dateien und Verzeichnisse, das explizite Aufrufen des Compilers und der Umgang mit Kommandos an einem Terminal ist etwas mühsam. Man kann es sich von einer integrierten Entwicklungsumgebung (IDE – *Integrated Development Environment*) abnehmen lassen. Die bekannteste, weitverbreiteste und dazu noch kostenlose Entwicklungsumgebung für Java ist *Eclipse*.[8] Es handelt sich um ein komplexes Programm das nicht ganz leicht zu beherrschen ist. Die Mühe, den Umgang mit Eclipse zu erlernen, macht sich aber schnell bezahlt. Größere Programme können zudem auch kaum ohne die Hilfe einer IDE erstellt werden. Irgendwann muss also der Umgang mit *Eclipse* erlernt werden – warum nicht jetzt?

Um das Programm zu erstellen, starten wir die Entwicklungsumgebung *Eclipse* und wählen im *File*–Menü

New → *Project*

aus. Wir sagen dem *Wizard*,[9] der daraufhin erscheint, dass wir ein Java–Projekt erstellen wollen und geben ihm einen Namen, z.B. `Lektion1`. Jetzt wählen wir *File* → *New Class* und es erscheint ein *Wizard* der uns bei der Klassenerstellung hilft.

Wir tippen den Klassennamen ein und kreuzen `public`, `final` und `public static void main` an. Eclipse erzeugt für uns dann die Klasse und die gewünschte Methode als *Stub* (Stummel), ein Textrahmen der weiter mit Text gefüllt werden kann. In den Text tippen wir den fehlenden Code unseres Beispiels.[10]

Das Programm ausführen

Das Programm ist jetzt fertig. Wir speichern es (Disketten–Icon!)[11] und dann kann es ausgeführt werden. Dazu wählen wir das Menü

Run → Run...

in dem dann erscheinenden Fenster wählen wir *Java Application*, klicken den *new*–Knopf und auf *run* und schon erscheint im Konsolenfenster unten der Text:

```
Hallo Welt
```

Noch einfacher geht es, wenn per Rechtsklick in das Editorfenster geklickt und `Run As Java Application` ausgewählt wird. Ist das Programm einmal auf diese Art gestartet worden, kann einfach der weiße Pfeil auf grünem Grund geklickt werden.

---

[8] `http://www.eclipse.org`

[9] Ein *Wizard* ist kein Zauberlehrling, sondern ein Dialogfenster, das bei bestimmten Aufgaben hilft.

[10] Eine ausführlichere Anleitung zur Bedienung von *Eclipse* kann hier nicht gegeben werden. Wir verweisen den Leser an die zahlreichen Quellen im Internet. Unter `http://help.eclipse.org/indigo/index.jsp` findet sich eine Anleitung zur aktuellen Version von September 2011.

[11] Das ist wohl das Schicksal der Diskette, sie wird als Ikone überleben. In wenigen Jahren werden Nachdenkliche sich fragen, warum "Speichern" durch ein solch seltsames Symbol dargestellt wird.

Projekte

Nach diesem ersten erfolgreichen Programmlauf wollen wir uns die Konstrukte jetzt etwas genauer ansehen. Als erstes haben wir ein *Projekt* erzeugt. Ein Projekt ist kein Bestandteil der Programmiersprache Java, es ist eine Organisationseinheit von Eclipse, eine sogenannte *Ressource*. Projekte werden als Verzeichnisse im *Workspace* von Eclipse gespeichert. Haben wir beispielsweise das Verzeichnis `eclipse` als *Workspace* ausgewählt und ein Projekt mit dem Namen *Lektion1* erzeugt, dann finden wir dort die Verzeichnisse `src` und `bin`. In beiden gibt es ein Verzeichnis mit dem Namen `Lektion1`.

Pakete

In den Verzeichnissen `src\Lektion1` und `bin\Lektion1` finden wir jeweils ein Verzeichnis `hallo`, das dem Paket entspricht. Dem Paket `hallo` entsprechen also jetzt zwei Verzeichnisse. Sie enthalten jetzt getrennt die Dateien, die zum Paket gehören. In `Lektion1\src\hallo` liegt die Datei `HalloWelt.java` und in `Lektion1\bin\hallo` die Datei `HalloWelt.class`.

```
+ Lektion1
    + src
        + hallo
            - HalloWelt.java
    + bin
        + hallo
            - HalloWelt.class
```

`HalloWelt.java` enthält den eingetippten Quelltext. `HalloWelt.class` enthält die Übersetzung des Quelltexts. Diese Datei hat Eclipse für uns mit Hilfe des Java–Compilers erzeugt. Sie wird von der JVM ausgeführt, wenn das Programm gestartet wird.

Möglicherweise hat Eclipse die Quell-Dateien und die Klassen-Dateien in einer statt in zwei Verzeichnisstrukturen (`bin` und `src`) gespeichert. Die Aufteilung in ein oder zwei Verzeichnisstrukturen ist eine Option, die beim Anlegen eines Projekts ausgewählt werden kann. Es ist allerdings üblich Quelldateien und übersetzte Dateien in unterschiedlichen Verzeichnissen zu organisieren.

Klassen

Im Gegensatz zu Projekten und so wie Pakete sind Klassen ein fundamentaler Bestandteil der Programmiersprache Java. Jedes Java–Programm besteht aus mindestens einer Klassendefinition. Eine Klasse hat einen Namen. Nach einer allgemein anerkannten Konvention beginnt der Name einer Klasse mit einem Großbuchstaben. Klassendefinitionen werden in Dateien gespeichert, die den gleichen Namen wie die Klasse und die Erweiterung `.java` haben. Eclipse übernimmt für uns die Organisation und Speicherung der Klassen.

Methoden und Anweisungen

Über Klassen wird noch viel zu sagen sein. Vorerst betrachten wir eine Klasse einfach als einen Behälter für Methoden. In unserem kleinen Beispiel ist `main` die einzige Methode der Klasse `HalloWelt`. Das lässt sich ändern. Wir fügen eine weitere Methode ein und übertragen ihr die Aufgabe “Hallo Welt” zu sagen:

```
package hallo;
public final class HalloWelt {

  private HalloWelt(){}         // Konstruktor, vorerst ignoriert

  public static void main(String[] args) {
    sagHallo();                 // Aufruf der neuen Methode
  }

  private static void sagHallo() { // neue Methode
    System.out.println("Hallo Welt");
  }
}
```

Projekte, Klassen und Methoden sind Strukturierungsmittel. Wirklich etwas getan wird erst, wenn eine *Anweisung* ausgeführt wird. Methoden enthalten Folgen von Anweisungen. Die beiden Methoden in unserem Beispiel haben jeweils nur eine einzige Anweisung. In `sagHallo` wird der gewünschte Gruß ausgegeben, in `main` wird `sagHallo` aktiviert. Statt "aktiviert" sagt man auch "aufgerufen".

Die main–Methode

Eine Anweisung wird aktiviert, wenn die Methode, in der sie steckt, aktiviert wurde und sie "an der Reihe" ist. Methoden werden von anderen Methoden aus durch eine Aufruf–Anweisung ((Methoden–) Aufruf) aktiviert. Das alles muss einen Anfang haben, und das ist die Methode `main`. Jedes Java–Programm beginnt damit, dass irgendwie *von außen* die main–Methode einer Klasse ausgeführt wird.

Compiler

Verwenden wir Eclipse oder eine andere Entwicklungsumgebung, dann ist dies alles, was notwendig ist, um `main` zu starten. Java Programme können aber auch ohne die Hilfe einer Entwicklungsumgebung ausgeführt werden. Sie müssen dazu zuerst *übersetzt* werden. Das ist die Aufgabe des Java–*Compilers*. Ein Compiler nimmt ein *Quell–Programm* und erzeugt daraus ein *ausführbares* Programm. Wie bereits weiter oben gesagt, gibt man dazu in einer Konsole ein Kommando ein

```
javac HalloWelt.java
```

`javac` ist der Compiler (`java c` ompiler). Das Quell–Programm ist der Text, den die / der Programmierer(in) getippt hat und der hier in der Textdatei `HalloWelt.java` zu finden ist. Der Compiler erzeugt daraus eine Datei `HalloWelt.class`.

Programmausführung: virtuelle Maschine

Die Datei `HalloWelt.class` enthält das Programm in einer anderen Form. Es ist vom Compiler vom menschenlesbaren Quellprogramm in ein *Maschinenprogramm* umgeformt worden und kann jetzt ausgeführt werden, beispielsweise indem wir eingeben:

```
java HalloWelt
```

In diesem Kommando ist `java` die *virtuelle Maschine (JVM)* und mit `HalloWelt` ist die Klasse `HalloWelt` gemeint, deren Maschinencode in der Datei `HalloWelt.class` steckt.

Der (Maschinen–) Code in `HalloWelt.class` wird *Bytecode* genannt. Er kann von einer speziellen, auf Java hin optimierten Hardware ausgeführt werden. In der Regel sind die Rechner nicht mit solcher Hardware ausgestattet. Damit sich trotzdem etwas tut, wird diese spezielle Hardware von einem Programm namens `java` simuliert. (Das Programm `java` wird durch das Kommando `java` aktiviert.)

`java` führt jetzt den Bytecode der Klasse aus. Es ist, wie gesagt keine richtige Hardware, sondern selbst auch wieder nur ein Programm. Es ist nur eine scheinbare (virtuelle) Hardware. Man nennt es darum die *virtuelle Java Maschine* (*Java Virtual Machine, JVM*) oder kurz *virtuelle Maschine*. Letztlich kann aber nur richtige, stromverbrauchende, wärmeerzeugende Hardware etwas tun. Die virtuelle Maschine ist darum in "echtem Maschinencode" geschrieben, und kann von der "echten Hardware" ausgeführt werden. (Siehe Abbildung 1.6) Die CPU[12] führt den Code der virtuellen Maschine (`java`) aus. Deren Befehle lesen und interpretieren (während sie ausgeführt werden) den Code von `HalloWelt.class` und dann passiert das was in `HalloWelt.java` angeordnet wurde.

Die Java–Maschine muss nicht unbedingt virtuell sein, man kann Hardware produzieren, die den Java Byte–Code direkt ausführen kann (und hat dies auch tatsächlich getan).

Unser erstes Programm läuft jetzt und wir haben ein erstes Verständnis dessen, was dabei passiert.

1.2.2 Programme erzeugen und ausliefern

Quellprogramme

Fassen wir noch einmal kurz alles zusammen, was einem kleinen Java–Programm gehört:

[12] CPU: *Central Processing Unit*, auch *Prozessor* genannt, ist der Teil des Rechners, der die Maschinenbefehle ausführt, der Teil der wirklich etwas tut.

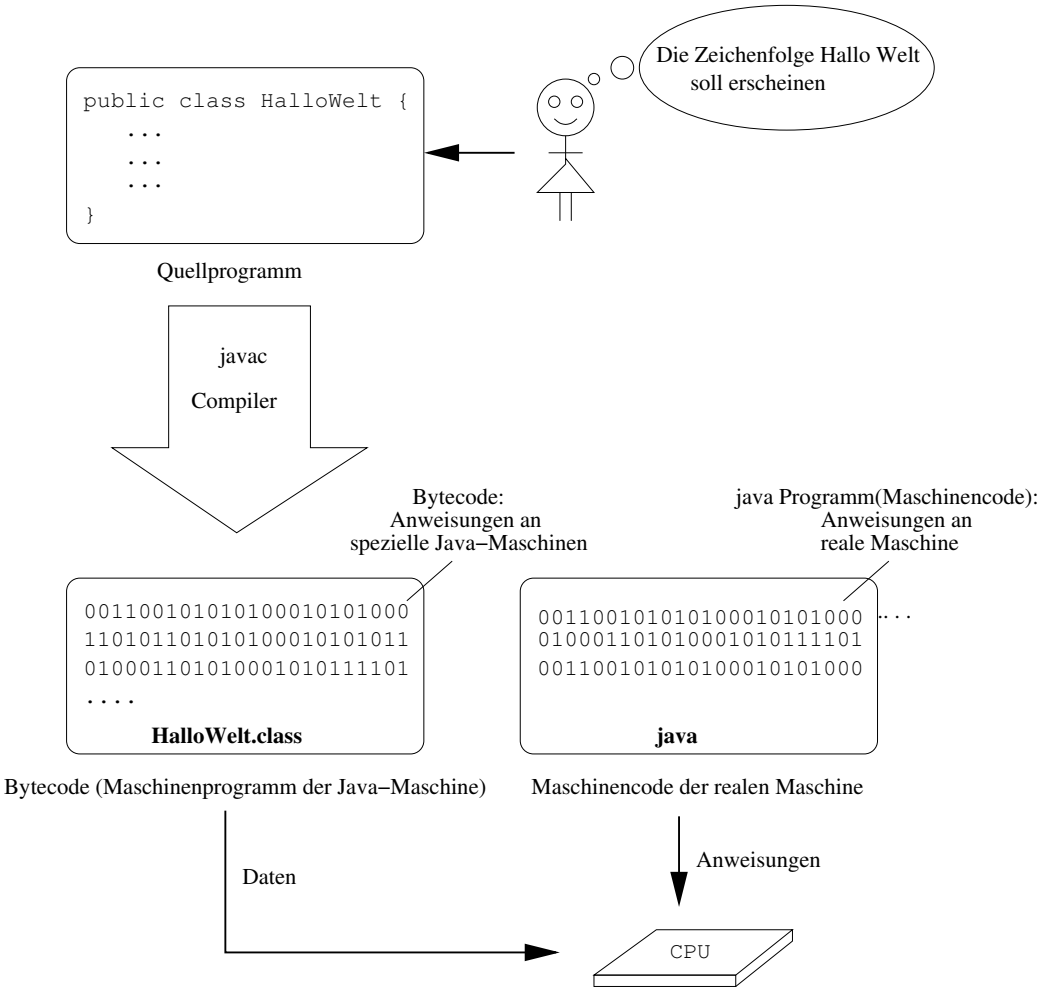


Abbildung 1.6: Compiler und virtuelle Maschine: javac und java

Klasse / Datei Eine Klasse, z.B. *HalloWelt*, wird als Text (Java–Quellcode) in eine Text–Datei mit dem Namen *HalloWelt.java* geschrieben.

Paket / Verzeichnis Der Quelltext der Klasse beginnt mit der Paketanweisung. Beispielsweise

```
package hallo;
```

Mit ihr wird definiert, zu welchem Paket die Klasse gehört. Ein Paket *ist* eine Kollektion von Klassen. Einem Paket *entspricht* ein Verzeichnis. Die Datei *HalloWelt.java* beispielsweise muss sich im Verzeichnis *hallo* befinden.

Im einfachsten Fall haben wir es mit einem Verzeichnis zu tun, das eine Datei enthält. Generell besteht der Quelltext aus vielen Dateien, die über einige Verzeichnisse verteilt sind.

Programme übersetzen und ausführen

Aus dem Quellcode erzeugt der Compiler den Bytecode. Der Compiler ist ein Programm mit dem Namen `javac`. Der Aufruf

```
javac hallo/HalloWelt.java
```

veranlasst den Compiler dazu im Verzeichnis `hallo` die Datei `HalloWelt.class` zu erzeugen. Mit dem Kommando

```
java hallo.HalloWelt
```

wird diese dann ausgeführt. Man beachte, dass `javac` sich auf Dateien und Verzeichnisse bezieht, `java` dagegen auf Pakete und Klassen operiert.[13] Der letzte Aufruf setzt voraus, dass das aktuelle Verzeichnis das Verzeichnis direkt über `hallo` ist. Nehmen wir an, dass das Verzeichnis `hallo` ein Unterverzeichnis von `SourceCode` ist. Das Ganze packen wir in ein Verzeichnis mit dem hochtraben Namen `HalloWeltProject`:

```
+ HalloWeltProject
    + src
        + hallo
            – HalloWelt.java
            – HalloWelt.class
```

Das Kommando `java hallo.HalloWelt` muss dann im Verzeichnis `src` abgegeben werden.

Wenn wir das Programm von einem beliebigen Verzeichnis starten wollen, dann geben wir den Klassenpfad mit an. Z.B.:

```
java -cp src hallo/HalloWelt
```

Man beachte hier: Der Klassenpfad zeigt auf das Verzeichnis *oberhalb* des Verzeichnisses, das einem Paket entspricht.

Quellprogramme und Bytecode trennen

Oft will man – so wie *Eclipse* es tut – die Bytecode–Dateien nicht in dem Verzeichnis ablegen, in dem sich die Quelldateien befinden. Wir folgen – so wie *Eclipse* – dieser Konvention und erzeugen ein Verzeichnis `bin` parallel zu `src`. Außerdem löschen wir `HalloWelt.class` im Verzeichnis `hallo`:

```
HalloWeltProject
    +src
        + hallo
          - HalloWelt.java
    + bin
```

Mit dem Kommando

```
javac -d bin src/hallo/HalloWelt.java
```

sorgen wir dafür, dass der Compiler die Dateien `HalloWelt.class` im Verzeichnis `bin/hallo` ablegt:

```
HalloWeltProject
    +src
       + hallo
         - HalloWelt.java
    + bin
        + hallo
          - HalloWelt.class
```

Bei der Ausführung müssen wir jetzt natürlich einen anderen Klassenpfad angeben:

```
java -cp bin hallo.HalloWelt
```

Jar-Dateien

Ein Java–Programm ist keine Einheit, keine einzelne Datei. Ein Java–Programm besteht aus Bytecode–Dateien die in einer bestimmten Verzeichnisstruktur stecken. In unserem trivialen Beispiel ist es eine Datei in einem Verzeichnis. Ein “richtiges” Programm umfasst aber schnell schon mal mehr als 500 Dateien in mehr als 50 Verzeichnissen.

[13] Diese unterschiedliche Anwendungslogik darf man durchaus kritisch sehen.

Das ist für die Auslieferung von Programmen an Kunden etwas unhandlich. Will der Kunde das ausgelieferte Java–Programm ausführen, dann muss er die richtige Verzeichnisstruktur auf seinem Rechner erstellen, mit den Dateien füllen und dann das richtige Kommando aufrufen.

Dieses Vorgehen ist kaum praktikabel und darum gibt eine kompaktere Form der Auslieferung eines Java–Programms: Die *Jar–Datei*. Eine Jar–Datei ist eine Verzeichnisstruktur in Form einer einzelnen Datei.[14] Wir erzeugen sie mit:

```
jar -cvf HalloWelt.jar hallo
```

Beispielsweise im Verzeichnis `bin`. Jedenfalls so dass `hallo` das Paketverzeichnis mit den Bytecode–Dateien ist. Die so erzeugte Datei hat den Namen `HalloWelt.jar` und enthält das Verzeichnis `hallo` mit der Datei `HalloWelt.class`. `HalloWelt.jar` kann jetzt an eine beliebige Stelle kopiert werden und dort mit

```
java -cp HalloWelt.jar hallo.HalloWelt
```

ausgeführt werden.

Der Klassenpfad (`-cp HalloWelt.jar`) verweist jezt auf die Jar–Datei und diese enthält alle für eine Programmausführungen notwendigen Verzeichnisse und Dateien. In gewisser Weise "ist" sie ein Java–Programm.

Mit dem Kommando:

```
jar -tf HalloWelt.jar
```

können wir feststellen, welchen Inhalt die Jar–Datei `HalloWelt.jar` hat. An der Ausgabe

```
~> jar -tvf HalloWelt.jar
META-INF/
META-INF/MANIFEST.MF
hallo/
hallo/HalloWelt.class
```

sehen wir, dass `HalloWelt.jar` neben den Java–Verzeichnissen und –Dateien noch ein Verzeichnis `Meta-Inf` und eine Datei `MANIFEST.MF` enthält. Diese sogenannte Manifest–Datei enthält Meta–Informationen, d.h. Informationen über den Inhalt der Jar–Datei.

Ausführbare Jar-Datei

Ein Kunde, der ein Java–Programm in Form einer Jar–Datei `halloWelt.jar` erhalten hat, tippt zur Ausführung

```
java -cp HalloWelt.jar hallo.HalloWelt
```

wobei `HalloWelt` im Paket `hallo` die Klasse ist, die die `main`–Methode enthält. Die Datei selbst und das Verzeichnis sind in `HalloWelt.jar` zu finden. – Besser, aber nicht gut. Der Kunde erspart sich das Erstellen von Verzeichnissen und Dateien, muss aber wissen, in welcher Klasse in welchem Paket die `main`–Methode zu finden ist. Dem kann abgeholfen werden – mit Hilfe der Manifest–Datei.

Damit es noch einfacher wird, fügen wir zur Jar–Datei noch die Information hinzu, dass die Klasse `hallo.HalloWelt` die Klasse ist, die `main`–Methode enthält, die ausgeführt werden soll. Diese Zusatzinformation

"hallo.HalloWelt" ist die Hauptklasse mit der main–Methode

wird in die Manifest–Datei geschrieben. Eine Manifest–Datei enthält generell beliebige Informationen über den Inhalt der Jar–Datei. Eine mögliche Information ist die über die Hauptklasse.

Die Manifest–Datei ist eine einfache Textdatei. Mit dem Kommando:

```
jar -cvfe HalloWelt.jar hallo.HalloWelt hallo
```

[14] Eine Jar–Datei ist eine Archiv–Datei. Archiv–Dateien haben oft die Endung `zip` und sind darum auch als *Zip–Dateien* bekannt. Eine Jar–Datei ist eine Zip–Datei mit speziellem Inhalt und der Endung `jar`.

sorgen wir dafür, dass `HalloWelt.jar` jetzt eine Manifest–Datei mit folgendem Inhalt umfasst:[15]

```
Manifest-Version: 1.0
Created-By: 1.7.0 (Oracle Corporation)
Main-Class: hallo.HalloWelt
```

`HalloWelt.jar` kann wieder an eine beliebige Stelle kopiert werden. Die Ausführung ist noch einfacher als oben:

```
java -jar HalloWelt.jar
```

Abhängig vom Betriebssystem kann `HalloWelt.jar` jetzt auch per Doppelklick gestartet werden. Für weitere Informationen über Jar–Dateien konsultiere man die Java–Dokumentation.[16]

Ausgabe in einem Ausgabefenster

Unser Hallo–Welt–Programm kann jetzt per Doppelklick auf die ausführbare Jar–Datei gestartet werden. Leider ist dann vom Wirken des Programms nichts zu bemerken. – Konsolenausgaben sieht man nicht, wenn das Programm ohne Konsole gestartet wird. Das ändern wir mit einer Ausgabe in ein Ausgabefenster:

```
package hallo;

import javax.swing.JOptionPane;

public final class HalloWelt {

 private HalloWelt(){}

 /**
  * Diese Klasse implementiert eine Anwendung die
  * "Hallo Welt" in einem Nachrichtenfenster ausgibt.
  */
 public static void main(String[] args) {
  JOptionPane.showMessageDialog(null, "Hallo Welt");
 }

}
```

Mit der Import–Anweisung wird nichts wirklich importiert. Es wird in unserem Fall nur möglich die Kurzform `JOptionPane` (ein *unqualifizierter Name*) statt der Langform `javax.swing.JOptionPane` (ein *qualifizierter Name*) zu verwenden.

Das Programm gibt seine Ausgabe in einem Fenster bekannt und ist somit nicht nur als Konsolenanwendung verwendbar.

Ausführbare Datei in Eclipse erzeugen

Eine ausführbare Version des Programms kann mit Doppelklick aktiviert werden und hat dann eine beobachtbare Wirkung: Ein Fenster mit “Hallo Welt”. Die ausführbare Version, also eine Jar–Datei mit

```
Main-Class: hallo.HalloWelt
```

im Manifest kann natürlich auch mit Eclipse leicht erzeugt werden:

- Rechtsklick auf das Paket im *Package Explorer* / *Export* / *Java* / *Runnable Jar*
- *Launch Configuration* auswählen. (Wenn es keine *Launch Configuration* gibt, dann starten Sie das Programm einmal aus Eclipse)
- Zielverzeichnis und Name der Jar–Datei angeben.
- Fertig: Datei kann mit Klick aktiviert werden.[17]

Uff – harter Stoff. Aber machbar. Lesen Sie den Abschnitt vielleicht noch einmal. Besser noch: Probieren und spielen Sie. Informatiker sind auch dazu da anderen den Computerfrust abzunehmen. Härten Sie sich darum rechtzeitig ab.

[15] Der genaue Inhalt ist von der Version des verwendeten jar–Programms abhängig. Das jar–Programm wiederum ist Bestandteil des installierten Java–Pakets.

[16] In `http://download.oracle.com/javase/7/docs/technotes/guides/jar/` wird das `jar`–Tool der Java–Version 7 beschrieben.

[17] Linux–User: Öffenen mit / Sonstige / java -jar

1.3 Lineare Programme

1.3.1 Variablen und Zuweisungen

Variablen in der Mathematik: Namen für Werte

Ein zentrales Konzept in fast allen Programmiersprachen sind die *Variablen*. Eine Variable ist hier als Behälter (Ablageplatz) für (wechselnde) Werte zu verstehen. Auch in der Mathematik spricht man von Variablen. Eine mathematische Variable ist aber etwas anderes als eine Variable in einem Programm! Es ist wichtig, sich den Unterschied zwischen mathematischen Variablen und Programmvariablen klar zu machen.

Eine mathematische Variable bezeichnet einen Wert, es ist der Name eines Wertes. Der Wert kann dabei durchaus beliebig sein. Beispielsweise ist eine Gleichung wie $x = 2 * x - 10$ eine Aussage, die wahr oder falsch sein kann, je nach dem, welcher Wert x zugeordnet ist. x ist eine Variable, die jeden (reellen) Wert annehmen kann. Für $x = 10$ ist diese Aussage wahr. Für alle anderen Werte von x ist sie falsch. Der Wert von x in der Gleichung ist zwar beliebig, aber er ist *fest*. x kann zwar jeden Wert annehmen, aber nicht links vom Gleichheitszeichen den einen und rechts vom Gleichheitszeichen einen anderen. Innerhalb einer Gleichung oder einer Berechnung ist der Wert einer mathematischen Variablen fest.[18]

Variable im Programm: Behälter für Werte

In einem Programm ist eine Variable nicht der Name eines Wertes, sondern der *Name eines Behälters*, in dem wechselnde Werte liegen können (Siehe Abbildung 1.7).

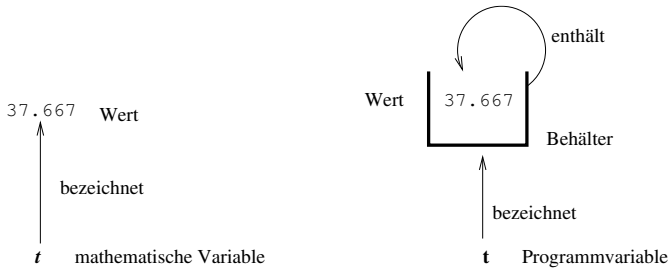


Abbildung 1.7: Mathematische Variablen und Programmvariablen

Variablendefinition

Variablen können an verschiedenen Stellen in einem Programm angelegt werden. Die wichtigste Stelle, an der Variablen existieren können, sind Methoden. In folgendem Programmstück[19]

```
public final class Temperatur {
  private Temperatur(){}
  public static void main(String[] args) {
    double t = 37.0;
    System.out.println(t);
  }
}
```

wird beispielsweise in der Methode `main` eine Variable `t` angelegt, mit dem Wert 37.0 belegt und ausgegeben. Das Wort `double` vor dem Namen `t` der Variablen gibt den *Datentyp*, oder kurz *Typ* der Variablen an. Der Typ `double` für `t` bedeutet, dass `t` einen gebrochenen Zahlwert aufnehmen kann. Andere wichtige Typen sind `int` (ganze Zahl) und `String` (Zeichenkette). Das kleine Programm

```
public final class Temperatur {
  private Temperatur(){}
```

[18] Mathematisch Gebildete mögen einwenden, dass eine Variable eine Lösungsmenge statt eines Wertes repräsentieren kann. Das ist aber eine weitergehende Interpretation der Gleichung als Menge von Aussagen über jeweils einen Wert und ändert nichts an der Tatsache, dass eine Gleichung nur sinnvoll ist, wenn in ihr jeweils jedem Vorkommen der gleichen Variablen der gleiche Wert zugeordnet wird.

[19] Der Einfachheit halber lassen wir im Folgenden oft die Paketangabe weg. Die Klassen können dann in beliebige Pakete platziert werden.

```
  public static void main(String[] args) {
      double t  = 37.0;    // 1. Variable
      int  i    = 3;       // 2. Variable
      String gruss = "Hallo"; // 3. Variable
      System.out.println(gruss
          + " ich habe gerade "
          + i + " Tee getrunken und bin dabei "
          + t + " Grad heiss geworden");
  }
}
```

verwendet drei Variablen von unterschiedlichem Typ und gibt die Zeile

```
Hallo ich habe gerade 3 Tee getrunken und bin dabei 37.0 Grad heiss geworden
```

aus. In diesem kleinen Beispiel sehen wir, wie Variablen definiert und in der Ausgabeanweisung verwendet werden können. Das Pluszeichen + bedeutet dort nicht "Addition", sondern "Zusammensetzen". Die Ausgabe wird dabei aus den Texten zwischen Hochkommas und dem Inhalt der Variablen zusammengesetzt und erzeugt so die Ausgabe.

Variablen, die in einer Methode definiert werden, gehören immer nur zu dieser Methode. Versuchen wir, sie in einer anderen Methode zu benutzen, dann wird sich das System beschweren.

```
...
 public static void main(String[] args) {
      String gruss = "Hallo";
      gruesse();
 }
 public static void gruesse() {
    System.out.println(gruss); <<-- FEHLER: gruss ist nicht definiert
 }
...
```

Zuweisung

Der Wert einer Variablen kann mit einer *Zuweisung* verändert werden.

```
double f = 37.0; // Variablendefinition
f = f*2;         // Zuweisung
```

Nach dieser Zuweisung hat `f` den Wert 74.0. Hier sehen wir den Unterschied zur Mathematik. Für einen Mathematiker ist

$$f = f * 2$$

eine Gleichung, d.h. eine Aussage, die für $f = 0$ *wahr* ist. In einem Programm ist es eine Zuweisung, also eine Handlungsanweisung:

> Nimm den aktuellen Wert von `f`, multipliziere ihn mit 2 und lege das Ergebnis in `f` ab.

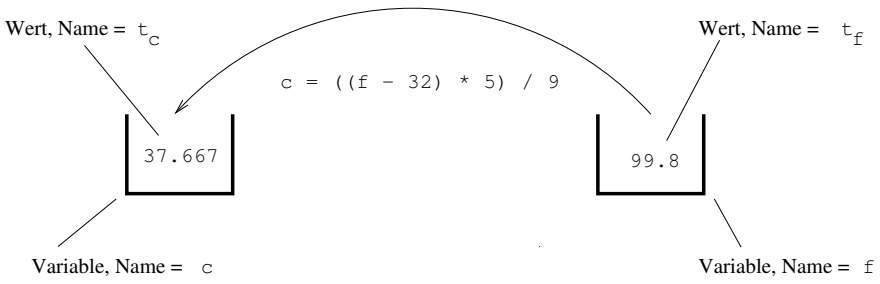


Abbildung 1.8: Die Zuweisung `c=((f-32)*5)/9`

Der *neue* Wert von `f` soll das Doppelte des *alten* Wertes sein. Der Wert von `f` ändert sich, während die Anweisung ausgeführt wird, von einem alten Wert – nennen wir ihn f_0 (beispielsweise 37.0) – zu einem neuen Wert f_1 (beispielsweise 74.0). Zwischen den beiden *Werten* besteht die mathematische Beziehung

$$f_1 = 2 * f_0.$$

Diese mathematische Gleichung beschreibt die Situation nach Ausführung der Zuweisung.

Die Zuweisung	führt zu
`f=2*f`	$f_1 = 2 * f_0$

Sehr oft unterscheidet man nicht explizit zwischen einer Variablen `f`, dem Wert der Variablen zu einem bestimmten Zeitpunkt (37.0) und dem Namen dieses Wertes (f_0). Es ist aber wichtig, sich immer genau bewusst zu sein, wovon die Rede ist, wenn der Name "f" fällt.

Form der Zuweisung

Die *Zuweisung* (engl. *assignment*) hat die allgemeine Form

$< Variable >=< Ausdruck >;$

Die Begriffe in spitzen Klammern – $< Variable >$ und $< Ausdruck >$ – sollen kenntlich machen, dass es sich um irgendeinen *Text* handelt, der vom Compiler als korrekte Notation für eine Variable bzw. einen Ausdruck akzeptiert wird.

Die Bedeutung der Zuweisung ist jetzt klar: Der Wert des Ausdrucks rechts vom Zuweisungszeichen soll berechnet und dann in der Variablen links gespeichert werden.

Der Ausdruck kann dabei beliebig kompliziert sein. Nehmen wir als Beispiel ein kleines Programm, mit dem die Temperatur von Fahrenheit in Celsius umgewandelt werden kann:

```
public final class Temperatur {

  private Temperatur(){}

  public static void main(String[] args) {
      double f = 100.0;
      double c = 0.0;
      c = ((f - 32) * 5) / 9;
      System.out.println("Der Temperatur " + f
              + " Grad Fahrenheit entspricht "
              + c + " Grad Celsius");
  }
}
```

Mit

```
c = ((f - 32) * 5) / 9;
```

wird $t_C = \frac{(t_F - 32) * 5}{9}$ in `c` gespeichert, wenn t_F der aktuelle Wert von `f` ist. Die gleiche Wirkung wird übrigens auch durch folgende Serie von Anweisungen erreicht:

```
c = f;
c = c - 32;
c = c * 5;
c = c / 9;
```

Wertverlaufstabelle

Hier wird `c` schrittweise auf den gewünschten Wert gebracht. Nehmen wir an `f` habe am Anfang den Wert 100, dann sind die einzelnen Schritte:

nach Anweisung	Wert von `f`	Wert von `c`
`c = f;`	100	100
`c = c - 32;`	100	68
`c = c * 5;`	100	340
`c = c / 9;`	100	37.77

Solche kleinen Tabellen, mit denen man selbst den Computer spielen und den Verlauf der Wertebelegung von Variablen verfolgen kann, sind sehr nützlich für das Verständnis von Programmen. Meist lässt man dabei allerdings die Anweisungen weg:

f	c
100	100
100	68
100	340
100	37.77

Die Tabelle beschreibt von oben nach unten den zeitlichen Verlauf der Belegung einer oder mehrerer Variablen.

Der Typ der Variablen und Werte

Der Typ einer Variablen ist so etwas wie ihre Art. Es gibt verschiedene Arten von Variablen: int–Variablen, double–Variablen, etc., je nachdem wie sie definiert wurden. Der Typ der Variablen hängt eng mit den Werten zusammen, die in ihnen gespeichert werden können. In einer int–Variablen werden beispielsweise nur ganzzahlige Werte gespeichert, in einer double–Variablen nur gebrochene Werte. Die Werte sind also auch von einer bestimmten Art. Werte haben wie die Variablen einen Typ. Beides ist verwandt aber nicht das Gleiche.

Werte und ihre Darstellung (Speicherung) im Rechner muss man unterscheiden. Im Rechner gibt es nur Bits, also 0–en und 1–en. Egal ob es sich um die Zeichenkette "Karl-Eugen Huber", die Zahl 37.77 oder ein Bild unseres Hundes handelt, alles sind nur Bits. Der Typ einer Variablen oder eines Wertes unterscheidet darum nicht verschiedene Arten von möglichem Inhalt von Speicherstellen, sondern:

- *wieviele* Bits zu einer Variablen (einem Wert) gehören, und
- *wie* die Bits in dieser Variablen (des Wertes) bei einer Ausgabe oder einer Rechenoperation zu *interpretieren* sind.

Eine `double`–Zahl beispielsweise wird mit vielen Bits in einem komplizierten Muster dargestellt, eine `int`–Zahl hat weniger Bits und ihr Bitmuster ist wesentlich leichter zu entschlüsseln. int–Werte sind darum auch keinesfalls eine Untermenge der double–Werte, wie man vielleicht nach seinen Erfahrungen aus dem normalen (mathematischen) Leben schließen könnte. Eine Operation kann nur dann korrekt ausgeführt werden, wenn bekannt ist, welche Art von Wert ein Bitmuster darstellen soll, also welchen Typ die Variable oder der Wert hat.

Eine Variablendefinition soll also:

- der Variablen einen Namen geben,
- den nötigen Platz für ihren Wert im Speicher schaffen und
- Auskunft darüber geben, wie der Inhalt dieses Speicherplatzes zu interpretieren ist.

Informationen über die Zahl der Bits oder Bytes, die zu einer Variablen gehören, werden beispielsweise benötigt, wenn ihr Wert kopiert werden soll. Die Interpretation der Bits wird gebraucht, wenn Operationen ausgeführt oder der Wert ausgegeben werden soll.

1.3.2 Kontakt mit der Außenwelt

Eingabe und Ausgabe

Ein Programm, das mit seiner Umwelt keine Daten austauscht, ist nutzlos. Jede Programmiersprache hat darum Konstrukte, die es ihren Programmen ermöglichen, Daten zu lesen und zu schreiben. Mit der Ein– und Ausgabe wird die geschlossene Welt des Programms verlassen. Das birgt allerdings auch eine gewisse Problematik. Ein Java–Programm soll im Prinzip auf jedem Rechner laufen können: auf einem PC genauso wie auf einem Großrechner, einem Handy oder einer Waschmaschine. Dabei sollten auch die unterschiedlichsten Methoden und Arten der Kommunikation eingesetzt werden können. Daten von einer oder auf einer Datei, Daten die eingetippt werden, Daten die über eine Netzwerkverbindung eintreffen, oder verschickt werden, oder ... Da Programmiersprachen erfahrungsgemäß eine sehr lange Lebensdauer haben, werden Java–Programme eventuell auf Rechnern laufen, von denen wir heute noch keine Vorstellung haben die mit ihren Benutzern eventuell per Gedankenübertragung kommunizieren.

Das Ein-/Ausgabekonzept von Java ist komplex und bietet sehr vielfältige Möglichkeiten. Zunächst reicht es uns aber, wenn wir einfach ein paar Daten eingeben können und unser Programm seine Ergebnisse anzeigen kann.

Ausgabe auf die Konsole

Die einfachste Art der Kommunikation haben wir bereits kennengelernt: Die Ausgabe auf die sogenannte *Konsole*. Die Konsole ist das simpelste Ein– oder Ausgabemedium, das das Betriebssystem einer Anwendung mitgibt. Im Normalfall sind das die Tastatur zur Eingabe und der Bildschirm (bzw. das Konsolen–Fenster von Eclipse). Die Konsole ist die *Standardeingabe*, die *Standardausgabe*, sowie die *Standardfehlerausgabe*, die von

`System.in` (Standardeingabe),
`System.out` (Standardausgabe) und
`System.err` (Standardfehlerausgabe)

repräsentiert werden. Auf die Standardausgabe schreiben wir normale Ausgaben, auf die Fehlerausgabe werden Fehlermeldungen geschrieben. Dazu gibt es die Methoden `print` (Ausgabe) und `println` (Ausgabe mit Zeilenvorschub). Beispielsweise erzeugt:

```
System.out.print("Hallo "); //Ausgabe ohne Zeilenvorschub
System.out.print("wer ");
System.out.println("da"); //Ausgabe mit Zeilenvorschub
```

die Ausgabe:

```
Hallo wer da
```

Das Gleiche wird natürlich mit:

```
System.out.println("Hallo wer da")
```

erreicht.

Die Eingabe von der Konsole ist fast so einfach wie die Ausgabe. Mit

```
import java.util.Scanner;

public final class Hallo {
    private Hallo(){}

public static void main(String[] args) {
  Scanner scan = new Scanner(System.in);
  int n = scan.nextInt();
  while ( n != 0 ) {
    System.out.println("Hallo");
    n = n-1;
  }
}
}
```

lesen wir von der Konsole eine ganze Zahl *n* ein, speichern sie in der Variablen `n` und geben dann *n*–mal `Hallo` aus. Hier sehen wir schon gleich wieder ein paar neue Konstrukte. Die *Import*–Anweisung am Anfang stellt uns eine Hilfsklasse mit dem Namen *Scanner* zur Verfügung. Wir benutzen sie, um Daten von der *Standardeingabe* `System.in` zu lesen. Das Lesen übernimmt die Methode `nextInt`. Mit:

```
int n = scan.nextInt();
```

werden dann von der Konsole Zeichen gelesen, als Ziffern einer Zahl interpretiert und diese dann in `n` gespeichert. Spaßeshalber können Sie ja mal etwas anderes eintippen als eine Zahl. Sie werden sehen, dass Ihr Programm mit dieser Eingabe nichts wird anfangen können und den Dienst verweigert.

Eingabe als Kommandozeilenargument

Eine Alternative zur Konsoleneingabe ist die Eingabe über die Kommandozeile. Mit Kommandozeile ist die Zeile gemeint, mit der das Java–Programm als Kommando aktiviert werden kann. Beispielsweise kann die `main`–Methode der Klasse `HalloWelt` mit der Kommandozeile

```
java HalloWelt
```

aktiviert werden. Über eine solche Zeile können auch Informationen in das Programm geschleust werden. Beispielsweise kann man dem Programm die Zeichenfolgen `Blubber` und `BlaBla` mit

```
java HalloWelt Blubber Blabla
```

übergeben.

In Eclipse werden Kommandozeilenargument über den `run..`–Dialog als *Arguments* eingegeben.

Geben wir jetzt auf eine der beiden Arten (Kommandozeile oder Eclipse run–Dialog) die Argumente ein, dann erzeugt das Programm

```
public final class HalloWelt {
  private HalloWelt(){}
  public static void main(String[] args) {
    System.out.println("Die Kommandozeilenargumente sind:");
    System.out.println(args[0]); //Ausgabe erstes Argument
    System.out.println("und:");
    System.out.println(args[1]); //Ausgabe zweites Argument
  }
}
```

die Ausgabe:

```
Die Kommandozeilenargumente sind:
Blubber
und:
BlaBla
```

Im Programm bezieht sich `args[0]` auf das erste Argument (`Blubber`) und `args[1]` auf das zweite (`BlaBla`).

Zahlen als Kommandozeilenargument

Die Kommandozeilenargumente werden immer als Zeichenketten (Strings) übergeben. Wollen wir eine Zahl in das Programm einschleusen, dann können wir nicht einfach schreiben:

```
  public static void main(String[] args) {
    int x = args[0]; // FEHLER: Type missmatch
  }
```

denn an die int–Variable `x` kann die Zeichenkette `args[0]` nicht zugewiesen werden. Wir müssen aus den Zeichen erst eine Zahl machen. Entsprechendes gilt für gebrochene Zahlen. Das Einlesen und Umwandeln geht so:

```
public final class HalloWelt {
  private HalloWelt(){}

  public static void main(String[] args) {
    int x = Integer.parseInt(args[0]); // Ganze Zahl von Kommandozeile
    double y = Double.parseDouble(args[1]); // Gebrochene Zahl von Kommandozeile
  }
```

Bei der Konsoleneingabe oben haben die Klasse `Scanner` und ihre Methode `nextInt` das "Umrechnen" von Zeichenfolgen in Zahlen übernommen. Wir merken uns, dass der Wechsel zwischen der Interpretation von Zeichenfolge als Zahlen – `123` als *hundertdreiundzwanzig* – oder als bloße Zeichenfolgen – `123` als Zeichen `1`, `2` und `3` – für ein Programm nicht so leicht und selbstverständlich ist wie für uns Menschen.[20]

Eingabe und Ausgabe mit JOptionPane, Zahlenformate

Die Eingabe über ein Fenster haben wir schon am Beispiel des Hallo–Welt–Programms gesehen. Selbstverständlich können damit auch Zahlwerte eingelsen werden. Eine "graphische" Variante des letzten Beispiels mit Ausgabe der eingegeben Werte ist:

```
package hallo;

import javax.swing.JOptionPane;
```

[20] Vielleicht haben Sie es vergessen, aber auch Sie mussten es einmal mehr oder weniger mühsam lernen.

```
public final class HalloWelt {
  private HalloWelt(){}

  public static void main(String[] args) {
     String xS = JOptionPane.showInputDialog("Bitte eine ganze Zahl");
     int x = Integer.parseInt(xS);

     String yS = JOptionPane.showInputDialog("Bitte eine gebrochene Zahl (mit Punkt!)");
     double y = Double.parseDouble(yS);

     JOptionPane.showMessageDialog(null, "Sie haben eingegen x = " + x + " y= " + y);
  }
}
```

Bei der Eingabe von gebrochenen Zahlen ist darauf zu achten, dass der angelsächsische Punkt statt des deutschen Kommas verwendet wird.

Natürlich ist es auch möglich gebrochene Zahlen statt mit Dezimalpunkt in der deutschen Form mit einem Dezimalkomma einzugeben. Dazu ist nur eine kleine Anpassung nötig:

```
package hallo;

import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

import javax.swing.JOptionPane;

public final class HalloWelt {

  private HalloWelt(){}

  public static void main(String[] args) throws ParseException {
    String xS = JOptionPane.showInputDialog("Bitte eine ganze Zahl");
    int x = Integer.parseInt(xS);

    String yS = JOptionPane.showInputDialog("Bitte eine gebrochene Zahl (mit Komma!)");

    // Zahl-String mit nach deutschen Konventionen analysieren:
    NumberFormat nf = NumberFormat.getInstance(Locale.GERMAN);
    double y = (Double) nf.parse(yS);

    JOptionPane.showMessageDialog(null, "Sie haben eingegen x = " + x + " y= " + y);
  }

}
```

Hier wird die Analyse des zweiten eingegeben Strings explizt “auf Deutsch umgeschaltet”.

Besser noch ist es die Eingabe so zu analysieren, dass sie sich an die lokalen Gegebenheiten automatisch anpasst. Auch das ist leicht möglich. Lässt man das Argument von `NumberFormat.getInstance` also `Locale.GERMAN` ganz weg, dann verwendet Java die lokalen Einstellungen des Computers und damit die Konventionen die der jeweiligen Benutzer gewohnt ist:

```
String yS = JOptionPane.showInputDialog("Bitte eine gebrochene Zahl eingeben");

// Zahl-String mit nach lokalen Konventionen analysieren:
NumberFormat nf = NumberFormat.getInstance();
double y = (Double) nf.parse(yS);
```

Wenn Zahlen im lokalen Format eingegeben werden können, dann wäre es schön, wenn sie auch entsprechend ausgegeben werden, also mit Dezimal–Komma statt –Punkt. Auch das ist, wie sich denken lässt, kein Problem. Mit

```
double y = 12.21;
NumberFormat nf = NumberFormat.getInstance();
String yAlsZahl = nf.format(y);
```

wird in `yAlsZahl` der String “12,21” gespeichert.

Unser Beispiel wird damit insgesamt zu:

```
package hallo;

import java.text.NumberFormat;
import java.text.ParseException;

import javax.swing.JOptionPane;

public final class HalloWelt {

 private HalloWelt(){}

 public static void main(String[] args) throws ParseException {
  String xS = JOptionPane.showInputDialog("Bitte eine ganze Zahl");
  int x = Integer.parseInt(xS);
  String yS = JOptionPane.showInputDialog("Bitte eine gebrochene Zahl eingeben");
  NumberFormat nf = NumberFormat.getInstance();
  double y = (Double) nf.parse(yS); // yS nach lokalen Konventionen parsen

  JOptionPane.showMessageDialog(null,
          "Sie haben eingegeben x = " + x + " y= "
          + nf.format(y)); // Wert von y im lokalen Zahlformat
 }

}
```

1.3.3 Programmentwicklung

Programmieren: Analyse, Entwurf, Codierung

Mit der Möglichkeit Daten ein– und auszugeben haben wir jetzt die Möglichkeit "richtige" Programme zu schreiben, die etwas Sinnvolles tun.

Ein Programm soll in der Regel ein Problem lösen oder eine Aufgabe erfüllen. Als erstes muss darum das Problem oder die Aufgabe *analysiert* werden. Als nächstes überlegt man sich, wie die Aufgabe gelöst werden soll: man *entwirft* ein allgemeines Vorgehen zur Lösung und ein Programmkonzept. Schließlich muss noch der Quellcode geschrieben werden: das Programm wird *codiert*, oder wie man auch sagt *implementiert*.

Betrachten wir als Beispiel noch einmal die Umwandlung von Temperaturwerten von Grad Fahrenheit in Grad Celsius.

Analyse

Am Anfang der Analyse wird die *Problemstellung präzise definiert*:

> Das Programm soll eine ganze oder gebrochene negative oder positive Zahl einlesen, sie als Grad–Angabe in Fahrenheit interpretieren und zwei Werte ausgeben: den eingegebenen Wert und den in Celsius umgerechneten Wert.

Als nächstes macht man sich *mit dem Problemfeld vertraut*, indem man sich die zur Lösung notwendigen Informationen besorgt. In unserem Fall sagt uns ein elementares Physikbuch wie Fahrenheit in Celsius umgewandelt wird:[21]

$$t_C = \frac{(t_F - 32) * 5}{9}$$

Spezifikation

Die Analyse wird mit einer *Spezifikation* abgeschlossen. In ihr wird exakt festgelegt was das Programm zu leisten hat. Bei unserer Temperaturumrechnung besteht die Spezifikation aus der Umrechnungsformel oben und einer Festlegung, wie genau die Ein– und Ausgabe zu erfolgen hat. Also etwa so:

> *Das Programm soll von der Konsole eine Temperaturangabe in Fahrenheit als gebrochene Zahl mit Dezimalpunkt einlesen und sie nach der Formel* $t_C = \frac{(t_F - 32) * 5}{9}$ *in eine äquivalente Temperaturangabe in Celsius umwandeln und auf der Konsole im gleichen Format ausgeben.*

[21] Ok, Ok, Sie haben gar kein Physikbuch oder es es ist zu weit von der Tastatur weg: *Ergooglen* Sie sich halt die notwendigen Informationen.

Entwurf

Nachdem in der Analyse festgestellt wurde, *was* zu tun ist und die zur Lösung notwendigen Informationen besorgt wurden, kann man sich jetzt dem *wie* zuwenden: dem Entwurf. Im *Entwurf* wird festgelegt, *wie* das Programm seine Aufgabe konkret lösen soll. Bei einer Berechnungsaufgabe, wie in unserem Beispiel, wird hier der *Algorithmus* festgelegt:

Algorithmus zur Umwandlung von Fahrenheit in Grad:

1. Grad–Fahrenheit einlesen und in der Variablen `f` speichern
2. `c` mit dem Wert $\frac{(Wert(f)-32)*5}{9}$ belegen
3. Wert von `f` und `c` ausgeben

Implementierung

Bei der Implementierung wird der Algorithmus aus dem Entwurf in der korrekten Notation der Programmiersprache niedergeschrieben:

```
/**
 * @author Hugo Hacker
 *
 * Umrechnung einer Temperaturangabe von Grad
 * Fahrenheit in Grad Celsius. Eingabe ueber
 * das erste Kommandozeilenargument
 */
public final class Temperatur {

  private Temperatur(){}

  public static void main(String[] args) {

    double f = 0.0; // Variable, enthaelt Grad in Fahrenheit
    double c = 0.0; // Variable, enthaelt Grad in Celsius

    //Grad in Fahrenheit von der Kommandozeile einlesen
    f = Double.parseDouble(args[0]);

    //Grad in Celsius berechnen:
    c = ((f - 32) * 5) / 9;

    //Grad in F. und C. ausgeben:
    System.out.println( f + " Grad Fahrenheit ist "
                    + c + " Grad Celsius ");
  }
}
```

Syntaxfehler

In der Regel wird man es nicht schaffen, ein Programm gleich beim ersten Versuch korrekt einzutippen. Bei Programmen, in denen nicht penibel alle Regeln der jeweiligen Programmiersprache eingehalten werden, verweigert der Compiler die Übersetzung. Vergisst man auch nur ein Komma, oder schreibt ein Komma an eine Stelle, an der ein Semikolon stehen muss, wird das Programm nicht übersetzt. Stattdessen erscheint eine Fehlermeldung.

Nehmen wir an, dass eine Ausgabeanweisung nicht – wie es richtig wäre – mit einem Semikolon, sondern mit einem Komma beendet wird:

```
System.out.println( " Grad in Celsius ist " + c), // Syntaxfehler
```

Prompt kommt die Quittung in Form einer Fehlermeldung:

```
Syntaxerror: ",", ; expected
```

Der Text mit dem Komma entspricht nicht den syntaktischen Regeln der Sprache Java. Er enthält einen *Syntaxfehler*. Am Anfang sehen solche Fehlermeldungen oft kryptisch und unverständlich aus. Es ist aber wichtig, sich damit zu beschäftigen, sie lesen und verstehen zu lernen.

Semantische Fehler – Inhaltliche Fehler in formal korrekten Programmen

Manchmal wird der von uns eingegebene Quellcode vom Compiler ohne Fehlermeldung akzeptiert, aber das erzeugte Programm verhält sich nicht so wie erwartet. Beispielsweise wollen wir, dass Grad Celsius in Fahrenheit umgerechnet werden, aber die Ausgabe entspricht nicht unseren Erwartungen. Man sagt dann, das Programm enthält einen *semantischen* (inhaltlichen) Fehler. Es tut etwas, aber nicht das, was wir von ihm erwarten. In diesem Fall ist die Ursache schnell festgestellt. Wir haben uns sicher einfach nur vertippt und das Programm sieht folgendermaßen aus:

```
public final class Temperatur {

  private Temperatur(){}

  public static void main(String[] args) {
    double f = 0.0; // Variable, enthaelt Grad in Fahrenheit
    double c = 0.0; // Variable, enthaelt Grad in Celsius

    //Grad in Fahrenheit von der Kommandozeile
    f = Double.parseDouble(args[0]);

    //Grad in Celsius berechnen:
    c = f - 32 * 5 / 90; // Klammern vergessen, 90 statt 9 !

    //Grad in Celsius ausgeben:
    System.out.println( " Grad in Celsius ist " + c);
  }
}
```

Im Gegensatz zu vorhin ist hier das Programm auch mit den Fehlern syntaktisch korrekt und kann übersetzt werden. Der Compiler kann ja nicht wissen, was wir eigentlich berechnen und ausgeben wollten. Er hält sich an das was im Quellprogramm steht. Leider sind die semantischen Fehler in der Regel nicht so leicht zu identifizieren wie in diesem Beispiel. Ein semantischer Fehler wird in der Regel nicht durch Tippfehler, sondern durch “Denkfehler” verursacht. Man glaubt, das Programm sei korrekt und mache das, was es tun soll. In Wirklichkeit macht es zwar etwas, aber nicht das, was wir wollen, sondern das was wir aufgeschrieben haben und von dem wir nur dachten, es sei das, was wir wollten.

Semantische Fehler sind die ernsten, die richtigen Fehler. Man lässt darum meist die Kennzeichnung “semantisch” weg und spricht einfach von “Fehler”. Nur wenn ausdrücklich betont werden soll, dass es sich um etwas so triviales wie einen syntaktischen Fehler handelt, spricht man von einem “Syntax–Fehler”.

1.4 Verzweigungen und Boolesche Ausdrücke

1.4.1 Bedingte Anweisungen

Die Bedingte Anweisung (If–Anweisung)

Der wesentliche Bestandteil der Programme, die wir bisher kennengelernt haben, ist die *main–Funktion*, auch *main-Methode* genannt. Genau gesagt handelt es sich um eine *statische Methode*. Man erkennt das an dem *Schlüsselwort* (engl. *Keyword*) **static**. Schlüsselworte sind Worte deren Form, Bedeutung und mögliche Postion in einem Programm von der Programmiersprache genau festgelegt ist. Andere Beispiele für Schlüsselworte sind *class*, *import* und *package*.

Statische Methoden nennen wir in der Regel einfach Funktionen, denn sie stellen in etwa das dar, was wir als Funktionen aus der Mathematik und dem Alltagsleben kennen. Eine *statische Methode* (Funktion) besteht aus Variablendefinitionen und Anweisungen. Die Anweisungen werden in der Reihenfolge, in der wir sie aufschreiben, ausgeführt. Anweisungen wie die Zuweisung und die Ein– und Ausgabeanweisung werden dabei in jedem Fall ausgeführt. Die bedingten Anweisungen, mit denen wir uns in diesem Abschnitt beschäftigen wollen, werden dagegen nur unter bestimmten Umständen ausgeführt.

Mit einer *bedingten Anweisung* (If–Anweisung) können also die Aktionen eines Programms davon abhängig gemacht werden, ob eine Bedingung erfüllt ist. Damit lassen sich Programme konstruieren, die flexibel auf ihre Eingabewerte eingehen. Ein Beispiel für eine bedingte Anweisung ist:

```
if ( x > y ) max = x;
```

Mit dieser Anweisung wird `max` nur dann mit dem Wert von `x` belegt, wenn dieser größer ist als der von `y`.

Die If–Anweisung gibt es in zwei Varianten, ein– und zweiarmig.

Die *einarmige If–Anweisung* hat die Form:

`if` (< *Bedingung* >) < *Anweisung* >

Die *zweiarmige If–Anweisung* sieht folgendermaßen aus:

`if` (< *Bedingung* >) < *Anweisung* >`else`< *Anweisung* >

Die einarmige If–Anweisung

Die einarmige If-Anweisung ist ein “Wenn–Dann”. Wenn die Bedingung erfüllt ist, dann wird die Anweisung ausgeführt. Wir betrachten dazu ein kleines Beispiel:

```
import java.util.Scanner;

public final class Maximum {

     private Maximum(){}

     public static void main(String[] args) {
           // Zahlen von der Konsole einlesen
           Scanner scan = new Scanner(System.in);
           int a = scan.nextInt();
           int b = scan.nextInt();

           // Maximum bestimmen:
           int max = 0; // Maximum von a und b

           if (a > b) { max = a; }
           if (b > a) { max = b; }
           if (a == b) { max = b; } // Achtung: == fuer Vergleich

           // Maximum ausgeben:
           System.out.println("Das Maximum ist: " + max);
     }

}
```

Das Programm erwartet zwei ganze Zahlen als Eingabe über die Konsole, bestimmt deren Maximum[22] und gibt es dann aus.

Mit den drei Bedingungen `a > b`, `b > a` und `a == b` werden alle möglichen Größenbeziehungen zwischen `a` und `b` abgedeckt. Genau eine trifft zu und genau eine Zuweisung an `max` wird ausgeführt.

Achtung: der *Vergleich* von zwei Werten wird mit == ausgedrückt! Ein Gleichheitszeichen allein ist in Java immer eine Zuweisung!

Die zweiarmige If–Anweisung

Die zweiarmige If-Anweisung ist ein "Wenn–Dann–Andernfalls". Wenn die Bedingung erfüllt ist, dann wird die erste Anweisung ausgeführt, andernfalls die zweite. Die Bestimmung des Maximums sieht mit ihr folgendermaßen aus:

```
if ( a > b )
  max = a;
else
  max = b;
```

Auch bei dieser Anweisung wird in jedem der drei möglichen Fälle genau eine der beiden Zuweisungen ausgeführt. Die Form oben ist korrekt, aber man schreibt besser und klarer mit geschweiften Klammern:

```
if ( a > b ) {
  max = a;
} else {
  max = b;
}
```

Die geschweiften Klammern fassen Anweisungen zusammen. Hier wird jeweils eine einzige Anweisung "zusammengefasst". Das ist eigentlich unnötig, aber wenn später Anweisungen eingefügt werden, dann werden die dann notwendigen Klammern nicht vergessen.

Formatierung von Java–Programmen

Es kommt dem Java–Compiler nicht auf die Formatierung und die Einrückung an. Er interessiert sich ausschließlich für die aufgeschriebenen Zeichen und nicht für die Art, wie sie in einer Datei verteilt sind. So sind beispielsweise die Anweisungen

```
// OK (besser mit Klammern):
if (a>b) max=a;
else max=b;

// Pfui:
if(a>b)
max = a; else
max = b;

// OK (besser mit Kl.):
if ( a > b )
     max = a;
else
     max = b;

// OK (besser mit Klammern):
if ( a > b ) max = a; else max = b;

// GUT:
if ( a > b ) {
   max = a;
} else {
   max = b;
}
```

```
// OK:
if ( a > b ) { max = a; } else { max = b; }

// Pfui:
if(a>b) max = a; else
max = b;

// Pfui:
if(a>b)
max=a;
else
max = b;

// OK (naja, altmodischer C-Stil):
if ( a > b )
{
   max = a;
}
else
```

[22] Das Maximum von zwei Zahlen a und b ist die kleinste Zahl c für die gilt $a \leq c$ und $b \leq c$.

```
{
   max = b;
}
```

vollkommen äquivalent. Man sollte sich allerdings an die üblichen Konventionen halten und seinen Programmen ein ansprechendes Aussehen geben. Die Formatierung eines Quellprogramms ist "ansprechend", wenn seine innere logische Struktur an der Textform erkennbar wird. Ein gewisser eigener Stil wird zwar allgemein toleriert, mit einer zu eigenwilligen Form riskiert man jedoch seinen Ruf und eventuell die kollegiale Zusammenarbeit.[23]

Statische und dynamische Auswahl von Anweisungen

Verzweigungen definieren Alternativen in der Abarbeitung, über die *dynamisch* entschieden wird. Der Prozessor (bzw. die virtuelle Maschine), der das (aus dem Quell–Programm erzeugte Maschinen–) Programm abarbeitet, entscheidet darüber, welche Anweisungen ausgeführt werden und welche nicht. Damit wird über den Verlauf des Programms nicht am Tisch oder Bildschirm des Programmierers, sondern während der Laufzeit des Programms entschieden. Dinge, die *während des Programmlaufs* entschieden werden, nennt man in der Welt der Programmiersprachen *dynamisch*. Das Gegenteil von "dynamisch" ist "statisch". *Statisch* ist alles, was vor der Laufzeit – im *Quelltext des Programms* – festgelegt wird.

Anweisungen erzeugen Zustandsänderungen

An einigen Beispielen zeigen wir, wie der *Zustand* des Programms – d.h. die aktuelle Belegung seiner Variablen – durch verschiedene Anweisungen verändert wird:

- – Zustand vorher: $a = 3, b = 4$
  – Anweisung: `if (a==b) a = 5; b = 6;`
  – Zustand nachher: $a = 3, b = 6$

- – Zustand vorher: $a = 3, b = 4$
  – Anweisung: `if (a==b) a = 5; else b = 6;`
  – Zustand nachher: $a = 3, b = 6$

- – Zustand vorher: $a = 2, b = 2$
  – Anweisung: `if (a==b) a = 5; b = 6;`
  – Zustand nachher: $a = 5, b = 6$

- – Zustand vorher: $a = 2, b = 2$
  – Anweisung: `if (a==b) a = 5; else b = 6;`
  – Zustand nachher: $a = 5; b = 2$

- – Zustand vorher: $a = 2, b = 2$
  – Anweisung: `if (a==b) a = 5; if (a==b) a = 5; else b = 6;`
  – Zustand nachher: $a = 5; b = 6$

Zusicherung

Eine *Zusicherung* (engl. *Assertion*) ist eine Aussage über den Programmzustand (d.h. die Belegung der Variablen) wenn der Kontrollfluss eine bestimmte Stelle erreicht hat. Beispiel:

```
if (x > y) max = x;
else       max = y;
// max >= x, max >= y
```

[23] Hinweis: Quelltext–Editoren bieten in der Regel eine Unterstützung bei der Formatierung von Java Programmen an. In Eclipse kann der Editor die Einrückung korrigieren (Source>Correct Indentation) oder ein Stück Quelltext komplett formatieren (Source>Format).

Hiermit wollen wir sagen dass, wenn immer und mit welchen Werten von `a` und `b` auch immer, der Kontrollfluss den Kommentar

```
// max >= x, max >= y
```

erreicht, dann ist der Wert von `max` größer–gleich dem von `x` und dem von `y`. Derartige Zusicherungen sind gut geeignet, die Wirkung eines Programmabschnitts klar zu machen. Sie helfen auch bei der Programmentwicklung, denn der erste, der die Wirkung der Anweisungen des Programms verstehen muss, ist dessen Autor(in).

Wenn Zusicherungen auch etwas sind, das in erster Linie den Geist der Programmierer beschäftigen soll, so ist es doch gelegentlich nützlich, eine Zusicherung im laufenden Programm zu überprüfen. In Java gibt es dazu die `assert`–Anweisung. In unser Beispiel kann sie eingesetzt werden, um zu prüfen, ob wir tatsächlich das Maximum berechnet haben:

```
import java.util.Scanner;

public final class Maximum {

    private Maximum(){}

    public static void main(String[] args) {
          // Zahlen von der Konsole einlesen
          Scanner scan = new Scanner(System.in);
          int a = scan.nextInt();
          int b = scan.nextInt();

          // Maximum bestimmen:
          int max = 0; // Maximum von a und b

          if (a > b) max = a;
          if (b > a) max = b;
          if (a == b) max = b; // Achtung: == fuer Vergleich

          assert max >= a ; // Achtung: Ausfuehrung nicht ohne weiteres
          assert max >= b ;

          // Maximum ausgeben:
          System.out.println("Das Maximum ist: " + max);
    }

}
```

Zusicherungen helfen bei der Entwicklung und dem Test eines Programms. Sie werden nach expliziter Aufforderung tatsächlich geprüft. Dazu muss die Option `-ea` der virtuellen Maschine gesetzt werden. Auf der Konsole müssen Sie das Programm ausführen mit:

```
java -ea <dateiname>
```

In Eclipse muss `-ea` bei *VM Arguments* eintragen werden.[24] So kann die Ausführung von Zusicherungen leicht ein– und ausgeschaltet werden. Sie helfen bei der Entwicklung und verzögern ausgetestete Programme nicht im produktiven Betrieb.

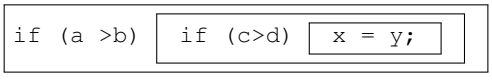
1.4.2 Geschachtelte und zusammengesetzte Anweisungen

Geschachtelte If–Anweisungen

Eine If–Anweisung besteht aus einer Bedingung und einer oder zwei (Unter–) Anweisungen. In den bisherigen Beispielen waren die Anweisungen innerhalb einer If–Anweisung stets Zuweisungen. Das muss jedoch nicht immer so sein. Eine If–Anweisung kann aus beliebigen anderen Anweisungen aufgebaut sein – auch aus anderen If–Anweisungen. Ein Beispiel (kein Druckfehler !) ist:

```
if (a>b) if (c>d) x = y;
```

Hier sieht man drei ineinander geschachtelte Anweisungen: eine Zuweisung in einer If–Anweisung in einer If–Anweisung:



Üblicherweise macht man die Struktur solcher etwas komplexeren Anweisungen durch das Layout des Quelltextes klar:

[24] Run / Run Configurations / *zutreffende auswählen* / Arguments / VM Arguments : -ea eintragen

```
if (a>b)
   if (c>d)
      x = y;
```

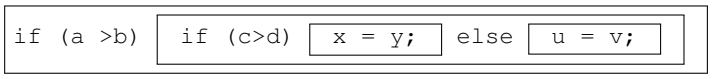
Mit jeder Stufe der Einrückung geht es dabei tiefer in die logische Struktur.

Geklammerte Anweisungen

Bei manchen etwas komplexeren Anweisungen ist die logische Struktur nicht sofort aus dem Text ersichtlich. Das gilt insbesondere, wenn in einer Anweisung mehr `if` als `else` vorkommen. Ein Beispiel ist:

```
if (a>b) if (c>d) x = y; else u = v;
```

Die Regeln von Java legen fest, dass ein `else` immer zum *nächsten vorherigen* `if` gehört. Damit ist die Anweisung oben als

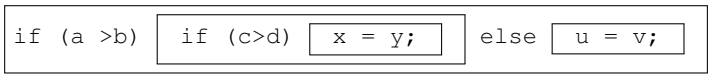


zu interpretieren. Das `else` gehört also zum *zweiten* `if`.[25]

Es ist wichtig zu beachten, dass die Einrückung dem Verständnis des menschlichen Lesers zwar hilft, dass die Einrückung die Interpretation durch den Compiler aber nicht beeinflusst. Selbst wenn wir

```
if (a>b)    // Achtung: FALSCHES und verwirrendes Layout
  if (c>d)
    x = y;
else        // diese Einrueckung entspricht nicht der wahren
  u = v;    // Zuordnung des else
```

schreiben, ändert das nichts daran, dass das `else` zum zweiten `if` gehört. Layout ist für Menschen, es wird vom Compiler ignoriert. Das Layout sollte immer der Interpretation des Textes durch den Compiler entsprechen. Soll das `else` im letzten Beispiel sich tatsächlich auf das erste `if` beziehen:



dann muss die innere If–Anweisung (in dem Fall ohne `else`–Teil) geklammert werden:

```
if (a>b) { if (c>d) x = y; } else u = v;
```

Ein passendes Layout macht die Struktur dann klar:

```
if (a>b) { // OK: Layout passt zur logischen Struktur
  if (c>d)
    x = y;
} else
    u = v;
```

Das Layout passt jetzt zur logischen Struktur des Programmtexts.[26]

Generell sollte man in bedingten Ausdrücken *immer Klammern* verwenden, selbst dann wenn sie formal nicht notwendig sind. Also auch statt

```
if (a>b)
    x = y;
else
    u = v;
```

besser

[25] Dies ist eine eher willkürliche Entscheidung. Man könnte auch das erste If als zweiarmig und das zweite als einarmig ansehen, aber der Compiler macht genau das nicht und dessen Meinung ist nun mal entscheidend.

[26] Es empfiehlt sich bei komplexeren Konstrukten den Quelltext durch die Entwicklungsumgebung formatieren zu lassen und dann zu prüfen, ob das Ergebnis dieser Formatierung der (gewollten und gemeinten) logischen Struktur entspricht.

```
if (a>b) {
     x = y;
} else {
     u = v;
}
```

schreiben.

Zusammengesetzte Anweisungen

Mit geschweiften Klammern kann man auch mehrere Anweisungen zu (formal) einer zusammenfassen. Das ist beispielsweise dann nützlich, wenn mehrere Anweisungen ausgeführt werden sollen, wenn eine Bedingung erfüllt ist. Mit

```
if (a>b) {
   t = a;
   a = b;
   b = t;
}
```

wird beispielsweise der Inhalt von `a` und `b` getauscht, wenn `a` größer als `b` ist. Die Klammern müssen sein. Schreibt man etwa – ein häufiger Fehler –

```
if (a>b)    // Achtung: FALSCHES und verwirrendes Layout
   t = a;
   a = b;
   b = t;
```

dann wird dies vom Compiler genau wie

```
if (a>b) t = a;
a = b;
b = t;
```

interpretiert. Formal folgt der Bedingung genau eine Anweisung. Mit geschweiften Klammern kann aber eine beliebige Folge von Anweisungen zu einer *zusammengesetzten Anweisung* werden:

{ < *Anweisung* > . . . < *Anweisung* > }

Beispiele: Maximum von drei Zahlen

Als Beispiel zum Umgang mit If–Anweisungen zeigen wir hier zwei Arten, das Maximum von drei Zahlen zu berechnen. Die Zahlen seien jeweils in den Variablen `a`, `b` und `c` zu finden.

Die einfachste Strategie besteht darin `a`, `b` und `c` der Reihe nach zu inspizieren und in einer Variablen `max` das *bis jetzt gefundene Maximum* aufzuheben.

```
...
  max = a;          // max = Maximum (a)
  if (b > max) max = b; // max = Maximum (a, b)
  if (c > max) max = c; // max = Maximum (a, b, c)

  // max >=a, max >= b, max >= c
...
```

Eine ausführliche Analyse aller möglichen Größenverhältnisse zwischen den Variablen ist komplexer und kann, im Gegensatz zur Strategie oben, nicht einfach auf mehr Vergleiche erweitert werden, wie folgenden umständlichen Code erkennt:

```
...
  if (a>b) {
    // a > b
    if (a>c) {
      // a > b && a > c
      max = a;
    } else {
      // a > b && ! (a > c)
```

```
      max = c;
    }
  } else {
    // a <= b
    if (a>c) {
      // a <= b && a > c
      max = b;
    } else {
      // a <= b && a<=c
      if (b>c) {
        // a <= b && a<=c && b>c
        max = b;
      } else {
        // a <= b && a<=c && b<=c
        max = c;
      }
    }
  }
...
```

Beispiele

Wir zeigen noch einige Beispiele für die Auswertung bedingter und zusammengesetzter Anweisungen (`int x, y;` Bitte nachvollziehen!):

- – Zustand vorher: $x = 3, y = 4$
  – Anweisung: `if (x==y) x = 5; y = 6;`
  – Zustand nachher: $x = 3, y = 6$

- – Zustand vorher: $x = 3, y = 4$
  – Anweisung: `if (x==y) {x = 5; y = 6;}`
  – Zustand nachher: $x = 3, y = 4$

- – Zustand vorher: $x = 3, y = 4$
  – Anweisung: `if (x==y) x = y; if (x==y) y = 6;`
  – Zustand nachher: $x = 3, y = 4$

- – Zustand vorher: $x = 3, y = 4$
  – Anweisung: `if (x==y) y = x; {y = x; if (x==y) y = 6;}`
  – Zustand nachher: $x = 3, y = 6$

1.4.3 Die Switch–Anweisung

Beispiel

Neben den beiden Varianten der If–Anweisung gibt es eine dritte Form der Verzweigung: die Switch–Anweisung. Beginnen wir mit einem Beispiel. Eine einfache Anwendung der Switch–Anweisung ist:

```
public class SwitchTest {

  public static void main(String[] args) {
    String opString = args[0];
    char opZeichen = opString.charAt(0); // erstes Zeichen holen
    int a        = Integer.parseInt(args[1]);
    int b        = Integer.parseInt(args[2]);

    switch ( opZeichen ) { // Verzweige je nach dem was opZeichen ist
      case '+':            // Falls es ein + ist ...
```

```
        System.out.println(a+b);
        break;
      case '*':            // Falls es ein * ist ...
        System.out.println(a*b);
        break;
      default :
        System.out.println("Unbekannter Operator");
    }
  }
}
```

Der Übersicht halber haben wir die in einer ernsthaften Anwendung notwendigen Prüfungen der Eingabe weggelassen.

Das Programm liest drei Argumente von Kommandozeile ein, beispielsweise `+ 12 13`. Das zweite und dritte Argument werden jeweils in Int–Wert konvertiert. Das erste wird zunächst als String gespeichert, dann holen wir uns mit `opString.charAt(0)` das erste Zeichen heraus, in unserem Beispiel ist + das erste (und einzige) des ersten Arguments. In der Switch–Anweisung wird je nach dem Wert in `opZeichen` in einen der drei angegebenen Fälle verzweigt. Das Schlüsselwort `default` bedeutet so viel wie "Wenn kein anderer Fall zutrifft".

De facto ist diese Switch–Anweisung nichts anderes, als eine längliche If–Anweisung:

```
if (opZeichen == '+')
    System.out.println(a+b);
else if (opZeichen == '*')
    System.out.println(a*b);
else System.out.println("Unbekannter Operator");
```

Die Switch–Anweisung ist eine Verallgemeinerung der bedingten Anweisung

Die Switch–Anweisung kann also als bedingte Anweisung verwendet werden. Sie kann aber mehr, sie ist eine verallgemeinerte Form des `if-then-else`. Die Verallgemeinerung besteht darin, dass beliebig viele Alternativen erlaubt sind. Dazu noch ein Beispiel:

```
char note;
string ton;
...
switch (note) {
  case 'C' : ton = "do"; break;
  case 'D' : ton = "re"; break;
  case 'E' : ton = "mi"; break;
  case 'F' : ton = "fa"; break;
  case 'G' : ton = "so"; break;
  case 'A' : ton = "la"; break;
  case 'H' : ton = "si"; break;
  default : ton = "??"; break;
}
```

Der Ausdruck, dessen Wert den weiteren Verlauf des Programms bestimmt, ist hier mit `note` eine `char`–Variable, also ein Ausdruck mit einem `char`–Wert. Die Entscheidung wird hier über acht Alternativen gefällt. Selbstverständlich lässt sich das auch mit If–s ausdrücken, nur eben viel umständlicher. Zudem ist die Ausführung einer Switch-Anweisung auch schneller: Mit nur einem Vergleich kann sofort an die richtige Stelle gesprungen werden.

Die `default`–Alternative

`default` ist so etwas wie das `else` der Switch–Anweisung. Die `default`–Alternative wird immer gewählt, wenn *vorher* keine andere `case`–Marke dem Wert des Ausdrucks entsprochen hat. Die `default`–Alternative muss nicht die letzte sein, die ihr folgenden werden auch beachtet. Üblicherweise schreibt man sie aber als letzte.

`break` in Switch–Anweisungen

Die `break`–Anweisung war in den bisherigen Beispielen stets die letzte Anweisung in einer Alternative. Das wird von den syntaktischen Regeln von Java *nicht unbedingt* gefordert. Ohne `break` wird die Switch–Anweisung mit dem Ende einer Alter-

native nicht beendet, sondern mit der textuell nächsten Alternative fortgesetzt. Sie "fällt" von einer Alternative in die nächste.[27] So erzeugt beispielsweise

```
int c = 2;
switch (c) {
  case 1 : System.out.println("c = 1");
  case 2 : System.out.println("c = 2"); // Einsprung hier
  case 3 : System.out.println("c = 3"); // kein break: weiter, durch alles durch
  default: System.out.println("c != 1, 2, 3");
}
```

die Ausgabe

```
c  = 2
c  = 3
c != 1, 2, 3
```

Ohne `break` werden alle Anweisungen vom Einsprung bis zum Ende ausgeführt werden, selbst wenn sie zu einer anderen Alternative gehören.

Um ein derartiges Verhalten zu vermeiden, sollte man sich darum angewöhnen jede Alternative mit einem `break` zu beenden. Bei der letzten Alternative ist ein `break` natürlich ohne Wirkung. Wir fügen es trotzdem ein, damit es nicht vergessen wird, wenn später noch eine weitere Alternative hinzugefügt wird.

```
int c = 2;
switch (c) {
  case 1 : System.out.println("c = 1");
       break;
  case 2 :                    // Einsprung hier
       System.out.println("c = 2");
       break;                 // Aussprung hier
  case 3 : System.out.println("c = 3");
       break;
  default: System.out.println("c != 1, 2, 3");
       break;
}
```

Mehrfache Marken

Eine Alternative kann mit mehreren Marken begonnen werden. Das ist nützlich, wenn bei unterschiedlichen Werten die gleiche Alternative betreten werden soll. Z.B.:

```
char c;
...
switch (c) {
  case 'a': case 'e': case 'i': case 'o': case 'u':
       System.out.println("c ist ein kleiner Vokal");
       break;
  case 'A': case 'E': case 'I': case 'O': case 'U':
       System.out.println("c ist ein grosser Vokal");
       break;
  default: System.out.println("c ist kein Vokal");
       break;
}
```

Der Text `c ist ein kleiner Vokal` wird ausgegeben, wenn `c` den Wert `a`, oder `e`, oder ... hat. Entsprechendes gilt für die Vokale in Großbuchstaben. Sollen also mehrere Möglichkeiten gleich behandelt werden, dann werden einfach die entsprechenden Fallwerte wie im Beispiel hintereinander aufgelistet.

[27] Diese Obskurität hat Java von C geerbt.

Allgemeine Form der Switch–Anweisung

Die Switch–Anweisung hat folgende allgemeine Form:

`switch` (< *Ausdruck* >) {< *Alternative* > ... < *Alternative* > }

Wobei jede < *Alternative* > entweder eine `case`–Alternative:

`case` < *Wert* >: ... `case` < *Wert* >: < *Anweisung* > ... < *Anweisung* >

oder eine `default`–Alternative

`default` : < *Anweisung* > ... < *Anweisung* >

ist.

Switch nicht auf allen Typen

Als letztes sei erwähnt, dass die Switch–Anweisung eine schwerwiegende Beschränkung hat: Mit ihr kann nur über die Werte von diskreten Typen verzweigt werden: `byte`–, `char`–, `short`–, `int`– und Aufzählungstypen sind in einer Switch–Anweisung erlaubt. Strings beispielsweise sind dagegen nicht erlaubt, selbst wenn sie nur aus einem einzigen Zeichen bestehen:

```
  String operator = "+"
  switch ( operator ) { // GEHT NICHT: Verzweigung auf String nicht erlaubt !
     ....
  }
```

Switch in Java 7

In Java 7 wurde die Switch–Anweisung etwas weiter entwickelt. Jetzt ist es erlaubt über konstante Zeichenketten zu verzweigen. Ein Beispiel ist:

```
int z1 = Integer.parseInt(JOptionPane.showInputDialog("Zahl 1 ?"));
int z2 = Integer.parseInt(JOptionPane.showInputDialog("Zahl 2 ?"));

String op = JOptionPane.showInputDialog("Operation ?");

switch (op) {
  case "+" : case "ADD": JOptionPane.showMessageDialog(null, z1+z2); break;
  case "-" : case "SUB": JOptionPane.showMessageDialog(null, z1-z2); break;
  default:  JOptionPane.showMessageDialog(null, "Undefinierte Operation");
}
```

1.4.4 Enumerationstypen: Enum

Manchmal hat man es mit einer Menge von Werten zu tun, für die es in der Programmiersprache keine direkte Entsprechung gibt. Beispielsweise könnte eine Anwendung sich auf die Wochentage *Montag*, *Dienstag*, und so weiter beziehen. In Java gibt es keinen Typ “Wochentag”. Als Ausweg könnte man Zahlen verwenden: 1 für *Montag*, 2 für *Dienstag*, etc. Betrachten wir ein kleines Beispiel, mit dem man morgens feststellen kann ob man zur Arbeit gehen muss.

```
import javax.swing.JOptionPane;

public final class Tage {

  private Tage() {}

  public static void main(String[] args) {
    // Zahl als String eingeben und in Zahl umwandeln:
    String tagS = JOptionPane.showInputDialog("Bitte Tag eingeben (ganze Zahl im Bereich 1 .. 7)");
```

```
    int tag = Integer.parseInt(tagS);

    // Arbeiten gehen ?
    // (Als Zahl codierten Tag verarbeiten)
    if (1 <= tag && tag <= 5) {
      JOptionPane.showMessageDialog(null, "Heute ist ein Werktag. Geh zur Arbeit");
    } else if (6 <= tag && tag <= 7) {
      JOptionPane.showMessageDialog(null, "Heute wird ausgeschlafen");
    } else {
      JOptionPane.showMessageDialog(null, "Bitte ganze Zahl im Bereich 1 .. 7");
    }
  }

}
```

Der Code arbeitet mit Zahlen statt der eigentlich gemeinten Tage. Bei derart kleinen Programmen und so einfachen Codierungen wie *Montag* $\rightarrow$ 1, etc. ist das recht unproblematisch. Generell verschlechtern solche Codierungen aber die Lesbarkeit der Programme und erhöhen die Fehleranfälligkeit. So ist es beispielsweise nicht offensichtlich, ob der Montag oder der Sonntag der erste Tag ist und ob die Nummerierung mit 1 oder 0 beginnt.

Besser man nimmt Tage. In Java und vielen anderen Sprachen geht das einfach mit Aufzählungstypen:

```
import javax.swing.JOptionPane;

public final class Tage {

  private Tage() {}

  // Definition des Typs
  enum Wochentag {Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag};

  public static void main(String[] args) {
    // Einlesen als String
    String tagS = JOptionPane.showInputDialog("Bitte eine Tag eingeben");

    // String in Wochentag umwandeln
    Wochentag tag = Wochentag.valueOf(tagS);

    // Arbeiten gehen ?
    // (Tag verarbeiten)
    switch (tag) {
    case Montag: case Dienstag: case Mittwoch: case Donnerstag: case Freitag:
      JOptionPane.showMessageDialog(null, "Heute ist ein Werktag. Geh zur Arbeit");
      break;
    case Samstag: case Sonntag:
      JOptionPane.showMessageDialog(null, "Heute wird ausgeschlafen");
      break;
    }
  }

}
```

Aufzählungstypen harmonieren gut mit der Switch-Anweisung.

1.4.5 Arithmetische, Boolesche und bedingte Ausdrücke

Boolesche Ausdrücke und Werte

Wie in vielen anderen Programmiersprachen ist in Java eine Bedingung wie

```
x > y
```

ein *Ausdruck* (engl. *Expression*), der sich von einem "normalen" *arithmetischen Ausdruck* wie beispielsweise

```
x * y + 3
```

nur dadurch unterscheidet, dass sein Wert keine Zahl, sondern ein *Wahrheitswert* ist.

Bedingungen sind *logische* – oder wie man allgemein sagt – *boolesche Ausdrücke*. Sie werden wie diese ausgewertet und haben den Wert *wahr* (engl. *true*) oder *falsch* (engl. *false*). Wahr und falsch sind die beiden möglichen *Wahrheitswerte*, oder die beiden *booleschen*[28] Werte.

Die Wahrheitswerte: `true` und `false`

Wahr und falsch haben auch in Java Namen: `true` und `false`. Diese beiden Namen kann man auch in Programmen verwenden. So kann statt

```
if (a == a) ...
```

ebenso gut

```
if (true) ...
```

geschrieben werden. `a == a` hat ja stets den Wert `true`. Zugegeben, weder das eine noch das andere ist besonders sinnvoll. Die Sache wird allerdings dadurch interessanter, dass boolesche Ausdrücke nicht nur als Bedingungen verwendet werden können.

Boolesche Variable: Variable mit logischem Wert

Wahrheitswerte sind in Java "richtige" Werte: Sie haben einen Datentyp (`boolean`) und man kann Variablen definieren, deren Wert ein Wahrheitswert (`true` oder `false`) ist. Die Definition boolescher Variablen hat die gleiche Form wie die aller anderen Variablen. Ein Beispiel ist:

```
boolean b = true;
```

Boolesche Variablen können in Zuweisungen und Ausdrücken verwendet werden. Im folgenden Beispiel wird festgestellt welche von zwei eingegebenen Zahlen die größere ist:

```
public final class TestBoolean {

 private TestBoolean(){}

 public static void main(String[] args) {
    int a = 0, b = 0;
    boolean ma = false, // ma soll den Wert true erhalten falls a > b
          mb = false; // mb soll den Wert true erhalten falls b > a

    a = Integer.parseInt(args[0]);
    b = Integer.parseInt(args[1]);

    if (a > b) ma = true; else ma = false;
    if (b > a) mb = true; else mb = false;

    if (ma == true) System.out.println("a > b");
    if (mb == true) System.out.println("b > a");
  }
}
```

Was gibt das Programm aus, wenn zwei verschiedene und was wenn zwei gleiche Zahlen eingegeben werden?[29]

Boolesche Ausdrücke

Eine Bedingung ist ein boolescher Ausdruck, also ein Ausdruck mit booleschem Wert. Boolesche Variablen sind Variablen denen boolesche Werte zugewiesen werden können. Damit ist klar, dass ein boolescher Ausdruck nicht nur in einer Bedingung verwendet werden darf, sondern auch rechts vom Zuweisungszeichen ("=") stehen kann.

```
if (a > b) ma = true; else ma = false;
```

kann also zu

```
ma = a > b;
```

[28] Nach einem engl. Mathematiker namens Boole benannt.
[29] Das Programm gibt bei der Eingabe gleicher Zahlen nichts aus. Warum nicht?

vereinfacht werden. Weiterhin ist jede boolesche Variable schon für sich ein boolescher Ausdruck. Statt

```
if (ma == true) ...
```

schreibt man darum einfach

```
if (ma) ...
```

Bedingte Ausdrücke

Eine bedingte Anweisung ist eine Anweisung, die abhängig vom Wert eines booleschen Ausdrucks ausgeführt wird. Ein *bedingter Ausdruck* ist ein *Ausdruck*, der abhängig vom Wert eines booleschen Ausdrucks berechnet wird. Ein Beispiel für einen bedingten Ausdruck ist:

```
a > b ? a : b;
```

Diesen Ausdruck kann man lesen als *"Wenn* `a` *größer* `b` *dann* `a` *sonst* `b`*"*. Ein solcher Ausdruck kann als rechte Seite in einer Zuweisung erscheinen:

```
max = a > b ? a : b;
```

Oft erlaubt ein bedingter Ausdruck eine kompaktere und übersichtlichere Notation als die Formulierung einer äquivalenten bedingten Anweisung:

```
max = a > b ? a : b; // Zuweisung mit bedingtem Ausdruck

if ( a > b )     // Zuweisung in bedingter Anweisung
   max = a;      // mit aequivalenter Wirkung
else
   max = b;
```

Die *bedingte Anweisung* hat die Form:

< Bedingung > `?` *< Ausdruck >* `:` *< Ausdruck >*

Arithmetische und logische Operatoren

Ein *Operator* wie z.B. + oder $\star$ verknüpft Teilausdrücke zu neuen Gesamtausdrücken. Operatoren können logische oder arithmetische Argumente und auch logische oder arithmetische Werte haben. Operatoren mit arithmetischen Argumenten und arithmetischem Ergebnis sind +, −, $\star$ /.

Operatoren mit logischem Wert sind:

!	nicht (Verneinung)
==	gleich
!=	ungleich
<	kleiner
<=	kleiner–gleich
>	größer
>=	größer–gleich

Einige Beispiele für die Auswertung von Ausdrücken mit arithmetischen und logischen Operatoren sind (`int x, y; boolean a, b;`. Bitte nachvollziehen!):

- – Zustand vorher: $a = true, b = true, x = 3, y = 4$
  – Anweisung:
  ```
  a = (x == y);
  if (a) x = 5; y = 6;
  ```
  – Zustand nachher: $a = false, b = true, x = 3, y = 6$

- – Zustand vorher: $a = true, b = true, x = 3, y = 4$
  – Anweisung:
  ```
  a=true; b=true; x=3; y=4;
  a = ((x != y) == false);
  if (a == false) x = 5; else y = 6;
  ```

  - Zustand nachher: $a = false, b = true, x = 5, y = 4$

- Zustand vorher: $a = true, b = true, x = 3, y = 4$
  - Anweisung: `if (a==b) x = 5; y = 6;`
  - Zustand nachher: $a = true, b = true, x = 5, y = 6$

Vorrangregeln

Die *Vorrangregeln* (Prioritäten) der Operatoren definieren, wie ein nicht vollständig geklammerter Ausdruck zu interpretieren ist. Durch den höheren Vorrang von $*$ gegenüber $+$ wird beispielsweise

$a*b+c*d$
als
$(a*b)+(c*d)$
und nicht etwa als
$a*(b+c)*d$
interpretiert.

Die Vorrangregeln sind (Vorrang zeilenweise abnehmend, innerhalb einer Zeile gleich):

`!, +, -`	unäre Operatoren
`*, /, %`	Multiplikation, Division, Modulo
`+, -`	Addition, Subtraktion
`<, <=, >, >=`	Vergleich
`==, !=`	Gleich, Ungleich
`&&`	Und
`||`	Oder
`=`	Zuweisung

Im Zweifelsfall und bei komplizierten Konstrukten sollten immer *Klammern verwendet* werden! Auch wenn jeder Ausdruck in Java eine eindeutige Interpretation hat, sollte man doch nicht zu sparsam mit Klammern umgehen. Schnell hat man die Vorrangregeln verwechselt oder vergessen und vielleicht hat der menschliche Geist Wichtigeres zu speichern als Vorrangregeln.

1.5 Funktionen

1.5.1 Konzept der Funktionen

Funktionen: Funktionsargumenten einen Funktionswert zuordnen

Funktionen sind aus der Mathematik als Zuordnungen von Werten zu Argumenten bekannt. Funktionen werden üblicherweise in Form einer Rechenvorschrift definiert. Beispielsweise wird mit

$f(x,y) = 2*x+y$

eine Funktion f definiert, die – beispielsweise – den *Argumenten* 3 und 2 den *Wert* 8 zuordnet.

Eine Funktion kann als Verfahren (Algorithmus) verstanden werden, mit dessen Hilfe Argumentwerten ein Ergebniswert zugeordnet wird (Siehe Abbildung 1.9):

Interpretation der Funktion f als **Algorithmus**:

> $f(x,y)$:
> Nimm den Wert x und verdoppele ihn.
> Addiere dazu den Wert y.
> Das Ergebnis ist der Funktionswert zu x und y

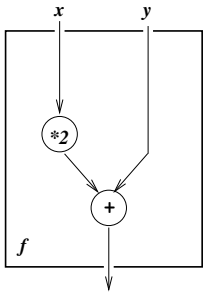


Abbildung 1.9: Funktion als Algorithmus

Eine Funktion in einem Java Programm: eine statische Methode

Eine Funktion ist also ein Algorithmus, ein Verfahren um etwas zu berechnen. In einem Programm kann dieser Algorithmus zu dem gesamten Algorithmus des Programms beitragen. Er muss dazu in der korrekten Notation exakt nach den Vorschriften der Programmiersprache ausgedrückt werden. In Java werden Funktionen als *statische Methoden* implementiert. Nehmen wir die Funktion f von oben als Beispiel:

in Java–Notation	in mathematischer Notation
`static int f(int x, int y) { return 2*x + y; }`	$f(x,y) = 2*x+y$

Die Java–Notation ist sicher intuitiv genauso verständlich wie die mathematische. Der Unterschied besteht eigentlich nur darin, dass dem Ergebnis und den Parametern Typen zugeordnet werden. Außerdem wird in Java `return` geschrieben.

Funktionsaufruf: Funktion verwenden

Ein Java–Programm, das die Definition einer Funktion enthält, kann diese auch verwenden. Man spricht dabei von einem *Aufruf der Funktion*. Im folgenden Beispiel–Programm wird f definiert, dann werden zwei Zahlen als Argumente der Kommandozeile eingelesen, die Funktion f auf sie angewendet und das Funktionsergebnis ausgegeben.

```
public final class FTest {

  private FTest(){}
```

```
  public static void main(String[] args) {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    int c = f(a, b);              // Aufruf
    System.out.println("f(" + a + "," + b + ") = " + c);
  }

  static int f(final int x, final int y) { // Definition
    return 2*x+y;
  }
}
```

Mit der (Funktions-) Definition wird gesagt, wie die Funktion arbeitet. Beim (Funktions-) Aufruf werden die Argumente festgestellt, an die Funktion übergeben und der Funktionswert entsprechend der Definition berechnet. Im Beispiel oben:

- liest das Programm zwei Zahlen von der Kommandozeile ein und speichert sie in `a` und `b`,
- dann wird die Funktion `f` aufgerufen und die Werte von `a` und `b` an `f` übergeben.
- `f` speichert die übergebenen Werte in `x` und `y`,
- `f` berechnet den Funktionswert `2*x+y`
- und gibt diesen dann zurück an die `main`–Funktion.
- `main` speichert den von `f` berechneten Wert in `c`
- und gibt ihn dann aus.

Funktionen sind Teil–Algorithmen des Programms. Sie agieren *innerhalb* des Programms. Ihre Argumente (Parameter) und Werte dürfen nicht mit der Ein– und Ausgabe des Gesamtprogramms verwechselt werden. Mit den Ein– / Ausgabe–Anweisungen (`System.out.println(..)`) kommuniziert das Programm mit dem Benutzer. Durch die Parameterübergabe und die Rückgabe des Ergebnisses kommunizieren `main` und `f`.

1.5.2 Funktionen genauer betrachtet

Ablauf von Programmen mit Funktionen

Bei Programmen ohne Funktionen kann man mit einiger Berechtigung sagen, dass ihr Ablauf “oben” beginnt und dann mehr oder weniger gerade bis “unten” weitergeht. Wenn ein Programm Funktionen enthält, dann ist das nicht mehr so: der Ablauf “springt” jetzt zwischen den Funktionen hin und her. Es beginnt *immer*(!) mit der `main`–Funktion, was auch immer im Text davor stehen mag! Von `main` aus kann es mit einem Funktionsaufruf in einer anderen Funktion weitergehen, dann zurück wieder in `main`, dann wieder in die gleiche oder eine andere Funktion, und so weiter; so lange, bis `main` endet. Beispielsweise wird im folgenden Programm zuerst `main` aktiviert, von dort geht es zu g, dann zurück nach main mit dem Wert den `g` berechnet hat, dann geht es zu `f` und wieder zurück:

```
public final class FTest {

  private FTest(){}

  static int f(final int x, final int y) { // Definition von f
    return 2*x+y;
  }

  public static void main(String[] args) { // Definition von main: Startpunkt
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    int c = g(a, b);               // Aufruf von g, Rueckkehr aus g
    System.out.println("g(" + a + "," + b + ") = " + c);
    c = f(a, b);                   // Aufruf von f, Rueckkehr aus f
    System.out.println("f(" + a + "," + b + ") = " + c);
  }

  static int g(int x, int y) {     // Definition von g
    return x-y+3;
  }

}
```

Definition einer Funktion

Funktionen können in Java als statische Methoden in beliebiger Anzahl in einer Klasse definiert werden. Wir haben die Funktionen f und g gesehen. Ein weiteres Beispiel ist:

```
...
static int max(final int x, final int y) {
  int resultat = x;
  if ( y > x ){
    resultat = y;
  }
  return resultat;
}
...
```

Hier wird eine Funktion mit dem Namen `max` definiert. Sie hat zwei Argumente und ein Ergebnis vom Typ `int`:

```
static int max(final int x, final int y) {
```

Mit dem Schlüsselwort **final** wird zum Ausdruck gebracht, dass der Wert des Parameters innerhalb der Funktion nicht verändert wird.

Eine Funktion kann beliebig viele Anweisungen enthalten. Sie endet mit der Ausführung der `return`–Anweisung. Eine korrekte Definition der Maximum–Funktion ist darum auch:

```
...
static int max(final int x, final int y) {
  if ( x > y ){
    return x;
  }
  return y;
}
...
```

Sichtbarkeit: `public`, Paket–Sichtbar, oder `private`

`main` wurde in unseren Beispielen stets als `public` definiert, alle anderen Funktionen dagegen nicht.

```
...
public static void main (....) ...
...
static int f(...) ....
static int g(...) ....
static int max(...) ....
```

Das liegt daran, dass `main` von "außen aus" aufgerufen wird, die anderen Funktionen aber nur eine interne Rolle spielen. Sie werden von `main` aus aufgerufen und müssen nur `main` bekannt sein, aber nicht der sonstigen weiteren Öffentlichkeit. Sie sind darum nicht `public`.

Selbstverständlich hätten wir sie auch als `public` definieren können, aber, nach dem Motto, nach dem man der Öffentlichkeit nur das zeigt, was sie auch etwas angeht, ist man bei der *Sichtbarkeit* nicht unnötig großzügig. `f`, `g` und `max` sind für den internen Gebrauch und darum nicht `public`. Dagegen ist `main` für den Aufruf durch andere (das Betriebssystem) da, es muss also `public` sein.

Eine noch größere Privatheit erreicht man, wenn man, statt nur auf `public` zu verzichten, eine Funktion als `private` definiert:

```
...
public static void main (....) ... // public maximal oeffentlich
static int f(...) ....      // keine public: privater
private static int g(...) .... // private: ganz privat
```

Mit diesen Regelungen der Sichtbarkeit werden wir uns später noch genauer auseindersetzen.[30] "*Sichtbarkeit*" bedeutet so viel wie "Wo ist die Definition sichtbar und kann infolgedessen verwendet werden?" Der Begriff *Sichtbarkeit* bezieht sich auf den

[30] Für Ungeduldige: Durch `public` wird eine Verwendbarkeit von überall aus zugelassen; der Verzicht auf `public` beschränkt die Verwendung auf das aktuelle `package` und `private` auf die aktuelle Klasse.

Quellcode, nicht auf die Bekanntheit oder das Sehen durch Menschen.[31]

Funktionsaufruf: formale und aktuelle Parameter

Die Verwendung einer Funktion wird allgemein auch Funktions–*Aufruf* genannt. Im Beispiel wird die Funktion `min` zweimal in einer Variablendefinition aufgerufen:

```
int m1 = min(a,b);
int m2 = min(c,d);
```

Funktionsaufrufe können wie beliebige Ausdrücke mit dem Typ des Ergebnisses verwendet werden, z.B:

```
c = 2 * min(a,b) + f(min(c,2), 5);          oder
System.out.println(min(c,d)+12);
```

Beim Funktionsaufruf werden die Argumente ausgewertet und als *aktuelle Parameter* an die Funktion übergeben. Bei dieser Parameterübergabe wird der Wert eines *aktuellen Parameters* zu dem des entsprechenden *formalen Parameters*.

- *Aktueller Parameter (Argument):* Der Wert der an die Funktion übergeben wird, mit dem sie also rechnet.
- *Formaler Parameter:* Die Variable in der der übergebene Wert in der Funktion gespeichert wird.

```
static int f(final int x, final int y) {
   ... // formale Parameter: x, y
}

...
int a = 1;
int b = 2
... f(a+2, b) ... // aktuelle Parameter = Argumente: 3, 2
```

Die Funktion wird mit den Werten abgearbeitet und liefert dann ihr Ergebnis zurück.

Funktionskopf und –körper

Eine Funktionsdefinition besteht aus einem *Funktionskopf* (engl. *Header*):

```
static int min (final int x, final int y) // Funktionskopf
```

und dem *Funktionskörper* (engl. *Body*), dem Rest der Definition. Im Kopf werden die *formalen Parameter* definiert. Im Funktionskörper folgen die Anweisungen, die ausgeführt werden, wenn die Funktion aufgerufen wird.

Die `return`–Anweisung

Der Körper einer Funktion ist eine zusammengesetzte Anweisung, die üblicherweise eine oder mehrere `return`–*Anweisung(en)* enthält. Eine `return`–Anweisung veranlasst die Funktion mit dem angegebenen Wert zurückzukehren. Die Funktion wird abgebrochen und der Wert der `return`–Anweisung wird zum Wert des Aufrufs. Beispielsweise hat der Aufruf

```
min (3, 4)
```

den Wert 3, da `min` mit `return x` – mit einem `x`–Wert von 3 – abgebrochen wird.

Achtung: Die `return`–Anweisung darf nicht mit der Ausgabe–Anweisung (`System.out.print`) verwechselt werden! Mit einer Ausgabe–Anweisung wird ein Wert aus dem gesamten Programm heraus transportiert. `return` dagegen bewegt einen Wert innerhalb des Programms.

Anwendung von Funktionen

Eine Funktion ist ein Programmstück zur Berechnung eines (Ergebnis–) Wertes aus einem oder mehreren (Argument–) Werten. Sie ist zunächst einfach als Mechanismus zur Abkürzung eines Programms zu verstehen. Die Abkürzung besteht darin, dass Wiederholungen vermieden werden. Nehmen wir an, in einem Programm soll an zwei Stellen das Minimum von zwei Zahlen berechnet werden.

[31] Sichtbarkeit als technischer Begriff und Bekanntheit durch Menschen sollten strikt getrennt werden, aber es gibt Zusammenhänge: Wer in seinem Code den Aufruf einer Funktion hinschreibt, sollte dies nicht nur ohne Gemecker des Compilers tun können (Funktion ist technisch sichtbar), er sollte die Funktion auch wirklich kennen. Umgekehrt braucht man das, was man nicht verwendet, auch nicht zu kennen.

```
...
int m1 = 0;
if (a < b) m1 = a;
else     m1 = b;

int m2 = 0;
if (c < d) m2 = c;
else     m2 = d;
...
```

Hier wird zweimal mit praktisch dem gleichen Code gearbeitet. Bei solchen kleinen und trivialen Programmstückchen ist die Wiederholung nicht sehr tragisch, aber wenn es sich um ein paar hundert Zeilen handelt, kann das schon sehr lästig sein. Bereits sehr früh hat man sich darum mit den Funktionen einen Abkürzungsmechanismus überlegt und in den ersten Sprachen realisiert.[32] Mit einer Funktion schreibt man kürzer und besser verständlich:

```
...
int m1 = min (a,b);
int m2 = min (c,d);
...
```

1.5.3 Argumente und Ergebnis einer Funktion

Funktionen sind typgebunden

Der deutlichste Unterschied zwischen Java–Funktionen und den Funktionen aus der Alltagswelt ist ihre Bindung an bestimmte Typen. Sowohl die Typen der Argumente einer Java–Funktion, als auch die des Ergebnisses müssen zu der Verwendung passen. Mit

```
static int f(int x, int y) {...}
```

sind die folgenden Aufrufe natürlich *nicht* erlaubt:

```
String d = f(1,2);  //Ergebnis passt nicht
int    c = f(a,"x"); //Argument 2 passt nicht
int    c = f(1,2,3); //Argumentzahl passt nicht
```

Konversionen

Die Typbindung ist streng, aber nicht dogmatisch. Wenn ein `double`–Wert gefordert ist, dann darf auch ein `int`–Wert kommen, aber nicht umgekehrt. Etwas allgemeiner und formaler gilt: Wenn ein Wert den falschen Typ hat, aber jeder Wert dieses Typs ohne Informationsverlust in einen des gewünschten Typs konvertiert werden kann, dann ist der Wert auch zugelassen.

Nehmen wir beispielsweise an, dass `f` wie folgt definiert ist:

```
static double f(double x) {...}
```

dann ist folgender Aufruf

```
int a = ...
double c = f(a);
```

erlaubt, denn jeder int–Wert kann ohne einen Informationsverlust in einen `double`–Wert konvertiert werden. Umgekehrt gilt das nicht. Darum ist:

```
static double f(int x) {...}
...
double c = f(1.0); // FEHLER !!
```

nicht erlaubt. Ein `double`–Wert kann nicht garantiert verlustfrei in einen `int`–Wert konvertiert werden. Darum wird keiner von ihnen an Stelle eines `double`–Wertes akzeptiert. Auch dann nicht, wenn es, wie eventuell bei der `1.0`, zufällig im Einzelfall doch möglich sein sollte!

Das Gleiche gilt entsprechend für das Ergebnis eines Aufrufs. Auch hier muss der Typ des gelieferten Wertes gleich oder “schwächer” sein, als der erwartete Typ.

[32] Funktionen zur Strukturierung von Programmen wurden von Turing bereits in den 1940–ern vorgeschlagen. Zuses Plankalkül (1945 veröffentlicht) enthielt darüberhinaus noch wesentlich ambitioniertere Konstrukte, die erst sehr viel später tatsächlich realisiert wurden.

```
double  ← int          : OK
int     ← double       : Nicht OK
```

Prozeduren: Funktionen ohne Ergebnis

Funktionen sind nicht nur da, um eine Wertberechnung im mathematischen Sinne durchzuführen. Sie können auch dazu verwendet werden, um beliebige Aktionen zu einer Einheit zusammenzufassen. Die Ausgabe von Werten kann beispielsweise in eine Funktion ohne Ergebnis gepackt werden:

```
public static void main(String[] args) {
   ...
   drucke(x,y);
   ...
}

static void drucke(final int w1, final int w2) { // void-Funktion: kein Ergebnis
   System.out.println("Der erste Wert ist: "+ w1 + ", "
                + "Der zweite Wert ist: "+ w2);
}
```

Die Funktion `drucke` gibt etwas aus, aber sie hat kein Ergebnis! Hier sehen wir wieder den Unterschied zwischen

- Ausgabe: aus dem Programm heraus transportieren; und
- der Rückgabe eines Wertes mit `return`: Transport eines Wertes innerhalb des Programms von einer Funktion an deren Aufrufstelle.

Da in jeder Funktionsdefinition aus formalen Gründen ein Ergebnistyp angegeben werden muss, gibt es den Pseudotyp *void* (engl. *void:* nichts, ungültig).

Funktionen ohne Ergebnis dürfen auch `return`–Anweisungen enthalten. Allerdings darf damit kein Wert zurückgegeben werden:

```
static void max(final int x, final int y) {
   if ( x > y ){
      System.out.println("x ist die groessere");
      return;                          // return ohne Wert
   }
   System.out.println("y ist die groessere");
}
```

Eine Berechnung, die kein Ergebnis hat, sollte man eigentlich nicht *Funktion* nennen. *Prozedur* ist der korrekte Begriff für eine zu einer Einheit zusammengefassten Serie von Anweisungen. D.h. für eine Funktion ohne Ergebnis.

Funktionen ohne Parameter und Dateneingabe

Funktionen deren Aufgabe es ist, Daten aus– statt zurückzugeben, haben kein Ergebnis. Komplementär dazu brauchen Funktionen, die ihre Daten einlesen, keine Parameter. Sie sollen ihre Werte ja von außerhalb des Programms holen. Eine Funktion, die einen `int`–Wert von der Konsole einliest, kann damit als

```
static int readInt() { ... }
```

definiert werden. `readInt` liefert Informationen an ihre jeweilige Aufrufstelle, nimmt aber von dort keine Informationen mit. Die Informationen kommen von außen, von der Konsole, hinein.

Eine der Möglichkeiten eine Routine zur Konsoleneingabe und ihre Verwendung ist:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public final class HalloWelt {

 private HalloWelt(){}

 public static void main(String[] args) {
   int x = readInt();
```

```
    int y = readInt();
    int m = max(x,y);
    System.out.println("die groesste eingegebene Zahl war "+ m);
  }

  static int readInt(){
    BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));
    String s;
    int i;
    try {
      s = br.readLine();
      i = Integer.parseInt(s.trim());
      return i;
    } catch (IOException e) {
      System.err.println("Eingabefehler" + e);
    }
    return 0;
  }

  static int max(final int x, final int y) {
    if ( x>y ) return x;
    return y;
  }
}
```

Hier kommen einige Konstrukte zum Einsatz, die wir später noch genauer betrachten wollen. Wir verwenden eine Funktion um `int`–Werte von der Konsole einzulesen. Sie nutzt dazu andere Bestandteile der Java–Bibliothek, als die Variante die wir weiter oben sahen. Dort wurde die Klasse `Scanner` eingesetzt. Selbstverständlich können wir auch eine Einlese–Funktion definieren, die mit der `Scanner`–Klasse arbeitet:

```
import java.util.Scanner;

...

static int readInt(){
  Scanner scan = new Scanner(System.in);
  int i = scan.nextInt();
  return i;
}

...
```

Diese Funktion ist einfacher, eine fehlerhafte Eingabe wird aber nicht abgefangen, sondern bringt das Programm kommentarlos zum Absturz.

Selbstverständlich könnte `readInt` seine Eingabe auch über ein Eingabefenster einfordern:

```
import javax.swing.JOptionPane;

...

static int readInt() {
  String xS = JOptionPane.showInputDialog("Bitte eine ganze Zahl");
  return Integer.parseInt(xS);
}

...
```

1.5.4 Funktionen als funktionale und prozedurale Abstraktionen

Funktionen als Abstraktionen

Funktionen dienen nicht nur dazu Schreibarbeit beim Verfassen zu reduzieren. Sie sind auch ein wesentliches Element bei der Strukturierung von Programmen. Dies kann man schon an recht einfachen Problemstellungen demonstrieren. Angenommen

wir wollen ein Programm zur Berechnung der Fläche eines Rings schreiben. Der Ring sei dabei durch einen inneren und einen äußeren Durchmesser gegeben. Die Lösung des Problems mit Funktionen ist:

```
import java.util.Scanner;

public final class Ring {

     private Ring (){}

     private static final double pi = 3.1415926;

     public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        double outer = scan.nextDouble();
        double inner = scan.nextDouble();

        System.out.println( areaRing(outer, inner) );
     }


     private static double areaRing( final double radiusOuter,
                              final double radiusInner) {
           return areaCircle(diameterOuter)
                  - areaCircle(diameterInner);
     }

     private static double areaCircle(final double radius) {
           return pi * radius * radius;
     }
}
```

Die Fläche des Rings wird als Fläche des äußeren Kreises minus Fläche des inneren Kreises berechnet. Im Programm erkennt man deutlich, dass das Problem *Fläche eines Rings* auf das Problem *Fläche eines Kreises* reduziert wird.

Wenn man die *Aufgabe* der Hilfsfunktionen kennt, kann man den Code der Funktion, die sie verwendet, verstehen, ohne sich im Einzelnen damit zu beschäftigen, wie genau die Hilfsfunktionen arbeiten. Beispielsweise versteht man die Funktion *main* mit folgenden Informationen:

```
import java.util.Scanner;

public final class Ring {

     private Ring(){}

     private static final double pi = 3.1415926;

     public static void main(String[] args) {
           Scanner scan = new Scanner(System.in);
        double outer = scan.nextDouble();
        double inner = scan.nextDouble();

        System.out.println( areaRing(outer, inner) );
     }

     // berechnet die Flaeche eines Rings mit
     // radiusOuter als aeusserem Radius und
     // radiusInner als innerem Radius
     //
     private static double areaRing( final double radiusOuter,
                              final double radiusInner) {
       .... ???? innere Arbeit uninteressant ??? .........
     }

}
```

ohne dass man sich weiter mit `areaRing` beschäftigen muss.

Um wiederum die Funktion *areaRing* zu verstehen reicht:

```
private static double areaRing( final double radiusOuter,
```

```
                        final double radiusInner) {
return areaCircle(radiusOuter)
   - areaCircle(radiusInner);
}

// berechnet die Flaeche eines Kreises mit
// radius als Radius
//
private static double areaCircle(final double radius) {
      .... ??? innere Arbeit uninteressant ???? .........
}
```

Vom Standpunkt der benutzenden Funktion aus, reicht es zu wissen, was eine Hilfsfunktion leistet und wie, mit welchen Parametern, sie aufzurufen ist. Wie genau sie arbeitet ist für den Benutzer so uninteressant, wie es für uns uninteressant ist zu wissen, wie unser neuer MP3–Player intern arbeitet. Wir sind zufrieden, wenn wir ihn bedienen können.

Dieses Prinzip, etwas zunächst einmal als schwarzen Kasten zu betrachten, dessen Innereien ignoriert werden, ist ein zentrales Prinzip, nicht nur der Informatik, sondern aller Ingenieurwissenschaften. In der Informatik spielt es aber eine besondere Rolle. Das liegt daran, dass das "Denken in schwarzen Kästchen" dabei hilft Komplexität zu beherrschen und die Informatik hat mit kaum einem anderen Problemen mehr zu kämpfen, als mit dem der Komplexität.

Ein solches schwarzes Kästchen bezeichnet man als *Abstraktion*. Eine Abstraktion abstrahiert von etwas, d.h. sie lässt Details weg. Man lässt uninteressante Einzelheiten weg und konzentriert sich auf das Wesentliche. Wesentlich *für den Benutzer* einer Funktion ist nur die korrekte Art des Aufrufs und die Wirkung einer Funktion.

> **Abstraktionen** dienen dazu Komplexität zu beherrschen. Es sind Sichtweisen auf Software–Konstrukte, bei den diese auf das Wesentliche reduziert werden. Was das Wesentliche ist, hängt vom Betrachter und seiner Sichtweise ab.

Die Schnittstelle einer Methode

Besonders wichtig sind die Abstraktionen, die die Sicht des Benutzers zum Ausdruck bringen. Das ist im Fall eines Toasters im Prinzip das Gleiche wie bei einer Funktion oder Methode: Wie wird er/sie bedient und welche Wirkungen werden damit erreicht? Im Gegensatz dazu steht die Implementierung. In ihr wird festgelegt wie die gewünschte Funktionalität erreicht wird.

Den Teil einer Softwarekomponente, über den diese bedient werden kann, nennt man in Übereinstimmung mit dem allgemeinen Sprachgebrauch *Schnittstelle*. Eine Schnittstelle ist eine Stelle, an der zwei Elemente zusammenkommen. Beispielsweise der Bediener und das zu bedienende Gerät. In unserem Fall handelt es sich um die Softwarekomponente, die mit einem Benutzer zusammentrifft. Im Fall einer Funktion / Methode ist der Benutzer eine aufrufende Funktion und die benutzte Komponente die aufgerufene Funktion. Sie kommen im Funktionsaufruf zusammen. Damit es zusammen passt, muss der Name passen und die Parameter müssen in Anzahl und Typ passen. Ausserdem muss der Rückgabetyp der Funktion an der Aufrufstelle stimmen.

Die *Schnittstelle einer Funktion oder Methode* beschreibt darum exakt dies: Name, Parameterzahl und Typen, Ergebnistyp. Dazu kann man noch die Information über die Sichtbarbeit (`public / private`) nehmen und die Eigenschaft statisch zu sein oder nicht.

```
// Schnittstelle einer Methode
private static double areaRing( final double diameterOuter,
                        final double diameterInner) {
   ... Implementierung,          ...
   ... nicht Bestandteil der Schnittstelle ...
}
```

> Die **Schnittstelle** einer Softwarekomponente ist Teil der Komponente und enthält die Bestandteile, die bei ihrer Benutzung direkt verwendet werden.

Die Schnittstelle einer Methode ist ein Konzept, *kein* eigenständiges Konstrukt der Sprache Java. Im Quellcode tritt sie implizit auf. Durch Kommentare kann man die rein syntaktische Information der Schnittstelle über Namen und Typen noch um inhaltliche Komponenten erweitern:

```
/**
 * Berechnet die Flaeche eines Kreises
```

```
  * @param diameter Kreisumfang
  * @return Kreisflaeche
  */
private static double areaCircle(final double diameter) {
  ... Implementierung ...
}
```

Es ist klar, dass diese Kommentierung dem Benutzer mehr Informationen zu einer korrekten Benutzung der Methode gibt. Namen und Typen der Parameter und des Rückgabewertes sind die minimale Basis. Ohne deren Kenntnis kann die Methode nicht ohne Beschwerden des Compilers aufgerufen werden. Eine sinnvolle inhaltlich korrekte Verwendung erfordert mehr: Das Wissen um die Wirkung der Funktion, so wie sie in einer genauere und inhaltliche Beschreibung dargelegt wird.

Die Spezifikation einer Funktion

Die Schnittstelle ist der Teil einer Komponente der für ihre Benutzer wichtig ist. Mit ihr wird die korrekte Bedienung und deren Ergebnis festgelegt. Die Innereien der Komponente, die Implementierung, wird nicht berücksichtigt, sie interessieren den Benutzer nicht und sie gehen ihn ja auch nichts an. Es gibt noch eine Situation, in der man eine Komponente nur von außen, ohne Berücksichtigung der Implementierung, beschreiben will. Das ist dann der Fall, wenn die Komponente noch gar nicht existiert, wenn sie sich noch im Stadium der Planung befindet. Auch in dem Fall beschreibt man Bedienung und Verhalten. Der Adressat ist in dem Fall aber nicht der Benutzer sondern der Implementierer.

> Die **Spezifikation** einer Softwarekomponenten ist nicht Teil der Komponente. Sie enthält die Informationen, die zu ihrer Implementierung notwendig sind.

Die Begriffe *Spezifikation* und *Schnittstelle* sind eng verwandt. Von einer Spezifikation wird man allerdings einen höheren Grad an Präzision erwarten. Bei einer Methode besteht diese Genauigkeit darin, dass man exakt angibt unter welchen Bedingungen und mit welchen Parameterwerten die Funktion korrekt arbeitet, welchen Wert sie *genau* zurück gibt und ob Variablen außerhalb der Funktion verändert werden und wenn ja, welche und wie.

Ein Beispiel ist:

```
/**
 * Berechnet die Flaeche eines Kreises
 * @param diameter Kreisdurchmesser
 * @pre  diameter >= 0
 *   pi enthaelt Wert von PI mit einer Genauigkeit von 6 Stellen
 * @return Flaeche eines Kreies mit Durchmesser diameter
    *          mit einer Genauigkeit von 4 Stellen
 * @post -
 */
private static double areaCircle(final double diameter) {
  return Math.PI * (diameter/2) * (diameter/2);
}
```

Hier steht `pre` für *Precondition*, d.h. *Vorbedingung*. Hiermit ist die Bedingung gemeint, unter der die Funktion ein korrektes Verhalten garantiert. Die Vorbedingung bezieht sich auf etwas, das außerhalb der Verantwortung der Funktion liegt, von denen sie aber abhängt. Das sind, wie hier, typischerweise Argumente (Parameter) oder die Werte von Variablen, die von der Funktion benutzt, aber nicht selbst vollständig kontrolliert werden.

Die *Nachbedingung* (englisch *Postcondition*) beschreibt die Wirkung der Funktion. Der zurückgegebene Wert ist eine ganz wichtige Wirkung. Die Beschreibung des Rückgabewertes gehört darum eigentlich zur Nachbedingung. Wegen seiner besonderen Bedeutung beschreiben wir ihn aber mit einem eigenen Kommentarabschnitt. Der strukturierte Kommentar `post` beschränkt sich dann auf Modifikationen von Variablen außerhalb der Funktion, die von dieser vorgenommen werden, oder andere Dinge, die sie verändert. Unser Beispiel oben ändert nichts. Die Nachbedingung ist darum “leer”.

1.6 Schleifen und ihre Konstruktion

1.6.1 Die While–Schleife

Schleifen: Anweisungen wiederholt ausführen

Schleifen sind ein Mechanismus zur Wiederholung. Mit ihnen können Anweisungen mehrfach ausgeführt werden. Es gibt mehrere Arten von Schleifen. Eine *While–Schleife* führt Anweisungen so lange aus, wie die Bedingung erfüllt ist. Das folgende Beispiel enthält eine While–Schleife die fünf mal “Hallo” ausgibt. Die Variable `i` startet mit dem Wert 0 und solange (engl. “*while*”) `i<n` ist, wird `Hallo` ausgegeben und `i` erhöht:

```
static void saghallo(final int n) {
   int i = 0; // wie oft bisher
   while (i<n) {
     System.out.println("Hallo");
     i = i+1;
   }
}
```

Die beiden Anweisungen

```
System.out.println("Hallo");
i = i+1;
```

stehen nur einmal im Programmtext, sie werden aber mehrfach ausgeführt. Genau gesagt werden sie *solange* wiederholt wie die Bedingung

```
i<n
```

erfüllt ist.

While–Schleifen: von einer Bedingung gesteuerte Wiederholungen

Eine While–Schleife hat die Form

`while (` < *Bedingung* > `)` < *Anweisung* >

Die Bedingung steuert dabei die Wiederholung der Anweisung, die – wie im Beispiel oben – selbstverständlich auch eine zusammengesetzte Anweisung sein kann. Bei der Ausführung wird zuerst die Bedingung getestet. Falls sie *nicht* zutrifft, wird die Schleife *verlassen*. Falls doch, wird die Anweisung ausgeführt und es geht danach zurück zur Bedingung. Ist die Bedingung am Anfang nicht erfüllt, wird die Schleife beendet bzw. völlig übersprungen.

Im nächsten Beispiel werden Zahlen eingelesen und dann ihre Summe ausgegeben. Die Anzahl der einzulesenden Werte wird als Argument an die Funktion übergeben, zur Konsoleneingabe nutzen wir eine Funktion `readInt`:

```
static void summiere(final int n) {
   int a,     // eingelesene Zahlen
   s = 0,     // aufsummierte Zahlen
   i = 0;     // Schleifendurchlaeufe bisher

   while (i < n) {
     a = readInt();
     s = s + a;
     i = i+1;
   }
   System.out.println("Die Summe der " + n
       + " eingegebenen Zahlen ist: " + s);
}
```

Dynamisch bestimmte Zahl der Schleifendurchläufe

Die Zahl der Schleifendurchläufe muss nicht unbedingt beim Eintritt in die Schleife festliegen. Sie kann dynamisch, d.h. während der Laufzeit des Programms, bestimmt werden. Ein Beispiel ist folgende Funktion, die beliebig viele Zahlen aufsummiert. Solange keine Null eingegeben wird, liest sie ein und summiert auf. Schließlich gibt sie die Summe zurück:

```
static int summiere() {
  int a, // eingelesene Zahlen
  s = 0; // aufsummierte Zahlen
  a = readInt();
  while (a != 0) {
    s = s + a;
    a = readInt();
  }
  return s;
}
```

In der Schleife kontrolliert die Bedingung nicht mehr, wie weiter oben, ob die Zahl der Durchläufe eine vorgegebene Grenze erreicht hat. Sie kontrolliert, ob als letztes eine Null eingegeben wurde. Da zuerst ein Wert eingelesen werden muss, bevor wir prüfen können, ob er gleich Null ist, wurde die erste Leseoperation aus der Schleife herausgezogen. Man beachte auch, dass hier, anders als im letzten Beispiel, innerhalb der Schleife zuerst aufsummiert und dann gelesen wird.

1.6.2 0 bis N Zahlen aufaddieren

Beispiel: Die ersten n ganzen Zahlen aufsummieren

Die Konstruktion korrekter Schleifen ist ein wichtiger Schritt auf dem Weg zur systematischen Programmerstellung. Mathematische Folgen und Reihen bieten hier ein ausgezeichnetes Übungsmaterial.

Wir beginnen mit dem Aufsummieren der ersten n Zahlen:

$$s = \sum_{i=1}^{n} i$$

Bei der Konstruktion eines Programms ist es wichtig, Variablen und die in ihnen enthaltenen Werte zu unterscheiden. Natürlich besteht zwischen beidem ein Zusammenhang: eine bestimmte Variable enthält einen bestimmten Wert oder soll irgendwann einen bestimmten Wert enthalten. Variablen und Werte sind aber unterschiedliche Dinge. Welche Variablen welche Werte enthalten werden, wird von der Lösungsstrategie – dem *Algorithmus* – des Programms bestimmt.

Die Funktion zur Summierung der erste n natürlichen Zahlen ist:

```
static int summeBis(final int n) {
  int s = 0, // Summe bisher
  i = 0;  // zuletzt aufsummierte Zahl
  while (i < n) {
    i = i + 1;
    s = s + i;
  }
  return s;
}
```

Werte und Variablen in der Summationsaufgabe

Betrachten wir zuerst die *Werte (!)*. Bei der Berechnung der Summe spielen folgende Werte eine Rolle:

- n: Die Zahl bis zu der summiert werden soll. Diese Zahl wird an die Funktion übergeben.
- $i_0 = 0, i_1 = 1, i_2 = 2, \ldots i_n = n$ die Folge der Zahlen die aufsummiert werden.
- $s_0 = 0, s_1 = 1, s_2 = 3, \ldots s_n = \sum_{i=1}^{n} i$ die Folge der Zwischenergebnisse.

Es gibt also einen Wert n, viele i–Werte $(i_0, i_1, \ldots)$ und viele s–Werte $(s_0, s_1, \ldots)$.

Die Werte haben jeweils einen Platz in einer *Variablen (!)*. Manche Variablen enthalten immer den gleichen Wert, andere Variablen enthalten Wertefolgen:

- `n` enthält den Wert n
- `i` enthält im Laufe des Programms die Folge der i–Werte
- `s` enthält im Laufe des Programms die Folge der s–Werte.

`s` enthält erst am Ende des Programms den gesuchten Wert s. Vorher sind in ihm Teilsummen zu finden.

Den sich verändernden Inhalt der Variablen macht man sich am besten in einer Wertverlaufstabelle klar. Hier sind die Werte der Variablen jeweils beim Test der Schleifenbedingung zu finden (Eingabe $n = 5$):

n	i	s
5	0	0
5	1	1
5	2	3
5	3	6
5	4	10
5	5	15

Schleifenbedingung und Reihenfolge der Anweisungen

Bei einer Schleife können kleine harmlos aussehende Änderungen aus einem funktionierenden Programm ein falsches machen. Vertauscht man beispielsweise die beiden Anweisungen in der Schleife:

```
static int summiere(final int n) {
  int i = 0;
  int s = 0;
  while (i < n) {
    s = s + i;
    i=i+1;
  }
  return s;
}
```

dann gibt die Funktion ein falsches Ergebnis zurück: das letzte i fehlt in der Summe. Mit einer Veränderung der Schleifenbedingung zu `i <= n` kann der Fehler wieder repariert werden. Die Reihenfolge der Anweisungen in der Schleife und die Schleifenbedingung müssen also zusammen passen:

```
OK:                    OK:                    FALSCH:
Kleiner-gleich;        Kleiner;               Kleiner-gleich;
Zuerst addieren,       zuerst erhoehen,       zuerst erhoehen,
dann erhoehen:         dann addieren:         dann addieren:

while (i <= n) {       while (i < n) {        while (i <= n) {
  s = s + i;             i=i+1;                 i=i+1;
  i=i+1;                 s = s + i;             s = s + i;
}                      }                      }
```

1.6.3 Schleifenkontrolle: break und continue

Abbruch mit break

Die `break`–Anweisung ist uns bereits im Zusammenhang mit der `switch`–Anweisung begegnet: Mit `break` wird die `switch`–Anweisung sofort verlassen. Zum gleichen Zweck kann `break` in einer Schleife benutzt werden. Beispiel:

```
while ( true ) {
  int x = readInt();
  if ( x == 0 ) break; // Abbruch, Schleife verlassen
  ... x verarbeiten ...
}
```

Bei ineinander geschachtelten Schleifen wird nur die innerste verlassen.

```
while ( ... ) {  // Schleife 1
  ...
    while ( ... ) { // Schleife 2
      ...
      break;     // verlaesst Schleife 2, weiter in Schleife 1
      ...
```

```
    }
  ...
}
```

Weiter mit continue

Während mit `break` eine Schleife insgesamt abgebrochen wird, beendet `continue` nur den aktuellen Schleifendurchlauf. Beispiel:

```
System.out.println("Bitte nur positive Zahlen eingeben");
while ( true ) {
  int x = readInt();
  if ( x < 0 ) {
    System.out.println("Das war wohl nichts!");
    continue;  // Abbruch des Durchlaufs, weiter mit Schleifenanfang
  }
  // weiter in der Schleife
  ... x > 0 verarbeiten ...
}
```

Wird hier eine negative Zahl eingegeben, dann wird der Schleifendurchlauf, aber nicht die ganze Schleife abgebrochen. Es geht weiter mit einer erneuten Eingabeaufforderung.

`break` und `continue` können in jeder Art von Schleife benutzt werden, nicht nur in `while`–Schleifen.

1.6.4 Die For–Schleife

For–Schleife: Eine Folge durchlaufen

Mit einer While–Schleife wird eine Aktion solange wiederholt wie ihre Bedingung zutrifft. Mit der For–Schleife wird üblicherweise eine vorherbestimmte Werte–Folge durchlaufen und für jeden Wert der Schleifenkörper ausgeführt.

Beispielsweise erzeugt die Schleife

```
for (int i = 0; i < 5; i=i+1)
  System.out.println("Hallo Nr " + i);
```

die Ausgabe “Hallo Nr 0”, ... “Hallo Nr 4”.

`i` ist hier die sogenannte *Laufvariable*. Sie durchläuft die Werte 0, 1, 2, 3 und 4 und bei jedem Durchlauf wird die Anweisung ausgeführt. Ein etwas komplexeres Beispiel ist:

```
int max = 0; // bisher gefundenes Maximum
int v;
for (int k = 0; k < 10; k=k+1) {
     v = readInt();
     if ( (v > max) || (k == 0) ) max = v;
}
System.out.println(max + " ist die groesste der eingegebenen Zahlen");
```

Hier werden zehn Zahlen eingelesen und die größte Zahl dann ausgegeben. Noch etwas komplexer ist:

```
int max = 0, // bisher gefundenes Maximum
maxIndex = 0; // Nummer des bisher gefundenen Maximums
for (int k = 1; k <= 10; k = k + 1) {
     int v = readInt();
     if (k == 1) {
       max = v;
       maxIndex = 1;
     } else if (v > max) {
       max = v;
       maxIndex = k;
     }
}
System.out.println(max + ", die Zahl Nr. " + maxIndex
       + " ist die groesste");
```

Hier werden mit Hilfe einer For–Schleife 10 Zahlen eingelesen und die Nummer der größten festgestellt. Wir sehen, dass der Schleifenkörper eine zusammengesetzte Anweisung sein kann, dass die Laufvariable nicht unbedingt `i` heißen muss und auch nicht unbedingt von 0 an laufen muss.

Allgemeine Form der For–Anweisung

Die For–Anweisung hat folgende allgemeine Form:

$$\texttt{for (} < \mathit{Init-Anweisung} > < \mathit{Bedingung} > \texttt{;} < \mathit{Ausdruck} > \texttt{)} < \mathit{Anweisung} >$$

Alle Initialisierungs–Anweisungen enden in einem Semikolon, Bedingungen jedoch nicht. Darum ist hier nur ein Semikolon Die Form hier ist etwas vereinfacht die exakte Syntax von Java ist in der *Java Language Specification*[33] zu finden.

Die For–Schleife ist eigentlich eine verkleidete While–Schleife:

```
for (Init-Anweisung Bedingung; Ausdruck) Anweisung
```

ist nichts anderes als

```
Init-Anweisung while (Bedingung) {Anweisung Ausdruck;}
```

Die Schleife

```
for (int i = 0; i < 10; i=i+i)
  tueEtwas();
```

entspricht darum exakt

```
int i = 0;
while (i < 10) {
  tueEtwas();
  ++i;
}
```

Das Weiterschalten der Laufvariablen ist also in der For–Schleife immer die letzte Aktion. In einer While–Schleife können wir uns dagegen aussuchen, wo die Laufvariable erhöht wird.

Initialisierungsanweisung und Laufvariablen

Eine For–Schleife beginnt mit der *Initialisierungsanweisung*. In

```
for (int i=0; ...) ...
```

ist `int i=0;` die Initialisierungsanweisung. In ihr wird normalerweise die Laufvariable definiert und initialisiert. Eine so *lokal* definierte Laufvariable kann nur innerhalb der Schleife verwendet werden. Folglich ist

```
int x;
for (int i = 0; i < 5; ++i)
  x = i;    // OK
x = i;      // FALSCH i ist hier nicht definiert !
```

fehlerhaft und wird vom Compiler nicht akzeptiert.

Eine For–Schleife kann auch eine “normale” Variable als Laufvariable benutzen. Die Laufvariable wird dann außerhalb der Schleife definiert und in der Initialisierungsanweisung nur noch initialisiert:

```
int i;
...                         // beliebige Verwendung von i
for (i = 0; i < 5; ++i)     // i ist Laufvariable der Schleife
  ...i...;                  // OK
...i...;                    // OK
...
```

Laufvariablen mit gleichem Namen können in verschiedenen Schleifen lokal definiert werden. Gleichzeitig kann es sogar noch eine Variable mit diesem Namen außerhalb der Schleife geben:

[33] `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`

```
double i = 1;          // i Nr. 1
for (int i=20; i<25; i=i+1) // i Nr. 2
   ...i...;            // i Nr. 2

for (int i=30; i<35; i=i+1) // i Nr. 3
   ...i...;            // i Nr. 3

....i...;              // i Nr. 1
```

Hier werden drei verschiedene `i`–s definiert, die sich gegenseitig aber nicht in die Quere kommen.

Der Operator++

Zur Abkürzung von `i = i+1` wird oft `i++` verwendet. Speziell in For–Schleifen ist es üblich diesen *Inkrement–Operator* zu verwenden:

```
for (int i = 0; i < 10; i++) ...
```

ist also eine Kurzform von

```
for (int i = 0; i < 10; i=i+1) ...
```

Die Schleifenbedingung

Genau wie die While–Schleife wird auch die For–Schleife durch eine Bedingung gesteuert:

```
for (int i = 0; i < 10; i++) ...
```

läuft solange wie `i < 10` den Wert `true` hat. Ebenfalls wie bei While wird die Schleife gar nicht erst betreten, wenn die Bedingung am Anfang nicht erfüllt ist. Man darf nicht vergessen, dass im Schleifenkörper die Laufvariable verändert und damit die Bedingung beeinflusst werden kann. So wird etwa die Schleife

```
for (int i = 0; i < 10; i++) i=i-1;
```

niemals terminieren (= enden).

Foreach–Schleife: Eine Kollektion durchlaufen

Mit einer speziellen Variante der For–Schleife können die Elemente einer Kollektion von Werten durchlaufen werden. Ihre allgemeine Form ist:

```
for ( <Typ> <Variable> : <Ausdruck> ) <Anweisung>
```

Der Ausdruck muss dabei eine Kollektion von Werten bezeichnen. Ein Beispiel ist:

```
for ( int x : new int[]{1,2,3,4,5} )
System.out.println(x);
```

Mit dieser Schleife werden die Zahlen `1` bis `5` ausgegeben. Weiter unten wird auf diese Form der Schleife, die sogenannte *For–Each*–Schleife genauer eingegangen.

1.6.5 Die Do-While–Schleife

Beispiel

Die Do-While–Schleife stellt eine weitere Variante der Schleifen in Java dar. Ihre Wirkung kann leicht an einem Beispiel gezeigt werden:

```
do {
  System.out.println("naechster Wert, Stop mit 0: ");
  i = readInt();
  sum = sum + i;
} while ( i != 0);
```

Hier werden so lange Zahlen eingelesen und aufsummiert, bis eine 0 angetroffen wird. Im Unterschied zur While–Schleife wird bei dieser Form die Bedingung nicht jeweils vor sondern *nach* der Anweisung geprüft. Die Do–While–Schleife setzt man darum immer dann ein, wenn wie hier der Schleifenkörper in jedem Fall mindestens einmal betreten werden muss.

Allgemeine Form

Die Do-While–Schleife hat folgende allgemeine Form:

```
do <Anweisung> while (<Bedingung>);
```

1.6.6 Die For-Each–Schleife

Beispiel

Die For–Each–Schleife ist eine elegante Form alle Werte einer Datenstruktur zu durchlaufen. Da wir uns noch nicht mit Datenstrukturen beschäftigt haben, wird sie hier der Vollständigkeit halber erwähnt. Eine genauere Betrachtung folgt im Zusammenhang mit Feldern.

Allgemeine Form

Die For-Each–Schleife hat folgende allgemeine Form:

```
for (<Typ> <Variable> : <Datenstruktur>) <Anweisung>
```

Die Anweisung wird für jeden Wert in der Datenstruktur ausgeführt. Vor der Ausführung wird der Wert der Variablen zugewiesen

1.6.7 Schleifenkonstruktion: Zahlenfolgen berechnen und aufaddieren

Beispiel: beliebige Zahlenfolgen aufsummieren

Neben Erfahrung und ein wenig Begabung ist auch etwas Systematik eine nützliche Zutat bei der Konstruktion einer Schleife.[34] Angenommen, wir wollen die Summe

$$a_1 + a_2 + a_3 + ... + a_n$$

mit

$$a_i = a_{i-1} + c$$

berechnen.

Für $c = 2$, $n = 6$ und $a_1 = 0$ gilt beispielsweise

$$0 + 2 + 4 + 6 + 8 + 10 = 30$$

Dazu schreiben wir eine Funktion `summe`, die die Zahlenfolge für beliebige Werte a_1, n und c aufsummiert. Der erste Entwurf nach diesen Vorgaben ist:

```
/**
 * @param a_1 Anfang: 1-ter Summand
 * @param n letzter Index
 * @param c Konstante
 * @return a_1 + a_2 + ... + a_n
           mit a_2 = a_1+c ...
```

[34] Nicht nur bei der Konstruktion von Schleifen, bei allen Programmkonstruktionen ist systematisches Vorgehen nützlich.

```
  */
static int summe(int a_1, int n, int c) {
  int s;   // Summe
  //s berechnen ??
  return s;
}
```

Dieser Entwurf dient in erster Linie dazu, sich über die Parameter und ihre Rolle klar zu werden und festzulegen, welche Aufgabe die Funktion überhaupt zu erledigen hat. Wir dokumentieren dies hier durch Kommentare im offiziellen Stil.[35]

Summe berechnen: der "händische Algorithmus" benötigt zu viele Variablen

Es fehlt noch die tatsächliche Berechnung der Summanden und der Summe

$$S = a_1 + a_2 + a_3 + ...a_n$$

wobei die einzelnen Summanden a_i jeweils durch

$$a_i = a_1 + c + c + \ldots$$

bestimmt werden. Bevor eine weitere Zeile Programmcode geschrieben wird, teste man unbedingt sein Problemverständnis, indem mindestens ein Beispiel *per Hand* gerechnet wird. Wir berechnen also als Beispiel die Summe für die Werte $a_1 = 0$, $n = 4$ und $c = 2$:

- Schritt 1: Wir bestimmen die Summanden a_i (die Summenglieder) und schreiben sie auf:

  $a_1 = 0 \quad a_2 = 0+2 = 2 \quad a_3 = 2+2 = 4 \quad a_4 = 4+2 = 6$

- Schritt 2: Wir berechnen aus ihnen Schritt für Schritt die Summe:

  $s_1 = 0+0 = 0 \quad s_2 = 0+2 = 2 \quad s_3 = 2+4 = 6 \quad s_4 = 6+6 = 12$

Dieser "händische Algorithmus" ist für das Programm *nicht* gut geeignet: Wir haben im Programm keinen Platz, um eine beliebig lange Wertefolgen abzulegen. In einzelnen Variablen kann zwar jeweils ein Wert abgelegt werden. Natürlich könnten wir 10 oder 50 Variablen anlegen, aber selbst wenn wir 100 Variablen für die Wertefolge vorsehen, kann es sein, dass 101 als Wert von n erscheint und 100 nicht ausreichen.

Summanden und Teilsummen gleichzeitig berechnen

Der Summationsalgorithmus, der für unser Programm geeignet ist, besteht darin, sukzessive neue a_i–Werte und gleichzeitig weitere Teilsummen s_i zu berechnen:

	Schritt 1	Schritt 2	Schritt 3	Schritt 4
Summand a_i:	0	$0+2=2$	$2+2=4$	$4+2=6$
Teilsumme s_i:	$0+0=0$	$0+2=2$	$2+4=6$	$6+6=12$

In jedem Schritt werden genau zwei Werte gebraucht: a_i und s_i. Egal wie groß n ist, also wie viele Schritte ausgeführt werden müssen, wir kommen mit zwei Variablen aus! Etwas formaler ausgedruckt, haben wir jetzt folgenden Algorithmus angewendet:

- Neue Summanden a_i werden aus alten a_{i-1} durch Addition von c gebildet: $a_i = a_{i-1} + c$
- Neue Teilsummen s_i werden durch Addition des neuen Summanden und der alten Teilsumme gebildet: $s_i = s_{i-1} + a_i$

Diese beiden Berechnungen werden so lange wie notwendig wiederholt. Sie sind die Aufgabe einer Schleife. Im Schleifenkörper müssen also s_i und a_i berechnet werden:

$a_i = a_{i-1} + c$
$s_i = s_{i-1} + a_i$

[35] Dem Leser wird ans Herz gelegt, seine Funktionen in dieser Art zu dokumentieren. Um den Text nicht unnötig aufzublähen halten wir uns hier nicht immer an diesen – empfohlenen – Stil.

Wertverlaufstabelle der a_i und s_i

Die a_i und s_i sind Werte–*Folgen*. Die Wertefolgen werden in jeweils einer Variablen abgelegt: Die Folge der a_i in der Variablen `a`, die Folge der s_i in der Variablen `s`. (Nicht vergessen: Variablen und ihre wechselnden Werte sind zu unterscheiden!) Wir schreiben zunächst eine Wertverlaufstabelle für `a` und `s` auf. (Werte jeweils vor dem Test der Bedingung):

a	s
a_1	$s_1 = a_1$
$a_2 = a_1 + c$	$s_2 = a_1 + a_2$
$a_3 = a_1 + c + c$	$s_3 = a_1 + a_2 + a_3$
$a_4 = a_1 + c + c + c$	$s_4 = a_1 + a_2 + a_3 + a_4$
..	..

Konstruktion des Schleifenkörpers aus der Wertverlaufstabelle

Aus dieser Tabelle wollen wir jetzt eine Schleife konstruieren. Jede Zeile stellt die Belegung der beiden Variablen zum Zeitpunkt des Tests der Bedingung dar. Der Übergang von einer Zeile zur nächsten beinhaltet darum die Wirkung der Anweisungen in der Schleife. Die Anweisungen sind gesucht. Also ist die Frage, welche Anweisungen führen uns von einer Zeile zur anderen?

Von a_0 zu a_1 und von a_1 zu a_2 etc. kommt man, indem jeweils c zum Inhalt von `a` addiert wird. Zum Inhalt von `s` muss dann dieser neue Wert von `a` addiert werden und man kommt zur nächsten Zeile von `s`:

```
//s und a_n berechnen:
int a = ...;
...
while (...) {
  a = a + c; //neues a = altes a + c
  s = s + a; //neues s = altes s + neues a
}
```

Das “neue a” ist der neue Inhalt der Variablen `a`. In der Tabelle ist es in der neuen Zeile in der `a`–Spalte zu finden. Das “neue s” wird aus dem “alten s” (eine Zeile weiter oben) und dem “neuen a” (gleiche Zeile) berechnet.

Der Schleifenkörper hat also die Aufgabe den Übergang von einer Zeile zur nächsten in der Wertverlaufstabelle zu bewerkstelligen (Siehe Abbildung 1.10).

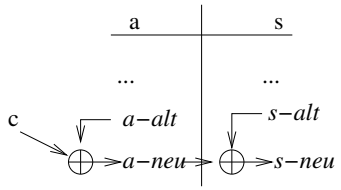


Abbildung 1.10: Schleifenkörper als Zeilenübergang

Berechnung einer neuen Zeile:

- Zuerst wird a_{neu} berechnet: $a_{neu} \leftarrow a_{alt} + c$
- Daraus dann s_{neu}: $s_{neu} \leftarrow a_{neu} + s_{alt}$

Schleifeninitialisierung

Die Schleife muss mit den korrekten Anfangswerten beginnen. Diese können sofort aus der ersten Zeile der Tabelle entnommen werden:

```
//s und a_n berechnen:
int a, s;
a = a_1; // Belegung der
s = a_1; // ersten Zeile
while (...) {
  a = a + c;
  s = s + a;
}
```

Schleifenbedingung

Die Bedingung für den Abbruch der Schleife kann nicht der Tabelle entnommen werden. Wir müssen entweder die Zahl der Durchläufe oder den Index der addierten Summanden mitprotokollieren. Entscheidet man sich für den Index des zuletzt addierten Summanden, dann bekommt die Tabelle folgende weitere Spalte:

a	s	i
a_1	$s_1 = a_1$	1
$a_2 = a_1 + c$	$s_2 = a_1 + a_2$	2
$a_3 = a_1 + c + c$	$s_3 = a_1 + a_2 + a_3$	3
$a_4 = a_1 + c + c + c$	$s_4 = a_1 + a_2 + a_3 + a_4$	4
..	..	..

Jede Zeile beschreibt den Zustand des Programms beim Test der Schleifenbedingung. Wir können darum stoppen, wenn `i` den geforderten Wert *n* erreicht hat:

```
//s und a_n berechnen:
int a = a_1;
int i = 1;
s = a_1;
while (i != n) {
  a = a + c;
  s = s + a;
  i = i + 1;
}
```

Man beachte die Reihenfolge der Anweisungen innerhalb der Schleife. Es ist wichtig, dass zuerst `a` und dann `s` verändert wird, da der neue Wert von `s` den neuen Wert von `a` benötigt.

Der Vollständigkeit halber noch die gesamte Funktion:

```
  /**
   * @param a_1 Anfang: 1-ter Summand
   * @param n letzter Index
   * @param c Konstante
   * @return a_1 + a_2 + ... + a_n
             mit a_2 = a_1 + c ...
   */
static int summe(int a_1, int n, int c) {
  int s;    // Summe
  int a = a_1; // aktueller Summand
  int i = 1; // aktueller Index
  s = a_1;
  while (i != n) {
    a = a + c;
    s = s + a;
    i = i + 1;
  }
  return s;
}
```

Die Grundbestandteile einer Schleife

Schleifen haben generell folgende *Grundbestandteile:*

1. *Die relevanten Variablen:* Die Variablen, die in der Schleife verwendet werden.
2. *Schleifeninitialisierung:* Die relevanten Variablen müssen mit den richtigen Initialwerten (ersten Werten) belegt werden.
3. *Schleifenbedingung:* Wann endet die Schleife?
4. *Schleifenkörper:* Wie werden die relevanten Variablen in einem Schleifendurchlauf verändert?

Am Anfang steht der Algorithmus: Wie wird das Ergebnis berechnet? Welche Werte bzw. Wertefolgen fallen dabei an?

Dann werden die relevanten Variablen festgelegt. Sie sind die Speicherplätze der Werte und Wertefolgen.

Als nächstes sollte der Schleifenkörper konstruiert werden. Dazu überlegt man systematisch, welche alten Werte die Variablen enthalten und wie aus den alten Werten die neuen berechnet werden können. Am besten benutzt man dazu eine Wertverlaufstabelle.

Als nächstes überlegt man, welches die richtigen Initialwerte für die Schleife sind. Hat man eine Wertverlaufstabelle, dann nimmt man einfach deren erste Zeile.

Schließlich kann die Schleifenbedingung formuliert werden.

1.7 Programmkonstruktion: Rekursion und Iteration

1.7.1 Rekursion

Rekursion: Definition mit Selbstbezug

Der Begriff *Rekursion* bedeutet so viel wie *zurück gehen*, meist wird es im Sinne eines *zurück gehen auf sich selbst* verwendet. Eine Definition ist *rekursiv*, wenn das zu Definierende mit Hilfe von sich selbst definiert wird. Das klassische Beispiel einer rekursiven Definition ist die Fakultätsfunktion fak:

$$\begin{aligned} fak(0) &= 1 \\ fak(n) &= fak(n-1)*n \end{aligned}$$

Die Definition unterscheidet zwei Fälle: entweder ist das Argument 0, dann ist der Wert 1, oder das Argument ist irgendeine andere Zahl n, dann ist der Wert $fak(n-1)*n$. Der Selbstbezug liegt im zweiten Fall, in dem $fak(n)$ mit Hilfe von $fak(n-1)$ definiert wird.

Rekursive Funktionen rufen sich also selbst auf. Dabei muss man auf zwei Dinge achten: Wird überhaupt etwas Sinnvolles definiert und für welche Argumente ergeben sich sinnvolle Ergebnisse. Unser Fakultätsbeispiel ist sinnvoll, aber nur für positive ganze Zahlen als Argumente. Mit einem negativen oder gebrochenen Argument funktioniert die Sache nicht: Die Definition definiert in diesem Falle nichts, praktisch laufen wir in eine *Endlos–Rekursion*:

$$fak(-1) = fak(-2)*-1 = fak(-3)*-1*-2 = fak(-4)*-1*-2*-3 = \cdots$$

Mit positiven ganzen Argumenten funktioniert es bestens:

$$fak(3) = fak(2)*3 = fak(1)*2*3 = fak(0)*1*2*3 = 1*1*2*3$$

Eine rekursive Definition muss auf ein Ende hinlaufen, wenn schon nicht für alle Argumente, dann doch für einige. Sie muss also einen *Definitionsbereich* haben, der nicht leer ist. Der Definitionsbereich der Fakultätsfunktion sind die natürlichen Zahlen inklusive der Null. Auf allen anderen Argumenten ist sie nicht definiert. Wobei "nicht definiert" sich als Endlos–Rekursion zeigen kann.

Direkte und indirekte Rekursion

Rekursive Funktionen sind Funktionen, die sich selbst aufrufen. Das kann wie oben direkt sein, oder indirekt über andere Funktionen. Ein Beispiel für *direkte Rekursion* haben wir oben mit der Fakultätsfunktion gesehen. Die Funktionen *gerade* und *ungerade*, die testen, ob ihr Argument gerade oder ungerade ist, sind *indirekt rekursiv*. Die eine benutzt die andere:

$$\begin{aligned} gerade(0) &= true \\ gerade(n) &= ungerade(n-1) \\ ungerade(0) &= false \\ ungerade(n) &= gerade(n-1) \end{aligned}$$

Rekursive Funktionen in Java

Alle modernen höheren Programmiersprachen unterstützen das Konzept der rekursiven Funktionsdefinitionen. Die Fakultätsfunktion kann darum quasi direkt in Java–Code übersetzt werden:

```
static int fak(final int n) {
  if ( n == 0 ) return 1;
  else       return fak(n-1)*x;
}
```

Entsprechendes gilt für die beiden indirekt rekursiven Funktionen *gerade* und *ungerade*:

```
static boolean gerade(final int n) {
  if ( n == 0 ) return true;
  else return ungerade(n-1);
```

```
}
static boolean ungerade(final int n) {
  if ( n == 0 ) return false;
  else return gerade(n-1);
}
```

1.7.2 Rekursion und Iteration

Rekursion zur Bearbeitung rekursiver Definitionen

Der einfachste Einsatz rekursiver Funktionen ist die Umsetzung von rekursiven Definitionen in Funktionen. Am Beispiel der Berechnung der Fakultätsfunktion haben wir die Äquivalenz bereits gezeigt.

Ein weiteres Beispiel ist die Berechnung des Binomialkoeffizienten:

$$\binom{n}{k} = \begin{cases} 0 & n < k \\ 1 & k = 0 \; oder \; n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \end{cases}$$

Auch diese mathematische Funktionsdefinition kann leicht in eine Java–Funktion umgesetzt werden:

```
static int binomRekursiv(final int n, final int k) {
  if (n==k || k == 0) return 1;
  if (n < k)      return 0;
  return binomRekursiv (n-1, k-1) + binomRekursiv (n-1, k);
}
```

Rekursive Funktionen[36] können auch mit Hilfe von Schleifen realisiert werden. Die Vorteile einer iterativen Version, also einer die mit Schleifen arbeitet, sind höhere Geschwindigkeit und geringerer Speicherbedarf. Ihr Nachteil ist, dass sie einem nicht so schnell einfallen wie die rekursiven, die ja oft mehr oder weniger direkt aus einer mathematischen Definition übertragen werden können. Die rekursive Java–Funktion zur Berechnung eines Binoms oben ist ja nicht mehr als eine Umformulierung der mathematischen Definition. Mit etwas Übung ist sie in höchstens zwei Minuten hingeschrieben. Eine iterative (Schleifen–) Version von `binomRekursiv` wird aber nicht so schnell zu erstellen sein.

Schleifen und Rekursionen

In Schleifen wird eine Folge von Anweisungen immer wieder ausgeführt – so lange, bis das Ziel der Schleife erreicht ist. Mit einer rekursiven Funktion verhält es sich genauso, sie wird so lange (von sich selbst) aufgerufen, bis sie ihr Ziel erreicht hat.

Die enge Verwandtschaft von Schleifen und rekursiven Funktionen erkennt man daran, dass jede `while`–Schleife (und damit auch jede andere Schleife) in eine rekursive Funktion transformiert werden kann:

```
while (B) A;
```

ist äquivalent zu:

```
static void doWhile () { if (B) { A; doWhile (); } }
```

Beispiel: Iteration und Rekursion

Als Beispiel einer Umwandlung in die andere Richtung betrachten wir noch einmal die Fakultätsfunktion f. Der Ausgangspunkt ist eine rekursive mathematische Definition:

$$\begin{aligned} f(0) &= 1 \\ f(x) &= f(x-1) * x \end{aligned}$$

Eine solche Definition kann mechanisch und ohne nennenswerten Einsatz von Gehirnmasse in eine Java–Funktion umgesetzt werden (siehe oben). Eine iterative Lösung ist nicht ganz so leicht zu finden. Mit etwas Nachdenken kommt man aber sicher zu einer Lösung wie:

[36] mit “rekursive Funktion” meinen wir hier genau genommen “rekursiv definierte Funktionen”

```
static int f(final int x) {
  int a = 1;
  int i = 0;
  while (i < x) {
    ++i;
    a = a*i;
  }
  return a;
}
```

Im Gegensatz zur rekursiven Version sieht man dieser Variante nicht unmittelbar an, dass sie die Fakultätsfunktion berechnet. Bei der Umsetzung in eine Schleife hat also eine gewisse Transformation stattgefunden. Da es sich dabei um den kreativen Akt des Programmierens handelt, wollen wir im Folgenden etwas genauer betrachten, wie man von einer rekursiven Definition zu einer Schleife kommt. In vielen Fällen ist das möglich. Man kann aus der Rekursion eine Iteration machen.[37]

Rekursionsformel zur Summenbildung

Ein einfaches Beispiel für eine rekursive Definition ist die *Rekursionsformel*[38] der Summen–Funktion S:

$$\begin{aligned} S(0) &= 0 \\ S(n) &= S(n-1)+n \end{aligned}$$

Statt diese Definition einfach in eine entsprechende rekursive Java–Funktion umzusetzen, wollen wir einen iterativen Algorithmus entwickeln, der das Problem löst.

Die Interpretation der Definition von S ist klar:

$$S(3) = S(2)+3 = S(1)+2+3 = S(0)+1+2+3 = 0+1+2+3$$

Soll $S(x)$ für ein beliebiges x berechnet werden, dann muss die Reihe der Werte $S(0), S(1), ...S(x-1)$ jeweils berechnet werden. Alle Berechnungen folgen der Definition und damit dem gleichen Schema. Wiederholungen sind ein Fall für Schleifen. Das Problem wird also mit einer Schleife gelöst.

Rekursion und Iteration

Die Rekursion geht "rückwärts", "von oben" an den gesuchten Wert heran. $S(n)$ wird auf $S(n-1)$, dieses auf $S(n-2)$ etc. zurück geführt. Eine Schleife dagegen geht "von unten" vor: aus $S(0)$ wird $S(1)$, aus $S(1)$ wird $S(2)$ berechnet etc. bis zu $S(n)$.

Die Art der Wiederholung die mit Schleifen möglich ist, nennt man *Iteration*. Eine Iteration *ist* eine Schleife. Eine Rekursionsformel beschreibt ebenfalls ein Berechnungsverfahren, das auf Wiederholung beruht. Während die Iteration (Schleife) sich jedoch sozusagen "von vorn nach hinten" zum Ergebnis bewegt, geht die Rekursion "vom Ziel aus zurück zu den Anfängen".

Funktion S iterativ berechnen

Jede Berechnung von $S(x)$ für ein beliebiges (ganzzahliges) x (größer–gleich 0) beginnt mit $S(0)$ und geht dann über $S(1), S(2), \ldots S(x-1)$ zu $S(x)$. Als Wertverlaufstabelle:

s	i	x
$S(0) = 0$	0	x
$S(1) = 0+1$	1	x
$S(2) = 0+1+2$	2	x
$S(3) = 0+1+2+3$	3	x
..	...	...

Die Rekursionsformel definiert dabei den Übergang von einem Wert zum nächsten. Daraus kann sofort eine Schleife konstruiert werden:

[37] Man kann sogar in *in allen* Fällen die Rekursion auf Schleifen abbilden, aber manchmal benötigt man dann mehr als nur eine.
[38] "Rekursionsformel" ist einfach ein anderes Wort für rekursive Definition.

```
int s = 0;  // S(0)
int i = 0;
while (i != x) {
  i = i + 1;
  s = s + i;
}
```

Vergleich rekursiver und iterativer Berechnungen

Die rekursive Definition beschreibt, wie der gesuchte Wert "von oben nach unten und wieder zurück" berechnet wird: Man geht nach unten bis zur Basis und sammelt dann auf dem Rückweg die Werte auf. Betrachten wir die Berechnung von $S(3)$:

Analysieren		Aufsammeln	
$S(3) = S(2)+3$	$\downarrow 3$	$\uparrow 6$	4. Wert: $3+3$
$S(2) = S(1)+2$	$\downarrow 2$	$\uparrow 3$	3. Wert: $1+2$
$S(1) = S(0)+1$	$\downarrow 1$	$\uparrow 1$	2. Wert: $0+1$
$S(0) = 0$	$\rightarrow 0$	$\uparrow 0$	1. Wert: 0

Das Argument wird solange gemäß der Formel zerlegt, bis die Basis erreicht wird. Bei dieser Analyse muss man sich die Rechenschritte merken, die dann anschließend auszuführen sind. Das Verfahren hat also zwei Phasen: Analyse (Runtergehen) und Aufsammeln (Raufgehen).

Die iterative Berechnung hat dagegen nur eine Richtung und eine Phase; sie sammelt nur auf:

$S(0) = 0$	$\downarrow 0$	1. Wert: 0
$S(1) = S(0)+1$	$\downarrow 1$	2. Wert: $0+1$
$S(2) = S(1)+2$	$\downarrow 3$	3. Wert: $1+2$
$S(3) = S(2)+3$	$\downarrow 6$	4. Wert: $3+3$

Beispiel: Berechnung der Potenzbildung

Ein weiteres Beispiel für eine rekursiv definierte Funktion ist die Potenzbildung:

$$\begin{aligned} x^0 &= 1 \\ x^n &= x^{n-1} * x \end{aligned}$$

Die Potenz kann mit der gleichen Strategie wie die Funktion S von oben iterativ berechnet werden. Wir definieren eine Variable `p` und bilden in ihr die Reihe der Werte $p_0 = x^0$, $p_1 = x^1$, $\ldots$ $p^n = x^n$:

p	i	n
$p_0 = 1$	0	n
$p_1 = p_0 * x$	1	n
$p_2 = p_1 * x$	2	n
$p_3 = p_2 * x$	3	n
..	...	...
$p_n = p_{n-1} * x$	n	n

Die erste Zeile definiert die Initialwerte der Schleife, die letzte die Schleifenbedingung, und der Schleifenkörper entspricht dem Übergang von einer Zeile zur nächsten.

```
static int potenz(final int x, final int n) {
  i = 0;
  p = 1;
  while (i != n) { // p == x^i
    i = i + 1;
    p = p * x;
  }
  return p;
}
```

Beispiel: Berechnung des GGT

Der größte gemeinsame Teiler (ggt) von zwei ganzen positiven Zahlen lässt sich nach folgender Rekursionsformel berechnen:

$$ggt(a,b) = \begin{cases} a & a = b \\ ggt(a-b,b) & a > b \\ ggt(a,b-a) & b > a \end{cases}$$

Auch hierzu kann man eine Tabelle angeben. Beispielsweise für die Berechnung von $ggt(120,48)$. Wir spielen Computer und rechnen per Hand:

ggt(120, 48)	=	ggt(72, 48)
ggt(72, 48)	=	ggt(24, 48)
ggt(24, 48)	=	gg(24, 24)
ggt(24, 24)	=	24

Schreiben wir die Parameterwerte in einer Tabelle untereinander

a	b
120	48
72	48
24	48
24	24

dann sehen wir sofort, wie die Zeilen der Tabelle von oben nach unten berechnet werden können. `a` und `b` sind die Variablen, sie werden mit der Eingabe initialisiert und in einer Schleife durch Subtraktion des kleineren vom größeren so lange verkleinert, bis beide den gleichen Wert haben:

```
static int ggt(int a, int b) {
  // a und b enthalten positive ganze Zahlen a_0, b_0
  while (a != b) {
    if (a > b) a = a-b;
    if (b > a) b = b-a;
  }
  // a enthaelt ggt (a_0, b_0)
  return a;
}
```

1.7.3 Beispiel: Berechnung von e

e als Grenzwert

Die Eulersche Konstante e kann als Grenzwert definiert werden:

$$e = \lim_{n \to \infty} a_n$$

mit

$$a_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots \frac{1}{n!}$$

Mit dieser Formel kann der Wert von e mit beliebiger Genauigkeit angenähert werden. Nehmen wir an e sollte mit der Genauigkeit ε berechnet werden. Wir müssen dazu nur a_n für steigende n berechnen bis irgendwann einmal $a_n - a_{n-1} < \varepsilon$.

Programmentwurf

Ein erster Entwurf ist:

```
double eps = ...;
double a, // a_n
       d; // a_n - a_n-1
int    n;
```

```
? initialisierung ?
while (d > eps) {
  n = n + 1;
  ? neues a berechnen ?
  ? neues d berechnen ?
}
```

Rekursionsformel für a_n

Wie man aus der Definition oben sieht kann ein Summand a_n leicht aus a_{n-1} berechnet werden:

$$\begin{aligned} a_0 &= 1 \\ a_n &= a_{n-1} + \frac{1}{n!} \end{aligned}$$

Damit haben wir eine Rekursionsformel für a_n. Rekursionsformeln sind immer gut. Man kann sie in der Regel gut in einen Schleifenkörper umsetzen.

Rekursionsformel für $\frac{1}{n!}$

Die Berechnung von $\frac{1}{n!}$ ist aufwändig und muss für steigende n beständig wiederholt werden. Es stellt sich die Frage, ob $\frac{1}{n!}$ nicht einfacher aus $\frac{1}{(n-1)!}$ berechnet werden kann. Natürlich kann es:

$$\begin{aligned} q_0 &= \frac{1}{1!} = 1 \\ q_n &= \frac{1}{n!} = \frac{1}{(n-1)!} * \frac{1}{n} = q_{n-1} * \frac{1}{n} \end{aligned}$$

$$\begin{aligned} a_0 &= 1 \\ a_n &= a_{n-1} + q_n \end{aligned}$$

Damit haben wir eine zweite Rekursionsformel und können somit q_n aus q_{n-1} und dann a_n aus a_{n-1} berechnen.

Die Schleife

Zur Verdeutlichung des aktuellen Stands der Überlegungen zeigen wir einen Ausschnitt aus der Wertverlaufstabelle:

n	q	a	d	eps
0	1	1	???	ε
...	...	...	...	...
$n-1$	$q_{n-1} = \frac{1}{(n-1)!}$	a_{n-1}	$a_{n-1} - a_{n-2}$	ε
n	$q_n = q_{n-1} * \frac{1}{n}$	$a_n = a_{n-1} + q_n$	$a_n - a_{n-1}$	ε
...	...	...	...	...

Man erkennt unmittelbar, dass die Variable `d` überflüssig ist: ihr Wert entspricht exakt dem von `q`. Damit können wir die Schleife sofort aufschreiben und den Programmentwurf zu einer vollständigen Funktion erweitern:

```
static double e() {
    double eps = 0.00001;
    double a, q;
    int  n;

    q = 1; a = 1; n = 0;
    while (q > eps) {
      n = n + 1;
      q = q/n;
      a = a + q;
    }
    return a;
}
```

1.7.4 Die Schleifeninvariante

Schleifeninvariante: Zusicherung in eine Schleife

Weiter oben haben wir "Zusicherung" definiert, als Aussage über die aktuelle Belegung der Variablen, also über den aktuellen (Programm–) Zustand. Eine *Schleifeninvariante* (kurz *Invariante*) ist eine Zusicherung, die den Zustand am Anfang eines Schleifenkörpers beschreibt. Bei einer Zusicherung versucht man wichtige Informationen über die Variablenbelegung zum Ausdruck zu bringen. Was ist wichtig bei einer Schleife und was sollte darum in der Invariante zum Ausdruck gebracht werden? Erstaunlicherweise ist die Fachwelt sich sicher, dass die wichtige Information über eine Schleife über das Auskunft gibt, was sich *nicht ändert*! Daher hat sie auch ihren Namen: Die *Invariante* beschreibt was *invariant*, also unveränderlich ist. Kurz: Eine Invariante ist eine Zusicherung in einer Schleife, die invariante (unveränderliche) Eigenschaften der Variablenwerte beschreibt.

Beispiel: GGT

Als Beispiel betrachten wir noch einmal die Schleife zur Berechnung des GGT (siehe weiter oben).

```
// a und b enthalten positive ganze Zahlen a_0, b_0
while (a != b) {
  if (a > b) a = a-b;
  if (b > a) b = b-a;
}
// a enthaelt ggt (a_0, b_0)
```

Die Variablen `a` und `b` werden mit positiven ganzen Zahlen initialisiert und am Ende der Schleife enthält `a` den gesuchten GGT. Warum ist das so? Wir erkennen auf den ersten Blick die Dynamik der Schleife, also das was sich in ihr bewegt: der kleinere Wert wird vom größeren abgezogen. Ein entsprechender Kommentar ist völlig überflüssig:

```
// a und b enthalten positive ganze Zahlen a_0, b_0
while (a != b) {
  // (Ueberfluessiger Kommentar zur Programm-DYNAMIK:
  // Wer das im Kommentar ausgesagte nicht am Code erkennt, der
  // sollte sich einen Beruf suchen in dem man keinen Code lesen muss.)
  //
  // Der kleinere Wert wird vom groesseren abgezogen
  if (a > b) a = a-b;
  if (b > a) b = b-a;
}
// a enthaelt ggt (a_0, b_0)
```

Das Offensichtliche muss und soll nicht kommentiert werden! Ein sinnvoller Kommentar würde hier Informationen darüber geben, warum die Subtraktion des Kleineren vom Größeren letztlich zum GGT führt, warum also die Schleife – nach Meinung des Programmautors – funktioniert.

Die GGT–Berechnung der Schleife beruht auf folgender mathematischen Erkenntnis

$$ggt(x,y) = \begin{cases} x & : \quad x = y \\ ggt(x-y,y) & : \quad x > y \\ ggt(x,y-x) & : \quad y > x \end{cases}$$

Etwas einfacher ausgedrückt: der GGT ändert sich nicht, wenn man von der größeren die kleinere Zahl abzieht. Damit haben wir schon das identifiziert, was sich *nicht ändert*: der GGT von `a` und `b`, egal welchen Wert die beiden Variablen gerade haben.

In der Schleife ändert sich der GGT von `a` und `b` nicht, obwohl die Werte von `a` und `b` bei jedem Durchlauf verändert werden.

```
// a und b enthalten positive ganze Zahlen a_0, b_0
while (a != b) {
  // (Sinnvoller Kommentar zu Programm-STATIK, die INVARIANTE:)
  // INV: ggt (a, b) == ggt (a_0, b_0)
  if (a > b) a = a-b;
  if (b > a) b = b-a;
}
// a == b UND ggt(a, b) == ggt(a_0, b_0) => a == ggt(a_0, b_0)
```

Beispiel: Ganzzahlige Division

Die Invariante hängt eng mit dem zugrunde liegenden Algorithmus zusammen, sie bringt "die Idee" der Schleife zum Ausdruck und sollte darum schon beim Entwurf der Schleife bedacht werden.

Betrachten wir eine Funktion zur ganzzahligen Division mit Rest. Ein Beispiel für eine solche Division ist:

$$15 : 4 = 3 \, Rest \, 3$$

Ein einfacher Algorithmus ist: Beginne mit einem Rest von 15 (in der Variablen `rest`) und ziehe solange 4 vom Rest ab, bis dieser kleiner als 4 ist. Zähle dabei in einer Variablen `faktor` die Zahl der Subtraktionen. Als Wertverlaufstabelle:

rest	faktor
15	0
11	1
7	2
3	3

Warum ist dieser Algorithmus korrekt? Nun, ganz einfach: Man beginnt mit einem zu großen aber ansonsten korrektem Rest von 15 und erniedrigt ihn bis er kleiner als 4 ist. Gleichzeitig erhöht man den Faktor. Die Beziehung $Rest + Faktor * 4 = 15$ bleibt dabei erhalten.

Rest	+	*Faktor* * 4	=	15
15	+	0 * 4	=	15
11	+	1 * 4	=	15
7	+	2 * 4	=	15
3	+	3 * 4	=	15

Als Schleife mit Invariante:

```
// Berechne die ganzzahlige Division
// a : b  ( a >= 0, b > 0 )
faktor = 0;
rest = a;
while (rest >= b) { //INV a == faktor * b + rest
  rest = rest - b;
  faktor = faktor + 1;
}
// a == faktor * b + rest UND !(rest >= b)
```

Die Schleifeninvariante gilt bei Eintritt in die Schleife und am Schleifenende. Am Schleifenende ist außerdem die Schleifenbedingung nicht mehr erfüllt – die Schleife wäre ja sonst nicht verlassen worden. Beides zusammen ergibt das, was wir wollten: $a = Faktor * b + Rest \wedge Rest < b$.

1.7.5 Schrittweise Verfeinerung

Geschachtelte Schleifen

Von geschachtelten Schleifen spricht man, wenn eine Schleife in einer Schleife auftritt. Ein Beispiel ist:

```
static void main (String[] args) {
  int n;
  do {
    System.out.println("naechster Wert, Stop mit 0: ");
    n = readInt();
    if (n > 0) {
      int s = 0;
      for (int i = 1; i <= n; ++i)
        s = s + i;
      System.out.println("Summe i = (1 .. " + n + ") = " + s);
    }
  } while ( n > 0);
}
```

Bei komplexeren Programmen – und geschachtelte Schleifen bringen immer eine gewisse Komplexität mit sich – ist es wichtig die Grobstruktur des Programms zu kennen. Das Beispiel beinhaltet eine äußere Do–While–Schleife und eine innere For–Schleife. Die äußere Schleife gehört zum Programmrahmen, mit dem Zahlen solange eingelesen werden, bis 0 oder eine negative Zahl auftaucht:

```
static void main (String[] args) {
  int n;
  do {
    System.out.println("naechster Wert, Stop mit 0: ");
    ... Verarbeitung von n ...
  } while ( n > 0);
}
```

Die innere Schleife gehört zu dem Programmteil, in dem die eingelesenen Werte von `n` verarbeitet werden:

```
if (n > 0) {
  int s = 0;
  for (int i = 1; i <= n; ++i)
    s = s + i;
  System.out.println("Summe i = (1 .. " + n + ") = " + s);
}
```

Hier werden einfach alle Zahlen von 1 bis `n` aufsummiert und die Summe dann ausgegeben.

Programme haben also eine hierarchische Struktur. Sie bestehen aus ineinander geschachtelten Teilprogrammen, die jeweils eine bestimmte Aufgabe erfüllen und dazu andere untergeordnete Teilprogramme in Anspruch nehmen. Diese Struktur zeigt sich nicht erst bei der Analyse eines fertigen Programms. Beim Entwurf eines neuen Programms empfiehlt es sich Teilprogramme zu identifizieren und ihnen jeweils einen Bearbeiter in Form eines Teilprogramms zuzuweisen.

Die groben Konzepte (Programmrahmen) werden zu feineren Strukturen ausgearbeitet, die dann in der nächsten Stufe selbst die groberen Konzepte darstellen. Diese Art des Vorgehens bei der Programmentwicklung nennt man *schrittweise Verfeinerung*.

1.7.6 Funktionen und schrittweise Verfeinerung

Primfaktorzerlegung

Am Beispiel des Problems der Primfaktorzerlegung wollen wir hier Funktionen als Mittel zur schrittweisen Verfeinerung eines Programms betrachten. Bei der Primfaktorzerlegung wird eine positive ganze Zahl als Produkt von Primzahlen dargestellt. Die Primfaktorzerlegung ist immer eindeutig. Beispiel:

$$60 = 2 * 2 * 3 * 5$$

Ein erster einfacher Ansatz (erster Verfeinerungsschritt) ist:

```
// Hilfsfunktionen, sie loesen Teilprobleme ueber deren Loesung
// ich mir spaeter Gedanken mache:
//
boolean teilt (final int n, final int t){...} //wird n von t geteilt
boolean prim (final int n){...}     //ist n eine Primzahl
int   exponent (final int n, final int p){...} //bestimmt Exponent des Primfaktors p von n

// Programmrahmen, Grobkonzept
//
public static void main (String args[]){
  int n;      // zu zerlegende Zahl
  int count = 0; // Zahl der Primfaktoren

  n = readInt();

  System.out.println("Die Primfaktorzerlegung von " + n + " :");

  // Jeder Teiler
  for (int i=2; i<n; ++i){
    if (teilt (n, i) && prim (i)) {
      System.out.println(i + " hoch " + exponent(n, i));
      count++;
    }
  }
```

```
  if (count == 0)
    System.out.println(n + " ist eine Primzahl");
 }
```

Hier wird einfach jede Zahl, die kleiner als die Eingabe n ist, daraufhin untersucht, ob sie ein Teiler von n und dazu noch prim ist. So erhält man alle Primfaktoren. Die Funktion `exponent` stellt noch fest, mit welchem Exponent ein gefundener Primfaktor in n auftritt.

Hilfsfunktionen: aufgeschobene Teilprobleme

Mit diesem ersten Ansatz wurde das Problem der Primfaktorzerlegung auf drei einfachere Teilprobleme reduziert deren Lösung an – noch ungeschriebene – Funktionen delegiert wurde:

- `teilt`: Feststellen ob eine Zahl ein Teiler einer anderen ist.
- `prim`: Feststellen, ob eine Zahl eine Primzahl ist.
- `exponent`: Unter der Voraussetzung, dass p ein Teiler von n ist, soll der größte Exponent von p gefunden werden der n teilt; d.h. die Funktion liefert das größte x mit: p^x teilt n.

Die Kunst der Programmierung besteht in der Beherrschung von Komplexität. Die erste und wichtigste Regel, um Komplexität in den Griff zu bekommen ist:

> Versuche nicht alle Probleme gleichzeitig zu lösen! Gehe immer schön eins nach dem anderen an.

Um diese Regel befolgen zu können, muss das Gesamtproblem in Teilprobleme zerlegt werden. Das haben wir hier mit einer Aufteilung im Rahmenprogramm und Hilfsfunktionen getan.

Die Funktion `teilt`

Das Rahmenprogramm steht. Wir wenden uns jetzt `teilt` zu, der einfachsten Hilfsfunktion:

```
 static boolean teilt (final int n, final int t) {
   /* Wird n von t geteilt */
   if (n % t == 0)
     return true;
   else return false;
 }
```

oder einfacher:

```
 static boolean teilt (final int n, final int t) {
   /* Wird n von t geteilt */
   return (n % t == 0);
 }
```

t ist ein Teiler von n, wenn gilt $n \, modulo \, t = 0$.

Die Funktion `prim`

Eine natürliche Zahl ist eine Primzahl, wenn sie keine Teiler außer 1 und sich selbst hat. Diese Definition wird einfach in eine Schleife umgesetzt:

```
 static boolean prim (final int n) {
   // ist n prim?
   if (n == 2) return true; // 2 ist eine Primzahl
   for (int i = 2; i<n; ++i) { // hat n einen Teiler?
     if (teilt (n, i)) return false;
   }
   return true;
 }
```

Die Funktion `exponent`

Es bleibt nur noch die Funktion `exponent`, die den Exponenten eines Primfaktors bestimmt. Auch hier gehen wir so einfach wie möglich vor. Gesucht ist das größte x mit p^x *teilt* n. Wir suchen systematisch danach:

```
static int exponent (final int n, final int p) {
  // liefert groesstes x mit p hoch x teilt n
  // falls p ein Teiler von n ist.
  int i = 0,
     pp = 1; /* pp == p**i */

  while (teilt (n, pp)) {
    ++i;
    pp *= p;
  }
  return i-1;
}
```

Aus diesen Bestandteilen kann jetzt das Gesamtprogramm zusammengesetzt werden.

Schrittweise Verfeinerung: Funktionale Programmstruktur

Die schrittweise Verfeinerung ist eine Entwicklungsmethode für Programme. Ihr Ziel ist die Zerlegung einer Gesamtaufgabe in Teilprobleme, die sukzessiv und unabhängig voneinander bearbeitet werden können. Man geht dabei davon aus, dass die zu lösende Problemstellung in eine Folge von Unteraufgaben aufgegliedert werden kann. Jede der Unteraufgaben wird dann in Unter–Unteraufgaben zergliedert. Mit diesem Prozess der Zergliederung – oder "Verfeinerung" – fährt man so lange fort, bis die gesamte Aufgabenstellung in einfache und elementare Basisaufgaben zerlegt ist. Funktionen erhöhen die Übersichtlichkeit einer Verfeinerung, wenn sie zur Lösung der Unter– und Unter–Unteraufgaben eingesetzt werden.

Ein Programmentwurf entsprechend der schrittweisen Verfeinerung führt zu einer funktionalen Programmstruktur: Das Gesamtprogramm besteht aus einer Serie von Funktionen. Die Main–Funktion repräsentiert den ersten Verfeinerungsschritt, die von `main` direkt aufgerufenen Funktionen stellen den zweiten Schritt der Verfeinerung dar. Sie rufen selbst wieder Unterfunktionen auf, die die nächste Stufe der Verfeinerung darstellen, und so weiter.

1.7.7 Programmtest

Testfall

Auch sorgfältig erstellte Programme enthalten gelegentlich Fehler. Sie müssen darum getestet werden. Bei einem Test wird experimentell festgestellt, ob das Programm die gewünschte Ausgabe liefert. Dazu werden Testfälle definiert. Ein *Testfall* besteht aus einer Programm– oder Funktionseingabe und der daraufhin erwarteten Aus– oder Rückgabe. Testfälle für das Programm aus dem letzten Abschnitt bestehen beispielsweise aus Folgen von Eingaben, die mit Null enden und den daraufhin erzeugten Listen von Teilern. Z.B. ("/" steht für einen Zeilenvorschub):

Testfall 1:
Eingabe: `5 6 0`
Ausgabe: `5 ist eine Primzahl / 2 / 3 sind die Teiler von 6`

Testfall 2:
Eingabe: `-3`
Ausgabe: `-3 ist eine Primzahl`

Testfall 3:
Eingabe: `111`
Ausgabe: `3 hoch 1 / 37 hoch 1`

Eine Kollektion von Testfällen bildet eine *Testsuite*. Die Testfälle einer Testsuite sollten natürlich so umfangreich sein, dass man ein gewisses Vertrauen in das Programm gewinnt. Sie sollten auch nach gewissen Kriterien zusammengestellt sein, damit keine wichtigen Fälle außer Acht gelassen werden. Beispielsweise kann man die Eingabe "normaler", "extremer" und "falscher" Eingaben in Betracht ziehen. In unserem Beispiel der Primzahlzerlegung wären das etwa Primzahlen und Zahlen die keine Primzahlen sind, kleine Werte und große Werte, die Null als erste Eingabe und die Eingabe von negativen Werten.

Testen mit JUnit

Eclipse enthalt das Testwerkzeug *JUnit* und bietet damit eine bequeme Möglichkeit des Testens. Angenommen, wir wollten die Funktion `prim` in einer Klasse `Zahlen` testen. Als erstes muss `prim` mit `public` zu einer öffentlichen Methode gemacht werden:

```
package zahlen;

public final class Zahlen {

   private Zahlen(){}

   public static void main (String args[]){...}

   ...

   public static boolean prim (final int n) { // zu testende Methode: public!
     // ist n prim?
     if (n == 2) return true; // 2 ist eine Primzahl
     for (int i = 2; i<n; ++i) { // hat n einen Teiler?
       if (teilt (n, i)) return false;
     }
     return true;
   }

   static boolean teilt (fianl int n, final int t) {
     return (n % t == 0);
  }
  ....
}
```

Jetzt rufen wir in Eclipse das per Rechtsklick auf die zu testenden Klasse *Zahlen* das Menü

New / JUnit Test Case

auf, wählen dort

New JUnit 4 test

aus. Weiter mit

next

und Eclipse bietet uns eine Auswahl der Methoden an. Wir wählen `prim` aus und eine Klasse `ZahlenTest` wird erzeugt:

```
package zahlen;

import static org.junit.Assert.*;
import org.junit.Test;

public class ZahlenTest {

      @Test
      public void testPrim() {
            fail("Not yet implemented");
      }

}
```

Wir ergänzen die Klassendefinition um **final** und tragen ein paar Tests ein: Hier tragen wir einige Tests ein. Alle Methoden, die mit `@Test` anotiert sind, werden automatisch als Testfälle erkannt:

```
package myPackage;

import static org.junit.Assert.*;
import org.junit.Test;

public final class ZahlenTest {

      @Test
      public void testPrim() {
```

```
            assertTrue(Zahlen.prim(2));
            assertTrue(Zahlen.prim(3));
            assertTrue(! Zahlen.prim(4));
            assertTrue(Zahlen.prim(17));
            assertTrue(! Zahlen.prim(256));
            assertTrue(! Zahlen.prim(14));
            assertTrue(! Zahlen.prim(1025));
        }

}
```

Wir starten den Test und sehen dass er gelingt.

Fehlerstellen identifizieren

Im günstigen Fall besteht das Programm die Testfälle, das heißt zu jeder Eingabe wird die im voraus bestimmte Ausgabe erzeugt. Leider ist der günstige Fall nicht der Normalfall. Auch systematisch entwickelte Programme sind nicht unbedingt fehlerfrei. Es erleichtert allerdings das Testen und Auffinden von eventuellen Fehlern ganz erheblich, wenn man sich etwas bei seinem Quellcode gedacht hat und sich beim Testen auch noch daran erinnert – eventuell mit Hilfe von sinnvollen Kommentaren![39]

Liefert das Gesamtprogramm nicht das erwartete Ergebnis, dann müssen wir die Fehlerstelle identifizieren, also herausfinden welcher Teilschritt nicht wie geplant funktioniert. Es ist dabei sehr hilfreich, wenn das Programm in klare Teilschritte aufgegliedert wurde. Mit Hilfsausgaben kann dann geprüft werden, was korrekt funktioniert und was nicht. Man stattet das Programm dazu mit Testausgaben aus. D.h.. man lässt sich an kritischen Stellen im Programm die Werte wichtiger Variablen ausgeben, um verifizieren zu können, ob sie die erwarteten Werte haben. Wichtig ist beispielsweise oft der Beginn eines Schleifenkörpers. Hier kann man sich davon überzeugen, ob die Invariante tatsächlich gültig ist – ob die Schleife also mit den richtigen Werten betreten wird. Am Beginn einer Funktion kann man sich davon überzeugen, ob die Parameter die erwarteten Werte haben und an deren Ende, ob sie das Erwartete berechnet hat, etc.

Debugger

Fehlerstellen im Programm werden auch *Bugs* (Wanzen, Schädlinge) genannt. Ihr Auffinden und Beseitigen nennt man *Debugging* (Entwanzen) oder auch eingedeutscht *Debuggen*. Dabei hilft oft ein Hilfs–Programm namens *Debugger*. Ein Debugger kann natürlich keine Fehlerstellen finden oder gar beseitigen, das bleibt Aufgabe der Programmierer. Der Debugger erlaubt es, das Programm Schritt für Schritt durchzugehen und dabei die Werte aller Variablen anzusehen. Man kann sich damit die Hilfsausgaben ersparen. Stattdessen lässt man den Debugger das Programm an definierten Stellen unterbrechen und prüft, ob die Variablen den an diesem Punkt erwarteten Wert haben.

Debugger sind sehr nützliche Hilfsmittel der Programmentwicklung. Jeder Entwickler sollte sich mit ihrem Gebrauch vertraut machen. Sie ersetzen nicht die systematische Programmentwicklung, sie ergänzen sie. Eclipse bietet einen integrierten Debugger, mit dem man sich leicht vertraut macht.

Wenn das alles aber nicht hilft und das Programm sich hartnäckig weigert die gewünschten Ergebnisse auszuspucken, dann gibt es nur eins: Es muss von neuem nachgedacht werden. Der beste Debugger sitzt nun mal zwischen den Ohren.

[39] Wenig erfahrene Programmierer überschätzen regelmäßig die Halbwertszeit ihrer Gedächnisinhalte. Vollkommen klare eigene (!) Programme verwandeln sich oft schon innerhalb weniger Tage in völlig abstruses Codegewirr ohne dass irgend etwas an ihnen geändert wurde.

1.8 Felder

1.8.1 Felder definieren und verwenden

Definition eines Felds

Strukturierte Werte sind Werte, die aus mehreren Teilwerten zusammengesetzt sind. Aus der Mathematik sind Vektoren und Matrizen als strukturierte Werte bekannt. Ein Vektor beispielsweise ist strukturiert, weil er aus Teilwerten – seinen Komponenten – besteht. Der Vektor $\vec{a}$ mit

$$\vec{a} = \begin{pmatrix} 5 \\ 2 \end{pmatrix}$$

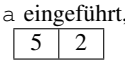
hat beispielsweise die Komponenten $a_1 = 5$ und $a_2 = 2$.

In Java kann ebenfalls mit Kollektionen von Werten gearbeitet werden. Eine von vielen Möglichkeiten dazu ist die Verwendung eines *Felds* (engl. *Array*). Felder sind die klassische Form um mit Wertkollektionen zu arbeiten. Sie sind seit etwa 50 Jahren Bestandteil höherer Programmiersprachen. Heutzutage bieten Java und andere moderne Sprachen Konstrukte an, die gelegentlich besser sind. Beginnen wir aber mit dem guten alten Feld. Wer, wie die Felder, 50 Jahre in der IT–Branche überlebt hat und dabei frisch geblieben ist, wird es sicher noch weitere 50 Jahre machen.

Ein Feld `a` mit zwei Int–Werten wird mit der Variablendefinition

```
int[] a = new int[2];
```

erzeugt.[40] Diese Definition erzeugt das Feld a  mit zwei Komponenten. `a` ist der *Feldname*, `int` der Typ der Komponenten und `2` die Größe (Zahl der Elemente).

Mit dieser Definition wird eine Variable `a` eingeführt, die dann im weiteren Programm genutzt werden kann. Beispielsweise um den Vektor von oben in der Form a  zu speichern.

Zugriff auf Feldelemente

Die Komponenten des Felds `a` können in Zuweisungen belegt werden und in Ausdrücken kann ihr Wert festgestellt werden. Dazu greift man mit einem *Index* auf die Feldelemente zu. Die erste Komponente hat den *Index* `0`, die zweite hat den Index `1` und so weiter. Der letzte Index ist die Feldgröße - 1. Beispiele:

```
a[0] = 5;          // Zuweisung an die erste Komponente von a
a[1] = 3;          // Zuweisung an die zweite Komponente von a
```

Solche indizierten Ausdrücke können auch in komplexerer Form verwendet werden:

```
a[1] = a[0]-3;    // Zuweisung an zweite Komponente von a
```

Hier wird die erste Komponente (Index `0`) auf 5 und die zweite (Index `1`) auf 2 ($= 5 - 3$) gesetzt:

a: 

`a[0]` ist die erste und `a[1]` die *zweite* Komponente von `a`. Ein Programmbeispiel mit Feldern ist:

```
  // Definition vo zwei Feldern
  int[] a = new int[2]; // a hat 2 Komponenten
  int[] b = new int[3]; // b hat 3 Komponenten

  a[0] = 1;
  a[1] = 2;
  b[0] = a[0] + 1; // b[0] hat jetzt den Wert 2
  b[1] = a[0] + a[1]; // b[1] hat jetzt den Wert 3
  b[2] = b[0] + b[1]; // b[2] hat jetzt den Wert 5

  // b mit Schleife belegen (b[0]=0, b[1]=1, b[2]=2)
  for ( int i=0; i<3; i++)
    b[i] = i;
}
```

Ein häufiger Fehler besteht darin, dass über die Feldgrenze hinausgegriffen wird:

[40] Es gibt (für C–Programmierer) noch eine alternative Version: `int a[] = new int[2];`. Wir ignorieren sie im Folgenden.

```
int [] = new int[10];
...
a[10] = .... // FEHLER: a[10] ist nicht mehr in a!
```

Achtung, der Compiler bemerkt solche Fehler nicht; sie treten zur Laufzeit auf!

Feldinitialisierung

Felder können bei der Definition gleich mit Werten belegt werden:

```
int[] a = {0,9,1,6,4,6,8,2,7,6};
```

Derartige Feldinitialisierungen sind eine bequeme Methode um Felder mit Werten zu belegen. Sie sind aber nur bei der Definition erlaubt, nicht in beliebigen Zuweisungen.

Felder sind strukturierte Variablen

Feldkomponenten können wie ganz normale Variablen verwendet werden – es sind ganz normale Variablen. Bei Feldern als Ganzes muss man dagegen etwas vorsichtig sein. Das Feld selbst ist keine ganz normale Variable. Felder sind *strukturierte Variablen*. Die wichtigste Konsequenz dieser Tatsache ist, dass Felder nicht als ganzes zugewiesen oder verglichen werden können. Will man also zwei Felder `a` und `b` vergleichen, dann müssen alle Elemente von `a` und `b` jeweils extra verglichen werden. Tun wir das nicht, dann ergeben sich eventuell Überraschungen. Beispielsweise gibt das Programmstück:

```
public static void main(String args[]) {
  int[] a = new int[2];
  int[] b = new int[2];
  a[0] = 1; a[1] = 1;
  b[0] = a[0]; b[1] = a[1];
  if ( a == b ) // sind a und b die selben Felder?
    System.out.println("a == b");
  else
    System.out.println("a != b"); // wird ausgegeben
}
```

den Text `a != b` aus, obwohl die beiden Felder den gleichen Inhalt haben! Es will damit sagen, dass `a` und `b` *nicht identisch* sind. Sie sind zwar gleich, aber als Individuen zu unterscheiden. Hier begegnet uns zum ersten Mal die Tatsache, dass Java Gleichheit und Identität unterscheidet und == mal das Eine und mal das Andere testet. Später werden wir diese Dinge genauer betrachten. Wir merken uns erst einmal, dass == bei Feldern ein Test auf Identität ist.

Beispiel, Programm mit einem Feld

Das folgende Programm gibt die geraden Zahlen von 0 bis 18 aus.

```
public static void main (String[] args) {
  int[] a = new int[10];

  for (int i=0; i<10; i++){
    a[i] = 2*i;
  }
  for (int i=0; i<10; i++){
    System.out.print(a[i]+" ");
  }
  System.out.println();
}
```

Das Feld wird hier zweimal mit einer For–Schleife durchlaufen. In der ersten Schleife wird es mit Werten belegt. In der zweiten werden diese ausgegeben. Die beiden Schleifen sind typisch für die Bearbeitung eines Feldes.

Indizierter Ausdruck

Ein *indizierter Ausdruck* ist ein Ausdruck in dem ein Index vorkommt. Die Zuweisung

```
a[2] = i + b[i];
```

enthält zwei indizierte Ausdrücke `a[2]` und `b[i]`. Mit ihr soll der dritten Komponente von `a` die Summe des aktuellen Wertes von `i` und des aktuellen Wertes der i–ten Komponente von `b` zugewiesen werden. (Genau genommen ist es der Wert der (Wert–von–`i`)–ten Komponente von `b`.)

Auswertung indizierter Ausdrücke

Indizierte Ausdrücke werden stets von innen nach außen ausgewertet.

- Beispiel `a[i]` :
  - Zuerst wird der Wert von `i` bestimmt, angenommen etwa zu i,
  - dann wird `a[` i `]` bestimmt: das Ergebnis ist eine Teilvariable (ein Element) von `a`
  - Findet sich der Ausdruck links vom Zuweisungszeichen, dann ist die Auswertung beendet.
  - Rechts vom Zuweisungszeichen geht es noch einen Schritt weiter mit der Bestimmung des aktuellen Wertes von `a[`i`]`.
- Beispiel `a[i]+i` :
  Dieser Ausdruck kann nur rechts vom Zuweisungszeichen erscheinen. Als erstes wird der Wert von `i` bestimmt (z.B. i). Damit wird dann `a[`i`]` identifiziert und der Wert dieser Teilvariablen von `a` bestimmt (z.B. k). Der Wert des Gesamtausdrucks ist dann $k+i$.
- Beispiel `a[i+i]` :
  Dieser Ausdruck kann links und rechts vom Zuweisungszeichen erscheinen.
  Links bezeichnet er die Teilvariable (das Element) `a[`k`]`, dabei sei $k = i+i$ und i der Wert von `i`, rechts bezeichnet er den Wert, den diese Teilvariable aktuell hat.

Beispiele

Wir wollen einige Beispiele für Zuweisungen und Ausdrücke mit indizierten Variablen (Variablen mit einem Index) betrachten:

Angenommen `i` habe den Wert 5 und `a[`i`]` für $i = 0..9$, den Wert $i+1$ *modulo* 10. Also:

i: | 5 |

a:

1	2	3	4	5	6	7	8	9	0

Jetzt werden einige Anweisungen nacheinander ausgeführt. Die Variablen `a` und `i` durchlaufen dann die entsprechenden Werte (wir geben jeweils zuerst die Anweisung und danach die Variablenbelegung an, die diese Anweisung erzeugt):

1. Ausgangssituation:
   i: | 5 |
   a:

1	2	3	4	5	6	7	8	9	0
2. `i = a[1];`
   i: | 2 |
   a:

1	2	3	4	5	6	7	8	9	0
3. `i = a[i];`
   i: | 3 |
   a:

1	2	3	4	5	6	7	8	9	0
4. `i = a[i]+1;`
   i: | 5 |
   a:

1	2	3	4	5	6	7	8	9	0
5. `i = a[i+1];`
   i: | 7 |
   a:

1	2	3	4	5	6	7	8	9	0
6. `a[i] = a[a[i]+1];`
   i: | 7 |
   a:

1	2	3	4	5	6	7	0	9	0

7. `a[a[i]+1] = a[a[i+1]-1];`

   i: | 7 |

   a:

1	9	3	4	5	6	7	0	9	0

8. `i = a[a[i]] + a[a[i]+7] + a[a[i-1]];`

   i: | 1 |

   a:

1	9	3	4	5	6	7	0	9	0

(Bitte nachvollziehen!)

Beispiel: Fakultät

Mit folgendem Programmcode kann die Fakultätsfunktion tabelliert werden:

```
int n = 10;
int[] f = new int[n]; // f[i] soll i! enthalten
int  i;

f[0] = 1; i = 0;
while ( i < n-1 ) {
   i = i+1;
   f[i] = f[i-1]*i;
}
```

Das Programmstück berechnet in den Variablen `f[i]` für ansteigendes i den Wert $i!$. Die While–Schleife kann durch eine äquivalente For–Schleife ersetzt werden:

```
...
  f[0] = 1;
  for (int i = 1; i < n; i++)
    f[i] = f[i-1]*i;
...
```

1.8.2 Suche in einem Feld

Suche in einem Feld

Ein bestimmter Wert s kann in einem Feld recht einfach gesucht werden. Man vergleicht einfach jedes Feldelement mit s. Nehmen wir an, in einem Feld `a` der Größe `n` soll der Wert s (in `s`) gesucht werden. Bei dieser Suche interessieren wir uns für den Index. D.h. es soll jeder Index i ausgegeben werden für den `a[`i`]` $= s$ ist. Das Feld wird mit der Standard–For–Schleife durchlaufen:

```
 for (int i = 0; i < n; i++) // Fuer jeden Index i
  if (a[i] == s)      // Vergleiche a[i] mit dem Wert von s
    System.out.println(i); // und gib i eventuell aus
```

Im folgenden Beispiel soll das größte Element in einem Feld `a` der Größe n gesucht werden. Die Strategie zur Suche besteht darin, dass ein immer größerer Bereich des Feldes untersucht wird und in einer Variablen `g` stets der bisher gefundene größte Wert zu finden ist:

g: | 6 | a: | 6 | 3 | 1 | 5 | 7 | 8 | 0 | 9 | 5 | 2 |

g: | 6 | a: | 6 | 3 | 1 | 5 | 7 | 8 | 0 | 9 | 5 | 2 |

...

g: | 6 | a: | 6 | 3 | 1 | 5 | 7 | 8 | 0 | 9 | 5 | 2 |

g: | 7 | a: | 6 | 3 | 1 | 5 | 7 | 8 | 0 | 9 | 5 | 2 |

g: | 8 | a: | 6 | 3 | 1 | 5 | 7 | 8 | 0 | 9 | 5 | 2 |

...

g: | 9 | a: | 6 | 3 | 1 | 5 | 7 | 8 | 0 | 9 | 5 | 2 |

Mit einer Schleife

```
int g = a[0];
for (int i=1; i<10; i++) // g enthaelt den bisher groessten Wert
  if ( a[i] > g )
    g = a[i];
```

kann diese Strategie realisiert werden. Die Laufvariable `i` gibt dabei die Grenze zwischen dem bereits untersuchten und dem noch zu untersuchenden Bereich von `a` an. `a[0...i-1]` ist untersucht. `a[i...n]` muss noch untersucht werden.

Da aber nicht der Wert, sondern der Index des größten gesucht ist, ändern wir die Schleife zu:

```
int gi = 0;
for (int i=1; i<10; i++) // a[gi] enthaelt den bisher groessten Wert
  if (a[i] > a[gi])
    gi = i;
```

als Gesamtprogramm:

```
public static void main (String[] args) {
  int  n = 10;
  int[] a = {0,9,1,6,4,6,8,2,7,6};

  int gi = 0;
  for (int i=1; i<n; ++i)
    if (a[i] > a[gi])
      gi = i;

  System.out.println("Der Index des groessten Wertes "
          + a[gi] + " ist " + gi);
}
```

1.8.3 Sortieren

Eine Sortierstrategie

Ein Feld soll aufsteigend sortiert werden. Wir verwenden die gleiche Strategie wie bei der Suche. Ein Anfangsbereich des Feldes ist sortiert und enthält alle kleinsten Elemente des Feldes. Dieser Bereich wird systematisch vergrößert. Am Anfang enthält der Bereich kein Element und am Ende alle Feldelemente. Die Vergrößerung des Bereichs besteht darin, dass das kleinste Element im unbearbeiteten Teil gesucht und mit dem neuen vertauscht wird:

Anfang:	a:	6	3	1	5	7	8	0	9	5	2
6 und 0 vertauscht:	a:	0	3	1	5	7	8	6	9	5	2
1 und 3 vertauscht:	a:	0	1	3	5	7	8	6	9	5	2
2 und 3 vertauscht:	a:	0	1	2	5	7	8	6	9	5	3
5 und 3 vertauscht:	a:	0	1	2	3	7	8	6	9	5	5
7 und 5 vertauscht:	a:	0	1	2	3	5	8	6	9	7	5
8 und 5 vertauscht:	a:	0	1	2	3	5	5	6	9	7	8
nichts vertauscht:	a:	0	1	2	3	5	5	6	9	7	8
9 und 7 vertauscht:	a:	0	1	2	3	5	5	6	7	9	8
9 und 8 vertauscht:	a:	0	1	2	3	5	5	6	7	8	9
fertig:	a:	0	1	2	3	5	5	6	7	8	9

Der Schleifenkörper

Für die Vergrößerung des bearbeiteten Teils des Feldes ist eine Schleife verantwortlich. Jeder Durchlauf (Betreten des Schleifenkörpers) hat eine Aufgabe und operiert unter bestimmten Voraussetzungen. Die Aufgabe besteht auch darin die Voraussetzungen für den nächsten Durchlauf zu schaffen:

- Voraussetzung: Feld ist bis i sortiert: `a[0..i-1]` ist sortiert und enthält die i–kleinsten Elemente des Feldes.
- Teilaufgabe eines Schleifendurchlaufs: i um eins erhöhen und dabei die Eigenschaft erhalten, dass `a[0..i-1]` sortiert ist und die i–kleinsten Elemente enthält.
- Lösung der Teilaufgabe: Im unsortierten Teil das kleinste Element suchen und mit dem letzten (neuen) im sortierten Bereich vertauschen!

Entwurf der Schleife:

```
for (int i=0; i<n; i++) {

  // a[0..i-1] ist aufsteigend sortiert und enthaelt kein
  // Element das groesser ist als irgend eines aus a[i .. n-1]

  m = Index des kleinsten in a[i .. n-1]
  tausche Inhalt von a[i] und a[m]
}
```

Die Suche nach dem Kleinsten kann mit dem Algorithmus aus dem letzten Abschnitt erfolgen. Das Vertauschen ist trivial:

```
t   = a[i];
a[i] = a[m];
a[m] = t;
```

Das Sortierprogramm

Das Programm insgesamt kann aus diesen Komponenten leicht zusammengesetzt werden:

```
public final class Sorter {

 private Sorter() {}

 public static void main (String[] args) {
    int[] a = {0,9,1,6,4,6,8,2,7,6};
    sortiere(a);
    drucke(a);
 }

 // Minimum im Feld ab Position i
 static int minAb(final int[] a, final int i) {
   int n = a.length;
   int m = i;
   for (int j = i + 1; j < n; j++)
     if (a[j] < a[m])
       m = j;
   return m;
 }

 // Feld sortieren
 static void sortiere(final int[] a) {
   int n = a.length;
   int m, t;

   for (int i = 0; i < n; i++) {
     // INVARIANTE:
     // a[0..i-1] ist aufsteigend sortiert und enthaelt kein
     // Element das groesser ist als eines aus a[i..n-1]

     // suche Index des kleinsten in a[i..n-1]:
     m = minAb(a, i);
     // m enthaelt Index des kleinsten in a[i..n-1]:

     // tausche Inhalt von a[i] und a[m]:
     t = a[i];
     a[i] = a[m];
     a[m] = t;
   }
```

```
    // a[0..i-1] ist aufsteigend sortiert, i = n,
    // also: a ist aufsteigend sortiert !
  }

  // Feld ausgeben
  static void drucke(final int[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++)
      System.out.print(a[i]+" ");
  }
}
```

Felder als Parameter, Seiteneffekt

Felder verhalten sich als Parameter völlig anders als Parameter einfacher Typen wie etwa `int` oder `double`. Wenn "einfache" Parameter übergeben werden, dann haben alle Aktionen innerhalb der Funktion keinerlei Einfluss auf die Argumente beim Aufrufer. Die Funktion `tauscheNicht`

```
  static void tauscheNicht(int a, int b) {
     int t = a;
     a = b;
     b = t;
  }
```

hat beispielsweise keinerlei Einfluss auf die Werte von `x` und `y` in einem Aufrufer:

```
  int x = 1;
  int y = 2;
  tauscheNicht(x,y);
  // immer noch x == 1, y == 2
```

Das liegt daran, dass `a` und `b` neue lokale Variablen von `tauscheNicht` sind. Sie werden zwar mit den Werten von `x` und `y` initialisiert, aber alle Aktionen der Funktion auf `a` und `b` sind ohne Einfluss nach außen.

Bei Feldern verhält es sich anders. Die Funktion

```
  static void tausche(final int[] a) {
    int t = a[0];
    a[0] = a[1];
    a[1] = t;
  }
```

tauscht den Inhalt des Feldes tatsächlich:

```
  int[] x = { 1, 2 };
  tausche(x);
  // jetzt x[0] == 2, x[1] == 1
```

Dies haben wir uns im Sortierbeispiel zunutze gemacht. Die Funktion `sortiere` sortiert das übergebene Feld und die Sortierung ist auch im Aufrufer wirksam.

Man nennt dieses Phänomen einen *Seiteneffekt*: `sortiere` und `tausche` haben eine Seiteneffekt: sie verändern fremde Variablen.

Felder als Individuen

Der Grund für das unterschiedliche Verhalten bei der Parameterübergabe hängt mit dem unterschiedlichen Verhalten beim Vergleich zusammen. Wir erinnern uns

```
a == b
```

ist ein Vergleich auf Identität statt auf Gleichheit, wenn `a` und `b` Felder sind. Die Felder werden als Individuen behandelt. Bei der Parameterübergabe verhält es sich genauso. Es wird nicht der Wert übergeben, sondern das "Individuum". Alle Veränderungen des Individuums in der Funktion bleiben auch nach deren Ende erhalten.

Statt Individuum sagen wir auch *"Objekt"*. Der Unterschied zwischen Werten und Objekten wird uns später noch intensiver beschäftigen.

1.8.4 Zweidimensionale Strukturen

Vektoren und Matrizen

Ein Vektor ist eine Folge von Zahlen. Sie haben eine eindimensionale Struktur. Vektoren können durch Felder dargestellt werden. Matrizen sind zweidimensionale Strukturen. Als solche können sie nicht im Speicher abgelegt werden. Der Speicher ist eine Folge von Bytes und damit eindimensional. Zweidimensionale Strukturen können aber immer auf eindimensionale abgebildet werden.

Eine Matrix kann auf zwei Arten eindimensional dargestellt, also als Kombination von Vektoren verstanden werden:

- Entweder sieht man sie als Folge von Spalten, oder
- als Folge von Zeilen.

In beiden Fällen ist die Matrix ein Vektor von Vektoren. Die Matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

wird also entweder als

- $((a_{11}, a_{12}, a_{13}), (a_{21}, a_{22}, a_{23}))$ (Zeilenform, Standard), oder als
- $((a_{11}, a_{21}), (a_{12}, a_{22}), (a_{13}, a_{23}))$ (Spaltenform, nicht üblich)

interpretiert. Die erste Variante ist die Zeilenform (Matrix = Vektor von Zeilen–Vektoren), die zweite ist die Spaltenform (Matrix = Vektor von Spalten–Vektoren).

Standarddarstellung einer Matrix

Für die Sprache Java, und fast alle anderen Programmiersprachen, ist festgelegt, dass Matrizen (zweidimensionale Strukturen) in der *Zeilenform* gespeichert werden, also als Vektor von Zeilen–Vektoren. Jede Matrix ist damit ein Feld von Zeilen und jede Zeile ein Feld von Werten. Der Typ einer Matrix kann entsprechend definiert werden.

Beispielsweise kann die 2×3 Matrix (2 Zeilen, drei Spalten) von oben wie folgt definiert werden:

```
int[][] a = new int[2][3];         // 2 Zeilen mit 3 Elementen
```

Eine Matrix ist ein Feld von Zeilen. Jede Zeile ist ein Feld von drei Elementen. Die Variable `a` hat den Typ `int[][]`, was man als "Feld von Feldern von ints" lesen kann. Die etwas seltsame Konstruktion `int[2][3]` liest man am besten von der Mitte nach rechts und dann in einem Sprung nach links (die umfassendere Struktur steht weiter innen):

`int`**`[2]`**`[3]`	Die Gesamtstruktur besteht aus 2 Haupt–Elementen (2 Zeilen)
`int[2]`**`[3]`**	von denen jedes wiederum 3 Elemente hat (2 Zeilen à 3 Elemente)
`int``[2][3]`	die wiederum alle vom Typ `int` sind.

Weiß man welche Elemente ein solches Feld haben soll, dann wird die Sache ganz einfach:

```
int[][] a = { { 11,12,13 }, { 21,22,23 } }
```

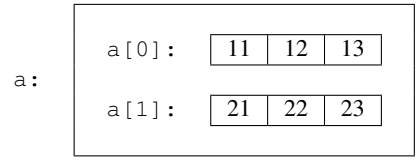
Man schreibt die Matrix als Folge Ihrer Zeilen auf.

Elementzugriff

Der Zugriff auf das Matrixelement $a_{i,j}$ wird zu `a[i][j]`: Element `j` von Element `i` von `a`. In `a[i]` – einer Variablen vom Typ `Zeile` – wird auf das Element Nr. `j` zugegriffen. (Man beachte immer, dass in Java *jeder Indexbereich mit Null beginnt* !) Die Matrix

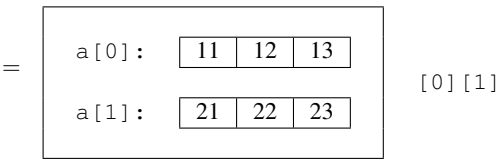
$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{pmatrix}$$

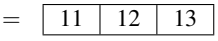
wird mit den Deklarationen von oben folgendermaßen in `a` gespeichert:



`a[i][j]` ist also als Element Nr `j` in der Zeile Nr. `i` zu interpretieren. `a[0][1]` beispielsweise ist Element Nr. `1` von `a[0]`:

`a[0][1]`





$= 12$

Beispiel: Determinante berechnen

Im folgenden Beispiel berechnen wir die Determinante einer 2×2 Matrix:

```
...
public static void main(String[] args) {
  int a[][] = {{11,12}, {21,22}};
  int det = determinante(a);
  System.out.println("Die Determinante von:");
  drucke(a);
  System.out.println("ist: " + det);
}

static int determinante(final int[][] m) {
  return m[0][0]*m[1][1] - m[0][1]*m[1][0];
}

static void drucke(final int[][] a) {
  for (int i = 0; i < a.length; i++){
    for (int j = 0; j < a[i].length; j++)
      System.out.print(a[i][j] + " ");
    System.out.println();
  }
}
...
```

Beispiel: Matrix–Multiplikation

Bekanntlich kann man zwei Matrizen **A** und **B** miteinander multiplizieren, wenn die Spaltenzahl der ersten gleich der Zeilenzahl der zweiten ist. (**A** ist eine $M \times K$ und **B** eine $K \times N$ Matrix.) Ein Element $c_{i,j}$ der Ergebnismatrix ist das Produkt einer Zeile und einer Spalte (Mathematische Notation: Indizes starten mit 1):

$c_{i,j} = \mathbf{A}_i * \mathbf{B}^j = \sum_{k=1}^{K} a_{i,k} * b_{k,j}$

Die Multiplikation einer 2×3 und mit einer 3×2 Matrix ergibt eine 2×2 Matrix (Java Notation: Indizes starten mit 0):

```
public final class MatMult {

  private MatMult() {}

  public static void main(String[] args) {
     int[][] a = {{1,2,3},{2,4,6}};
     int[][] b = {{0,1},{1,2},{2,3}};
     int[][] c = mult(a, b);
    drucke(c);
```

```
  }

  static int[][] mult(final int[][] x, final int[][] y) {
    if ( x[0].length != y.length )
      return null;              // Fehler!

    int[][] z = new int[x.length][y[0].length];

     for (int i=0; i<x.length; i++)
        for (int j=0; j<y[0].length; j++){
          int s = 0;
          for (int k=0; k<x[0].length; k++)
            s = s + x[i][k] * y[k][j];
          z[i][j] = s;
        }
    return z;
  }
  ....
}
```

In `mult` testen wir zuerst, ob die übergebenen Matrizen die Bedingung erfüllen, also ob die Spaltenzahl der einen, gleich der Zeilenzahl der anderen ist. Wenn nicht, geben wir den undefinierten Wert `null` zurück. Dann wird ein Feld der passenden Größe erzeugt und entsprechend der mathematischen Vorschrift mit Werten belegt und zurückgegeben.

1.8.5 Beispiel: Pascalsches Dreieck

Das Pascalsche Dreieck

Das Pascalsche Dreieck hat die Form:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
        ...
```

Das Bildungsgesetz ist wohl offensichtlich. Man kann die Zeilen auch linksbündig hinschreiben.

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
...
```

Dabei erhält man eine Folge von Zeilen, deren Länge stetig wächst. Man kann natürlich die “fehlenden” Werte als Null annehmen um eine Folge von Zeilen fester Länge zu erhalten:

```
1  0  0  0  0  0  0  0  0
1  1  0  0  0  0  0  0  0
1  2  1  0  0  0  0  0  0
1  3  3  1  0  0  0  0  0
1  4  6  4  1  0  0  0  0
...
```

Das Dreieck als Inhalt einer Matrix

Das Pascalsche Dreieck kann leicht als Inhalt eines zweidimensionalen Feldes berechnet werden. Jede Zeile wird durch einfache Additionen aus der darüberliegenden berechnet. Die erste Zeile benötigt natürlich eine Sonderbehandlung, sie hat ja keinen Vorgänger.

Innerhalb einer Zeile berechnet sich jedes Element als Summe der Elemente direkt und links über ihm. Auch hier benötigt das erste Element eine Sonderbehandlung: links über ihm gibt es ja nichts.

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

// erste Zeile setzen
x[0][0] = 1; // Erstes Element, Sonderbehandlung
for (int i= 1; i< 10; i++)
  x[0][i] = 0;

// Zeilen aus ihren Vorg"angern berechnen
for (int i = 1; i< 10; i++) {
  x[i][0] = 1; // Erstes Element, Sonderbehandlung
  for (int j = 1; j < 10; j++)
    x[i][j] = x[i-1][j-1] + x[i-1][j];
}
```

Zweidimensionale Felder müssen nicht quadratisch sein

Eigentlich ist es Platzverschwendung ein quadratisches Feld anzulegen, wenn die Hälfte der Einträge aus Nullen besteht. Es ist auch nicht notwendig. Ein zweidimensionales Feld besteht aus einer Reihe von Zeilen. Es gibt kein Gesetz, das verlangt, dass diese Zeilen alle die gleiche Länge haben müssen. Es ist durchaus möglich Felder anzulegen, die nicht quadratisch sind:

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

for (int i= 0; i< 10; ++i)
  x[i] = new int[i+1];
```

Hier wird `x` zuerst mit drei unbestimmten Zeilen belegt. In der Schleife werden dann die Zeilen genauer als Kollektionen von einem, zwei, ... zehn Elementen spezifiziert.

1.8.6 Foreach–Schleife und Varargs

Foreach–Schleife

Die Elemente eines Feldes können mit einer vereinfachten Form der *for*–Schleife, der sogenannten *foreach*–Schleife durchlaufen werden. Ihr Name kommt nicht von einem neuen Schlüsselwort, sondern von der Tatsache, dass mit ihr *auf jeden Wert* im Feld zugegriffen werden kann – nicht auf jeden Speicherplatz des Feldes wie sonst. Ein Beispiel ist:

```
String a[] = {"Hallo", "wer", "da", "?"};
for (String s : a) { // Lies: Fuer jeden String s in a
  System.out.println(s);
}
```

Das Feld kann mit dieser Form nicht verändert werden. Es handelt sich, wie gesagt, um eine Schleife über alle Werte im Feld.

Varargs: Variable Argumentliste

Funktionen werden normalerweise mit einer festen Argumentliste definiert. Die Zahl der formalen und der aktuellen Parameter muss dann exakt übereinstimmen. In manchen Fällen ist etwas mehr Flexibilität jedoch nützlich. Ein einfaches Beispiel für diese sogenannten *Varargs* ist:

```
public final class Hallo {

    private Hallo() {}

static void printStrings(final String... a) {
  for (String s : a)
    System.out.println(s);
}

public static void main(String[] args) {
  printStrings("Hallo", "Du");
```

```
   printStrings("da!");
   printStrings("Blubber", "Blabber", "Plitsch");
   printStrings();
  }
  }
```

Variable Argumentlisten werden in Felder gepackt und als solche an die Funktion übergeben.

1.8.7 Felder als Datenbehälter

Felder können zur Modellierung von strukturierten Daten eingesetzt werden. Ein einfaches Beispiel sind Vektoren in der Ebene. Ein solche ebener Vektor hat zwei Komponenten: Eine x– und eine y–Koordinate und kann darum in einem Feld mit zwei Elementen gespeichert werden. Für zwei Vektoren nehmen wir zwei Felder:

```
double[] v1 = new double[]{1.0, 2.0};
double[] v2 = new double[]{2.0, 1.0};
```

Natürlich hätten wir auch schreiben können:

```
double v1x = 1.0;
double v1y = 2.0;
double v2x = 2.0;
double v2y = 1.0;
```

aber die erste Form bringt die Zusammengehörigkeit der Wertepaare besser zum Ausdruck.

Kommen Funktionen ins Spiel dann werden die Vorteile des Zusammenfassens noch offensichtlicher:

```
package geo;

public final class Vektoren {
  private Vektoren() {}

  public static void main(String[] args) {
    double[] v1 = new double[]{1.0, 2.0};
    double[] v2 = new double[]{2.0, 1.0};

    double[] v3 = add(v1, v2);
  }

  /**
   * Addiert ebene Vektoren als Felder und in Koordinatendarstellung.
   * @param v1 Vektor 1
   * @param v2 Vektor 2
   * @return Vektorsumme
   */
  static double[] add(double[] v1, double[] v2) {
    return new double[]{
       v1[0]+v2[0],
       v1[1]+v2[1]};
  }
}
```

Matrizen können genauso behandelt werden. Ein Funktion, die eine Matrix mit einem Vektor multipliziert wird dann zu:

```
/**
 * Multipliziere Matrix mit Vektor, beide als Felder.
 * @param m die Matrix
 * @param v der Vektor
 * @pre m[i].length == v.length fuer alle i<=0<m.length
 * @return m X v
 */
static double[] matXVek(double[][] m, double[] v) {
  double[] result = new double[m.length];
  for (int i=0; i<v.length; i++) {
    result[i] = 0.0;
    for (int j=0; j<m.length; j++) {
      result[i] = result[i] + m[i][j]*v[j];
```

```
    }
  }
  return result;
}
```

Leider lässt sich im Kopf der Funktion nicht ausdrücken, dass Matrix und Vektor zueinander passen müssen. D.h. dass der Vektor so “hoch” sein muss, wie die Matrix “breit” ist.

Kapitel 2

Objektorientierung I: Module und Objekte

2.1 Modularisierung und Objektorientierung

2.1.1 Objektorientierung I: Klassen als Module

Klassen als Kollektionen statischer Methoden

Bis jetzt hatten wir es stets nur mit Java–Programmen zu tun, die aus einer einzigen Klasse bestehen. Diese Klassen waren stets `final`, enthielten einen privaten Konstruktor und umfassten in der Regel mehrere Funktionen, eine davon öffentlich und mit dem Namen `main`. `main` ist der Ausgangspunkt jeden Programmlaufs. Von dort geht es zu anderen Funktionen. Das gesamte Programm endet, wenn `main` zu Ende ist.

Eine Klasse ist also ein Behälter von Funktionen, oder wie wir auch sagen, von statischen Methoden. Die als `public` definierten Methoden sind zur öffentlichen Verwendung freigegeben, die anderen sind deren interne Gehilfen.

Modul

Das Konzept der Zusammenfassung von Elementen zu einer Einheit hat eine lange Tradition, nicht nur in der Informatik. Es wird oft als *Modulkonzept* bezeichnet. Ein Modul hat eine klar definierte Schnittstelle und ein Innenleben, von dem wir wissen, *was* es tut, aber *wie* es das macht, ist unerheblich. Ein Modul ist damit eine austauschbare Einheit. Jederzeit kann es gegen ein anderes mit der gleichen Schnittstelle und dem gleichen Verhalten ersetzt werden. Die *Modularisierung* ist die Aufteilung eines Gesamtsystems in Module, die in sich jeweils möglichst abgeschlossen und voneinander möglichst unabhängig sind.

Module begegnen uns tagtäglich in der Welt der Technik. Der Monitor meines Computers ist ein Modul, er kann jederzeit ohne weitere Probleme (hoffentlich) gegen einen anderen Monitor ausgetauscht werden. Ein kaputtes Getriebe im Auto kann leicht ausgetauscht werden. Die Modularierung des Hochschulunterrichts hat das Ziel, dass beispielsweise der Java–Dozent samt seiner Veranstaltung jederzeit problemlos durch einen anderen mit einer anderen Lehrveranstaltung ersetzt werden kann. Die einzige Bedingung ist, dass beide sich an die Modulbeschreibung – die Spezifikation des Moduls – halten.

Der Sinn eines Moduls ist klar: Eine bestimmte Funktionalität soll zu einer standardisierten Einheit gekapselt werden, die – durch ihre beschränkte Funktionalität – leichter entwickelt werden kann als das Gesamtsystem und bei Fehlfunktion schnell und kostengünstig ausgetauscht werden kann.[1]

In der OO–Euphorie der frühen neunziger Jahre des letzten Jahrhunderts, als Java konzipiert wurde, galt das Modulkonzept nicht mehr als ausreichend *hip*, um mit einem eigenen Sprachkonstrukt ausgestattet zu werden. Es wird darum als *Beschränkung einer Klasse* definiert:

> Module sind abgegrenzte Programmteile mit definierter Aufgabe und Schnittstelle.
> In Java werden Module als finale Klassen mit privatem Konstruktor und nur statischen Methoden und Attributen realisiert.

Es wirkt etwas befremdlich und ist auch umständlich, das Einfache als Beschränkung des Komplizierten definieren zu müssen. Oft wird darum `final` und der private Konstruktor weggelassen. Das ist aber ein schlechter Programmierstil, den man sich als ernsthafter Software-Entwickler gar nicht erst angewöhnen sollte.

Ein Telefonverzeichnis als Modul: Eine Klasse repräsentiert ein Exemplar

Wir beginnen mit einem einfachen Beispiel eines Moduls: einem Telefonverzeichnis. Ein (elektronisches) Telefonverzeichnis enthält Einträge mit denen einem Namen eine Nummer zugeordnet wird. Es sollte zudem eine Funktion umfassen mit dem man nach einer Nummer suchen kann und zur Verwaltung sollten auch Funktionen zum Hinzufügen und zum Löschen von Einträgen vorhanden sein.

Ein Telefonbuch umfasst also insgesamt:

- Daten: Paare aus Namen und Nummern
- Eine Suchfunktion: Nummer zu einem Namen finden
- Zwei Verwaltungsfunktionen: Erweitern und Löschen

und ist leicht zu implementieren:

[1] Ok, bei Java–Dozenten steht möglicherweise das Beamtenrecht noch einem kostengünstigen Austausch entgegen.

```
package telefonBuch;

/**
 * Diese Klasse repraesentiert ein Telefonverzeichnis
 *
 */
public final class TelefonVerzeichnis {
  private TelefonVerzeichnis() {} // Kein nutzbarer Konstruktor: Es gibt nur eins!

  // Kapazitaet des Verzeichnisses
  private static int kapazitaet = 100;

  // Alle Namen, Maximal Anzahl: kapazitaet
  private static String[] name = new String[kapazitaet];

  // Alle Nummern, Maximal Anzahl: kapazitaet
  // name[i] ist zugeordnet nummer[i]
  private static int[] nummer = new int[kapazitaet];

  //Anzahl der Eintraege
  private static int anzahl = 0;

  /**
   * Suche Nummer zu Name
   * @param n der Name
   * @return die zugeordnete Nummer falls vorhanden, sonst -1
   */
  public static int suche(String n) {
    for (int i=0; i<anzahl; i++) {
      if (name[i].equals(n)) {
        return nummer[i];
      }
    }
    return -1;
  }

  /**
   * Erweitere um neuen Eintrag
   * @param n Name
   * @param nr zugeordnete Nummer
   * @pre anzahl < kapazitaet
   * @post n -> nr ist eingetragen falls noch nicht vorhanden
   */
  public static void eintrage(String n, int nr) {
    if (suche(n) != -1) { // Eintrag schon vorhanden
      return;
    }
    name[anzahl] = n;
    nummer[anzahl] = nr;
    anzahl = anzahl+1;
  }

  public static void loesche(String n) {
    for (int i=0; i<anzahl; i++) {
      if (name[i].equals(n)) {
        for (int j= i; j<anzahl; j++) {
          name[j] = name[j+1];
          nummer[j] = nummer[j+1];
        }
        break;
      }
    }
    anzahl = anzahl -1;
  }

}
```

Die Klasse enthält Variablen und Methoden. Die Variablen werden benötigt, um die Daten zu speichern und zu verwalten.

Sie sind `private` und darum für den Benutzer (den benutzenden Java–Code) nicht zugreifbar. Das ist das Modulkonzept: Innereien unterliegen dem Geheimnisprinzip.

Die Methoden stellen die Schnittstelle nach außen dar, sie sind `public` und können darum in anderen Klassen genutzt werden, etwa so:

```
package telefonBuch;

public final class Nutzer {
  private Nutzer() {}

  public static void main(String[] args) {
    TelefonVerzeichnis.eintrage("Hugo", 4711);
    TelefonVerzeichnis.eintrage("Karla", 4712);
    TelefonVerzeichnis.eintrage("Natascha", 4713);
    TelefonVerzeichnis.loesche("Karla");
    TelefonVerzeichnis.eintrage("Egon", 4714);

    for (String n : new String[]{"Hugo", "Karla", "Natascha", "Karla", "Egon"}) {
      System.out.println(n + " -> " + TelefonVerzeichnis.suche(n));
    }
  }

}
```

Importe und statische Importe

Soll eine Funktion einer anderen Klasse benutzt werden, dann gibt man vor der Funktion den Namen der Klasse an, zu der sie gehört.

$< KlassenName > . < FunktionsName >$

Liegt der Nutzer nicht einmal mehr im gleichen Paket, dann muss die benutzte Klasse importiert

```
package nutzer; // <<------

import telefonBuch.TelefonVerzeichnis;

public final class Nutzer {
  private Nutzer() {}

  public static void main(String[] args) {
    TelefonVerzeichnis.eintrage("Hugo", 4711);
    ... etc. ...
  }

}
```

oder “voll qualifiziert” werden:

```
package nutzer;

public final class Nutzer {
  private Nutzer() {}

  public static void main(String[] args) {
    telefonBuch.TelefonVerzeichnis.eintrage("Hugo", 4711);
    ... etc. ...
  }

}
```

Mit einem *statischen Import* kann man sich auch die Erwähnung des Klassennamens ersparen:

```
package nutzer;

import static telefonBuch.TelefonVerzeichnis.eintrage; // <<------
```

```
public final class Nutzer {
 private Nutzer() {}

 public static void main(String[] args) {
  eintrage("Hugo", 4711);
  ... etc. ...
 }

}
```

Private Konstruktoren

Mit der Klasse `TelefonVerzeichnis` haben wir ein Telefonverzeichnis als abgetrennte Komponente, als ein Modul, implementiert. Die Klasse repräsentiert / implementiert *genau ein* Telefonverzeichnis. Damit können wir endlich ein Geheimnis lüften. Das Geheimnis der seltsamen privaten Definition am Anfang aller Klassen bisher. Im Beispiel:

```
 private TelefonVerzeichnis() {}
```

Mit dieser Definition soll gesagt werden, dass die Klasse `TelefonVerzeichnis` genau ein Ding darstellt. Dass es also nicht viele Telefonverzeichnisse in einem Programm geben kann, sondern nur dieses eine mit diesen Methoden und internen Variablen.

Eine Funktion die den Namen der Klasse und keinen Ergebnistyp hat ist der *Konstruktor* der Klasse. Ihn als privat zu deklarieren bewirkt, dass die Konstruktion nicht möglich ist. Telefonverzeichnisse können nicht erzeugt werden. Es gibt eins und damit ist es genug.

Lokale Sichtbarkeit durch `private`

Die Definitionen einer Klasse können mit Hilfe von `public` und `private` explizit sortiert werden in:

- öffentlich verfügbare Leistungen und
- rein interne Hilfsfunktionen/Hilfsvariablen.

Unser Telefonverzeichnis hat öffentliche statische Methoden und private statische Variablen.

Es ist nicht unbedingt notwendig, dass nur (alle) Methoden öffentlich und nur (alle) Variablen privat sind. Eine Funktion zur Primfaktorzerlegung kann beispielsweise diverse Hilfsfunktionen benutzen. Wollen wir etwa nur die Primfaktorzerlegung zur allgemeinen Benutzung freigeben, die Hilfsfunktionen aber vor der Benutzung durch andere schützen, dann packen wir alles in eine Klasse (die als Modul–Implemenierung dient) und definieren alles außer der Funktion `prim` als `privat`:

```
public final class Prim {

 private Prim() {}

 static private boolean prim (final int n) {
    // ist n prim?
    if (n == 2) return true; // 2 ist eine Primzahl
    for (int i = 2; i<n; ++i) { // hat n einen Teiler?
     if (teilt (n, i)) return false;
    }
    return true;
  }
  static private int potenz (final int n, final int p) {
    // liefert groesstes x mit p hoch x teilt n
    // falls p ein Teiler von n ist.
    int i = 0,
       pp = 1; /* pp == p**i */
    while (teilt (n, pp)) {
      ++i;
      pp *= p;
    }
    return i-1;
  }
  static private boolean teilt (final int n, final int t) {
    /* Wird n von t geteilt */
```

```
    return (n % t == 0);
  }
  public static String zerlege (final int n) { // oeffentlich
    String res = "";
    int  count = 0;
    // Jeder Teiler der prim ist, ist ein Primfaktor:
    for (int i=2; i<n; ++i){
      if (teilt (n, i) && prim (i)) {
        res = res + i + " hoch " + potenz(n, i) + "\n";
        count++;
      }
    }
    if (count == 0)
      return " ist eine Primzahl";
    else
      return res;
  }
}
```

Private Funktionen können außerhalb der Klasse nicht benutzt werden:

```
public final class EineAndereKlasse {
  ...
  public static void main(String[] args) {
     System.out.println(Prim.zerlege(12)); // OK
     System.out.println(Prim.prim(12)); // Nicht erlaubt
  }
  ...
}
```

Man sagt auch, dass die Klasse `Prim` die Funktion `zerlege` *exportiert*. Die Schnittstelle von `Prim` ist

```
public static String zerlege (final int n)
```

Der Rest der Klasse ist Implementierung.

Wozu aber gibt es die Unterscheidung in `public` und `private`, wenn sie doch nur zu einer Beschränkung in der Benutzung von Definitionen führt?

2.1.2 Module und das Geheimnisprinzip

Geheimnisse einer Klasse nutzen deren Anwendern: Was ich nicht weiß, kann ich nicht missverstehen oder vergessen.

Die *Kapselung*, also die Trennung von Schnittstelle und Implementierung, mit Hilfe von `public` und `private` wird oft auch als *Geheimnisprinzip* bezeichnet. Die Implementierung, das Private, ist ein Geheimnis der Klasse. Sie geht keinen ihrer Benutzer etwas an.

Die klare Trennung von Interna und benutzbarer Schnittstelle nutzt zunächst einmal den Benutzern/Anwendern der Klasse. Sie können sich auf das konzentrieren, was als öffentlich deklariert und somit zur Benutzung freigegeben wurde.

Jeder, der ein komplexes System anwenden muss, wird dankbar alles zur Kenntnis nehmen, was er *nicht* kennen oder wissen muss. Ein Lichtschalter wird einfach gedrückt. Ich muss mir dabei keine Gedanken darüber machen, ob es sich um einen einfachen Schalter oder einen Wechselschalter handelt, ob er ein Relais bedient oder direkt schaltet, welche Kontakte verbunden werden, und so weiter. Ich sehe die Schnittstelle und kann das Ding bedienen. Ich weiß, dass ich nicht wissen muss, wie ein Wechselschalter funktioniert und das ist erfreulich.

Geheimnisse nutzen den Implementierern: Was nur ich weiß, brauche ich nicht abzusprechen oder zu erklären.

Auf die privaten Komponenten kann nur von innerhalb der Klasse selbst aus zugegriffen werden. Damit ist garantiert, dass sie jederzeit und ohne Absprache mit dem Benutzer der Klasse geändert, gestrichen oder durch gänzlich andere ersetzt werden kann.

Die Elektrik eines Hauses kann geändert werden, ohne dass jeder, der das Licht anschalten will, danach einen Lehrgang besuchen muss. Alle Lichtschalter haben die gleiche Schnittstelle, egal, ob es sich um Wechsel– oder Relaisschalter handelt. Die

einen können darum jederzeit ohne Absprache gegen die anderen ausgetauscht werden.[2]

Kopplung von Softwarekomponenten

Unter *Kopplung* versteht man die Verzahnung von Softwarekomponenten die dadurch entsteht, dass das Wissen über das Wesen der einen, ein Bestandteil der anderen ist. So ist der Benutzer der Klasse `Prim` an diese Klasse gekoppelt, weil er beispielsweise wissen muss, wie die Methode zur Primfaktorzerlegung heißt und welche Parameter und welches Ergebnis sie hat. Dieses Wissen ist im Quellcode des Benutzers "fest verdrahtet" und beide sind damit gekoppelt.

Die Kopplung von Softwarekomponenten sollte einerseits klar erkennbar und andererseits so klein wie möglich sein. Der Grad der Kopplung wird durch das Geschick oder Ungeschick der Software–Entwickler bestimmt. Die Unterscheidung von Schnittstelle und Implementierung kann die Kopplung nur *dokumentieren*: alles was zur Schnittstelle gehört, aber nur das, kann in einer anderen Komponenten benutzt werden. Die Kopplung zwischen dem Benutzer und der Realisation einer Klasse wird so klar ersichtlich auf die Schnittstelle – also die öffentlichen Komponenten – begrenzt.

`private` und `public` dienen dazu den Programmcode zu organisieren

Mit dem Schlüsselwort `private` wird das Aussehen von Programmtexten beeinflusst. Mit ihm sollen bestimmte Komponenten einer Klasse – *nicht vor Menschen* (!) –, sondern *vor anderen Textstücken* im gleichen Programm "verborgen" werden. Wobei "verborgen" nichts anderes heißt, als dass der Compiler eine Fehlermeldung ausgibt, wenn er einen "verborgenen" (privaten) Bezeichner an der falschen Stelle antrifft.

Dies alles bezieht sich nur auf den Quelltext von Programmen. Was wann welcher Mensch sehen oder nicht sehen darf, ist eine völlig andere Frage. Ob die Programmiererin, die die öffentlichen Definitionen einer Klasse benutzt, die Definition der privaten Komponenten mit eigenen Augen sehen darf, hat vielleicht etwas mit dem Betriebsklima oder mit Lizenzverträgen zu tun. Es wird ihr aber vom Compiler weder verboten noch erlaubt.[3]

Der Compiler interessiert sich nicht für die Beziehungen zwischen Menschen sondern nur für Programmtexte. Er prüft und übersetzt einen Text ohne zu wissen von wem er stammt. Mit `private` kann der Programmierer dem Compiler nur eine Absicht für die Organisation des Quelltextes bekannt geben. Dieser prüft dann, ob diese Absicht im gesamten Programm auch eingehalten wird.

`private` hat also ganz und gar nichts mit der Privatsphäre, den Geheimnissen oder den Besitzverhältnissen zwischen Menschen zu tun. Es trennt lediglich die Schnittstelle (das "Öffentliche") und die Implementierung (das "Private") in einem Stück Software.

Vektoren als Modul

Angenommen, wir hätten es mit dreidimensionalen Vektoren zu tun, also mit Vektoren, die aus drei Komponenten bestehen. Beispiele von solchen Vektoren sind

$$\vec{a} = \begin{pmatrix} 5 \\ 2 \\ 4 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

Im Abschnitt über Felder haben wir uns schon mit Vektoren und ihrer Modellierung in einem Programm beschäftigt: Vektoren wurden einfach auf Felder (Arrays) abgebildet. In unserem Beispiel könnten wir Vektoren auf diese Art definieren:

```
double[] a = new double[]{5.0, 2.0, 4.0};
double[] a = new double[]{1.0, 0.0, 1.0};
```

Das ist gut, aber es geht besser: modularsieren wir. Modularisieren bedeutet ja nichts anderes als Zusammengehöhriges zusammen zu führen und öffentliche Dienstleistungen von privaten Hilfskonstrukten zu unterscheiden.

In der Art unseres Telefonverzeichnisses haben wir schnell eine Klasse `Vektor` geschrieben:

```
package geo;
```

[2] Benutzer, die direkt auf die Implementierung zugreifen, müssen umlernen, oder sterben wegen Stromschlag aus.
[3] Compiler sehen nicht, was Menschen so alles tun, sie sehen nur das was sie schreiben und ihnen vorlegen.

```
public final class Vektor {
  private Vektor() {}

  private static double[] koordinaten = new double[3];
}
```

Dazu können wir noch ein paar Funktionen nehmen. Beispielhaft definieren wir die Vektoraddition:

```
public final class Vektor {
  private Vektor() {}

  private static double[] koordinaten = new double[3];

  public static double[] add(double[] v1, double[] v2) {
    return new double[]{v1[0]+v2[0], v1[1]+v2[1], v1[1]+v2[1]};
  }

  ... etc. ...
}
```

Hmm, ... da stimmt etwas nicht. Jetzt gibt es dreimal die Koordinaten. Die Addition nimmt zweimal einen Satz Koordinaten an und ignoriert (sinnvollerweise) die priavte Variable `koordinaten`. Hmm, irgend etwas ist hier anders als beim Telefonverzeichnis. Nur was?

Nun das Telefonverzeichnis arbeitet mit einem Satz Daten, die Funktionen `loesche`, `suche` etc. kommen nicht mit eigenen Daten. Die Vektoroperationen haben dagegen ihre eigenen Daten. Es macht keinen Sinn dass lokal im Modul – in der Klasse – Vektorkoordinaten gespeichert werden:

```
public final class Vektor {
  private Vektor() {}

  // UNSINNIG: private static double[] koordinaten = new double[3];

  public static double[] add(double[] v1, double[] v2) {
    return new double[]{v1[0]+v2[0], v1[1]+v2[1], v1[1]+v2[1]};
  }

  ... etc. ...
}
```

Ein Telefonverzeichnis, das seine eigenen einmaligen Daten besitzt ist OK. Aber ein Programm mit einem einzigen Vektor macht wenig Sinn. Wir brauchen viele Vektoren und jeder Vektor hat seine eigenen Koordinaten. Darstellung / Speicherung der Vektoren kann nicht zusammen mit den Vektoroperationen innerhalb einer Klasse konzentriert werden. Es kann ja beliebig viele Vektoren geben, aber aber nur eine Klasse mit den Vektoroperationen. – Eine Verletzung des Geheimnisprinzips ist unausweichlich!

Der Nutzer der Vektoren kann auf die “Innereien” der Vektoren zugreifen und dabei beliebig viel falsch machen:

```
public final class Vektor {
  private Vektor() {}

  // UNSINNIG: private static double[] koordinaten = new double[3];

  public static double[] add(double[] v1, double[] v2) {
    return new double[]{v1[0]+v2[0], v1[1]+v2[1], v1[1]+v2[1]};
  }

  ... etc. ...
}

...

double v1 = new double[]{0.5};
double v2 = new double[]{2.5, 3.5};
double v3 = Vektor.add(v1, v2); // OK
v3[0] = v1[1];            // Pfui: Geheimnisprinzip verletzt
```

Die Vektor–Klasse und ihre Benutzer sind (zu) eng gekoppelt: Die Implementierung (Darstellung) der Vektoren kann nicht ohne Auswirkung auf die Benutzer der Vektoren geändert werden.

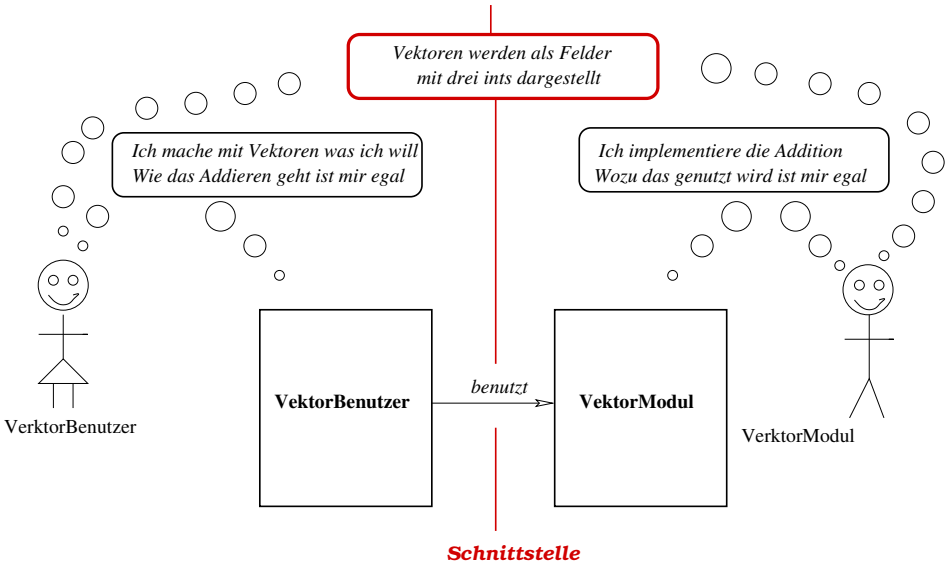


Abbildung 2.1: Hohe Kopplung trotz Modularisierung

Eine **Klasse** als **Modul**:
Statische Variablen– und Methoden–Definitionen werden zu einer Einheit zusammengefasst.
Diese Modularisierung ist geeignet

- als Hülle um eine Sammlungen verwandter Funktionen
- als Hülle um ein einziges Exemplar aus dem Anwendungsbereich

Es versagt, wenn es um die Behandlung beliebig vieler gleichartiger Dinge geht.

2.1.3 Objektorientierung II = Datenabstraktion: Klassen als Typen

Datenabstraktion: Die Darstellung von Vektoren gehört zu den Vektor–Funktionen

Die Darstellung (Art der Speicherung) der Vektoren, hier als Feld von drei Int–Werten, ist *nicht* naturgemäß etwas, auf das sich die Benutzer der Vektorrechnung und deren Implementierer einigen sollten. Es ist etwas, das eigentlich in den Bereich der Implementierung der Vektorrechnung gehört. Es sollte ein *Geheimnis* der Implementierung sein. Solange die Vektoraddition und andere Operationen funktionieren, sollte es deren Benutzer egal sein, wie sie intern abgespeichert werden.[4]

Man spricht von *Datenabstraktion*, wenn Operationen auf den Daten und die interne Darstellung der Daten in einer Software–Komponenten zusammengefasst werden. Die Benutzer der Daten brauchen (müssen und und dürfen, Geheimnis!) dann nicht mehr wissen, wie sie intern gespeichert werden. Die Implementierer der Komponenten können die interne Darstellung jederzeit ohne Absprache ändern. Kurz: Die Datenabstraktion eliminiert die unerwünschte Kopplung durch gemeinsame Kenntnis der Speicherung von Daten. Datenabstraktion ist ein Grundbaustein der *objektorientierten Programmierung*.

Java unterstützt die Datenabstraktion: Die Definition der Speicherung von Vektoren kann zusammen mit den Funktionen in eine Klasse gelegt werden:

```
public final class Vektor { // Klasse Vektor, der TYP der Vektoren

  // Kein privater Konstruktor

  private int[] darst = new double[3]; // <-<-- interne Darstellung
                          // privat, mein Geheimnis !
```

[4] Wenn ich zur Tankstelle fahre, dann möchte ich auch nicht erst mit dem Besitzer darüber diskutieren müssen, wo und wie genau er sein Benzin lagert und wie ich dran komme. Ich will die wohlbekannte Schnittstelle – die Zapfsäule – benutzen. Alles andere interessiert mich nicht.

```
                            // nicht static !

 // Erzeugung eines Vektors
 public static Vektor neuerVektor(final double i, final double j, final double k) {
      Vektor res = new Vektor();
      res.darst[0] = i;
      res.darst[1] = j;
      res.darst[2] = k;
      return res;
 }

 // Vektoraddition
 public static Vektor add(final Vektor x, final Vektor y) {
    Vektor res = new Vektor();
    res.darst[0] = x.darst[0] + y.darst[0];
    res.darst[1] = x.darst[1] + y.darst[1];
    res.darst[2] = x.darst[1] + y.darst[2];
    return res;
 }

 // Skalarmultiplikation
 public static Vektor skalarMult(final int x, final Vektor v) {
    Vektor res = new Vektor();
    res.darst[0] = v.darst[0] * x;
    res.darst[1] = v.darst[1] * x;
    res.darst[2] = v.darst[2] * x;
    return res;
 }
}
```

Die Information über die interne Darstellung liegt jetzt (als Variable `darst`) *in der Klasse* `Vektor` und nicht mehr beim Benutzer. Damit wir trotzdem mit vielen Vektoren umgehen können, muss jeder einzelne Vektor seine eigene Darstellung (Speicherung der Komponenten) haben. Dies wird durch Weglassen von `static` vor der Variablen `darst` erreicht.

Eine einzige Variable `darst` der Klasse `Vektor` zugeordnet:

```
static int[] darst = new int[3];
```

Beliebig viele Variablen `darst` die jeweils einem Vektorexemplar zugeordnet sind:

```
int[] darst = new int[3];
```

Die Erzeugung eines Vektors muss darum jetzt auch von der Klasse `Vektor` übernommen werden. Die Funktion `neuerVektor` ist mit dieser Aufgabe betraut. Sie erzeugt zunächst einen neuen Vektor (`res`)

```
Vektor res = new Vektor();
```

Diese Anweisung ist nur möglich, wenn es keinen privaten Konstruktor gibt.

```
public final class Vektor {

 private int[] darst = new double[3]; // jeder Vektor hat eine Variable darst

 public static Vektor neuerVektor(final double i, final double j, final double k) {
      Vektor res = new Vektor(); // res ist ein Vektor
      res.darst[0] = i;    // innerhalb von res, die Variable darst belegen
      ...
 }
```

Benutzer der Vektorenklasse müssen sich an diese Definitionen halten. Sie können aber jetzt mit Vektoren arbeiten, ohne sich darum kümmern zu müssen, wie sie intern gespeichert werden:

```
 ....
 public static void main(String[] args) {
    Vektor a = Vektor.neuerVektor(5, 2, 4);
    Vektor b = Vektor.neuerVektor(1, 0, 1);
    Vektor c = Vektor.add(a, b);
    Vektor d = Vektor.skalarMult(3, c);
    ....
 }
 ...
```

Wir sehen hier: Die Klasse `Vektor` ist nicht mehr nur eine Sammlung von Funktionen, *sie kann jetzt auch als Typ verwendet werden.* Wir können Variablen vom Typ `Vektor` anlegen:

```
Vektor a = Vektor.neuerVektor(5, 2, 4);
```

Gleichzeitig ist es weiterhin möglich Funktionen aus der Funktionensammlung `Vektor` aufzurufen:

```
Vektor.add(a, b);
```

Der doppelte Charakter einer Klasse muss unbedingt im Auge behalten werden. Man kann je nach Anwendung und Geschmack Klassen als Funktionssammlung, oder als Typen verwenden. Klassen können also zu beiden Zwecken verwendet werden:

- *Funktionale Abstraktion:* Klassen können eine Sammlung von Funktionsdefinitionen sein und (eventuell gleichzeitig)
- *Datenabstraktion:* einen Typ definieren.

> Eine **Klasse** ist
> 1. ein Modularisierungskonstrukt mit dem Definitionen zu einer Einheit zusammengefasst werden können (statische Klassenkomponenten)
> und
> 2. ein Mechanismus zur Datenabstraktion mit dem neue Typen definiert werden können (Typ ihrer Objekte mit den nicht statischen Klassenkomponenten).

Im Programmcode sieht man die Unterschiede:

Klasse mit Modul–Charakter:

```
public final class C { // Klasse mit Modul-Charakter

  private C () {}  // privater Konstruktor: Keine einzelnen Exemplare von C

  ... static T v;  // statische Variable: Genau einmal vorhanden

}
```

Klasse mit Typ–Charakter:

```
public final class C { // Klasse mit Typ-Charakter

  // kein privater Konstruktor: Einzelne Exemplare von C sind erzeugbar

  ... T v;    // nicht statische Variable: Pro Exemplar vorhanden

}
```

Datenabstraktion und das Geheimnisprinzip

Bei dieser Organisation des Programmcodes ist es jederzeit möglich die interne Darstellung eines Vektors zu ändern, ohne dass ein Benutzer der Vektoren davon überhaupt in Kenntnis gesetzt werden muss. Nehmen wir an, das Feld gefällt uns nicht mehr. Wir hätten lieber drei Variablen `x`, `y` `z` als Vektorkomponenten:

```
public final class Vektor {
  private double x; // interne Darstellung
  private double y; // mit drei Variablen
  private double z; // x, y, z

  public static Vektor neuerVektor(final double i, final double j, final double k) {
     Vektor res = new Vektor();
     res.x = i;
     res.y = j;
     res.z = k;
     return res;
  }

  public static Vektor add(final Vektor x, final Vektor y) {
    Vektor res = new Vektor();
```

```
    res.x = x.x + y.x;
    res.y = x.y + y.y;
    res.z = x.z + y.z;
    return res;
  }

  public static Vektor skalarMult(final int x, final Vektor v) {
    Vektor res = new Vektor();
    res.x = v.x * x;
    res.y = v.y * x;
    res.z = v.z * x;
    return res;
  }
}
```

Ein Benutzer der Klasse muss trotz dieser massiven Änderung der Implementierung nicht angefasst werden.

In dieser Variante hat `x` in `add` eine doppelte Bedeutung. Vor dem Punkt ist der Parameter `x` gemeint, hinter dem Punkt ist die `x`–Variable (die `x`–Komponente) gemeint, die zur Darstellung der Vektoren gehört. In

```
res.x = x.x + y.x;
```

ist mit "`x.x`" die `x`–Komponente des Parameters `x` gemeint.

Objekterzeugung mit new und Konstruktor

Da die Erzeugung neuer Exemplare eine so wichtige Operation ist, wird sie in modernen Sprachen durch stets zwei Mechanismen unterstützt:

- Konstruktor
- new

Der Konstruktor ist eine spezielle Erzeugungsfunktion die automatisch durch new aufgerufen wird.

Mit diesen Mechanismen vereinfacht sich die Klasse der Vektoren zu:

```
public final class Vektor {

  private double x; // interne Darstellung
  private double y; // mit drei Variablen
  private double z; // x, y, z

  public Vektor(final double i, final double j, final double k) {
    x = i;
    y = j;
    z = k;
  }

  public static Vektor add(final Vektor x, final Vektor y) {
    return new Vektor(x.x + y.x, x.y + y.y, x.z + y.z);
  }

  public static Vektor skalarMult(final int x, final Vektor v) {
    return new Vektor(v.x * x, v.y * x, v.z * x);
  }
}
```

2.1.4 Statisch oder nicht statisch, das ist hier die Frage

Methoden: Funktionen die zu Instanzen gehören

Eine Funktion wird in Java als *statische Methode* definiert. Neben statischen Methoden gibt solche, die nicht statisch sind. Wodurch unterscheiden sich statische von nicht statischen Methoden? Bevor wir zu einem Beispiel kommen, erinnern wir uns noch einmal kurz an den Unterschied zwischen Klassenvariablen (statisch) und Objektvariablen (nicht statisch). Klassenvariablen haben nichts mit einzelnen Objekten zu tun: sie gehören zur Klasse. Objektvariablen sind dagegen immer an bestimmte einzelne Objekte gebunden.

Für Methoden gilt eine äquivalente Regel:

- *Statische Methoden* (Funktionen, Klassenmethoden) operieren im Kontext einer Klasse. Sie sind unabhängig von allen Objekten. Sie sind Bestandteil einer Klasse, die verstanden wird als Sammlung von Funktionen und Klassenvariablen. (Modul-Charakter der Klasse)
- *Methoden die nicht statisch sind* gehören zu individuellen Objekten der Klasse. Sie operieren immer im Kontext eines ganz bestimmten Exemplars der Klasse. (Typ-Charakter der Klasse)

Ein Beispiel macht das leicht klar. Wir definieren eine Methode "`laenge`", die die Länge eines Vektors berechnet. Als Methode gehört sie zu einem Vektor und berechnet demnach dessen Länge. Ein Vektor kann mit ihr seine Länge bekannt geben:

```
public final class Vektor {
  private final int x; // interne Darstellung
  private final int y;
  private final int z;

  public static final Vektor nullVektor = neuerVektor(0,0,0);

  public static Vektor neuerVektor(final int i, final int j, final int k) {...}

  // NICHT statische Methode laenge,
  // sie operiert im Kontext eines bestimmten Vektors und
  // berechnet dessen Laenge
  //
  public double laenge () {
    return Math.sqrt(x*x+y*y+z*z);
  }

  public static Vektor add(final Vektor x, final Vektor y) {...}

  public static Vektor skalarMult(final int x, final Vektor v) {...}
}
```

Die Verwendung sieht so aus:

```
public final class VektorBenutzer {
   public static void main(String[] args) {
     Vektor a = Vektor.neuerVektor(3, 0, 4);

     // a's Laenge berechnen:
     double l = a.laenge(); // Aufruf einer (nicht statischen) Methode

     System.out.println(l);

     // noch ein Aufruf einer Methode
     // die Laenge des Nullvektor bestimmen:
     System.out.println(Vektor.nullVektor.laenge());
   }
}
```

Beim Aufruf einer statischen Methode schreibt man vor den Punkt die Klasse zu der sie gehört. Beim Aufruf einer nicht statischen Methode gibt man stattdessen ein *Objekt der Klasse* an. Die Methode wird dann für dieses Objekt (in diesem Objekt) aktiv.

Hier im Beispiel wird zuerst die Länge von `a` berechnet. Die Instanzvariablen `x`, `y` und `z` von `laenge` haben bei dieser Berechnung die Werte 3, 4 und 0. Beim zweiten Aufruf wird die Länge des Nullvektors berechnet. `x`, `y` und `z` haben dabei jeweils den Wert 0.

Statisch oder nicht: eine philosophische Frage

Die Frage, ob eine bestimmte Funktionalität als statische oder nicht statische Methode implementiert wird, ist eine rein "philosophische" Frage. Die Länge hätte genauso gut als statische Methode definiert werden können:

```
public final class Vektor {
  ...
  // statische Variante der Laenge
```

```
  //
  public static double laenge (final Vektor v) {
    return Math.sqrt(v.x*v.x + v.y*v.y + v.z*v.z);
  }
  ,,,
}
```

An den Verwendungsstellen muss dann der Vektor, für den die Länge berechnet werden soll, als Parameter übergeben werden:

```
public final class VektorBenutzer {
  public static void main(String[] args) {
    Vektor a = Vektor.neuerVektor(3, 0, 4);
    double l = Vektor.laenge(a); // laenge von a: Aufruf einer statischen Methode
    System.out.println(l);
    System.out.println(Vektor.laenge(Vektor.nullVektor)); // und noch einer
  }
}
```

Damit haben wir die objektorientierte Berechnung der Vektorlänge zurück in eine klassisch funktionale übersetzt. Es sieht etwas anders aus, ist aber im Ergebnis das Gleiche. (Siehe Abbildung 2.2.)

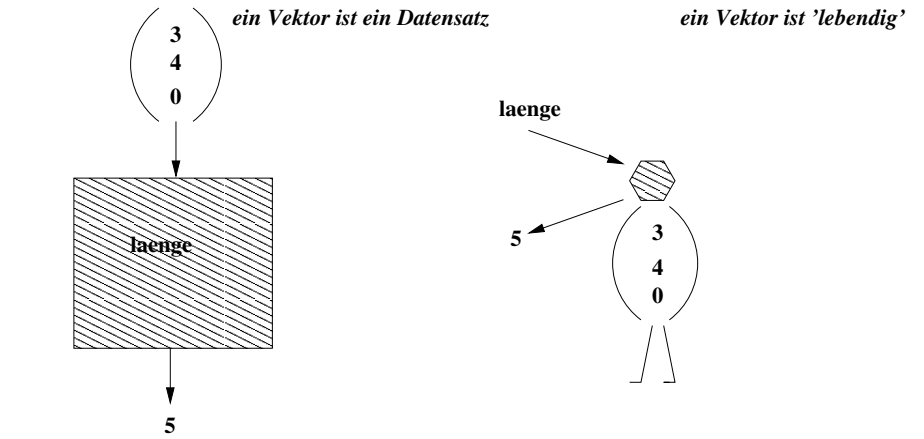


Abbildung 2.2: Methode oder Funktion

Software–Philosophie ist wichtig

Die Frage "Funktion oder Methode" kann nicht klar und eindeutig beantwortet werden, es ist eine rein "philosophische Frage". Die Bedeutung solcher philosophischen Fragen in der Software–Entwicklung wird von Anfängern häufig *unter*schätzt! Wenn es um Objektorientierung geht, dann suchen sie oft nach geheimnisvollen oder komplizierten Mechanismen, die irgendwie "mehr können". Tatsächlich geht es aber in der Softwaretechnik nicht in erster Linie um Mechanismen die mehr oder weniger "können", es geht um eine bessere oder schlechtere Organisation des Programmcodes und "gut" heißt oft "philosophisch richtig" oder schlicht "einfach und natürlich".

Eine Methode kann immer in eine äquivalente statische Methode umgewandelt werden. Man nimmt einfach nur das Objekt für das sie aktiv werden soll als zusätzlichen Parameter an. Aus

```
C o = ...
o.m(x);
```

wird dann schlicht

```
C.m(o, x);
```

Genau das ist es auch, was der Compiler macht wenn das Programm übersetzt wird. Ihn interessieren unsere Philosophien nicht, der Code läuft bestens ohne sie. Trotzdem wurden Millionen von Programmierern auf diese Sicht der Welt umgeschult – ein Aufwand an Milliarden von Euros, die man wohl für gut investiert hält.

Zustand: Von Funktionen zu Methoden die den Zustand verändern

Die Vektoraddition haben wir weiter oben als Funktion (statische Methode) definiert: Zwei Vektoren hinein, einer heraus:

```
public static Vektor add(final Vektor x, final Vektor y) {
  Vektor res = new Vektor();
  res.x = x.x + y.x;
  res.y = x.y + y.y;
  res.z = x.z + y.z;
  return res;
}
```

Die Funktion wird wie eine mathematische Funktion genutzt:

```
Vektor a = new Vektor(3, 0, 4);
Vektor b = new Vektor(1, 2, 3);
Vektor c = Vektor.add(a,b);
```

Nun kann man aber der Meinung sein, dass es nicht richtig ist, eine Vektoraddition *als eine Funktion* zu betrachten. Nach dieser alternativen, objektorientierten Sicht, ist eine Vektoraddition etwas, das *einen Vektor verändert*. Wir haben einen Vektor und wenn wir einen anderen Vektor dazu addieren, dann verändern wir den ersten (Siehe Abbildung 2.3).

Addiert man nach dieser Sicht zu

$$\vec{a} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \textit{ den Vektor } \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$$

dann wird $\vec{a}$ verändert zu

$$\vec{a} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$$

Vektoren sind jetzt nicht für immer und ewig mit einem festen Wert belegt, sie können sich ändern. Mal haben sie diesen und mal haben sie jenen Wert. Sie haben einen veränderlichen *Zustand* und Methoden können den Zustand ändern. Die Addition als zustandsverändernde Methode:

```
public final class Vektor {
  private int x;
  private int y;
  private int z;

  ...

  // Vektoraddition als Methode
  //
  public void add(final Vektor v) {
    x = x + v.x; // zu meinem x addiere das x von v
    y = y + v.y;
    z = z + v.z;
  }
}
```

Mit entsprechend angepassten Aufrufen:

```
a.add(b);    // a!: addiere b zu dir
```

Mit dieser "Addition" wird der Zustand von `a` verändert:

Zustand von `a` *vor* Aufruf der `add`–Methode: $x = 1, y = 2, z = 3$
Zustand von `a` *nach* Aufruf der `add`–Methode: $x = 3, y = 4, z = 5$

Welche Sicht ist die richtige? Ein Mathematiker wird sicherlich einen erneuten Besuch der Grundschule empfehlen, wenn man ihm die Vektoraddition als zustandsverändernde Operation erklären will. Umgekehrt wird ein OO–Dogmatiker die Aberkennung jeglicher Informatik–Qualifikation verlangen, wenn ihm jemand erklärt, dass die Addition von Vektoren etwas ist, bei dem aus zwei gegebenen ein neuer Vektor konstruiert wird.

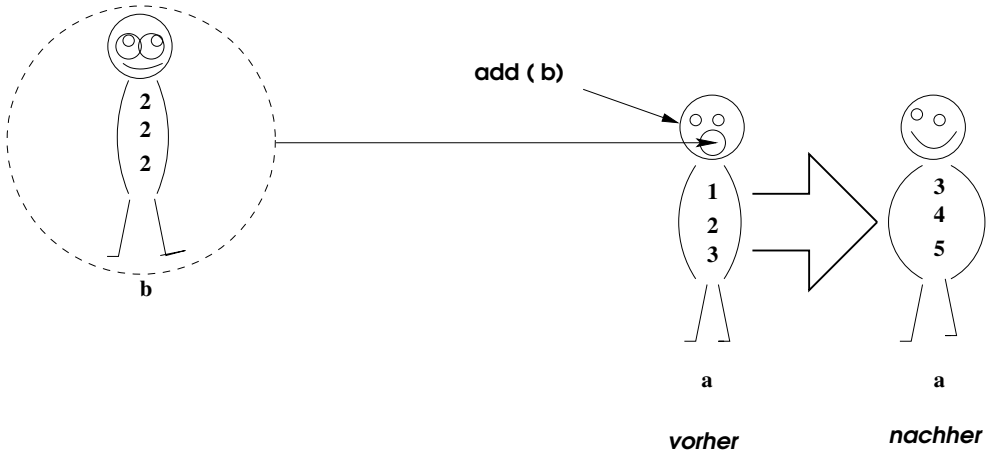


Abbildung 2.3: Vektoraddition als zustandsverändernde Methode

Statische und nicht statische Komponenten mischen

Normalerweise sollte man misstrauisch werden, wenn in einer Klasse statische und nicht statische Komponenten gemischt werden. Der Verdacht liegt nahe, dass die Klasse kein durchdachtes Konzept hat. Manchmal ist eine solche Mischung aber auch angebracht.

Betrachten wir noch einmal das Telefonverzeichnis von oben. Nimmt man konsequent alle `static`–Markierungen weg, dann können beliebig viele Verzeichnisse erzeugt werden:

```
package telefonBuch;

/**
 * Exemplare dieser Klasse repraesentieren Telefonverzeichnisse.
 *
 */
public final class TelefonVerzeichnis {

 public TelefonVerzeichnis() {}

 // Kapazitaet des Verzeichnisses
 private int kapazitaet = 100;

 // Alle Namen, Maximal kapazitaet
 private String[] name = new String[kapazitaet];

 // Alle Nummern, Maximal kapazitaet
 // name[i] ist zugeordnet nummer[i]
 private int[] nummer = new int[kapazitaet];

 //Anzahl der Eintraege
 private int anzahl = 0;

 /**
  * Suche Nummer zu Name
  * @param n der Name
  * @return die zugeordnete Nummer falls vorhanden, sonst -1
  */
 public int suche(String n) {
   for (int i=0; i<anzahl; i++) {
     if (name[i].equals(n)) {
       return nummer[i];
     }
   }
   return -1;
 }
```

```
 /**
  * Erweitere um neuen Eintrag
  * @param n Name
  * @param nr zugeordnete Nummer
  * @pre anzahl < kapazitaet
  * @post n -> nr ist eingetragen falls noch nicht vorhanden
  */
 public void eintrage(String n, int nr) {
  if (suche(n) != -1) { // Eintrag schon vorhanden
   return;
  }
  name[anzahl] = n;
  nummer[anzahl] = nr;
  anzahl = anzahl+1;
 }

 public void loesche(String n) {
  for (int i=0; i<anzahl; i++) {
   if (name[i].equals(n)) {
    for (int j= i; j<anzahl; j++) {
     name[j] = name[j+1];
     nummer[j] = nummer[j+1];
    }
    break;
   }
  }
  anzahl = anzahl -1;
 }

}
```

Ein paar `static`–Markierungen gelöscht und wir haben etwas völlig anderes mit anderer Nutzung:

```
package nutzer;

import telefonBuch.TelefonVerzeichnis;

public final class Nutzer {
 private Nutzer() {}

 public static void main(String[] args) {
  TelefonVerzeichnis tv1 = new TelefonVerzeichnis();
  TelefonVerzeichnis tv2 = new TelefonVerzeichnis();
  tv1.eintrage("Hugo", 4711);
  tv2.eintrage("Karla", 4712);
  tv2.eintrage("Natascha", 4713);
  tv1.loesche("Karla");
  tv1.eintrage("Egon", 4714);

  for (String n : new String[]{"Hugo", "Karla", "Natascha", "Karla", "Egon"}) {
   System.out.println(n + " -> " + tv1.suche(n));
  }

  for (String n : new String[]{"Hugo", "Karla", "Natascha", "Karla", "Egon"}) {
   System.out.println(n + " -> " + tv2.suche(n));
  }

 }

}
```

Angenommen alle Telefonverzeichnisse haben eine gemeinsame Eigenschaft. Beispielsweise die Notrufnummer. Diese gemeinsame Eigenschaft muss nur einmal gespeichert werden: machen wir sie statisch:

```
public final class TelefonVerzeichnis {

 public TelefonVerzeichnis() {}

 private static int notruf = 101;
```

```
 /**
  * Setze Notrufnummer fuer alle Telefonverzeichnisse
  * @param nr die Notrufnummer
  */
 public static void setNotruf(int nr) {
   notruf = nr;
 }

 ... etc. ...

 /**
  * Suche Nummer zu Name
  * @param n der Name
  * @return die zugeordnete Nummer falls vorhanden, sonst -1
  */
 public int suche(String n) {
   if (n.equals("Notruf")) {
     return notruf;
   }
   for (int i=0; i<anzahl; i++) {
     if (name[i].equals(n)) {
       return nummer[i];
     }
   }
   return -1;
 }

 ... etc. ...
}
```

Die Suche nach der Notrufnummer liefert in jedem Verzeichnis die selbe Nummer. Aus einer nicht–statischen Methode kann durchaus auf eine statische Variable zugegriffen werden. – Umgekehrt geht es allerdings nicht. Statische Methoden haben naturgemäß keinen Zugriff auf nicht–statisches.

Insgesamt sehen wir, dass es durchaus sinnvoll sein kann statische und nicht–statische Elemente zu mischen. Das entscheidende Kriterium ist immer ob etwas zur Klasse insgesamt oder zu einzelnen Exemplaren gehört.

statisch:	Eigenschaften und Fähigkeiten die alle Exemplare gemeinsam haben.
nicht statisch:	Eigenschaften und Fähigkeiten die jedes Exemplar individuell hat.

2.1.5 Typen, Werte, Objekte und Referenzen

Primitive– und Klassentypen

Typen sind dazu da, um Variablen anzulegen. Wenn `T` ein Typ ist, dann wird mit

```
T t;
```

eine Variable vom Typ `T` angelegt. Ein Typ kann ein primitiver Typ oder ein Klassentyp[5] sein. Ein *primitiver Typ* ist so etwas wie `int`, `char`, `boolean` und so weiter. Die primitiven Typen sind vordefiniert, d.h. sie sind fest in die Sprache eingebaut. Sie beinhalten einfache (primitive) Werte. Wenn der Typ `T` dagegen eine Klasse ist, dann wird eine Variable mit einem *Klassentyp* angelegt.

```
int i;          // i hat primitiven Typ
double x;       // x hat primitiven Typ
Vektor v;       // v hat Klassentyp
```

Werte und Objekte

Eine Variable mit einem primitiven Typ enthält immer einen Wert von genau diesem Typ, auch dann, wenn die Variable nicht initialisiert wurde.

[5] Neben primitiven– und Klassentypen gibt es noch Interface– und Array–Typen auf die wir später gesondert eingehen werden.

```
int i = 17;     // i enthaelt 17
int j;          // j enthaelt 0
```

Variablen mit einem Klassentyp enthalten *Referenzen*. Eine Referenz ist entweder `null`, oder sie verweist auf ein *Objekt*. Ist `T` eine Klasse und `t` eine Variable vom Typ `T`

```
T t;
```

dann enthält `t` immer entweder `null` oder eine Referenz auf ein Objekt der Klasse `T` oder einer ihrer Unterklassen.

Werte und Objekte bei Vergleich, Zuweisung und Parameterübergabe

Es macht durchaus einen Unterschied, ob eine Referenz auf ein Objekt oder ein einfacher Wert in einer Variablen enthalten ist: Durch den Zugriff über Referenzen werden Objekte als *Individuen* behandelt, einfache Werte dagegen nicht.

Den Unterschied sehen wir beim Vergleich, bei der Zuweisung und bei der Parameterübergabe. Nehmen wir den Vergleich. Objekte werden auf Identität geprüft. Mit

```
Vektor a = Vektor.neuerVektor(1, 2, 3);
Vektor b = Vektor.neuerVektor(1, 2, 3);
```

liefert darum der Test

```
a == b // IST FALSE: a und b sind nicht IDENTISCH
```

den Wert `false`! `a` und `b` sind gleich, aber nicht dieselben. Der Test auf *Identität*, der hier ausgeführt wird, liefert darum `false`. Mit

```
int a = 1;
int b = 1;
```

liefert

```
i == j // IST TRUE: a und b sind GLEICH
```

dagegen den Wert `true`, denn `i` und `j` haben den gleichen Wert und `i == j` testet hier auf Gleichheit.

Eine äquivalente Unterscheidung wird bei der Zuweisung und der Parameterübergabe gemacht. Bei der Zuweisung eines Objektes sind die beiden anschließend nicht nur gleich, sondern sogar dieselben. Bei der Übergabe eines Objekts wird das Individuum weitergegeben.

Werte und Objekte im Speicher

Die unterschiedliche Behandlung von Objekten und Werten ist zunächst einmal verwirrend. Es handelt sich dabei aber nicht um eine spezielle Eigenschaft von Java. Auch andere Sprachen machen diesen Unterschied. Kennt man die interne Behandlung der Objekte und der einfachen Werte, dann kann man den Unterschied auch leicht verstehen.

Wird eine Variable mit einem einfachen Typ definiert, dann enthält sie *immer* einen Wert mit diesem Typ. Auch dann, wenn sie nicht explizit im Programm initialisiert wurde.

Variablen mit einer Klasse als Typ haben als (direkten) Wert entweder eine *Referenz* oder `null`. Eine Referenz ist ein Verweis auf ein Objekt und `null` ist eine besondere Referenz, die auf nichts verweist. Betrachten wir ein Beispiel:

```
int   i = 3;
boolean b = false;
Vektor v = null;
Vektor w = Vektor.neuerVektor(1, 2, 3);
Vektor z = Vektor.neuerVektor(1, 2, 3);
```

Nach der Ausführung dieser Anweisungen

- enthalten `i` und `b` die Werte 3 und $false$,
- `v` enthält die `null`–Referenz und
- `w` enthält eine Referenz, die auf ein Objekt verweist, das selbst drei Int–Variablen `x`, `y` und `z` umfasst, deren Werte 1, 2 und 3 sind.
- `z` enthält eine Referenz auf ein Objekt, das genauso aussieht, wie das, auf das die Referenz in `w` verweist.

Werden jetzt `w` und `z` verglichen, dann ist das ein Test, ob die *Referenzen* gleich sind. Sie sind es nicht, sie zeigen ja auf unterschiedliche Speicherplätze, also gilt

```
w != z
```

Bei einem Vergleich mit == wird also immer das verglichen, was direkt in den Variablen liegt. Liegen dort Werte, werden Werte verglichen, liegen dort Referenzen, werden Referenzen verglichen.

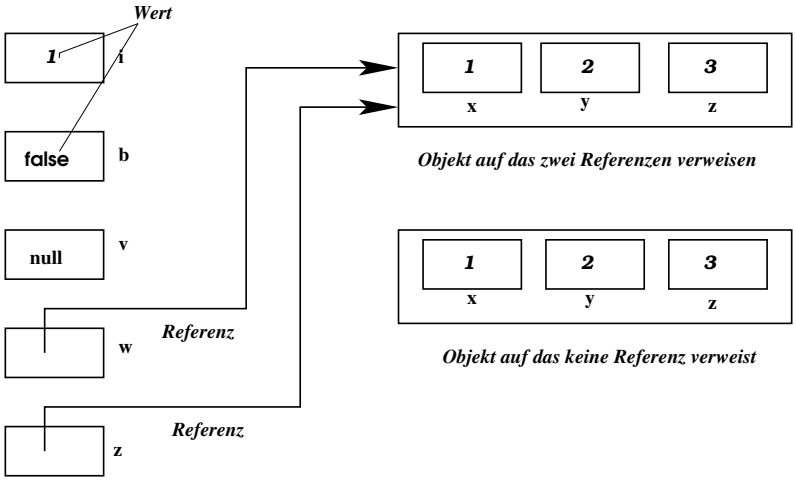


Abbildung 2.4: Objekte und Zuweisungen

Objekte, Zuweisungen und Parameterübergaben

Genau wie der Vergleich, so arbeitet auch die Zuweisung mit dem, was direkt in einer Variablen zu finden ist. Wird nach den Initialisierungen von oben jetzt

```
z = w;
```

ausgeführt, dann zeigen `z` und `w` auf dasselbe (nicht nur das gleiche) Objekt. (Siehe Abbildung 2.4.) Jetzt gilt natürlich `w == z`. Beide Variablen verweisen ja auf dasselbe Objekt.

Bei der Übergabe eines Objektes an eine Funktion passiert das gleiche wie bei einer Zuweisung. Nicht das Objekt wird transferiert, sondern nur die Referenz. So kommt es, dass Veränderungen, die innerhalb einer Funktion an dem Objekt vorgenommen wurden, nach deren Ende noch gültig sind. Die Funktion arbeitet ja mit dem Original, nicht mit einer Kopie.

Wir sehen jetzt, nach der Einführung von Klassen als Typen, gibt es zwei verschiedene Beziehungen zwischen Variablen und Werten, eine direkte und eine indirekte:

- Bei einfachen Typen ist die Beziehung direkt. Die Werte sind Inhalt der Variablen. Vergleich, Zuweisung und Parameterübergabe arbeiten direkt mit den Werten.
- Haben die Variablen Klassen als Typen, dann ist die Beziehung indirekt. Variablen enthalten Referenzen, Referenzen verweisen auf Objekte. Vergleich, Zuweisung und Parameterübergabe beziehen sich auf Referenzen.

Objekte sind unabhängiger von Variablen als einfache Werte. Viele Referenzen in vielen Variablen können auf das gleiche Objekt verweisen. Umgekehrt kann es auch Objekte geben, auf die keine Referenz verweist. Solche Objekte sind zu nichts mehr nutze, sie sind *Speichermüll*. Auch die Typbindung ist bei Referenzen weniger streng als bei Werten. Eine Variable mit einem primitiven Typ enthält immer einen Wert von genau diesem Typ. Hat die Variable einen Klassentyp, dann kann die in ihr enthaltene Referenz auf ein Objekt der Klasse aber auch auf ein Objekt einer Subklasse verweisen.

2.1.6 Exemplare, Instanzen, Instanzvariablen

Exemplare werden mit `new` erzeugt

Definiert man eine Variable mit einem Klassentyp, beispielsweise

```
Vektor v = null;
```

dann enthält sie eine Referenz, hier in diesem Beispiel ist es die `null`–Referenz, die auf nichts verweist. Sie hat den Typ `Vektor`, enthält aber keinen Vektor. Wollen wir tatsächlich einen Vektor erzeugen, dann benutzen wir `new`:

```
Vektor v = new Vektor();
```

Hier wird, durch `new`, ein Objekt der Klasse `Vektor` erzeugt und in `v` wird eine Referenz auf dieses Objekt abgelegt. Dieses Objekt ist eine *Instanz* oder besser *Exemplar der Klasse* `Vektor`. "Instanz" ist verbreiteter als das bessere "Exemplar". Der Begriff "Instanz" leitet sich von engl. *instance* (Fall, Einzelfall, Beispiel) ab. Die treffendere korrekte deutsche Bezeichnung ist "Exemplar", aber im Widerstreit zwischen Gleichklang, gleicher Bedeutung und gar nicht übersetzen entscheidet man sich in der Informatik hin und wieder für Gleichklang. [6] Also: Variablen haben Typen und enthalten Werte, Objekte haben Klassen. Jede Variable hat einen Wert. Objekte werden mit `new` erzeugt. Der Ausdruck "`new K()`" erzeugt ein Objekt, das ein Exemplar (eine Instanz) der Klasse *K* ist. Oder weniger holprig ausgedrückt:

```
new K()
```

erzeugt ein Exemplar der Klasse *K*.

Typen und Klassen

Variablen haben einen Typ, Objekte haben eine Klasse. Beides ist nicht beliebig kombinierbar. Eine Variable mit einem Klassen–Typ *K* kann, außer `null`, nur Referenzen auf Exemplare von *K* aufnehmen, oder Referenzen auf Exemplare von Klassen, die zu *K* passen.

```
Vektor v = null;             //OK
Vektor v = new Vektor();     //OK
Vektor v = new Matrix();     //nicht OK
```

Hier haben wir angenommen, dass `Matrix` nicht zu `Vektor` passt, dass also nicht jede `Matrix` gleichzeitig auch ein `Vektor` ist.

Objektvariable

Jedes Exemplar der Klasse `Vektor` mit der Definition

```
public final class Vektor {
  private int x; // interne Darstellung:
  private int y; // die Komponenten
  private int z; // eines JEDEN Vektors

  public static Vektor neuerVektor(int i, int j, int k) {...}
  public static Vektor add(Vektor x, Vektor y) {...}
  public static Vektor skalarMult(int x, Vektor v) {...}
}
```

hat ihr *eigenes* `x`, `y` und `z`. Gibt es zu irgendeinem Zeitpunkt im Programmlauf drei Vektoren (d.h. drei Exemplare der Klasse `Vektor`), dann gibt es jeweils auch `x`, `y` und `z` dreimal.

Die Variablen `x`, `y` und `z` sind an die Objekte der Klasse gebunden. Man nennt sie darum *Objektvariablen* oder auch *Instanzvariablen* gelegentlich auch *Felder* oder *Attribute*.

Objektvariablen sind in der Regel privat.[7] Die Tatsache, dass sie Objektvariablen sind, hat aber nicht direkt etwas mit `public` und `private` zu tun. Wir hätten sie auch als `public` deklarieren können. Sie wären dann auch außerhalb der Klasse Vektor zugreifbar, aber immer noch Objektvariablen mit einer an ein Objekt geknüpften Existenz:

```
public final class Vektor {
  public int x; // interne Darstellung:
  private int y; // die Komponenten
  private int z; // eines Vektors
  ....
}
```

[6] "Instanz" und *"instance"* sind mit dem schönen deutschen Wort "stanzen" verwandt. Eine Instanz ist ein mit der Klasse als Stanze produziertes Exemplar. (Hinweis von Oliver Correll - danke)

[7] Eigentlich sind sie immer private. Man braucht schon sehr gute Gründe um ohne Ehrverlust von dieser Regel abweichen zu können.

```
...
public final class VektorBenutzer {
 ...
 public static void main(String[] args) {
    Vektor v = new Vektor();
    Vektor w = null;
    v.x = 1;   // OK -- das x von v sei 1
    v.y = 0;   // Fehler -- y ist private
    x = 0;     // Fehler -- x gibt es nicht
    Vektor.x = 0; // Fehler -- auch die Klasse Vektor hat kein x
    w.x = 0;   // Fehler -- w enthaelt keine Referenz die auf
    ...        //        ein Exemplar von Vektor verweist
 }
 ...
}
```

Man beachte, dass auch `Vektor.x` nicht erlaubt ist. `x` gehört zu *Objekten* der Klasse `Vektor`, nicht zur *Klasse selbst*. Jeder Vektor hat sein eigenes `x`, so wie jede Katze ihren eigenen Schwanz hat.

Klassenvariablen, statische Variable

Eine in einer Klasse definierte Variable ist eine *Klassenvariable*, wenn wir `static` davor schreiben. Beispiel:

```
public final class Vektor {
  public int x; // Objektvariablen
  private int y;
  private int z;

  // eine Klassenvariable:
  //
  public static Vektor nullVektor = neuerVektor(0,0,0);

  ...
}
```

Hier ist `nullVektor` eine Klassenvariable der Klasse `Vektor`. Sie gehört zur Klasse, dem Konzept der Vektoren, nicht wie `x`, `y` und `z`, die zu jeweils einem bestimmten Vektor gehören.

In den Benutzern (Programmcode und die die ihn schreiben!) der Vektoren wird die Zugehörigkeit von `nullVektor` zur Klasse dadurch zum Ausdruck gebracht, dass man den Variablennamen mit den Klassennamen qualifiziert.

- Zugriff auf Objektvariable: <*Referenz auf das Objekt*> **.** <*Objektvariable*>
- Zugriff auf Klassenvariable: <*Klasse*> **.** <*Klassenvariable*>

Im Beispiel:

```
...
public final class VektorBenutzer {
 ...
 public static void main(String[] args) {
    int i;
    Vektor v = new Vektor();
    Vektor w = null;
    v.x = 0;         // OK -- das x von v
    w = Vektor.nullVektor;// OK -- der in Vektor definierte nullVektor
    i = Vektor.x;    // Fehler -- das x von Vektor gibt es nicht
    ...
 }
 ...
}
```

Statt Klassenvariable sagt man oft auch *statische Variable*. Damit soll zum Ausdruck gebracht werden, dass die Existenz einer solchen Variablen völlig unabhängig vom Programmlauf ist. Das dynamische Verhalten eines Programms spielt keine Rolle. Es gibt sie genau einmal, was immer zur Laufzeit auch passieren mag. Ihre Existenz ist statisch. (Siehe Abbildung 2.5.)

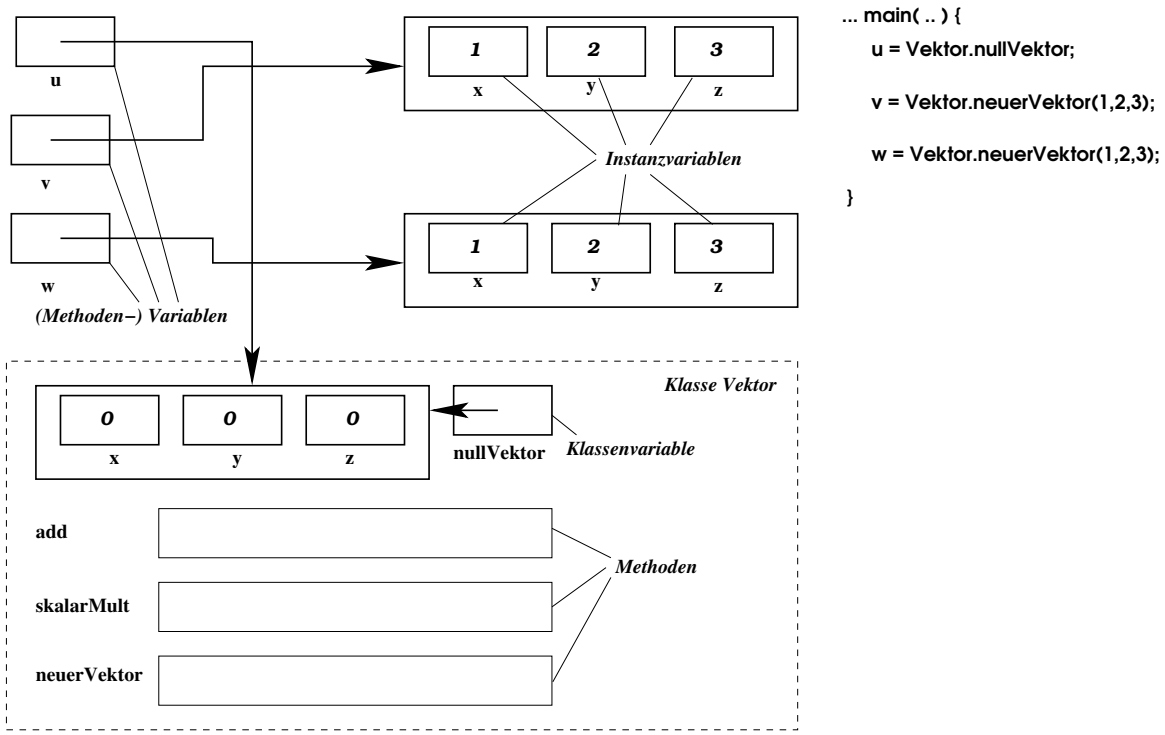


Abbildung 2.5: Klassen– und Objektvariablen

`final`: Variablen zu Konstanten machen

Ein kleines Problem hat unsere Klasse mit dem NullVektor noch. Der NullVektor muss öffentlich sein, damit er außerhalb der Klasse benutzbar ist. Damit ist es aber möglich, dass ein Benutzer folgendes schreibt:

```
Vektor.nullVektor = Vektor.neuerVektor(5, 2, 4);
```

Das mag aus Boshaftigkeit oder Unwissenheit passieren. Es ist in jedem Fall unerwünscht. Glücklicherweise kann es verhindert werden. Wir definieren die zu schützende Variable als `final`:

```
public final class Vektor {
  ...
  // final: Variable wird zur Konstanten
  //
  public static final Vektor nullVektor = neuerVektor(0,0,0);
  ...
}
```

Damit wird `nullVektor` zu einer Konstanten. Alle Zuweisungen an `nullVektor` lehnt der Compiler jetzt als illegal ab.

Der Einsatz von `final` ist nicht auf Klassenvariablen beschränkt. Alle Variablen können damit zu Konstanten gemacht werden. Aber Achtung: `final` bezieht sich auf eine Variable und deren Inhalt. Wenn der Inhalt eine Referenz ist, dann kann *die Referenz* nicht geändert werden. Damit ist aber nichts darüber ausgesagt, ob das, worauf die Referenz zeigt, verändert werden kann oder nicht.

In unserem Beispiel setzen wir voraus, dass Vektoren nicht verändert werden können. Ist das nicht der Fall, dann hilft auch kein *final*:

```
public final class Vektor {
  private double x;
  private double y;

  public void setX(final double newX) { x = newX; }

  public static final Vektor nullVektor = neuerVektor(0,0,0);
  ...
```

```
}
...
Vektor null = Vektor.nullVektor;
...
Vektor.null = ... // nicht moeglich
null.setX(12345); // moeglich
```

Auf diese Weise lässt sich das vermeintlich Unveränderliche leicht verändern. Wir sehen, der Unterschied zwischen einer Variablen und dem auf das sie verweist, muss immer im Auge behalten werden. Soll beides konstant sein, dann müssen wir es unveränderlich machen:

```
public class Vektor {
  // final: Objekt-Variable werden zur Konstanten
  private final double x;
  private final double y;

  public Vektor(final double x, double int y) { this.x=x; this.y=y; }

  // final: Klassen-Variable wird zur Konstanten
  //
  public static final Vektor nullVektor = neuerVektor(0,0,0);
  ...
}
```

Wären `x` und `y` Referenzvariablen, dann müssten wir unser Augenmerk auch noch auf das lenken, auf das sie zeigen.

2.2 Instrumente der Objektorientierung

2.2.1 Methoden

Überladung

Methoden können genauso wie Funktionen überladen werden. So können wir die beiden Arten der Produktbildung bei Vektoren als Methoden definieren und beiden den Namen `mult` geben:

```
public final class Vektor {
  private int x; private int y; private int z;

  ...

  // Skalarmultiplikation als Methode mult
  //
  public void mult(final int s) {
     x = s*x;
     y = s*y;
     z = s*z;
  }

  // Vektormultiplikation als Methode mult
  //
  public void mult(final Vektor b) {
     x = y*b.z - z*b.y;
     y = -(x*b.z - z*b.x);
     z = x*b.y - y*b.x;
  }
```

Der Aufruf bezieht sich, wie bei Methoden üblich, auf ein bestimmtes Objekt. Die Argumente entscheiden darüber welche Methode aktiviert wird:

```
 Vektor v = ...
 Vektor w = ...
 v.mult(2);
 v.mult(w);
```

Es ist auch erlaubt überladene Methoden und Funktionen zu mischen:

```
public final class Vektor {
  private int x; private int y; private int z;
  ...
  public void mult(final int s) {...}       // Skalarmult als Methode
  public void mult(final Vektor b) {...}    // Vektormult als Methode
  public static Vektor mult(final double x, final Vektor v) {...} // Skalarmult als Funktion
  public static Vektor mult(final double x, final Vektor v) {...} // Vektormult als Funktion
}
```

und in Aufrufen zu verwenden:

```
 Vektor v = ...
 Vektor w = ...
 v.mult(2);       // Skalarmult als Methode
 v.mult(w);       // Vektormult als Methode
 w = Vektor.mult(2,w); // Skalarmult als Funktion
 v = Vektor.mult(v,w); // Vektormult als Funktion
```

Es ist natürlich nicht unbedingt eine besonders gute Idee, die gleiche Funktionalität sowohl als Funktion, als auch als Methode anzubieten.

Gültigkeitsbereiche und Überdeckungen

Die Überladung ist ausschließlich auf (statische oder nicht statische) Methoden beschränkt. In allen anderen Fällen gilt das Verbot von Doppeldefinitionen. In einem *Gültigkeitsbereich* darf ein Name nur einmal definiert werden. So ist es nicht erlaubt,

dass zwei Variablen einer Methode, zwei Parameter oder zwei Objektvariablen den gleichen Namen haben, auch dann nicht, wenn sie unterschiedliche Typen haben.

```
public final class C {
 int x; double x;             // Verboten: Doppeldefinition

 void f(final int x, final double x) { // Verboten: Doppeldefinition
   int x; double x;           // Verboten: Doppeldefinition
 }
}
```

Erlaubt ist allerdings die *Überdeckung* eines Namens durch eine Definition "weiter innen". So deckt beispielsweise eine lokale Variablendefinition einen Parameter ab:

```
 void f(final int x) {
   double x;
   ... x .... // das lokale double x / der Parameter x ist nicht mehr sichtbar
 }
```

und Parameter oder lokale Variablen decken eine Instanzvariable ab:

```
public final class C {
 int x;

 void f(final double x) {
   ... x .... // der Parameter double x
 }
}
```

this: dieses; das Objekt für das eine Methode aktiv ist

Auf Instanzvariablen kann stets explizit zugegriffen werden. Man gibt das Schlüsselwort `this` an:

```
public final class C {
 int x;

 void f(final double x) {
   ... x     .... // der Parameter double x
   .... this.x .... // die Instanzvariable int x
 }
}
```

Mit `this.x` ist *das aktuelle* `x` gemeint, also *das* `x`, das zu dem Objekt gehört, dessen Methode `f` gerade aktiv ist. In

```
 C c = new C;
 c.f(0.45);
```

bezieht sich `this` während der Ausführung von `f` auf `c`.

Bei der Verwendung einer nicht abgedeckten Instanzvariablen wird `this` vom Compiler immer automatisch hinzugefügt. D.h.

```
public final class C {
 int x;

 void f() {
   ... x ....
 }
}
```

ist das Gleiche wie

```
public final class C {
 int x;

 void f() {
   ... this.x ....
 }
}
```

Mit einem expliziten `this` können Abdeckungen zurückgenommen werden, z.B. wenn Parameter und Objektvariablen den gleichen Namen haben. Man kann es aber, im Sinne der besseren Lesbarkeit des Codes, auch dann einsetzen, wenn seine Verwendung, technisch gesehen, überflüssig ist:[8]

```
public class C {
 int x;

 void setX(int pX) {
   this.x = pX;  // entspricht x = pX;
 }
}
```

2.2.2 Konstruktoren

Objektinitialisierung

Objekte enthalten Objektvariablen, die am besten sofort nach der Erzeugung des Objekts mit sinnvollen Werten belegt werden. In unserem Vektorbeispiel haben wir dazu eine (statische) Methode `public static neuerVektor` definiert, die einen neuen Vektor erzeugt und dessen Objektvariablen initialisiert:

```
public final class Vektor {
  private int x;
  private int y;
  private int z;

  // Initialisierungs-FUNKTION
  public static Vektor neuerVektor(final int i, final int j, final int k) {
       Vektor res = new Vektor();
       res.x = i;
       res.y = j;
       res.z = k;
       return res;
  }
  ....
}
```

mit Aufrufen wie:

```
Vektor v = Vektor.neuerVektor(1, 2, 3);
```

werden dann Vektor–Objekte erzeugt. Diese Methode ist eine Funktion: man gibt drei Int–Werte hinein und heraus kommt ein Vektor. Das objektorientierte Äquivalent einer solchen Funktion wäre eine Initialisierungs–*Methode* – nennen wir sie `init` –, die für ein bestimmtes Objekt aufgerufen wird und dieses mit Werten belegt:

```
public final class Vektor {
  private int x;
  private int y;
  private int z;

  // Initialisierungs-Methode
  public void init(final int i, final int j, final int k) {
       x = i;
       y = j;
       z = k;
  }
  ....
}
```

Das Objekt muss natürlich existieren bevor es initialisiert werden kann. Der Vektor–Anwender muss darum `new` aufrufen:

```
Vektor v = new Vektor(); // Vektor erzeugen
v.init(1,2,3);
```

[8] manche empfehlen, im Sinne eines guten und klaren Stils, `this` immer zu verwenden.

Konstruktor: Objektinitialisierer mit Compilerunterstützung

Die Erzeugung von Objekten und die unmittelbar folgende Initialisierung kommt nicht nur sehr häufig vor, es ist auch sehr empfehlenswert, jedes Objekt sofort mit sinnvollen Werten zu belegen. Für die Initialisierung von Objekten gibt es darum ein eigenes Sprachelement: den *Konstruktor*.

Ein Konstruktor wird als Methode besonderer Art definiert: Sie heißt so wie die Klasse und hat keinen Rückgabetyp. Ein Konstruktor für Vektoren kann beispielsweise so aussehen:

```
public final class Vektor {
  private int x; private int y; private int z;

  // Konstruktor
  //
  public Vektor(final int i, final int j, final int k) {
    x=i; y = j; z = k;
  }
  ...
}
```

Die Erzeugung und Initialisierung eines Vektors vereinfacht sich mit dem Konstruktor zu:

```
  Vektor v = new Vektor(1,2,3); // Vektor erzeugen und initialisieren
```

Durch diese einfache Konvention der Benennung (heißt wie die Klasse) kann also ein Konstruktor vom Compiler erkannt werden. Er nutzt sein Wissen, um an allen Stellen, an denen ein neues Exemplar der Klasse erzeugt wird, einen Aufruf des Konstruktors einzufügen.

Überladung: Eine Klasse kann mehrere Konstruktoren definieren

Manchmal sollen die Objekte einer Klasse nicht immer auf die gleiche Art initialisiert werden. Es ist darum möglich mehrere Konstruktoren zu definieren, von denen dann bei der Objekterzeugung der passende ausgewählt wird. Wir definieren als Beispiel einen zweiten Konstruktor für Vektoren. Der neue Konstruktor soll einen Vektor als Kopie eines anderen erzeugen. Mit ihm hat unsere Klasse zwei Konstruktoren:

```
  ....
  public Vektor(final int i, final int j, final int k) { // Konstruktor 1
    x=i; y = j; z =k; // Vektor mit Komponentenwerten initialisieren
  }
  public Vektor(final Vektor v) {           // Konstruktor 2
    x=v.x; y=v.y; z=v.z; // Vektor als Kopie eines anderen initialisieren
  }
  ....
```

Bei einer Objekterzeugung ruft der Compiler den Konstruktor auf, der zu der `new`–Anweisung passt:

```
  Vektor v = new Vektor(1,2,3); // Vektor erzeugen, mit Konstr. 1 initialisieren
  Vektor w = new Vektor(v); // Vektor erzeugen, mit Konstr. 2 initialisieren
```

Ein Konstruktor “passt”, wenn die aktuellen Argumente zu den formalen Parametern passen.

Wir sehen, Konstruktoren können genau wie andere Methoden *überladen* werden.

Der Standard–Konstruktor

Der Konstruktor ohne Parameter wird *Standardkonstruktor* (engl. *default constructor*) genannt. Als Standardkonstruktor für unsere Vektoren könnten wir definieren

```
public final class Vektor {
  int x, int y, int z;

  public Vektor() { // Standard-Konstruktor
    x=0; y = 0; z =0;
  }
  ....
}
```

Wenn in einer Klassendefinition kein Konstruktor angegeben wird (und "kein" meint hier gar– und überhaupt kein), dann erzeugt der Compiler einen Standardkonstruktor. Sobald irgendein Konstruktor definiert wird, erzeugt der Compiler *nicht* mehr selbständig einen Standardkonstuktor. Deshalb ist mit

```
public final class C {
  private int x;
}
```

der Aufruf

```
C c = new C(); // OK Standardkonstruktor automatisch erzeugt
```

korrekt, aber mit

```
public final class C {
  private int x;
  public C(int y) { x = y; }
}
```

ist es nicht mehr erlaubt den Standardkonstruktor zu verwenden und

```
C c = new C(); // FEHLER Standardkonstruktor wurde nicht erzeugt
```

ist somit nicht mehr korrekt.

Definiert man nur einen Konstruktor und macht ihn `private`, dann ist damit die Erzeugung von Exemplaren ausgeschlossen.[9]

Konstruktoraufrufe

Es ist nicht möglich einen Konstruktor wie eine Methode für ein explizit angegebenes Objekt aufzurufen, auch dann nicht, wenn man das Objekt selbst (`this`) ist:

```
public final class Vektor {
  ...
  Vektor(){...}
  public void methode(final Vektor v){
     v.Vektor(); // FALSCH, VERBOTEN
     this.Vektor(); // FALSCH, VERBOTEN
     ...
  }
}
```

Innerhalb eines Konstruktors kann ein anderer Konstruktor aufgerufen werden. Die Sache darf dabei aber nicht direkt oder indirekt zyklisch (rekursiv) werden:

```
public final class Vektor {
  private int x; private int y; private int z;

  Vektor(final int x, final int y, final int z){
    this.x=x; this.y=y; this.z=z;
  }

  Vektor() {
    this(1,1,1); // OK Konstruktoraufruf im Konstruktor
  }

  Vektor(int x){
      this(x-1); // Nicht OK, rekursiver Konstruktor
  }

}
```

Ein expliziter Konstruktoraufruf im Konstruktor ist nur erlaubt, wenn er die *erste Anweisung* ist.

[9] Genau gesagt ist die Erzeugung von Exemplaren auf das Innere der Klasse beschränkt. Benutzer der Klasse können keine Exemplare erzeugen. Ein einziger privater Konstruktor bringt zum Ausdruck, dass die Klasse nicht dazu geeignet ist, Exemplare (Objekte) zu erzeugen; beispielsweise weil sie eine Klasse mit reinem Modul-Charakter ist.

2.2.3 Initialisierungen

Initialisierung von Objektvariablen

Objektvariablen können an der Stelle ihrer Definition mit Werten belegt werden:

```
public final class Vektor {
  private int x = 1; // Objekt- (Instanz-) Variable mit Initialisierung
  private int y = 1;
  private int z = 1;
  ...
}
```

Solche *Initialisierungen von Instanzvariablen* werden behandelt, als würde jeder Konstruktor mit den entsprechenden Zuweisungen beginnen. Mit der Definition

```
public final class Vektor {
  private int x = 1;
  private int y = 1;
  private int z = 1;

  public Vektor() {
     // Compiler ergaenzt: x=1; y=1, z=1;
     x=x+y+z;
     y=x+y+z;
     z=x+y+z;
  }
}
```

erhält die Variable `v` durch

```
v = new Vektor();
```

den Wert (3,5,9) = (1+1+1, 3+1+1, 3+5+1). Die Initialisierung im Konstruktor findet darum nach der Initialisierung in der Definition statt.[10]

Initialisierung von Klassenvariablen

Klassenvariablen können ebenfalls initialisiert werden. Diese Initialisierung findet statt, wenn die Klasse der virtuellen Maschine bekannt wird, also vor jeder Initialisierung von Objektvariablen. [11] Mit

```
public final class Vektor {
  private int x = kv;   // 2. Vor-Initialisierung der Objektvariablen
  private int y = kv;
  private int z = kv;

  private static int kv = 10; // 1. Klassenvariable wird zuerst belegt

  public Vektor() {     // 3. Fertigstellung der Initialisierung
     x=x+y+z;           // von Objektvariablen
     y=x+y+z;
     z=x+y+z;
  }
}
```

erhält die Variable `v` durch

```
v = new Vektor();
```

den Wert (30,50,90) = (10+10+10, 30+10+10, 30+50+10).

[10] Eigentlich werden beide Initialisierungen als Teil des Konstruktors ausgeführt, entscheidend ist aber die Reihenfolge und da merkt man sich am besten: erst Zuweisung in der Definition, dann Zuweisung im Konstruktor.

[11] Klassen sind nichts anderes als eine besondere Art von Objekten. Sie werden von der virtuellen Maschine in den Speicher geladen und dann initialisiert. Für's erste und unsere einfachen Programme können wir aber davon ausgehen, dass alle Klassen zu Beginn einer Programmausführung geladen und initialisiert werden. Tatsächlich kann der Prozess des Klassenladens aber sehr komplex werden.

Objektinitialisierer

Die Initialisierung einer Objektvariablen kann auch in einen *Objektinitialisierer* gepackt werden:

```
public final class Vektor {
  private int x;
  private int y;
  private int z;

  { // Objektinitialisierer
    x = 1;
    y = 1;
    z = 1;
    System.out.println("Objekt fertig initialisiert");
  }
  ....
}
```

Das entspricht einer Initialisierung der Objektvariablen:

```
public final class Vektor {
  private int x = 1;
  private int y = 1;
  private int z = 1;
  ....
}
```

Der Vorteil ist die größere Flexibilität. Man kann beliebigen Initialisierungcode hinschreiben, der dazu noch andere Dinge tun kann. Beispielsweise einen Text ausgeben.

Klasseninitialisierer

In der gleichen Weise können Klassenvariablen durch einen *Klasseninitialisierer* initialisiert werden:

```
public final class Vektor {
  private int x;
  private int y;
  private int z;

  // Klasseninitialisierer (statischer Initialisierer)
  // Wird einmal bei der Klasseninitialisierung ausgefuehrt
  static {
     nullVektor = new Vektor(0,0,0);
     System.out.println("Klassenvariable fertig initialisiert");
  }

  { // Objektinitialisierer
    // Wird als erster Bestandteil jedes Konstruktors ausgefuehrt
     x=1; y=1; z=1;
     System.out.println("Objektvariable "+this+" vorinitialisiert");
  }
}
```

Dieses Beispiel enthält den interessanten Fall, dass die Klasseninitialisierung einen Konstruktoraufruf enthält. Im Allgemeinen gilt, dass Klasseninitialisierer vor allen Objektinitialisierungen ausgeführt werden. Enthält der Klasseninitialisierer einen Konstruktoraufruf, dann kann diese Regel natürlich nicht mehr gelten. Der Nullvektor wird hier folglich mit einem Konstruktor ausgeführt, der vor Beendigung der Klasseninitialisierung abläuft. Das ist hier kein Problem, kann aber unter Umständen beachtenswert sein.

Redundante Initialisierung

Initialisierungen sind gut, man kann es aber auch übertreiben. Jede Variable wird zunächst einmal schon bei der Speicherbeschaffung mit ihrem Standard–Nullwert belegt. Darum ist so etwas wie:

```
public final class C {
  private int x = 0;  // ueberfluessiges explizites Nullen
```

```
  private Vektor v = null; // ueberfluessiges explizites Nullen
  ...
}
```

ein überflüssiges doppeltes Initialisieren mit Null. Ebenso ist es unsinnig in einem Konstruktor einen Wert zu setzen, der schon vorher gesetzt wurde:

```
public final class C {
  private int x = 1;
  C() {
    x = 1; // ueberfluessig, ist schon 1
  }
}
```

Oder besonders redundantes Initialisieren:

```
public final class C {
  private int x = 0; // ueberfluessig: ist schon Null
  C() { x = 0; } // ueberfluessig: ist schon 2-mal Null
  ...
}
```

Initialisierung von Feldern

Felder werden in Java als Objekte behandelt. Nach der Definition

```
Vektor[] a; // a == null
```

enthält `a` zunächst den Wert `null`. Erst mit einem `new`

```
a = new Vektor[5]; //a[0] == null ... a[4] == null
```

wird Speicherplatz für ein Feld reserviert und `a` enthält jetzt eine Referenz auf dessen Speicherplatz. Wir müssen allerdings beachten, dass die einzelnen Feldelemente damit noch keine Vektoren, sondern nur die `null`–Referenz enthalten. Erst mit Zuweisungen wie

```
 for ( int i = 0; i < 2; i++ ) {
     a[i] = Vektor.nullVektor;
 }
```

werden die Feldelemente mit Vektoren belegt.

Auch wenn die Feldelemente einen Klassentyp haben, können sie mit expliziten Werten initialisiert werden:

```
// Vektor-Feld mit Initialisierer erzeugen und belegen
//
Vektor[] b = { Vektor.nullVektor,
               new Vektor(),
               new Vektor(1,2,3),
               null,
               Vektor.add(Vektor.nullVektor, new Vektor(4,5,6))
             };
...
```

2.2.4 Speicherverwaltung

Speicherplatz für Variablen und Objekte

Ein Java–Programm benötigt Speicherplatz um ablaufen zu können. Dieser Platz wird ihm nicht von Anfang an als Block fester Größe zugeteilt. Die Zuteilung erfolgt nach Bedarf (dynamisch). Immer wenn das Programm in seinem Lauf mehr Platz benötigt, wird ihm dieser von der Speicherverwaltung zugeteilt. Dabei müssen zwei Arten von Speicherverbrauchern bedient werden: Platz der für die Ausführung von Methoden gebraucht wird und Platz für neue Objekte.

Platz für Methoden und ihre lokalen Variablen und Parameter: Jedesmal, wenn ein Gültigkeitsbereich betreten wird, dann wird Platz für seine lokalen Variablen gebraucht. Wird beispielsweise die Methode `add`

```
public final class Vektor {
  ...
  public static Vektor add(final Vektor x, final Vektor y) {
    Vektor res = new Vektor();
    res.x = x.x + y.x;
    res.y = x.y + y.y;
    res.z = x.z + y.z;
    return res;
  }
  ...
}
```

aus der Klasse Vektor aufgerufen, beispielsweise mit:

```
u = Vektor.add(v, w);
```

dann wird *Platz* für die Parameter `x` und `y` und die lokale Variable `res` benötigt.

Platz für Objekte und ihre Objektvariablen: Speicherplatz für Objekte wird dagegen benötigt und bereitgestellt, wenn `new` ausgeführt wird. Das sind zwei unterschiedliche Aktionen, auch wenn eine einzige Variable involviert ist. In unserem Beispiel wird Platz für die Variable `res` angelegt, in dem Augenblick, in dem `add` aktiv wird. Zusätzlich wird Platz für ein Vektor–Objekt angelegt, wenn `res` innerhalb von `add` belegt wird:

```
res = new Vektor();
```

Wird `add` verlassen, kann der Speicherplatz für die Variablen `x`, `y` und `res` freigegeben werden. Der Speicherplatz für das neu erzeugte Objekt, auf das die Referenz in `res` verweist, kann aber keineswegs freigegeben werden, wenn `add` zu Ende ist. Zwar ist `res` weg, aber in `u` gibt es immer noch eine Referenz auf das Objekt.

Die Speicherverwaltung der Variablen ist einfach, die der Objekte schwierig:

- Speicherplatz für *Variablen* wird angelegt und entfernt mit Betreten und Verlassen des Konstruktes (Gültigkeitsbereichs), in dem sie definiert sind.
- Speicherplatz für *Objekte* wird angelegt, wenn `new` ausgeführt wird. Die Lebensdauer dieses Speicherplatzes ist unbestimmt.

Stack und Heap

Entsprechend ihrem unterschiedlichen Charakter ist die Verwaltung des Speichers zweigeteilt in Stack und Heap:

- Stack: Speicherregion für lokale Variablen und Parameter
- Heap: Speicherregion für Objekte inklusive ihrer Objektvariablen.

Der *Stack (Stapel)* ist eine Speicherregion, in der die Variablen verwaltet werden. Mit jedem Eintritt in einen Gültigkeitsbereich (Methode, Block) wächst er um die Variablen des Bereichs, beim Verlassen schrumpft er entsprechend. Das entspricht einem Stapel. Man legt Dinge auf den Stapel und nimmt sie in umgekehrter Reihenfolge wieder weg.

Der *Heap (Haufen)* hat dagegen ein völlig ungeordnetes Verhalten. Mit jedem `new` werden dort Objekte angelegt. Entfernen kann man sie dann, wenn keine Referenz mehr auf sie verweist. Wann das ist, ist nicht vorhersehbar. Referenzen können ja beliebig zugewiesen und damit in beliebige Gültigkeitsbereiche transportiert werden.

Der Garbage Collector

Um Speicherplatz freigeben zu können, der von Objekten belegt ist, müssen systematisch alle Referenzen verfolgt und die entsprechenden Objekte markiert werden. Objekte, die dabei nicht markiert wurden, können weggeräumt werden. Genau das macht der *Garbage Collector* (Müllsammler). In regelmäßigen Abständen, oder wenn das Programm knapp an Speicher ist, wird er aktiv um im Heap aufzuräumen. Dabei werden alle Objekte entfernt, die nicht mehr direkt oder indirekt von lokalen Variablen oder Parametern aus über Referenzen zu erreichen sind.

In einigen Sprachen gibt es keine automatische Speicherfreigabe durch einen *Garbage Collector*. Das erschwert das Schreiben korrekter Programme erheblich, kann Laufzeit die Programme aber deutlich steigern und macht so vor allem vorhersehbar: Der Programmierer muss darauf gefasst sein, dass der *Garbage Collector* den Programmlauf jederzeit unterbrechen kann, weil gerade mal Lust zum Aufräumen hat.

2.2.5 Pakete

Klassen als Module und Typen passen oft zusammen

Klassen haben in Java einen doppelten Charakter, einen Typ und einen Modulcharakter. Mit ihnen kann man neue Typen definieren, gleichzeitig können sie dazu eingesetzt werden, um Zusammengehöriges zusammen zu fassen. Das passt oft zusammen. Der Nullvektor und die Vektormultiplikation passen bestens zusammen mit der Definition des Typs Vektors.

```
public final class Vektor { // Typ und Modul
  int x; int y; int z; // Typ-Info zum Typ Vektor
  ...
  public static final Vektor nullVektor = ... // Bestandteil des Moduls Vektor
  public double laenge() {...}     // Bestandteil eines Objekts
  public static Vektor mult (..){...} // Bestandteil des Moduls Vektor
  ...
}
```

Pakete als Subsysteme: Kollektionen von Klassen

Die doppelte Funktion des Klassenkonstrukts passt oft aber nicht immer und für alles. Gelegentlich benötigt man eine größere Flexibilität und Unabhängigkeit in der Gestaltung seiner Module, also in dem, was man als zusammengehörig kennzeichnen will.

Nehmen wir an, dass wir es neben den Vektoren noch mit anderen geometrischen Objekte zu haben: mit Punkten, Geraden und so weiter. Zwar gehören Vektoren, Punkte, Geraden, Ebenen irgendwie zusammen, aber es macht wenig Sinn sie in einer einzigen Klasse zusammenzufassen. Es sind unterschiedliche Dinge, die nur schlicht eine gewisse Beziehung zueinander haben. Java bietet ein einfaches Konstrukt, um diese Zusammengehörigkeit

auszudrücken: das *Paket*, (`package`). Ein Paket ist einfach eine Kollektion von Klassen.

Die "Zusammengehörigkeit" von Konzepten ist eine vielschichtige und eventuell recht komplexe Beziehung mit verschiedenen Ausprägungen, die in Java auf verschiedene Art ausgedrückt werden kann. Neben der Paketstruktur eines Programms kann man auch andere Konstrukte einsetzen.[12] Die Verwendung von Paketen ist eine einfache Art Zusammengehöriges zusammen zu fassen. Also:

> Ein Paket ist eine Kollektion von Definitionen, die zu einem Subsystem des gesamten Softwaresystems zusammengefasst werden.

Klassendefinitionen sind nicht die einzigen, aber die wichtigsten Definitionen. Später (siehe Seite 163) werden wir mit den *Schnittstellen* (*Interfaces*) eine weitere Art von Definition kennenlernen. Pakete kann man also benutzen um ein Softwaresystem aus organisatorischen Gründen in *Subsysteme* zu gliedern. Pakete organisieren den Quellcode und den übersetzten Code. In einem laufenden Programm gibt es keine Pakete. Es sei denn indirekt über die Paketzugehörigkeit der Klassen der Objekte die gerade im Speicher liegen.

Pakete definieren

Pakete werden dadurch definiert, dass man einer Klassendefinition[13] eine *Paket–Deklaration* vorausschickt. Sollen also beispielsweise Vektoren, Punkte und Geraden zu einem Paket namens `geometrie` gehören, dann setzen wir einfach vor die jeweiligen Klassen die Paket–Deklaration `package geometrie;`. Beispielsweise:

```
package geometrie;

public final class Vektor { .... }
```

oder:

```
package geometrie;

public final class Punkt { .... }
```

[12] geschachtelte Klassen, Vererbung, Teil–Ganzes–Beziehungen, etc.
[13] allgemeiner: einer Definition

Der Quelltext der zu einem Paket gehört wird in einem Verzeichnis gespeichert. Diese Struktur ist den Java–Werkzeugen nicht nur bekannt, sie setzen sie auch voraus. Die Paketstruktur muss sich also in der Verzeichnisstruktur widerspiegeln. Eclipse unterstützt die Arbeit mit Paketen. Man muss sich also nicht um die Speicherung am richtigen Platz kümmern. Das übernimmt Eclipse.

Pakete und Sichtbarkeit

Bei Klassen haben wir gesehen wie Methoden und Variablen in unterschiedlichem Grad nach außen sichtbar werden können. Eine als `private` definierte Methode kann nur innerhalb der Klasse verwendet werden etc. Auf der Ebene der Klassen gelten ähnliche Regeln für die gesamte Klassendefinition:

Sichbarkeitsqualifizierung von Definitionen auf Paketebene:

- `public`, *öffentliche Sichtbarkeit*:
  Die Definition kann innerhalb und außerhalb des Pakets verwendet werden.
- keine Qualifizierung, *paketweite Sichtbarkeit*:
  Die Definition kann nur innerhalb des Pakets verwendet werden.
- `private`, *private Sichtbarkeit:*
  – nicht erlaubt ! –

Wenn wir also definieren:

```
package geometrie;
public final class Vektor { .... } // oeffentlich

package geometrie;
final class Punkt { .... }  // paketweit

package geometrie;
private final class Gerade { .... } // GEHT NICHT, GIBT'S Nicht, Falsch
```

Dann kann die Klasse `Vektor` überall verwendet werden, die Klasse `Punkt` nur in Klassen die zum Paket `geometrie` gehören und die Definition von `Gerade` ist falsch. Auf Paketebene kann eine Definition also nicht als privat erklärt werden. Das ist logisch: `public` bedeutet "innerhalb und außerhalb des Pakets verwendbar". Kein Qualifizierer bedeutet "nur innerhalb des Pakets verwendbar", was könnte da noch `private` bedeuten.

Sichtbarkeit: Klassensichtbarkeit und Paketsichtbarkeit

Die Sichtbarkeit einer ganzen Klasse auf Paketebene (Paketsichtbarkeit) und die Sichtbarkeit von lokalen Definitionen auf Klassenebene (Klassensichtbarkeit) müssen unterschieden werden. Auf Paketebene haben wir ein zweistufiges Konzept (Paket–lokal, global). Bei Definitionen innerhalb einer Klasse haben wir eine Ebene mehr:

Klassensichtbarkeit: Sichtbarkeitsqualifizierung von Definitionen innerhalb einer Klassen:

- `public`, *öffentliche Sichtbarkeit*:
  Die Definition kann von allen Klassenn innerhalb und außerhalb des Pakets verwendet werden.
- keine Qualifizierung, *paketweite Sichtbarkeit*:
  Die Definition kann nur von Klassen verwendet werden, die zum gleichen Paket gehören.
- `private`, *private Sichtbarkeit:*
  Die Definition kann nur innerhalb dieser Klasse verwendet werden.

Paketsichtbarkeit: Sichtbarkeitsqualifizierung von Definitionen innerhalb eines Pakets:

- `public`, *öffentliche Sichtbarkeit*:
  Die Definition kann in allen Paketen verwendet werden.
- keine Qualifizierung, *paketweite Sichtbarkeit*:
  Die Definition kann nur im gleichen Paket verwendet werden.

Wenn wir also definieren:

```
package geometrie;

public final class Vektor {
  public static Vektor v1 = ... // ueberall
         static Vektor v2 = ... // im Paket
  private static Vektor v3 = ... // in der Klasse
}

package geometrie;
final class Punkt {
  public static Punkt p1 = ... // eigentlich ueberall, defacto im Paket
         static Punkt p2 = ... // im Paket
  private static Punkt p3 = ... // in der Klasse
}
```

dann haben `Vektor` und `Punkt` jeweils eine Paketsichtbarkeit (`public` bzw. *paketweit*). `v1`, ... `p2` haben eine Klassensichtbarkeit.

`v1` kann überall benutzt werden, `v2` nur von einer Klasse, die zum Paket `geometrie` gehört und `v3` nur innerhalb von `Vektor`. Die Klasse `Punkt` kann nur innerhalb des Pakets `geometrie` benutzt werden. Zwar ist `p1` dort als öffentlich deklariert, faktisch kann es wegen der beschränkten Sichtbarkeit von `Punkt` nur innerhalb des Pakets benutzt werden. `p1` und `p2` haben darum im Endeffekt die gleiche Sichtbarkeit.

Das Standard–Paket

Das Standardpaket (engl. *default package*) ist das Paket, in das alle Typdefinitionen platziert werden, die explizit einem anderen Paket zugeordnet werden. Lassen wir die `package`–Deklaration weg:

```
// kein package
public final class Vektor {
  ....
}
```

dann gehört der Typ zum Standardpaket.

In kleinen Programmen macht es oft keinen Sinn, den Quellcode in Pakete aufzuteilen. In dem Fall arbeitet man mit dem Standardpaket. In realistischen Anwendungen macht eine Paketaufteilung aber Sinn. Man sollte sich darum frühzeitig angewöhnen den Code in Pakete aufzuteilen.

Paketfremde Namen zugänglich machen

Ein Name aus einem anderen Paket muss explizit mit dem Namen des Pakets erweitert werden. Haben wir beispielsweise die Defintion der Vektoren im Paket `gemometrie` angesiedelt

```
package geometrie;
public final class Vektor {...}
```

dann müssen sie außerhalb mit dem Paketnamen `geometrie` qualifiziert werden:

```
// wir sind im default package
public final class C {
  public static void main(..) {
    geometrie.Vektor v = new geometrie.Vektor(..);
    geometrie.Punkt p = new geometrie.Punkt(..);
    ....
  }
}
```

Das ist außerordentlich lästig. Glücklicherweise kann es mit der *Import–Anweisung* vereinfacht werden:

```
import geometrie.Vektor; // Vektor wird importiert
import geometrie.Punkt; // Punkt wird importiert

public final class C {
  public static void main(..) {
    Vektor v = new Vektor(..);
```

```
    Punkt p = new Punkt(..);
    ....
  }
}
```

Mit `import paketName.*` kann eine ganze Kollektion von Typen importiert werden. Beispielsweise alles aus dem Paket `geometrie`:

```
import geometrie.*      // alles von geometrie importieren

public final class C {
  public static void main(..) {
    Vektor v = new Vektor(..);
    Punkt p = new Punkt(..);
    ....
  }
}
```

Pakethierarchien

Manchmal reicht eine einstufige Aufteilung der Codes nicht aus. Man hätte es gerne feiner. In Java ist das kein Problem: Pakete können selbst wieder in Sub–Pakete aufgeteilt werden und diese wieder in Sub–Sub–Pakete und so weiter. Ganz nach Belieben. Nehmen wir an, wir schreiben ein komplexes System zur linearen Algebra. Die geometrischen Typen bilden dann nur einen Teil des Subsystems zur linearen Algebra. Im Quellcode machen wir diese Aufteilung deutlich, indem das Paket `geometrie` als Sub–Paket in ein Paket `lineareAlgebra` platziert wird. Den Klassendefinitionen geht dann eine entsprechende Sub–Paket–Deklaration voraus:

```
package lineareAlgebra.geometrie; // Paketdeklaration fuer Sub-Paket

public final class Vektor {
    ...
}
```

An den Verbindungsstellen muss auf die feinere Struktur Rücksicht genommen werden:

```
import lineareAlgebra.geometrie.*

public final class C {
  public static void main(..) {
    Vektor v = new Vektor(..);
    Punkt p = new Punkt(..);
    ....
  }
}
```

In den Import–Anweisungen kann man beliebig fein werden:

```
import lineareAlgebra.*;                oder
import lineareAlgebra.geometrie.*;      oder
import lineareAlgebra.geometrie.Vektor;
```

Es ist allerdings nicht erlaubt einfach alles zu importieren:

```
import *; //SO NICHT!
```

Die Gliederung der Pakete in Sub–Pakete dient nur der Übersichtlichkeit. In Bezug auf Importe ist sie ohne jede Bedeutung. Mit der Anweisung

```
import lineareAlgebra.*;
```

stehen *nur* Namen des Pakets `lineareAlgebra` zur Verfügung. Das gilt auch dann, wenn `lineareAlgebra` das Subpaket `lineareAlgebra.geometrie` hat. Sollen Namen aus diesem Paket verwendet werden, dann muss explizit importiert werden:

```
import lineareAlgebra.*;
import lineareAlgebra.geometrie.*;
```

Auch für die Sichtbarkeit spielen die Paket–Hierarchien keine Rolle. Mit der Paket–Sichtbarkeit `public` werden Namen für alle anderen Pakete zugänglich – wo auch immer das exportierende und das importierende Paket in der Hierarchie angesiedelt sind. Ohne `public` ist die Sichtbarkeit auf das definierende Paket beschränkt. Auch Sub–Pakete haben keinen Zugriff.

Klassenpfade

Die Paketstruktur eines Programms spiegelt sich in der Verzeichnisstruktur wieder. Bei einer Aufteilung in Subpakete wird diesen ein Unterverzeichnis zugeordnet. Quell– und Class-Dateien, die zu einem Paket gehören, liegen im Verzeichnis oder Unterverzeichnis das dem Paket zugeordnet ist. Das Verzeichnis hat den gleichen Namen wie das Paket. Die Dateien haben den gleichen Namen wie die Typen (Klassen oder Schnittstellen), die in ihnen definiert sind.

Diese Struktur ist sowohl beim Übersetzen, als auch beim Aktivieren von Java–Programm zu beachten. (Es sei denn das übernimmt die Entwicklungsumgebung für uns.) Nehmen wir an unser Quellprogramm besteht aus drei Klassendefinitionen die wir in Eclipse als Bestandteil eines Projektes mit dem Namen `Projekt2` angelegt haben. Die Quellen liegen dann in drei Dateien:

- Definition der Klasse `Vektor` im Paket `lineareAlgebra.geometrie` in:
  `/ein/pfad/.eclipse/Projekt2/lineareAlgebra/geometrie/Vektor.java`
- Definition der Klasse `Punkt` im Paket `lineareAlgebra.geometrie` in:
  `/ein/pfad/.eclipse/Projekt2/lineareAlgebra/geometrie/Punkt.java`
- Definition der Klasse `TestGeo` im Paket `lineareAlgebra` in:
  `/ein/pfad/.eclipse/Projekt2/lineareAlgebra/TestGeo.java`

Der Pfad zur Paketstruktur im Dateisystem wird *Klassenpfad* (engl. *classpath*) genannt. In unserem Beispiel ist das als `/ein/pfad/.eclipse/Projekt2`. Dieser Klassenpfad muss zunächst beim Übersetzen beachtet werden. Mit Kommandos wie (jeweils in einer Zeile, unter Windows die Querstriche in anderer Orientierung):

```
javac –classpath /ein/pfad/.eclipse/Projekt2
       /ein/pfad/.eclipse/Projekt2/lineareAlgebra/TestGeo.java
javac –classpath /ein/pfad/.eclipse/Projekt2
       /ein/pfad/.eclipse/Projekt2/lineareAlgebra/geometrie/Vektor.java
javac –classpath /ein/pfad/.eclipse/Projekt2
       /ein/pfad/.eclipse/Projekt2/lineareAlgebra/geometrie/Punkt.java
```

können die Quelldateien aus einem beliebigen Verzeichnis heraus übersetzt werden. Von der Wurzel des Paketbaums aus, vom Klassenpfad aus also, kann man sich die Angabe des Klassenpfades ersparen:

```
cd /ein/pfad/.eclipse/Projekt2
javac lineareAlgebra/TestGeo.java
javac lineareAlgebra/geometrie/Vektor.java
javac lineareAlgebra/geometrie/Punkt.java
```

Der Klassenpfad muss auch der virtuellen Maschine (dem Interpreterprogramm `java`) bekannt sein:

```
java –classpath /ein/pfad/.eclipse/Projekt2 lineareAlgebra.TestGeo
```

Dem Interpreter wird, im Gegensatz zum Compiler `javac`, kein Dateiname sondern ein voll qualifizierter Klassenname `lineareAlgebra.TestGeo` übergeben. Sind wir dort wohin der Klassenpfad führt, also in der Wurzel des Paketbaums, dann kann auch beim Interpretierer die Angabe des Klassenpfades entfallen:

```
cd /ein/pfad/.eclipse/Projekt2
java lineareAlgebra.TestGeo
```

Klassenpfade können auch als Umgebungsvariablen gesetzt werden. Dies wird jedoch nicht empfohlen.

Eclipse setzt den Klassenpfad im Allgemeinen selbständig und korrekt. In komplizierten Fällen kann es nötig sein die Einstellungen zu kontrollieren, zu modifizieren oder zu erweitern. Das ist über den `run..`–Dialog möglich.

2.3 Klassendefinitionen

2.3.1 Objekte in Programmen und in der Welt

Was ist die richtige Klassendefinition?

Die Mechanik der Klassendefinition zu verstehen ist eine Sache. Eine andere ist, die richtigen Klassen zu einer bestimmten Anwendung zu finden. Es gibt in der Regel nicht *die* richtige Klasse zur Darstellung eines Sachverhalts. Es gibt viele Möglichkeiten, von denen einige mehr, andere weniger und viele gar nicht geeignet sind.

Klassen modellieren Objekte. Objekte sind darum der Ausgangspunkt der Überlegungen. Eine Frage, die man sich beim Entwurf einer Klasse stellen kann, ist

"Was ist ein Objekt; aus was besteht es?".

Dieser Ansatz ist nicht falsch, aber die Frage nach den Bestandteilen eines Objekts ist nicht immer die wichtigste und fast nie die einzige, die zu stellen ist. Andere Fragen, mit denen man das Wesen einer Klasse erforschen kann und sollte, sind

"Was kann ein Objekt; welche Aufgaben übernimmt es?",

oder

"Welche Zustände hat ein Objekt; wie verhält es sich im Lauf der Zeit?"

Aus den Antworten auf solche Fragen kann man das Wesen der Objekte erkennen und dieses dann in einer Klassendefinition konzentrieren.

Objekte: die Gleichen und die Selben

Objekte leben innerhalb von Computerprogrammen. Sie sind dort unterscheidbare Individuen: Speicherbereiche, die durch ihre Adresse[14] eindeutig charakterisiert sind. Dabei wird auf einer technischen Basis zwischen "dem Gleichen" und dem "dem Selben" Objekt unterschieden. Zwei Objekte einer Klasse sind *gleich*, wenn ihre Instanzvariablen die gleichen Werte haben. Zwei Objekte sind *dieselben*, wenn sie durch eine einzige `new`–Anweisung erzeugt wurden, also tatsächlich eins sind. Zuweisung, Vergleich und Parameterübergabe operieren auf Referenzen. Beim Vergleich wird auf "dasselbe" getestet, nach einer Zuweisung beziehen sich zwei Variablen auf "denselben" und bei einer Parameterübergabe wandert "dasselbe" in die aufgerufene Methode.

Wir unterscheiden darum auf technischer Ebene:

- Gleichheit: Zwei Speicherbereiche können mit "gleichen" Werten belegt sein.
- Identität: Zwei Referenzen können sich auf "denselben" Speicherbereich beziehen.

Objekte und Werte technisch

Von Objekten abgegrenzt sind Werte. Sie haben keine eigene Individualität. Die 5 in der Variablen `a` ist nicht von der 5 in der Variablen *b* unterscheidbar. Beim Vergleich wird auf "der gleiche Wert" getestet, nach einer Zuweisung enthalten zwei Variablen "den gleichen" Wert und bei einer Parameterübergabe wandert "der Gleiche" Wert in die aufgerufene Methode.

Wir unterscheiden in Java Programmen – also auf einer rein technischen Ebene:

- Werte: Inhalte von Variablen. Ein Vergleich testet die Gleichheit; Zuweisung und Übergabe bewegen Kopien.
- Objekte: Speicherbereiche im Heap. Zugriff über Referenzen; ein Vergleich testet die Identität; Zuweisung und Übergabe bewegen Referenzen.

Individuen und Werte in der Welt

Es ist wichtig Javas Konzept von Werten, Objekten, Gleichheit und Identität zu verstehen. Man darf es aber nicht mit der Wirklichkeit verwechseln! Auch in der Realität gibt es Dinge, bei denen es Sinn macht, ihnen eine Individualität zuzusprechen und andere bei denen es keinen Sinn macht. Beispielsweise ist es unsinnig, zwei 5–en oder zwei Null–Vektoren zu unterscheiden. Zwei bissige Pinscher mit dem Namen "Fifi" können dagegen sehr wohl völlig unterschiedliche Individuen sein. Sie sind gleich

[14] die Referenz die auf sie verweist

aber nicht dieselben. Umgekehrt ist es möglich, dass derselbe nicht immer der gleiche bleibt. Einer der beiden Fifis besucht die Hundeschule und ist danach nicht mehr bissig.

Wir unterscheiden darum auch in der Realität die beiden Grundkonzepte von Dingen:

- Werte: Ewige Dinge die außerhalb von Raum und Zeit existieren. Sie ändern sich nicht, sie haben keine eigene Individualität.
- Individuen: Eine Einheit, die in Raum und Zeit existiert, sich dabei verändern kann aber trotzdem ihre Individualität behält.

Werte sind konzeptionell einfacher als Individuen. Eine Fünf bleibt für immer eine Fünf. In welchem Zustand Fifi ist, ist dagegen schwierig zu klären. Beißt er oder beißt er nicht? Das Konzept der Individuen (Objekte) scheint natürlicher zu sein als das abstraktere Konzept eines Wertes. (Wer hat schon mal gesehen wie eine Fünf um die Ecke kommt?)

Wir lassen uns auf keine Diskussion darüber ein, welches das bessere, einfachere, natürlichere, einzige Konzept ist. Werte sind ein elementares und einfaches Konzept.[15] Individuen (Objekte in der Realität) sind ein anderes elementares Konzept. Was davon angeboren und was angelernt ist, mögen Psychologen klären. Die Softwaretechnik interessiert diese Klärung nicht.

Technische Objekte modellieren reale Werte oder Individuen

Objekte im Programm sollen Dinge der Welt modellieren. Selbstverständlich sollte man sich bemühen, das Modell so nahe wie möglich oder nötig an sein Vorbild anzugleichen. Das Modell, das Objekt im Programm, und die Realität sind und bleiben aber zwei verschiedene Dinge. Das wichtigere ist immer die Realität.

Am Anfang einer Klassendefinition steht darum die Frage nach dem Grundcharakter der Dinge, die im Programm modelliert werden sollen. Sind es Werte oder sind es Individuen? Entsprechend dem Vorbild der Wirklichkeit definieren wir dann eine *wert–* oder eine *zustands–orientierte* Klasse.

Wertorientierte Klassen modellieren (unveränderliche) **Werte** in der Realität. **Zustandsorientierte Klassen** modellieren (meist wechselhafte) **Individuen** in der Realität.

Die “Realität” ist dabei das Anwendungsgebiet der Software.

In der englischsprachigen Literatur wird eine Instanz einer wertorientierten Klasse oft *Value Object* genannt.

> *Value Object: A small simple object, like money or a date range, whose equality isn't based on identity.* [16]

Werte und Individuen werden dabei auf Objekte im Programm abgebildet. Dabei mag es eventuell irritieren, dass reale Werte nicht unbedingt durch Programm–Werte sondern durch Programm–Objekte repräsentiert werden. Unterscheidet man aber konsequent Modell und Realität, dann ist das nicht problematisch.

Übereifrige Evangelisten der Objektorientierung wollen uns gelegentlich einreden, dass der Unterschied zwischen Modell und Realität möglichst verwischt und vergessen werden soll. Das ist ein Holzweg auf dem man entweder zu der Überzeugung kommt, alles Reale sei auf Programmkonstrukte von Java abbildbar (ein falsches Verständnis von Java) oder, noch schlimmer, die Realität sei nach dem Vorbild von Java geschaffen (ein jämmerliches Verständnis der Realität).

2.3.2 Wertorientierte Klassen

Vektoren sind Werte

Im Mathematikunterricht lernen wir, was Vektoren sind:

> *Ein Vektor ist eine gerichtete, orientierte Strecke.*

Nach dieser Definition hat ein Vektor eine Richtung, eine Orientierung und eine Länge. Diese Charakteristika eines Vektors lassen sich auf unterschiedliche Art darstellen. Üblich ist die Koordinatendarstellung.

In etlichen tausend Jahren Mathematik hat es sich als sehr erfolgreich erwiesen, mit Werten zu arbeiten. Das ist für Mathematiker so selbstverständlich, dass es erst gar nicht thematisiert wird. Für Nichtmathematiker ist es allerdings nicht ganz so selbstverständlich. Es bedarf darum einer gewissen Diskussion, um zu klären, dass die Vektoren $\vec{x}$ und $\vec{y}$ mit den Koordinaten

[15] Das gilt zumindest für Mitglieder entwickelter Zivilisationen, die viele Jahre Rechen– und Mathematikunterricht hinter sich haben.
[16] Martin Fowler in *Patterns of Enterprise Application Architecture*, Addison-Wesley, p. 486

$$\vec{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \vec{y} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

völlig gleich sind, obwohl man sie an verschiedenen Stellen in der Ebene graphisch darstellen kann. (Siehe 2.6)

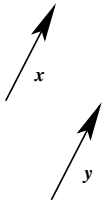


Abbildung 2.6: Vektoren sind Werte: x und y sind gleich

Die graphische Darstellung ist etwas verwirrend, da Vektoren als Pfeile an einer bestimmten Stelle in der Ebene oder im Raum dargestellt werden. Solche Pfeile haben eine Identität: Zwei von ihnen, obwohl gleich (mit gleicher Länge, Orientierung und Richtung) können sich an unterschiedlichen Stellen befinden. Vielleicht kann man sie verschieben und übereinander legen. Der eine oder andere ist vielleicht aus elastischem Material und kann in die Länge gezogen werden. Vektoren scheinen also Individuen mit jeweils eigener Identität zu sein, bei denen es sehr wohl Sinn macht zwischen Gleichheit und Identität zu unterscheiden und denen man sogar einen veränderlichen Zustand (Ort, Länge) zubilligen kann.

Tatsächlich ist aber im mathematischen Verständnis[17] ein Vektor genau das *nicht*. Ein Vektor ist vielmehr so etwas wie die Menge aller "Pfeile" mit einer bestimmten Richtung, Orientierung und Länge. Ein Vektor kann darum keinen Ort im Raum haben. Er ist nicht verschiebbar, nicht veränderbar und hat keine Individualität. Kurz: ein "wirklicher Vektor" ist kein Objekt sondern ein Wert. Wir modellieren ihn darum im Programm durch eine wertorientierte Klasse.

Eine Klasse Vektor

Software–Entwicklung ist kein Teilgebiet der Philosophie oder der Theologie. Es ist eine Ingenieurskunst. Es geht also nicht darum zu klären, welches das wahre Wesen eines Vektors ist. Es geht darum etwas Nützliches zu erzeugen das den Bedürfnissen und Erwartungen seiner Benutzer am Besten entspricht. Nehmen wir also an, unser Ziel sei die Entwicklung von Software im Bereich der linearen Algebra. Das ist eine mathematische Disziplin und unsere Klasse muss darum genau das mathematische Verständnis eines Vektors modellieren.

```
public final class Vektor {
  private final double x; // Vektoren sind unveraenderlich und
  private final double y; // jeder Vektor hat die Komponenten
  private final double z; // x, y und z

  // eine Klassenvariable
  public static final Vektor nullVektor = new Vektor(0.0,0.0,0.0);

  // Konstruktoren
  public Vektor() { x=1; y=1; z=1; }
  public Vektor(final double i, final double j, final double k) {
    x=i; y = j; z =k;
  }

  // eine Methode (ohne Zustandsaenderung)
  public double laenge () {
    return Math.sqrt(x*x+y*y+z*z);
  }

  //mathematische (wert-orientierte) Sicht der Operationen
  public static Vektor add(final Vektor x, final Vektor y) {
     return new Vektor(x.x + y.x, x.y + y.y, x.z + y.z);
  }
  public static Vektor sub(final Vektor x, final Vektor y) {
```

[17] das nicht nur tausende von Jahren alt ist, sondern sich auch als extrem erfolgreich und nützlich erwiesen hat

```
      return new Vektor(x.x - y.x, x.y - y.y, x.z - y.z);
   }
   public static Vektor mult(final int x, final Vektor v) { //mult ist ueberladen
      return new Vektor(v.x * x, v.y * x, v.z * x);
   }
   public static Vektor mult(final Vektor a, final Vektor b) {
      return new Vektor(a.y*b.z - a.z*b.y,
                  -(a.x*b.z - a.z*b.x),
                  a.x*b.y - a.y*b.x);
   }
   public String toString(){
      return "("+x+","+y+","+z+")";
   }
}
```

Die Klasse bildet die mathematische–, die Wert–Sicht, der Vektoren nach. Die Komponenten des Vektors sind darum unveränderlich. Operationen, die zu neuen Vektoren führen, werden als statische Methoden modelliert. Wenn es, wie mit `laenge` eine Methode gibt, dann modifiziert sie ihren Vektor nicht. Die Methode `toString` benutzen wir, wenn ein Vektor ausgegeben werden soll.

Ortsvektoren, Punkte und ihr Modell

In einem Koordinatensystem können *Punkte* identifiziert werden. Jeder Punkt ist durch seinen Ortsvektor eindeutig bestimmt. Der *Ortsvektor* führt vom Ursprung des Koordinatensystems zum Punkt. Sind Punkte ebenfalls wertartig? – In der Mathematik: Ja. Punkte, wie sie die Geometer sehen, sind ebenfalls ewige unveränderliche Werte: Man kann Punkte auf Punkte abbilden, aber man kann sie nicht bewegen. Zwei Punkte “an der gleichen Stelle” sind nicht unterscheidbar, und so weiter. Das führt uns zu einer Klassendefinition wie:

```
public final class Punkt {
 private final Vektor ortsVektor;       // Ortsvektor des Punktes

 public static final Punkt nullPunkt = new Punkt(); // eine Klassenvariable

 // Konstruktoren
 public Punkt(final Vektor ort){
   ortsVektor = ort;
 }
 public Punkt(final double x, final double y, final double z){
   ortsVektor = new Vektor(x, y, z);
 }
 public Punkt(){                        // Standardkonstruktor
   this(0.0, 0.0, 0.0);
 }
 public String toString() {
   return "P"+ortsVektor.toString();
 }
}
```

Punkte sind hier durch ihren Ortsvektor charakterisiert. Sie sind für immer und ewig unveränderlich und können aus ihrem Ortsvektor oder ihren drei Koordinaten erzeugt werden. Der Standardkonstruktor erzeugt den Nullpunkt.

Punkte und Vektoren in einem Paket organisieren

Punkte und Vektoren sind eng verwandt. Es liegt darum schon aus rein organisatorischen Gründen nahe sie in einem Paket `geometrie` zusammenzufassen:

```
package geometrie;                  package geometrie;
public final class Vektor {...}     public final class Punkt {...}
```

Pakete haben aber nicht nur den Zweck einer Klammer um zusammengehörige Klassendefinitionen. Sie sind auch dazu da, ihren Klassen eine gemeinsame Privatsphäre zu geben. Die Klassen eines Pakets sind “befreundet”: Nach dem Motto

> Privates geht nur mich was an, öffentliches darf jeder sehen, alles andere ist nur für meine Freunde aus dem gleichen Paket.

können sie sich mit der paketlokalen Sichtbarkeit gegenseitig Dinge zur Benutzung freigeben, die der Außenwelt verschlossen bleiben.

Die gegenseitige Offenheit können wir beispielsweise für eine Methode nutzen, die einen Vektor als Weg von einem Punkt zu einem anderen konstruiert:

```
package geometrie;
public final class Vektor {
  private final double x;
  private final double y;
  private final double z;
  ....
  // Vektor aus zwei Punkten:
  public Vektor(final Punkt p,
                final Punkt q) {
     this(
       sub(q.ortsVektor,
          p.ortsVektor) );
  }
  ....
}
```

```
package geometrie;
public final class Punkt {
  // NICHT PRIVATE:
  final Vektor ortsVektor;
  ....
}
```

Die jetzt paketlokale Sichtbarkeit von `ortsVektor` in `Punkt` macht es der befreundeten Klasse Vektor möglich, ohne Umstände darauf zuzugreifen, hier in einem Konstruktor, der einen Vektor aus der Differenz der Ortsvektoren von zwei Punkten erzeugt. Der Ortsvektor eines Punktes ist aber immer noch vor dem Zugriff fremder Klassen geschützt. Wir könnten die Darstellung des Punktes also immer noch ändern, ohne dass dies Auswirkungen auf "fremde" Klassen hätte.

Ein Paket ist also nicht einfach ein Sack voll Klassen. Es ist auch ein wichtiges Mittel zur Gestaltung ihrer Schnittstellen und damit ihrer Kooperations-Struktur.[18]

Diagramme

Softwarekomponenten werden wegen der besseren Übersichtlichkeit oft grafisch dargestellt. Nach vielen Jahren des völligen Chaos' durch konkurrierende Darstellungskonventionen hat sich jetzt UML (*Unified Modeling Language*) allgemein durchgesetzt. UML ist in erster Linie eine Notation für Spezifikationen, also für Code, den man noch schreiben will. Es kann und wird aber auch zur Dokumentation von existierendem Code genutzt.

Unsere beiden Klassen `Vektor` und `Punkt` als Klassendiagramme sind in Abbildung 2.7 dargestellt.

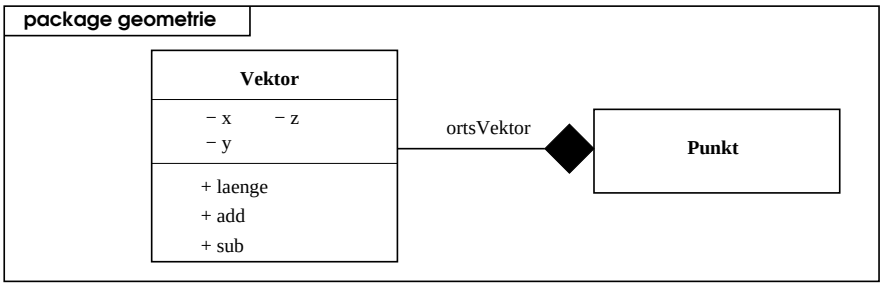


Abbildung 2.7: Klassendiagramm für Punkt und Vektor

Klassen werden durch *Klassendiagramme* in Form eines Rechtecks dargestellt. Oben enthalten sie den Klassennamen, es folgen die Datenkomponenten und dann die Methoden. Öffentliche Komponenten werden mit einem + gekennzeichnet, private mit einem −. Alles mit einem + gehört also zur Schnittstelle. So wie in diesem Beispiel sind sehr oft alle Datenkomponenten privat.

Beziehungen zwischen Klassen werden durch Verbindungen der Diagramme dargestellt. In unserem Fall haben wir eine "Teil–Ganzes"–Beziehung. Ein Punkt hat mit seinem Ortsvektor einen Vektor als Komponente.

Die Kennzeichnungen + und − können auch weggelassen werden. Beispielsweise dann, wenn im Laufe eines Entwurfsprozesses noch nicht bekannt ist, was sichtbar sein soll und was nicht, oder wenn die Sichtbarkeit (noch) nicht wichtig ist. Generell gilt,

[18] Die Paket–*Hierarchie* spielt dabei allerdings *keine* Rolle. Eine Definition bleibt in einem Paket oder sie wird exportiert. Es ist nicht möglich die Sichtbarkeit auf eine bestimmte Stufe der Pakethierarchie zu beschränken. Insgesamt gibt es keine besonderen Regeln für die Sichtbarkeit von einem Paket auf Definitionen in einem übergeordneten oder untergeordneten anderen Paket.

dass Quellcode und Diagramme nicht eins zu eins aufeinander abbildbar sind. Die Diagramme stellen eine abstraktere Sicht der Dinge dar. Das bedeutet, dass unwichtige oder unbekannte Informationen in ihnen fehlen können.

Die abstraktere Sicht bedeutet aber auch, dass Dinge, die im Quellcode ähnlich aussehen, im Diagramm deutlich unterschieden werden können. So haben in unserem Quellcode die Vektoren eine `x`–, `y`– und `z`–Komponente. Die Punkte haben eine Komponente `ortsVektor`. Wir sehen das als gleichartige Implementierung von zwei unterschiedlichen Konzepten: Vektoren haben *Attribute* (Eigenschaften) mit den Namen `x`, `y` und `z`. Punkte dagegen *bestehen* aus einem Vektor. Der Ortsvektor ist kein Attribut des Punktes, sondern eine Komponente. Im Code sind die Attribute `x`, `y` und `z` und die Komponente `ortsVektor` jeweils als Instanzvariablen implementiert. Im Diagramm sind sie deutlich unterschieden.

Selbstverständlich kann man beliebig lange diskutieren, was ein Attribut und was eine Komponente ist. Typischerweise sind Werte die Implementierung von Attributen, Referenzen dagegen sind die Implementierung von Komponenten–Beziehungen.

Gerade: eine weitere wertorientierte Klasse

Eine Gerade kann in Punktrichtungsform mit Hilfe von zwei Vektoren dargestellt werden:

$\vec{x} = \vec{p} + \lambda\vec{a}$

Dabei ist $\vec{p}$ der Ortsvektor eines Punktes, durch den die Gerade geht und $\vec{a}$ ist der Richtungsvektor. Geraden sind im mathematischen Verständnis Werte: ewige unveränderliche Einheiten. Wir modellieren sie durch eine wertorientierte Klasse `Gerade`:

```
package geometrie;

public final class Gerade {
  private final Vektor p;        // Ortsvektor
  private final Vektor a;        // Richtungsvektor

  public Gerade(){               // Standardkonstruktor
    p = Punkt.nullPunkt.ortsVektor;
    a = Vektor.nullVektor;
  }

  public Gerade(final Punkt p, final Vektor richtung){// Vektor aus Punkt und Richtung
    this.p = p.ortsVektor;
    this.a = richtung;
  }
  ....
}
```

Zur Illustration wollen wir unsere wertorientierten Klassen noch mit ein wenig Funktionalität ausstatten: Geraden sollen auf Parallelität geprüft werden. Die entsprechende Mathematik ist recht einfach: Zwei Geraden

$\vec{p}_1 + \lambda\vec{a}_1$ und $\vec{p}_2 + \lambda\vec{a}_2$

sind parallel, wenn sie die gleiche Richtung haben. Wenn also die Richtungsvektoren $\vec{a}_2$ und $\vec{a}_1$ kollinear sind. Die Vektoren $\vec{a}_2$ und $\vec{a}_1$ sind kollinear wenn es ein s gibt mit:

$\vec{a}_2 = s * \vec{a}_1$

Dies wiederum ist der Fall, wenn das Gleichungssystem

$$\begin{aligned} s * a_1.x &= a_2.x \\ s * a_1.y &= a_2.y \\ s * a_1.z &= a_2.z \end{aligned}$$

eine Lösung hat, d.h. die Determinante $\left|\ \vec{a}_1\ \vec{a}_2\ \right|$ nicht den Wert 0 hat.

Verantwortlichkeit als weiteres Entwurfskriterium

Bevor wir uns in mathematischen Details verlieren muss geklärt werden, welche Klasse für welche Funktionalität die *Verantwortung* übernehmen kann und muss. Die Verantwortlichkeit ist ein wichtiges Kriterium beim Entwurf von Klassendefinitionen. Wir gehen dabei von Objekten aus und ergänzen unsere Frage nach den Komponenten und die nach der Verantwortlichkeit.

Entwurfskriterien beim Entwurf von Klassen:

- *Welche Klassen gibt es:* Welche Dinge kommen im Problembereich vor, wie kann man sie zu Klassen gruppieren?
- *Was ist ihr jeweiliger Grundcharakter:* Welchen Charakter haben sie im Problembereich: sind es Werte oder Individuen?
- *Was sind ihre Attribute/Komponenten:* Was wissen / woraus bestehen diese Dinge?
- *Welche Verantwortung übernehmen sie:* Was *können* sie, für welche Funktionalität übernehmen sie die Verantwortung?

Die Antwort auf die Frage nach den Dingen/Objekten ist bei unserem kleinen Beispiel sehr einfach: Wir haben es mit Punkten, Vektoren und Geraden zu tun. Alle haben einen wertorientierten Charakter und ihre Attribute/Komponenten konnten direkt aus der mathematischen Definition übernommen werden.

Die Verantwortlichkeit ist klar: Vektoren haben zu prüfen, ob sie kollinear sind und Geraden müssen entscheiden können, ob sie parallel sind. Wir erweitern die Klassendefinitionen entsprechend:

```
package geometrie
public final class Gerade {
  ...
  public static boolean parallel(final Gerade g1, final Gerade g2) {
    return Vektor.kollinear( g1.a, g2.a );
  }
  ...
}
```

und

```
package geometrie
public final class Vektor {
  ...
  public static boolean kollinear(final Vektor a, final Vektor b) {
    return (a.y * (a.x / b.x) == b.y)
         && (a.z * (a.x / b.x) == b.z);
  }
  ...
}
```

Abstrakte Datentypen

Datentypen wie `int, char, double`, etc., sind in der Sprache Java vordefiniert. Man bezeichnet sie als *konkrete* Datentypen. Man kann Variablen von diesem Typ definieren, ihnen entsprechende Werte zuweisen und mit speziellen Operatoren verknüpfen. Vektoren, Punkte und Geraden kann man als Klassen definieren und dann so ähnlich benutzen wie die in der Sprache fest vorgegebenen konkreten Datentypen. Sie sind dadurch aber nicht in der gleichen Art wie diese in die Sprache integriert. Sie sind in gewisser Weise nur Fiktionen richtiger Datentypen und werden darum oft *abstrakte Datentypen* genannt.[19]

Abstrakte Datentypen existieren also im Gegensatz zu konkreten Datentypen nicht wirklich. Eine Variable vom Typ `Vektor` enthält ja keinen Vektor sondern eine Referenz auf ein Exemplar der Klasse `Vektor`. Sie sind somit eine Fiktion des Programms bzw. seiner Autoren. Immerhin eine recht realitätsnahe Fiktion. Mit den richtigen Klassendefinionen kann man (fast) so tun, als gäbe es Vektoren, Punkte und Geraden, oder was immer man sonst so braucht.

Die Modellierung von "realen" Werten (Vektoren) durch Konstrukte des Programms nennt man *Datenabstraktion*. Die Datenabstraktion ist ein wichtiger Schritt bei der Entwicklung von Software. Es hat sich schon früh herausgestellt, dass es vorteilhaft ist, die Modellierung "realer" Dinge, die Datenabstraktion also, möglichst klar darzulegen und an einer Stelle im Quelltext zu konzentrieren. Dazu wurden verschiedene sprachtechnische Hilfsmittel vorgeschlagen. Auch Java (und andere moderne Sprachen) wurde unter anderem mit dem Ziel definiert, die Arbeit mit Datenabstraktionen zu erleichtern. Das Klassenkonzept ist heute der übliche Mechanismus zur Realisierung von Datenabstraktionen. Es unterstützt die Definition abstrakter Datentypen gut: Man kann eine Klasse `Vektor` mit allen benötigten Methoden definieren und hat dann den Datentyp "Vektor".

[19] Natürlich sind die konkreten Datentypen ebensolche Fiktionen wie die abstrakten. In beiden Fällen handelt es sich real um Bitmuster, die in einer bestimmten Weise interpretiert werden. Der Unterschied liegt darin, dass es sich bei den konkreten Datentypen um Fiktionen des Sprachentwurfs handelt, die mit Hilfe des Compilers realisiert werden. Die abstrakten Datentypen dagegen sind nur Fiktionen des Programmierers, die er mit eigenem Code realisieren muss.

2.3.3 Zustandsorientierte Klassen

Das Gleiche ist nicht das Gleiche wie das Selbe

Zustandsorientierte Klassen modellieren Dinge, die sich in Raum und Zeit verändern können und dabei ihre Individualität nicht verlieren. Wenn ich zu fünf eins addiere dann wird die Fünf nicht zu einer Sechs verändert. Eine so kaputte Fünf wäre bei weiteren Rechenoperationen fatal. Wenn der Autor dieser Zeilen aber zum Mittagessen zwei Pizzen verspeist, dann ist er danach noch derselbe, nur anders.

Ich bin also im Gegensatz zu einer *Fünf* keine Instanz einer wert– sondern einer zustandsorientierten Klasse. Die meisten und die interessanteren Klassen sind zustandsorientiert. Die zustandsorientierte Sicht wird oft für "natürlicher" gehalten als die wertorientierte, ob das so ist, mögen Philosophen und Psychologen entscheiden. Sie ist aber in jedem Fall die kompliziertere und vielleicht haben Mathematiker auch ihre Gründe dafür, alles wertorientiert zu modellieren. Wie dem auch sei – wir sind ja Informatiker und keine Mathematiker.

Punkte zustandsorientiert

Nicht alle sind Mathematiker und Punkte muss man darum nicht unbedingt als mathematische Objekte ansehen. Für viele Anwendungen ist es wesentlich sinnvoller, ihnen eine eigene Individualität und einen veränderbaren Zustand zuzubilligen. Eine Diskussion über das "wahre Wesen" der Punkte ist sinnlos. Es kommt immer auf den Kontext, die Verwendung an: "Wer benutzt Punkte zu welchem Zweck?".

In einer mathematisch orientierten Anwendung wie oben angedeutet sind Punkte als Werte anzusehen. In einer graphischen Anwendung, in der Punkte in einem Fenster umherwandern und dabei noch verschiedene Farben annehmen können, wäre diese Sicht völlig unangebracht. Dort sind Punkte zustandsorientiert:

```
// Punkte Zustandsorientiert

public final class Punkt {
  private double x; // aktuelle Position
  private double y; // der Zustand eines Punktes

  // Konstruktoren
  public Punkt() {
    this(0.0, 0.0);
  }
  public Punkt(final double x, final double y){
    this.x = x; this.y=y;
  }

  // Methode, nicht statisch
  // veraendert den Zustand
  public void bewege(final double dx, final double dy) { // Punkt verschieben
    x = x+dx; y = y+dy;
  }
}
```

Diese Punkte kann man bewegen. Danach sind sie anders aber immer noch dieselben. Die Koordinaten sind nicht konstant. Ihr aktueller Wert bildet den *Zustand* eines Objekts. Die Methoden sind typischerweise nicht statisch und verändern den Zustand ihres Objekts.

Zustand und abstrakter Zustand

Eine bekannte und wichtige Art zustandsorientierter Dinge sind Konten, die als Exemplare einer Klasse `Konto` modelliert werden können:

```
public final class Konto {
  private int betrag; // Zustand: aktueller Betrag in Cent

  public Konto() { betrag = 0; }

  public void einzahlen(final int betrag) {
    this.betrag = this.betrag+betrag;
  }
```

```
  public void abheben(final int betrag) {
    this.betrag = this.betrag-betrag;
  }
}
```

Das Konzept des Zustands eines Kontos ist ebenfalls den meisten vertraut. Er ist durch den aktuellen Betrag auf dem Konto gegeben.

Zustände werden oft auch etwas abstrakter charakterisiert. So sagt man, das Konto ist “überzogen”, “leer” oder ”im Positiven” wenn der Betrag kleiner, gleich oder größer Null ist. Die unendliche Zahl der möglichen Zustände wird dadurch auf drei Klassen zusammengefasst. Es ist üblich, eine große oder unendliche Zahl von Zuständen in dieser Art zu *abstrakten Zuständen* zusammenzufassen, die man dann aber auch meist einfach als “Zustand” bezeichnet.

Zustandsdiagramme

Die Zustände und die möglichen Übergänge von einem zum anderen sind eine wesentliche Information über die Objekte einer zustandsorientierten Klasse. Ein einfacher Schalter kann den Zustand `an` bzw. `aus` annehmen. In einem Zustandsdiagramm (siehe 2.8) wird das durch zwei Knoten, die Zustände, mit den Übergängen als gerichtete Kanten dargestellt. Die Kanten werden mit den Aktionen beschriftet, die zu dem Zustandsübergang führen.

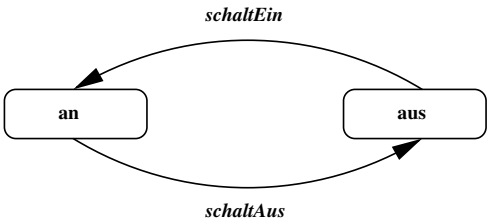


Abbildung 2.8: Zustandsdiagramm eines einfachen Schalters

Der Stapel: die Mutter aller zustandsorientierten Klassen

Das was in der Biologie die Fruchtfliege Drosophila ist, das ist in der Softwaretechnik der Stapel (engl. *Stack*). An ihm wurden alle modernen Ideen zur Modularisierung, dem Geheimnisprinzip und zur Objektorientierung ausprobiert und erläutert. Stapel sind einfach aber gleichzeitig interessant genug, um die Ideen und Probleme illustrieren zu können.

Das natürliche Verständnis eines Stapels ist zustandsorientiert. Ein Stapel hat eine Identität, die er bei allen Veränderungen im Laufe seines Lebens beibehält. Zwei Stapel mögen gleich sein, sie sind aber nie derselbe, auch wenn auf beiden das Gleiche aufgestapelt ist. Ein Stapel hat einen Zustand: das was gerade auf ihm gestapelt ist. Sein Zustand kann verändert werden, indem man etwas auf ihm ablegt oder wegnimmt.

Ein Stapel kann durch ein Feld und eine Zählvariable repräsentiert werden. Das Feld enthält die Stapelelemente und die Zählvariable gibt an welche Positionen belegt sind:

```
public final class Stack {
  private int count;      // aktuelle Zahl der Elemente - 1
                          // Index des obersten Elements
  private int[] a = new int[10]; // Ablageplatz fuer die Elemente
  // a[0]...a[count] sind die gestapelten Elemente
  // a[count] ist das oberste, a[0] das unterste
  // INV: -1 <= count <= 9

  public Stack () { // Konstruktor, erzeugt den leeren Stapel
    count = -1;
  }

  //Leseoperation, keine Zustandsaenderung
  public int top() { // liefert das oberste Element
                     // Vorbedingung: Stapel nicht leer
    if (count<=9 && count>=0) {
       return a[count];
```

```
      } else
        return -999;
  }

  // Schreiboperationen, zustandsaendernd, ohne Ergebnis
  // veraendert Stapel-Zustand: legt Element ab
  public void push(final int x){
                    // Vorbedingung: Stapel nicht voll
    if (count<9) {
       count++;
       a[count] = x;
     }
  }
  public void pop(){  // veraendert Stapel-Zustand: entfernt oberstes Element
                      // Vorbedingung: Stapel nicht leer
    if (count>=0) {
       count--;
    }
  }
}
```

Ein Stapel–Objekt

- hat einen Zustand, der sich aus den in ihm abgelegten Elementen ergibt. Konkret ergibt sich der Zustand aus den Werten der Variablen `count` und `a`.
- Mit lesenden Zugriffsoperationen kann man etwas über den Zustand erfahren. Lesende Operationen sind nicht–statische Methoden mit einem Ergebnis. Sie verändern den Zustand nicht.
- Mit schreibenden Zugriffsoperationen kann der Zustand verändert werden. Es sind nicht–statische Methoden ohne Ergebnis die die Zustandsvariablen verändern.

Eine zustandsorientierte Klasse wird in der Regel dazu verwendet, eine begrenzte Zahl von Objekten zu erzeugen, die dann wechselnde Zustände durchlaufen.

Zuweisung und Übergabe bei zustandsorientierten Klassen

In Java werden alle Objekte per Referenz zugewiesen und übergeben. Das passt in der Regel gut zum Charakter dieser Klassen. Bei der Zuweisung und Übergabe wird keine Kopie übergeben, sondern das “Ding selbst”. Die Identität bleibt damit erhalten. Bei einer Zuweisung wird kein Wert transferiert, sondern der Bezug auf ein Objekt. Alle Aktionen, die dasselbe Objekt über den einen Namen ansprechen, verändern dann das, was über den anderen Namen ansprechbar ist – es sind ja dieselben:

```
Stack a = new Stack();
Stack b = a;      // a und b beziehen sich auf den gleichen Stack
b.push(5);        // b (das auf das sich b bezieht) wird mitveraendert
```

2.3.4 Ausnahmen und illegale Zustände

Abstrakter und konkreter Zustand beim Stapel, Datentypinvariante

Ein Stapel hat ein “oben” und ein “unten” und kann unterschiedlich lang sein. Wenn wir ihn, wie oben, durch ein Feld fester Länge darstellen, brauchen wir eine zusätzliche Information darüber, wie lang er ist. Außerdem wird eine Verabredung darüber gebraucht, wo oben und wo unten ist.

Ein senkrecht stehender Stapel unterschiedlicher Länge ist etwas, das nur in unserer Vorstellung[20] existiert. Es ist abstrakt. Konkret sind die Inhalte der Variablen `a` und `count`: die konkreten Zustände des Stapels. Jeder abstrakte Zustand sollte eine konkrete Repräsentation haben und und jeder *legale* konkrete Zustand sollte einen abstrakten Zustand repräsentieren.

Beispielsweise wird der Stapel

1
5
2

[20] und in der Welt der physischen Dinge (aber wie real ist diese Welt für Informatiker?)

sowohl durch

```
count = 2
a = {2, 5, 1, 6, 3, 8, 5, 3, 3, -9999 }
```

als auch durch

```
count = 2
a = {2, 5, 1, 0, -67, 0, 0, 0, -773, 0 }
```

repräsentiert. Man kann sagen, dass, bei Vernachlässigung (Wegabstraktion) irrelevanter Details, beide konkreten Zustände den gleichen abstrakten Zustand darstellen.

```
count = 2                                          Abstraktion    | 1 |
a = {2, 5, 1, 0, -67, 0, 0, 0, -773, 0 }           ------->       | 5 |
                                                                  | 2 |
```

```
count = 2                                          Abstraktion    | 1 |
a = {2, 5, 1, 0, -67, 0, 0, 0, -773, 0 }           ------->       | 5 |
                                                                  | 2 |
```

Der konkrete Zustand

```
count = -9993481
a = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

ist nicht legal. Die Zählvariable ist außerhalb ihres gültigen Bereichs. Es ist von großer Wichtigkeit, sich über die abstrakten und konkreten Zustände und deren Beziehungen klar zu sein. Oft sind nur bestimmte Belegungen der Objektvariablen legal. Nur wenn `count` sich im Bereich -1 (leerer Stapel) bis 9 (Stapel voll) bewegt, ist der Stapel in einem definierten legalen Zustand. Diese Forderung muss immer, vor und nach jeder Operation, gültig sein. Man nennt sie darum oft in Anlehnung an die Schleifeninvariante, eine *Datentypinvariante*.

Illegale Operationen

Nicht immer ist jede Operation in jedem Zustand erlaubt. Auch dann nicht, wenn der Zustand legal ist. Ist der Schalter aus, dann kann er nicht mehr ausgeschaltet werden. Ist der Stapel voll, dann kann nichts mehr abgelegt werden. In unserem Stapel oben haben wir das Problem der Operationen in nicht passenden Zuständen auf zwei Arten gelöst, die beide nicht zufrieden stellen können:

- `top` liefert einen "Fehlerwert" (-999) wenn es eigentlich nicht ausgeführt werden kann, weil der Stapel leer ist.
- `push` und `pop` ignorieren die Aufforderung etwas zu tun völlig, wenn der Stapel nicht in einem passenden Zustand ist: `push` tut nichts, wenn der Stapel voll ist, `pop` tut nichts wenn er leer ist.

Die Lösung mit dem Fehlerwert ist dann zu akzeptieren, wenn es wirklich ein Fehlerwert ist. Da jede int–Zahl im Stapel abgelegt werden kann, kann keine int–Zahl, und sei sie noch so seltsam, als Fehlerwert dienen. Ein Fehlerwert muss immer klar außerhalb des Bereichs der möglichen Werte liegen.

Die Entscheidung eine illegale Operation zu ignorieren, oder in irgendeiner anderen unkonventionellen Art zu behandeln, ist dann OK, wenn der 'Kunde" die *Vorbedingung* jeder Operation kennt, also darüber informiert ist, dass die Prüfung in seiner Verantwortung liegt und auch die Chance hat, eine solche Prüfung vorzunehmen.

In unserem Fall des Stapels ist das nicht gegeben. Für den Kunden gibt es keine Möglichkeit zu testen, ob der Stapel voll oder leer ist. Er kann die Verantwortung für eine korrekte Bedienung darum nicht übernehmen.

Manche Operationen können in manchen Zuständen nicht ausgeführt werden. In dem Fall muss exakt klar sein wer – Kunde oder Objekt – dafür verantwortlich ist, dass eine solche Situation aufgetreten ist und wer – Kunde oder Objekt – darauf in angemessener Weise zu reagieren hat.

Ausnahmen werfen

Das Objekt, das eine Operation ausführen soll, erkennt am leichtesten, wenn und dass sie nicht ausgeführt werden kann. Der Kunde (Programmcode, der das Objekt verwendet) dagegen ist derjenige der am besten weiß, was alternativ zu tun ist. Das Objekt muss ihn nur unmissverständlich auf die besondere Situation hinweisen. Weiß der Kunde selbst nicht wie es weitergehen soll, ist auch er in einem Zustand, in dem die normale Arbeit nicht fortgesetzt werden kann. Er muss dann seinen eigenen Auftraggeber darüber informieren, dass er mit seiner Weisheit am Ende ist und es ist an diesem zu reagieren.

Dieses Muster der Problembehandlung wird in modernen Programmiersprachen durch das Konzept der Ausnahmen (engl. *Exception*) unterstützt. Der Auftragnehmer stellt fest, dass er einen Auftrag nicht ausführen kann und *wirft* (engl. *throws*) eine Ausnahme.

Stellt beispielsweise die `top`–Operation des Stapels fest, dass es keinen Wert gibt, den sie liefern kann, dann wirft sie mit der `throw`–Anweisung eine Ausnahme:

```
final class Stack {
  ...
  public int top() {// liefert das oberste Element
    if (count <= 9 && count >= 0) {
      return a[count];
    } else
      throw new RuntimeException("Stapel leer oder illegaler Index count");
  }
  ...
}
```

Mit der Anweisung

`throw new` *ExceptionTyp("optionale Fehlermeldung")*

wird ein Ausnahme–Objekt erzeugt und die aktuell laufende Ausführung abgebrochen.

Zwei Arten von Ausnahmen

Eine Programmkomponente kann auf zwei Arten in eine Ausnahmesituation kommen. Entweder wird von ihr eine Aktion erwartet, die sie, zwar prinzipell, aber nicht im aktuellen Zustand ausführen kann, oder sie ist in einem Zustand, in dem gar nichts mehr geht. Die `top`–Operation kann beispielsweise nicht ausgeführt werden, wenn der Stapel leer ist. Man erkennt das daran, dass `count` den Wert `-1` hat. Mit einem `count`–Wert von `-1` ist der Stapel also in einem ehrenwerten Zustand, man kann nur keine Werte holen. Hat `count` aber einen Wert von `-2` oder `1407`, dann ist etwas grundsätzlich schief gegangen. Wir unterscheiden darum

- Ungewöhnliche Situationen: Zustände die eine ungewöhnliche, aber nicht völlig unerwartete Situation darstellen.
- Katastrophen: Zustände die eigentlich niemals auftreten dürften und die nicht mehr wirklich korrigierbar sind.

Eine ungewöhnliche Situation macht die Fortsetzung der laufenden Aktion unmöglich, sollte aber nicht den gesamten Programmlauf abbrechen. Das Programm muss mit einer solchen Situation umgehen können und auf sie vorbereitet sein. Katastrophen werden typischerweise durch Programmier–, Hardware– oder Systemfehler ausgelöst. Man behandelt sie meist am besten durch einen sofortigen Programmabbruch.

Java unterstützt diese Unterscheidung durch zwei Arten von Ausnahmen. Diese beiden Arten spiegeln allerdings die Konzepte *Ungewöhnliche Situation* und *Katastrophe* nicht direkt wider. Sie beziehen sich auf den Umgang des Compilers mit Ausnahmen:

- *Geprüfte Ausnahmen* (*Checked Exceptions*) sind Ausnahmen, die eine erwartete Situation darstellen, die behandelt werden kann und muss. Der Compiler prüft ob eine geprüfte Ausnahme im Programm ordnungsgemäß behandelt oder weitergereicht wird,
- *Ungeprüfte Ausnahmen* (*Unchecked Exceptions*) sind Ausnahmen für die es nicht unbedingt eine andere sinnvolle Behandlung gibt, als den sofortigen Programmabbruch. Ungeprüfte Ausnahmen können im Programmcode behandelt oder explizit weitergereicht werden, sie müssen aber nicht.

Der Grundgedanke dieser Unterscheidung ist folgender: Geprüfte Ausnahmen sind Ausnahmen, die im Programm explizit als Gestaltungsmittel eingesetzt werden. Statt eines Fehlerwertes wird eine Ausnahme geworfen, etc. Der Compiler prüft, ob dieses Gestaltungsmittel auch konsequent verwendet wird. Solche Ausnahmen verwendet man typischerweise zur Behandlung ungewöhnlicher Situationen.

Ungeprüfte Ausnahmen sind Ausnahmen die nicht als Gestaltungsmittel des Programms eingesetzt werden. Sie spiegeln Situationen wider, für die es im Allgemeinen keine sinnvolle Reaktion gibt. Der Compiler prüft darum konsequenterweise auch nicht, ob das Programm eine entsprechende Ausnahmebehandlung enthält. Solche Ausnahmen verwendet man um Fehlersituationen anzuzeigen. Das Programm wird vom Compiler nicht gezwungen solche Ausnahmen zu behandeln, es kann das aber tun, etwa weil eine Behandlung doch möglich, sinnvoll oder notwendig ist.

`RuntimeException` ist eine vordefinierte *ungeprüfte* Ausnahme. Wird sie nicht abgefangen, dann führt sie zum Programmabbruch. Das ist das was wir brauchen, wenn `count` einen unsinnigen (katastrophalen) Wert hat. Der Wert `-1` ist nicht

unsinnig: er zeigt, dass der Stapel leer ist, die `top`–Operation kann nicht ausgeführt werden. Das ist aber kein Grund zum Programmabbruch, der Kunde muss informiert werden, damit er sich etwas anderes überlegen kann. Wir arbeiten darum in unserem Stapel mit zwei unterschiedlichen Ausnahmen, einer geprüften und einer ungeprüften:

```
class Stack {
  ...
  public int top() //liefert das oberste Element
      throws StackEmptyException {
    if (count <= 9 && count >= 0) { // Normalfall
      return a[count];
    } else if ( count == -1 ) // kann gewuenschte Operation
      throw new StackEmpty(); // jetzt nicht ausfuehren
    else                      // Katastrophe (Programmierfehler ?)
      throw new RuntimeException(); // Aus! Schluss! Ende!
  }
  ...
}
```

Die Unterscheidung zwischen Katastrophensituationen und Situationen, die ungewöhnlich, aber behandelbar sind, ist leider nicht immer klar und eindeutig. Den Zugriff auf einen leeren Stapel kann man sowohl als ungewöhnlich, als auch als Programmierfehler betrachten. Es gibt keine klaren Regeln darüber, welche Situtation katastrophal und welche behandelbar ist. Wichtig ist allerdings, die Entscheidung darüber innerhalb eines Programms einheitlich zu treffen. Der Zugriff auf einen leeren Stapel sollte also konsistent vom Stapel und seinem Benutzer entweder als Programmierfehler oder als ungewöhnliche Situation betrachtet werden.

An der Aufrufstelle muss also `StackEmptyException` abgefangen und behandelt werden, die `RuntimeException` dagegen nicht:

```
public static void main(String[] args) {
Stack s = new Stack();
int i = 1;
     for ( int i=0; i<10; i++ )
           s.push(i);
try {
  while ( true ) {
    System.out.println(s.top(i));
    s.pop();
  }
} catch(StackEmptyException e) {
  System.out.println("Das war alles");
}
}
```

Das Ausnahmekonzept von Java setzen wir im Allgemeinen so um, dass “Katastrophe” mit “ungeprüft” und so mit “muss nicht behandelt werden” verbunden wird. Das ist meist sinnvoll. In der Regel handelt es sich bei der Katastrophe um einen Programmierfehler, der möglichst schnell entdeckt und nicht durch irgendeine Ausnahmebehandlung vertuscht werden sollte. Aber das gilt nicht immer. Ein Flugsteuerungsprogramm beispielsweise bricht man auch dann besser nicht einfach komplett ab, wenn sich im Programmlauf ein Programmierfehler zeigt. In diesem Fall muss man sehen, dass es trotzdem irgendwie weitergeht. Auch eine *ungeprüfte Ausnahme* kann und muss also gelegentlich behandelt werden.

Laufzeitprüfungen

Es ist nicht notwendig, sein Programm mit Prüfungen auf alle möglichen Arten von Programmierfehlern zu zupflastern. Viele besonders häufige Fehler werden durch *Laufzeitprüfungen* automatisch entdeckt, sie führen zu ungeprüften Ausnahmen und damit einem Programmabbruch. So wird der Zugriff auf eine `null`–Referenz mit einer `NullPointerException` quittiert. Auch der Zugriff auf ein Feld wird automatisch daraufhin überwacht, ob der Index in seinen Grenzen liegt. Wenn nicht, dann gibt es eine `IndexOutOfBoundsException`.

Die `top`–Operation des Stapels kann darum darauf verzichten, den Feldzugriff explizit zu prüfen:

```
final class Stack {
  ...
  public int top() //liefert das oberste Element
      throws StackEmptyException {
    if ( count == -1 )     // Stapel ist leer
        throw new StackEmpty(); // Aufrufer tu was anderes!
```

```
    return a[count];       // illegale Werte von count
  }                        // fuehren zu IndexOutOfBoundsException:
  ...                      // Programmfehler: Schluss mit allem
}
```

Geprüfte Ausnahmen: throws

Jede Ausnahme führt zum Abbruch der laufenden Aktion. Die laufende Aktion ist oft eine Methode. Wenn eine geprüfte Ausnahme eine Methode abbrechen kann, dann muss das Schlüsselwort `throws` im Kopf der Methode erwähnt werden. In unserem Beispiel kann `top` die Ausnahme `StackEmptyException` werfen und darum:

```
public int top() throws StackEmptyException {
  ....
}
```

Mit `throws` deklariert die `top`–Methode, dass sie die `StackEmpty` Ausnahme nicht selbst behandeln will, sondern an ihren Aufrufer weitergibt. Dieser kann sie dann behandeln oder selbst wieder weitergeben. Behandelt er sie nicht selbst, dann muss sie mit `throws` deklariert werden:

```
public void topCaller(Stack s)
    throws StackEmptyException {
  ....
  s.top();
  ....
}
```

Das Weitergeben darf sich sogar `main` erlauben:

```
public static void main(String[] args) throws StackEmptyException {
Stack s = new Stack();
int i = 1;
    for ( int i=0; i<10; i++ )
        s.push(i);
while ( true ) {
  System.out.println(s.top(i));
  s.pop();
}
}
```

Im Endeffekt entspricht das dann natürlich einer unbehandelten Ausnahme.

Ausnahmen fangen: try – catch

Ausnahmen können abgefangen werden. Das Programmstück, in dem sie auftreten können, wird dazu in einen `try`–Block eingeschlossen, dem eine `catch`–Klausel folgt:

```
Stack s = new Stack();
s.push(2);
...
try {
   while ( true ) {
     System.out.println(s.top());
     s.pop();
   }
} catch (StackEmptyException e) {
    System.out.println("Das waren alle Elemente");
}
```

Gerpüfte Ausnahmen müssen irgendwo gefangen werden. Ungeprüfte Ausnahmen kann man fangen, muss aber nicht.

Ein `catch`–Klausel kann auch mehr als eine Ausnahme auflisten. Nehmen wir an unser Stack wird in einem Flugsteuerungsprogramm eingesetzt in dem auch Programmfehler abgefangen werden müssen, und wir wollen auch ungeprüfte Ausnahmen fangen:

```
  ...
  try {
    while ( true ) {
      System.out.println(s.top());
      s.pop();
    }
  } catch (StackEmptyException e) {  // gepruefte Ausnahme
    System.out.println("OK Das waren alle Elemente");
  } catch (IndexOutOfBoundsException e) { // ungepruefte Ausnahme
    System.err.println("OH jeh"+e);
    ... Notlandung einleiten ...
  }
```

finally

Ein `try`–Block kann noch mit einer anderen Klausel abgeschlossen werden: der `finally`–Klausel. Die Anweisungen einer `finally`–Klausel werden immer ausgeführt, egal, ob eine behandelte, eine unbehandelte oder keine Ausnahme aufgetreten ist.

```
try {
  ...
  s.top());
  ...
} catch (StackEmptyException e) {
    System.out.println("Sorry: Stack leer");
} finally {
    System.out.println("Fertig!");
}
```

Dieses Programmstück gibt `Fertig` in jedem Fall aus: Egal ob `top` eine `StackEmptyException` geworfen hat oder nicht und sogar auch dann, wenn in `top` mit einer `IndexOutOfBoundsException` abgebrochen wurde. Man beachte dass in diesem letzten Fall ein Programmstück ohne `finally`

```
try {
  ...
  s.top());
  ...
} catch (StackEmptyException e) {
    System.out.println("Sorry: Stack leer");
}                                  // keine finally-Klausel
System.out.println("Fertig!");     // wird nicht in jedem Fall ausgefuehrt
```

keine `Fertig`–Meldung ausgeben würde.

Ausnahmeklassen definieren

Java bietet vordefinierte Ausnahmeklassen aus beiden Kategorien an. Die wichtigsten sind:

- `Exception` *geprüfte* Ausnahme.
- `RuntimeException` *UNgeprüfte* Ausnahme.

Alle vordefinierten Ausnahmeklassen erlauben die Übergabe einer Diagnosemeldung als String an den Konstruktor. Sie können zudem auch ausgegeben werden:

```
try {
  ...
    ...
    throw new Exception("Weiss nicht weiter hier");
   ...
  ...
} catch (Exception e) {
    System.out.println(e);
}
```

Anwendungen können neue Ausnahmeklassen definieren: Geprüfte als Ableitungen von `Exception`, ungeprüfte als Ableitungen von `RuntimeException`.

```
public class MyCheckedException extends Exception {}
public class MyUncheckedException extends RuntimeException {}
```

Mit der Definition von abgeleiteten Klassen wollen wir uns aber später beschäftigen. Wir belassen es also beim Beispiel und darum hier noch noch einmal zur Übersicht das Stapelbeispiel:

```
public final class Stack {

  public class StackEmptyException extends Exception {}
  public class StackFullException extends Exception {}

  private int count; // aktuelle Zahl der Elemente -1

  private int[] a = new int[10]; // Ablageplatz fuer die Elemente

  // a[0]...a[count] sind die gestapelten Elemente
  // a[count] ist das oberste, a[0] das unterste
  // INV: -1 <= count <= 9

  public Stack() { // Konstruktor, erzeugt den leeren Stapel
    count = -1;
  }

  public int top() //liefert das oberste Element
      throws StackEmptyException {
    try {
      if ( count == -1 ) throw new StackEmptyException();
      return a[count];
    } catch (IndexOutOfBoundsException e) {
      throw new RuntimeException();
    }
  }

  public void push(final int x) // legt Element ab
      throws StackFullException {
    if (count == 9)
      throw new StackFullException();
    if (count < 9) {
      count++;
      a[count] = x;
    }
  }

  public void pop() { // entfernt oberstes Element falls vorhanden
    if (count >= 0) {
      count--;
    }
  }

  public static void main(String[] args) {
    Stack s = new Stack();
    try {
      int i=1;
      while ( true ) {
        s.push(i);
        i++;
      }
    } catch (StackFullException e) {
      System.out.println("Stack Voll");
    }
    try {
      while ( true ) {
        System.out.println(s.top());
        s.pop();
      }
    } catch (StackEmptyException e) {
```

```
        System.out.println("Das waren alle Elemente");
      }
    }
  }
```

Die Ausnahmeklassen wurden hier sehr minimalistisch definiert. Eine etwas ausführlichere Variante, die sich an die Konventionen von Java hält wäre:

```
  class final StackEmptyException extends Exception {
    public StackEmptyException() {
       super("Stack Empty");
    }
    public StackEmptyException(String msg) {
       super(msg);
    }
  }
```

Wir gehen hier nicht weiter auf diese Definition ein, da sie Konzepte der Vererbung nutzt, die erst später behandelt werden.

Error

Java kennt neben den Ausnahmen noch die Klasse `Error`. Formal gesehen handelt es dabei nicht um Ausnahmen, aber `Error`–Objekte werden wie Ausnahmen geworfen und können abgefangen werden. Ein Beispiel ist `OutOfMemoryError`. Dieser *Fehler* wird geworfen, wenn der virtuellen Maschine der Speicher ausgeht. Im Sinne unserer Diskussion von oben entspricht ein `Error` natürlich einer Katastrophe. Fehler sind Katastrophen der besonders schlimmen Art, die wir Java überlassen. Wir werfen im Programm nicht explizit einen `Error` und im Gegensatz zu einer `RuntimeException`, deren Behandlung vom Compiler ja auch nicht erzwungen wird, denken wir nicht einmal darüber nach, ob wir sie nicht doch behandeln wollen.

Multi–Catch

In Java 7 ist die *catch*–Klausel dahingehend erweitert, dass mehrere Exceptions zu einem *Multi–Catch* zusammengefasst werden können. So kann man beispielsweise zwei Ausnahmen gleichzeitig fangen:

```
Stack s = new Stack();
try {
  int z1 = Integer.parseInt(JOptionPane.showInputDialog("Zahl 1 ?"));
  s.push(z1);
} catch (NumberFormatException | IllegalStateException e) {
  JOptionPane.showMessageDialog(null, "Operation kann nicht ausgefuehrt werden.");
}
```

2.4 Spezifikation von Klassen

2.4.1 Spezifikation wertorientierter Klassen

Spezifikation einer Funktion und Funktionale Abstraktion

Funktionen kennen wir aus der Mathematik und eventuell auch aus dem wirklichen Leben. Es sind Verfahren, nach denen aus gegebenen Werten, den Argumenten, ein neuer Wert, das Resultat, berechnet wird. Wir können dabei das Konzept einer Funktion und deren Implementierung unterscheiden. Die Funktion *wurzel* ist beispielsweise konzeptionell die Funktion, die zu ihrem Argument ein Ergebnis liefert, das mit sich selbst multipliziert das Argument ergibt. Weiter oben, im Kapitel über Funktionen, haben wir gelernt, dass Informatiker dies *Spezifikation* und Implementierung nennen und, dass Funktionen (Methoden) in vertragsorientierter Form mit Vor– und Nachbedingung spezifiziert werden. Bei der *wurzel*–Funktion könnte die Spezifikation als Kommentar im Programm auftreten und etwa wie folgt aussehen:

```
/**
 * Berechnet die Wurzel
 * @param x die Zahl aus der die Wurzel gezogen wird
 * @pre  x >= 0
 * @return ein y mit x-0.0001 < y*y < x+0.0001
 * @post -
 */
public static double wurzel(final double x) {
          ...
}
```

Zu dieser Spezifikation kann es viele unterschiedliche Implementierungen geben. Alle diese Implementierungen erfüllen die Spezifikation. Die Spezifikation ist damit eine Eigenschaft der Implementierungen. Unabhängig davon kann man sich nun fragen, ob es das Spezifizierte auch unabhängig von den Implementierungen gibt. Gibt es also so etwas wie

Die Wurzelfunktion mit der Genauigkeit von 0,0001

oder haben wir es lediglich mit Stücken von Java–Programmtexten zu tun, die die Eigenschaft

erfüllt die Spezifikation
Pre: $x \geq 0$
Post: **returns** y, $x - 0.0001 < y * y < x + 0.0001$

haben. Nun die Frage ist sehr akademisch. Speziell für Informatiker, die Abstraktionen so sehr lieben und für die der Unterschied zwischen virtuell und real rein virtuell ist. Natürlich gibt es die spezifizierte Funktion unabhängig von jeder Implementierung: Es handelt sich dabei um eine *funktionale Abstraktion* oder kurz und volkstümlich, um eine *Funktion*.

Zusammengefasst:

- (Java–) Funktionen / Methoden werden vertragsorientiert spezifiziert.
- Die Spezifikation beschreibt eine *Funktion (im mathematischen Sinn)*, die *funktionale Abstraktion*.
- Korrekte (Java–) Funktionen / Methoden erfüllen die Spezifikation und implementieren damit die *Funktion*.

$$\textit{Spezifikation} \overset{\textit{beschreibt}}{\longrightarrow} \textit{Funktion}$$
$$\textit{Java-Methode} \overset{\textit{implementiert}}{\longrightarrow} \textit{Funktion}$$

Eine **Funktion** (**funktionale Abstraktion**) ist eine Zuordnung von Argumenten zu Werten. Sie wird vertragsorientiert spezifiziert und als Java-Methode implementiert.

Eine Klasse als Datenabstraktion

Bei Klassen kann man die gleichen Überlegungen anstellen:

- *Wie wird eine Klasse spezifiziert?* und
- *Was ist das, das da spezifiziert wird?*

Bei Funktionen/Methoden ist der Zusammenhang zwischen dem, das spezifiziert wird, und seinen Implementierungen so offensichtlich und eindeutig, dass er kaum wahrgenommen wird. $Funktionen_{Mathematik}$ werden spezifiziert und als $Funktionen/Methoden_{Java}$ implementiert.

Bei Klassen ist die Sache etwas komplizierter. Wie eine Klasse implementiert wird, wissen wir. Aber was implementiert sie eigentlich? In Java und im wirklichen Leben gibt es Funktionen. Das eine ist eine Implementierung des anderen. Aber gibt es in der Mathematik oder im "Leben" auch Klassen?

Im letzten Abschnitt haben wir etwas einfaches Mathematisches mit einer Klasse implementiert, die Vektoren im Raum:

```
public final class Vektor {
  private final double x;
  private final double y;
  private final double z;

  public Vektor() { x=1; y=1; z=1; }
  public Vektor(final double i, final double j, final double k) {
    x=i; y = j; z =k;
  }
  public double laenge () {
    return Math.sqrt(x*x+y*y+z*z);
  }
  public static Vektor add(final Vektor x, final Vektor y) {
     return new Vektor(x.x + y.x, x.y + y.y, x.z + y.z);
  }
  ... etc. ...
}
```

Diese Klasse implementiert das Konzept der mathematischen Vektoren: Eine Menge von Werten mit Operationen auf diesen Werten. Die Zuordnung *Argumente* $\rightarrow$ *Wert*, unabhängig von einer konkreten Implementierung und einem konkreten Algorithmus, nennt man *Funktion*. Eine Kollektion von *Werten* mit *zugehörigen Funktionen*, unabhängig von einer konkreten Implementierung der Werte und der Funktionen hat auch einen Namen: man nennt es *abstrakten Datentyp* oder kurz *ADT*. Also $Vektor_{Java}$ implementiert $Vektor_{ADT}$:

$$\textit{Java-Klasse } \texttt{Vektor} \overset{implementiert}{\longrightarrow} \textit{ADT Vektor}$$

Bei Funktionen ist alles recht einfach. Das Konzept einer Funktion ist recht intuitiv, wird von der Menschheit seit Jahrhunderten verwendet und von uns in jahrzehntelangem Mathematikunterricht eingeübt. Software–Konstrukte zur Implementierung von Funktionen sind seit etwa 60 Jahren allgemein bekannt und üblich.

Bei ADTs und Klassen dagegen sieht die Sache etwas anders aus. Das Konzept der abstrakten Datentypen wird erst seit etwa 40 Jahren in der Softwareentwicklung verwendet. Im Gegensatz zu Funktionen begegnen uns ADTs im alltäglichen Leben eher selten. Klassen zur Realisation von ADTs sind dazu erst seit etwa 20 Jahren allgemein üblich und vor allem sind Klassen nicht die einzige mögliche Art einen ADT zu implementieren und umgekehrt sind ADTs nicht das einzige, was man mit einer Klasse implementieren kann.

Spezifikation einer Datenabstraktion

Eine Klasse kann einen ADT implementieren, so wie eine Methode eine Funktion implementieren kann. Um entscheiden zu können, ob sie das richtig macht, also korrekt ist, benötigen wir eine Spezifikation. Spezifiziert wird immer das gewünschte Verhalten. Wir spezifizieren also einen ADT, implementieren ihn als Klasse und fragen uns dann, ob die Implementierung korrekt ist. Wie spezifiziert man einen ADT?

Ein ADT hat Werte und Funktionen. Die Funktionen spezifizieren wir in der gewohnten Art mit Vor– und Nachbedingungen. Die Spezifikation der Werte muss festlegen, welche Werte es überhaupt geben soll. Dazu stehen zwei Vorgehensweisen zur Auswahl. Eine "algebraische" und eine "modellorientierte". Nach der *algebraischen Methode* werden die Werte indirekt nur über ihre Eigenschaften charakterisiert. Bei der *modellorientierten* Vorgehensweise definiert man die Werte durch ein "Modell". Bei den Vektoren etwa sagt man, ein Vektor ist ein Tripel von Zahlen. Das Modell ist eine Darstellung / Modellierung der zu definierenden Werte durch Bekanntes. Das Bekannte ist dabei vorzugsweise von mathematischer Natur. Modellorientierte Spezifikationen von Werten sind einfacher zu verstehen als die algebraischen. Algebraische ADT–Spezifikationen betrachten wir darum nicht weiter.

Ein "Modell" hat große Ähnlichkeit mit einer Implementierung. Es gibt aber subtile Unterschiede zwischen beiden:

- In einer *Implementierung* der Vektoren legen wir beispielsweise fest, dass Vektoren als Objekte mit drei `double`–Komponenten (Variablen der Sprache Java) dargestellt werden.
- In einer Spezifikation des ADTs Vektoren sagen wir, dass Vektoren durch drei reelle Zahlen (mathematische Werte) modelliert werden.

Es liegt in diesem einfachen Fall dann zwar nahe, Vektoren die als Tripel von reellen Zahlen spezifiziert wurden mit drei Variablen vom Typ `double` zu implementieren, es bleibt aber explizit offen, wie die Implementierung aussieht und es kann auch eine völlig andere Darstellung gewählt werden. Für die Korrektheit der Implementierung ist es völlig belanglos, ob die Objekte, die Vektoren implementieren, die gleiche oder eine ähnliche Struktur haben, wie das in der Spezifikation gewählte Modell. Das einzige Kriterium der Korrektheit einer Implementierung ist deren von außen beobachtbares Verhalten.

Ein **abstrakter Datentyp** (**ADT, Datenabstraktionen**) ist eine Menge von Werten mit zugehörigen Funktionen. Er wird spezifiziert

- durch ein Modell der Werte (abstrakte Werte) und
- durch eine vertragsorientierte Spezifikation der Funktionen.

Spezifikation des ADT Vektor

Betrachten wir noch einmal die Vektoren. Die Spezifikation ist offensichtlich und soll auch zeigen, dass wir hier von einfachen und offensichtlichen Dingen reden. Das einzige, was vielleicht ein wenig ungewohnt ist, ist die strenge Unterscheidung zwischen Spezifikation und Implementierung. Beginnen wir mit der Spezifikation. Spezifikationen von abstrakten Datentypen können in UML verfasst werden. Für unseren ADT *Vektoren* sieht das wie folgt aus:

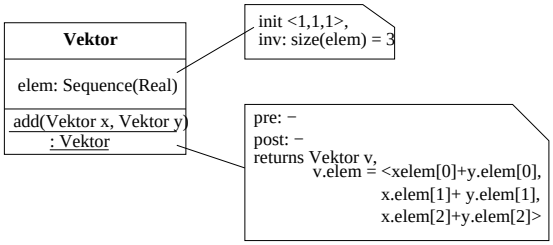


Abbildung 2.9: Spezifikation des ADT Vektor in UML mit Annotationen

Wir sehen hier ein Klassendiagramm mit zwei Annotationen. Im Diagramm wird definiert, dass ein Exemplar der Klasse *Vektor* das Attribut *elem* vom Typ *Sequenz von Real–Werten* hat. Eine solche Sequenz ist ein *abstrakten Wert*. In der Annotation wird erklärt, dass der abstrakte Wert mit drei Einsen zu initialisieren ist und, dass er immer die Lange drei hat. Die Spezifikation der Methode erfolgt im üblichen vertragsorientierten Stil.

Wir haben es hier mit einem annotierten Klassendiagramm zu tun. Das Piktogramm der Klasse wird dabei mit Annotationspiktogrammen angereichert, in denen die Spezifikation eingetragen wird. Die verwendete Notation nennt sich *OCL* (*Objekt Constraint Language*). OCL ist eine textuelle Notation, die als Bestandteil der UML deren graphische Notation ergänzt. OCL–Annotationen (Anmerkungen) können als Annotationspiktogramme in die UML–Diagramme eingetragen werden. Alternativ dazu kann ein Klassendiagramm ohne Annotationen erstellt werden. Die Annotationen werden dann in einem gesonderten Textdokument vermerkt. In unserem Fall wäre dies:

```
context Vektor
  inv: size(elem) = 3

context Vektor.elem
  init: <1,1,1>        ;; entspricht init: Sequence{1,1,1}

context Vektor.add
  pre: –
  post: –
  returns: Vektor v;
         v.elem = < x.elem[0]+y.elem[0],
                  x.elem[1]+y.elem[1],
                  x.elem[2]+y.elem[2] >
```

Mit **context** wird Bezug auf ein Element des Klassendiagramms genommen. Das kann eine Klasse oder auch ein Element innerhalb einer Klasse sein. Mit **inv** wird eine Invariante bezeichnet, also eine Eigenschaft die sich niemals ändert. In unserem Falle muss die Liste der Vektorwerte immer die Länge drei haben. Klar, unsere Vektoren haben ja drei Koordinaten. Mit **init** wird ein Initialwert angegeben.

ADTs wie *Vektor* können also in UML in Form einer (UML–) Klasse spezifiziert werden. Das Klassendiagramm wird dazu mit Annotationen versehen. Die Annotationen können Bestandteile des Klassendiagramms sein (in Annotationspiktogrammen) oder in einem gesonderten Textdokument vermerkt werden.

Implementierung des ADT Vektor

Als Implementierung unserer Spezifikation nehmen wir die (Java–) Klasse:

```
public final class Vektor {
  private final double x; // konkrete Darstellung
  private final double y; // der Werte
  private final double z; // realisiert abstrakten Wert

  // Default-Konstruktor realisiert init
  public Vektor() { x=1; y=1; z=1; }


  // Methode realisiert spezifizierte Funktion
  public static Vektor add(final Vektor x, final Vektor y) {
     return new Vektor(x.x + y.x, x.y + y.y, x.z + y.z);
  }
}
```

In der Implementierung werden den abstrakten Werten und Funktionen konkrete Java–Konstrukte gegenübergestellt. Die in der Spezifikation verwendeten Sequenzen (Listen) der Länge drei werden als drei konstante Objektvariablen realisiert. Es ist klar, eine fixe Liste mit drei Elementen kann mit drei Variablen dargestellt werden. Wie eine Spezifikation implementiert wird liegt im Ermessen des Implementierers. Insbesondere können die abstrakten Werte in beliebiger Form konkret dargestellt werden. Ein zulässige alternative Implementierung ist darum:

```
public final class Vektor {
   private final double[] v = new double[3];
   Vektor() {
     v[0] = 1.0; v[1] = 1.0; v[2] = 1.0;
   }
   Vektor(final double x, final double y, final double z) {
     v[0] = x; v[1] = y; v[2] = z;
   }
   public static Vektor add(final Vektor x, final Vektor y) {
     return new Vektor( x.v[0] + y.v[0],
                        x.v[1] + y.v[1],
                        x.v[2] + y.v[2]);
  }
}
```

Ein ADT wird spezifiziert und als Klasse implementiert. Das Ziel ist die Klasse, deren Grundlage ist die Spezifikation. Es ist darum zwar oft etwas ungenau, aber allgemein üblich, davon zu sprechen, dass die Klasse spezifiziert wird.

> **Implementierung eines ADT als Klasse:**
> Ein **abstrakter Datentyp** wird durch eine wertorientierte Klasse implementiert. Den abstrakten Werten entsprechen Objekte dieser Klasse. Den Funktionen entsprechen die Methoden der Klasse.

Signaturdiagramm eines ADT

Mit einem *Signaturdiagramm* wird die Signatur eines ADTs dargestellt: Welche Funktionen mit welchen Argument– und Ergebnistypen gibt es? Ein Beispiel ist 2.10.

In einem Signaturdiagramm werden Typen und Funktionen dargestellt. Pfeile von Typen zu einer Funktion zeigen an welche Parameter die Funktion hat und der Pfeil von der Funktion zu einem Typ, zeigt den Ergebnistyp an.

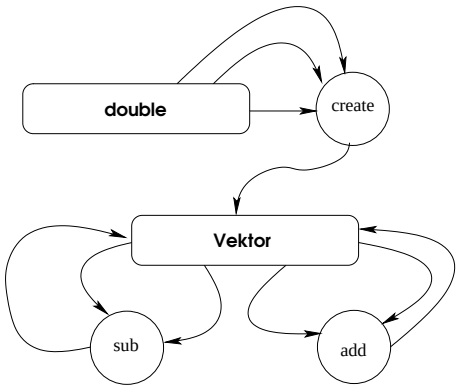


Abbildung 2.10: Signaturdiagramm

Abstraktionsfunktion und Korrektheit der Implementierung eines ADT

Um die Korrektheit der Implementierung einer ADT–Spezifikation zu zeigen, müssen wir als erstes darlegen, in welcher Beziehung abstrakte und konkrete Werte stehen. Genau gesagt müssen wir festlegen, was die konkreten Werte bedeuten sollen. In unserem Vektorbeispiel ist das einfach und offensichtlich: In der ersten Implementierung haben wir drei Variablen `x, y` und `z`. Deren Werte sollen die drei Vektorkomponenten darstellen:

Wert(`x`)	$\rightarrow$	elem[0]
Wert(`y`)	$\rightarrow$	elem[1]
Wert(`z`)	$\rightarrow$	elem[2]

Diese Zuordnung wird *Abstraktionsfunktion* genannt. Sie gibt kurz gesagt an, welche Bedeutung die Objektvariablen in ihrem aktuellen Zustand (Wert) im Sinne der Spezifikation haben. Umgekehrt sind `x, y` und `z` (in der Vektor–Implementierung mit drei Objektvariablen `x`, `y` und `z`, mit ihren aktuellen Werten, die konkrete *Repräsentation* des abstrakten Wertes *elem*. In unserem Beispiel repräsentiert also ein

Repräsentant:	Objekt `o` vom Typ `Vektor` mit `o.x` = 1, `o.y` = 2 und `o.z` = 3
Repräsentiertes:	einen *Vektor v* mit *v.elem* = *Sequence*$\{1,2,3\}$.

> Die **Abstraktionsfunktion** bildet Objekte einer Klasse, die einen ADT implementiert, auf die (abstrakten) Werte des ADTs ab. Sie gibt an, welcher abstrakte Wert von einem Objekt mit seiner aktuellen Belegung der Objektvariablen **repräsentiert** wird.

Mit einer Abstraktionsfunktion können wir definieren, wann eine Implementierung eines ADTs korrekt ist: Die Implementierung eines ADTs ist korrekt, wenn die Implementierung $f_{konkret}$ einer spezifizierten Funktion f_{Spez} Repräsentanten der Argumente von f_{Spez} in einen Repräsentanten des Ergebnisses von f_{Spez} überführt.

In unserem Beispiel repräsentieren beispielsweise die beiden Objekte `v1` und `v2` mit:

```
v1:
  - x = 1.0,
  - y = 2.0,
  - z = 3.0

v2:
  - x = 2.0,
  - y = 3.0,
  - z = 4.0
```

die Vektoren $v1_{Spez} =< 1,2,3 >$ und $v2_{Spez} =< 2,3,4 >$. Wenden wir nun die Methode `Vektor.add` auf `v1` und `v2` an, dann erhalten wir das Objekt `v3` mit:

```
v3:
  - x = 3.0,
  - y = 5.0,
  - z = 7.0
```

was offensichtlich ein Repräsentant von $v3_{Spez} = add(v1_{Spez}, v2_{Spez}) =< 3,5,7 >$ ist.

> Die **Implementierung** eines ADTs ist korrekt, wenn die Implementierung f_k jeder Funktion f_A des ADTs Repräsentanten a_k der Argumente a_A in Repräsentanten r_k des Ergebnisses $r_A = f_A(a_A)$ abbildet:
>
> $$A(f_k(a_k)) = f_A(A(a_k))$$
>
> wobei A die Abstraktionsfunktion ist.

Die hier aufgefahrene Begrifflichkeit entspricht ganz und gar dem intuitiv Einsichtigen. Sie dient in erster Linie dazu, klar über unsere Software–Entwürfe und die entsprechenden Implementierungen reden zu können. Auch in den Fällen die nicht so einfach sind wie Vektoren.

2.4.2 Spezifikation zustandsorientierter Klassen

Vom Wert zum Automaten

Wertorientierte Klassen realisieren ADTs: Werte und Funktionen, die von Werten zu neuen oder anderen Werten führen. Werte sind unveränderlich. Sie haben keinen, bzw. immer den gleichen Zustand. Können sich Werte verändern, dann sind es keine Werte mehr. Wir könnten sie *Objekte* nennen, aber um deutlich zu machen, dass noch keine Konstrukte einer Programmiersprache gemeint sind, sprechen wir hier lieber von Automaten.

Ein *Automat* ist ein Ding, das sich in verschiedenen Zuständen befinden kann, auf externe Einflüsse reagiert und dabei (eventuell) einen Wert produziert und (eventuell) in einen neuen Zustand übergeht. Ein Konto ist ein Automat. Wird es angelegt, dann befindet es sich im Zustand *leer*. Zahle ich etwas ein, dann geht es in den Zustand *nicht leer* über und hebe ich mehr ab als ich eingezahlt habe, dann befindet es sich im Zustand *überzogen*.

Automaten werden mit Hilfe von Klassen implementiert. Ein Objekt der Klasse stellt einen Automaten dar. Die Zustände des Automaten werden als Zustände des Objekts repräsentiert. Der Zustand eines Objekts ist dabei durch die aktuelle Belegung der Objektvariablen gegeben.

Zustandsdiagramm

Automaten sind ein sehr altes weitverbreitetes Konzept, das schon sehr lange zur Spezifikation von Systemen unterschiedlichster Art eingesetzt wird. Die übliche graphische Notation sind *Zustandsdiagramme* (auch *Zustandsübergangsdiagramm*, engl. *state chart*). Zustandsdiagramme sind Bestandteil von UML. In ihnen werden dargestellt:

- *Zustände:* die Zustände, in denen sich ein Automat befinden kann.
- *Ereignisse:* die Ereignisse, die einen Automaten in einem bestimmten Zustand zu einem Übergang in einen neuen Zustand veranlassen können,
- *Aktionen:* die Aktionen, die der Automat beim Übergang in den neuen Zustand ausführt.

Ein einfaches Beispiel ist der Schalter (siehe 2.11).

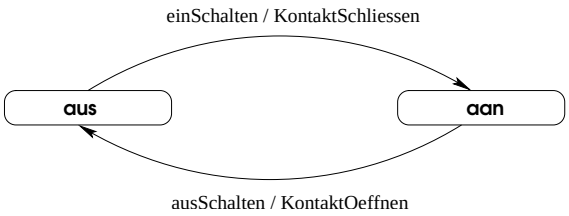


Abbildung 2.11: Zustandsdiagramm eines Schalters

Ein Schalter kann in zwei Zuständen sein: *an* oder *aus*. Im Zustand *an* akzeptiert er das Ereignis *einschalten*. Im Zustand *aus* akzeptiert er das Ereignis *ausSchalten*. Ereignisse führen zu Aktionen und Zustandsübergängen. Tritt im Zustand *aus* das Ereignis *einschalten* auf, dann führt der Automat (ein Schalter) die Aktion *KontaktSchliessen* aus.

Man beachte, dass ein Zustandsdiagramm völlig andere Aussagen macht als ein Signaturdiagramm. In einem Signaturdiagramm kommen Werte nicht vor. Ein Zustandsdiagramm redet dagegen nur über Automaten, das veränderliche Äquivalent von Werten.

Spezifikation eines Automaten mit annotiertem Klassendiagramm

Graphische Notationen sind oft hilfreich. In komplexeren Fällen, in denen viele Details zu beachten sind, wird man aber eine textuelle Notation bevorzugen, oder zumindest als Ergänzung der Graphiken verwenden wollen. Neben Zustandsdiagrammen können darum in UML auch Klassendiagramme mit Annotationen zur Spezifikation von Automaten verwendet werden (siehe 2.12).

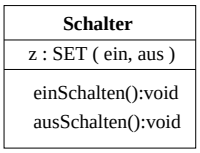


Abbildung 2.12: Klassendiagramm eines Schalters

Die Annotationen zu diesem Diagramm verfassen wir in rein textueller Notation:

```
context Schalter.z
  init: aus

context Schalter.einSchalten
  pre: z = aus
  post: z = ein

context Schalter.ausSchalten
  pre: z = ein
  post: z = aus
```

Die Aktionen *KontaktSchliessen* und *KontaktOeffnen* bleiben hier (zunächst einmal) unerwähnt, da sie zu keinem von außen beobachtbaren Verhalten führen.

Ein **Automat** hat einen Zustand und reagiert auf Ereignisse mit Aktionen und Zustandsübergängen. Er wird spezifiziert

- durch ein Modell der Zustände (abstrakte Zustände) und
- durch eine vertragsorientierte Spezifikation der Ereignisse als Methoden.

Beispiel: Spezifikation von Konten

Betrachten wir ein etwas interessanteres Beispiel: Ein Konto hat einen Kontostand, dessen Wert seinen aktuellen Zustand ausmacht. Man kann den aktuellen Kontostand abfragen und einen Betrag *b* einzahlen oder abheben (siehe 2.13).

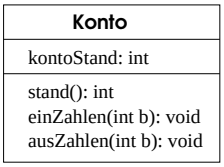


Abbildung 2.13: Klassendiagramm eines Kontos

Die Annotationen schreiben wir wieder textuell:

```
context Konto
  init: kontoStand = 0
  inv: kontoStand >= 0

context Konto.stand
  pre: -
  post: -
  returns: kontoStand
```

```
context Konto.einZahlen
  pre: b > 0
  post: kontoStand = kontoStand@pre + b

context Konto.ausZahlen
  pre: b > 0, kontoStand >= b
  post: kontoStand = kontoStand@pre - b
```

Hier wird festgelegt, dass ein Konto mit einem Kontostand null erzeugt wird. Ein negativer Kontostand ist nicht erlaubt. Ein– und Auszahlen kann man nur positive Beträge. Auszahlungen sind nicht erlaubt, wenn der Betrag den Kontostand übersteigt. Die Wirkung der Methoden wird mit Nachbedingungen beschrieben. In den Methoden `einZahlen` und `ausZahlen` wird der Zustand verändert. Die Spezifikation gibt den neuen Kontostand an.

```
  ...
  post: kontoStand = kontoStand@pre + b
  ...
  post: kontoStand = kontoStand@pre - b
  ...
```

Es ist der Kontostand der beim Verlassen der Methode aktuell sein soll. Bei der Definition des neuen Wertes wird auf den Parameter `b` und den alten Wert, den Kontostand beim Eintritt in die Methode (und der Auswertung der Vorbedingung) `kontoStand@pre`, Bezug genommen.

Genau genommen wird hier ein Typ von Automaten mit dem Namen *Konto* spezifiziert. Ein Konto ist eine Instanz dieses Typs: ein Automat vom Typ *Konto*. Die übliche Sprechweise ist weniger genau: Die Unterscheidung zwischen Typ und Exemplar eines Typs wird oft vernachlässigt. Man spezifiziert dann "ein Konto als Zustandsautomat". Gegen eine so vereinfachte Sprache ist nichts einzuwenden, wenn alle Beteiligten sich darüber klar sind, wovon tatsächlich die Rede ist.

Implementierung von Automaten

Der Automatentyp *Konto* kann als Klasse `Konto` in Java implementiert werden. Einzelne Objekte, Instanzen der Klasse Konto, repräsentieren dann einzelne Automaten. Automaten werden genauso implementiert wie ADTs. Zunächst überlegt man sich eine konkrete Repräsentation der abstrakten Zustände.

Schalter sind in den (abstrakten) Zuständen *ein* und *aus*, die als Elemente der Menge $\{ein, aus\}$ (in UML: *SET(ein, aus)*), spezifiziert wurden. Repräsentationen, konkrete Zustände, können nach Belieben gewählt werden. Etwa in Form einer booleschen Variablen:

```
public final class Schalter {
  private boolean ein = false;
  ...
}
```

oder mit einem Enumerationstyp:

```
public final class Schalter {
  private enum Z { an, aus };
  private Z z = Z.aus;
  ...
}
```

oder wie auch immer. Wichtig ist, dass der Bezug zum abstrakten Zustand entweder klar ist oder ausreichend dokumentiert wird. Bei der Verwendung des Enumerationstyps haben wir eine klare Entsprechung zwischen den Elementen der Menge $\{an, aus\}$ und den möglichen Werten der Variablen `z`. Bei der Verwendung der booleschen Variablen `ein` ist der Bezug indirekter, aber immer noch klar.

Konkreter und abstrakter Zustand sind wie im Fall der ADTs und ihrer Implementierungen durch eine *Abstraktionsfunktion* miteinander verbunden. Die Abstraktionsfunktion bildet den Repräsentanten, den konkreten Zustand, auf den damit gemeinten abstrakten Zustand ab. Bei der Repräsentation des Schalterzustands durch eine boolesche Variable `ein` haben wir beispielsweise die Abstraktionsfunktion **A**:

$$\mathbf{A}(\texttt{ein}) = \begin{cases} z = ein & : \quad \texttt{ein} = true \\ z = aus & : \quad \texttt{ein} = false \end{cases}$$

Die Methoden werden so implementiert, dass sie auf den Repräsentanten die abstrakten Operationen "simulieren". Es ist klar was damit gemeint ist: Wenn eine abstrakte Operation von einen abstrakten Zustand z_A zu einen abstrakten Zustand z'_A führt

dann muss die Implementierung von einem Repräsentanten von z_A zu einem Repräsentanten von z'_A führen. Eine Schalterimplementierung wäre:

```
final class Schalter {
     /**
        ein == false -> z==aus
        ein == true -> z==ein
     */
private boolean ein = false;

     /**
       @pre: z == aus
       @post: z == ein
     */
public void einSchalten() {
  ein = true;
}

     /**
       @pre: z == aus
       @post: z == ein
     */
public void ausSchalten() {
  ein = false;
}
 }
```

Die Methode `einSchalten` beispielsweise führt von `ein == false`, einem Repräsentanten von z = *aus* zu `ein == true`, einem Repräsentanten von z = *ein*.

Es ist sicher klar was gemeint ist. Triviale Klassen wie der Schalter müssen in dieser Weise nicht weiter analysiert werden. In realistischen Fällen ist die Abstraktionsfunktion – als die Darstellung des abstrakten Zustands durch konkrete Objektvariablen weniger offensichtlich. Insgesamt kann man sagen, dass Automaten in gleicher Weise wie ADTs implementiert werden.

Implementierung eines Automaten als Klasse:
Ein **Automaten–Typ** wird durch eine zustandsorientierte Klasse implementiert. Den Automaten entsprechen Objekte der Klasse und den abstrakten Zuständen entsprechen die konkreten Zustände der Objekte. Den Ereignissen entsprechen die Methoden der Klasse.

Das Korrektheitskriterium entspricht ebenfalls dem der ADTs:

Die **Implementierung** eines Automaten–Typs ist korrekt, wenn die Implementierung f_k jedes Ereignisses f_A des Automaten Repräsentanten z_k der abstrakten Zustände z_A in Repräsentanten z_k des neuen Zustands $z_A = f_A(z_A)$ abbildet:

$$A[f_k(z_k)] = f_A(A[z_k])$$

wobei A die Abstraktionsfunktion ist.

Beispiel Warteschlange: Spezifikation

Eine Warteschlange (engl. *Queue*) ist wie ein Stapel eine Speicherkomponente, in dem Elemente abgelegt und später wieder entnommen werden können. Dafür stehen zwei Operationen zur Verfügung: einfügen und entnehmen. Die Warteschlange unterscheidet sich vom Stapel dadurch, dass das zuerst gespeicherte Element auch zuerst entnommen wird. Die Warteschlange speichert Daten nach dem FIFO–Prinzip (FIFO: *First In, First Out*), der Stapel dagegen nach dem LIFO–Prinzip (LIFO: *Last In, First Out*)

Das Konzept einer Warteschlange mit Integer–Werten sei wie folgt (rein textuelle UML–Notation) spezifiziert:

```
CLASS Queue {
  attribut
    entries: Sequence(int)

  method
```

```
    enqueue (int e):void
    dequeue (): int
}

context Queue.entries
   int: <>
   inv: len(entries) <= 10

context Queue.enqueue
   pre: len(entries) < 10
   post: entries = entries@pre + <e>

context Queue.dequeue
   pre: entries != <>
   post: entries = tail(entries@pre)
   returns: head(entries@pre)
```

Eine Warteschlange wird hier als Sequenz (Liste) von int–Werten modelliert. Am Anfang ist die Liste leer. Ihre Länge ist auf maximal 10 begrenzt. Beim Einfügen wird das neue Element an das Ende der Liste angehängt

```
entries = entries@pre + <e>
```

und beim Entnehmen wird das erste Element geliefert

```
head(entries@pre)
```

und die Liste auf ihren Rest gekürzt

```
entries = tail(entries@pre)
```

Statt mit Vorbedingungen wird man eventuell lieber mit Ausnahmen arbeiten:

```
CLASS Queue {
  attribut
    entries: Sequence(int)

  method
    enqueue (int e):void throws IllegalStateException
    dequeue (): int throws IllegalStateException
}

context Queue.entries
   int: <>
   inv: len(entries) <= 10

context Queue.enqueue
   if len(entries) >= 10
          throw IllegalStateException
   pre: -
   post: entries = entries@pre + <e>

context Queue.dequeue
   if entries == <> throw IllegalStateException
   pre: -
   post: entries = tail(entries@pre)
   returns: head(entries@pre)
```

Implementierung der Warteschlange: Abstraktionsfunktion

Eine Warteschlange kann mit Hilfe eines Feldes implementiert werden. Das erste (älteste) Element kann dabei immer im ersten Platz des Feldes zu finden sein. Das letzte liegt irgendwo weiter rechts, je nachdem, wie voll die Schlange ist. Wir benötigen eine weitere Variable, um den Füllstand der Schlange zu bestimmen. Elemente werden links an der ersten Feldposition entnommen und rechts eingefügt. Wird ein Element entnommen, dann muss der Rest nachrücken, damit die korrekte Interpretation des Feldinhalts (Abstraktionsfunktion!) gewahrt bleibt.

Um dieses Nachrücken zu verhindern wählt man oft eine andere Repräsentation des abstrakten Zustands *Sequence(int)*. Man betrachtet das Feld als eine ringförmige Struktur so, als folge hinter dem letzten wieder der erste Platz. Eine solche Interpretation eines Feldes nennt man allgemein einen *Ringpuffer*. Mit einem Ringpuffer kann man endliche beschränkte Folgen (Sequenzen, Listen) von Werten gut repräsentieren.

Statt des Nachrückens merkt man sich die Position des ersten Elementes (linke, untere Position) und die des ersten freien Platzes (rechte, obere Position). Beim Entnehmen rückt die linke (untere) Position eins hoch, modulo der Feldgröße natürlich. Beim Einfügen rückt die rechte (obere) Position eins hoch, ebenfalls modulo der Feldgröße. Um ein Überschreiben zu verhindern halten wir die aktuelle Feldgröße fest.

```
public final class Queue {
     private static final int SIZE = 10;
     private int[] a = new int[10];
     private int count = 0; // Anzahl belegte Plaetze
     private int lower = 0; // erster belegter Platz
     private int upper = 0; // erster freier Platz

     ...
}
```

Die Abstraktionsfunktion beschreibt die Interpretation der Objektvariablen im Sinne der ursprünglichen Spezifikation der Warteschlange als Liste:

$\mathbf{A}$(a, lower, upper) = $< \texttt{a}[\texttt{lower}\ \mathbf{mod}\ 10], \cdots \texttt{a}[\texttt{upper}\ \mathbf{mod}\ 10] >$

Es ist nicht notwendig die Abstraktionsfunktion formal auszudrücken. Ein sinnvoller Kommentar darüber, wie die Objektvariablen zu interpretieren sind, reicht vollkommen aus.

Implementierung der Warteschlange: Klasseninvariante

In der Spezifikation wird verlangt, dass die Warteschlange niemals die Länge 10 überschreitet:

inv :$len(entries) <= 10$

Die Invariante bringt dies als Anforderung an den abstrakten Zustand *entries* zum Ausdruck. In der Implementierung muss entsprechendes für die konkrete Repräsentation dieser Liste gelten. Zusätzlich kann der konkrete Zustand noch weiter eingeschränkt werden, etwa auf solche Werte, die eine sinnvolle Interpretation (Abstraktion) erlauben. Wir formulieren dies als *Klasseninvariante*:

```
public final class Queue {
     /**
      * @invariant: 0 <= count < SIZE
      *          0 <= lower < SIZE
      *          0 <= upper < SIZE
      */
     private static final int SIZE = 10;
     private int[] a = new int[SIZE];
     private int count = 0; // Anzahl belegte Plaetze
     private int lower = 0; // erster belegter Platz
     private int upper = 0; // erster freier Platz


     ...
}
```

Ein `count`–Wert 10 würde gegen die Invariante der Spezifikation verstoßen. Ein Wert kleiner Null oder ein Wert von `upper` und `lower` außerhalb des Bereichs von Null bis zehn erlaubt keine sinnvolle Interpretation: die Abstraktionsfunktion ist undefiniert, wenn `lower` oder `upper` nicht in ihrem Bereich sind.

Implementierung der Warteschlange: Methoden

In der Implementierung der Methoden vermerken wir natürlich die Vor– und Nachbedingungen als Kommentare:

```
public final class Queue {
  /**
   * @invariant: 0 <= count < SIZE
   *          0 <= lower < SIZE
   *          0 <= upper < SIZE
   */
  private static final int SIZE = 10;
  private int[] a = new int[SIZE];
  private int count = 0; // Anzahl belegte Plaetze
```

```
  private int lower = 0; // erster belgter Platz
  private int upper = 0; // erster freier Platz

  /**
   *
   * @param e Wert der einzufuegen ist
   * @throws IllegalStateException falls Schlange voll
   * @pre -
   * @post Warteschlange enthaelt e an ihrem Ende
   */
   public void enqueue(final int e) throws IllegalStateException {
            if ( count >= SIZE ) throw new IllegalStateException("full");
            a[upper] = e;
            upper = (upper+1)%SIZE; ;
            count++;
   }

   /**
    * Liefert den ersten Wert der Warteschlange
    * @return erster Wert der Warteschlange
    * @throws IllegalStateException falls Schlange leer
    * @pre -
    * @post Warteschlange enthaelt den ersten Wert nicht mehr
    */
    public int dequeue() throws IllegalStateException {
            if ( count <= 0 ) throw new IllegalStateException("empty");
            int res = a[lower];
            lower = (lower+1)%SIZE; ;
            count--;
            return res;
    }
}
```

Kapitel 3

Datentypen und Datenstrukturen

3.1 Spezifikation, Schnittstelle, Implementierung

3.1.1 Schnittstelle und Interface

Spezifikation und Schnittstelle

Die Spezifikation beschreibt das Verhalten einer Softwarekomponente. Sei es das *gewünschte* Verhalten einer geplanten Komponente, das dem Implementierer vorgelegt wird, oder das *realisierte* Verhalten, auf das Benutzer der Komponenten sich verlassen können. Eng mit "Spezifikation" verwandt ist der Begriff der "Schnittstelle". Unter einer *Spezifikation* stellt man sich ein Dokument vor, in dem ein Gerät beschrieben wird, während die *Schnittstelle* ein fester Bestandteil des Gerätes ist. Über sie kann man auf die in der Spezifikation beschriebene Funktionalität zugreifen. Diese Unterscheidung mag für eine physische Komponente unproblematisch sein. In der Software, wo alles nur Text und Beschreibung ist, ist sie nicht so völlig klar. Auch Programmtexte sind ja letztlich nur Texte, die ein gewünschtes Verhalten beschreiben. Trotzdem wollen wir bei der intuitiven Unterscheidung zwischen Spezifikation und Schnittstelle bleiben. Auf Seite 56 haben wir dies am Beispiel einer Methode bereits diskutiert.

Die **Schnittstelle** einer Softwarekomponente ist Teil dieser Komponente und enthält das, was zu ihrer Benutzung notwendig ist.

Die **Spezifikation** einer Softwarekomponente ist nicht Teil der Komponente und enthält die Informationen, die zu ihrer Implementierung notwendig sind.

Schnittstelle einer Klasse: paketintern und paketextern

Hier wollen wir uns nun mit Schnittstellen von Klassen beschäftigen. Die Schnittstelle einer Klasse ist der Teil der Klasse, über den sie benutzt werden kann. Benutzen kann man bei einer Klasse alle öffentlichen Methoden, seien sie statisch oder nicht und alle öffentlichen Variablen, seien sie nun statisch oder nicht. Befindet sich der Benutzer im gleichen Paket wie die Klassendefinition, dann kann er nicht nur die öffentlichen Komponenten, sondern darüber hinaus auch die mit Paket–Sichtbarkeit benutzen. Beispiel:

```
public final class C {
  private int x;              // nicht Teil der Schnittstelle
        int y;                // Teil der paketinternen Schnittstelle
  static int z;               // Teil der paketinternen Schnittstelle

             void m1(int x) {...} // Teil der paketinternen Schnittstelle
  public     void m2(int x) {...} // Teil der Schnittstelle
  public static void m3(int x) {...} // Teil der Schnittstelle
  private static void m4(int x) {...} // nicht Teil der Schnittstelle
}
```

Bei dieser Klasse gehört alles außer `x` und `m4` zur Schnittstelle. Die *paketinterne Schnittstelle*, d.h. die Schnittstelle zu Softwarekomponenten die im gleichen Paket definiert sind, besteht aus `y`, `z`, `m1`, `m2` und `m3`. Die *nicht paketinterne Schnittstelle* umfasst dagegen nur `m2` und `m3`.

Schnittstelle einer Klasse: statische Schnittstelle plus Objekt–Schnittstelle

Eine Klasse in Java hat einen doppelten Charakter. Sie ist Modul und Typ ihrer Objekte. Als Modul ist sie "ein Sack voller Definitionen" von Variablen und Funktionen. Das ist der statische Anteil der Klassendefinition. Im Beispiel oben sind das die Klassenvariable `z` und die Klassenmethoden `m3` und `m4`. Diese Bestandteile einer Klasse existieren nur einmal, sozusagen unter der Obhut der Klasse. Sie werden über den Namen der Klasse angesprochen.

```
C.z = 0;
C.m3(42);
```

Von völlig anderem Charakter sind die Objektvariablen und –Methoden `x`, `y`, `m1` und `m2`. Sie gehören jeweils zu einem bestimmten Objekt der Klasse `C` und werden immer nur über ein Objekt angesprochen:

```
C o = new C();
o.x = 0;
o.m2(42);
```

Es ist klar, dass diese Unterscheidung auch auf die Schnittstelle der Klasse übertragen werden kann und muss. Eine Klasse hat eine *statische Schnittstelle* und eine *Objekt–Schnittstelle*:

Die **statische Schnittstelle einer Klasse** umfasst die extern benutzbaren statischen Komponenten der Klasse.
Die **Objekt–Schnittstelle einer Klasse** umfasst die extern benutzbaren nicht–statischen Komponenten der Klasse.

Die Unterscheidung *statisch – nicht statisch* kann mit der Unterscheidung *paketintern – nicht paketintern* kombiniert werden. Beispiel:

```
public final class C {
  private int x;
        int y;               // Objekt-Schnittstelle, paketintern
  static int z;              // statischen Schnittstelle, paketintern

             void m1(int x) {...} // Objekt-Schnittstelle, paketintern
  public     void m2(int x) {...} // Objekt-Schnittstelle, oeffentlich
  public static void m3(int x) {...} // statischen Schnittstelle, oeffentlich
  private static void m4(int x) {...}
}
```

Daten–Komponenten, wie `y` und `z` im Beispiel, zu Bestandteilen der Schnittstelle zu machen, gilt allgemein als unprofessionell und nicht empfehlenswert. So wie die Klasse definiert ist, sind `y` und `z` Bestandteile der Schnittstelle.

Interface: öffentliche wohldefinierte Schnittstelle von Objekten

Für die Objekt–Schnittstelle einer Klasse gibt es in Java ein eigenständiges Sprachkonstrukt, das *Interface*. Statt *Interface* sagen wir auch oft einfach *Schnittstelle*, wenn es aus dem Kontext heraus klar ist, dass es sich um die *Objekt–Schnittstelle einer Klasse* handelt. Dabei gibt es allerdings zwei gravierende Einschränkungen:[1]

- Ein *Interface* enthält nur Methoden.
- Ein *Interface* enthält nur Öffentliches.

Um anständige Software–Entwürfe zu unterstützen, sind in einem *Interface* also keine Datenkomponenten erlaubt! Die Regelung, dass nur Öffentliches in ein *Interface* gehört, macht die Verwendung einfacher: Wir müssen nicht darauf achten, ob wir uns im gleichen Paket befinden oder nicht.

Das *Interface* zu unserer Klasse `C` von oben ist damit:

```
public interface CInterface {
   public void m2(final int x); // oeffentliche Objekt-Methode
}
```

Ein *Interface* umfasst also die Objektschnittstelle einer Klasse wobei, wie gesagt, die Erwähnung von Datenkomponenten und Komponenten mit paketlokaler Sichtbarkeit nicht erlaubt ist. Das Schlüsselwort `public` kann in einem *Interface* nach Belieben hingeschrieben oder weggelassen werden. Die Bestandteile eines *Interface* werden unabhängig davon immer als `public` betrachtet. Es ist guter Stil `public` in jedem *Interface* vor jede Methode entweder immer oder nie zu schreiben.

```
public interface I {
  void f(int x); // == public void f(int x);
  public void g(double d); // schlechter Stil: public immer oder nie !
}
```

Die Methoden in einem *Interface* haben keine Implementierung. Sie sind darum abstrakt. Man darf sie auch als `abstract` kennzeichnen:

```
public interface I {
  void f(final int x);           // OK
  public void g(final double d); // auch OK
  abstract public void m3(final char c); // ebenfalls OK
}
```

Ein solches Durcheinander an Attributen, ohne jede unterscheidende Aussagekraft, ist natürlich ganz besonders übel. Am besten lässt man alles Überflüssige und Selbstverständliche weg:[2]

[1] Genau genommen dürfen in einem Interface auch innere Klassen und Enum–Typen definiert werden. Diese Möglichkeit ist wenig sinnvoll, bietet keine erweiterten Möglichkeiten, verwischt den Charakter als Objekt–Schnittstelle und sollte darum nicht ausgenutzt werden.

[2] Es gibt in Java so wenige Gelegenheiten weniger hinzuschreiben, dass man sie begierig ergreifen sollte.

```
public interface I {
  void f(final int x);
  void g(final double d);
  void m3(final char c);
}
```

Eine Klasse implementiert ein Interface

Zwischen `C` und `CInterface` besteht eine Beziehung. `C` implementiert das `Interface CInterface`, d.h. `C` stellt eine Implementierung aller öffentlichen nicht–statischen Methoden von `CInterface` zur Verfügung. Im Quellcode kann das explizit ausgedrückt werden:

```
  public interface CInterface {
        void m2(final int x);
  }
```

```
  public final class C implements CInterface { // C implementiert CInterface
        private     int x;      // ... und bietet dazu noch mehr
                    int y;
              static int z;

                    void m1(final int x) { ... }

        @Override
        public void m2(final int x) { ... }


        public static void m3(final int x) { ... }
        private static void m4(final int x) { ... }
  }
```

`C` erklärt damit, dass es eine Implementierung für alle in `CInterface` aufgeführten Methoden zur Verfügung stellt. Man beachte, dass wir weder etwas aus `C` heraus genommen noch irgendeine Sichtbarkeit verändert haben. `y`, `z` und `m1` gehören zur Objekt–Schnittstelle von `C` und jedes Objekt vom Typ `C` bietet seinen Benutzern (je nach dem ob sie im gleichen Paket sind oder nicht) Zugriff auf`y`, `z` und `m1`.

Die Aussage: "`implements CInterface`" sagt dass *mindestens* das in `CInterface` Aufgeführte angeboten wird. Darüber hinaus können beliebige weitere Methoden und auch (psst! nicht darüber reden) Objekt–Variablen zur Objekt–Schnittstelle gehören. Über die statische Schnittstelle einer Klasse macht ein *Interface* prinzipiell keine Aussage: da kann die Klasse Beliebiges hinzufügen.

Mit `@Override` wird zum Ausdruck gebracht, dass es sich bei `m2` um die Implementierung einer Methode handelt, deren Existenz vom Interface vorgeschrieben ist.

Ein Interface kann von beliebig vielen Klassen implementiert werden. Beispielsweise wird das `Cinterface` oben auch von der Klasse `D` implementiert. Allerdings etwas minimalistischer als von `C`:

```
public class D implements CInterface {
     public void m2(int x) { ... }
}
```

> Ein **Interface** ist eine Sammlung von abstrakten öffentlichen nicht–statischen Methoden. Eine Klasse **implementiert** ein *Interface*, wenn sie eine Implementierung der entsprechenden Methoden enthält.

Interface als Typ

Wozu ist eine Schnittstelle gut? Die Schnittstelle ist das, auf das ein Benutzer Zugriff haben muss, wenn er eine Komponente benutzen will. Angenommen der Benutzer ist ein Milchbauer. Er will Milch produzieren und verkaufen. Dazu benötigt er irgendwelche Tiere, die diese Milch produzieren können. Er stellt sie in seinen Stall und melkt sie regelmäßig. Das einzige was für diese Arbeit relevant ist, ist die Tatsache, dass die Tiere gemolken werden können und dabei Milch produzieren. Die *Anforderung* des Milchbauern an die Bewohner seines Stalls lässt sich bestens in Form eines *Interface*s formulieren:

```
public interface MilchTier {
     Milch melken();
}
```

Ein Milchbauer, der mit beliebigen Milchtieren arbeiten kann, sieht dann etwa so aus:

```
public final class MilchBauer {
     private static final int StallGroesse = 5;
     private MilchTier[] stall = new MilchTier[StallGroesse];
     private int belegt = 0;

     // beliebiges Milchtier in den Stall stellen:
     //
     public void milchTierVerkauf(final MilchTier t ) {
           if ( belegt < StallGroesse )
                 stall[belegt] = t;
                 belegt++;
     }

     // Alle Milchtier im Stall melken
    //
     public Milch[] milchKauf() {
           Milch[] eimer = new Milch[belegt];
           int i = 0;
           for ( MilchTier t : stall ) {
                 if ( i == belegt ) break;
                 eimer[i] = t.melken();
                 i++;
           }
           return eimer;
     }
}
```

Die Klasse `MilchBauer` ist, wie man sagt, *gegen ein Interface* programmiert, nicht gegen eine bestimmte Klasse. Er enthält die minimalen Anforderungen, die an Milchtiere gestellt werden. Das erhöht seine Flexibilität, Wiederverwendbarkeit und Unabhängigkeit deutlich. `Milchbauer` wird mit beliebigen Milchtieren umgehen können, auch solchen, die es jetzt noch gar nicht gibt. Tatsächlich gibt es momentan noch gar kein Milchtier, unser Bauer ist aber schon fertig. Die Beziehungen zwischen der Klasse `MilchBauer` und allen benutzten Klassen von Milchtieren ist klar und eindeutig auf das *Interface* `MilchTier` reduziert. Alles was dieses *Interface* nicht betrifft kann in `MilchBauer` und allen Milchtier–Klassen unabhängig voneinander geändert werden. Ein UML–Diagramm bringt diese Entkopplung über ein *Interface* gut zum Ausdruck (siehe 3.1).

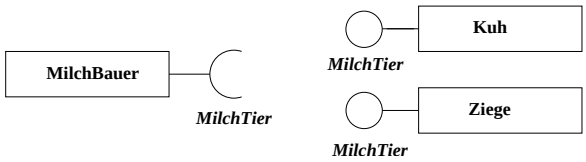


Abbildung 3.1: Schnittstellen in UML, Kurzform

`MilchTier` ist die Schnittstelle. `Ziege` und `Kuh` sind zwei Klassen, die diese Schnittstelle implementieren und `Milchbauer` ist eine Klasse, die Klassen mit dieser Schnittstelle benutzt. Für `Kuh` und `Ziege` ist `MilchTier` eine *exportierte* oder *Export–Schnittstelle*, für `MilchBauer` ist `MilchTier` eine *importierte* oder *Import–Schnittstelle*. Der Export, das Zur–Verfügung–Stellen muss explizit im Kopf einer Klassendefinition angegeben werden, beispielsweise muss die Kuh sagen, dass sie ein Milchtier ist:

```
public final class Kuh implements MilchTier {
     @Override
     public Milch melken() {
           return new Milch();
     }
}
```

Ein Bauer kann jetzt eine solche Kuh als MilchTier kaufen, melken und die Milch verkaufen:

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

```
    Kuh berta = new Kuh();
    MilchBauer josef = new MilchBauer();
    josef.milchTierVerkauf(berta); // main verkauft berta an josef
    Milch[] milchEimer = josef.milchKauf(); // main kauft Milch bei josef
}
```

Für eine Kuh reicht es nicht, die Methode `melken` zu implementieren, sie muss explizit mit

```
final class Kuh implements MilchTier
```

explizit sagen, dass sie die für Milchtiere geforderten Methoden zur Verfügung stellt.

Umgekehrt muss die Klasse `MilchBauer` nicht explizit erwähnen, dass sie intern die Schnittstelle `MilchTier` nutzt, es sei denn, dass `MilchTier` in einem anderen Paket definiert ist und daher wie jede andere Definition eines anderen Pakets explizit importiert werden muss.

In UML kann man Schnittstellen in Kurzform mit Kreis (Export) und Halbkreis (Import) darstellen. Schnittstellen haben aber auch noch eine ausführliche Darstellung (siehe 3.2).

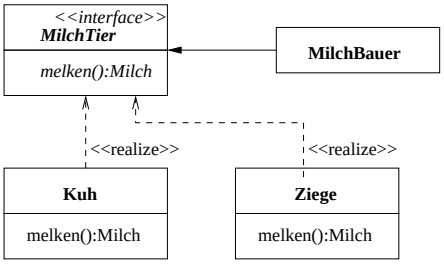


Abbildung 3.2: Schnittstellen in UML, ausführliche Form

Jetzt definieren wir noch Ziegen

```
public final class Ziege implements MilchTier {
    @Override
    public Milch melken() {
        return new Milch();
    }
}
```

und verkaufen eine an den Bauern:

```
public static void main(String[] args){
    Kuh berta = new Kuh();
    MilchBauer josef = new MilchBauer();
    josef.milchTierVerkauf(berta);
    josef.milchTierVerkauf(new Ziege());
    Milch[] milchEimer = josef.milchKauf();
}
```

Dafür erhalten wir beim Milchkauf auch zwei Eimer Milch.

3.1.2 Wichtige *Interfaces* der Java–API

Vergleiche: Comparable und Equals

Die Java–Bibliothek, üblicherweise Java–API genannt,[3] enthält nicht nur Klassen mit einer fertigen Funktionalität, sie enthält auch viele Interfaces. Damit wird die Organisation von Java–Code unterstützt. Beispielsweise gibt es ein vordefiniertes Interface *Comparable* das die Eigenschaft der Vergleichbarkeit ausdrückt. Jede Klasse, deren Objekte miteinander vergleichbar sind, d.h. miteinander auf größer, gleich oder kleiner verglichen werden können, sollten diese Schnittstelle implementieren und den Vergleich mit der in *Comparable* definierten Methode realisieren. Benutzer dieser Klasse brauchen dann nicht lange nachzudenken, ob und wie die Objekte verglichen werden können: sie implementieren *Comparable* oder eben nicht.

[3] Völlig korrekt wäre Java–API–Spezifikation oder genauer *Java 2 Platform Standard Edition 5.0 API Specification*

Die Benutzer sind dabei nicht nur andere Programmierer,[4] die den eigenen Code benutzen, es sind auch andere, richtige Arbeit leistende Bibliothekskomponenten, die darauf ausgerichtet sind, dass Java–Code nach den Java–Konventionen geschrieben wird. So gibt es etwa vorgefertigte Sortier–Routinen, die natürlich nur solche Objekte sortieren können, die vergleichbar sind – vergleichbar natürlich in der üblichen Art von Java als Implementierer der Schnittstelle *Comparable*.

Nehmen wir an, dass unsere Milchtiere vergleichbar sein sollen, etwa nach ihrer Milchleistung, ihrem Gewicht oder nach was auch immer, dann wird die Vergleichbarkeit in die Spezifikation von `MilchTier` aufgenommen:

```
public interface MilchTier extends Comparable<MilchTier> {
     Milch melken();
}
```

Mit

```
Milchtier extends Comparable<MilchTier>
```

wird gesagt, dass das Interface `MilchTier` eine Erweiterung (`extends`) der Schnittstelle

```
Comparable<MilchTier>
```

ist. das bedeutet, dass `MilchTier` die Methoden von `Comparable<MilchTier>` enthält plus das hier definierte `melken`. Um herauszufinden, was das Interface `Comparable<MilchTier>` fordert, sehen wir in die API nach und finden dort, dass `Comparable` eine einzige Methode verlangt:

```
java.lang
interface Comparable<T>
int compareTo(T o)
       // Compares this object with the specified object for order.
```

Ein etwas längerer Text erklärt dann wie genau eine Implementierung von `comparable` arbeiten soll. Kurz gefasst wird gesagt, dass eine Klasse, die `Comparable` implementiert, eine damit verträgliche Vergleichsmethode *equals* definieren sollte und dass

```
x.compareTo(y)
```

eine negativen int–Wert, Null oder einen positiven int–Wert liefern soll, je nach dem ob x kleiner, gleich oder größer y ist. Angenommen wir wollen unsere Milchtiere nach ihrer Milchleistung vergleichen. Die Milchleistung der Tiere sei dabei definiert als:

- Schwache Leistung: weniger als 1 Liter pro Tag
- Normale Leistung: 1 bis 5 Liter pro Tag
- Gute Leistung: Mehr als 5 Liter pro Tag

Das codieren wir einfach als Enum–Typ:

```
public enum MilchLeistung {
     SCHWACH, NORMAL, GUT
}
```

Der Typ MilchTier ist damit also jetzt:

```
final class Milch { }

public interface MilchTier extends Comparable<MilchTier> {
     Milch melken();
     MilchLeistung getMilchLeistung();
}
```

Die Realisation dieser Definition ist Aufgabe der Klassen `Kuh` und `Ziege`, also der Implementierungen des Interfaces `MilchTier`. In beiden Klassen fügen wir dazu eine entsprechende Objektvariable und eine Zugriffsmethode ein und können dann die Vergleichsoperationen definieren:

```
public final class Kuh implements MilchTier {
     private MilchLeistung milchLeistung = MilchLeistung.GUT;

     Kuh() {}
     Kuh( final MilchLeistung milchLeistung ) {
```

[4] In deren Achtung man auf das Niveau “Stümper” zurückfällt, wenn man sich nicht an diese Konvention hält.

```
            this.milchLeistung = milchLeistung; }

      @Override
      public Milch melken() {
            return new Milch();
      }

      @Override
      public int compareTo(final MilchTier m) {
            return milchLeistung.compareTo(m.getMilchLeistung());
      }

      @Override
      public boolean equals(final Object o) {
            if (o instanceof MilchTier)
                  return compareTo((MilchTier)o) == 0;
            else
                  return false;
      }

      @Override
      public MilchLeistung getMilchLeistung() {
            return milchLeistung;
      }
}
```

Die Klasse `Ziege` sieht entsprechend aus:

```
public final class Ziege implements MilchTier {
      private MilchLeistung milchLeistung = MilchLeistung.SCHWACH;

      Ziege(){}

      Ziege(final MilchLeistung milchLeistung ) {
            this.milchLeistung = milchLeistung; }

      @Override
      public MilchLeistung getMilchLeistung() {
            return milchLeistung;
      }

      @Override
      public Milch melken() {
            return new Milch();
      }

      @Override
      public int compareTo(final MilchTier m) {
            return milchLeistung.compareTo(m.getMilchLeistung());
      }

      @Override
      public boolean equals(final Object o) {
            if (o instanceof MilchTier)
                  return compareTo((MilchTier)o) == 0;
            else
                  return false;
      }
}
```

Zwei Milchtiere sind jetzt gleich, wenn sie die gleiche Milchleistung haben! Auch dann, wenn das eine eine Kuh und das andere eine Ziege ist. Wenn das nicht gewollt ist, kann `equals` umdefiniert werden. Normalerweise sollte es aber konsistent mit `compareTo` definiert sein. Wenn wir Kühe mit Ziegen vergleichen wollen, und dabei nur die Milchleistung zählt, dann sollte auch konsequenterweise eine Kuh gleich einer Ziege sein können.

Die Methode `equals` hat als Parametertyp `Object`. Auch das ist eine Konvention von Java, an die wir uns halten. Sie kommt daher, dass Java auf dem Standpunkt steht, dass alles mit allem auf Gleichheit geprüft werden kann, unabhängig vom Typ dessen, was da verglichen wird. Diese Konvention hat *nicht* die Konsequenz, dass bei diesem Vergleich *automatisch* etwas

Vernünftiges passiert. Das zu gewährleisten ist Aufgabe der Programmierer. In unserem Fall testen wir zuerst, ob es sich bei dem anderen auch um ein Milchtier handelt

```
if (o instanceof MilchTier) ...
```

Wenn nicht, dann sind die beiden nicht gleich, wenn doch prüfen wir, ob `compareTo` Null liefert. In der `compareTo`–Methode selbst werden die Enum–Werte verglichen. Zu unserer Bequemlichkeit sind Enum–Typen immer vergleichbar. Die "Größe" ergibt sich implizit aus der Reihenfolge in der die Werte aufgeschrieben werden. In unserm Beispiel gilt darum:

```
SCHWACH < NORMAL < GUT
```

Da in `compareTo` mit der Methode `getMilchLeistung` auf ein anderes Milchtier zugegriffen wird, muss dies im Interface vermerkt sein. Es reicht nicht, dass jedes Milchtier (Kuh und Ziege) die Methode `getMilchLeistung` hat, diese Tatsache muss auch explizit im Interface vermerkt sein.

Unsere Milchtiere sind jetzt vergleichbar und sie halten sich an die Java–Konventionen. Als Belohnung dafür können sie mit einer API–Funktion `Arrays.sort` sortiert werden:

```
Arrays.sort(stall, 0, belegt-1);
```

`Arrays` ist eine Hilfsklasse der API, die diverse Hilfsfunktionen für Arrays bereitstellt. Darunter sind etliche Sortierfunktionen, die von den Elementen eines Feldes verlangen, dass sie nach ihrer "natürlichen Ordnung" (engl.: *natural order*) sortiert werden können. Die natürliche Ordnung ist für Java, die, die sich aus `compareTO` und `equals` ergibt. In einem UML–Diagramm können wir den Code wie in 3.3 darstellen.

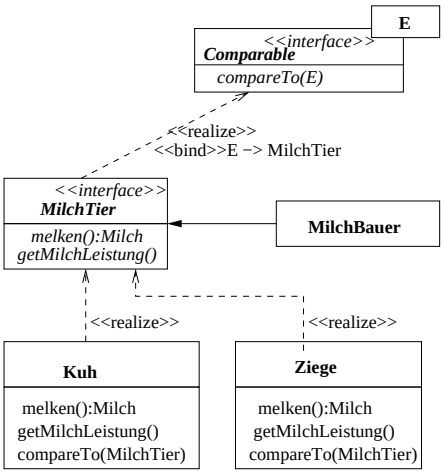


Abbildung 3.3: Vergleichbare Milchtiere in UML

Test auf Gleichheit, Größer–Kleiner–Vergleiche gehören zum Standardverhalten von Objekten. In der Philosophie von Java kann jedes Objekt mit jedem anderen auf Gleichheit getestet werden. Dazu unterstützt *jedes* Objekt die Methode *equals*. Da dies für alle gilt, gibt es auch kein Interface `Equals` und es ist auch nicht notwendig, dass irgendeine Klasse mit einem `implements Equals` oder Ähnlichem kund tut, dass ihre Objekte eine `equals`–Methode haben. Da ist also anders als bei Größenvergleichen mit `compareTo`.

Auch wenn jedes Objekt eine `equals`–Methode hat, so ist damit noch nicht gesagt, dass diese etwas Vernünftiges tut, oder gar das, was wir von ihr erwarten. Klassen der Java–API haben eine sinnvolle Implementierung der Gleichheit. Bei eigenen Klassen müssen wir überlegen, was Gleichheit für deren Exemplare bedeuten soll. Wie bereits weiter oben (siehe Seite 140) erläutert, hängt die Interpretation des Begriffs *Gleichheit* stark vom Charakter der Klasse ab. So müssen wir als erstes entscheiden, ob `equals` eine Realisation von das *"das Gleiche"*, oder *"das Selbe"* sein soll. Die vordefinierte Bedeutung von `equals` ist *"das Selbe"*. Wird es also nicht umdefiniert, dann liefert

```
x.equals(y)
```

genau dann `true`, wenn `x` und `y` sich auf exakt dasselbe[5] Objekt beziehen. Das passt dazu, dass Java implizit von einer Referenzsemantik ausgeht, also Objekte als unverwechselbare Individuen ansieht, die über eine eindeutige Referenz angesprochen

[5] Ja, man schreibt *dasselbe* in einem Wort! Damit ist es also nicht dasselbe wie mit *das Gleiche*. Das schreibt man in zwei Worten und das zweite groß. Raum für weitere Jahre der Rechtschreibreform!

werden. Will man etwas anderes für seine Objekte, dann muss man vom Standard abweichen und eine eigene Implementierung liefern.

Insgesamt haben wir bei den Vergleichen mit `equals`:

- Alle Klassen (solche der API und auch alle selbst definierten) unterstützen *equals*.
- Klassen der Java–API liefern mit *equals* eine sinnvolle und zu ihnen passende Interpretation von "Gleichheit". Welche das genau ist, liest man eventuell in der API–Dokumentation nach.
- Eigene Klassen, für die *Gleichheit* etwas *anderes* sein soll als die *Identität*, müssen

  ```
  public boolean equals(final Object o)
  ```

  implementieren.

In Vergleich (!) dazu haben wir bei Vergleichen mit `compareTo` zu beachten:

- Nur Klassen der Java–API, für die dies sinnvoll ist, unterstützen *compareTo*.
- Eine Klasse `T`, deren Exemplare verglichen werden können, sollte diesen Vergleich über *compareTo* anbieten, dazu das Interface `Comparable<T>` implementieren und eine Methode `int compareTo(T o)` enthalten. Das gilt für alle Klassen der Java–API und sollte für eigene Klassen selbstverständlich sein.
- Wird die Methode *compareTo* definiert, dann sollte ihr Verhalten konsistent zum Verhalten von *equals* sein.[6]

Kopien erzeugen: clone

Mit einer Zuweisung wie

```
x = y;
```

Werden die Bits vom Speicherplatz der Variablen `y` in den der Variablen `x` kopiert. Handelt es sich bei `x` und `y` um Variablen mit einem primitiven Typ, dann wird auf die Art der "gemeinte Wert" kopiert. Bei Variablen mit Klassentyp wird einfach die Referenz kopiert und nach der Zuweisung beziehen sich `x` und `y` auf das gleiche Objekt. Gelegentlich möchte man aber nicht dass, die zweite Referenz auf das gleiche Objekt zeigt, sondern, dass sie sich auf eine identische Kopie bezieht.

In Java gibt es zur Erzeugung identischer Kopien die `clone`–Methode. Im Gegensatz zu den Gegebenheiten im wirklichen Leben muss ein Java–Objekt nicht in ein medizinisches Labor gehen, um sich klonen zu lassen: Es kann sich selbst klonen. So wie jedes Objekt eine `equals`–Methode hat, hat es eine `clone`–Methode. Mit der Zuweisung

```
x = y.clone();
```

wird beispielsweise erreicht, dass `x` und `y` unterschiedliche Objekte zeigen, von denen das eine eine identische Kopie des anderen ist. D.h. anschließend `sollten` also die Vergleiche

```
x == y   // false
```

und

```
x.equals(y) // true
```

`false` bzw. `true` liefern. Die Betonung liegt wieder auf *sollten*. Wie bei *equals* haben nur die Klassen der API eine sinnvolle und konsistente Implementierung von `clone`. Eigene Klassen haben eine von der Klasse `Object` übernommene `clone`–Methode, mit vordefiniertem Verhalten. Soll also eine eigene Klasse `clone` unterstützen, dann sollte genau überlegt werden, was dieses "Klonen" bedeuten soll, ob es von `clone` in seiner vordefinierten Bedeutung unterstützt wird und wenn nicht dann muss die Klasse eine eigene Variante definieren. In der Regel wird sie dann die Eigenschaft haben, dass

```
x != x.clone();
```

gilt und, dass

```
x.equals(x.clone());
```

den Wert `true` hat. Ob das aber so ist, liegt in der Verantwortung des Entwicklers, der eventuell (hoffentlich mit einem guten Grund) von dieser Vorgabe abweichen kann.

In Bezug auf *Interfaces* und vordefinierte Methoden nimmt `clone` eine Mittelstellung zwischen `equals` und `compareTo` ein. Wir erinnern uns:

[6] In der API–Dokumentation von *Comparable* finden sich bei Bedarf weitere Informationen darüber wie *compareTo* und *equals* zu definieren sind und was genau kompatibel bedeutet.

- `equals`: Alle Objekte aller Klassen können mit *equals* verglichen werden. Es gibt kein Gleichheits–Interface, das man dazu implementieren muss und außerdem wird für alle Klassen eine vordefinierte `equals`–Methode bereitgestellt.
- `compareTo`: Es gibt ein `Comparable`–Interface, das explizit implementiert werden muss (...`implements Comparable<E>`...). Eine vordefinierte Implementierung für eigene Klassen gibt es nicht.

Das Klonen muss, wie *Comparable*, explizit implementiert werden und wie bei `equals` gibt es eine vordefinierte Variante. Im Gegensatz zu `equals` muss diese aber explizit in einer eigenen `clone`–Methode aufgerufen werden. Nehmen wir an Kühe – etwa solche mit besonders hoher Milchleistung – sollen geklont werden. Wir definieren dann beispielsweise:

```
public final class Kuh implements MilchTier, Cloneable {

     ... wie bisher ...

     public Object clone() throws CloneNotSupportedException {
           return super.clone(); // vordefiniertes clone aufrufen
     }
}
```

Um klonbar zu sein muss `Kuh` also erstens `Cloneable` implementieren und zweitens eine `clone`–Methode mit einem Methodenkopf wie im Beispiel definieren. In dieser Methode kann, und wird in aller Regel, mit

```
super.clone()
```

das vordefinierte `clone` von `Object` aufgerufen.[7] Eine Anwendung von `clone` ist:

```
public static void main(String[] args){
     Kuh berta = new Kuh();
     berta.createKaelbchen();

     Kuh elsa = null;
     try {
           elsa = (Kuh) berta.clone();
     } catch (CloneNotSupportedException e) {/* passiert nicht */}
     System.out.println( berta == elsa ); // false
     System.out.println( berta.equals(elsa) ); // true
}
```

Jede Aktivierung von `clone` muss gegen die `CloneNotSupportedException` abgesichert sein. Auch wenn, wie in diesem Fall, diese Ausnahme nicht auftreten kann.

Das vordefinierte `clone` ist immer dann ohne weiteres Nachdenken OK, wenn sich die Objekte der Klasse nicht ändern können. Können sie sich ändern, dann ist Vorsicht angebracht. Das vordefinierte `clone` kopiert alle Objektvariablen, wie man sagt, *flach*.

Eine flache Kopie schaufelt einfach die Bits der Variablen um. Enthält die Variable eine Referenz, dann enthält die Kopie die gleiche Referenz, die sich dann naturgemäß auf dasselbe Objekt bezieht. Nehmen wir an, dass zu einer Kuh ein Kälbchen gehört, das ebenfalls ein Exemplar der Klasse Kuh ist:

```
public final class Kuh implements MilchTier, Cloneable {
     ...
     private Kuh kaelbchen;
     ...

     public void createKaelbchen() {
           kaelbchen = new Kuh();
     }

     public Kuh getKaelbchen() {
           return kaelbchen;
     }

     public Object clone() throws CloneNotSupportedException {
           return super.clone();
     }

}
```

[7] Genau genommen wird unter Umständen `clone` aus der Basisklasse aufgerufen, das wiederum unter Umständen `clone` seiner Basisklasse aktiviert, und so weiter, bis wir schließlich bei `Object` angelangt sind. Derartige Feinheiten diskutieren wir später in einer fortgeschritteneren Veranstaltung.

Wird jetzt die `Kuh berta` mit einem Kälbchen ausgestattet und geklont, dann haben `berta` und der Klon dieselbe Kuh als Kälbchen. Wir sehen das mit einen kleinen Testprogramm

```
try {
     Kuh berta = new Kuh();
     berta.createKaelbchen();
     Kuh elsa = (Kuh) berta.clone();

     System.out.println(berta == elsa); // false
     System.out.println(berta.equals(elsa)); // true

     System.out.println(
           berta.getKaelbchen()
                 == elsa.getKaelbchen()); // true
     System.out.println(
           berta.getKaelbchen().equals(
                 elsa.getKaelbchen())); // true

} catch (CloneNotSupportedException e) { }
```

Die Kühe `berta` und `elsa` sind gleich aber nicht dieselben. Ihre Kälbchen sind dieselben und dann natürlich auch gleich.

Soll eine tiefe Kopie eines Objektes erzeugt werden, dann müssen auch die Sub–Komponenten eines Objekts kopiert werden. Auch das ist kein Problem. Wir klonen rekursiv nach unten bis zum letzen tiefsten Bit:

```
public final class Kuh implements MilchTier, Cloneable {

     private MilchLeistung milchLeistung = MilchLeistung.GUT;
     private Kuh      kaelbchen;

     ...

     public Object clone() throws CloneNotSupportedException {
           Kuh klon = (Kuh) super.clone();
           if ( kaelbchen != null)
                 klon.kaelbchen = (Kuh) kaelbchen.clone();
           klon.milchLeistung = milchLeistung; // ueberfluessig
           return klon;
     }
     ...
}
```

Die Zuweisung der `milchLeistung` sorgt dafür, dass die Milchleistung geklont wird. Das ist nicht falsch aber überflüssig. *Enum*–Objekte können nicht verändert werden. Unveränderliches dürfen sich beliebig viele Objekte problemlos, d.h. ohne beobachtbare Effekte, teilen.

Text erzeugen: toString

Die Methode `toString` transformiert beliebige Objekte in Textform. Sie ist für jedes Objekt definiert. Für Klassen der API ist die Implementierung sinnvoll. Für eigene Klassen gibt es eine vordefinierte Implementierung, die aber in der Regel nicht zufriedenstellend ist. Ein Interface muss dazu nicht im Kopf der Klassendefinition erwähnt werden. In dieser Hinsicht entspricht `toString` der Methode `equals`.

Zusammengefasst:

> **equals** und **toString**: Für alle Klassen vordefiniert, in eigenen Klassen ist diese Definition sinnvoll zu überschreiben mit:
> `public boolean equals( Object o ) { .... }`
> `public String toString() { .... }`

compareTo: Nicht vordefiniert. Eigene Klassen C müssen `Comparable` implementieren und `compareTo` definieren:

```
class C implements Comparable<C>{
        ...
        public int compareTo( C c ) { .... } ...
        ...
}
```

clone: Nur eine Hilfsmethode ist vordefiniert. Eigene Klassen C müssen `Cloneable` implementieren und `clone` mit Hilfe der Hilfsmethode `super.clone` definieren:

```
class C implements Cloneable{
        ...
        public Object clone () throws CloneNotSupportedException {
             super.clone(); ...
        }
        ...
}
```

3.1.3 Beispiel Rationale Zahlen

Brüche und rationale Zahlen

Abstrakte Datentypen sind nicht nur etwas für Informatiker und es gibt sie auch nicht erst seit es Computer gibt. Wir, und viele Generationen vor uns, haben mit den *Rationalen Zahlen* bereits in der Schule einen abstrakten Datentyp kennen gelernt. Zuerst begegnen sie uns als Brüche. Ein *Bruch* ist ein Paar von ganzen Zahlen. Man schreibt die eine über die andere, trennt sie durch einen Strich, den Bruchstrich. Die eine Zahl, die obere, nennt man *Zähler*, die untere *Nenner* und das Ganze dann *Bruch*.

Unsere Lehrer versuchen uns dann später zu erklären, dass ein solcher Bruch ein *Teilungsverhältnis* darstellt und, dass ein Bruch nicht mit einem Zahlenpaar zu identifizieren ist. Wir erkennen, dass zwei ungleiche Zahlenpaare, trotz aller Ungleichheit, dasselbe sein können. So ist beispielsweise

$$\frac{1}{3} = \frac{3}{9}$$

Die Regeln, mit denen Addition, Subtraktion, Multiplikation und Division auf solchen Teilungsverhältnissen ausgeführt werden, übten wir dann als Algorithmen ein und nennen sie seit dem *Bruchrechnen*.

Später lernten wir, dass Brüche eine Darstellung von *rationalen Zahlen* sind. Als Informatiker erkennen wir die *rationalen Zahlen* als ADT und die *Brüche* als eine mögliche Implementierung dieses ADTs. Eine andere sind die Dezimalzahlen. Ein Paar von ganzen Zahlen, mit einem waagrechten Strich zwischen ihnen, ist ein Exemplar der Implementierung (ein Repräsentant) und die *Abstraktionsfunktion* **A** bildet sie auf die "gemeinte" rationale Zahl (das Repräsentierte) ab. So *meinen* (repräsentieren) beispielsweise $\frac{1}{3}$ und $\frac{3}{9}$ dasselbe:

$$\mathbf{A}(\frac{1}{3}) = \mathbf{A}(\frac{3}{9})$$

Rationale Zahlen sind *abstrakt* relativ zu konkreten Datentypen wie *natürlichen* oder *ganzen Zahlen*, weil es kein ursprüngliches[8] Verständnis für Teilungsverhältnisse gibt. Wir implementieren – in unserem Hirn (!) – diesen ADT durch das Lernen der Regeln der Bruchrechnung. Die Abstraktionsfunktion sagt uns, was wir dabei eigentlich meinen und treiben.

Rationale Zahlen

Rationale Zahlen sind ein wohlbekanntes mathematisches Konzept und in einer Spezifikation dürfen wir uns auf wohlbekannte mathematische Konzepte beziehen. Die Spezifikation rationaler Zahlen ist darum ganz einfach:

> Der **ADT Rational** besteht aus den rationalen Zahlen mit den Operationen Addition, Subtraktion, Multiplikation und Division, so wie sie auf rationalen Zahlen definiert sind.

[8] angeborenes oder in den ersten Grundschuljahren gelerntes ?

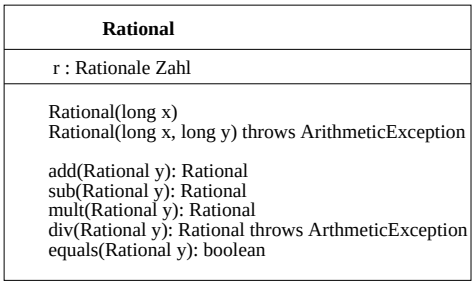


Abbildung 3.4: Klassendiagramm zur Spezifikation des ADT Rational

Zur Auffrischung nehmen wir ein Mathematikbuch zur Hand und rekapitulieren kurz die Definition, die wir dort finden:

Eine *rationale Zahl* ist eine Äquivalenzklasse von Paaren ganzer Zahlen mit

$(a,b) \sim (c,d) \Leftrightarrow a*d = b*c$

Mit $\frac{x}{y}$ bezeichnen wir die Äquivalenzklasse der Zahlenpaare, die zum Paar (x,y) äquivalent sind. Das Paar (x,y) ist ein Repräsentant der Äquivalenzklasse $\frac{x}{y}$.

Die *Operationen auf rationalen Zahlen* werden als Operationen auf deren Repräsentanten definiert:

$\frac{a}{b} * \frac{c}{d} := \frac{a*c}{b*d}$

$\frac{a}{b} + \frac{c}{d} := \frac{a*d+c*b}{b*d}$

Die hochgeschätzten Mathematiker sagen also nichts weiter, als dass wir guten Gewissens Bruchrechnung betreiben dürfen und dass Brüche Äquivalenzklassen sind, die durch ein beliebiges Exemplar vertreten werden können.

Rationale Zahlen: Spezifikation

Das Konzept der rationalen Zahlen ist mathematisch klar und allgemein bekannt. In einer Spezifikation können wir uns direkt darauf beziehen.

```
context Rational
  init: r = 0
  inv: r ist konstant

context Rational.Rational(int x)
  pre  : -
  post : r = x/1

context Rational.Rational(int x, int y)
  pre  : -
  post : r = x/y
  throws: ArithmeticException if y=0

context Rational.add(Rational y)
  pre  : -
  post : return r1,
         r1.r = r+y.r

context Rational.sub(Rational y)
  pre  : -
  post : return r1,
         r1.r = r-y.r

context Rational.mult(Rational y)
  pre  : -
  post : return r1,
         r1.r = r*y.r

context Rational.div(Rational y)
  pre  : -
  post : return r1,
```

```
        r1.r = r/y.r +/-genauigkeit
  throws: ArithmeticException if y=0
```

Wir sehen, in der Spezifikation wird nichts anderes gesagt, als dass unser ADT Rational die Bruchrechnung beherrschen soll. `r` ist eine rationale Zahl, also eine Äquivalenzklasse von Paaren ganzer Zahlen. Der Wert von `r` soll unveränderlich sein. `Rational` ist darum ein wertorientierter ADT. Die arithmetischen Operation in der Spezifikation (`+, -, *, /`) beziehen sich auf rationale Zahlen. Sie sind "mathematisch gemeint" und damit unendlich genau. In einer Implementierung können wir diese Genauigkeit natürlich nicht erreichen. Wir können lediglich versuchen eine beliebige Genauigkeit zu erreichen. Dabei wird es dann unter Umständen zu Speicherproblemen kommen. Dies wird hier bewusst einkalkuliert. Der Sinn unseres Datentyps `Rational` liegt genau darin, gebrochene Zahlen beliebiger Genauigkeit anzubieten.

Spezifikation und Interface

Für die Spezifikation kann spontan ein *Interface* angegeben werden:

```
public interface Rational {
     Rational add (Rational y);
     Rational sub (Rational y);
     Rational mult(Rational y);
     Rational div (Rational y) throws ArithmeticException;
}
```

Doch welchen Sinn hat es, ein solches Interface zu definieren?

Die Definition eines Interfaces ist nur dann sinnvoll, wenn es mehr als eine Klasse gibt oder geben könnte, die das Interface implementiert. Nun könnte man sich durchaus vorstellen, dass es zu `Rational` zwei Implementierungen gibt. Beispielsweise eine, bei der die rationale Zahl `r` intern als Dezimalbruch und eine, bei der sie als natürlicher Bruch dargestellt wird.

```
public final class Dezimalbruch implements Rational {...}

public final class Bruch implements Rational {...}
```

Zwei Implementierungen bedeuten einen erheblichen Aufwand, auch weil die Parameter der Operationen von `Rational` den Typ `Rational` haben. Mit der Konsequenz, dass die beiden Implementierungen beliebige Vermischungen unterstützen müssen. Z.B.:

```
Rational r1 = new DezimalBruch(1,-4);
Rational r2 = new Bruch(12,17);
Rational r3 = r1.add(r2);
```

Der Einfachheit halber beschränken wir uns auf eine Implementierung der rationalen Zahlen als Brüche. Die Definition einer Schnittstelle ist dann nicht mehr notwendig.

Welche Schnittstelle sollen nun umgekehrt die rationalen Zahlen – jetzt in der Implementierung als Brüche – selbst erfüllen? Rationale Zahlen sollten vergleichbar sein, also eine sinnvolle Implementierung von `equals` liefern und `compareTo` bieten. Selbstverständlich erwarten wir auch, dass sie mit `toString` in Textform gebracht werden können. Die Methode `clone` muss nicht angeboten werden. Rationale Zahlen sind unveränderlich. Es ist darum nicht nötig, Kopien des gleichen Werts zu unterscheiden. Zur Illustration sollen rationale Zahlen trotzdem klonbar sein. Insgesamt kommen wir damit zu folgendem Interface `Rational`:

```
public interface Rational extends Comparable<Rational>, Cloneable {
     Rational add (Rational y);
     Rational sub (Rational y);
     Rational mult(Rational y);
     Rational div (Rational y) throws ArithmeticException;
}

public final class Bruch implements Rational {
     ...
}
```

Oder, bei Beschränkung auf eine Implementierungsvariante und Verzicht auf das Interface:

```
public final class Rational implements Comparable<Rational>, Cloneable {
     ...
}
```

Implementierung: Repräsentant von r

Da Java über keine direkte Implementierung von Äquivalenzklassen ganzer Zahlen verfügt, können wir die Spezifikation nicht ohne weiteres übertragen. Für

```
r: rationale Zahl
```

muss ein Repräsentant in Form von Java–Typen gefunden werden. Wie bereits erwähnt wollen wir rationale Zahlen als Brüche beliebiger Genauigkeit darstellen. Der Einfachheit halber und um Speicherprobleme so weit wie möglich zu reduzieren, werden nicht wie in der mathematischen Definition beliebige Paare ganzer Zahlen als Repräsentanten zugelassen. Wir arbeiten stattdessen mit einer *normierten Bruch–Darstellung*:

- Vorzeichen `vz` plus oder minus.
- Zähler `z` Eine ganze Zahl größer oder gleich Null.
- Nenner `n` Eine ganze Zahl größer Null.
- Zähler und Nenner maximal gekürzt.

Maximal gekürzt bedeutet, dass der GGT von Zähler und Nenner stets Eins sein soll. Damit können wir deren Größe eventuell einigermaßen unter Kontrolle halten. Die Darstellung mit einem expliziten Vorzeichen und positivem Zähler und Nenner vereinfacht die Operationen der Bruchrechnung. Die Abstraktionsfunktion bildet die Darstellung auf eine rationale Zahl ab:

$\mathbf{A}(\text{plus, z, n}) = z/n$
$\mathbf{A}(\text{minus, z, n}) = -z/n$ mit $z =$ Wert von z, $n =$ Wert von n

Um die gewünschte beliebige Genauigkeit zu erreichen, speichern wir Zähler und Nenner als `java.math.BigInteger`.

Implementierung des ADT Rational

Die Implementierung von `Rational` ist jetzt nur noch eine Fleißarbeit in Bruchrechnen:

```
import java.math.BigInteger;

public final class Rational implements Comparable<Rational>, Cloneable {

    public Rational( ){
          zaehler = BigInteger.valueOf(0);
          nenner = BigInteger.valueOf(1);
    }

    public Rational(final long z) {
          vz          = z >= 0 ? Vorzeichen.plus : Vorzeichen.minus;
          zaehler = z >=0 ? BigInteger.valueOf(z) : BigInteger.valueOf(-z);
          nenner = BigInteger.valueOf(1);
          kuerze();
    }

    public Rational(final long z, final long n) {
          if ( n == 0 ) throw new ArithmeticException("divide by zero");
          vz = (z >= 0 && n > 0) || (z <= 0 && n < 0)
                            ? Vorzeichen.plus
                            : Vorzeichen.minus;
          zaehler = z >=0 ? BigInteger.valueOf(z) : BigInteger.valueOf(-z);
          nenner = n >=0 ? BigInteger.valueOf(n) : BigInteger.valueOf(-n);
          kuerze();
    }

    public boolean equals(final Object o) {
          if ( ! (o instanceof Rational) ) return false;
          return vz          == ((Rational)o).vz
                && zaehler  == ((Rational)o).zaehler
                && nenner   == ((Rational)o).nenner;
    }

    public Rational add (final Rational y) {
          Rational res = new Rational();
          Rational yy = null;
```

```
        Rational xx = null;
        try {
              yy = (Rational) y.clone();
              xx = (Rational) this.clone();
        } catch (CloneNotSupportedException e) { e.printStackTrace(); }

        xx.gleichnamig(yy);
        res.nenner = xx.nenner;
        if (xx.vz == yy.vz) { // x und y haben gleiche Vorzeichen
              res.vz = xx.vz;
              res.zaehler = xx.zaehler.add(yy.zaehler);
        } else {
              // x und y haben unterschiedliche Vorzeichen
              if ( xx.zaehler.compareTo(yy.zaehler) > 0 ) {
                    res.vz = xx.vz;
                    res.zaehler = (xx.zaehler).subtract(yy.zaehler);
              } else {
                    res.vz = yy.vz;
                    res.zaehler = (yy.zaehler).subtract(xx.zaehler);
              }
        }
        res.kuerze();
        return res;

  }

  public Rational sub (final Rational y) {
     Rational yy = null;
        try {
              yy = (Rational) y.clone();
        } catch (CloneNotSupportedException e) { e.printStackTrace(); }
     if ( yy.vz == Vorzeichen.minus )
        yy.vz = Vorzeichen.plus;
     else
        yy.vz = Vorzeichen.minus;
    return add(yy);

  }

  public Rational mult(final Rational y) {
     Rational res = new Rational();
     if ( vz == y.vz ) res.vz = Vorzeichen.plus;
     else
        res.vz = Vorzeichen.minus;
     res.zaehler = zaehler.multiply(y.zaehler);
     res.nenner = nenner.multiply(y.nenner);
     res.kuerze();
     return res;
  }

  public Rational div (final Rational y) {
     Rational kehrwert = new Rational();
     kehrwert.vz = y.vz;
     kehrwert.zaehler = y.nenner;
     kehrwert.nenner = y.zaehler;
     return mult(kehrwert);
  }

  public int compareTo(final Rational r) {
        try {
              Rational yy = (Rational) r.clone();
              Rational xx = (Rational) this.clone();
              xx.gleichnamig(yy);
              return xx.zaehler.compareTo(yy.zaehler);
        } catch (CloneNotSupportedException e) { }
        return 0;
  }
```

```
    public String toString() {
        return ( vz.equals(Vorzeichen.plus) ? "+" : "-" )
                    + zaehler.toString()
                    + "/"
                    + nenner.toString();
    }

    /*************************************************************/

    private enum Vorzeichen {
        plus, minus
    };

    /*
     * @inv: zaehler und nenner positiv,
     *     und maximal gekuerzt
     */
    private Vorzeichen vz = Vorzeichen.plus;
    private BigInteger zaehler;
    private BigInteger nenner;

    private void erweitere(final BigInteger x) {
        if ( x.signum() == -1 ) { // < 0
            x = x.negate();
            if ( vz == Vorzeichen.plus)
                vz = Vorzeichen.minus;
            else
                vz = Vorzeichen.plus;
        }
        zaehler = zaehler.multiply(x);
        nenner = nenner.multiply(x);
    }

    private void kuerze() {
        if ( !(zaehler.signum() == 0) ) {
            BigInteger ggt = zaehler.gcd(nenner);
            zaehler = zaehler.divide(ggt);
            nenner = nenner.divide(ggt);
        }
    }

    /*
     * macht this und r gleichnamig
     */
    private void gleichnamig(final Rational r) {
        BigInteger n = kgv (nenner, r.nenner);
        erweitere( n.divide(nenner) );
        r.erweitere( n.divide(r.nenner) );
        nenner = n;
        r.nenner = n;
    }


    private static BigInteger kgv (final BigInteger x, final BigInteger y) {
        return ( x.multiply(y) ).divide( x.gcd(y) );
    }
}
```

Als Anwendung wollen wir die Wurzel nach dem Verfahren von Heron ziehen:

```
public static Rational heron(Rational x) {
    Rational a     = new Rational(1);
    Rational a_alt = new Rational(0);
    Rational zwei = new Rational(2);
    Rational eps = new Rational(1, 100000000000L);

    while ( (fabs(a.sub(a_alt))).compareTo(eps) > 0 ) {
        a_alt = a;
        a = (a.add( x.div(a) ).div(zwei)); // a = (a + x/a)/2;
```

```
    }
    return a;
}

public static Rational fabs(final Rational x) {
    if ( x.compareTo(new Rational(0)) < 0 )
          return (new Rational(0)).sub(x);
    else return x;
}

public static void main(String[] args) {
    Rational x = new Rational(9);
    Rational r = heron(x);
    System.out.println( "Wurzel("+x+")= "+ r);
}
```

Das Ergebnis

```
Wurzel(+9/1)=
+340282366920938463463374607431768211457 / 113427455640312821154458202477256070485
```

stellt eine recht gute Näherung an 3 dar.

3.2 Generische Klassen, Schnittstellen und Methoden

3.2.1 Generische Klassen und Schnittstellen

Generische Definitionen

Eine generische Definition ist eine Definition, bei der Typinformationen offen bleiben. Sie kann sich damit an unterschiedliche Gegebenheiten anpassen. Die Typen, in denen die Definitionen offen sind, werden als Parameter angegeben. Etwa so wie eine Funktion eine Berechnung ist, die in einigen Werten offen ist und diese als Parameter annimmt. Generische Definitionen gibt es in Java in drei Varianten:

> Eine **generische Klasse**, ein **generisches Interface**, oder eine **generische Methode**, ist eine Klasse, ein Interface, eine Methode deren Definition in einem oder mehreren Typen parametrisiert ist.

Wir geben hier nur eine erste Einführung in die Thematik. Eine tiefergehende Behandlung erfordert ein Verständnis der Mechanismen der Vererbung, die in diesem Kurs nicht behandelt werden. Wir beschränken uns darum hier auf eine erste informale Einführung, die an manchen Stellen etwas vereinfachend ist. Für die Nutzung der Java–API und die Definition einfacher eigener generischer Konstrukte sollte das jedoch ausreichend sein.

Generische Klassen

In Java können Klassen, Schnittstellen und Methoden definiert werden, die in einem oder mehreren Typen "offen" sind. Am einfachsten macht man sich das an einem Beispiel klar. Die Klasse `Pair` im folgenden Beispiel legt fest, dass ihre Objekte aus Paaren von Werten bestehen, sie lässt aber offen, welchen Typ diese Werte haben. Sie haben *irgendeinen* Typ `T1` bzw. `T2`.

```
public final class Pair<T1, T2> {
     private T1 v1;
     private T2 v2;

     Pair( T1 v1, T2 v2 ) { this.v1 = v1; this.v2 = v2; }
     Pair()               { this.v1 = null; this.v2 = null; }

     public T1 getV1() { return v1; }
     public void setV1(T1 v1){ this.v1 = v1; }

     public T2 getV2() { return v2; }
     public void setV2(T2 v2){ this.v2 = v2; }
}
```

Im Kopf der Klassendefinition werden die *generischen (Typ–) Parameter* `T1` und `T2` in spitzen Klammern angegeben. In der folgenden Klassendefinition können `T1` und `T2` *im wesentlichen* so verwendet werden, als seien sie normale Klassentypen. – Zu den Einschränkungen kommen wir gleich noch.

Um aus `Pair` einen echten Typ zu machen, müssen die Typ–Parameter `T1` und `T2` durch echte Typen ersetzt werden:

```
// Ein Integer-String-Paar erzeugen:
Pair<Integer, String> p1 = new Pair<Integer, String>(1, "Hallo");

Integer x = p1.getV1(); // ... und verwenden:
p1.setV2("Blubber");
```

Hierbei ist:

```
Pair<Integer, String>
```

der (echte) Typ der Paare, bei denen `T1` an `Integer` und `T2` an `String` gebunden ist. `p1` ist die Variable und

```
Pair<Integer, String>(1, "Hallo")
```

ist ein Aufruf des Konstruktors der Klasse `Pair<Integer, String>`. Selbstverständlich können `T1` und `T2` auch an den gleichen Typ gebunden werden:

```
// Ein Paar von Double-Werten:
Pair<Double, Double> p2 = new Pair<Double, Double>(0.1, 0.5);
```

Bei der Objekterzeugung dürfen also generische Konstrukte – kurz *Generics* verwendet werden. Der Ausdruck hinter `new` in

```
new Pair<Double, Double>(0.1, 0.5); // OK!
```

enthält den generischen Typ `Pair` und zweimal das gleiche generische Argument `Double`.

So wie bei Methoden ist es wichtig generische *Argumente* (*aktuelle* generische Parameter) von *Parametern* (*formalen* generische Parametern) zu unterscheiden. Ein `new` mit generischen Argumenten wie oben ist erlaubt. Ein `new` mit (formalen) generischen Parametern ist ebenfalls möglich:

```
class G<T> {
  ...
  Pair<T, T> p = new Pair<T, T>(); // auch OK!
  ...
}
```

Diamont–Operator in Java 7

In Java 7 ist die Notation bei generischen Klasse etwas vereinfacht worden. Bei der Erzeugung einer Instanz kann die Wiederholung des Typparameters entfallen. Der Conpiler ist in der Lage den sogenannten *Diamont–Operator* – spitze Klammer auf, spitze Klammer zu – mit den richtigen Werten zu füllen. Ein Beispiel ist:

```
class C<T> {
  T x;
}

public class WithDiamont {

  public static void main(String[] args) {
    C<String> c = new C<>(); // String muss nicht wiederholt werden.
  }

}
```

Restriktionen bei der Verwendung generischer Typen

Typ–Parameter, wie `T1` und `T2` im Beispiel oben, können nur Klassentypen repräsentieren. Es ist darum nicht erlaubt primitive Typen als aktuelle Parameter zu verwenden:

```
Pair<int, double> p1; // VERBOTEN: primitive Typen als Typ-Parameter
```

Innerhalb einer generischen Definition können Typ–Parameter auch nur mit gewissen Einschränkungen verwendet werden: So können Objekte, deren Typ durch einen Typ–Parameter angegeben wird, nur so behandelt werden als seien sie vom Typ `Object`. Beispiel:

```
 final class C<T> {
   private T x;
   public void m(T y) {
      ... x.toString() .... // OK
      ... y.equals(x) .... // OK
      ... x < y    .... // NICHT OK
      ... x.m(...) .... // NICHT OK, wenn
   }                    // m keine Methode von Object ist
 }
```

Neue Objekte, deren Typ ein generischer Typ (d.h.. Typ–Parameter) ist, können nicht angelegt werden:

```
 final class C<T> {
   private T x = new T(); // VERBOTEN <<--
   private T y;         // OK
   private Pair<T,T> p; // OK

   public C() {
      x = null;         // OK
      p = new Pair<T,T>(); // OK
      y = new T();      // VERBOTEN <<--
```

```
    }
    public C(T x) {
       this.x = x;      // OK
    }

    ...
}
```

Generics und Felder

Felder und Generics vertragen sich nicht besonders gut. Darum gibt es gelegentlich überraschende Probleme. So können beispielsweise Felder mit einem generischen Typ zwar definiert aber nicht instanziiert werden:

```
final class C<T> {
   T[] a;              // OK
   T[] a = new T[10];  // VERBOTEN <<--
}
```

Felder mit Elementtyp `Object` können erzeugt und dann in Felder mit einem generischen Typ konvertiert werden:

```
final class C<T> {
   T[] a;                  // OK
   void m(T y) {
       a = (T[]) new Object[10]; // OK mit Warnung
       a[0] = y;           // OK
   }
}
```

Bei der Erzeugung von Feldern nimmt man immer den passenden "rohen" Typ um Instanzen des Feldes zu erzeugen:

```
T[] a = (T[]) new Object[10]; // T -> Object
Pair<T,T>[] pa = (Pair<T,T>[]) new Pair[10]; // Pair<T,T> -> Pair
```

Solange das Feld mit generischem Typ innerhalb der Klasse verwendet wird, ist dies unproblematisch. Man kann also beliebige Containertypen definieren und in ihnen Felder verwenden:

```
final class MyContainer<T> {
    ....
    private T[] elements = (T[]) new Object[...];
    ....
    public void put(T x) {.....}
    public T get() {.....}
    ....
}
```

Probleme gibt es, wenn ein intern erzeugtes Feld aus der generischen Klasse heraustransportiert werden soll. Beispielsweise wenn der Containerinhalt als Feld herausgegeben werden soll:

```
final class MyContainer<T> {
    ....
    private T[] elements = (T[]) new Object[...];
    ....
    public void put(T x) {.....} // OK
    public T get() {.....} // OK
    ....
    public T[] asArry() {
        return elements;   // Problem
    }
}

   public static void main(...) {
     MyContainer<String> c = new MyContainer<String>(10);
     ...
     String[] a = c.asArray();  // ClassCastException !!
   }
}
```

Dieses Programm ist übersetzbar führt aber zur Laufzeit zu einer *ClassCastException*

```
java.lang.ClassCastException:
[Ljava.lang.Object; cannot be cast to [Ljava.lang.String;
```

Trotz des Casts bleibt das intern erzeugte Feld ein Feld vom Typ `Object[]`.

Ein Cast führt keine wirkliche Modifikation durch. Mit ihm wird lediglich der Compiler über eine Meinung des Programmierers informiert. Bei der Zuweisung werden die Laufzeittypen geprüft und die passen nun mal nicht.

Das Phänomen ist nicht auf Felder beschränkt. Auch mit folgendem Beispiel handeln wir uns eine *ClassCastException* ein:

```
final class GenC {

    static class C<T> {
          T a;

          @SuppressWarnings("unchecked")
          C() {
                a = (T) new Object();
          }

          T getA() { return a; }
    }

    public static void main(String[] args) {
          C<String> c = new C<String>();
          String  s  = c.getA(); // ClassCastException !
    }
}
```

Objekte per Reflection erzeugen

Objekt mit generischem Typ können nur per *Reflection* erzeugt werden.[9] Wird das Objekt mit `Class.newInstance()` erzeugt, dann hat es den richtigen Typ:

```
final class GenC {

    static class C<T> {
          T a;

          @SuppressWarnings("unchecked")
          C(T dummy) throws InstantiationException, IllegalAccessException {

                //STATT a = (T) new Object():

                a = (T) dummy.getClass().newInstance();
          }

          T getA() { return a; }
    }

    public static void main(String[] args) throws InstantiationException,
                                        IllegalAccessException {
          C<String>          c = new C<String>(new String());
          String             s = c.getA();
    }

}
```

Leider wird dazu explizit zur Laufzeit der Typ des generischen Parameters benötigt. Im Beispiel oben übergeben wir dazu ein Dummy–Objekt. Alternativ kann man auch die Klasse übergeben:

```
final class GenC {

    static class C<T> {
```

[9] Das Thema *Reflection* geht über den Stoff dieser Einführung hinaus. Wir beschränken uns darum auf einige Hinweise. Der Leser konsultiere bei Bedarf die API–Dokumentation oder das Tutorial http://download.oracle.com/javase/tutorial/reflect/index.html.

```
            T a;

            @SuppressWarnings("unchecked")
            C(Class c) throws InstantiationException, IllegalAccessException {
                  a = (T) c.newInstance();
            }

            T getA() { return a; }
      }

      public static void main(String[] args) throws InstantiationException,
                                                     IllegalAccessException {
            C<String>         c = new C<String>(String.class);
            String            s = c.getA();
      }
}
```

Interessanter und praktisch relevanter wird die Sache, wenn wir es mit Feldern zu tun haben. Konstruieren wir als Beispiel einen einfachen Containertyp *Tuple*, der seine Elemente als Array ausliefern kann. Die Klasse `Array` bietet dazu die Methode `newInstance`, die es erlaubt ein Feld mit vorgegebenem Typ zu erzeugen.

```
import java.lang.reflect.Array;

public class Tuple<T> {
      private T[] a;

      @SuppressWarnings("unchecked")
      Tuple(T x, T y) {
            a = (T[]) new Object[2];
            a[0] = x;
            a[1] = y;
      }

      @SuppressWarnings("unchecked")
      private T[] getA() {

            // Feld mit korrektem Typ erzeugen:
            T [] result = (T[]) Array.newInstance(a[0].getClass(), 2);

            result[0] = a[0];
            result[1] = a[1];
            return result;
      }

      public static void main(String[] args) throws InstantiationException,
                                                     IllegalAccessException {
            Tuple<String> t = new Tuple<String>("Hallo", "Welt");
            String[]    s = t.getA();
            System.out.println("Tuple: "+ s[0] + ", " + s[1]);
      }
}
```

Generische Schnittstellen

Ein *Interface* kann genau wie eine Klasse in einem Typ parametrisiert sein:

```
interface PairInteface<T1, T2> {
      T1 getV1();
      void setV1(T1 v1);
      T2 getV2();
      void setV2(T2 v2);
}
```

Ein generisches Interface kann von einer generischen Klasse implementiert werden:

```
public final class PairImpl<T1, T2> implements PairInterface<T1, T2>{
      ...
}
```

Ebenso gut kann aber auch eine nicht–generische Klasse ein generisches *Interface* implementieren:

```
public class IntStringPair implements PairInterface<Integer, String>{
     private Integer v1;
     private String v2;
     ...
}
```

3.2.2 Generische Methoden

Nicht nur Klassen und *Interfaces*, auch Methoden können generisch sein. Ein Beispiel ist die folgende `swap`–Funktion, die den Inhalt von zwei Feldelementen austauscht:

```
final class C {
     public static <T> void swap(T[] a, int i, int j) {
          T tmp = a[i];
          a[i] = a[j];
          a[j] = tmp;
     }
}
```

Der Typparameter erscheint vor dem Ergebnistyp der Methode in spitzen Klammern. An der Verwendungsstelle muss, anders als bei generischen Klassen und *Interfaces*, der aktuelle Typparameter nicht angegeben werden. Der Compiler kann ihn aus der Verwendung erschließen:

```
String[] a = new String[2];
a[0] = "Hallo";
a[1] = "Wer Da?";

C.swap(a, 0, 1);
swap(a, 0, 1); // falls Aufruf in C
```

Der aktuelle Typparameter kann auch explizit angegeben werden. Allerdings muss dann die ausführliche Variante des Aufrufs mit Angabe der Klasse verwendet werden.

```
C.<String>swap(a, 0, 1);
<String>swap(a, 0, 1); // NICHT ERLAUBT <<--
```

Generische Methoden sind oft statisch, sie müssen es aber nicht sein. Sie können mehr als einen Typparameter haben und innerhalb einer generischen Klasse definiert werden:

```
final class C<T> {
     ...
     public <TX, TY> T f(TX x, TY y, T z) {
           if ( x.equals(y)) return z;
           else return null;
     }
     ...
}
```

`TX` und `TY` sind Typparameter von `f` und `T` ist Typparameter der Klasse `C` in der `f` definiert ist. Eine Anwendung von `f` sieht wie folgt aus:

```
C<Integer> p = new C<Integer>();
Integer x1 = p.f(1, 0.5, 2);       // implizite Typangabe
Integer x2 = p.<Integer, Double>f(1, 0.5, 2); // explizite Typangabe
```

Oder innerhalb von `C`:

```
final class C<T> {
     ...
     public <TX, TY> T f(TX x, TY y, T z) {
           if ( x.equals(y)) return z;
           else return null;
     }
     ...
     public void m() {
```

```
            ....
            T t = ...;
            T x1 = f(1, 0.5, t);
            T x2 = this.<Integer, Double>f(1, 0.5, t);
            ....
    }

}
```

In der Methode `m` ist `T` der ungebundene generische Parameter von `C`. `T1` und `T2` sind dagegen implizit bzw. explizit an `Integer` und `Double` gebunden.

3.2.3 Beschränkungen generischer Parameter

Generische Methoden mit beschränktem Typparameter

Gelegentlich ist nicht jeder Typ als generischer Parameter geeignet. Suchen wir beispielsweise in einem Feld von Hunden nach dem, der am lautesten bellen kann, dann kann die Suchfunktion nur auf solche Felder angewendet werden, deren Inhalt auf die Lautstärke des Bellens hin untersucht werden kann.

Nehmen wir an, dass die Lautstärke des Bellens durch einen `int`–Wert angegeben wird, den die Methode `belle` liefert. Die Eigenschaft, ein bellender Hund mit messbarer Lautstärke zu sein, wird dann durch das *Interface* `Klaeffer` ausgedrückt:

```
public interface Klaeffer {
     /*
      * returns: Lautstaerke des Bellens
      */
     int belle();
}
```

Ein kläffender Hund ist dann etwa:

```
public final class Hund implements Klaeffer {
     private int lautstaerke = 0;

     public Hund(int lautstaerke) {
           this.lautstaerke = lautstaerke;
     }

     public int belle() {
           ....
           return lautstaerke;
     }

     ...
}
```

In einem Feld voller Hunde suchen wir den lautesten mit `maxKlaeffer`:

```
Hund[] h = new Hund[5];
for ( int i=0; i< 5; i++)
     h[i] = new Hund((i+13)%7);

Hund lautester = maxKlaeffer(h);

System.out.println( lautester.belle() );
```

Die Methode `maxKlaeffer` könnte mit einem Parameter vom Typ `Hund[]` ausgestattet werden.

```
static Hund maxKlaeffer(Hund[] a) { ... }
```

Wir bräuchten aber dann eine zweite Methode, wenn wir später den lautesten in einer Schar bellender Wölfe finden wollten. Es ist darum besser, die Methode allgemeiner zu halten. Ein möglicher Ansatz wäre, das *Interface* `Klaeffer` als Parameter– und Ergebnistyp zu verwenden:

```
static Klaeffer maxKlaeffer(Klaeffer[] a) { ... }
```

Der Nachteil hierbei ist, dass der Ergebnistyp `Klaeffer` die weitere Verwendung des gefundenen Kläffers stark einschränkt. Wir müssten diese Methode ihr Ergebnis in eine Variable vom Typ `Klaeffer` ablegen lassen:

```
Hund   lautester = maxKlaeffer(h); // FEHLER: lautester liefert einen Klaeffer <<--
Klaeffer lautester = maxKlaeffer(h); // Typ Klaeffer statt Hund
```

Die Verwendung der Variablen `lautester` ist jetzt auf die Methode `belle` beschränkt, auch wenn in ihr in jedem Fall ein Hund sitzen wird.

Die Suchfunktion sollte generisch sein. Leider funktioniert Folgendes nicht:

```
public static <T> T maxKlaeffer(T[] a) {
     if ( a.length < 1)
         throw new NoSuchElementException();
     T max = a[0];
     for (T k: a)
           if ( k.belle() > max.belle() )
                 max = k;
     return max;
}
```

Klar: `belle` ist keine Methode von `Object`.

Wir brauchen eine Flexibilität im Typ der Kläffer, aber es dürfen nur Kläffer sein, die da kommen um ihre Lautstärke messen zu lassen. Diese Beschränkung lässt sich wie folgt ausdrücken:

```
public static <T extends Klaeffer> T maxKlaeffer(T[] a) {
     if ( a.length < 1)
         throw new NoSuchElementException();
     T max = a[0];
     for (T k: a)
           if ( k.belle() > max.belle() )
                 max = k;
     return max;
}

...
Hund[] h = ....
Hund lautesterHund = maxKlaeffer(h); // findet lautesten Hund
...
Wolf[] w = ...
Hund lautesterWolf = maxKlaeffer(w); // findet lautesten Wolf
...
Klaeffer[] k = ...
Klaeffer lautesterKlaeffer = maxKlaeffer(w); // findet lautesten Klaeffer
...
```

Durch den generischen Parameter `T` passt sich die Methode an die Aufrufstelle an: Gibt man ihr Hunde, liefert sie einen Hund. Gibt man ihr (kläffende) Wölfe, liefert sie einen Wolf. Gibt man ihr Kläffer, liefert sie Kläffer. Die *Beschränkung* des generischen Parameters

```
T extends Klaeffer
```

führt einerseits dazu, dass nur solche Typen als aktuelle Parameter akzeptiert werden, die das *Interface* `Klaeffer` implementieren, inklusive `Klaeffer` selbst. Zum anderen können innerhalb von `maxKlaeffer` alle Methoden verwendet werden, die in `Klaeffer` spezifiziert wurden.

Generische Klassen mit beschränktem Typparameter

Generische Parameter dürfen natürlich auch bei Klassen beschränkt werden. Die Syntax ist die gleiche wie bei Methoden. Im folgenden Beispiel definieren wir einen Stall mit Milchtieren als generische Klasse. Milchtiere sind Objekte, die man melken kann:

```
public interface MilchTier {
     Milch melken();
}
```

Ein generischer Stall für Milchtiere kann jetzt definiert werden als:

```
public class Stall<T extends MilchTier> {
     private static final int maxAnzahl = 10;
```

```
    private T[] tiere = (T[]) new Object[maxAnzahl];
    private int anzahl = 0;

    public void hinein(T t){
          if ( anzahl == maxAnzahl)
                throw new IllegalStateException();
          tiere[anzahl] = t;
          anzahl++;
    }

    public T hinaus(){
          if ( anzahl == 0)
                throw new IllegalStateException();
          anzahl--;
          return tiere[anzahl];
    }

    public Milch[] melkeAlle() {
          Milch[] eimer = new Milch[anzahl];
          for ( int i=0; i < anzahl; i++ )
                eimer[i] = tiere[i].melken();
          return eimer;
    }
}
```

Man beachte auch hier wieder den Unterschied zwischen einer generischen Lösung mit beschränktem Typparameter und einer Lösung ohne Typparameter, die sich auf das *Interface* bezieht. Bei der generischen Lösung geht keine Typinformation verloren. In einen Kuhstall gehen Kühe herein und auch als Kühe wieder hinaus.

```
Stall<Kuh> kuhStall = ....
Stall<Ziege> ziegenStall = ....
...
Kuh berta = kuhStall.hinaus(); // Beide Aufrufe von hinaus liefern ein Objekt mit
Ziege elsa = zeigenStall.hinaus(); // genauem Typ: entweder Ziege oder Kuh
```

Eine nicht–generische Lösung die sich auf das *Interface* bezieht ist dem sehr ähnlich:

```
public final class Stall<T extends MilchTier> {
    private static final int maxAnzahl = 10;
    private MilchTier[] tiere = new MilchTier[maxAnzahl];
    private int anzahl = 0;

    public void hinein(final Milchtier t){
          if ( anzahl == maxAnzahl)
                throw new IndexOutOfBoundsException();
          tiere[anzahl] = t;
          anzahl++;
    }

    public MilchTier hinaus(){
          if ( anzahl == 0)
                throw new IndexOutOfBoundsException();
          anzahl--;
          return tiere[anzahl];
    }

    public Milch[] melkeAlle() {
          Milch[] eimer = new Milch[anzahl];
          for ( int i=0; i < anzahl; i++ )
                eimer[i] = tiere[i].melken();
          return eimer;
    }
}
```

Allerdings gehen in einen solchen Stall Kühe und Ziegen hinein und kommen als bloße Milchtiere wieder heraus:

```
Stall<Kuh> kuhStall = ....
Stall<Ziege> ziegenStall = ....
...
```

```
MilchTier berta = kuhStall.hinaus(); // hinaus liefert Objekt nur
MilchTier elsa = ziegenStall.hinaus(); // mit ungenauem Typ Milchtier
```

Mit Hilfe generischer Klassen, *Interfaces* und Methoden kann also genauer mit Typinformationen umgegangen werden. Generisches passt sich der jeweiligen Situation besser an – wie es sich so gehört für Generisches.

3.3 Kollektionen und Kollektionstypen

3.3.1 Kollektionstypen

Kollektionen und Kollektionstypen: Beispiel Listen

Kollektionen sind Ansammlungen von Objekten. Ein Kollektionstyp ist ein Typ dessen Exemplare jeweils eine Kollektion von Objekten verwalten. Beispiel sind Menge, Listen, Stapel, und so weiter. Ein besonderes Merkmal von Kollektionen ist, dass sie meist mehr oder weniger unabhängig von der Art ihrer Elemente sind. Was die möglichen Operationen betrifft, unterscheidet sich eine Liste von Vektoren kaum von einer Liste von rationalen Zahlen, Katzen, oder Milchbauern. Man kann die jeweiligen Elemente einfügen, entnehmen, die ganze Liste auf der Suche nach einem bestimmten Element durchlaufen, und so weiter. Völlig unabhängig davon, um welche Art von Elementen es sich handelt. Die Kollektionstypen sind *generisch*.

Die Java–API enthält eine ganze Reihe von Kollektionstypen. Dass es sich dabei um generische Typen handelt, erkennt man an ihrem Typparameter. Eine Liste von `Integer`–Werten wird etwa wie folgt definiert und genutzt:

```
// Liste anlegen:
List<Integer> l = new ArrayList<Integer>();

// Werte einfuegen
l.add(1);
l.add(2);

// Liste durchlaufen
for ( Integer i: l )
   System.out.println(l);
```

Die Kollektionstypen der Java–API sind generisch. Sie gliedern sich jeweils in ein *(generisches) Interface* und mehrere *(generische) Klassen*, die es implementieren. In unserem Beispiel wird zunächst mit

```
List<Integer> l ...
```

eine Variable `l` vom Typ `List<Integer>` definiert. Bei `List<Integer>` handelt es sich um ein *generisches Interface*. Mit ihm sind die Fähigkeiten aller Listen über beliebigen Typen `E` festgelegt. In der API finden wir eine Beschreibung von

```
java.util Interface List<E>
```

In der Variablendefinition oben wird der generische Parameter `E` durch den aktuellen Typ `Integer` ersetzt. Damit ist der Typ von `l` festgelegt und damit die Methoden, die jedes Objekt ausführen kann, auf das `l` verweist. Mit der Zuweisung

```
... l = new ArrayList<Integer>();
```

wird `l` dann mit einem Wert belegt. Bei `ArrayList<Integer>` handelt es sich um eine generische Klasse, die das *Interface* `List<Integer>` implementiert. Es ist nicht die einzige Klasse mit dieser Fähigkeit. Ein Alternative wäre `LinkedList<Integer>`. Diese beiden Klassen unterscheiden sich nicht in ihren Fähigkeiten sondern in der Art der Implementierung. Die Klasse `ArrayList<E>` ist eine Listenimplementierung, die auf Feldern (Arrays) basiert, `LinkedList<Integer>` verkettet dagegen die Elemente über Referenzen. In manchen Anwendungen ist die eine Variante effizienter als die andere. Bei kleinen oder wenig genutzten Listen kann der Unterschied ignoriert werden.

- `List<E>` generisches *Interface*, beschreibt die Funktionalität von Listen.
- `ArrayList<E>` generische *Klasse*, Listenimplementierung auf Basis von Feldern.
- `LinkedList<E>` generische *Klasse*, Listenimplementierung auf Basis von verketteten Werten.

Die Fähigkeiten von Listen: Das Interface

Ein Blick in die API[10] auf die Beschreibung von `List<E>` zeigt, dass Listen eine Vielzahl von Methoden anbieten. Einige der wichtigsten Methoden sind:

`boolean add(E o)`	o an die Liste anhängen
`void add(int index, E element)`	Element an Position index in die Liste einfügen
`E get(int index)`	Element an Position index auslesen
`E remove(int index)`	Element an Position index auslesen und entfernen
`E set(int index, E element)`	Element an Position index ersetzen

[10] Sehen Sie wirklich öfter mal hinein. Die API–Dokumentation ist eine wichtige Informationsquelle. Sie sollten sich darin ein wenig auskennen.

Details und weitere Informationen entnehme man der API–Dokumentation.

Die Klassen `ArrayList<E>` und `LinkedList<E>` enthalten Implementierungen. Welche man für seine Anwendung auch auswählt, der Vorteil von Listen gegenüber Feldern ist in beiden Fällen eine Vereinfachung des Codes der Anwendung: Listen haben im Gegensatz zu Feldern keine feste Größe, sie können darum bei Bedarf beliebig wachsen. Listenelemente können bequem an jeder beliebigen Stelle eingefügt oder entnommen werden. Der Rest der Liste verschiebt sich automatisch.

Hat man eine Anwendung, bei der nichts eingefügt oder entnommen wird und bei der die Zahl der Elemente von Anfang an bekannt und fest ist, dann spricht natürlich nichts gegen den Einsatz eines Feldes.

Hinter dem *Interface* `List<E>` steckt das Konzept eines (zustandsorientierten) abstrakten Datentyps *Liste*, als einer geordneten Folge von Elementen, bei der man mit einem Index auf jede Position schreibend und lesend zugreifen kann.

Die Implementierung von Listen: Die Klassen

Die Klasse `ArrayList<E>` stellt eine Listenimplementierung zur Verfügung, die auf einem Feld basiert. Sie nimmt uns alle Verwaltungsoperationen ab. Ist das Feld voll, und wird ein Element hinzugefügt, dann kümmert sie sich darum, dass ein neues größeres Feld angelegt und die alten Elemente umkopiert werden. Ebenso kümmert sie sich um das Umkopieren der Elemente, wenn mitten im Feld etwas entfernt oder eingefügt wird.

Diese Art der Implementierung ist naturgemäß dann empfehlenswert, wenn oft auf bestimmte Indexpositionen lesend oder ersetzend zugegriffen wird. Weniger günstig sind häufiges Einfügen und Entfernen.

Eine `LinkedList<E>` organisiert die Listenelemente in verketteten Knoten. D.h. jedes Element wird in einem kleinen Speicherbereich – dem sogenannten Knoten (engl. *node*) – abgelegt, der neben dem Wert selbst noch einen Verweis auf den nächsten und den vorherigen Knoten enthält. Der Zugriff auf beliebige Elemente innerhalb der Liste ist bei einer solchen Organisation weniger günstig. Es muss ja jedes Mal die ganze Kette bis zum gewünschten Element durchlaufen werden. Einfügen und Entfernen können dagegen effizienter implementiert werden, da nichts verschoben werden muss.

Datenstruktur

Mit dem Begriff *Datenstruktur* bezeichnet man die Art der Organisation von Daten im Speicher. Ein Feld ist eine Datenstruktur: Die Daten liegen dicht nebeneinander. Eine verkettete Liste ist eine andere Datenstruktur: Die Daten liegen verstreut im Speicher und zu jedem Datenwert wird ein Verweis auf den nächsten gespeichert.

Das sind nur zwei Beispiele. Die Untersuchung von Datenstrukturen und den zu ihnen passenden Algorithmen ist ein wichtiges Feld der Informatik. Eine geschickte Organisation der Daten ist ganz wesentlich für die Effizienz, mit der Operationen auf ihnen ausgeführt werden können. Speziell dann, wenn es sich um große Mengen von Daten handelt.

Datentyp

Der Begriff *Datentyp* (ob abstrakt oder nicht) ist komplementär zum Begriff *Datenstruktur*. Mit *Datenstruktur* bezieht man sich auf die “Innereien”, d.h. auf die Art wie etwas implementiert ist. Der *Datentyp* bezieht sich dagegen auf die äußere Erscheinung: was kann das Ding. Wie es sein Können realisiert, ist unerheblich.

Der Datentyp ist mit einer Schnittstelle, einem *Interface* verbunden. Die Verwendung einer bestimmten Datenstruktur ist eine Entwurfsentscheidung für die Implementierung von Klassen. Natürlich gilt dies speziell dann, wenn es sich um Klassen handelt, die einen Kollektionstyp implementieren. In dem Fall haben sie ja viele Elemente zu verwalten, die in irgendeiner Form im Speicher abgelegt werden müssen.

Datentyp und Datenstruktur

Zwischen Datentypen und Datenstrukturen bestehen gewisse Beziehungen. Das führt gelegentlich zu einer Verwechslung oder Identifizierung der beiden Begriffe. Eine bestimmte Datenstruktur eignet sich oft gut für bestimmte Operationen. Die Datenstruktur *verkettete Liste (!)* eignet sich gut dazu Elemente einzufügen und zu entfernen. Natürlich kann man sie auch durchlaufen und Werte an bestimmten Indexpositionen auslesen oder ersetzen. Man muss dazu ja nur von Knoten zu Knoten gehen und dabei mitzählen. Die *Organisation* der Datenkollektion unterstützt oder ermöglicht also eine bestimmte Menge an *Operationen* auf der Datenkollektion. Trotzdem sind die *Organisation* (die *Datenstruktur*) und die Menge der unterstützten Operationen (der *Datentyp*) zwei unterschiedliche Dinge.

Dass Datenstruktur und Datentyp nur bedingt in Beziehung zu setzen sind, sehen wir an der Klasse *ArrayList*. Die Datenstruktur ist ein *Feld*. Der Datentyp ist *Liste*. *LinkedList* dagegen ist eine Implementierung des Datentyps *Liste* mit Hilfe der Datenstruktur

verkettete Liste.

Das *Collection Framework*

Das Paket `java.util` enthält eine Reihe von Klassen und Schnittstellen zum Umgang mit Kollektionen, die zusammen als *Collection Framework* (etwa *Kollektions–Rahmen*) bezeichnet werden. Einige der wichtigsten Bestandteile dieses *Frameworks* sind:

- **Interfaces:** Datentypen / Funktionalitäten werden durch Interfaces beschreiben
  - `List<E>` Listen
  - `Set<E>` Mengen
  - `Map<K, V>` Abbildungen
- **Kollektionsklassen:** unterschiedliche Implementierungen der Datentypen / Funktionalitäten
  - Zu `List<E>`
    - `ArrayList<E>` Implementierung durch die Datenstruktur Feld
    - `LinkedList<E>` Implementierung durch die Datenstruktur verkettete Liste
  - Zu `Set<E>`
    - `TreeSet<E>` Implementierung durch die Datenstruktur Baum
    - `HashSet<E>` Implementierung durch die Datenstruktur Hash–Tabelle
  - Zu `Map<K, V>`
    - `TreeMap<K, V>` Implementierung durch die Datenstruktur Baum
    - `HashMap<K, V>` Implementierung durch die Datenstruktur Hash–Tabelle
- **Iteratoren:** `Iterator<E>:` Läufer durch eine Liste oder eine Menge
- **Funktionen:** `Collections:` Eine Klasse mit nützlichen Funktionen in Form statischer Methoden
- **Diverse Hilfsklassen** zur Unterstützung der Implementierung eigener Kollektionsklassen

Das hier sind, wie gesagt, nur einige der Bestandteile des *Collection Frameworks* und auch auf Details wollen wir hier nicht eingehen. Die vollständige Beschreibung findet sich in der API–Dokumentation.Das Organisationsprinzip der vorgefertigten Kollektionsklassen[11] ist klar zu erkennen. Einem (Daten–) Typ in Form eines *Interface* sind Implementierungen auf Basis unterschiedlicher Datenstrukturen gegenüber gestellt.

3.3.2 Iteratoren

Iteratoren

Mengen und Listen sind *iterierbar*. Das bedeutet, dass man mit einer *Foreach*–Schleife ihre Elemente durchlaufen kann:

```
List<X> l = ...;
....
for ( X x : l )
  // tue etwas mit x
```

Die Variable `x` nimmt dabei sukzessive jeden Wert der Liste `l` an. Diese Schleife ist kein direktes Sprachelement von Java. Der Compiler setzt sie in ein etwas komplexeres Konstrukt folgender Form um:

```
Iterator<Integer> i = l.iterator();
while ( i.hasNext() )
// tue etwas mit i.next()
```

Ein *Iterator* ist ein "abstrakter Zeiger" der in eine Containerklasse, wie beispielsweise eine Liste, zeigt (siehe Abb. 3.5). Mit einem Iterator können die Elemente systematisch durchlaufen werden, ohne dass die Anwendung die Implementierung des Containers kennen muss. Sie muss wie in diesem Beispiel nicht einmal wissen, in welcher Datenstruktur die Elemente überhaupt gespeichert sind. Egal, ob die Liste ein Exemplar von `ArrayList<X>` oder von `LinkedList<X>` ist, sie kann mit dem Iterator durchlaufen werden.

[11] Man nennt die fertigen Kollektionen wie `ArrayList<E>` *legacy collections*.

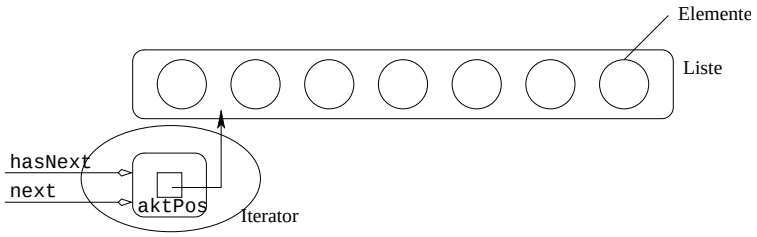


Abbildung 3.5: Iteratorkonzept

Der Iterator selbst kennt natürlich die Implementierung der Liste. Die Klassen `ArrayList<E>` und `LinkedList<E>` haben jeweils ihre eigene *Iterator–Implementierung*. Und je nach dem, ob es sich bei der Liste um ein Exemplar von `ArrayList<X>` oder von `LinkedList<X>` handelt, wird `i` auf eine entsprechende Instanz verweisen. Die Anwendung (also wir!) kennt aber nicht einmal den Namen dieser Iteratorklassen. Das einzige, was wir wissen und wissen müssen, ist, dass `i` sich auf ein Exemplar irgendeiner Iteratorklasse bezieht, also einer Klasse die das *Interface* `Iterator<X>` implementiert.

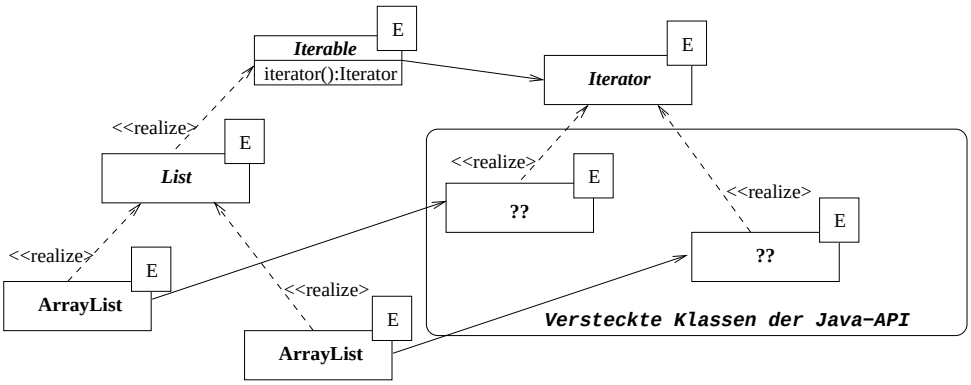


Abbildung 3.6: Iteratoren: Interface und unbekannte Implementierungsklassen

Spezielle Iteratoren auf Listen: ListIterator

Iteratoren haben eine speziell auf Listen zugeschnittene Variante `ListIterator<E>`. Mit einem Listeniterator kann man eine Liste in beide Richtungen durchlaufen und Elemente einfügen. Beispiel:

```
// an der richtigen Stelle s in l einfuegen:
public void insert(final String s, final List<String> l) {
  ListIterator<String> iter = l.listIterator();
  boolean found = false;
  while ( iter.hasNext() ) {
    if ( iter.next().compareTo(s) >= 0 ) {
      found = true;
      break;
    }
  }
  if ( found )   // Position gefunden
    iter.previous(); // ein Schritt zurueck
  iter.add(s);   // einfuegen, eventuell am Ende
}
```

3.3.3 Mengen

Mengen: `HashSet` oder `TreeSet`

Der Datentyp der Mengen mit dem generischen *Interface* `Set<E>` beschreibt das Konzept der Mengen. Eine Menge ist eine Kollektion ohne Duplikate und ohne innere Ordnung. In einer Liste kann ein Element mehrfach vorhanden sein, in einer Menge

nicht. In einer Liste kann man auf die i–te Position zugreifen, in einer Menge nicht.

Eine Menge setzt man ein, wenn man – ohne eigenes Zutun – vermeiden will, dass Elemente mehrfach in eine Kollektion eingefügt werden können. Etwas ist drin oder eben nicht. Das zweite, wichtigere, Einsatzgebiet von Mengen ergibt sich daraus, dass sie *nicht* verpflichtet sind, ihre Elemente bestimmten Positionen zuordnen müssen. Eine Menge ist damit frei, sie intern nach eigenen Kriterien zu organisieren, beispielsweise so, dass alle oder bestimmte Zugriffe besonders schnell ausgeführt werden können.

Das *Interface* `Set` definiert die möglichen Operationen. Die beiden wichtigsten (*legacy*) Implementierungen sind `TreeSet` und `HashSet`. Ein `TreeSet` speichert seine Daten in einer baumartigen Struktur. Dadurch ist ein Zugriff auf das kleinste Element besonders schnell. Ein Iterator über ein `TreeSet` liefert die Elemente in aufsteigender Ordnung. Ein `TreeSet` erwartet, dass seine Elemente vergleichbar sind, d.h. dass sie `Comparable` implementieren.

Ein `HashSet` speichert seine Elemente in einer Hash-Tabelle. Damit ist keine Ordnung verbunden. Ein Iterator liefert die Elemente in willkürlicher Reihenfolge. Operationen auf einer solchen Menge sind gleichmäßig effizient, unabhängig von der Größe des Elements, auf das sie sich beziehen. Die Elemente in einem `HashSet` müssen nicht vergleichbar sein.

Beispiel: Sortieren mit einem TreeSet

Ein `Tree` hält seine Elemente stets in sortierter Form. Genau gesagt ist ein `TreeSet` schwach sortiert. Mit *schwach sortiert* ist eine Organisation der Daten gemeint, bei der das kleinste Element am Anfang steht und die kleineren Elemente in der Nähe des Anfangs zu finden sind, ohne dabei genau nach Größe sortiert zu sein. Die schwache Sortierung erlaubt es, schnell auf das kleinste Element zugreifen zu können. Einfügeoperationen sind aber immer noch halbwegs schnell, da sie die Ordnung nicht perfekt erhalten müssen.

Das macht ein `TreeSet` zu einem recht guten Sortiermechnanismus. Man fügt die zu sortierenden Elemente ein und der Iterator liefert sie dann sortiert wieder aus. Beispielsweise können wir so die Worte in einer Datei sortieren:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.StringTokenizer;
import java.util.TreeSet;

import javax.swing.JOptionPane;

public final class TreeSorting {
    private TreeSorting() { }

public static void main( String[] args ) {

          // TreeSet in das sortiert wird:
  TreeSet<String> treeSet = new TreeSet<String>();

          // zu sortierende Datei:
  String fileName = JOptionPane.showInputDialog("Datei:");
  File file = new File(fileName);
  String inputLine = null;

  long t0 = 0; // Zeitpunkt
  try {
       BufferedReader reader = new BufferedReader(new FileReader(file));

       t0 = System.currentTimeMillis(); // Zeitnehmen

           // Wort fuer Wort in TreeSet einfuegen:
       while ( (inputLine = reader.readLine()) != null ) {
         StringTokenizer st = new StringTokenizer(inputLine);
         while ( st.hasMoreElements() ){
           treeSet.add(st.nextToken());
         }
       }
  } catch(Exception e) {
   e.printStackTrace();
      }
      long t1 = System.currentTimeMillis();
```

```
          // gelesene Worte ohne Duplikate sortiert ausgeben:
  for(String s: treeSet)
    System.out.println( s );

  long runTime = t1-t0;
  JOptionPane.showMessageDialog( null,
     "verbrauchte Zeit: " + runTime);
  System.exit(0);
}
}
```

Die Zeitmessung haben wir eingeführt um zu demonstrieren wie schnell eine derartige Sortierung ist. Algorithmisch ambitionierte Leser mögen diesen mit eigenen Sortieralgorithmen vergleichen.

3.3.4 Listen

Das *Interface* `List`

Listen sind Kollektionen von Elementen, die an bestimmten *Positionen* zu finden sind. Der Unterschied zu Mengen besteht zum einen darin, dass bei Listen die Anwendung der Liste die Kontrolle darüber behält, an welcher Position ein Element entnommen oder eingefügt wird. Zum anderen kontrolliert eine Liste nicht, wie oft ein Element eingefügt wird.

Die wichtigsten Operationen auf Listen haben wir bereits weiter oben aufgelistet. In der API–Dokumentation findet sich die vollständige Information.

Beispiel: Sortieren mit einer Liste

Als Beispiel für den Umgang mit einer Liste zeigen wir eine einfache Sortierfunktion. Man beachte, dass die Funktion Listen beliebiger Implementierung und beliebigen Inhalts sortieren kann. Die einzige Anforderung ist, dass die Listenelemente vergleichbar sind.

```
public static <T extends Comparable<T> > void sort(final List<T> l) {
   int n = l.size();
   int m;
   for (int i = 0; i < n; i++) {
     m = minAb(l, i);
     swap(l, i, m);
   }
}
```

mit den Hilfsfunktionen:

```
static <T extends Comparable<T>> int minAb(final List<T> l, final int i) {
int n = l.size();
int m = i;
for (int j = i + 1; j < n; j++) {
  if ( l.get(j).compareTo(l.get(m)) < 0);
    m = j;
}
return m;
}

static <T> void swap(final List<T> l, final int i, final int j ) {
T e_i = l.get(i);
T e_j = l.get(j);
l.set(i, e_j);
l.set(j, e_i);
}
```

Der Algorithmus ist der gleiche, den wir weiter oben (siehe 87) zur Sortierung von Feldern eingeführt haben.

3.3.5 Abbildungen

Das *Interface* `Map`

Abbildungen ordnen einem Wert – allgemein *Schlüssel* (engl. *Key*) genannt – einen anderen Wert zu, den *Wert* (engl. *Value*) zum Schlüssel. Jedem Schlüssel ist dabei immer genau ein Wert zugeordnet. Einige wichtige Operationen auf Abbildungen sind:

`V get(Object key)`	Liefert den Wert der unter dem Schlüssel `key` gespeichert ist
`Set<K> keySet()`	Liefert die Menge aller Schlüssel
`V put(K key, V value)`	Speichert `value` als Wert des Schlüssels `key`
`boolean containsKey(Object key)`	Enthält die Abbildung einen Wert für Schlüssel `key`?
`Collection<V> values()`	Liefert die Menge aller Werte

Eine vollständige Auflistung und Beschreibung findet sich in der API–Dokumentation.

Beispiel: Alle Vorkommen eines Wortes in einer Datei ausgeben

Ein Wort kann in einer Datei in mehreren Zeilen vorkommen. Das kann in einer Abbildung von Worten (Strings) auf Zeilennummer festgehalten werden. Als Beispiel für die Verwendung von Abbildungen zeigen wir in Variation zu oben eine kleine Funktion, die alle Vorkommen eines Wortes sortiert und mit Angabe der Zeilennummer(n) des Wortes ausgibt. Die Funktionalität *Abbildung* liefert uns jede `Map`, die Sortiertheit ergibt sich aus der Verwendung einer `TreeMap`:

```
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;

public final class MapSorting {

      private MapSorting() { }

      public static void main(String[] args) {
            Map<String, List<Integer>> occ = new TreeMap<String, List<Integer>>();
            String fileName = JOptionPane.showInputDialog("Datei:");
            File file = new File(fileName);
            String inputLine = null;
            long t0 = 0;
            try {
                  BufferedReader reader = new BufferedReader(new FileReader(file));

                  int lineNr = 0;
                  while ((inputLine = reader.readLine()) != null) {
                        StringTokenizer st = new StringTokenizer(inputLine);
                        lineNr++;
                        while (st.hasMoreElements()) {
                              String word = st.nextToken();
                              if ( ! occ.containsKey(word)) {
                                    List<Integer> lines = new LinkedList<Integer>();
                                    lines.add(lineNr);
                                    occ.put(word, lines);
                              } else {
                                    List<Integer> lines = occ.get(word);
                                    lines.add(lineNr);
                              }
                        }
                  }
            } catch (Exception e) {
                  e.printStackTrace();
            }

            for (String w : occ.keySet()) {
```

```
                    System.out.print(w);
                    for (Integer line: occ.get(w))
                          System.out.print(" "+line);
                    System.out.println();
              }
        }
}
```

3.4 Definition von Kollektionstypen

3.4.1 Schlangen und Warteschlangen

Selbst definierter oder API–Typ

Bevor man sich der Mühe unterzieht eine eigene Klasse zu definieren, sollte geprüft werden, ob das Gesuchte nicht schon in der Java–API zu finden ist. Speziell dann, wenn es um allgemeine Funktionalitäten wie die einer Kollektionsklasse geht, hat die Java–API für viele gängige und einige weniger gängige Anforderungen eine vorgefertigte (*"legacy"*) Lösung zu bieten. Deren Qualität wird man nur mit einigem Aufwand erreichen oder gar übertreffen. Sollte es sich aber doch herausstellen, dass es sinnvoll ist, eine bestimmte Funktionalität mit einer eigenen Klasse zu realisieren, dann muss geklärt werden, ob

- ein bekannte Schnittstelle mit einer anderen, besseren Implementierung versehen werden soll, oder
- eine neue Schnittstelle definiert und implementiert werden soll.

Im Zweifelsfall ist natürlich das erste vorzuziehen. Zur Illustration zeigen wir die Reimplementation einer bekannten Funktionalität. Das ist eine interessante Aufgabe, denn hierbei können wir auf die Konventionen und Restriktionen der Java–API eingehen, die dabei zu beachten sind.

Java-API: `Queue`

Eine *Warteschlange*, (engl. *Queue*) ist ein Kollektionstyp der seine Elemente nach Art der Warteschlangen verwaltet: An einem Ende ("hinten") werden Elemente eingereiht, am anderen Ende ("vorn") werden sie wieder entnommen. Diese Funktionalität bringt

`java.util` *Interface* `Queue<E>`

mit seinen beiden zentralen Methoden

```
boolean add(E o)
E remove()
```

zum Ausdruck: `add` fügt (hinten) an, `remove` holt (vorn) weg.

Eine Warteschlange kann nach *FIFO*–Art agieren. *FIFO* bedeutet *First In, First Out*. Wer sich zuerst hinten anstellt kommt auch zuerst aus der Warteschlange wieder heraus. `java.util.Queue<E>` verlangt keine FIFO–Verwaltung. Es bleibt der Implementierung überlassen, ob und eventuell wie die Elemente nach jedem Einfügen umsortiert werden. Im Normalfall wird die Implementierung die Elemente nach FIFO–Art verwalten. Beispielsweise implementiert `java.util.LinkedList<E>` das *Interface* `java.util.Queue<E>` und verhält sich dabei in der erwarteten Art. Ein FIFO–Verhalten ist aber nicht unbedingt erforderlich.

Java-API: `PriorityQueue`

Das *Interface* `Queue` hat mehrere Implementierungen. Die meisten, wie etwa `LinkedList<E>`, sind *FIFO*–Implementierungen. Eine andere Art der Verwaltung bietet

`java.util` *Class* `PriorityQueue<E>`

Bei ihr handelt es sich nicht um eine FIFO–Warteschlange, sondern um eine Prioritäts–Warteschlange.

Eine *Prioritäts–Warteschlange* (engl. *Priority Queue*) ist eine Warteschlange in der die Objekte nach "Priorität" eingefügt werden. Die `remove`–Operation liefert hier immer das Element mit der *höchsten Priorität*. Prioritäten sind eine Eigenschaft der in der Schlange gespeicherten Elemente. Die Priorität der Elemente stellt `java.util.PriorityQueue` durch einen Vergleich mit `compareTo` fest. Wird also in eine `PriorityQueue` ein neues Element `e` eingefügt, dann vergleicht die Warteschlange es mit jedem Element `x` das bereits gespeichert ist, um herauszufinden welches jetzt das kleinste Element ist und um dieses dann nach vorn zu bringen. *Priorität* wird also als *Kleinsein* interpretiert.

Ein solches Verhalten könnte man leicht über eine Sortierung der Elemente spezifizieren:

```
   entries: Sequence(E)
   inv: entries sortiert entsprechend E.compareTo, kleinere zuerst

context PriotityQueue<E>.add(x)
     PRE:
     POST: y IN entries@pre => y IN entries
```

```
          x IN entries

context PriotityQueue<E>.remove
      PRE:
      POST: entries = rest(entries@pre)
      RETURNS:first(entries@pre)
            if len(entries@pre) = 0 throw NoSuchElementException
    }
```

Hiermit soll ausgedrückt werden, dass eine `PriorityQueue` sich so verhält, als bestände sie aus einer sortierten Folge von Einträgen. Beim Einfügen wird dem neuen Element ein Platz entsprechend seiner Größe zugewiesen. In der Spezifikation wird das durch die Invariante zum Ausdruck gebracht. Beim Herausholen wird das erste entnommen das dann natürlich das Kleinste ist.

Wie gesagt, `PriorityQueue` ist über eine sortierte Folge *spezifiziert*. Man kann sich also *vorstellen* sie sei auch so *implementiert*. Wie sie tatsächlich implementiert ist, bleibt das Geheimnis der Entwickler der Java–API, ganz sicher aber wird es nicht so etwas Simples wie eine sortierte Liste sein.

Sortieren mit einer PriorityQueue

Mit einer `PriorityQueue` kann man alles sortieren, was vergleichbar ist. Beispielsweise Strings. Man sortiert sie einfach durch Einfügen in eine `PriorityQueue`.

```
import java.util.PriorityQueue;
import java.util.Queue;

public final class Sorting {

  private Sorting() { }

  public static void main(String[] args) {
    Queue<String> q = new PriorityQueue<String>();

    q.add("hallo"); // einfuegen
    q.add("ist");
    q.add("da");
    q.add("jemand");
    q.add("hallo");

    for( String s : q ) // sortiert ausgeben
      System.out.println(s);
  }
}
```

Hier sieht man, dass `PriorityQueue` eine starke Ähnlichkeit mit `TreeSet` hat. Beide sortieren. Eine `PriorityQueue` erhält Duplikate, `TreeSet` eliminiert sie. Beide Kollektionen ordnen ihre Elemente, die eine, `PriorityQueue`, weil das Ordnen zu ihrer Spezifikation gehört, die andere, `TreeSet`, weil es für eine effiziente Implementierung der Mengenoperationen eingesetzt wird.

3.4.2 Warteschlange als Liste

Integration in den Rahmen der Kollektionsklassen

Wenn wir eine eigene Klasse für Warteschlangen definieren, dann stellt sie eine Alternative zu `PriorityQueue` dar. Sie sollte sich zunächst einmal wie diese in den Rahmen (das *Framework*) der Kollektionsklassen einpassen. `PriorityQueue` implementiert `Queue`. Es liegt also nahe zu definieren:

```
public final class ListPriorityQueue<E extends Comparable<E>> implements Queue<E> {
  ....
}
```

Die Klasse ist generisch im Typ `E` der Elemente, `E` muss ein vergleichbarer Typ sein und schließlich muss das *Interface* `Queue<E>` implementiert werden.

Geben wir dies beispielsweise in Eclipse ein, dann werden wir sofort darüber informiert, dass etliche Methoden zu implementieren sind. Wir lassen uns von Eclipse die entsprechenden Methoden erzeugen und erhalten so etwas wie:

```
public final class ListPriorityQueue<E extends Comparable<E>> implements Queue<E> {

public E element() {
   // TODO Auto-generated method stub
   return null;
}

public boolean offer(E arg0) {
   // TODO Auto-generated method stub
   return false;
}

public E peek() {
   // TODO Auto-generated method stub
   return null;
}

    .... ETC. ....
}
```

Implementierung mit einer Liste

Eine besonders einfache Implementierung einer Warteschlange setzt die Spezifikation direkt um. Ein solche Warteschlange verhält sich nicht nur so, als bestände sie aus einer sortierten Liste, sie besteht tatsächlich aus einer sortierten Liste.

```
public final class ListPriorityQueue<E extends Comparable<E>> implements Queue<E> {

 private List<E> entries = new LinkedList<E>();
     ...
}
```

Die meisten Methoden sind leicht zu implementieren. Wir lesen in der API–Dokumentation nach, was sie leisten sollen und in der Regel kann dies direkt an eine Methode der Liste delegiert werden:

```
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Queue;

public final class ListPriorityQueue<E extends Comparable<E>> implements Queue<E> {

  private List<E> entries = new LinkedList<E>();

  public E element() {
     return entries.get(0);
   }

  public boolean offer(E arg0) {
    if ( arg0 == null ) // null kann nicht eingefuegt werden
      return false;
    entries.add(arg0);
    Collections.sort(entries);
    return true;
  }

  public E peek() {
    E first = null;
    try {
      first = entries.get(0);
    } catch (NoSuchElementException ex) { }
    return first;
```

```
}

public E poll() {
  E first = null;
  try {
    first = entries.remove(0);
  } catch (NoSuchElementException ex) { }
  return first;
}

public E remove() {
  return entries.remove(0);
}

public boolean add(E arg0) {
  if ( arg0 == null ) // add muss null mit Exception ablehnen
    throw new NullPointerException();
  return offer(arg0);
}

// das wollen wir (noch) nicht implementieren
public boolean addAll(Collection<? extends E> arg0) {
  throw new UnsupportedOperationException();
}

public void clear() {
  entries.clear();
}

public boolean contains(Object arg0) {
  if ( arg0 == null )
    throw new NullPointerException();
  return entries.contains(arg0);
}

//  das wollen wir (noch) nicht implementieren
public boolean containsAll(Collection<?> arg0) {
  throw new UnsupportedOperationException();
}

public boolean isEmpty() {
  return entries.isEmpty();
}

public Iterator<E> iterator() {
  // TODO Auto-generated method stub
  return null;
}

// Entfernen beliebiger Elemente soll noch moeglich sein
public boolean remove(Object arg0) {
  throw new UnsupportedOperationException();
}

   // Entfernen beliebiger Elemente soll nicht moeglich sein
public boolean removeAll(Collection<?> arg0) {
  throw new UnsupportedOperationException();
}

// Entfernen beliebiger Elemente soll nicht moeglich sein
public boolean retainAll(Collection<?> arg0) {
  throw new UnsupportedOperationException();
}

public int size() {
  return entries.size();
}
```

```
  // das wollen wir (noch) nicht implementieren
  public Object[] toArray() {
    throw new UnsupportedOperationException();
  }

  // das wollen wir (noch) nicht implementieren
  public <T> T[] toArray(T[] arg0) {
    throw new UnsupportedOperationException();
  }
}
```

Einige Operationen wollen wir nicht implementieren, da sie nicht zu unserem Konzept einer Prioritätswarteschlange passen:

- remove
- removeAll
- retainAll

Diese Methoden entfernen Elemente "mitten" aus der Warteschlange. Das soll nicht möglich sein. Die entsprechenden Methoden werfen eine `UnsupportedOperationException`.

Dann gibt es Methoden, die wir nicht implementieren wollen, weil sie über unseren aktuellen Kenntnisstand hinausgehen. Das sind:

- removeAll
- retainAll
- toArray()
- toArray(T[] arg0)

Es trifft sich dabei gut, dass wir `removeAll` und `retainAll` mit ihrem seltsamen Fragezeichen nicht nur nicht implementieren können, wir wollen es auch nicht.

Beim Einfügen hängen wir das neue Element an und sortieren danach die Liste. Auf die Art bleibt die Sortierung erhalten:

```
public boolean offer(E arg0) {
  if ( arg0 == null ) // null kann nicht eingefuegt werden
    return false;
  entries.add(arg0);
  Collections.sort(entries);
  return true;
}
```

Die Methode `offer` reagiert ein Angebote ein neues Element anzunehmen. Unsere Kollektion mag es nicht wenn `null` eingefügt wird. Prinzipiell kann `null` in eine Kollektion eingefügt werden. In unserem Fall ist das aber nicht ratsam, da ein Vergleich mit `null` nicht möglich ist und `null`–Elemente darum zu Problemen beim Sortieren führen werden.[12]

ListIterator

Das Sortieren mit `Collections.sort` ist einfach aber etwas zu aufwändig. Die Liste, in die eingefügt werden soll, ist ja bereits sortiert. Hängen wir ein Element an, dann ist höchstens dieses eine Element am falschen Platz. Statt zu sortieren laufen wir darum einfach so lange durch die Liste, bis der richtige Platz zum Einfügen gefunden wurde. Der richtige Platz ist direkt vor der ersten Position mit einem größeren Element:

```
public boolean offer(E arg0) {
  if ( arg0 == null )
    return false;
  ListIterator<E> iter = entries.listIterator();
  while ( iter.hasNext() ) {
    if ( iter.next().compareTo(arg0) >= 0 ) {
      iter.previous();
      break;
```

[12] Die Java–API stellt *abstrakte Klassen*, wie etwa `AbstractQueue` zur Verfügung, mit denen die Implementierung eigener Kollektionsklassen erleichtert wird. Da die Unterstützung nicht wirklich sehr essentiell ist und wir uns noch nicht mit dem Konzept der abstrakten Klassen beschäftigt haben, verzichten wir hier auf deren Einsatz.

```
    }
  }
  iter.add(arg0);

  return true;
}
```

Zum Durchlaufen der Liste nehmen wir einen `ListIterator`. Ein `ListIterator` ist eine erweiterte Variante des Iterators, der es erlaubt Elemente einzufügen und rückwärts zu gehen. Das ist hier notwendig. Wir müssen Einfügen und vorher rückwärts gehen, da das neue Element ja *vor* die erste Position mit einem größeren eingefügt wird.

Iterator

Die letzte unimplementierte Methode ist `iterator`. Die wollen und können wir implementieren. Jede anständige Kollektion unterstützt Iteratoren. Wir können das Problem hier ganz besonders einfach lösen:

```
public Iterator<E> iterator() {
  return entries.iterator();
}
```

Der Iterator der Liste reicht als Iterator unserer Prioritäts–Warteschlange völlig aus.

3.4.3 Abbildung als binärer Suchbaum

Binäre Suchbäume und Abbildungen

Ein *Baum* ist eine Datenstruktur die aus Knoten besteht, die Informationen tragen, und die durch gerichtete Kanten miteinander verbunden sind. Führt eine Kante von einem Knoten k_1 zu einem Knoten k_2, dann sagt man auch k_2 ist ein Nachfolger (ein Kind) von k_1. Ein *Binärbaum* ist ein Baum, bei dem ein Knoten maximal 2 Nachfolger besitzt.

Ein binärer Baum ist ein *binärer Suchbaum* oder kurz *Suchbaum*, wenn der linke Unterbaum eines Knotens nur Informationen enthält die kleiner sind, als die des Knotens selbst und der rechte Unterbaum nur Informationen die größer sind als die des Knotens. Ein Suchbaum eignet sich für eine schnelle Suche. Man kann sie einsetzen um Abbildungen zu realisieren.

Eine Abbildung ordnet einem Schlüssel (engl. *Key*) einen Wert zu. Ein Telefonbuch ist eine typische Abbildung: Den Namen als Schlüsseln werden Telefonnummern als Werte zugeordnet. Abbildungen sind in vielen Anwendungen eine sehr hilfreiche Datenabstraktion. Die Java–API bietet dazu die Schnittstelle `Map` mit zwei Implementierungen, `TreeMap` und `HashMap` an. Zur Illustration soll hier eine eigene Implementierung entwickelt werden, die binäre Suchbäume als Datenstruktur nutzt.

Einbettung in den Rahmen der Kollektionstypen

Auch eine eigene Implementierung sollte an die üblichen Java–Konventionen angepasst werden. Wir implementieren eine Abbildungs–Variante. Also muss das *Interface* `Map` implementiert werden. Da intern ein Suchbaum verwendet werden soll, müssen die Schlüssel vergleichbar sein. Das führt uns zu folgendem Ansatz einer Klassendefinition:

```
public final class SearchTreeMap<K extends Comparable<K>, V> implements Map<K, V>{
  ....
}
```

`K` steht für den Typ der Schlüsselwerte und `V` für den Typ der zugeordneten Werte. Eclipse generiert uns gleich die notwendigen Methodendefinitionen:

```
public final class SearchTreeMap<K extends Comparable<K>, V> implements Map<K, V>{
public void clear() {
  // TODO Auto-generated method stub

}

public boolean containsKey(Object arg0) {
  // TODO Auto-generated method stub
  return false;
}
```

```
 public boolean containsValue(Object arg0) {
    // TODO Auto-generated method stub
    return false;
 }

     ... diverse andere Methoden mehr ....

 }
```

In der API–Dokumentation informieren wir uns über diese Methoden und entscheiden, welche eventuell nicht implementiert werden sollen. Nach dieser Entscheidung sieht unsere Klasse etwa wie folgt aus:

```
public final class SearchTreeMap<K extends Comparable<K>, V> implements Map<K, V>{

 public boolean containsKey(Object arg0) {
    // TODO Auto-generated method stub
    return false;
 }
 public V get(Object arg0) {
    // TODO Auto-generated method stub
    return null;
 }
 public boolean isEmpty() {
    // TODO Auto-generated method stub
    return false;
 }
 public V put(K arg0, V arg1) {
    // TODO Auto-generated method stub
    return null;
 }
 public V remove(Object arg0) {
    // TODO Auto-generated method stub
    return null;
 }
 public int size() {
    // TODO Auto-generated method stub
    return 0;
 }


     // Nicht unterstuetzte Methoden-------------------------
 public Collection<V> values() {
    throw new UnsupportedOperationException();
 }
 public Set<K> keySet() {
    throw new UnsupportedOperationException();
 }
 public void clear() {
    throw new UnsupportedOperationException();
 }
 public boolean containsValue(Object arg0) {
    throw new UnsupportedOperationException();
 }
 public Set<java.util.Map.Entry<K, V>> entrySet() {
    throw new UnsupportedOperationException();
 }
 public void putAll(Map<? extends K, ? extends V> arg0) {
    throw new UnsupportedOperationException();
 }
}
```

Statische innere Klasse

Wollen wir unsere Abbildung als Suchbaum realisieren, dann müssen die Elemente in Knoten gespeichert werden. Dazu ist eine entsprechende Klasse `Node` notwendig. `Node` wird *nur hier* in `SearchTreeMap`, benötigt. Wir machen darum Gebrauch von der Möglichkeit, eine Klasse innerhalb einer anderen definieren zu können:

```
public final class SearchTreeMap<K extends Comparable<K>, V> implements Map<K, V>{

 private static class Node<T1, T2> {
   Node(T1 k, T2 v) {
     this.key = k;
     this.value = v;
   }

   Node(T1 k, T2 v, Node left, Node right) {
     this.key = k;
     this.value = v;
     this.left = left;
     this.right = right;
   }

   T1  key;
   T2  value;
   Node left;
   Node right;
 }

 private Node<K, V> root = null;

     ...
}
```

Wir definieren den Typ `Node<T1, T2>` als die generische Klasse der Knoten mit einem Inhalt von beliebigem Schlüsseltyp `T1` und Wertetyp `T2` sowie mit zwei Nachfolgern. `Node` wird hier als *statische innere Klasse* definiert. Eine *innere Klasse* ist eine Klasse deren Definition sich innerhalb einer anderen Klasse befindet. Eine *statische innere Klasse* wird mit dem Schlüsselwort `static` gekennzeichnet. Eine statische innere Klasse hat, wie alles andere in Java, das statisch ist, keinem Bezug zu einem Objekt. Hier besteht kein Bezug zu einem Objekt der umfassenden Klasse `SearchTreeMap`. In einer statischen inneren Klasse darf nichts objektspezifisches verwendet werden, also nichts, was in der umfassenden Klasse nicht ebenfalls statisch ist. Generische Parameter gelten dabei auch als objektspezifisch.

> Eine **statische innere Klasse** ist eine Klasse, die innerhalb einer anderen Klasse definiert wurde. Sie wird mit `static` gekennzeichnet. In ihrer Definition dürfen nur statische Komponenten der umfassenden Klasse verwendet werden. Generische Parameter gelten nicht als statische Komponenten einer Klasse.

Die Definition einer statischen inneren Klasse wie `Node<T1, T2>` unterscheidet sich nur in der Sichtbarkeit von einer beliebigen anderen "normalen" Klassendefinition. Alle Mechanismen sind die gleichen, nur, dass `Node<T1, T2>` innerhalb von `SearchTreeMap` definiert ist und damit den entsprechenden Sichtbarkeitsregeln unterliegt.

Die Objektvariable `root` hat den Typ `Node<K, V>`. Dabei sind `K` und `V` die Typparameter der umfassenden Klasse. Warum gibt es hier die generischen Typparameter `K` und `V` und dazu noch die `T1` und `T2`? Bis auf die Sichtbarkeit, verhält sich eine statische innere Klasse so wie eine "normale" Klasse. Wir können darum innerhalb von Node die generischen Parameter `K` und `V` *nicht* verwenden. Das macht man sich leicht klar, wenn man sich vorstellt, die Definition von Node würde sich in einer anderen, eigenen Datei befinden.

`K` und `V` stehen für den Typ der Schlüssel und Werte in einem Exemplar einer `SearchTreeMap`. Natürlich haben alle Schlüssel und Werte der Knoten in einem Exemplar einer Abbildung die Typen die zu dieser Abbildung gehören. Es kann aber beliebig viele Abbildungen geben und jede Abbildung hat beliebig viele Knoten. Nur jeweils die Knoten, die zu einer bestimmten Abbildung gehören, haben die gleichen Typen wie diese. Dieser Bezug zwischen `K` und `V` auf der einen Seite und `T1` und `T2` auf der anderen Seite kann mit einer *statischen* inneren Klasse nicht ausgedrückt werden.

Innere Klasse

Eine innere Klasse, die nicht statisch ist, hat einen Bezug zu einem Objekt der Klasse, innerhalb derer sie definiert wurde. Diesen Bezug können wir nutzen, um zum Ausdruck zu bringen, dass alle Knoten, die zu einer bestimmten Abbildung gehören, mit den gleichen Typen arbeiten wie diese:

```
public final class SearchTreeMap<K extends Comparable<K>, V> implements Map<K, V>{
```

```
 private final class Node {// innere Klasse ohne static
   Node(K k, V v) { // K und V koennen verwendet werden
     this.key = k;
     this.value = v;
   }

   Node(K k, V v, Node left, Node right) {
     this.key = k;
     this.value = v;
     this.left = left;
     this.right = right;
   }

   K  key;
   V  value;
   Node left;
   Node right;
 }

 private Node root = null;
     ....
}
```

Mit dieser Definition wird zum Ausdruck gebracht, dass alle Exemplare von `Node` die zu einem Exemplar von `SearchTreeMap` gehören, mit den gleichen Typen `K` und `V` arbeiten wie diese. Bei einer inneren Klasse gibt es einen Bezug zu dem erzeugenden Objekt der umfassenden Klasse. `Node` steht also in Verbindung zu einem Exemplar der Klasse `SearchTreeMap`. Für dieses Exemplar sind `K` und `V` an bestimmte Typen gebunden und können darum jetzt innerhalb von `Node` auch `K` und `V` verwendet werden.

> Eine **innere Klasse** ist eine Klasse, die ohne das Schlüsselwort `static` innerhalb einer anderen Klasse oder einer Methode definiert wurde. In ihrer Definition dürfen alle Komponenten der umfassenden Klasse und alle finalen Komponenten einer eventuell umfassenden Methode verwendet werden. Jedes Exemplar einer inneren Klasse ist an das erzeugende Exemplar der umfassenden Klasse gebunden. Objektvariablen und generische Parameter sind dabei an dessen Objektvariablen und generische Parameter gebunden.

Die Verwendungsmöglichkeiten einer inneren Klassen sind vielfältiger als die einer statischen inneren Klasse. Durch die implizite Bindung an ein Objekt sind die statischen inneren Klassen etwas effizienter als die nicht–statischen.

Suchen

Die Datenstruktur *binärer Suchbaum* ist auf das Suchen von Elementen hin ausgelegt. Ein gesuchter Schlüssel kann durch Vergleich mit dem Schlüssel des aktuellen Knotens schnell und einfach gefunden werden. Er ist gleich dem Schlüssel des aktuellen Knotens, oder er ist links (kleinere Werte) oder rechts (größere Werte) zu suchen:

```
public class SearchTreeMap<K extends Comparable<K>, V> implements Map<K, V>{

 private final class Node {
         ....
 }

 private Node root = null;

 private Node find( Node n, K k ) {
   while ( n != null ) {
     if ( n.key.compareTo(k) == 0 ) return n;
     if ( n.key.compareTo(k) < 0 )
       if ( n.right != null ) n = n.right; else return null;
     if ( n.key.compareTo(k) > 0 )
       if ( n.left != null ) n = n.left; else return null;
   }
   return n;
 }
```

```
 /*
  * prueft ob der Schluessel arg0 enthalten ist (siehe API-Doku zu Map) @pre
  * arg0 != null @returns arg0 hat einen zugeordneten Wert @throws
  * ClassCastException : Typ von arg passt nicht zu K @throws
  * NullPointerException : arg0 == null
  */
 @SuppressWarnings("unchecked")
 public boolean containsKey(Object arg0) {
    if ( arg0 == null ) // null ist nicht erlaubt
      throw new NullPointerException();
    return find( root, (K) arg0 ) != null;
 }


 /*
  * liefert Wert zu Schluessel arg0
  * (siehe API-Doku zu Map)
  * @pre arg0 != null
  * @returns den arg0 zugeordneten Wert oder null
  * @throws ClassCastException : Typ von arg passt nicht zu K
  * @throws NullPointerException : arg0 == null
  */
 @SuppressWarnings("unchecked")
 public V get(Object arg0) {
    if ( arg0 == null ) // null ist nicht erlaubt
      throw new NullPointerException();
    Node n = find( root, (K) arg0 );
    if ( n == null ) return null;
    return n.value;
 }

     ...
}
```

Die Funktionalität der Methoden richtet sich nach den Vorgaben in `java.util.Map`. Der Deutlichkeit halber haben wir sie hier noch einmal kommentiert. Das ist eigentlich überflüssig. Die Beschreibung in der API–Dokumentation ist ausführlich und die Implementierung darf nicht von dem dort Beschriebenen abweichen.

Einfügen

Das Einfügen eines Elementes ist auch noch relativ einfach. Wir nutzen eine Hilfsfunktion `insert`:

```
public final class SearchTreeMap<K extends Comparable<K>, V> implements Map<K, V>{

 private class Node {
          ....
 }

 private Node root = null;

 private Node find( Node n, K k ) {
    ....
 }

 private Node insert(Node n, K k, V v) {
    if (n == null) {
      n = new Node(k, v);
    } else {
      if (k.compareTo(n.key) > 0) {
        n.right = insert(n.right, k, v);
      } else {
        n.left = insert(n.left, k, v);
      }
    }
    return n;
 }
```

```
  /*
   * siehe API-Doku zu Map
   */
  public V put(K arg0, V arg1) {
    V previous = null;
    if ( arg0 == null )
      throw new NullPointerException();
    Node n = find( root, arg0 );
    if ( n != null ) {
      previous = n.value;
      n.value = arg1;
    } else {
      root = insert( root, arg0, arg1 );
    }
    return previous;
  }

  ....
}
```

Entfernen

Das Entfernen eines Elementes ist die komplizierteste Operation. Ist der zu entfernende Knoten ein Blatt, dann kann er einfach gelöscht werden. Hat er nur ein Kind, dann kann er durch dieses Kind ersetzt werden. Hat er zwei Kinder, dann wird er durch durch den Nachkommen mit den kleinsten unter den größeren Nachkommen ersetzt. Der Nachkomme mit dem kleinsten der größeren Schlüssel ist der linkeste Nachkomme seines rechten Kindes:

```
public final class SearchTreeMap<K extends Comparable<K>, V> implements Map<K, V>{

  private class Node {
    ...
  }

  private Node root = null;

  private Node find( Node n, K k ) {
    ...
  }

  private Node insert(Node n, K k, V v) {
    ...
  }


  private Node delete(Node parent, Node n) {
    if (n.left == null && n.right == null)
      return null;
    if (n.left == null)
      return n.right;
    if (n.right == null)
      return n.left;
    // wir suchen den rechtesten Knoten im linken Teilbaum
    // und setzen diesen statt dem Knoten ein
    Node vorrechtester = null;
    Node rechtester = n.left;
    while (rechtester.right != null) {
      vorrechtester = rechtester;
      rechtester = rechtester.right;
    }
    if (vorrechtester != null){
      // linken Teilbaum des rechtesten an dessen Pos. setzen
      vorrechtester.right = rechtester.left;
      rechtester.left = n.left;
    }
    rechtester.right = n.right;
```

```
    return rechtester;

  }

      ...

  /*
   * siehe API-Doku zu Map
   */
  @SuppressWarnings("unchecked")
  public V remove(Object arg0) {
    V previous = null;
    if ( arg0 == null )
      throw new NullPointerException();
    Node n = find( root, (K) arg0 );
    if ( n != null ) {
      previous = n.value;
      if ( n == root ) root = delete(n, root);
      else {
        // Suche Vorgaenger
        Node parent = root;
        while ( true ) {
          if ( parent.left == n || parent.right == n ) break;
          else if ( n.key.compareTo(parent.key) < 0 )
                                    parent = parent.left;
          else if ( n.key.compareTo(parent.key) > 0 )
                                    parent = parent.right;
          else throw new RuntimeException("Programmfehler");
        }
        if ( n == parent.left )
                              parent.left = delete(n, parent.left);
        if ( n == parent.right )
                              parent.right = delete(n, parent.right);
      }
    }
    return previous;
  }

  ....
}
```

Kapitel 4

Objektorientierung II: Vererbung und Polymorphismus

4.1 Vererbung

4.1.1 Basisklasse und Abgeleitete Klasse

Basisklasse und Abgeleitete Klasse: Art und Unterart

Die Unterscheidung zwischen *Objekt* und *Klasse* teilt die Welt auf in Objekte (konkrete Dinge wie die Kuh *Elsa*) und Klassen (Konzepte, Arten, Typen wie *Kuh*). Die Kuh *Elsa* ist ein Exemplar oder eine "Instanz" der Klasse *Kuh*.

Bei einer *Vererbung* werden Klassen miteinander in Bezug gesetzt. Alle Kühe sind Säugetiere, Elsa ist darum sowohl eine Kuh, als auch ein Säugetier. Man sagt *Säugetier* ist die *Basisklasse* und *Kuh* die *abgeleitete Klasse*, oder auch *Kuh* ist eine Ableitung von *Säugetier*. Klasse und abgeleitete Klasse entsprechen Art und Unterart. Statt von "abgeleiteter Klasse" spricht man darum gelegentlich auch von "Unterklasse".

Mit einer Art will man das Gemeinsame ihrer Unterarten zum Ausdruck bringen. Kühe, Katzen, Menschen und Affen haben Gemeinsamkeiten, die durch das Konzept "Säugetier" zum Ausdruck gebracht werden. Umgekehrt unterscheiden sie sich aber trotzdem noch so weit, dass es sinnvoll ist weiterhin von Kühen, Katzen, Menschen und Affen zu sprechen. Es geht also um die Unterschiede und die Gemeinsamkeiten von Objekten unterschiedlicher Klassen, wenn diese als Basis– und abgeleitete Klasse in Beziehung gesetzt werden.

Vererbung: Spezielles OO–Verständnis von Art und Unterart

Die Analogien zwischen der realen Welt und der Welt der Programmierung helfen oft beim Verständnis abstrakter programmiertechnischer Konzepte. Man muss jedoch aufpassen, dass man sie nicht zu weit treibt. So hat die objektorientierte Programmierung ihr eigenes spezielles Verständnis von Art und Unterart als Klasse und Unterklasse.

Eine Unterart (Unterklasse) im OO–Verständnis repräsentiert immer Exemplare die all das haben und all das können, was Objekte der Art haben und können und eventuell noch mehr. Dieses Konzept nennt sich *Vererbung*. Die Unterklasse "erbt" (d.h. sie übernimmt) alle Eigenschaften und Fähigkeiten der Oberklasse. Bei einer Vererbung stirbt also niemand, es geht um ein Konzept der Übernahme und Erweiterung, also um ein spezielles Verständnis der Beziehungen von Art und Unterart.

Vererbung entspricht durchaus sehr oft dem intuitiven Verständnis von Art und Unterart. Ein Schäferhund kann alles und hat alles was ein Hund so kann und hat. Art und Unterart können aber auch abweichend vom Konzept der Vererbung verstanden werden. Ein schönes Beispiel sind *Frisöre* und *Herrenfrisöre*. Ein Herrenfrisör ist eine spezielle Art von Frisör. Das Spezielle besteht aber nicht (!) darin, dass ein Herrenfrisör all das kann, was ein Frisör kann und noch mehr, sondern – genau im Gegenteil – er kann weniger. Er ist auf Herrn spezialisiert und kann Damen nicht bedienen.

Frisöre und *Herrenfrisöre* sind darum "natürlich" eine Art und eine Unterart, aber nicht im Sinne der Objektorientierung. Sie können *nicht* als Klasse und Unterklasse modelliert werden, denn sie stehen *nicht* in einer Vererbungsrelation!

Ursprünglich haben die Evangelisten der Objektorientierung diese Fälle von Art und Unterart, die keine Vererbung darstellen, unter den Teppich gekehrt. Sie wollten keine Zweifel an der "Natürlichkeit" ihrer Konzepte aufkommen lassen. Inzwischen glauben alle an die Objektorientierung und propagandistische Vereinfachungen können fallen gelassen werden.

Tiere, Kühe und Tiger: Basisklasse und Abgeleitete Klassen

Die Objekte der Klassen *Kuh* und *Tiger* haben eine Gemeinsamkeit, es sind *Tiere*. Jede *Kuh* und jeder *Tiger* ist *auch* ein *Tier*. Dies kann in Java definiert werden:

```
abstract class Tier { ... } // Basisklasse definieren

class Kuh extends Tier { ... } // abgeleitete Klasse definieren
class Tiger extends Tier { ... }
```

`Tier` ist die *Basisklasse*. `Kuh` und `Tiger` sind *abgeleitete Klassen*. Damit wird gesagt, dass jedes Objekt der Klassen `Kuh` und `Tiger` auch ein Objekt der Klasse `Tier` ist.

Abstrakte Basisklasse: reines Konzept

Es gibt kein Objekt das nur ein Tier ist, ohne gleichzeitig eine Kuh, ein Tiger, eine Maus, etc. zu sein, Die Basisklasse *Tier* ist darum ein reines Konzepte. In Java machen wir dies dadurch kenntlich, dass `Tier` als *abstrakte Klasse* deklariert wird.

Abstrakte Basisklassen können nicht dazu genutzt werden um Exemplare zu erzeugen. Ein Verwendung mit new ist darum nicht erlaubt.

```
Tier t = new Tier(); // FEHLER: geht nicht, Tier ist abstrakt
```

Nicht–abstrakte Basisklasse: Grundversion

Basisklassen und ihre Ableitungen sind jedoch nicht auf die Konstellation beschränkt, bei der Ableitungen in einer Basisklasse als übergeordnetem abstrakten Konzept zusammengefasst werden. Die Basisklasse kann durchaus auch der Typ realer Objekte sein. Beispielsweise kann *Angestellte* eine Klasse mit vielen realen Objekten sein: Fräulein Meier, Herr Schulze, etc. sind *Angestellte*. Manche *Angestellte* gehören zum Management und sind darum *Angestellte* und *Manager*. Damit haben wir eine Basisklasse *Angestellte* und eine abgeleitete Klasse, die beide der Typ realer Objekte sind. Die Basisklasse repräsentiert in solchen Fällen die "Grundversion" eines Konzeptes.

Verallgemeinerung und Spezialisierung

`Tier` ist die Basisklasse, von ihr sind die Klassen `Kuh` und `Tiger` abgeleitet. Die Beziehung zwischen den Klassen kann man in zwei Richtungen sehen:

- *Verallgemeinerung:* Die Basisklasse `Tier` ist eine Verallgemeinerung ihrer Ableitungen `Kuh` und `Tiger`.
- *Spezialisierung:* Die Ableitung `Manager` ist eine Spezialisierung der Basisklasse `Angestellte`.

Basisklassen flexibilisieren das Typsystem

Ein Tier kann eine Kuh oder ein Tiger sein. Dies gilt auch für Variablen und Parameter mit diesen Typen:

```
Tier t;
Kuh  berta = new Kuh(...);
Tiger holger = new Tiger(...);
t = new Kuh(...); // OK
t = new Tiger(...); // OK
t = berta; // OK
t = holger; // OK
berta = t; // FEHLER: Ein Tier muss nicht unbedingt eine Kuh sein
holger = t; // FEHLER: Ein Tier muss nicht unbedingt ein Tiger sein
```

Man kann also eine Variable vom Basistyp `Tier` definieren und sie mit Werten eines abgeleiteten Typs belegen; aber nicht umgekehrt. Das passt zum Konzept der Vererbung: Immer wenn ein Tier benötigt wird, kann man einen Tiger verwenden, aber wenn ein Tiger verlangt wird, kann man nicht mit einem x–beliebigen Tier kommen.

Den Java–Compiler interessieren bei der Prüfung der Zuweisung nur die Typen der Variablen. Ihr aktueller Inhalt ist nicht relevant:

```
Tier t = new Kuh(...); // OK
Kuh  berta = t; // Fehler !
```

Bei der Parameterübergabe gilt das Gleiche. An einen formalen Parameter vom Basistyp kann ein Objekt mit einem abgeleiteten Typ übergeben werden aber nicht umgekehrt.

```
void f (Tier t) { ... }
void h (Kuh k) { ... }

...

f(new Kuh(...)); // OK
Tier t = new Kuh(...);
h(t);          // FEHLER
```

4.1.2 Vererbung

Vererbung: Übernahme der Eigenschaften und Fähigkeiten vom Basistyp

Statt von *Ableitung* spricht man auch von *Vererbung*. Der Grund für den – in diesem Zusammenhang – etwas seltsamen Begriff liegt im Verständnis der Beziehung der Basisklasse zu ihren Ableitungen. Die Basisklasse stellt das Allgemeine dar, die Ableitung das Spezielle. Das Spezielle hat alle Eigenschaften des Allgemeinen und darüber hinaus noch eigene spezielle Eigenschaften.

Definiert man eine Klasse `B` als Ableitung einer anderen Klasse `A`, dann übernimmt – *erbt (!)* – `B` alle Eigenschaften und Fähigkeiten von `A`: alle Komponenten, Daten, oder Methoden. Beispiel:

```
class A {
  void f(int) { ... }
  int i;
};

// Ableitung              Entspricht

class B extends A { //    class B {
   float x;         //         void f(int) { ... } // von A geerbt
};                  //         int i;          // von A geerbt
                    //         float x;
                    //      };
```

Die abgeleitete Klasse `B` hat alles was die Basisklasse `A` hat. Das spart Definitionsarbeit bei `B` und macht die Klassendefinition kompakter.

Basisklasse und Interface

Das Konzept *Basisklasse und abgeleitete Klasse* hat vieles gemeinsam mit dem Konzept *Interface und implementierende Klasse*. Die Objekte einer implementierenden Klasse können an allen Stellen auftreten, an denen das Interface gefordert ist. Genau das gleiche gilt für die Beziehungen von abgeleiteter Klasse und Basisklasse. Ein Interface und eine Basisklasse sind darum gleichermaßen Repräsentanten von gemeinsamen Können und Wissen ihrer abgeleiteten bzw. implementierenden Klassen.

Basisklasse und Interface sind also eng verwandt. Es gibt aber auch Unterschiede. Die liegen im Technischen und vor allem im Konzeptionellen ("Philosophischen").[1]

Technisch gesehen "übernimmt" die Implementierung eines Interfaces lediglich die Signatur von Methoden (sowie Definition von Konstanten, Klassen und weiterer Interfaces). Eine abgeleitete Klasse übernimmt dagegen alles von ihrer Basisklasse inklusive der Implementierung von Methoden. Konzeptionell gesehen sagt ein Interface etwas über das Können von Objekten einer Klasse, die Basisklasse etwas über das Grundkonzept.

Nehmen wir ein Beispiel: Die Fähigkeit Fahrkarten ausgeben zu können ist bestens durch ein Interface repräsentiert:

```
class Ticket {...}
class Geld {...}

interface FahrkartenAusgabe {
  Ticket ausgabe(Geld preis);
}
```

Einen Zugbegleiter können wir modelieren als

```
class Zugbegleiter implements FahrkartenAusgabe {
  @Override
  public Ticket ausgabe(Geld preis) { ... }
  ...
}
```

und einen Fahrkartenautomaten als

```
class FahrkartenAutomat implements FahrkartenAusgabe {
  @Override
  public Ticket ausgabe(Geld preis) { ... }
```

[1] Anfänger, in ihrem Bemühen das Klappen der Technik zu verstehen, unterschätzen oft die Bedeutung der "philosophischen" Konzepte.

```
  ...
}
```

Die Fahrkartenausgabe wird von beiden ausgeführt. Da wo es nur um die Fahrkartenausgabe geht, kann der eine den anderen ersetzen. Die Implementierungen der Ausgabe–Methoden werden komplett unterschiedlich sein, aber vom Standpunkt des Benutzers aus gesehen ist das unerheblich.

Zugbegleiter als Ableitung einer Basisklasse *FahrkartenAusgabe* definieren zu wollen, ist dagegen völlig abwegig.[2] Es würde bedeuten, dass Zugbegleiter und Fahrkartenautomaten jeweils Varianten, Sonderfälle eines gemeinsamen Grundwesens sind.

```
class FahrkartenAusgabe {
  public Ticket ausgabe(Geld preis) { ... }
}

// Abwegiger Missbrauch des Ableitungskonzepts
class Zugbegeleiter extends FahrkartenAusgabe {
  ...
}

// Abwegiger Missbrauch des Ableitungskonzepts
class FahrkartenAutomat extends FahrkartenAusgabe {
  ...
}
```

Bei einem Interface liegt das Augenmerk auf der *Verwendung* der Objekte. Bei einer Basisklasse liegt dagegen das Augenmerk auf den *"inneren" Gemeinsamkeiten*. Objekte unterschiedlicher implementierender Klassen habe die gleiche (Bedien–) Oberfläche. Objekte unterschiedlicher abgeleiteter Klassen haben die gleichen Bestandteile.

> Ein **Interface** drückt Gemeinsamkeiten der Bedienung von Objekten unterschiedlicher Klassen aus. Ein **Basisklasse** drückt Gemeinsamkeiten des Wesens von Objekten unterschiedlicher Klassen aus.

Kombination von Basisklasse und Interface

Basisklassen und Interfaces können durchaus sinnvoll kombiniert werden. Ein Beispiel reicht zur Illustration aus:

```
interface FahrkartenAusgabe {
  Ticket ausgabe(Geld preis);
}

abstract class Bahnangestellter {
  void streiken(int Dauer) { ... }
  ...
}

abstract class Automat {
  void oelen(int oelMenge) { ... }
  ...
}

class Zugbegleiter extends Bahnangestellter implements FahrkartenAusgabe {
  @Override
  public Ticket ausgabe(Geld preis) { ... Ticket mit biologischem Verfahren ausgeben ... }
}

class FahrkartenAutomat extends Automat implements FahrkartenAusgabe {
  @Override
  public Ticket ausgabe(Geld preis) { ... Ticket mit mechanischem Verfahren ausgeben ... }
}
```

[2] So abwegig, dass es bestenfalls einem Bahnmanager einfallen könnte.

4.1.3 Vererbung und Initialisierungen

Initialisierungsreihenfolge

Die Initialisierungsschritte werden auf die Vererbungshierarchien ausgedehnt. Die Reihenfolge der Initialisierung von Klassen und Objekten ist:

1. Klassen werden initialisiert:
   (a) Die Basisklasse wird initialisiert:
      i. Die Basisklasse der Basisklasse wird initialisiert (Rekursion der Klasseninitialisierung)
      ii. Klassenvariablen der Basisklasse ohne Zuweisung werden initialisiert
      iii. Klassenvariablen der Basisklasse mit Zuweisung werden belegt
      iv. Der statische Initialisier der Basisklasse wird ausgeführt (falls vorhanden)
   (b) Die abgeleitete Klasse selbst wird initialisiert:
      i. Klassenvariablen ohne Zuweisung werden initialisiert
      ii. Klassenvariablen mit Zuweisung werden belegt
      iii. Der statische Initialisier der Klasse wird ausgeführt (falls vorhanden)
2. Objekte werden initialisiert:
   (a) Der Basisanteil wird initialisiert:
      i. Der Basisanteil der Basisklasse wird initialisiert (Rekursion der Objektinitialisierung)
      ii. Objektvariablen ohne Zuweisung werden initialisiert
      iii. Objektvariablen mit Zuweisung werden belegt
      iv. Der Objekt-Initialisierer wird ausgeführt (falls vorhanden)
      v. Der Default–Konstruktor wird ausgeführt
   (b) Der Anteil der abgeleiteten Klasse wird initialisiert:
      i. Objektvariablen ohne Zuweisung werden initialisiert
      ii. Objektvariablen mit Zuweisung werden belegt
      iii. Der Objekt-Initialisierer wird ausgeführt (falls vorhanden)
      iv. Ein Konstruktor wird ausgeführt

Beispiel (die Zahlen beschreiben die Reihenfolge der Initialisierungen):

```
class Basis {

  static X b1 = _1_

  X b2 = 5

  static {
    b1 = _2_
  }

  {
    b2 = _6_
  }

  Basis() {
    b2 = _7_
  }
}

class Ab extends Basis {

  static Y a1 = _3_

  Y a2 = _8_

  static {
```

```
    a1 = _4_
  }

  {
    a2 = _9_
  }

  Ab() {
    a2 = _10_
  }
}

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

Super: Einen anderen als den Defaultkonstruktor aufrufen

Soll bei der Initialisierung des Basisanteils eines Objekts ein anderer als der Default–Konstruktor aufgerufen werden, dann verwendet man das Schlüsselwort `super`. Beispiel:

```
public class Kuh {
  private String name;
  public Kuh() {
    name = "Berta"; // wenn nichts anderes gesagt wird, dann heist eine Kuh Berta
  }
  public Kuh(String name) {
    this.name = name;
  }
  public String toString () {
    return "Kuh "+name;
  }
}

public class MilchKuh extends Kuh {
  public MilchKuh() {
    super("Clara"); // wenn nichts anderes gesagt wird, dann heist eine Milchkuh Clara
  }
  public MilchKuh(String name) {
    super(name);
}
```

4.1.4 Vererbung und Typen

Klassen und Typen

Jede Definition einer Klasse und eines Interface definiert auch gleichzeitig einen Typ. Typen und Klassen hängen eng zusammen, sind aber nicht das Gleiche. So bezeichnen in

```
class C {}

int x = 5;
C  c;
```

sowohl `int` als auch `C` einen Typ, aber nur `C` bezeichnet auch eine Klasse. In

```
class C {}
C  c // c hat den TYP C
  = new C(); // erzeuge eine Instanz der KLASSE C
```

wird `C` in beiden Bedeutungen verwendet: Es wird eine Variable mit dem *Typ* `C` definiert und ein Exemplar der *Klasse* `C` erzeugt.

Statischer und dynamischer Typ

Das Konzept des *statischen Typs* und des *dynamischen Typs*[3] ist enorm hilfreich beim Verständnis von Javaprogrammen.

- **statischer Typ** Jedem Ausdruck wird vom Compiler zur Übersetzungszeit ein Typ zugewiesen: Das ist der (statische) Typ des Ausdrucks. Er ist unveränderlich.
- **dynamischer Typ** Jedem Ausdruck entspricht zur Laufzeit bei jeder Auswertung durch die JVM ein Wert. Der Typ dieses Wertes wird dynamischer Typ des Ausdrucks genannt. Der dynamische Typ kann mit jeder Auswertung wechseln.

Genau genommen hat ein Ausdruck also beliebig viele dynamische Typen. So hat `x` in

```
void f(C x) { ... x ... }
```

genau einen statischen Typ: `C`. Bei einem Aufruf von `f` könnte aber ein Exemplar einer anderen Ableitung von `C` übergeben werden. Der dynamische Typ würde dann mit jedem Aufruf wechseln. Man spricht darum von dem dynamischen Typ den ein Ausdruck *zu einem bestimmten Zeitpunkt hat*. Jeder dynamische Typ eines Ausdrucks ist kompatibel mit seinem statischen Typ. Idealerweise ist der statische Typ eines Ausdrucks so etwas wie der "kleinste gemeinsame Nenner" all seiner möglichen dynamischen Typen.

Typprüfung

Statische und dynamische Typen werden geprüft. Erst prüft der Compiler, dann werden zur Laufzeit von der JVM gelegentlich auch noch die dynamischen Typen geprüft. Beispiel:

```
class Kuh {...}
class MilchKuh extends Kuh {...}

Kuh    k = new Kuh();
MilchKuh mk = new MilchKuh();
k = mk;         //OK
mk = k;         //Compilerpruefung: Fehler
mk = (MilchKuh)k;  //Compilerpruefung: OK; Laufzeitpruefung: Fehler
k = (Kuh)mk;    //Compilerpruefung: OK;
```

Der Compiler ist stets konservativ und lehnt zunächst einmal alles ab, dessen (Typ–) Korrektheit nicht aus den *statischen* Typen der beteiligten Variablen und Ausdrücken hervorgeht. So wird die Zuweisung an `mk` in

```
Kuh    k = new MichKuh();
MilchKuh mk = k; //Compilerpruefung: Fehler
```

nicht akzeptiert: der statische Typ von `k` garantiert nicht, dass die Zuweisung ausgeführt werden kann. Ist der Compiler allerdings auf Grund der statischen Typen davon überzeugt, dass eine Operation ohne Typfehler ausgeführt werden kann, dann finden keine Laufzeitprüfungen mehr statt.

```
MilchKuh mk = new MilchKuh();
Kuh    k = mk; //Compilerpruefung: OK, keine Laufzeitpruefung.
```

Casts

Wenn ein Programmierer glaubt, es besser zu wissen als der Compiler, dann kann sie oder er mit einem *Cast* die Compiler–Prüfung außer Kraft setzen:

```
Kuh    k = new MichKuh();
MilchKuh mk = (MilchKuh) k; // Cast; Compilerpruefung: OK; Laufzeitpruefung: OK
```

Nach dem Motto "Vertrauen ist gut, Kontrolle ist besser" findet dann aber ein Laufzeitprüfung statt. So wird

```
Kuh    k = new Kuh();
MilchKuh mk = (MilchKuh) k; // Cast; Compilerpruefung: OK; Laufzeitpruefung: Fehler
```

mit einer *IllegalCastException* abgebrochen werden.

[3] "Dynamischer Typ", also der "Typ eines Wertes" ist keine offizielle Java–Terminologie, aber sinnvoll und oft verwendet.

Typanpassungen

Ein Cast ist ein Beispiel für eine Typanpassung. Bei einer Typanpassung wird der statische Typ eines Ausdrucks verändert. Manche Typanpassungen werden automatisch vom Compiler vorgenommen, manche nur auf Initiative des Programmierers. Bei einer Typanpassung kann der Wert des Ausdrucks unverändert bleiben, oder die Typanpassung erfordert auch eine *Konversion*, eine Anpassung des Wertes.

Die Typanpassungen können wie folgt zusammengefasst werden:

- Typanpassung mit Konversion, ohne Cast: *widening* auf primitiven Typen
- Typanpassung mit Konversion, mit Cast: *narrowing* auf primitiven Typen
- Typanpassung ohne Konversion, ohne Typprüfung zur Laufzeit: *Upcast*, *widening* auf Klassen–Typen
- Typanpassung ohne Konversion, mit Cast, mit Typprüfung zur Laufzeit: *Downcast*, *narrowing* auf Klassen–Typen

Beispiel sind:

```
int i = 5;
double d = i; // widening, mit Konversion, ohne Cast
i = (int)d; // narrowing, mit Konversion, mit Cast

Kuh k = new MilchKuh(); // upcast, ohne Konversion, ohne Cast, ohne Laufzeitpruefung
MilchKuh mk = (MilchKuh) k; // downcast, ohne Konversion, mit Cast, mit Laufzeitpruefung
```

Bei einem *widening* wird ein Wert eines "engeren" auf einen "weiteren" Typ konvertiert. Beim *narrowing* geht die Konversion in die umgekehrter Richtung. Ein Typ ist dabei "weiter" als ein anderer, wenn seine Werte ohne Informationsverlust in einen Wert des anderen Typs konvertiert werden können. Z.B. gilt:

$$\text{byte} < \text{short} < \text{int} < \text{long} < \text{float} < \text{double}$$

Ein *upcast* geht in der Typhierarchie "nach oben", d.h. von abgeleiteten zu Basistypen. Ein *downcast* geht in die umgekehrte Richtung.

Ein *widening* ist immer erlaubt und wird vom Compiler automatisch eingefügt. Bezieht es sich auf primitive Typen, dann erzeugt der Compiler automatisch den notwendigen Konversionscode. Eine Laufzeitprüfung der Typen findet nicht statt.

Ein *widening* ohne Cast erzeugt eine Fehlermeldung des Compilers.

Ein *widening* mit Cast auf primitiven Typen veranlasst den Compiler den notwendigen Konversionscode zu erzeugen. Ein *widening* mit Cast auf Klassen–Typen veranlasst den Compiler Typprüfungen zur Laufzeit zu erzeugen.

Subtyp–Relation

Wenn ein Ausdruck mit Typ T_2 an allen Stellen auftreten kann, an denen ein Ausdruck vom Typ T_1 gefordert ist, dann sagt man T_2 ist ein Subtyp von T_1 oder $T_2 < T_1$. Beispiele sind:

$\text{int} < \text{long}$
$\text{Tiger} < \text{Tier}$
$\text{Milchkuh} < \text{Kuh}$

Engere Type sind Subtypen von weiteren Typen. Abgeleitete Typen sind Subtypen von Basistypen.

Java ist statisch typisiert und typsicher

Java ist eine statisch typisierte Sprache. D.h. alle Ausdrücke in einem Programm haben entweder einen eindeutig definierten statischen Typ oder sie werden vom Compiler abgelehnt. Der Compiler akzeptiert zunächst einmal alle Zuweisungen oder Parameterübergaben die garantiert (d.h. auf Basis der statischen Typen) korrekt sind. Er lässt sich durch Casts überreden Zuweisungen oder Parameterübergaben zu akzeptieren die aus seiner Sicht nur korrekt sein könnten, fügt dann aber Laufzeitprüfungen ein.

Das sorgt insgesamt dafür, dass Java *typsicher* ist: Entweder es passieren bei der Ausführung keine Typfehler oder – wenn doch – dann stürzt das Programm ab. Typfehler bleiben also niemals unbemerkt.

Beispiel mit Typfehlern:

```
static void f1(MilchKuh mk) { ... }
```

```
static void f2(Kuh k){ ... }

static Kuh g1() {
  return new MilchKuh();
}

static Kuh g2() {
  return new Kuh();
}

public static void main(String[] args){
  Kuh elke = new MilchKuh("Elke");
  f1(elke); // statischer Typfehler
  f2(elke);
  Kuh     k1 = g1();
  MilchKuh mk1 = g1(); // statischer Typfehler
  Kuh     k2 = g2();
  MilchKuh mk2 = g2(); // statischer Typfehler
  Kuh     k3 = (Kuh)g1(); // Cast ueberfluessig
  MilchKuh mk3 = (MilchKuh)g2();
}
```

Mit eingefügten Casts an den Stellen, an denen der Compiler sich beschwert, erhalten wir ein übersetzbares Programm (ohne die überflüssigen Casts):

```
static void f1(MilchKuh mk) { ... }

static void f2(Kuh k){ ... }

static Kuh g1() {
  return new MilchKuh();
}

static Kuh g2() {
  return new Kuh();
}

public static void main(String[] args){
  Kuh elke = new MilchKuh("Elke");
  f1( (MilchKuh)elke );
  f2(elke);
  Kuh     k1 = g1();
  MilchKuh mk1 = (MilchKuh)g1();
  Kuh     k2 = g2();
  MilchKuh mk2 = (MilchKuh)g2(); // Laufzeitfehler
  Kuh     k3 = /*(Kuh)*/ g1();
  MilchKuh mk3 = (MilchKuh)g2(); // Laufzeitfehler
}
```

das jetzt allerdings zwei Typfehler enthält, die zur Laufzeit auftreten.

4.1.5 Subtyp–Relation bei strukturierten Typen

Subtyp–Relation und Felder

Die Subtyp–Relation überträgt sich auf Felder. D.h. wenn $S < T$ dann gilt auch $S[] < T[]$. Beispielsweise ist `int[]` ein Subtyp von `long[]` und `Kuh[]` ein Subtyp von `Tier[]`. Die Subtyprelation erlaubt so die Parameterübergabe in folgendem Beispiel:

```
static void fuetterAlle(Kuh[] l){
  for ( Kuh k : l )
    k.frisst(new Futter());
}

public static void main(String[] args){
  Kuh[] stall_1 = new Kuh[10];

  fuetterAlle(stall_1); //OK: fuetterAlle nimmt Kuh[] (klar!)
```

```
 MilchKuh[] stall_2 = new MilchKuh[10];

 fuetterAlle(stall_2); //OK: fuetterAlle nimmt MilchKuh[] (Typanpassung: MilchKuh[] < Kuh[])
}
```

Subtyp–Relation und Kollektionstypen

Die Subtyp–Relation übertragt sich *nicht* auf Kollektionstypen. Gilt $S < T$ dann gilt weder $List < S > < List < S >$ noch $List < T > < List < T >$. Die Parameterübergabe in folgendem Beispiel ist darum nicht erlaubt:

```
static void fuetterAlle(List<Kuh> l){
 for ( Kuh k : l )
  k.frisst(new Futter());
}

public static void main(String[] args){
 List<Kuh> stall_1 = new ArrayList<Kuh>();

 fuetterAlle(stall_1); //OK

 List<MilchKuh> stall_2 = new ArrayList<MilchKuh>();

 fuetterAlle(stall_2); //FEHLER: fuetterAlle nimmt NICHT List<MilchKuh> (KEINE Typanpassung)!
}
```

Typmarkierung und Laufzeitprüfungen bei Feldern

Es fällt auf, dass Felder anders behandelt werden als Kollektionstypen. Die Übertragung der Subtyprelation bei Feldern macht den Umgang mit ihnen leichter. Überall dort wo ein Feld von Objekten eines bestimmten Typs erwartet wird, kann ein Feld mit Elementen von einem Subtyp des erwarteten Typs verwendet werden. Damit daraus keine gefährliche Situation entstehen kann, werden Felder mit dem Typ ihrer Elemente markiert und die Typen zur Laufzeit überwacht.

Die Konsequenz sehen wir an folgendem Beispiel. Es enthält keinen statischen Typfehler führt aber zu einer *ArrayStoreException*:

```
static void fuetterAlle(Tier[] tiere){
 for (Tier t : tiere)
     t.frisst(new Futter());
 tier[0] = new Tiger(); // ArrayStoreException: Markierung "KUH" passt nicht zu Tiger
}

public static void main(String[] args){
 Kuh[] stall = new Kuh[2]; // das Feld wird mit einer Markierung "KUH" versehen
 stall[0] = new Kuh();
 stall[1] = new Kuh();

 fuetterAlle(stall); // OK: Subtyp-Eigenschaft uebertraegt sich

 Bauer bert = new Bauer();

 for (Kuh k : stall)
  bert.melke(k); // Oh je, gluecklicherweise hat die JVM den Tiger vorher entdeckt!
}
```

Wegen der Übertragung der Subtyp–Eigenschaft ist das Programm für den Compiler fehlerfrei. Damit aber dem armen Bauern beim Melken der (vermeintlichen) Kühe (unter die sich ein Tiger gemischt hat) nichts passiert, wird zur Laufzeit der Typ der Feldelemente überwacht und das "Einschmuggeln" eines Tigers in die Kuhherde verhindert. Der Versuch führt zu der *ArrayStoreException*.

Die Überwachung des Typs der Feldelemente ist aufwendig und entspricht nicht der Philosophie von Java nach der nur Programme mit expliziten Casts eine Typprüfung zur Laufzeit benötigen. Bei den "moderneren" Kollektionstypen wurde darum auf eine Übertragung der Subtypeigenschaft verzichtet.[4]

[4] Mit *wildcard extend* (siehe unten) kann man jedoch ein Verhalten erreichen das typsicher ist und keine Laufzeitprüfungen benötigt.

4.1.6 Ableiten: Übernehmen, Erweitern, Überdecken oder Überladen

Übernehmen und Erweitern

Führt eine abgeleitete Klasse unter einem neuen Namen eine Komponente ein, sei es ein Attribut (d.h. eine Objekt– oder eine Klassenvariable) oder eine Methode, dann werden die Bestandteile der Basisklasse übernommen und um die neue Komponente erweitert. – Das ist das Grundprinzip der Vererbung.

Bei der Übernahme bleiben die in der Basisklasse deklarierten Sichtbarkeiten erhalten. Das ist speziell bei privaten Komponenten beachtenswert. So enthält in folgendem Beispiel jedes Objekt der abgeleiteten Klasse eine Objektvariable `p`, von Methoden der abgeleiteten Klasse kann aber nicht darauf zugegriffen werden.

```
class Basis {
  private int p; // p ist nur in Methoden von Basis zugreifbar
  void mBasis() { p = 5; }
}

class Ab extends Basis {
  // p ist in jeder Instanz von Ab vorhanden
  String s;
  void mAb() {
    p = 5;  // Verboten: vorhanden aber nicht zugreifbar
    mBasis(); // modifiziert die Objektvariable p
  }
}
```

Erzeugt man ein Exemplar von Ab:

```
Ab ab = new Ab();
```

dann enthält diese zwei Objektvariablen (`p` und `s`) und zwei Methoden (`mBasis` und `mAb`).

Gelegentlich sagt man, dass “private Komponenten nicht vererbt werden”. Bei Erklärungen der Vererbung sollten aber die Verben “erben” und “vererben” möglichst nicht verwendet werden![5] Sprechen wir besser darüber, ob eine Komponente in einer abgeleiteten Klasse und / oder ihren Exemplaren vorhanden ist und in wie weit sie für wen sichtbar (zugreifbar) ist.

> **Erweitern** Führen abgeleitete Klassen unter neuem Namen eine neue Komponente (Attribut oder Methode) ein, dann enthalten Exemplare der abgeleiteten Klasse alle Komponenten der Basisklasse und diese neue Komponente.
> Der statische Typ und die definierten Sichtbarkeiten entscheiden darüber, ob ein Zugriff möglich ist.

Überdecken bei Attributen

Bei statisch typisierten Sprachen wie Java ist der *statische Typ* der Variablen oder eines Ausdrucks von entscheidender Bedeutung. Im folgenden Beispiel wird zweimal in der selben Art auf das selbe Objekt zugegriffen, Das Ergebnis ist aber unterschiedlich:

```
class Basis {
  String gruss = "Hallo! Ich bin ein Basis-Objekt";
}

class Abgeleitet extends Basis {
  String gruss = "Hallo! Ich bin ein abgeleitetes Objekt";
}

public class Main {

  public static void main(String[] args) {
    Abgeleitet a = new Abgeleitet();
    Basis b = a;

    System.out.println(a.gruss); // Ausgabe: Hallo! Ich bin ein abgeleitetes Objekt
```

[5] Ansonsten haben wir es schnell mit einer Endlos–Rekursion zu tun, die das Hirn (meist unbemerkt) zum Absturz bringt.

```
    System.out.println(b.gruss); // Ausgabe: Hallo! Ich bin ein Basis-Objekt
  }
}
```

Das mit `new` erzeugte Objekt enthält zwei (!) Objektvariablen mit dem Namen `gruss`: die eine wurde in der Basisklasse definiert und die andere in der abgeleiteten Klasse. Je nach dem statischen Typ eines Ausdrucks *e*, `Basis` oder `Abgeleitet`, bezieht sich *e*`.gruss` auf die eine oder die andere Variable. Beide sind in jedem Fall vorhanden.

Mit einem Cast kann der statische Typ manipuliert werden. Das wirkt sich auf den Zugriff aus:

```
public class Main {

  public static void main(String[] args) {
    Abgeleitet a = new Abgeleitet();
    Basis b = a;

    System.out.println(((Basis)a).gruss); // Ausgabe: Hallo! Ich bin ein Basis-Objekt
    System.out.println(((Abgeleitet)b).gruss); // Ausgabe: Hallo! Ich bin ein abgeleitetes Objekt
  }
}
```

> **Überdeckung:** Führen abgeleitete Klassen unter gleichem Namen ein neues Attribut(!) ein, dann enthalten Exemplare der abgeleiteten Klasse beide Attribute.
> Der statische Typ entscheidet auf welches der beiden zugegriffen wird.
> Das hinzugefügte Attribut überdeckt das bereits definierte.

super: Zeiger auf den Basisanteil

Das Schlüsselwort `this` zeigt in einer Methode auf das Objekt für das die Methode ausgeführt wird. Das Schlüsselwort `super` hat eine ähnliche Funktion: es zeigt auch auf das Objekt für das die Methode ausgeführt wird, allerdings auf dessen Basisanteil. Im folgenden Beispiel werden von `a` darum beide Grüße ausgegeben.

```
class Basis {
  String gruss = "Hallo! Ich bin ein Basis-Objekt";
}

class Abgeleitet extends Basis {

  String gruss = "Hallo! Ich bin ein abgeleitetes Objekt";

  void hallo() {
    System.out.println(this.gruss); // Hallo! Ich bin ein abgeleitetes Objekt
    System.out.println(super.gruss); // Hallo! Ich bin ein Basis-Objekt
  }
}

public class Main {

  public static void main(String[] args) {
    Abgeleitet a = new Abgeleitet();
    a.hallo();
  }
}
```

Mit `super` kann man allerdings nur bis zur nächsten Oberklasse gehen. `super.super` ist nicht erlaubt.

Überladen

Werden in einer Klasse zwei Methoden definiert, die den gleichen Namen haben, sich aber in Zahl und / oder Typ der Parameter unterscheiden, dann spricht man von *Überladung*. Das Prinzip der Überladung wird bei Java auf abgeleitete Klassen ausgedehnt. Der statische Typ ist dabei wieder von entscheidender Bedeutung. Beispiel:

```
class Futter {
  public String toString() { return "Futter"; }
```

```
}
class Gras extends Futter {
  public String toString() { return "Gras"; }
}

class Tier {
  void frisst(Futter f) {
    System.out.println("Tier-iges Fressen von "+f);
  }
}

class Kuh extends Tier {
   // Klassenuebergreifende Ueberladung
   // die Klasse Kuh hat ZWEI frisst Methoden
  void frisst(Gras g) {
    System.out.println("Kuh-iges Fressen von "+ g);
  }
}

public class Main {

  public static void main(String[] args) {
    Kuh k = new Kuh();
    Tier t = k;

    // Statischer Typ = Kuh: beide Methoden stehen zur Verfuegung
    k.frisst(new Futter()); // Tier-iges Fressen von Futter
    k.frisst(new Gras()); // Kuh-iges Fressen von Gras

    // Statischer Typ = Tier: nur eine Methode steht zur Verfuegung
    t.frisst(new Futter()); // Tier-iges Fressen von Futter
    t.frisst(new Gras()); // Tier-iges Fressen von Gras (Cast: Gras => Futter)
  }
}
```

Der statische Typ des Ausdrucks e entscheidet welche Methoden für `frisst` in `e.frisst(...)` zur Verfügung stehen. Dann kommen die statischen Parametertypen ins Spiel und am Ende werden dann eventuell noch Konversionen ausgeführt. Im Beispiel stehen für `k` zwei `frisst`–Methoden zur Auswahl. Je nach übergebenen Parameter wird eine davon ausgewählt. Bei `t.frisst` steht nur eine `frisst`–Methode zur Auswahl. Der `Gras`–Parameter wird zu `Futter` konvertiert.

> **Überladung:** Führen abgeleitete Klassen unter gleichem Namen eine neue Methode mit anderen Parametern ein, dann ist in Ausdrücken mit statischem Typ der abgeleiteten Klasse der Name überladen. Der statische Typ der Parameter entscheidet darüber, welche Methode aktiviert wird.

Überschreiben, polymorph Redefinieren

Führt eine abgeleitete Klasse unter gleichem Namen eine Methode ein, die sich auch nicht in ihren Parametern von einer vorhandenen unterscheidet, dann wird der Name *überschrieben* (man sagt auch: *(polymorph) redefiniert*).[6] Die polymorphe Redefinition ist keine Überladung und keine Überdeckung sondern etwas drittes. Bevor wir ins Detail gehen, fassen wir die Varianten des "Vererbens" erst einmal kurz zusammen:

Überladung gibt es nur bei Methoden und bedeutet, dass alle Methoden aus der Klassenhierarchie mit gleichem Namen und unterschiedlichen Parametern zur Verfügung stehen und der statische Typ des Objekts und dann der Typ der Parametertypen darüber entscheiden, welche Methode ausgewählt wird.

Überdeckung gibt es in Java nur bei Attributen und bedeutet, dass beide Attribute, überdecktes und überdeckendes, in jedem Objekt der abgeleiteten Klasse vorhanden sind. Die Definition der Basisklasse ist aber von der der abgeleiteten Klasse "abgedeckt". Der statische Typ entscheidet welches Attribut zum Zuge kommt.[7]

Überschreiben / polymorph Redefinieren bedeutet dass in Objekten der abgeleiteten Klasse die Methoden-Definition der Basisklasse entfernt und durch die der abgeleiteten Klasse ersetzt wird.

[6] Das ist nicht in jeder OO–Sprache so.

[7] Es gibt OO–Sprachen, z.B. C++, bei denen im Gegensatz zu Java auch Methoden überdeckt werden können.

Also kurz: Redefinitionen von Attributen überdecken. Redefinitionen von Methoden überschreiben. Ein Beispiel macht den Unterschied klar:

```
class Basis {
  String gruss = "Hallo! Ich bin ein Basis-Objekt";
  void hallo() {
    System.out.println("Hallo! Ich bin ein Basis-Objekt");
  }
}

class Abgeleitet extends Basis {
  String gruss = "Hallo! Ich bin ein abgeleitetes Objekt";
  void hallo() {
    System.out.println("Hallo! Ich bin ein abgeleitetes Objekt");
  }
}

public class Main {

  public static void main(String[] args) {
    Abgeleitet a = new Abgeleitet();
    Basis b = a;

    a.hallo();   // Hallo! Ich bin ein abgeleitetes Objekt
    System.out.println(a.gruss); // Hallo! Ich bin ein abgeleitetes Objekt

    // Methodenaufruf: die Methode des dynamischen Typs von b wird aktiviert
    b.hallo();   // Hallo! Ich bin ein abgeleitetes Objekt (!)

    // Attributzugriff: der statische Typ bestimmt welches Attribut verwendet wird
    System.out.println(b.gruss); // Hallo! Ich bin ein Basis-Objekt
  }
}
```

Man beachte speziell die Behandlung von `b`. Der Methodenaufruf führt trotz des statischen Typs `Basis` zu der Methode, die in der abgeleiteten Klasse definiert wurde. Der Zugriff auf das Attribut hängt dagegen vom statischen Typ ab.

Mit einem Cast kann Einfluss auf die Auswahl eines Attributs genommen werden. Bei Methoden ist das nicht so. Bei ihnen entscheidet der dynamische Typ des Objekts über die Methode, ob nun gecastet wird oder nicht.

```
Basis b = new Abgeleitet();

b.hallo();  // Hallo! Ich bin ein abgeleitetes Objekt
((Basis) b).hallo(); // Hallo! Ich bin ein abgeleitetes Objekt
```

Die in `Abgeleitet` definierte Methode `hallo` erweitert Objekte vom Typ `Abgeleitet` nicht, sondern ersetzt in deren Basisanteil die `hallo`–Methode.

`Basis`–Objekte haben folgende Struktur:

gruss	`Hallo! Ich bin ein Basis-Objekt`
hallo	`System.out.println("Hallo! Ich bin ein Basis-Objekt");`

Objekte vom Typ `Abgeleitet` haben diese Struktur:

gruss	`Hallo! Ich bin ein Basis-Objekt`	*Basis–Anteil*
hallo	`System.out.println("Hallo! Ich bin ein abgeleitetes Objekt");`	
gruss	`Hallo! Ich bin ein abgeleitetes Objekt`	*Abgeleitet–Anteil*

Das Überdeckungs–Verfahren wird in allen statisch typisierten OO–Sprachen auf Attribute angewendet. Das Verfahren des Überschreibens wird in Java bei Methoden immer angewendet. Manche Sprachen lassen den Programmierer zwischen Überschreiben und Überdeckung wählen.

Überschreiben / polymorphe Redefinition: Führen abgeleitete Klassen unter gleichem Namen und mit gleichen Parametern eine neue Methode ein, dann ist der Name im Basisanteil aller Objekte der abgeleiteten Klasse redefiniert. D.h. die neu definierte Methode wird unabhängig vom statischen Typ verwendet.

Zusammenfassung

Fassen wir die Varianten des "Erbens" und "Vererbens" noch einmal kurz zusammen:

- **Übernehmen** Die Subklasse enthält das Gleiche unter dem gleichem Namen wie die der Oberklasse.
- **Erweitern** Die Subklasse enthält unter neuem Namen etwas Neues.
- **Überdecken** (Nur Attribute) Unter gleichem Namen führt die Subklasse etwas Zusätzliches ein, das alte existiert, ist aber abgedeckt.
- **Überladen** (Nur Methoden) Unter gleichem Namen mit unterschiedlichen Parametertypen enthält die Subklasse etwas Anderes. Die Definition der Oberklasse wird dabei nicht ersetzt.
- **Überscheiben, polymorph Redefinieren** (Nur Methoden) Unter gleichem Namen mit gleichen Parametertypen enthält die Subklasse etwas Anderes. Die Definition der Oberklasse wird dabei ersetzt.

Namensauflösung

Die interessanteste Frage in Zusammenhang mit Vererbung ist die Frage, was mit x in

```
  o . x ...
```

gemeint ist. Man spricht von Namensauflösung. Der Name x wird hier "aufgelöst". Man könnte auch sagen, das Rätsel "Was ist mit x gemeint?", wird gelöst. Der Algorithmus der Namensauflösung ist:

1. Statischen Typ T_S des Objekts (o) feststellen. Der statische Typ des Objekts legt die Klasse fest, in der die Suche nach der Methode oder dem Attribut x beginnt.
2. Ist x ein Attribut dann ist mit x das erste Attribut mit Namen x gemeint, das in Klassenhierachie ab T_S aufwärts zu finden ist. Die Suche nach der Definition von x ist damit beendet.
3. Ist x eine Methode, dann suche in der Klassenhierarchie ab T_S aufwärts nach der ersten Methodendefinition mit Namen x, deren Parameter zu den statischen Typen der aktuellen Argumente – eventuell mit Konversionen – passen. Angenommen wir finden ein passendes x in der Klasse T_x. T_x enthält also eine Definition von x und liegt in der Klassenhierarchie oberhalb von T_S (oder ist gleich T_S).
4. Dann stelle den dynamischen Typ T_D von o fest. D.h. den Typ des Objekts das aktuell durch o bezeichnet wird.
5. Suche in der Klassenhierachie von T_x abwärts in Richtung T_D nach der letzten Definition von x.
6. Diese Methode wird aktiviert.

Bei Attributen suchen wir also ausgehend vom statischen Typ von o aufwärts nach der ersten passenden Definition.

Bei Methoden suchen wir zuerst ausgehend vom statischen Typ von o aufwärts nach der ersten passenden Definition. Von dort aus suchen wir dann abwärts nach der letzten Redefinition des zuerst Gefundenen.

Ein etwas komplexeres Beispiel ist:

```
class C1 {
  void f(C1 c) {
    System.out.println("Hallo C1! Ich bin ein C1-Objekt");
  }
}

class C2 extends C1 {
  void f(C2 c) { // Ueberladung
    System.out.println("Hallo C2! Ich bin ein C2-Objekt");
  }
}

class C3 extends C2 {
  void f(C2 c) { // Redefinition
```

```
    System.out.println("Hallo C2! Ich bin ein C3-Objekt");
  }
  void f(C3 c) { // Ueberladung
    System.out.println("Hallo C3! Ich bin ein C3-Objekt");
  }
}


public class Main {

  public static void main(String[] args) {
    C2 c2 = new C3();

    // Aufruf von C1.f(C1)
    c2.f(new C1()); // Hallo C1! Ich bin ein C1-Objekt

    // Aufruf von C3.f(C2)
    c2.f(new C2()); // Hallo C2! Ich bin ein C3-Objekt

    // Aufruf von C3.f(C2)
    c2.f(new C3()); // Hallo C2! Ich bin ein C3-Objekt
  }
}
```

Im ersten Aufruf hat `c2` den statischen Typ `C2`. In `C2` ist `f` überladen. Der Parameter hat den statischen Typ `C1`. Die am besten passende Methode mit dem Namen `f` ist die in `C1` definierte. Diese Methode wird nicht polymorph redefiniert und folglich verwendet.

Im zweiten Aufruf beginnen wir wieder in `C2` mit der Suche nach `f`. Wieder ist `f` überladen. Diesmal passt aber die in `C2` definierte Methode mit Parametertyp `C2` am besten. Jetzt wird auch in `C3` eine polymorphe Redefinition gefunden und aktiviert.

Der letzte Aufruf ist vielleicht der interessanteste: `c2` hat den statischen Typ `C2`. In `C2` ist eine passende Methode definiert: `f(C2 c)`. Das Argument muss allerdings von `C3` nach `C2` konvertiert werden. Diese Methode ist dann in `C3` redefiniert und diese Redefinition wird aktiviert. Achtung: Das Objekt hat den dynamischen Typ `C3` und `f` einen Parameter vom Typ `C3`. Es wird aber nicht die in `C3` definierte Methode mit Parameter vom Typ `C3` aufgerufen sondern die in `C3` definierte Methode mit Parameter vom Typ `C2`.

Redefinition und der Ergebnistyp von Methoden

Bis zu Java 1.5 war es zwingend notwendig, dass bei der Redefiniton einer Methode die Ergebnistypen exakt übereinstimmen mussten.

```
class Basis {
  TE_1 f(TP_1 x) { ... }
}

class Abgeleitet extends Basis{
  TE_2 f(TP_2 x) { ... } // TP_2 == TP_1 => TE_2 == TE_1
}
```

Ab Java 1.5 sind *kovariante Modifikationen im Ergebnistyp* erlaubt. D.h. der Ergebnistyp von f in der Ableitung darf ein Subtyp des Ergebnistyps von `f` in der Basis sein. (Kovariant: Variant in der gleichen Richtung: Basis zu Ableitung, Ergebnistyp in der Basis zu Ergebnistyp in der Ableitung)

```
class Wolle {
  public String toString() { return "reine Wolle"; }
}
class Schafwolle extends Wolle {
  public String toString() { return "beste Schafwolle"; }
}

class Wolltier {
  Wolle scheren() { return new Wolle(); }
}

class Schaf extends Wolltier {
```

```
  // polymorphe Redefinition
  Schafwolle scheren() { return new Schafwolle(); }
}

public class Main {
  public static void main(String[] args) {
    Wolltier heinz = new Schaf();
    Wolle w = heinz.scheren(); // Aufruf der Schaf--Methode
    Schafwolle sw = heinz.scheren(); // FEHLER Typfehler
    Schafwolle sw = (Schafwolle) heinz.scheren(); // OK
    System.out.println(w);
  }
}
```

Immer noch gilt bei den Typen der Vorrang der statischen Typen. `heinz` hat den statischen Typ `Wolltier`, er liefert also statisch, d.h. für den Compiler, immer nur `Wolle`.

4.2 Generizität und Polymorphismus

4.2.1 Polymorphismus

Polymorphismus und Generizität: Mittel und Ziel

Generisch Der Begriff "generisch" leitet sich aus dem Lateinischen ab und bedeutet so viel wie "den Stamm, die Klasse, die Familie, etc. (den *genus*) betreffend". Also nichts anderes als "allgemein für eine bestimmte Gruppe gültig". Letztlich geht es wie so oft in der Informatik um Abstraktion. Etwas Generisches behandelt nicht einen konkreten Einzelfall sondern etwas Abstraktes: eine Kollektion von Individuen deren Mitglieder etwas gemeinsam haben.

Etwas Generisches ist darum immer etwas Allgemeines. In der Pharmazie nennt man *Generika* Medikamente die ähnliche Kopien von einem Original sind: Gleicher Wirkstoff, andere Produktion, andere Hilfsstoffe. Von den Hilfsstoffen und der Produktion wird abstrahiert, es bleibt das Wesentliche in Form der Generika.

Polymorph Der Begriff "polymorph" kommt aus dem Griechischen und bedeutet "vielgestaltig" (*poly* = viel *morphe* = Gestalt). Etwas Polymorphes tritt in unterschiedlicher Gestalt auf. In der Mineralogie bezeichnet man mit Polymorphie das Phänomen, dass Stoffe mit exakt gleicher chemischer Zusammensetzung in unterschiedlichen Formen auftreten können. Graphit und Diamant sind beispielsweise zwei Erscheinungsformen von Kohlenstoff.

Etwas Polymorphes kann sich ändern indem es unterschiedliche Gestalten annehmen kann, oder es ist das Gleiche in unterschiedlichen aber stabilen Erscheinungsformen.

Generizität und Polymorphismus hängen eng zusammen. Polymorphie ist ein Mechanismus mit dem Generzität erreicht werden kann. Das Polymorphe kann seine Gestalt verändern und sich so unterschiedlichen Kontexten anpassen. Es wird damit allgemein verwendbar, generisch halt. Generzität ist das angestrebte Ziel und Polymorphismus ein Mittel um dieses Ziel zu erreichen. Nicht unbedingt das einzige Mittel, aber ein sehr wichtiges.

Generics Im Kontext von Java hat sich der Name *Generics* eingebürgert. *Generics* sind generisch. Sie sind es, weil sie den Entwicklern eine bestimmte Variante des Polymorphismus zur Verfügung stellen: den parametrischen Polymorphismus. Statt von "Generics" sollte man darum besser von "parametrisch–polymorphen Klassen und Methoden" reden. Nun, ok, "Generics" ist kürzer und jeder weiß hoffentlich was gemeint ist.

Parametrischer und Vererbungs–Polymorphismus

In der Programmierung gibt es Polymorphismus, also Anpassung durch Wandlung, in zwei Varianten:

- Vererbungs– oder Schnittstellen–Polymorphismus (Vererbungsgenerizität)
- Parametrischer Polymorphismus (Parametrische Generizität)

Vererbungspolymorphismus erreicht die Anpassungsfähigkeit von *Objekten* über die *Subtyp–Relation*. Ein Objekt vom Typ T_S kann überall dort verwendet werden, wo ein Objekt vom Typ T benötigt wird, falls gilt $T_S \leq T$. Vererbung ist dabei das zentrale Mittel um Subtyp–Beziehungen zu definieren.

Parametrischer Polymorphismus erreicht die Anpassungsfähigkeit von *Klassen* und *Methoden* über explizite oder implizite Typparameter.

Beide Mechanismen haben das gleiche Ziel: Generizität. Sie können auch gelegentlich alternativ verwendet werden. Es gibt Gemeinsamkeiten aber auch einige subtile Unterschiede. Betrachten wir erst einmal Vererbungspolymorphismus am Beispiel von Hunden:

```
class Hund{ ... }
class WolfsHund extends Hund { ... }
...
Hund h = new Wolfshund();
```

Jeder *Wolfshund* kann dort auftreten, wo ein *Hund* gefordert ist. Diese Generizität kann nur mit Vererbungspolymorphismus erreicht werden.

Etwas anders sieht es bei einem Hundezwinger aus. Er kann definiert werden als

```
// Vererbungspolymorphismus
public class HundeZwinger {
  private Hund hund;

  public void sperreEin(Hund hund) {
    this.hund = hund;
  }

  public Hund befreie() {
    Hund hund = this.hund;
    this.hund = null;
    return hund;
  }
}
```

und ist damit generisch auf Grund des Vererbungspolymorphismus. Jeder *HundeZwinger* z kann dort verwendet werden, wo ein Objekt mit einem Subtyp von von *HundeZwinger* gefordert ist. Beispielsweise ist ein *WolfshundZwinger*, ein Zwinger für einen *Wolfshund* ein solcher Subtyp:

$$T_{Zwinger} \leq T_{WolfshundZwinger}$$

mit

```
// hypothetischer Zwinger fuer Wolfshunde
public class WolfshundZwinger {
  private WolfsHund wolfshund;

  public void sperreEin(WolfsHund wolfshund) {
    this.wolfshund = wolfshund;
  }

  public WolfsHund befreie() {
    WolfsHund wolfshund = this.wolfshund;
    this.wolfshund = null;
    return wolfshund;
  }
}
```

denn bei der Subtyp–Relation auf Klassen gilt Kontravarianz im Parametertyp von Methoden.[8]

$$T_{WolfsHund} \leq T_{Hund} \Rightarrow T_{HundeZwinger} \leq T_{WolfshundZwinger}$$

Ein *WolfshundZwinger* muss natürlich gar nicht erst definiert werden. Der *Hundezwinger* ist anpassungsfähig (polymorph) und darum generisch in seiner Verwendung:

```
HundeZwinger z = new HundeZwinger();
WolfsHund volker = new WolfsHund();
z.sperreEin(volker);
```

Beim Herausholen eines *Wolfshunds* muss allerdings ein Cast eingesetzt werden:

```
volker = (WolfsHund) z.befreie();
```

Warum ist das so? Nun, bei der Subtyp–Relation auf Klassen sind Methoden kovariant, nicht kontravariant, im Ergebnistyp.

Vergleichen wir den Vererbungspolymorphismus mit parametrischen Polymorphismus:

```
public class Zwinger<T> {
  private T x;

  public void sperreEin(T x) {
    this.x = x;
  }

  public T befreie() {
    T x = this.x;
    this.x = null;
    return x;
  }
}
```

[8] Wer mit dem Allgemeineren klar kommt hat kein Problem mit dem Speziellen.

Diese Lösung unterscheidet sich kaum von der vorherigen. Allerdings können wir uns beim Herausholen jetzt den Cast ersparen:

```
Zwinger<WolfsHund> z = new Zwinger<WolfsHund>();
WolfsHund volker = new WolfsHund();
z.sperreEin(volker);
volker = z.befreie();
```

Der parametrische Polymorphismus benötigt mehr Typinformationen zur Übersetzungszeit. Die Programme haben dadurch einen höheren Grad an Selbstdokumentation. Sie erfordern aber auch mehr Schreibaufwand und sind weniger flexibel.

Das Einsatzgebiet von parametrischem Polymorphismus sind in erster Linie Container und Algorithmen. In beiden Fällen wird mit Objekten umgegangen deren Fähigkeiten wenig bis gar nicht relevant sind. Vererbungspolymorphismus setzt man ein, wenn bestimmte Fähigkigkeiten der Objekte, mit denen umgegangen wird, wichtig sind, er aber irrelevant ist, welche weiteren Fähigkeiten vorhanden sind und wie genau die Objekte ihre Fähigkeiten ausführen.

4.2.2 Parametrischen und Vererbungs–Polymorphismus kombinieren

Beschränkte Typparameter

Bei Generics mit beschränkten Typparametern werden beide Arten des Polymorphismus kombiniert. Mit

```
public class HundeZwinger<T extends Hund> {
  private T x;

  public void sperreEin(T x) {
    this.x = x;
  }

  public T befreie() {
    T x = this.x;
    this.x = null;
    return x;
  }
}
```

wird ein Zwinger definiert, der nur Hunde aufnehmen kann. Natürlich ist die Beschränkung hier in diesem Beispiel nicht wirklich sinnvoll. Sie wäre es, wenn der Zwinger etwas mit seinen Insassen tun wollte, zu dem nur Hunde fähig sind. Die Beschränkung im Typparameter bringt dann die notwendigen Fähigkeiten zum Ausdruck.

Ein sinnvolleres Beispiel ist darum:

```
interface Milchtier {
  Milch melken();
}

class Stall<T extends Milchtier> {
  private T x;

  public Milch melke() {
    return x.melken();
  }
  ...
}
```

Ein Typparameter kann mit mehr als einer Grenze versehen sein:

```
interface Milchtier {
  Milch melken();
}

interface Grasfresser {
  void fressen(Gras g);
}

class Stall<T extends Milchtier & Grasfresser> {
  private T x;
```

```
  public Milch melke() {
    return x.melken();
  }

  public void fuettern() {
    x.fressen(new Gras());
  }
}
```

Die Notation mag etwas überraschen, aber das Komma war bereits vergeben als Trenner der Liste von Typparametern.

Generische Container sind nicht kovariant im Typargument

Weiter oben (Abschnitt 3.2.1) haben wir bereits erwähnt, dass die modernen Kollektionstypen und die altmodischen Felder sich unterschiedlich in Bezug auf Generics verhalten. Felder sind kovariant im Typparameter. Ein Feld von Kühen kann darum als Feld von Tieren behandelt werden. Kollektionstypen sind dagegen weder kovariant noch kontravariant im Typparameter. Eine Liste von Kühen wird darum nicht als Liste von Tieren akzeptiert. Die Kovarianz der Felder macht es notwendig, dass Felder mit Typmarkierungen ausgestattet werden müssen, die zur Laufzeit geprüft werden. Ein Tier schleicht sich in eine Kuhherde, die gerade als Tierherde angesehen wird.

```
class Gras{}
class Milch{}

class Tier {
  public void fuettere(Gras g) {}
}

class Kuh extends Tier {
  public Milch melke() { return new Milch(); }
}

class Tiger extends Tier {}

public class Tiere {

  public static void fuettere(Tier[] ta) {
    for (Tier t: ta) {
      t.fuettere(new Gras());
    }
    ta[0] = new Tiger(); // <<-- java.lang.ArrayStoreException
  }

  public static void melke(Kuh[] ka) {
    for (Kuh k: ka) {
      k.melke();
    }
  }


  public static void main(String[] args) {
    Kuh[] kuhHerde = {new Kuh(), new Kuh()};
    fuettere(kuhHerde);
    melke(kuhHerde);
  }

}
```

Man beachte, dass das Programm ohne den böswillig eingeschmuggelten Tiger völlig korrekt wäre. Typprüfungen zur Laufzeit bei Feldern, einer Datenstruktur die geradezu das Musterbeispiel für Effizienz sein sollte, ist ein Unding und passt auch nicht zur Philosophie von Java nach der ansonsten nur explizite Casts solche Prüfungen provozieren.

Die Invarianz der Kollektionstypen macht die Programme typsicher und gleichzeitig effizient. Leider geht damit auch eine starke und gelegentlich sehr hinderliche Beschränkung der Ausdruckskraft einher. Viele völlig unproblematische Programme werden aus formalen Gründen vom Compiler abgelehnt. So ist die Listenvariante des Programms von oben gar nicht erst übersetzbar. – Egal ob mit oder ohne Tiger:

```
class Gras{}
```

```
class Milch{}
class Tier { public void fuettere(Gras g) {} }

class Kuh extends Tier {
  public Milch melke() { return new Milch(); }
}

class Tiger extends Tier {}

public class Tiere {

  public static void fuettere(List<Tier> tl) {
    for (Tier t: tl) {
      t.fuettere(new Gras());
    }
    //nichts Boeses passiert hier
  }

  public static void melke(List<Kuh> kl) {
    for (Kuh k: kl) {
      k.melke();
    }
  }


  public static void main(String[] args) {
    List<Kuh> kuhHerde = Arrays.asList(new Kuh[]{new Kuh(), new Kuh()});

    fuettere(kuhHerde); // FEHLER: "The method fuettere(List<Tier>)
            //is not applicable for the arguments List<Kuh>"
    melke(kuhHerde);
  }
}
```

Der Compiler akzeptiert das Programm nicht obwohl es völlig unproblematisch ist. Das Böse wird verhindert, aber viel Gutes ist auch nicht mehr möglich.

Wildcard–Extend

Es ist klar, dass die Einschränkungen, die diese fehlende Kovarianz mit sich bringt, für eine moderne Programmiersprache nicht tolerierbar sind. Ein Programm wie das im letzten Beispiel muss einfach übersetzbar und ausführbar sein.

Ist es auch. Allerdings wird dazu ein weiteres Sprachmittel benötigt, die sogenannten *Wildcards*. Ein *Wildcard* ist ein Typ–Joker. Ein Platzhalter für einen Typ der nach Bedarf gefüllt wird. Ein *Wildcard* wird mit einem Fragezeichen angegeben. Die Methode zum Füttern mit einem *Wildcard–Extend* sieht wie folgt aus:

```
public static void fuettere(List<? extends Tier> tl) {
  for (Tier t: tl) {
    t.fuettere(new Gras());
  }
}

public static void main(String[] args) {
  List<Kuh> kuhHerde = Arrays.asList(new Kuh[]{new Kuh(), new Kuh()});

  fuettere(kuhHerde); // OK
}
```

Das Gute ist jetzt möglich und auch das Böse wird verhindert.

```
public static void fuettere(List<? extends Tier> tl) {
  for (Tier t: tl) {
    t.fuettere(new Gras());
  }
  tl.add(new Tiger()); // <<-- FEHLER-Meldung des Compilers !
}
```

Der Parametertyp *List<? extends Tier>* verwendet ein *Wildcard–Extend* von *Tier*. Er ist kompatibel zu jeder Liste von Subtypen von *Tier*. Der Parametertyp wird damit als kovariant deklariert.

Allerdings bringt ein *Wildcard–Extend* auch eine Einschränkung mit sich. Der Parameter *tl* vom Typ *List<? extends Tier>* akzeptiert den Tiger nicht in seiner *add*–Methode. Ganz im Gegensatz zu einer Liste von Tieren:

```
public static void fuettere(List<Tier> tl) {
  for (Tier t: tl) {
    t.fuettere(new Gras());
  }
  tl.add(new Tiger()); // <<-- OK, alles Bestens
}
```

Wird mit einem *Wildcard–Extend* Kovarianz deklariert, dann sind schreibende Zugriffe nicht mehr erlaubt. Vergleichen wir kurz:

- `f(List<Tier> tl)`
  Alle Operationen auf `tl` sind in `f` erlaubt, aber die Parameterübergabe ist streng beschränkt.
  (Nimm nur Tier-Listen aber erlaube alle Operationen auf ihnen.)
- `f(List<? extends Tier> tl)`
  Nur lesende Operationen auf `tl` sind in `f` erlaubt, aber dafür ist die Parameterübergabe nicht streng beschränkt.
  (Nimm Listen von allem möglichen Getier aber erlaube kein Einfügen.)

Diese Lösung sieht auf den ersten Blick etwas kompliziert aus, aber sie ist einfach und elegant.

Ohne *Wildcard–Extend*–Parameter akzeptiert eine Methode *f* eine Liste von Tieren und nur eine Liste von Tieren.

- In `f` kann jedes Element problemlos als Tier behandelt werden, und
- der Benutzer von `f` wird nach dem Aufruf nichts in der Liste finden, das nicht zum Typ `Tier` passt.

Mit *Wildcard–Extend*–Parameter akzeptiert eine Methode *f* eine Liste von Tieren oder eine Liste von Elementen mit irgendeinem Subtyp von `Tier`.

- In `f` kann jedes Element problemlos als Tier behandelt werden, und
- der Benutzer von `f` wird nach dem Aufruf nichts in der Liste finden, das nicht zum Typ der Elemente passt. Da die Methode `f` nichts genaues über den Typ der Elemente weiß, außer dass es ein ihm völlig unbekannter Subtyp von `Tier` ist, darf sie einfach gar nichts einfügen.

Für `T<? extends Object>` gibt es eine Kurznotation: `T<?>`.

Wildcard–Extend–Parameter verwendet man bei Methoden, die flexibel aufrufbar sein sollen und denen man lesenden Zugriff auf eine Kollektion von Elementen geben will. Die Restriktion richtet sich dabei nach dem, was die Elemente an Funktionalität bereitstellen müssen.

```
class Tier {
  public void fuettere(Gras g) {}
}

abstract class MilchTier extends Tier {
  abstract public int melke();
}

class Kuh extends MilchTier {
  public int melke() { return 50; }
}

class Ziege extends MilchTier {
  public int melke() { return 5; }
}

public static int melken(List<? extends MilchTier> l) {
  int milchMenge = 0;
  for (MilchTier t : l) {
    milchMenge += t.melke();
```

```
  }
  return milchMenge;
}
```

Wildcard–Super

Komplementär zum *Wildcard–Extend* gibt es ein *Wildcard–Super*. Mit ihm wird eine kontravariante Parameterübergabe erlaubt. D.h. eine Methode `f`, die ihren Parameter mit einem *Wildcard–Super*–Typ definiert, akzeptiert alle kontravarianten Argumente und erlaubt zudem das Einfügen von Elementen.

```
public static void f(List<? super Kuh> tl) {
  tl.add(new Kuh()); // OK
  tl.add(new MilchKuh()); // OK
  tl.add(new Tier()); // FEHLER
  tl.add(new Tiger()); // FEHLER
}

public static void main(String[] args) {
  List<Kuh> kuhHerde = Arrays.asList(new Kuh[]{new Kuh(), new Kuh()});
  List<Tier> tierHerde = Arrays.asList(new Tier[]{new Tier(), new Tier()});
  List<Tiger> tigerHerde = Arrays.asList(new Tiger[]{new Tiger(), new Tiger()});
  f(kuhHerde); // OK
  f(tierHerde); // OK
  f(tigerHerde); // <<-- FEHLER: List<Tiger> passt nicht zu List<? super Kuh>
}
```

Wildcard–Super verwendet man bei Methoden, die flexibel aufrufbar sein sollen und denen man schreibenden Zugriff geben will. Mit folgender Methode können wir beispielsweise alle Milchtiere zusammenführen, die mehr als 5 Liter Milch geben:

```
public static void suche GuteMilchTiere (
    List<? extends MilchTier> l, // Nutzung als Milchtier
    List<? super MilchTier> gute // Einfuegen erlauben
  ) {
  for (MilchTier t : l) {
    if (t.melke() > 5) {
      gute.add(t);
    }
  }
}
```

Der Parameter `l` akzeptiert nur Listen deren Elemente Milchtiere sind. Der Parameter `gute` kann Listen von beliebigem Typ oberhalb von `MilchTier` annehmen. Das bringt die minimalen Anforderungen zum Ausdruck, die die Verwendung `t.melke()` und `gute.add(t)` erfordern.

Wildcard–Super und Vergleiche

Nehmen wir Äpfel und Birnen, die beides Früchte mit einem gewissen Gewicht sind:

```
abstract class Frucht {
  protected int gewicht;
  Frucht(int gewicht) { this.gewicht = gewicht; }
}

class Apfel extends Frucht implements Comparable<Apfel>{
  public Apfel(int gewicht) { super(gewicht); }
  public int compareTo(Apfel other) {
    return this.gewicht - other.gewicht;
  }
}

class Birne extends Frucht implements Comparable<Birne>{
  public Birne(int gewicht) { super(gewicht); }
  public int compareTo(Birne other) {
    return this.gewicht - other.gewicht;
  }
}
```

Mit der folgenden generischen Methode kann das Maximum in einer beliebigen Kollektion vergleichbarer Objekte gesucht werden:

```
static <T extends Comparable<T>> T max(Collection<T> c) {
  T m = null;
  for (T x: c) {
    if (m == null || m.compareTo(x) < 0)
      m = x;
  }
  return m;
}
```

Die Methode sieht auf den ersten Blick gut aus. Sie hat aber ein Problem. In einer Kollektion von Äpfeln kann der Schwerste gesucht werden. In einer Kollektion von Birnen kann die schwerste Birne gesucht werden. Aber in einer gemischten Kollektion von Früchten kann nicht die schwerste Frucht gesucht werden:

```
public static void main(String[] args) {
  List<Frucht> fruechte = Arrays.asList(new Frucht[]{ new Apfel(10), new Birne(15) });
  Frucht f = max(fruechte); // FEHLER: Bound mismatch:
  // max(Collection<T>) is not applicable for the arguments (List<Frucht>)
}
```

Klar, auf Früchten ist der Vergleich nicht definiert. Die Lösung ist einfach: definieren wir die Vergleichsfunktion in der Basisklasse:

```
abstract class Frucht implements Comparable<Frucht> {
  protected int gewicht;
  Frucht(int gewicht) { this.gewicht = gewicht; }
  public int compareTo(Frucht other) {
    return this.gewicht - other.gewicht;
  }
}

class Apfel extends Frucht {
  public Apfel(int gewicht) { super(gewicht); }
}

class Birne extends Frucht {
  public Birne(int gewicht) { super(gewicht); }
}
```

dann funktioniert der Vergleich:

```
List<Frucht> fruechte = Arrays.asList(new Frucht[]{ new Apfel(10), new Birne(15) });
Frucht f = max(fruechte); // OK
```

Jetzt können Äpfel mit Birnen verglichen werden!

Leider können aber Äpfel nicht mehr mit Äpfeln und Birnen nicht mehr mit Birnen verglichen werden:

```
List<Apfel> aepfel = Arrays.asList(new Apfel[]{ new Apfel(10), new Apfel(15) });
List<Birne> birnen = Arrays.asList(new Birne[]{ new Birne(10), new Birne(15) });
Apfel a = max(aepfel); // FEHLER: Bound mismatch
Birne b = max(birnen); // FEHLER: Bound mismatch
```

Wieder ist die Fehlerursache klar: Auf Äpfeln und Birnen ist der Vergleich nicht definiert. Diese Erkenntnis hilft aber nicht wirklich weiter. Äpfel und Birnen sind vergleichbar, egal in welcher Mischung, und dies sollte auch ausdrückbar sein. Mit der Beschränkung im Typ–Parameter von `max` wird gesagt, dass der übergebene Typ `T` einen Vergleich "auf T–Ebene" bieten muss. In `T` muss ein Vergleich mit einem anderen `T`–Objekt definiert sein.

Das ist mehr als wir brauchen um einen Vergleich ausführen zu können: Wenn in der *Ableitungshierarchie* von `T` *irgendwo* ein passender Vergleich definiert ist, dann kann verglichen werden. In der Ableitungshierachie von `Apfel` gibt es beispielsweise die Klasse `Frucht` mit einer Vergleichsoperation die auf Äpfel und Birnen passt. Genau das "Irgendwo in einer Superklasse gibt es eine passende Vergleichsoperation" wird mit einem *Wildcard–Super* ausgedrückt.

```
T extends Comparable<? super T>
```

Mit diesem Typ können wir die Vergleichsfunktion mit der gewünschten Flexibilität definieren:

```
static <T extends Comparable<? super T>> T max(Collection<T> c) {
  T m = null;
  for (T x: c) {
  if (m == null || m.compareTo(x) < 0)
    m = x;
  }
  return m;
}
```

Jetzt funktioniert der Vergleich wie erwartet:

```
List<Apfel> aepfel = Arrays.asList(new Apfel[]{ new Apfel(10), new Apfel(15) });
Apfel a = max(aepfel); // OK!

List<Birne> birnen = Arrays.asList(new Birne[]{ new Birne(10), new Birne(15) });
Birne b = max(birnen); // OK!

List<Frucht> fruechte = Arrays.asList(new Frucht[]{ new Apfel(10), new Birne(15) });
Frucht f = max(fruechte); // OK!
```

Brückenmethoden und Vergleiche

Angenommen wir definieren in einer Ableitungshierarchie mehrere Vergleichsoperationen. So könnten beliebige Früchte in Bezug auf ihr Gewicht verglichen werden. Zwei Äpfel sollen aber auf Basis ihrer Röte verglichen werden. Je roter der Apfel um so besser.

```
abstract class Frucht implements Comparable<Frucht> {
  protected int gewicht;
  Frucht(int gewicht) { this.gewicht = gewicht; }
  public int compareTo(Frucht other) {
    return this.gewicht - other.gewicht;
  }
}

class Apfel extends Frucht implements Comparable<Apfel> { // FEHLER
        // interface can not be implemented more than once with different arguments.
  private int roete;
  public Apfel(int gewicht, int roete) {
    super(gewicht);
    this.roete = roete;
  }
  public int compareTo(Apfel other) {
    return this.roete - other.roete;
  }
}
```

Die `compareTo`–Methode soll also in `Apfel` überladen sein. Leider funktioniert das so nicht.

Der Compiler weigert sich bedauerlicherweise die Definition von `Apfel` anzunehmen, weil dort seiner Meinung nach `Comparable` mehr als einmal implementiert ist. Wir wundern uns denn wir haben nur ein *implements* nämlich `implements Comparable<Apfel>`

Der Grund für dieses etwas seltsame Verhalten ist, dass der Compiler Code generieren muss, der einen Apfel mit einen Apfel mit einer ersten Methode vergleicht und einen Apfel mit einer anderen Frucht mit einer zweiten Methode. Das alles muss zur Übersetzungszeit korrekt vorbereitet werden und zur Laufzeit ohne Zugriff auf die generischen Parameter ausgeführt werden. – Diese werden ja bei der Übersetzung eliminiert.

Der Compiler entfernt also alle *Generics* aus dem Code, fügt Casts ein und müsste aus den Definitionen oben so etwas generieren wie:

```
abstract class Frucht implements Comparable {
  protected int gewicht;
  Frucht(int gewicht) { this.gewicht = gewicht; }

  // Compiler generierter Code (Brueckenmethode):
  public int compareTo(Object other) {
    return compareTo((Frucht) other);
```

```
  }

  public int compareTo(Frucht other) {
    return this.gewicht - other.gewicht;
  }
}

class Apfel extends Frucht implements Comparable {
  private int roete;
  public Apfel(int gewicht, int roete) {
    super(gewicht);
    this.roete = roete;
  }

  // Compiler generierter Code (Brueckenmethode):
  public int compareTo(Object other) {
    return compareTo((Apfel) other);
  }

  public int compareTo(Apfel other) {
    return this.roete - other.roete;
  }

  // Fuer den Fruchtvergleich generierter Code
  public int compareTo(Object other) {
    return compareTo((Frucht) other);
  }

  public int compareTo(Frucht other) {
    return super.compareTo(other);
  }

}
```

Was dann zurück übersetzt so etwas ergibt wie:

```
abstract class Frucht implements Comparable<Frucht> {
  ...
}

// FEHLER
// doppelte Implementierung eines generischen Interfaces mit unterschiedlichen Parametern
class Apfel extends Frucht implements Comparable<Apfel>, Comparable<Frucht> {
  ...
}
```

Das Design der *Generics* folgt dem Prinzip, dass der Compiler dafür garantiert, dass die von ihm eingefügten *Casts* immer ohne Fehler ausgeführt werden können. Das ist bei der Definition oben nicht der Fall. Wird ein Apfel mit einer Birne verglichen, dann würde der *Casts*

```
compareTo((Apfel) other);
```

sicher schief gehen. Der Compiler müsste eine dynamische Typprüfung generieren um in die richtige Methode zu verzweigen. Das widerspricht aber dem Geist der *Generics* mit seiner Forderung nach Effizienz und Sicherheit.

Lassen wir die *implements*–Klausel weg, dann wird keine *Bridge*–Methode generiert und alles funktioniert wie gewünscht:

```
package vergleiche;

import java.util.List;
import java.util.Arrays;
import java.util.Collection;

abstract class Frucht implements Comparable<Frucht> {
  protected int gewicht;
  Frucht(int gewicht) { this.gewicht = gewicht; }
  public int compareTo(Frucht other) {
    return this.gewicht - other.gewicht;
  }
```

```
}

class Apfel extends Frucht /*(unterdruecke Bridge-Gen.) implements Comparable<Apfel>*/ {
  private int roete;
  public Apfel(int gewicht, int roete) {
    super(gewicht);
    this.roete = roete;
  }

  public int compareTo(Apfel other) {
    return this.roete - other.roete;
  }
}

class Birne extends Frucht {
  public Birne(int gewicht) { super(gewicht); }
}


public class Comp {

  static <T extends Comparable<? super T>> T max(Collection<T> c) {
    T m = null;
    for (T x: c) {
      if (m == null || m.compareTo(x) < 0)
        m = x;
      }
    return m;
  }

  public static void main(String[] args) {
    List<Apfel> aepfel = Arrays.asList(new Apfel[]{ new Apfel(10, 2), new Apfel(15, 1) });
    Apfel a = max(aepfel); // OK a == new Apfel(10, 2)!
    List<Birne> birnen = Arrays.asList(new Birne[]{ new Birne(10), new Birne(15) });
    Birne b = max(birnen); // OK!
    List<Frucht> fruechte = Arrays.asList(new Frucht[]{ new Apfel(10, 5), new Birne(15) });
    Frucht f = max(fruechte); // OK!
  }
}
```

Das Weglassen der *implements*–Klausel ist natürlich nicht ohne Wirkung. So kann eine Methode die einen Vergleich auf "gleicher Ebene" fordert:

```
static <T extends Comparable<T>> T max(Collection<T> c) {
  T m = null;
  for (T x: c) {
    if (m == null || m.compareTo(x) < 0)
      m = x;
  }
  return m;
}
```

nicht auf Äpfel angewendet werden:

```
List<Apfel> aepfel = Arrays.asList(new Apfel[]{ new Apfel(10, 2), new Apfel(15, 1) });
Apfel a = max(aepfel); // FEHLER: Bounds mismatch
```

Das können wir aber verschmerzen:

```
static <T extends Comparable<? super T>> T max(Collection<T> c) {
  T m = null;
  for (T x: c) {
    if (m == null || m.compareTo(x) < 0)
      m = x;
  }
  return m;
}
```

funktioniert noch.

Allerdings nicht ganz so wie erwartet: In einer Liste von Äpfel wird nicht der roteste sondern der schwerste gesucht. Der Vergleich findet als nach Art der Früchte statt und nicht nach Art der Äpfel, obwohl nur Äpfel mit Äpfeln verglichen werden.

```
List<Apfel> aepfel = Arrays.asList(new Apfel[]{ new Apfel(10, 2), new Apfel(15, 1) });
Apfel a = max(aepfel); // Hmm... liefert den schwersten Apfel
```

Auch das Problem lässt sich mit einer eigenen *Bridge*–Methode lösen

```
class Apfel extends Frucht /*(unterdruecke Bridge-Gen.) implements Comparable<Apfel>*/ {
 private int roete;
 public Apfel(int gewicht, int roete) {
  super(gewicht);
  this.roete = roete;
 }

 // eigene Bridge--Methode
 public int compareTo(Frucht other) {
  if (other instanceof Apfel) {
   return compareTo((Apfel) other);
  } else return super.compareTo(other);
 }

 public int compareTo(Apfel other) {
  return this.roete - other.roete;
 }
}
```

```
List<Apfel> aepfel = Arrays.asList(new Apfel[]{ new Apfel(10, 2), new Apfel(15, 1) });
Apfel a = max(aepfel); // OK, liefert den rotesten Apfel
```

Kapitel 5

Nützliches auf einen ersten Blick

5.1 Dateien

Achtung: Mit Java 7 wurde einige wichtige Neuerungen und Verbesserungen in Bezug auf die Datei–Behandlung eingeführt. Die Ausführungen in diesem Abschnitt beziehen sich darauf und setzen Java 7 voraus.

5.1.1 Dateien und ihre interne Repräsentanten

Dateien als externe Datenbehälter

Dateien sind Datenspeicher. Man kann in ihnen Daten ablegen (schreiben) und sie später wieder entnehmen (lesen). Eine Datei ist damit einem Feld oder einem Kollektionstyp sehr ähnlich. In gewissem Sinne ist eine Datei als Zusammenfassung von Werten zwar ein weiterer Behälter, von anderen Datentypen unterscheidet sie sich aber in einem sehr wesentlichen Punkt. Dateien sind, im Gegensatz etwa zu Feldern, *außerhalb* des Programms. Ihre Existenz ist nicht an die eines Programms gebunden. Sie existieren in der Regel bevor das Programm gestartet wurde und leben nach dessen Ende meist weiter.

Dateien: In der Hardware, im Betriebssystem, im Programm

Dateien sind physikalische Objekte, z.B. bestimmte Bereiche auf einer Magnetplatte oder einer CD. Sie werden vom Betriebssystem verwaltet. Programme greifen (fast) immer über das Betriebssystem auf Dateien zu. Jedes Betriebssystem hat dabei seinen eigenen Satz von Systemaufrufen[1] mit deren Hilfe Programme auf die Hardware von Dateien zugreifen können. Damit ein Programm, das mit Dateien arbeitet, nicht von dem Betriebssystem abhängig ist, enthalten viele Programmiersprachen noch ein eigenes Dateikonzept, das dann vom Compiler auf die Gegebenheiten des Betriebssystems abgebildet wird.

Dateien gibt es also in drei unterschiedlichen Ebenen:

- Hardware: Dateien sind Bereiche auf physikalischen Medien wie Platten, CDs, etc.
- Betriebssystem: Das Betriebssystem verwaltet die physikalischen Medien, ordnet verschiedene Bereiche eines Mediums einem konzeptionell zusammenhängenden Bereich, einer “Datei” zu, verwaltet Zugriffsrechte, Verwaltungsinformationen etc. Dabei hat jedes Betriebssystem sein Konzept von “Datei”, das sich in den Systemaufrufen niederschlägt, mit denen auf die Dateien zugegriffen werden kann. Die Systemaufrufe werden in Operationen auf physikalischen Objekten umgesetzt.
- Programmiersprache: Die Programmiersprachen haben ihr eigenes Konzept von “Datei”. Das vereinfacht für die Programmierer den Umgang mit Dateien und macht Quellprogramme vom Betriebssystem unabhängig. Auf Dateien wird jetzt mit bestimmten Konstrukten der Sprache zugegriffen. Diese werden vom Compiler dann auf das jeweilige Betriebssystem abgebildet.

Die Sprachkonstrukte können normalerweise auch umgangen werden. Das Programm verwendet dann nicht die “Dateien der Programmiersprache”, sondern benutzt direkt über Systemaufrufe die “Dateien des Systems”. So etwas funktioniert natürlich nur, wenn das Betriebssystem eine Schnittstelle hat über die Funktionen aufgerufen werden können. Üblicherweise wird eine solche Schnittstelle in C angeboten.

In manchen Systemen kann ein Programm sogar das Betriebssystem umgehen und direkt auf die Hardware zugreifen. Solche Betriebssysteme sind allerdings Spezialsysteme[2] die nicht für typische Anwendungsentwickler gedacht sind.

Die Klasse java.io.File

Die Dateikonzepte der verschiedenen Programmiersprachen sind zwar unterschiedlich, gewisse Gemeinsamkeiten gibt es aber dennoch. Typischerweise arbeiten die Programme aller Programmiersprachen mit “logischen Dateien”, Programm–interne Repräsentanten (*“Programmdatei”*), die mit einer “wirklichen Datei” (*“Betriebssystemdatei”*) verknüpft werden müssen.

In Java können externe Dateien auf vielfältige Art mit Programm–internen Repräsentanten verbunden werden. Die Klasse `java.io.File` ist der “klassische” Datentyp für die interne Darstellung einer externen Datei. Ein Exemplar dieser Klasse repräsentiert eine Datei oder ein Verzeichnis. Ein kleines Beispiel illustriert die Verwendung:

```
import java.io.File;

public class Test {
```

[1] Im Betriebssystem implementierte Funktionen, die von Anwendungsprogrammen aufgerufen werden können.
[2] oder sehr alt und primitiv, wie etwa DOS

```
 public static void main(String[] args) {
   File f = new File("/home/thomas/tmp");
   if (f.exists()) {
     System.out.print(f+" exisitiert");
     if (f.isFile()) {
       System.out.println(" und ist eine Datei");
     } else if (f.isDirectory()) {
       System.out.println(" und ist ein Verzeichnis mit dem Inhalt");
       for (File ff : f.listFiles()) {
         System.out.println(" " + ff);
       }
     }
   }
 }

}
```

Das Programm prüft ob `/home/thomas/tmp` existiert, und wenn ja, ob es sich um eine Datei oder ein Verzeichnis handelt. Wenn es sich um ein Verzeichnis handelt dann wird auch dessen Inhalt (Dateien und Unterverzeichnisse) ausgegeben. Man bemerke, dass das Programm auf Unix–artige Betriebssysteme ausgelegt ist, in denen "/" das Trennzeichen in Pfadangaben ist.

Das Interface java.nio.file.Path

Ab Java 7 übernimmt das Interface `java.nio.file.Path` die Rolle des internen Repräsentanten einer Datei oder eines Verzeichnisses. Mit den neuen Klassen von Java 7 lässt sich das Beispiel von oben (das aber auch ein korrektes Java 7 Programm ist!) wie folgt formulieren:

```
import java.io.File;
import java.io.IOException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;


public class Test {

 public static void main(String[] args) throws IOException {
   Path path = Paths.get("/home/thomas/tmp");
   if (Files.exists(path)) {
     System.out.print(path + " exisitiert");
   if (Files.isRegularFile(path)) {
     System.out.println(" und ist eine Datei");
   } else if (Files.isDirectory(path)) {
     System.out.println(" und ist ein Verzeichnis mit dem Inhalt:");
     DirectoryStream<Path> ds = Files.newDirectoryStream(path);
     for (Path p: ds) {
       System.out.println(p);
     }
   }
 }

}
```

Die Struktur der Dateiverarbeitung wurde in Java 7 verbessert mit den Helferklassen `Paths` und `Files`, aber ansonsten ist der Unterschied nicht so beeindruckend. Wir sehen allerdings nur die "Oberfläche". Die neue Struktur erlaubt tatsächlich eine größere Unabhängigkeit von der verwendeten Plattform und bietet mehr Möglichkeiten. Dazu ist allerdings ein etwas höherer Schreibaufwand erforderlich:

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.FileVisitResult;
import java.nio.file.Files;
import java.nio.file.Path;
```

```
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;

public class Test {

 public static void main(String[] args) throws IOException {
  // Ein Wurzelverzeichnis (In Unix-systemen gibt es nur eins, ansonsten das erste)
  Path rootDir = FileSystems.getDefault().getRootDirectories().iterator().next();

  // Ein relativer Path
  Path path = FileSystems.getDefault().getPath("home", "thomas", "tmp");

  // Das Wurzelverzeichnis und der relative Pfad werden zusammen gefuehrt
  Path absolutePath = rootDir.resolve(path);

  if (Files.isDirectory(absolutePath))
   System.out.println(absolutePath + " ist ein Verzeichnis mit dem Inhalt (rekursiv):");
   Files.walkFileTree (
      absolutePath,
      new SimpleFileVisitor<Path>() {
       @Override
       public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
       throws IOException {
         System.out.println(file);
         return FileVisitResult.CONTINUE;
           }
      }
   );
 }
}
```

Man sieht dass der Bezug zu irgendeinem Dateisystem weg gefallen ist. Der Pfad wird abstrakt von einer Wurzel aus gebildet. Dann wird der Inhalt des Verzeichnisses nicht “flach” sondern rekursiv bis in alle Verästelungen aufgelistet. Im Java–Tutorial[3] finden sich weitere Erläuterungen und Beispiele.

java.nio.file.Path und java.io.File

Die “alte” Klasse `File` und die neue Klasse `Path` können ineinander konvertiert werden:

```
Path path = ...
File f = path.toFile();
Path path = f.toPath();
```

Dies dient aber nur der Kompatibilität von alten und neuen Code. Generell wird nicht mehr empfohlen mit der Klasse `java.io.File` zu arbeiten.

5.1.2 Operationen auf Textdateien

5.1.3 Textdatei lesen

Um den Inhalt einer Datei zu lesen, muss zugeordneter Datenstrom erzeugt werden. Datenströme sind das zentrale Konzept mit dem eine externe Datenquelle oder Datensenke mit einem Programm in Verbindung treten kann. Man unterscheidet

- Zeichenströme *Character Streams* und
- Byteströme *Byte Streams*

Zeichenströme liefern die Text oder nehmen Text auf. Byteströme liefern binäre Daten oder nehmen solche auf.

Um eine Textdatei zu lesen, muss sie mit einem Zeichenstrom verbunden werden:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.Charset;
```

[3] http://docs.oracle.com/javase/tutorial/essential/io/fileio.html

```
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Scanner;

public class Test {

 public static void main(String[] args) throws IOException {
  // Pfad zur Datei:
  Path path = FileSystems.getDefault().getPath("/home/thomas/tmp/test.txt");
  // Platz fuer den gelesenen Text:
  StringBuilder text = new StringBuilder();
  // gepufferten Eingabestrom erzeugen
  // der zweite Parameter sagt dass der Zeichensatz des Systems verwendet werden soll.
  BufferedReader reader = Files.newBufferedReader(path, Charset.defaultCharset());

  // Eingabestrom mit Scanner lesen
  Scanner scanner = new Scanner(reader);
  try {
   // zeilenweise lesen
   while (scanner.hasNextLine()){
   text.append(scanner.nextLine());
   }
  } finally{
   scanner.close();
  }
  // gelesene Text ausgeben:
  System.out.println(text.toString());
 }
}
```

Beim Lesen von Textdateien muss ein Zeichensatz angegeben werden. In der Regel ist das der Zeichensatz des Systems auf dem das Programm abläuft: der *default Charset*. Hier kann auch explizit ein anderer Zeichensatz angegeben werden. Das ist beispielsweise nützlich, wenn die zu lesenden Datei auf einem anderen System mit anderem Zeichensatz erstellt wurde. Zeilenweises Lesen einer Textdatei geht auch noch einfacher:

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;

public class Test {

 public static void main(String[] args) throws IOException {
  Path path = FileSystems.getDefault().getPath("/home/thomas/tmp/test.txt");
  for (String line: Files.readAllLines(path, Charset.defaultCharset())) {
   System.out.println(line);
  }
 }

}
```

Wenn wir alles in einen String lesen wollen:

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;

public class Test {

 public static void main(String[] args) throws IOException {
  Path path = FileSystems.getDefault().getPath("/home/thomas/tmp/test.txt");
  String content = new String(Files.readAllBytes(path), Charset.defaultCharset());
  System.out.println(content);
 }
```

```
}
```

5.1.4 Dateien erzeugen, löschen und kopieren

Dateien können leicht erzeugt werden. Beispiel:

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;

public class Test {

  public static void main(String[] args) throws IOException {
    Path path = FileSystems.getDefault().getPath("D:\\Users\\Thomas\\neuedatei.txt");
    Files.createFile(path);
  }

}
```

Wenn die Datei bereits existiert, dann wird das Programm mit einer `FileAlreadyExistsException` abgebrochen.

Auch das Kopieren ganzer Dateien ist kein Problem:

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;

public class Test {

  public static void main(String[] args) throws IOException {
    Path path1 = FileSystems.getDefault().getPath("D:\\Users\\Thomas\\altedatei.txt");
    Path path2 = FileSystems.getDefault().getPath("D:\\Users\\Thomas\\neuedatei.txt");

    if (Files.exists(path1) && !Files.exists(path2)) {
      Files.copy(path1, path2);
    }

  }

}
```

Die Kopieraktion wird mit einer `FileAlreadyExistsException` abgebrochen, wenn die die Zieldatei schon existiert. Will man die Zieldatei überschreiben, dann setzt man eine entsprechende Option:

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;

public class Test {

  public static void main(String[] args) throws IOException {
    Path path1 = FileSystems.getDefault().getPath("D:\\Users\\Thomas\\altedatei.txt");
    Path path2 = FileSystems.getDefault().getPath("D:\\Users\\Thomas\\neuedatei.txt");

    Files.copy(path1, path2, REPLACE_EXISTING);
  }

}
```

Wir sehen, dass die Helferklasse `Files` eine Vielzahl an Operationen bereit hält, mit denen Dateien untersucht und manipuliert werden können. Für das Löschen gibt natürlich auch etwas:

```
import java.io.IOException;
```

```
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.NoSuchFileException;
import java.nio.file.Path;

public class Test {

 public static void main(String[] args) throws IOException {
  Path path = FileSystems.getDefault().getPath("D:\\Users\\Thomas\\altedatei.txt");
  try {
   Files.delete(path);
  } catch (NoSuchFileException e) {
   System.err.println("Die Datei " + path + " existiert gar nicht!");
  }

 }

}
```

Der Leser ist ermuntert sich in der API–Dokumentation weitere Anregungen oder, bei Bedarf, Problemlösungen zu holen.

5.1.5 Textdateien beschreiben

Das Schreiben einer Datei ist symmetrisch zum Lesen:

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Test {

 public static void main(String[] args) throws IOException {
  Path dir = Paths.get("/home/thomas/tmp"); // Verzeichnis
  Path f = Paths.get("datei.txt"); // Datei

  Path path = dir.resolve(f); // Pfad zur Datei

  // Im UTF-8-Format speichern
  BufferedWriter writer = Files.newBufferedWriter(path, Charset.forName("UTF-8"));

  for (int i=0; i< 100; i++) {
   writer.write("Zeile " + i + "\n");
  }
  writer.flush(); // Puffer leeren
  writer.close(); // Datei schliesen
 }

}
```

Hat man ein iterierbares Objekt das Strings liefert, dann kann man diese mit `Files.write` in einer Datei speichern. Beispiel:

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Iterator;

public class Test {

 public static void main(String[] args) throws IOException {
  Path f = Paths.get("/home/thomas/tmp/datei.txt");

  // lines ist ein iterierbares Objekt
```

```
   Iterable<String> lines = new Iterable<String>() {
    @Override
    public Iterator<String> iterator() {
     return new Iterator<String>() {
      int nr = 1;

      @Override
      public boolean hasNext() {
       return nr < 100;
      }
      @Override
      public String next() {
       return "Zeile Nr. " + nr++;
      }
      @Override
      public void remove() {
       throw new UnsupportedOperationException();
      }};
    }
   };

   // Alle Zeilen speichern
   Files.write(f, lines, Charset.forName("UTF-8"));
  }

}
```

Der Zeilenvorschub wird automatisch erzeugt. `lines` könnte natürlich auch eine beliebige iterierbare Datenstruktur sein, z.B. eine Liste.

Wenn eine Datei zum Schreiben geöffnet wird, dann wird ihr bisheriger Inhalt gelöscht. Dieses Verhalten ist nicht immer erwünscht. Manchmal möchte man zu den Informationen in einer Datei neue hinzufügen. Die Datei muss dazu mit der *Append–* Option geöffnet werden. Wollen wir beispielsweise weitere Zeilen zu der eben beschrieben Datei hinzufügen, dann geht das mit:

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.OpenOption;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.Arrays;
import java.util.List;

public class Test {

 public static void main(String[] args) throws IOException {
  Path f = Paths.get("/home/thomas/tmp/datei.txt");

  List<String> moreLines = Arrays.asList(new String[]{"line A", "line B"});
  Files.write(f,
   moreLines,
   Charset.forName("UTF-8"),
   new OpenOption[] {StandardOpenOption.APPEND});
 }

}
```

Das Programm hängt an den bestehenden Inhalt der Datei noch zwei weitere Zeilen an.

Das anhängende Schreiben funktioniert auch über einen `BufferedWriter`. Die Datei muss dazu nur wieder mit der *Append–* Option geöffnet werden:

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.OpenOption;
```

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class Test {

 public static void main(String[] args) throws IOException {
  Path f = Paths.get("/home/thomas/tmp/datei.txt");

  BufferedWriter writer = Files.newBufferedWriter(f,
     Charset.forName("UTF-8"),
     new OpenOption[] {StandardOpenOption.APPEND});

  writer.write("und noch eine letzte Zeile hinten dran \n");

  writer.flush();
  writer.close();

 }

}
```

Dateien auswählen

Eine nützliche Hilfe bei der Auswahl von Dateien und Verzeichnissen ist der `JFileChooser`. Wir beschränken uns auf ein Beispiel:

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

import javax.swing.JFileChooser;

public class Test {

 public static void main(String[] args) throws IOException {

  JFileChooser fileChooser = new JFileChooser();

  // Biete Verzeichnisse und Dateien zur Auswahl an (default-Verhalten)
  fileChooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);

  int result = fileChooser.showOpenDialog(null);

  if (result == JFileChooser.APPROVE_OPTION) {
   File f = fileChooser.getSelectedFile();
   Path p = f.toPath();

   // Info ueber die Auswahl:
   System.out.print("Sie haben gewaehlt: " + p);
   if (Files.isDirectory(p)) {
    System.out.println(" (Verzeichnis)");
   } else if (Files.isRegularFile(p)) {
    System.out.println(" (Datei)");
    System.out.println("Die Datei ist " +
       (Files.isExecutable(p)?" ausfuehrbar":" ") +
       (Files.isReadable(p)?" lesbar":" ") +
       (Files.isWritable(p)?" schreibbar":" "));
    System.out.println("Der Dateityp ist: "+Files.probeContentType(p));
   }
  }
 }

}
```

Diese Beispiele sollen erst einmal reichen um den Leser von der Mächtigkeit und Eleganz der Datei–Operationen in Java 7 zu überzeugen und zu einem Studium der API–Dokumentation und des Tutorials[4] anzuregen.

5.1.6 Textdateien analysieren

Mit dem Lesen von Textdateien ist oft die Analyse des Dateiinhalts verbunden. An Hand von Beispielen wollen wir hier eine kurze Einführung geben. Die "eingebauten" Mittel zur Textanalyse in Java sind:

- Die Klassen `StringTokenizer` und `StreamTokenizer`
- Die Methode `String.split`
- Die Klasse `Scanner`
- Die Klassen `Pattern` und `Matcher`

`StringTokenizer` und `StreamTokenizer` gelten als veraltet und werden darum hier nicht weiter beachtet.

Die split–Methode

Die Methode `split`–Methode der Klasse `String` bietet die einfachste Art einen Text zu analysieren. Sie zerlegt einen String an Trennsymbolen zu zerlegen. Ein einfaches Beispiel ist:

```
String s ="ein Text mit Leerzeichen\n und Zeilenvorschub";
String[] teile = s.split(" ");
for (String st : teile) {
  System.out.println("<<"+st+">>");
}
```

Es erzeugt die Ausgabe

```
<<mit>>
<<Leerzeichen
>>
<<und>>
<<Zeilenvorschub>>
```

Der Parameter von `split` ist hier das Trennsymbol, das Leerzeichen. Das ist aber nur ein sehr einfaches Beispiel dafür, wie die Trennstelle definiert werden kann. Mit

```
String s ="ein Text mit Leerzeichen\n und Zeilenvorschub";
String[] teile = s.split("\\s+");
for (String st : teile) {
  System.out.println("<<"+st+">>");
}
```

können wir alle Folgen von "weißen Zeichen" (Leerzeichen, Zeilenvoschub, Tabulatoren) als Trenner definieren und erzeugen so die Ausgabe

```
<<ein>>
<<Text>>
<<mit>>
<<Leerzeichen>>
<<und>>
<<Zeilenvorschub>>
```

Das Argument von `split` ist nicht einfach eine Zeichenfolge, sondern ein *regulärer Ausdruck*. Reguläre Ausdrücke (oft als "*Regex*" abgekürzt) sind ein vielseitiges und weit über die Java–Welt hinaus verbreitetes Mittel um Textstrukturen auszudrücken.[5] Wer sich ernsthaft mit der Analyse von Textdateien beschäftigen will, kommt um ein Studium der regulären Ausdrücke nicht herum.

[4] `http://docs.oracle.com/javase/tutorial/essential/io/fileio.html`
[5] siehe etwa `http://de.wikipedia.org/wiki/Regulärer_Ausdruck`

Die Scanner–Klasse

Die Klasse `Scanner` nutzt ebenfalls reguläre Ausdrücke. Beispielsweise kann das Trennsymbol mit einen regulären Ausdruck definiert werden. Mit:

```
String s ="ein Text mit Leerzeichen\n und Zeilenvorschub";
Scanner scan = new Scanner(s);
scan.useDelimiter("\\s+");
while (scan.hasNext()) {
  System.out.println("<<"+scan.next()+">>");
}
```

kann die gleiche Ausgabe wie im letzten Beispiel oben erzeugt werden.

Ein Scanner kann mit einer Datei als Datenquelle kombiniert werden. Im folgenden Beispiel wird geprüft wie oft ein bestimmtes Wort in einer Datei vorkommt.

```
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Scanner;

public class Test {

  public static void main(String[] args) throws IOException {
    Path path = Paths.get("/home/thomas/tmp/datei.txt");
    Scanner scan = new Scanner(path);
    int count = 0;
    while ( scan.hasNextLine() ){
         String line = scan.nextLine();
         String[] words = line.split("\\W+");
         for (String word: words) {
           if (word.equals("Blabla")) { count++; }
         }
     }
     System.out.println("Die Datei "+path+" enthaelt"+count+"-mal das Wort \"Blabla\"");
  }

}
```

Die Klassen Pattern und Matcher

Mit den Klassen `Pattern` und `Matcher` stellen die volle Kraft regulärer Ausdrücke für die Analyse von Texten zur Verfügung. Das Thema ist zu fortgeschritten für diese Einführung. Wir begnügen uns darum mit einen kleinen Beispiel mit dem in einer Datei jedes Vorkommen einer Aussage nach einem der Muster

... X liebt Y ...
... X liebt den Y ...
... X liebt die Y ...
... X steht auf den Y ...
... X steht auf die Y ...

entdeckt und in einer Abbildung als

$X \rightarrow Y$

abgespeichert werden.

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Map;
import java.util.TreeMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
public class Test {

 public static void main(String[] args) throws IOException {
  Path path = Paths.get("/home/thomas/tmp/datei.txt");

  Map<String, String> relation = new TreeMap<String, String>();

  // das Muster: ein regulaerer Ausdruck
  String regex = "((.* )|^^)(?<wer>\\w+) (liebt|(steht auf)) ((den|die) )?(?<wen>\\w+).*";

  // der uebersetzte regulaere Ausdruck
  Pattern pattern = Pattern.compile(regex);

  // Schleife ueber alle Zeilen
  for (String line : Files.readAllLines(path, Charset.defaultCharset())) {

   // Jede Zeile in "Saetze" (Punkt als Trenner) aufbrechen
   String[] parts = line.split("\\.");

   // Jeden "Satz" verarbeiten
   for (String sentence : parts) {
    Matcher matcher = pattern.matcher(sentence.trim());

    // Passt der "Satz" zum Muster ?
    if (matcher.matches()) {
     // die mit WER und WEN markierten Gruppen einander zuordnen
     relation.put(matcher.group("wer"), matcher.group("wen"));
    }
   }
  }

  for (String wer: relation.keySet()) {
   System.out.println(wer + " liebt " + relation.get(wer));
  }
 }
}
```

Das Beispiel enthält einen komplexen regulären Ausdruck. Wir wollen ihn hier nicht die Details regulärer Ausdrücke besprechen, das ist an genügend anderen Stellen bereits geschehen.[6] Im Kern läuft es darauf hinaus, dass nach dem Muster

... WER liebt / steht auf (den/die) WEN ...

gesucht wird. Mit

```
(?<wer>\\w+)
```

und

```
(?<wen>\\w+)
```

wird eine *Gruppe* identifiziert. Gruppen sind Teilausdrücke, die von geklammerten Teilen im Muster erkannt werden. Mit `(?<N>`*pattern*`)` wird der Gruppe, die zu *pattern* passt, der Name `N` gegeben. In

```
Klaus liebt den Klausi
```

werden beispielsweise `Klaus` und `Klausi` als Gruppen mit den Namen `wer` und `wen` im Muster

```
... (?<wer>\\w+) (liebt|(steht auf)) ((den|die) )?(?<wen>\\w+) ...
```

erkannt.

Mustererkennung mit Gruppen ist ein mächtiger Mechanismus zur Textverarbeitung auf den hier nur hingewiesen werden soll. Der Leser sei wieder an die API–Dokumentation als Einstieg in eine tiefer gehende Beschäftigung mit der Thematik verwiesen.[7]

[6] `docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html` ist ein guter Einstiegspunkt.

[7] `docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html` enthält im Abschnitt *Groups and capturing* Informationen zum Konzept der Gruppen.

5.2 Graphische Oberflächen: Erste Einführung

5.2.1 Grundprinzipien

Vorbemerkung

Graphische Oberflächen sind komplexe Software–Gebilde, die dazu noch etliche fortgeschrittene Java–Technologien erfordern. Technologien, die wir in dieser Einführung nicht behandeln wollen. Um aber nicht zu lange auf wenigstens einfache Oberflächen verzichten zu müssen, beschäftigen wir uns hier mit einem einfachen Beispiel. Die verwendeten Konstrukte werden dabei nicht in aller Ausführlichkeit behandelt werden. Dieses Kapitel ist damit, wie das Kapitel über Dateien, als ein einfaches Kochrezept für simple Gerichte gedacht.

AWT, Swing, SWT

Die Gestaltung graphischer Oberflächen ist eine Aufgabe, die über die Fähigkeiten des Sprachkerns von Java hinausgeht. Man benötigt die Unterstützung von speziellen Paketen, die aber glücklicherweise mit einer Java–Installation zur Verfügung stehen. Man hat dabei die Auswahl aus drei Varianten an GUI– (*Graphical User Interface*) Unterstützung, die auch gemischt angewendet werden können:

- *AWT*, das *Abstract Window Toolkit*
  mit dem Paket `java.awt`.
- *SWING*
  mit dem Paket `javax.swing`.
- *SWT*
  Das *Standard Windowing Tollkit* von IBM.

Das AWT benutzt die Elemente des darunter liegenden Systems, auf dem die Programme ablaufen. Es ist nicht in Java, sondern in Maschinensprache realisiert, läuft ohne virtuelle Maschine direkt auf dem System und dementsprechend schnell. Der Nachteil des AWT ist, dass es in seinen Möglichkeiten beschränkt ist und das Aussehen der Oberflächen vom System abhängt.

SWING–Klassen sind dagegen in Java implementiert. Sie haben eine umfangreichere Funktionalität und das Aussehen der Oberflächen ist unabhängiger vom verwendeten System. SWING basiert auf AWT und ist naturgemäß etwas langsamer als dieses.

SWT wird weniger oft benutzt und wir gehen hier nicht weiter darauf ein. Es bleiben AWT und Swing. Die Frage welches der beiden besser ist, soll hier nicht interessieren. Wir verwenden von beiden das gerade Passende.

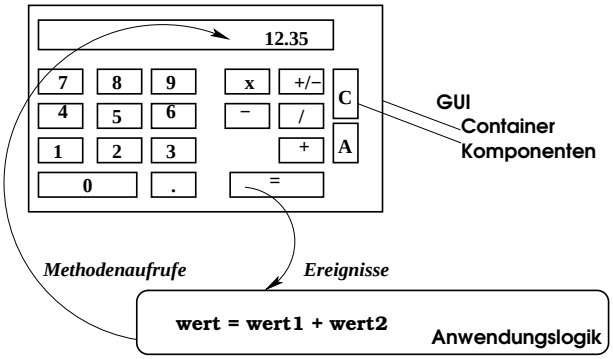


Abbildung 5.1: Struktur einer GUI-Anwendung

Struktur einer GUI

Eine Anwendung mit graphischer Oberfläche, kurz eine GUI–Anwendung, besteht immer aus zwei wesentlichen Bereichen: der Anwendungslogik und der Oberfläche. Beide Bereiche setzen sich aus Objekten zusammen, die mit anderen Objekten des

gleichen und des anderen Bereichs kommunizieren (siehe Abbildung 5.1). Die GUI–Objekte sind *Komponenten* wie Schaltflächen, Textanzeigen etc. und *Container* in denen die Komponenten zu einer Einheit zusammengesetzt sind. Das *Layout* legt dabei fest wie die Komponenten in einem Container anzuordnen sind.

Benutzeraktionen wie das Anklicken einer Schaltfläche, werden als *Ereignisse* von den Komponenten registriert und an die Anwendungslogik gemeldet, die wiederum durch Methodenaufrufe deren Aussehen modifizieren kann. Zusammengefasst:

Eine GUI–Anwendung besteht aus:

- Graphischer Oberfläche (GUI) mit
  - Komponenten
  - Container
- Anwendungslogik

Eine GUI hat ihren eigenen Handlungsstrang

Ein Programm startet mit dem Aufruf der Funktion `main` und endet mit deren Ende. Das war bisher die Grundregel für die Ausführung von Java–Anwendungen. Mit graphischen Oberflächen ändert sich die Sache etwas. Zwar starten die Programme immer noch mit der `main`–Funktion aber sie enden nicht mit ihr. Die Oberfläche wird zwar aus `main` heraus aufgebaut, entfaltet aber dann ihr eigenes Leben. Sie registriert Ereignisse – vom Benutzer ausgelöste Aktionen – informiert die Anwendung darüber und diese kann dann wiederum auf die Oberfläche einwirken.

Ein Beispiel für eine sehr einfache GUI–Anwendung ist:

```
import javax.swing.JFrame;

public class Gui0 extends JFrame {
  //Fenstergroesse in Pixel:
  public static final int BREIT = 400;
  public static final int HOCH = 300;

  //Konstruktor, Titel setzen
  public Gui0(String title) {
    super(title);
  }

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

    //Erzeugt GUI (hier einfach ein Fenster):
    Gui0 einFenster = new Gui0("HUI, eine GUI!");
    einFenster.setSize(BREIT, HOCH);

    //macht sie sichtbar:
    einFenster.setVisible(true);

    // Ende von main, aber GUI lebt weiter
    System.out.println("Ende von main, Programm lebt weiter");
    System.out.println("GUI hat Kontrolle uebernommen!");
  }
}
```

Dieses einfache Fenster kann es nicht, aber in einer realistischen Anwendung könnte das Fenster auf Benutzereingaben reagieren und sie an eine Verarbeitungslogik weiter melden.

Die `main`–Funktion erzeugt ein Fenster als Exemplar der Klasse `Gui0` und endet dann. Das Programm ist damit aber nicht zu Ende, es existiert in Form des Fensters weiter. Ein Java–Programm kann beliebig viele “Handlungsstränge”, sogenannte *Threads* aufmachen.

Threads

Die GUI wird immer als eigenständiger *Thread* geführt. Threads sind ein fortgeschrittenes Thema mit dem wir uns in dieser Einführung nicht beschäftigen. Wir merken uns lediglich, dass GUI und alles was von `main` aus aktiviert wird, unabhängig voneinander aktiv ist. `main` und die graphische Oberfläche sind eigenständige Threads: Handlungsfäden, die unabhängig

voneinander verfolgt werden, so wie zwei Programme, der Code ineinander verwoben ist, die aber unabhängig voneinander ausgeführt werden.

5.2.2 Graphische Objekte erzeugen

Container und Komponenten

Das Fenster, das als Exemplar der Klasse `Gui0` erzeugt wird, ist ein Beispiel für einen *Container*. Praktisch seine gesamte Funktionalität hat `Gui0` von der Klasse `JFrame` geerbt (übernommen). Mit

```
public class Gui0 extends JFrame {...}
```

haben wir `Gui0` als *Ableitung* von `JFrame` definiert. Damit besitzt es automatisch alle Fähigkeiten dieser vorgegebenen Container–Klasse.

Ein Container ist genau das, was sein Name suggeriert: ein Behälter für andere Dinge. Dies können entweder Komponenten oder selbst wieder Container sein. Eine graphische Anwendung besteht also aus einer Hierarchie von Containern und Komponenten. Ganz oben steht ein `JFrame`. In ihm können "Unter–Container" platziert werden oder Komponenten, d.h graphische Elemente wie `JLabel` (ein "Label", eine feste Beschriftung) oder `JButton` (ein Knopf). Wie haben also eine Anwendung stets als hierarchische Anordnung der Bestandteile:

- `JFrame`: Das oberste (äußerste) Fenster, der Container, der alles umfasst,
- `JPanel`: Container auf einer Zwischenebene,
- Komponenten wie
  - `JLabel`
  - `JButton`
  - ...

Der äußerste Rahmen, ein Exemplar der Klasse `JFrame`, enthält automatisch einen Container in den Komponenten und/oder weitere `JPanel`s eingefügt werden können. Im folgenden Beispiel setzen wir ein Label mit der Beschriftung `Ich bin ein Label` in unser Fenster:

```
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Gui0 extends JFrame {
  public static final int WIDTH = 400;
  public static final int HEIGHT = 300;

  public Gui0(String title) {
    super(title);
  }

  public static void main(String args[]) {

    // Oberstes Fenster erzeugen
    //
    Gui0 einFenster = new Gui0("HUI, eine GUI!");
    theWindow.setSize(WIDTH, HEIGHT);

    // Container des obersten Fensters holen
    // (in diesen koennen weitere Elemente platziert werden)
    //
    Container container = einFenster.getContentPane();

    // Ein Label erzeugen
    JLabel label = new JLabel("Ich bin ein Label");

    // Label in den container setzen
    //
    container.add(label);

    einFenster.setVisible(true);
```

```
  }
}
```

Das Fenster wird hier aus einer `main`–Funktion erzeugt, die Bestandteil der Klasse ist. Selbstverständlich hätte dies auch jede andere Funktion oder Methode einer beliebigen anderen Klasse übernehmen können.

5.2.3 Graphische Objekte anordnen

Layout

Sobald mehr als ein Element in einen Container eingefügt wird, müssen wir kontrollieren, an welche Stelle sie platziert werden. Wenn nicht, dann werden sie einfach übereinander gesetzt und nur das letzte ist sichtbar.

Die Verantwortung für das Layout der Komponenten in einem Container trägt der Layout–Manager. Es gibt verschiedene Arten von Layout–Managern die jeweils andere Strategien oder Prinzipien zur Layout–Kontrolle anwenden. Das Layout kann also auf verschiedene Arten kontrolliert werden, je nach dem welcher *Layout–Manager* verwendet wird. Layout–Manager sind Objekte einer Layout–Klasse. Beispielsweise wird mit

```
int zeilen = 3;
int spalten = 4;
LayoutManager lm = new GridLayout(zeilen, spalten);
container.setLayout(lm);
```

ein `GridLayout`–Manager erzeugt und an `container` übergeben. Jetzt können Komponenten eingefügt werden, ohne dass sie sich gegenseitig überdecken. Mit

```
for ( int i = 0; i<2*3; i++)
   container.add(new JLabel("L-"+i));
```

werden dann 6 Labels erzeugt und in zwei Zeilen und drei Spalten positioniert.

Verwendet man das *Flow–Layout*, dann erfolgt die Positionierung völlig automatisch. Die Komponenten werden zeilenweise von links nach rechts in dem verfügbaren Bereich positioniert. Mit

```
container.setLayout( new FlowLayout() );
for ( int i = 0; i<2*3; i++)
   container.add(new JLabel("L-"+i));
```

werden beispielsweise alle sechs Labels zentriert in einer Zeile angeordnet, wenn diese breit genug ist. Eine linksbündige Ausrichtung wird mit

```
container.setLayout( new FlowLayout(FlowLayout.LEFT) );
```

gefordert.

Mit dem *Border–Layout* können die Komponenten in den vier Himmelsrichtungen oder in der Mitte positioniert werden. Beispiel:

```
container.setLayout(new BorderLayout());}

container.add(new JButton(" NORD "),BorderLayout.NORTH);
container.add(new JButton(" SUED "),BorderLayout.SOUTH);
container.add(new JButton(" OST "),BorderLayout.EAST);
container.add(new JButton(" WEST "),BorderLayout.WEST);
container.add(new JButton(" MITTE "),BorderLayout.CENTER);
```

Die wichtigsten Layout–Manager (–Klassen) sind zusammengefasst:

`FlowLayout`	aneinander gereiht
`BorderLayout`	geografisch angeordnet
`GridLayout`	gitterförmig bei gleicher Größe
`CardLayout`	zeigt eine von vielen
`GridBagLayout`	flexible u. komplexe Variante von GridLayout

Packen oder Größe explizit setzen

Alle Layout–Manager brauchen Informationen über die Größe der Komponenten, die sie anordnen sollen. Ein Label beispielsweise braucht genügend Platz, um seine Beschriftung darstellen zu können. Um diese Dinge muss man sich im Allgemeinen nicht selbst kümmern. Die Größen der Komponenten und die Größenverhältnisse zwischen einem Container und seinen Bestandteilen regeln diese und der Layout–Manager.

Bei einem `JFrame`–Objekt sollte man allerdings sagen, ob es

- `setSize`–Methode: eine bestimmte vorgegebene Größe haben soll oder
- `pack`–Methode: ob es gerade so groß sein soll, dass seine Bestandteile hineinpassen.

```
JFrame einFenster = new JFrame( .... );
Container c = einFenster.getContentPane();
c.setLayout( ... );
c.add( ... );
...
//einFenster.setSize( 300, 200 ); // ENTWEDER explizite Groesse setzen
einFenster.pack();         // ODER  gerade gross genug

einFenster.setVisible( true );
```

Verschachtelte Container

Container können Komponenten und oder andere Container enthalten, dabei kann jedem Container sein eigenes Layout zugeordnet werden. Im folgenden Beispiel haben wir einen äußeren Container (`c`) mit 5 Bestandteilen im Border–Layout. Der erste Bestandteil von `c` ist ein Container (`jp`) im Flow–Layout mit zwei Sub–Komponenten. In der Übersicht:

- Das JFrame–Objekt `einFenster` enthält den
  - Container `c` bestehend aus:
    - einem JPanel `jp` mit:
      - einem JButton `NORD-1`
      - einem JButton `NORD-2`
    - einem JButton `OST`
    - einem JButton `SUED`
    - einem JButton `WEST`
    - einem JButton `ZENTRUM`

und im Programmcode:

```
JFrame einFenster = new JFrame( "BorderLayout" );
Container c = einFenster.getContentPane();
c.setLayout(new BorderLayout() );

JPanel jp = new JPanel( new FlowLayout() );
c.add( jp, BorderLayout.NORTH );

jp.add( new JButton( "NORD-1" ) );
jp.add( new JButton( "NORD-2" ) );

c.add( new JButton( "OST" ), BorderLayout.EAST );
c.add( new JButton( "SUED" ), BorderLayout.SOUTH );
c.add( new JButton( "WEST" ), BorderLayout.WEST );
c.add( new JButton( "ZENTRUM" ), BorderLayout.CENTER );

einFenster.pack();
einFenster.setVisible( true );
```

Abbildung 5.2: Verschachtelte Container mit Border- und FlowLayout

5.2.4 Graphische Objekte aktivieren

Ereignisse und Listener

Jetzt haben wir graphische Oberflächen erzeugt mit Label, Knöpfen etc. Nur, es passiert nichts, wenn ein Knopf angeklickt wird. Um das zu ändern, muss den graphischen Objekten jeweils ein *Listener* zugeordnet werden, der beispielsweise das *Ereignis* "Knopf gedrückt" registriert und eine entsprechende Aktion einleitet.

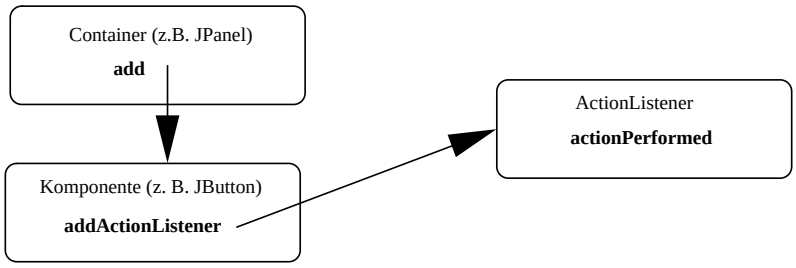


Abbildung 5.3: Komponente und ihr Listener

Ereignisse

Wenn der Benutzer eine Taste drückt oder die Maus bewegt, erzeugt er damit ein sogenanntes *Ereignis* (*Event*). Steuerelemente, wie beispielsweise Knöpfe, können solche Ereignisse registrieren und auch selbst neue Ereignisse erzeugen. Klickt beispielsweise der Benutzer einen Knopf (ein `JButton`–Objekt) an, dann wird ein Ereignis erzeugt.

Ereignisse gibt es in zwei Varianten:

- *ActionEvent*: von einer Komponente ausgelöstes Ereignis
- *WindowEvent*: von einem Fenster ausgelöstes Ereignis

Ein Action–Event wird von Komponenten wie Textfeldern, Knöpfen, etc. ausgelöst. Sie entstehen also wenn der Benutzer die graphische Oberfläche benutzt. Ein Window–Event entsteht, wenn Fenster geschlossen oder sonst manipuliert werden. Ein Event ist ein Objekt einer Event–Klasse. Es hat Attribute, die weitere Informationen über das Ereignis liefern.

Action–Listener

Nur wenn ein "Zuhörer", ein *Listener* für ein Ereignis registriert wurde, kann auch das Ereignis registriert werden. Im folgenden Beispiel versehen wir einen Knopf mit einem Listener, der registriert, wenn der Knopf gedrückt wird:

```
...
// Knopf erzeugen:
JButton knopf = new JButton(" Klick-Mich ");
```

```
// Instanz einer Action-Listener-Klasse erzeugen und
// Knopf mit diesem Action-Listener versehen
knopf.addActionListener( new KlickListener() );

// Knopf im Container platzieren
container.add(knopf, BorderLayout.CENTER);
...
```

Jeder Listener ist eine Instanz einer Listener–Klasse. Hier wird ein Action–Listener gebraucht. Also müssen wir eine Action–Listener–Klasse definieren und eine entsprechende Instanz erzeugen. Die Instanz wurde schon erzeugt, es fehlt noch die Klasse:

```
class KlickListener
     implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Huch, der Knopf wurde geklickt!");
  }
}
```

Ein Action-Listener muss die Methode `actionPerformed` implementieren. In ihr wird festgelegt, was passieren soll, wenn der Action-Listener über ein Ereignis informiert wird.

Window–Listener

Ein Fenster–Ereignis wird von einem Window–Listener registriert. Ein wichtiges Ereignis, das ein Window–Listener registrieren kann, ist das Anklicken des `X` in der oberen rechten Ecke, mit der ein Fenster geschlossen wird. Ein sinnvolle Reaktion auf dieses Ereignis ist, die gesamte Anwendung zu beenden. Das passiert nicht automatisch. Wenn wir in unserem Programm nicht festlegen, dass das Programm mit seinem Fenster beendet werden soll, dann läuft es eben ohne Fenster (in der Regel sinnlos) weiter.

Im folgenden Beispiel installieren wir neben dem Knopf mit seinem Action–Listener noch einen Window–Listener der diese Action ausführt:

```
public class Gui0 extends JFrame {
  public static final int WIDTH = 400;
  public static final int HEIGHT = 300;

  public Gui0(String title) {
    super(title);
  }

  public static void main(String args[]) {
    Gui0 fenster = new Gui0("HUI, eine GUI!");
    fenster.setSize(WIDTH, HEIGHT);
    Container container = fenster.getContentPane();
    container.setLayout(new BorderLayout());

    // Window-Listener hinzufuegen:
    fenster.addWindowListener( new FensterBeobachter() );

    // Knopf mit Listener versehen und hinzufuegen:
    JButton knopf = new JButton(" Klick-Mich ");
    knopf.addActionListener( new KlickListener() );
    container.add(knopf, BorderLayout.CENTER);

    fenster.setVisible(true);
  }
}
```

Mit der Klasse des Action-Listeners des Knopfs:

```
class KlickListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Huch, ich wurde geklickt!");
  }
```

und dem Listener für das Fenster-Ereignis:

```
// Progamm-Ende wenn x angeklickt wird
class FensterBeobachter extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.out.println("OK Schluss jetzt");
    System.exit(0) ;
  }
}
```

Als Listener haben wir hier einen sogenannten *Adapter* verwendet. Der `WindowAdapter` implementiert alle notwendigen Methoden eines Window–Listeners. Wir müssen dann nur noch die uns interessierende Methode überschreiben, statt, wie bei der direkten Verwendung der Schnittstelle `WindowListener` alle Methoden (die meisten dabei mit einem leeren Rumpf).

Anonyme innere Klassen

Die Verbindung von Komponenten mit ihrem Listener erfordert die Definition vieler Klassen mit jeweils einer Methode, die auch nur an genau einer Stelle im Quelltext verwendet werden: Es wird jeweils eine einzige Instanz erzeugt, die als Listener einer Komponente agiert. Als eine Maßnahme gegen die damit verbundene Aufblähung des Quellcodes haben die Sprachdesigner von Java das Konzept der *anonymen inneren Klassen* eingeführt. Damit geht das letzte Beispiel noch etwas kompakter:

```
public class Gui0 extends JFrame {
  public static final int WIDTH = 400;
  public static final int HEIGHT = 300;

  public Gui0(String title) {
    super(title);
  }

  public static void main(String args[]) {
    Gui0 fenster = new Gui0("HUI, eine GUI!"); // ein Fenster
    fenster.setSize(WIDTH, HEIGHT);
    Container container = fenster.getContentPane();
    container.setLayout(new BorderLayout());

    fenster.addWindowListener( // Fenster mit Close-Listener versehen
           new WindowAdapter(){
             public void windowClosing(WindowEvent e) {
               System.out.println("OK Schluss jetzt");
               System.exit(0) ;
              }
           }
         );

    JButton knopf = new JButton(" Klick-Mich "); // Knopf

    knopf.addActionListener( // Knopf mit Klick-Listener versehen
           new ActionListener() {
             public void actionPerformed(ActionEvent e) {
               System.out.println("Huch, ich wurde geklickt!");
              }
           }
         );
    container.add(knopf, BorderLayout.CENTER);
    fenster.setVisible(true);
  }
}
```

Hier folgt nach `new` nicht der Name einer Listener–Klasse, sondern gleich die *gesamte Klassendefinition*. In der Klassendefinition wird nur die interessante Methode definiert. Wir merken uns dies als Muster, nach dem Listener definiert und gleich mit dem Objekt verbunden werden, das sie beobachten sollen.

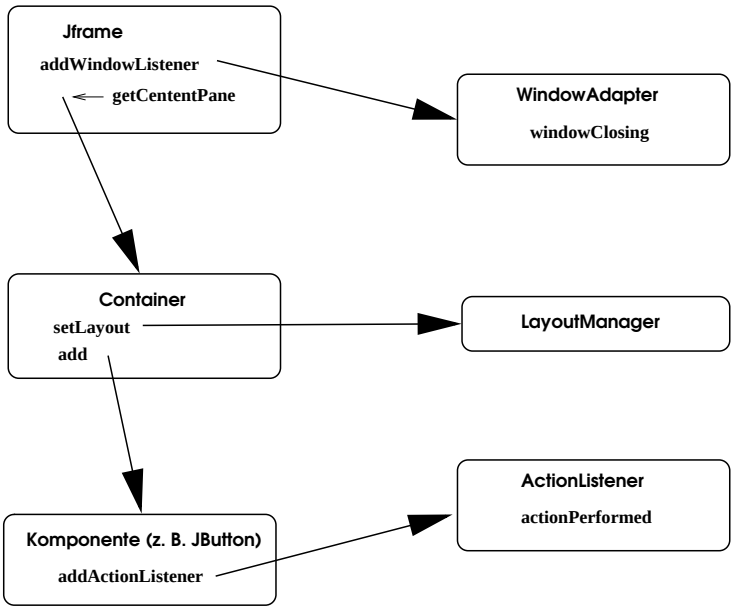


Abbildung 5.4: Zentrale Bestandteile einer GUI

5.2.5 Struktur einer GUI-Anwendung

MVC: Model Controller View

Eine GUI–Anwendung kann schnell sehr unübersichtlich werden. Es ist darum wichtig, auf eine gute Strukturierung zu achten. Ein allgemein anerkanntes und meist auch angemessenes Muster, nach dem graphische Anwendungen gestaltet werden, ist unter der Bezeichnung *Model–Controller–View* bekannt. Die Anwendung wird dabei in drei wesentliche Bestandteile aufgeteilt:

- *Model*: Die interne Darstellung des Problembereichs, das "Modell". Hier sind vor allem die Daten der Anwendung zu finden. Bei Modifikation der Daten wird ein Modell–Beobachter informiert. Typischerweise ist das die View–Komponente.
- *Control*: Die Steuerkomponente, die reagiert, wenn etwas passiert, beispielsweise wenn der Benutzer eine Aktion ausführt. Der *Controller* vermittelt zwischen den anderen Komponenten und steuert den logischen Ablauf der Anwendung. Er wird von der View–Komponente über Ereignisse informiert und passt das Modell entsprechend an.
- *View*: Die externe Darstellung, also die graphische Oberfläche. Ereignisse werden dem Controller gemeldet, das Modell kann über Modifikationen der Daten informieren und damit die Darstellung ändern.

Dieses Muster trennt den Problembereich mit seinen Daten, die Aktionen der Anwendung und die graphische Darstellung. Die Trennung macht die Anwendung insgesamt übersichtlicher und leichter änderbar.

Als Beispiel wollen wir einen einfachen Rechner betrachten, der nichts weiter kann, als zwei Zahlen zu addieren bzw. zu subtrahieren. Die Anwendung des MVC–Musters ist bei einer derart simplen Problemstellung natürlich softwaretechnischer *Overkill*, aber zur Illustration ist das Beispiel bestens geeignet.

Wir beginnen mit der graphischen Oberfläche.

AddSubView ist ein Beobachter des Modells

Die Oberfläche ist ein Objekt einer Unterklasse von `JFrame`, das auf Veränderungen der darzustellenden Daten innerhalb des Modells reagiert.

```
public class AddSubView extends JFrame{
  ... GUI Komponenten erzeugen:
  ... Zwei Eingabefelder
  ... Ein Ergebnisfeld
  ... Ein Additionsknopf
  ... Ein Subtraktionsknopf
```

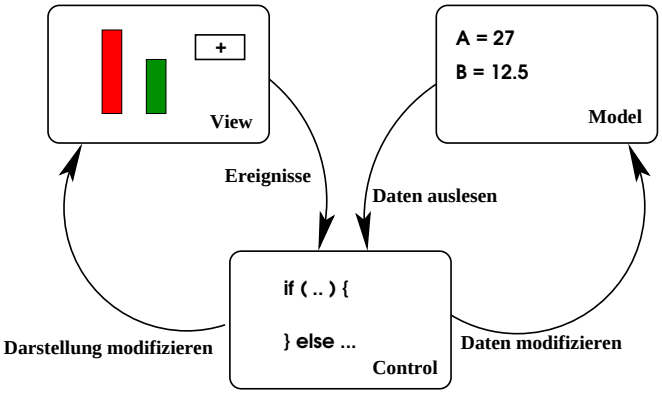


Abbildung 5.5: MVC

```
  // Neuen Wert im Ergebnisfeld darstellen
  public void nRes(String str) {
    res_txt.setText(str);
  }

}
```

Zu beobachten gibt es in diesem Fall nicht viel. Das einzige, was sich ändern kann, sind die Zahlen im Textfeld, etwa wenn das Ergebnis einer Berechnung feststeht.

Die Methode `nRes` wird vom Modell aufgerufen, wenn seine Daten sich ändern. Genauer gesagt, es wird am Ende einer Addition oder einer Subtraktion aufgerufen, um das berechnete Ergebnis darzustellen.

AddSubView baut die graphischen Komponenten auf

Die zentrale Aufgabe von `AddSubView` ist natürlich die Erzeugung der graphischen Komponenten. Das kann recht einfach erledigt werden:

```
public class AddSubView extends JFrame{
  private JButton add_button;
  private JButton sub_button;
  private JTextField op1_txt;
  private JTextField op2_txt;
  private JTextField res_txt;

  private JPanel argsPanel;
  private JPanel opsPanel;

  // Verbindung zu den anderen Komponenten der Anwendung
  private AddSubModel model;
  private AddSubController controller;

  public AddSubView(AddSubModel model, AddSubController controller) {
    super("Add/Sub MVC");
    this.model = model;
    this.controller = controller;
    viewInit();
    setListener();
    model.setObserver(this);
  }

  private void viewInit() {
    Container c = getContentPane();
    add_button = new JButton("+");
    sub_button = new JButton("-");
    op1_txt  = new JTextField(10);
```

```
    op2_txt  = new JTextField(10);
    res_txt  = new JTextField(10);
    op1_txt.setHorizontalAlignment(JTextField.RIGHT);
    op1_txt.setFont(new Font("monspaced", Font.PLAIN, 20));
    op2_txt.setHorizontalAlignment(JTextField.RIGHT);
    op2_txt.setFont(new Font("monspaced", Font.PLAIN, 20));
    res_txt.setHorizontalAlignment(JTextField.RIGHT);
    res_txt.setFont(new Font("monspaced", Font.PLAIN, 20));


    c.setLayout(new BorderLayout());
    argsPanel = new JPanel();
    opsPanel = new JPanel();
    argsPanel.setLayout(new FlowLayout());
    opsPanel.setLayout(new FlowLayout());

    argsPanel.add(op1_txt);
    argsPanel.add(op2_txt);

    opsPanel.add(add_button);
    opsPanel.add(sub_button);

    c.add(argsPanel, BorderLayout.NORTH);
    c.add(opsPanel, BorderLayout.CENTER);
    c.add(res_txt, BorderLayout.SOUTH);
    pack();
  }

  private void setListener(){ ... }

  public void nRes(String str) { ... }

}
```

Zum Aufbau der Oberfläche muss nichts mehr angemerkt werden. Der Konstruktor dient dazu, diese View–Komponente mit ihren Partnern zu verbinden.

Graphische Komponenten bekommen ihre Listener

Jede graphische Komponente, die ein Ereignis erzeugen kann, benötigt einen Listener, der dieses Ereignis verarbeitet. Wir müssen also allen, außer den Labels, einen Listener zuordnen. Der Listener leitet in unserem Fall die Information über ein Ereignis an den Controller weiter, indem eine von dessen Methoden aufgerufen wird:

```
public class AddSubView extends JFrame{
  ....
  /**
   * Installation der Listener: Ereignisse melden.
   * Die graphischen Komponenten werden mit ihren Listenern (Handlern) verbunden.
   * Alle Ereignisse werden dem Controller gemeldet, durch Aufruf
   * entsprechender Methoden.
   */
 private void setListener(){
    add_button.addActionListener(
        new ActionListener(){
          public void actionPerformed(ActionEvent e) {
            controller.action('+', op1_txt.getText(),
                        op2_txt.getText());
          }
        }
      );

    sub_button.addActionListener(
        new ActionListener(){
          public void actionPerformed(ActionEvent e) {
            controller.action('-', op1_txt.getText(),
                        op2_txt.getText());
          }
```

```
        }
      );
    this.addWindowListener(
        new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
            System.exit(0) ;
          }
        }
      );

  }
   ....
}
```

Wird beispielsweise der Additionsknopf `add_button` angeklickt, dann wird die Methode `action` des Controllers mit dem Argument + und den Werten der Eingabetextfelder aktiviert.

Das Modell

Das Modell ist bei diesem Beispiel ganz besonders einfach. Es verwaltet drei Variablen `wert1`, `wert2` und `res`. Die beiden ersten, `wert1`, `wert2` spiegeln den Wert der Textfelder wider. Die Variable `res` enthält das Ergebnis der zuletzt ausgeführten Operation. Bei jeder Änderung seines Wertes wird das Modell informiert und der dargestellte an den berechneten Wert angepasst.

```
public class AddSubModel {
  private int wert1;
  private int wert2;
  private int res;

  AddSubView observer; // das Modell, das die Werte wiederzugeben hat.

  public void setObserver(AddSubView observer){
    this.observer = observer;
  }

  public void add(int i, int j) {
    // internes Modell entsprechend der Benutzer-Eingaben anpassen
    wert1 = i;
    wert2 = j;
    res = wert1 + wert2;
    // Anpassung der Darstellung an modifiziertes Modell veranlassen
    observer.nRes(""+res);
  }

  public void sub(int i, int j) {
    wert1 = i;
    wert2 = j;
    res = wert1 - wert2;
    observer.nRes(""+res);
  }

}
```

Der Controller

Der Controller hat in unserem kleinen Beispiel auch nicht sehr viel zu tun. Die vom *View* festgestellten Ereignisse kommen hier als Methodenaufrufe an. Es müssen nur noch die entsprechenden Aktionen im Modell angestoßen werden:

```
public class AddSubController {

  private AddSubModel model;

  public AddSubController(AddSubModel model) {
    this.model = model;
  }
```

```
  public void action(char c, String str1, String str2) {
    if ( c == '+') model.add(Integer.parseInt(str1), Integer.parseInt(str2));
    if ( c == '-') model.sub(Integer.parseInt(str1), Integer.parseInt(str2));
  }
}
```

Alles zusammenbauen

Es bleibt nur noch die Aufgabe, alle Komponenten zu erzeugen und miteinander zu verknüpfen. Das macht die Hauptklasse `AddSub` in ihrer `main`–Funktion:

```
public class AddSub {
   public static void main(String[] args) {
      AddSubModel    model    = new AddSubModel();
      AddSubController controller = new AddSubController(model);
      AddSubView     view     = new AddSubView(model, controller);
      view.setVisible(true);
   }
}
```

Zum Zusammenbau gehört auch die Verknüpfung der Komponenten untereinander. Das Modell muss seinen View kennen:

```
public class AddSubModel {
  ...
  AddSubView observer;

  public void setObserver(AddSubView observer){
    this.observer = observer;
  }
  ....
}
```

Der Controller muss den View kennen:

```
public class AddSubController {

  private AddSubModel model;

  public AddSubController(AddSubModel model) {
    this.model = model;
  }
  ...
}
```

Ein `AddSubView` muss mit dem Controller verbunden werden und im Modell einer Bezug auf sich selbst setzen:

```
public class AddSubView extends JFrame{
  ....
  private AddSubModel model;
  private AddSubController controller;

  public AddSubView(AddSubModel model, AddSubController controller) {
    ....
    this.model  = model;
    this.controller = controller;
    model.setObserver(this);
    ...
  }
  ...
}
```

Damit beenden wir unsere Ausführungen zu graphischen Oberflächen. Als Kochrezept für sehr einfache GUIs wird das ausreichen. Eine gründliche Beschäftigung mit der Materie setzt eine etwas tiefergehende Kenntnis der Sprachkonstrukte von Java voraus und erfordert vor allem eine eingehende Diskussion einer Vielzahl von Klassen, Schnittstellen, etc. , eine Diskussion mit der man problemlos dicke Bücher füllen kann (und gefüllt hat).

Index