Objektorientierte Softwareentwicklung

(Entwurf)

WS 10/11

Sven Eric Panitz

Hochschule RheinMain

Version 29. November 2010

Die vorliegende Fassung des Skriptes hat mitlerweile schon eine kleine Geschichte hinter sich: es startete als Skript für die Programmiervorlesungen an der TFH-Berlin im Studiengang Medieninformatik im WS02/03 als dreiteilige Vorlesung. Im SS05 sind diese drei Teile zu einem Skript zusammengeführt worden, das speziell für die Vorlesung Java des 3. Semester im Studiengang allgemeine Informatik an der FH Wiesbaden ausgerichtet war. Dabei wurde versucht auf die Vorkenntnisse der Studenten aus den Vorlesungen C und C++ aufzubauen.

Mit dem WS05/06 habe ich mich entschieden, das Skript wieder als allgemeine Einführung in die Programmierung mit Java zu gestalten und somit ein monolythisches Skript, das für sich stehen kann, zu erhalten.

Schließlich mit dem WS10/11 wird es im Laufe des Semester angepasst auf das neue Modul ≫Objektorientierte Softwareentwicklung≪, das wiederum in Inhalt und Umfang sehr nah bei der ursprünglichen Berliner Vorlesung aus dem Jahre 2002 liegt.

Trotz der nun schon etwas längeren Historie des Skripts, ist es naturgemäß in Aufbau und Qualität nicht mit einem Buch vergleichbar. Flüchtigkeits- und Tippfehler werden sich im Eifer des Gefechtes nicht vermeiden lassen und wahrscheinlich in nicht geringem Maße auftreten. Ich bin natürlich stets dankbar, wenn ich auf solche aufmerksam gemacht werde und diese korrigieren kann.

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird. Beispielprogramme werden direkt aus dem Skriptquelltext extrahiert und sind auf der Webseite herunterzuladen.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Ziel der Vorlesung

Diese Vorlesung setzt sich zum Ziel, die Grundlagen der Programmierung zu vermitteln. Hierzu gehören insbesondere gängige Algorithmen und Programmiermuster sowie die gebräuchlichsten Datentypen. Die verwendete Programmiersprache ist aus pragmatischen Gründen *Java*. Die Vorlesung will aber kein reiner Javakurs sein, sondern ermöglichen, die gelernten Konzepte schnell auch in anderen Programmiersprachen umzusetzen. Programmierung wird nicht allein als die eigentliche Codierung des Programms verstanden, sondern Tätigkeiten wie Spezifikation, Modellierung, Testen etc. werden als Teildisziplinen der Programmierung verstanden.

Desweiteren soll die Vorlesung mit der allgemeinen Terminologie der Informatik vertraut machen.

1.2 Programmieren

1.2.1 Disziplinen der Programmierung

Mit dem Begriff Programmierung wird zunächst die eigentliche Codierung eines Programms assoziiert. Eine genauerer Blick offenbart jedoch, dass dieses nur ein kleiner Teil von vielen recht unterschiedlichen Schritten ist, die zur Erstellung von Software notwendig sind:

- **Spezifikation:** Bevor eine Programmieraufgabe bewerkstelligt werden kann, muss das zu lösende Problem spezifiziert werden. Dieses kann informell durch eine natürlichsprachliche Beschreibung bis hin zu mathematisch beschriebenen Funktionen geschehen. Gegen die Spezifikation wird programmiert. Sie beschreibt das gewünschte Verhalten des zu erstellenden Programms.
- **Modellieren:** Bevor es an die eigentliche Codierung geht, wird in der Regel die Struktur des Programms modelliert. Auch dieses kann in unterschiedlichen Detaillierungsgraden geschehen. Manchmal reichen Karteikarten als hilfreiches Mittel aus, andernfalls empfiehlt sich eine umfangreiche Modellierung mit Hilfe rechnergestützter Werkzeuge. Für die objektorientierte Programmierung hat sich UML als eine geeignete Modellierungssprache durchgesetzt. In ihr lassen sich Klassendiagramme, Klassenhierarchien und Abhängigkeiten graphisch darstellen. Mit bestimmten Werkzeugen wie *Together* oder *Rational Rose* läßt sich direkt für eine UML-Modellierung Programmtext generieren.
- **Codieren:** Die eigentliche Codierung ist in der Regel der einzige Schritt, der direkt Code in der gewünschten Programmiersprache von Hand erzeugt. Alle anderen Schritte der Programmierung sind mehr oder weniger unabhängig von der zugrundeliegenden Programmiersprache.

Für die Codierung empfiehlt es sich, Konventionen zu verabreden, wie der Code geschrieben wird, was für Bezeichner benutzt werden, in welcher Weise der Programmtext eingerückt wird. Entwicklungsabteilungen haben zumeist schriftlich verbindlich festgeschriebene Richtlinien für den Programmierstil. Dieses erleichtert, den Code der Kollegen im Projekt schnell zu verstehen.

- **Testen:** Beim Testen sind generell zu unterscheiden:
  - *Entwicklertests:* Diese werden von den Entwicklern während der Programmierung selbst geschrieben, um einzelne Programmteile (Methoden, Funktionen) separat zu testen. Es gibt eine Schule, die propagiert, Entwicklertests vor dem Code zu schreiben (*test first*). Die Tests dienen in diesem Fall als kleine Spezifikationen.
  - *Qualitätssicherung:* In der Qualitätssicherung werden die fertigen Programme gegen ihre Spezifikation getestet (*black box tests*). Hierzu werden in der Regel automatisierte Testläufe geschrieben. Die Qualitätssicherung ist personell von der Entwicklung getrennt. Es kann in der Praxis durchaus vorkommen, dass die Qualitätsabteilung mehr Mitarbeiter hat als die Entwicklungsabteilung.
- **Optimieren:** Sollten sich bei Tests oder in der Praxis Performanzprobleme zeigen, sei es durch zu hohen Speicherverbrauch als auch durch zu lange Ausführungszeiten, so wird versucht, ein Programm zu optimieren. Hierzu bedient man sich spezieller Werkzeuge (*profiler*), die für einen Programmdurchlauf ein Raum- und Zeitprofil erstellen. In diesem Profil können Programmteile, die besonders häufig durchlaufen werden, oder Objekte, die im großen Maße Speicher belegen, identifiziert werden. Mit diesen Informationen lassen sich gezielt inperformante Programmteile optimieren.
- **Verifizieren:** Eine formale Verifikation eines Programms ist ein mathematischer Beweis der Korrektheit bezüglich der Spezifikation. Das setzt natürlich voraus, dass die Spezifikation auch formal vorliegt. Man unterscheidet:
  - *partielle Korrektheit:* wenn das Programm für eine bestimte Eingabe ein Ergebnis liefert, dann ist dieses bezüglich der Spezifikation korrekt.
  - *totale Korrektheit:* Das Programm ist partiell korrekt und terminiert für jede Eingabe, d.h. liefert immer nach endlich langer Zeit ein Ergebnis.

  Eine formale Verifikation ist notorisch schwierig und allgemein nicht automatisch durchführbar. In der Praxis werden nur in ganz speziellen kritischen Anwendungen formale Verifikationen durchgeführt, z.B. bei Steuerungen gefahrenträchtiger Maschinen, so dass Menschenleben von der Korrektheit eines Programms abhängen können.
- **Wartung/Pflege:** den größten Teil seiner Zeit verbringt ein Programmierer nicht mit der Entwicklung neuer Software, sondern mit der Wartung bestehender Software. Hierzu gehören die Anpassung des Programms an neue Versionen benutzter Bibliotheken oder des Betriebssystems, auf dem das Programm läuft, sowie die Korrektur von Fehlern.
- **Debuggen:** Bei einem Programm ist immer damit zu rechnen, dass es Fehler enthält. Diese Fehler werden im besten Fall von der Qualitätssicherung entdeckt, im schlechteren Fall treten sie beim Kunden auf. Um Fehler im Programmtext zu finden, gibt es Werkzeuge, die ein schrittweises Ausführen des Programms ermöglichen *(debugger)*. Dabei lassen sich die Werte, die in bestimmten Speicherzellen stehen, auslesen und auf diese Weise der Fehler finden.
- **Internationalisieren (I18N)[1]:** Softwarefirmen wollen möglichst viel Geld mit ihrer Software verdienen und streben deshalb an, ihre Programme möglichst weltweit zu vertreiben. Hierzu muss gewährleistet sein, dass das Programm auf weltweit allen Plattformen läuft

[1] I18N ist eine Abkürzung für das Wort *internationalization*, das mit einem i beginnt, mit einem n endet und dazwischen 18 Buchstaben hat.

und mit verschiedenen Schriften und Textcodierungen umgehen kann. Das Programm sollte ebenso wie mit lateinischer Schrift auch mit Dokumenten in anderen Schriften umgehen können. Fremdländische Akzente und deutsche Umlaute sollten bearbeitbar sein. Aber auch unterschiedliche Tastaturbelegungen bis hin zu unterschiedlichen Schreibrichtungen sollten unterstützt werden.

Die Internationalisierung ist ein weites Feld, und wenn nicht am Anfang der Programmerstellung hierauf Rücksicht genommen wird, so ist es schwer, nachträglich das Programm zu internationalisieren.

- **Lokalisieren (L12N):** Ebenso wie die Internationalisierung beschäftigt sich die Lokalisierung damit, dass ein Programm in anderen Ländern eingesetzt werden kann. Beschäftigt sich die Internationalisierung damit, dass fremde Dokumente bearbeitet werden können, versucht die Lokalisierung, das Programm komplett für die fremde Sprache zu übersetzen. Hierzu gehören Menueinträge in der fremden Sprache, Beschriftungen der Schaltflächen oder auch Fehlermeldungen in fremder Sprache und Schrift. Insbesondere haben verschiedene Schriften unterschiedlichen Platzbedarf; auch das ist beim Erstellen der Programmoberfläche zu berücksichtigen.
- **Portieren:** Oft wird es nötig, ein Programm auf eine andere Plattform zu portieren. Ein unter Windows erstelltes Programm soll z.B. auch auf Unix-Systemen zur Verfügung stehen.
- **Dokumentieren:** Der Programmtext allein reicht in der Regel nicht aus, damit das Programm von Fremden oder dem Programmierer selbst nach geraumer Zeit gut verstanden werden kann. Um ein Programm näher zu erklären, wird im Programmtext Kommentar eingefügt. Kommentare erklären die benutzten Algorithmen, die Bedeutung bestimmter Datenfelder oder die Schnittstellen und Benutzung bestimmter Methoden.

  Es ist zu empfehlen, sich anzugewöhnen, Quelltextdokumentation immer auf Englisch zu schreiben. Es ist oft nicht abzusehen, wer einmal einen Programmtext zu sehen bekommt. Vielleicht ein japanischer Kollege, der das Programm für Japan lokalisiert, oder der irische Kollege, der, nachdem die Firma mit einer anderen Firma fusionierte, das Programm auf ein anderes Betriebssystem portiert, oder vielleicht die englische Werksstudentin, die für ein Jahr in der Firma arbeitet.

1.2.2 Was ist ein Programm

Die Frage danach, was ein Programm eigentlich ist, läßt sich aus verschiedenen Perspektiven recht unterschiedlich beantworten.

pragmatische Antwort

Eine Textdatei, die durch ein anderes Programm in einen ausführbaren Maschinencode übersetzt wird. Dieses andere Programm ist ein Übersetzer, engl. *compiler*.

mathematische Antwort

Eine Funktion, die deterministisch für Eingabewerte einen Ausgabewert berechnet.

sprachwissenschaftliche Antwort

Ein Satz einer durch eine Grammatik beschriebenen Sprache mit einer operationalen Semantik.

operationale Antwort

Eine Folge von durch den Computer ausführbaren Befehlen, die den Speicher des Computers manipulieren.

1.2.3 Klassifizierung von Programmiersprachen

Es gibt mittlerweile mehr Programmiersprachen als natürliche Sprachen.[2] Die meisten Sprachen führen entsprechend nur ein Schattendasein und die Mehrzahl der Programme konzentriert sich auf einige wenige Sprachen. Programmiersprachen lassen sich nach den unterschiedlichsten Kriterien klassifizieren.

Hauptklassen

Im folgenden eine hilfreiche Klassifizierung in fünf verschiedene Hauptklassen.

- **imperativ** (C, Pascal, Fortran, Cobol): das Hauptkonstrukt dieser Sprachen sind Befehle, die den Speicher manipulieren.
- **objektorientiert** (Java, C++, C#, Eiffel, Smalltalk): Daten werden in Form von Objekten organisiert. Diese Objekte bündeln mit den Daten auch die auf diesen Daten anwendbaren Methoden.
- **funktional** (Lisp, ML, Haskell, Scheme, Erlang, Clean): Programme werden als mathematische Funktionen verstanden und auch Funktionen können Daten sein. Dieses Programmierparadigma versucht, sich möglichst weit von der Architektur des Computers zu lösen. Veränderbare Speicherzellen gibt es in rein funktionalen Sprachen nicht und erst recht keine Zuweisungsbefehle.
- **Skriptsprachen** (Perl, AWK): solche Sprachen sind dazu entworfen, einfache kleine Programme schnell zu erzeugen. Sie haben meist kein Typsystem und nur eine begrenzte Zahl an Strukturierungsmöglichkeiten, oft aber eine mächtige Bibliothek, um Zeichenketten zu manipulieren.
- **logisch** (Prolog): aus der KI (künstlichen Intelligenz) stammen logische Programmiersprachen. Hier wird ein Programm als logische Formel, für die ein Beweis gesucht wird, verstanden.

Ausführungsmodelle

Der Programmierer schreibt den lesbaren Quelltext seines Programmes. Um ein Programm auf einem Computer laufen zu lassen, muss es erst in einen Programmcode übersetzt werden, den der Computer versteht. Für diesen Schritt gibt es auch unterschiedliche Modelle:

- **kompiliert** (C, Cobol, Fortran): in einem Übersetzungsschritt wird aus dem Quelltext direkt das ausführbare Programm erzeugt, das dann unabhängig von irgendwelchen Hilfen der Programmiersprache ausgeführt werden kann.
- **interpretiert** (Lisp, Scheme): der Programmtext wird nicht in eine ausführbare Datei übersetzt, sondern durch einen Interpreter Stück für Stück anhand des Quelltextes ausgeführt. Hierzu muss stets der Interpreter zur Verfügung stehen, um das Programmm auszuführen. Interpretierte Programme sind langsamer in der Ausführung als übersetzte Programme.

[2] Wer Interesse hat, kann im Netz einmal suchen, ob er eine Liste von Programmiersprachen findet.

- **abstrakte Maschine über *byte code*** (Java, ML): dieses ist quasi eine Mischform aus den obigen zwei Ausführungsmodellen. Der Quelltext wird übersetzt in Befehle nicht für einen konkreten Computer, sondern für eine abstrakte Maschine. Für diese abstrakte Maschine steht dann ein Interpreter zur Verfügung. Der Vorteil ist, dass durch die zusätzliche Abstraktionsebene der Übersetzer unabhängig von einer konkreten Maschine Code erzeugen kann und das Programm auf auf allen Systemen laufen kann, für die es einen Interpreter der abstrakten Maschine gibt.

Es gibt Programmiersprachen, für die sowohl Interpreter als auch Übersetzer zur Verfügung stehen. In diesem Fall wird der Interpreter gerne zur Programmentwicklung benutzt und der Übersetzer erst, wenn das Programm fertig entwickelt ist.

Übersetzung und Ausführung von Javaprogrammen Da Java sich einer abstrakten Maschine bedient, sind, um zur Ausführung zu gelangen, sowohl ein Übersetzer als auch ein Interpreter notwendig. Der zu übersetzende Quelltext steht in Dateien mit der Endung `.java`, der erzeugte *byte code* in Dateien mit der Endung `.class`

Der Javaübersetzer kann von der Kommandozeile mit dem Befehl `javac` aufgerufen werden. Um eine Programmdatei `Test.java` zu übersetzen, kann folgendes Kommando eingegeben werden:

```
1 javac Test.java
```

Im Falle einer fehlerfreien Übersetzung wird eine Datei `Test.class` im Dateisystem erzeugt. Dieses erzeugte Programm wird allgemein durch folgendes Kommando im Javainterpreter ausgeführt:

```
1 java Test
```

Übersetzen eines erste Javaprogramms

Um ein lauffähiges Javaprogramm zu schreiben, ist es notwendig, eine ganze Reihe von Konzepten Javas zu kennen, die nicht eigentliche Kernkonzepte der objektorientierten Programmierung sind. Daher geben wir hier ein minimales Programm an, dass eine Ausgabe auf den Bildschirm macht:

FirstProgram.java

```
1 class FirstProgram{
2   public static void main(String [] args){
3     System.out.println("hello world");
4   }
5 }
```

In den kommenden Wochen werden wir nach und nach die einzelne Bestandteile dieses Minimalprogrammes zu verstehen lernen.

Aufgabe 1 Öffnen Sie die Kommandozeile ihres Betriebssystems. Machen Sie sich mit rudimentären Befehlen der Kommandozeile vertraut. Erzeugen Sie mit dem Befehl `mkdir` einen neuen Ordner. Wechseln Sie mit `cd` in diesen Ordner.

Aufgabe 2 Schreiben Sie das Programm `FirstProgram.java` des Skriptes mit einem Texteditor ihrer Wahl. Speichern Sie es als `FirstProgram.java` in dem auf der Kommandozeile neu erstellten Ordner ab.

Aufgabe 3 Lassen Sie sich das Programm, in der Kommandozeile mit dem Befehl `more` einmal anzeigen.

Aufgabe 4 Übersetzen Sie Das Programm auf der Kommandozeile mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `FirstProgram.class`. Lassen Sie sich mit dem Befehl `ls -l` anzeigen, ob die Datei `FirstProgram.class` tatsächlich erzeugt wurde.

Führen Sie nun das Programm mit dem Javainterpreter `java` aus.

Sofern Ihnen unterschiedliche Betriebssysteme zur Verfügung stehen, führen Sie dieses sowohl einmal auf Linux als auch einmal unter Windows durch.

Wir können in der Folge diesen Programmrumpf benutzen, um beliebige Objekte auf den Bildschirm auszugeben. Hierzu werden wir das `"hello world"` durch andere Ausdrücke ersetzen.

Wollen Sie z.B. eine Zahl auf dem Bildschirm ausgeben, so ersetzen Sie den in Anführungszeichen eingeschlossenen Ausdruck durch diese Zahl:

Answer.java
```
1 class Answer {
2   public static void main(String [] args){
3     System.out.println(42);
4   }
5 }
```

Aufgabe 5 Experimentieren Sie ein wenig mit den ersten kleinen Javaprogramm herum.

Aufgabe 6 Von ihrem heimischen Rechner loggen Sie sich mit dem Befehl `ssh` von daheim auf den Server an der Hochschule auf der Kommandozeile ein. Dafür müssen Sie sich mit Ihrer Informatik-Benutzerkennung einloggen:
`ssh ihreKennung@login1.cs.hs-rm.de`

Starten Sie nun das Programm mit dem Javainterpreter auf dem Server.

Aufgabe 7 Loggen Sie sich von zu Hause mit dem Befehl `sftp` auf dem Server `login1.cs.hs-rm.de` ein. Navigieren Sie dann mit dem Befehl `cd` in den Ordner, in dem Ihr Programm liegt. Holen Sie es dann mit dem Befehl `get FirstProgram.java` auf Ihren heimischen Rechner.

1.2.4 Arbeitshypothese

Bevor im nächsten Kapitel mit der eigentlichen objektorientierten Programmierung begonnen wird, wollen wir für den weiteren Verlauf der Vorlesung eine Arbeitshypothese aufstelllen:

> Beim Programmieren versuchen wir, zwischen Daten und Programmen zu unterscheiden. Programme manipulieren Daten, indem sie sie löschen, anlegen oder überschreiben.
>
> Daten können dabei einfache Datentypen sein, die Zahlen, Buchstaben oder Buchstabenketten (Strings) repräsentieren, oder aber beliebig strukturierte Sammlungen von Daten wie z.B. Listen, Tabellen, Baum- oder Graphstrukturen.

Wir werden im Laufe der Vorlesung immer wieder zu prüfen haben, ob diese starke Trennung zwischen Daten und Programmen gerechtfertigt ist.

Kapitel 2

Objektorientierte Programmierung

2.1 Grundkonzepte der Objektorientierung

2.1.1 Objekte und Klassen

Die Grundidee der objektorientierten Programmierung ist, Daten, die zusammen ein größeres zusammenhängendes Objekt beschreiben, zusammenzufassen. Zusätzlich fassen wir mit diesen Daten noch die Programmteile zusammen, die diese Daten manipulieren. Ein Objekt enthält also nicht nur die reinen Daten, die es repräsentiert, sondern auch Programmteile, die Operationen auf diesen Daten durchführen. Insofern wäre vielleicht *subjektorientierte Programmierung* ein passenderer Ausdruck, denn die Objekte sind nicht passive Daten, die von außen manipuliert werden, sondern enthalten selbst als integralen Bestandteil Methoden, die ihre Daten manipulieren können.

Objektorientierte Modellierung

Bevor wir etwas in Code gießen, wollen wir ersteinmal eine informelle Modellierung der Welt, für die ein Programm geschrieben werden soll, vornehmen. Hierzu empfiehlt es sich durchaus, in einem Team zusammenzusitzen und auf Karteikarten aufzuschreiben, was es denn für Objekte in der Welt gibt, die wir modellieren wollen.

Stellen wir uns hierzu einmal vor, wir sollen ein Programm zur Bibliotheksverwaltung schreiben. Jetzt überlegen wir einmal, was gibt es denn für Objektarten, die alle zu den Vorgängen in einer Bibliothek gehören. Hierzu fällt uns vielleicht folgende Liste ein:

- Personen, die Bücher ausleihen wollen.
- Bücher, die ausgeliehen werden können.
- Tatsächliche Ausleihvorgänge, die ausdrücken, dass ein Buch bis zu einem bestimmten Zeitraum von jemanden ausgeliehen wurde.
- Termine, also Objekte, die ein bestimmtes Datum kennzeichnen.

Nachdem wir uns auf diese vier für unsere Anwendung wichtigen Objektarten geinigt haben, nehmen wir vier Karteikarten und schreiben jeweils eine der Objektarten als Überschrift auf diese Karteikarten.

Jetzt haben wir also Objektarten identifiziert. Im nächsten Schritt ist zu überlegen, was für Eigenschaften diese Objekte haben. Beginnen wir für die Karteikarte, auf der wir als Überschrift *Person* geschrieben haben. Was interessiert uns an Eigenschaften einer Person? Wahrscheinlich ihr Name mit Vornamen, Straße und Ort sowie Postleitzahl. Das sollten die Eigenschaften einer Person sein, die für ein Bibliotheksprogramm notwendig sind. Andere mögliche Eigenschaften wie Geschlecht, Alter, Beruf oder ähnliches interessieren uns in diesem Kontext nicht. Jetzt

schreiben wir die Eigenschaften, die uns von einer Person interessieren, auf die Karteikarte mit der Überschrift Person.

Schließlich müssen wir uns Gedanken darüber machen, was diese Eigenschaften eigentlich für Daten sind. Name, Vorname, Straße und Wohnort sind sicherlich als Texte abzuspeichern oder, wie der Informatiker gerne sagt, als Zeichenketten. Die Postleitzahl ist hingegen als eine Zahl abzuspeichern. Diese Art, von der die einzelnen Eigenschaften sind, nennen wir ihren Typ. Wir schreiben auf die Karteikarte für die Objektart *Person* vor jede der Eigenschaften noch den Typ, den diese Eigenschaft hat.[1] Damit erhalten wir für die Objektart *Person* die in Abbildung 2.1 gezeigte Karteikarte.

Abbildung 2.1: Modellierung einer Person.

Gleiches können wir für die Objektart *Buch* und für die Objektart *Datum* machen. Wir erhalten dann eventuell die Karteikarten aus Abbildung 2.2 und 2.3 .

Wir müssen uns schließlich nur noch um die Objektart einer Buchausleihe kümmern. Hier sind drei Eigenschaften interessant: wer hat das Buch geliehen, welches Buch wurde verliehen und wann muss es zurückgegeben werden. Wir können also drei Eigenschaften auf die Karteikarte schreiben. Was sind die Typen dieser drei Eigenschaften? Diesmal sind es keine Zahlen oder Zeichenketten, sondern Objekte der anderen drei bereits modellierten Objektarten. Wenn wir nämlich eine Karteikarte schreiben, dann erfinden wir gerade einen neuen Typ, den wir für die Eigenschaften anderer Karteikarten benutzen können.

Somit erstellen wir eine Karteikarte für den Objekttyp *Ausleihe*, wie sie in Abbildung 2.4 zu sehen ist.

Klassen in Java

Wir haben in einem Modellierungsschritt im letzten Abschnitt verschiedene Objektarten identifiziert und ihre Eigenschaften spezifiziert. Dazu haben wir vier Karteikarten geschrieben. Jetzt können wir versuchen, diese Modellierung in Java umzusetzen. In Java beschreibt eine Klasse eine Menge von Objekten gleicher Art. Damit entspricht eine Klasse einer der Karteikarten in unsrer Modellierung. Die Klassendefinition ist eine Beschreibung der möglichen Objekte. In ihr ist definiert, was für Daten zu den Objekten gehören. Zusätzlich können wir in einer Klasse

[1] Es mag vielleicht verwundern, warum wir den Typ vor die Eigenschaft und nicht etwa hinter sie schreiben. Dieses ist eine sehr alte Tradition in der Informatik.

Abbildung 2.2: Modellierung eines Buches.

Abbildung 2.3: Modellierung eines Datums.

noch schreiben, welche Operationen auf diesen Daten angewendet werden können. Klassendefinitionen sind die eigentlichen Programmtexte, die der Programmierer schreibt.

> **In Java steht genau eine Klassendefinition[2] in genau einer Datei. Die Datei hat dabei den Namen der Klasse mit der Endung `.java`.**

In Java wird eine Klasse durch das Schlüsselwort `class`, gefolgt von dem Namen, den man für die Klasse gewählt hat, deklariert. Anschließend folgt in geschweiften Klammern der Inhalt der Klasse bestehend aus Felddefinitionen und Methodendefinitionen.

Die einfachste Klasse, die in Java denkbar ist, ist eine Klasse ohne Felder oder Methoden:

[2] Auch hier werden wir Ausnahmen kennenlernen.

Abbildung 2.4: Modellierung eines Ausleihvorgangs.

Minimal.java

```
1 class Minimal {
2 }
```

Beachten Sie, dass Groß- und Kleinschreibung in Java relevant ist. Alle Schlüsselwörter wie `class` werden stets klein geschrieben. Klassennamen starten per Konvention immer mit einem Großbuchstaben.

Die Stringklasse Java kommt bereits mit einer großen Anzahl zur Verfügung stehender Standardklassen. Es müssen also nicht alle Klassen neu vom Programmierer definiert werden. Eine sehr häufig benutzte Klasse ist die Klasse `String`. Sie repräsentiert Objekte, die eine Zeichenkette darstellen, also einen Text, wie wir ihn in unserer ersten Modellierung bereits vorausgesetzt haben.

Objekte

Eine Klasse deklariert, wie die Objekte, die die Klasse beschreibt, aussehen können. Um konkrete Objekte für eine Klasse zu bekommen, müssen diese irgendwann im Programm einmal erzeugt werden. Dieses wird in Java syntaktisch gemacht, indem einem Schlüsselwort `new` der Klassenname mit einer in runden Klammern eingeschlossenen Argumentliste folgt. Ein Objekt der obigen minimalen Javaklasse läßt sich entsprechend durch folgenden Ausdruck erzeugen: `new Minimal();`

Wenn ein Objekt eine Instanz einer bestimmten Klasse ist, spricht man auch davon, dass das Objekt den Typ dieser Klasse hat.

Objekte der Stringklasse Für die Klasse `String` gibt es eine besondere Art, Objekte zu erzeugen. Ein in Anführungsstrichen eingeschlossener Text erzeugt ein Objekt der Klasse `String`.

Aus zwei Objekten der Stringklasse läßt sich ein neues Objekt erzeugen, indem diese beiden Objekte mit einem Pluszeichen verbunden werden:

```
1 "hallo "+"welt"
```

Hier werden die zwei Stringobjekte `"hallo "` und `"welt"` zum neuen Objekt `"hallo welt"` verknüpft.

2.1.2 Felder und Methoden

Im obigen Abschnitt haben wir gesehen, wie eine Klasse definiert wird und Objekte einer Klasse erzeugt werden können. Allerdings war unsere erste Klasse noch vollkommen ohne Inhalt: es gab weder die Möglichkeit, Daten zu speichern, noch wurden Programmteile definiert, die hätten ausgeführt werden können. Hierzu können Felder und Methoden deklariert werden. Die Gesamtheit der Felder und Methoden nennt man mitunter auch *Features* oder, um einen griffigen deutschen Ausdruck zu verwenden, *Eigenschaften*.

Felder

Zum Speichern von Daten können Felder für eine Klasse definiert werden. In einem Feld können Objekte für eine bestimmte Klasse gespeichert werden. Bei der Felddeklaration wird angegeben, welche Art von Objekten in einem Feld abgespeichert werden sollen. Die Felder entsprechen dabei genau den Eigenschaften, die wir auf unsere Karteikarten geschrieben haben.

Syntaktisch wird in Java der Klassenname des Typs, von dem Objekte gespeichert werden sollen, den frei zu wählenden Feldnamen vorangestellt. Eine Felddeklaration endet mit einem Semikolon.

Beispiel 2.1.1 *Im Folgenden schreiben wir eine Klasse mit zwei Feldern:*

ErsteFelder.java

```
1 class ErsteFelder {
2   Minimal meinFeld;
3   String  meinTextFeld;
4 }
```

Das erste Feld soll dabei einmal ein Objekt unserer minimalen Klasse sein und das andere Mal eine Zeichenkette.

Feldnamen werden per Konvention immer klein geschrieben.

Zuweisung In Feldern können Objekte gespeichert werden. Hierzu muss dem Feld ein Objekt zugewiesen werden. Syntaktisch geschieht dieses durch ein Gleichheitszeichen, auf dessen linker Seite das Feld steht und auf dessen rechter Seite das Objekt, das in diesem Feld gespeichert werden soll. Auch Zuweisungen enden mit einem Semikolon.

Beispiel 2.1.2 *In der folgenden Klasse definieren wir nicht nur, dass die Objekte der Klasse zwei Felder eines bestimmten Typs haben, sondern weisen auch gleich schon Werte, d.h. konkrete Daten diesen Feldern zu.*

ErsteFelder2.java

```
1 class ErsteFelder2 {
2   Minimal meinFeld = new Minimal();
3   String  meinTextFeld = "hallo";
4 }
```

Nach einer Zuweisung repräsentiert das Feld das ihm zugewiesene Objekt.

Methoden

Methoden[3] sind die Programmteile, die in einer Klasse definiert sind und für jedes Objekt dieser Klasse zur Verfügung stehen. Die Ausführung einer Methode liefert meist ein Ergebnisobjekt. Methoden haben eine Liste von Eingabeparametern. Ein Eingabeparameter ist durch den gewünschten Klassennamen und einen frei wählbaren Parameternamen spezifiziert.

Methodendeklaration In Java wird eine Methode deklariert durch: den Rückgabetyp, den Namen der Methode, der in Klammern eingeschlossenen durch Kommas getrennten Parameterliste und den in geschweiften Klammern eingeschlossenen Programmrumpf. Im Programmrumpf wird mit dem Schlüsselwort `return` angegeben, welches Ergebnisobjekt die Methode liefert.

Beispiel 2.1.3 *Als Beispiel definieren wir eine Klasse, in der es eine Methode* `addString` *gibt, die den Ergebnistyp* `String` *und zwei Parameter vom Typ* `String` *hat:*

StringUtilMethod.java

```
1 class StringUtilMethod {
2   String addStrings(String leftText, String rightText){
3     return leftText+rightText;
4   }
5 }
```

Methoden und Parameternamen werden per Konvention immer klein geschrieben.

Zugriff auf Felder im Methodenrumpf In einer Methode stehen die Felder der Klasse zur Verfügung[4].

Beispiel 2.1.4 *Wir können mit den bisherigen Mitteln eine kleine Klasse definieren, die es erlaubt, Personen zu repräsentieren, so dass die Objekte dieser Klasse eine Methode haben, um den vollen Namen der Person anzugeben:*

PersonExample1.java

```
1 class PersonExample1 {
2   String vorname;
3   String nachname;
4
5   String getFullName(){
6     return (vorname+" "+nachname);
7   }
8 }
```

Methoden ohne Rückgabewert Es lassen sich auch Methoden schreiben, die keinen eigentlichen Wert berechnen, den sie als Ergebnis zurückgeben. Solche Methoden haben keinen Rückgabetyp. In Java wird dieses gekennzeichnet, indem das Schlüsselwort `void` statt eines Typnamens in der Deklaration steht. Solche Methoden haben keine `return`-Anweisung.

Beispiel 2.1.5 *Folgende kleine Beispielklasse enthält zwei Methoden zum Setzen neuer Werte für ihre Felder:*

[3] Der Ausdruck für *Methoden* kommt speziell aus der objektorientierten Programmierung. In der imperativen Programmierung spricht man von *Prozeduren*, die funktionale Programmierung von *Funktionen*. Weitere Begriffe, die Ähnliches beschreiben, sind *Unterprogramme* und *Subroutinen*.

[4] Das ist wiederum nicht die volle Wahrheit, wie in Kürze zu sehen sein wird.

PersonExample2.java

```
class PersonExample2 {
  String vorname;
  String nachname;

  void setVorname(String newName){
    vorname = newName;
  }
  void setNachname(String newName){
    nachname = newName;
  }
}
```

Obige Methoden weisen konkrete Objekte den Feldern des Objektes zu.

Konstruktoren

Wir haben oben gesehen, wie prinzipiell Objekte einer Klasse mit dem `new`-Konstrukt erzeugt werden. In unserem obigen Beispiel würden wir gerne bei der Erzeugung eines Objektes gleich konkrete Werte für die Felder mit angeben, um direkt eine Person mit konkreten Namen erzeugen zu können. Hierzu können Konstruktoren für eine Klasse definiert werden. Ein Konstruktor kann als eine besondere Art der Methode betrachtet werden, deren Name der Name der Klasse ist und für die kein Rückgabetyp spezifiziert wird. So läßt sich ein Konstruktor spezifizieren, der in unserem Beispiel konkrete Werte für die Felder der Klasse übergeben bekommt:

Person1.java

```
class Person1 {
  String vorname;
  String nachname;

  Person1(String derVorname, String derNachname){
    vorname  = derVorname;
    nachname = derNachname;
  }

  String getFullName(){
    return (vorname+" "+nachname);
  }
}
```

Jetzt lassen sich bei der Erzeugung von Objekten des Typs `Person` konkrete Werte für die Namen übergeben.

Beispiel 2.1.6 *Wir erzeugen ein Personenobjekt mit dem für die entsprechende Klasse geschriebenen Konstruktor:*

TestePerson1.java

```
class TestePerson1 {

  public static void main(String [] _){
    new Person1("Nicolo","Paganini");
  }
}
```

Wie man sieht, machen wir mit diesem Personenobjekt noch nichts. Das Programm hat keine Ausgabe oder Funktion.

lokale Felder

Wir kennen bisher die Felder einer Klasse. Wir können ebenso lokal in einem Methodenrumpf ein Feld deklarieren und benutzen. Solche Felder werden genutzt, um Objekten für einen relativ kurzen Programmabschnitt einen Namen zu geben, mit dem wir das Objekt benennen:

FirstLocalFields.java

```
1  class FirstLocalFields{
2    public static void main(String [] args){
3      String str1 = "hello";
4      String str2 = "world";
5      System.out.println(str1);
6      System.out.println(str2);
7      String str3 = str1+" "+str2;
8      System.out.println(str3);
9    }
10 }
```

Im obigen Programm deklarieren wir drei lokale Felder in einem Methodenrumpf und weisen ihnen jeweils ein Objekt vom Typ `String` zu. Von dem Moment der Zuweisung an steht das Objekt unter dem Namen des Feldes zur Verfügung.

Zugriff auf Eigenschaften eines Objektes

Wir wissen jetzt, wie Klassen definiert und Objekte einer Klasse erzeugt werden. Es fehlt uns schließlich noch, die Eigenschaften eines Objektes anzusprechen. Wenn wir ein Objekt haben, lassen sich die Eigenschaften dieses Objekts über einen Punkt und den Namen der Eigenschaften ansprechen.

Feldzugriffe Wenn wir also ein Objekt in einem Feld `x` abgelegt haben und das Objekt ist vom Typ `Person`, der ein Feld `vorname` hat, so erreichen wir das Objekt, das im Feld `vorname` gespeichert ist, durch den Ausdruck: `x.vorname`.

GetPersonName.java

```
1  class GetPersonName{
2    public static void main (String [] args){
3      Person1 p = new Person1("August","Strindberg");
4
5      String name = p.vorname;
6      System.out.println(name);
7    }
8  }
```

Methodenaufrufe Ebenso können wir auf den Rückgabewert einer Methode zugreifen. Wir nennen dieses dann einen Methodenaufruf. Methodenaufrufe unterscheiden sich syntaktisch darin von Feldzugriffen, dass eine in runden Klammern eingeschlossene Argumentliste folgt:

CallFullNameMethod.java

```
1  class CallFullNameMethod{
2    public static void main (String [] args){
3      Person1 p = new Person1("August","Strindberg");
4
5      String name = p.getFullName();
6      System.out.println(name);
7    }
8  }
```

Aufgabe 8 Schreiben Sie Klassen, die die Objekte des Bibliotheksystems repräsentieren können:

- Personen mit Namen, Vornamen, Straße, Ort und Postleitzahl.
- Bücher mit Titel und Autor.
- Datum mit Tag, Monat und Jahr.
- Buchausleihe mit Ausleiher, Buch und Datum.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben Sie für jede dieser Klassen eine Methode `public String toString()` mit dem Ergebnistyp `String`. Das Ergebnis soll eine gute textuelle Beschreibung des Objektes sein.[5]

c) Schreiben Sie eine Hauptmethode in einer Klasse `Main`, in der Sie Objekte für jede der obigen Klassen erzeugen und die Ergebnisse der `toString`-Methode auf den Bildschirm ausgeben.

Aufgabe 9 Suchen Sie auf Ihrer lokalen Javainstallation oder im Netz auf den Seiten von Sun (`http://www.javasoft.com`) nach der Dokumentation der Standardklassen von Java. Suchen Sie die Dokumentation der Klasse `String`. Testen Sie einige der für die Klasse `String` definierten Methoden.

2.1.3 der this-Bezeichner

Eigenschaften sind an Objekte gebunden. Es gibt in Java eine Möglichkeit, in Methodenrümpfen über das Objekt, für das eine Methode aufgerufen wurde, zu sprechen. Dieses geschieht mit dem Schlüsselwort `this`. *this* ist zu lesen als: dieses Objekt, in dem du dich gerade befindest. Häufig wird der `this`-Bezeichner in Konstruktoren benutzt, um den Namen des Parameters des Konstruktors von einem Feld zu unterscheiden:

UseThis.java

```
1 class UseThis {
2   String aField ;
3   UseThis(String aField){
4     this.aField = aField;
5   }
6 }
```

2.2 statische Eigenschaften

Bisher haben wir Methoden und Felder kennengelernt, die immer nur als Teil eines konkreten Objektes existiert haben. Es muss auch eine Möglichkeit geben, Methoden, die unabhängig von Objekten existieren, zu deklarieren, weil wir ja mit irgendeiner Methoden anfangen müssen, in der erst Objekte erzeugt werden können. Hierzu gibt es statische Methoden. Die Methoden, die immer an ein Objekt gebunden sind, heißen im Gegensatz dazu dynamische Methoden. Statische Methoden brauchen kein Objekt, um aufgerufen zu werden. Sie werden exakt so deklariert wie dynamische Methoden, mit dem einzigen Unterschied, dass ihnen das Schlüsselwort `static` vorangestellt wird. Statische Methoden werden auch in einer Klasse definiert und gehören zu einer Klasse. Da statische Methoden nicht zu den einzelnen Objekten gehören, können sie nicht auf dynamische Felder und Methoden der Objekte zugreifen.

[5] Sie brauchen noch nicht zu verstehen, warum vor dem Rückgabetyp noch ein Attribut `public` steht.

StaticTest.java
```
class StaticTest {
  static void printThisText(String text){
    System.out.println(text);
  }
}
```

Statische Methoden werden direkt auf der Klasse, nicht auf einem Objekt der Klasse aufgerufen.

Auf statische Eigenschaften wird zugegriffen, indem vom Klassennamen per Punkt getrennt die Eigenschaft aufgerufen wird:

CallStaticTest.java
```
class CallStaticTest {
  public static void main(String [] args){
    StaticTest.printThisText("hello");
  }
}
```

Im nächsten Kapitel werden wir statische Methoden als mathematische Funktionen im Detail betrachten.

Ebenso wie statische Methoden gibt es auch statische Felder. Im Unterschied zu dynamischen Feldern existieren statische Felder genau einmal, nämlich in der Klasse. Dynamische Felder existieren für jedes Objekt der Klasse.

Statische Eigenschaften nennt man auch Klassenfelder bzw. Klassenmethoden.

Mit statischen Eigenschaften verläßt man quasi die objektorientierte Programmierung, denn diese Eigenschaften sind nicht mehr an Objekte gebunden. Man könnte mit statischen Eigenschaften Programme schreiben, die niemals ein Objekt erzeugen.

Aufgabe 10 Ergänzen Sie jetzt die Klasse `Person` aus der letzten Aufgabe um ein statisches Feld `letzerVorname` mit einer Zeichenkette, die angeben soll, welchen Vornamen das zuletzt erzeugte Objekt vom Typ `Person` hatte. Hierzu müssen Sie im Konstruktor der Klasse Person dafür sorgen, dass nach der Zuweisung der Objektfelder auch noch das Feld `letzerVorname` verändert wird. Testen Sie in einer Testklasse, dass sich tatsächlich nach jeder Erzeugung einer neuen Person dieses Feld verändert hat.

2.3 Vererben und Erben

Eines der grundlegendsten Ziele der objektorientierten Programmierung ist die Möglichkeit, bestehende Programme um neue Funktionalität erweitern zu können. Hierzu bedient man sich der Vererbung. Bei der Definition einer neuen Klassen hat man die Möglichkeit, anzugeben, dass diese Klasse alle Eigenschaften von einer bestehenden Klasse erbt.

Wir haben in einer früheren Übungsaufgabe die Klasse `Person` geschrieben:

Person2.java
```
class Person2 {

  String name ;
  String address;

  Person2(String name, String address){
    this.name = name;
    this.address = address;
  }
```

```
10
11   public String toString(){
12     return name+", "+address;
13   }
14 }
```

Wenn wir zusätzlich eine Klasse schreiben wollen, die nicht beliebige Personen speichern kann, sondern Studenten, die als zusätzliche Information noch eine Matrikelnummer haben, so stellen wir fest, dass wir wieder Felder für den Namen und die Adresse anlegen müssen; d.h. wir müssen die bereits in der Klasse `Person` zur Verfügung gestellte Funktionalität ein weiteres Mal schreiben:

StudentOhneVererbung.java

```
1  class StudentOhneVererbung {
2
3    String name ;
4    String address;
5    int matrikelNummer;
6
7    StudentOhneVererbung(String name, String address,int nr){
8      this.name = name;
9      this.address = address;
10     matrikelNummer = nr;
11   }
12
13   public String toString(){
14     return    name + ", " + address
15             + " Matrikel-Nr.: " + matrikelNummer;
16   }
17 }
```

Mit dem Prinzip der Vererbung wird es ermöglicht, diese Verdoppelung des Codes, der bereits für die Klasse `Person` geschrieben wurde, zu umgehen.

Wir werden in diesem Kapitel schrittweise eine Klasse `Student` entwickeln, die die Eigenschaften erbt, die wir in der Klasse `Person` bereits definiert haben.

Zunächst schreibt man in der Klassendeklaration der Klasse `Student`, dass deren Objekte alle Eigenschaften der Klasse `Person` erben. Hierzu wird das Schlüsselwort `extends` verwendet:

Student.java

```
1  class Student extends Person2 {
```

Mit dieser `extends`-Klausel wird angegeben, dass die Klasse von einer anderen Klasse abgeleitet wird und damit deren Eigenschaften erbt. Jetzt brauchen die Eigenschaften, die schon in der Klasse `Person` definiert wurden, nicht mehr neu definiert zu werden.

Mit der Vererbung steht ein Mechanismus zur Verfügung, der zwei primäre Anwendungen hat:

- **Erweitern**: zu den Eigenschaften der Oberklasse werden weitere Eigenschaften hinzugefügt. Im Beispiel der Studentenklasse soll das Feld `matrikelNummer` hinzugefügt werden.
- **Verändern**: eine Eigenschaft der Oberklasse wird umdefiniert. Im Beispiel der Studentenklasse soll die Methode `toString` der Oberklasse in ihrer Funktionalität verändert werden.

Es gibt in Java für eine Klasse immer nur genau eine direkte Oberklasse. Eine sogenannte multiple Erbung ist in Java nicht möglich.[6] Es gibt immer maximal eine `extends`-Klausel in einer Klassendefinition.

[6] Dieses ist z.B. in C++ möglich.

2.3.1 Hinzufügen neuer Eigenschaften

Unser erstes Ziel der Vererbung war, eine bestehende Klasse um neue Eigenschaften zu erweitern. Hierzu können wir jetzt einfach mit der `extends`-Klausel angeben, dass wir die Eigenschaften einer Klasse erben. Die Eigenschaften, die wir zusätzlich haben wollen, lassen sich schließlich wie gewohnt deklarieren:

Student.java
```
2   int matrikelNummer;
```

Hiermit haben wir eine Klasse geschrieben, die drei Felder hat: `name` und `adresse`, die von der Klasse `Person` geerbt werden und zusätzlich das Feld `matrikelNummer`. Diese drei Felder können für Objekte der Klasse `Student` in gleicher Weise benutzt werden:

Student.java
```
3 String writeAllFields(Student s){
4    return s.name+" "+s.address+" "+s.matrikelNummer;
5 }
```

Ebenso so wie Felder lassen sich Methoden hinzufügen. Z.B. eine Methode, die die Matrikelnummer als Rückgabewert hat:

Student.java
```
6 int getMatrikelNummer(){
7   return matrikelNummer;
8 }
```

2.3.2 Überschreiben bestehender Eigenschaften

Unser zweites Ziel ist, durch Vererbung eine Methode in ihrem Verhalten zu verändern. In unserem Beispiel soll die Methode `toString` der Klasse `Person` für Studentenobjekte so geändert werden, dass das Ergebnis auch die Matrikelnummer enthält. Hierzu können wir die entsprechende Methode in der Klasse `Student` einfach neu schreiben:

Student.java
```
9   public String toString(){
10    return    name + ", " + address
11            + " Matrikel-Nr.: " + matrikelNummer;
12  }
```

Obwohl Objekte der Klasse `Student` auch Objekte der Klasse `Person` sind, benutzen sie nicht die Methode `toString` der Klasse `Person`, sondern die neu definierte Version aus der Klasse `Student`.

Um eine Methode zu überschreiben, muss sie dieselbe Signatur bekommen, die sie in der Oberklasse hat.

2.3.3 Konstruktion

Um für eine Klasse konkrete Objekte zu konstruieren, braucht die Klasse entsprechende Konstruktoren. In unserem Beispiel soll jedes Objekt der Klasse `Student` auch ein Objekt der Klasse `Person` sein. Daraus folgt, dass, um ein Objekt der Klasse `Student` zu erzeugen, es auch notwendig ist, ein Objekt der Klasse `Person` zu erzeugen. Wenn wir also einen Konstruktor für `Student` schreiben, sollten wir sicherstellen, dass mit diesem auch ein gültiges Objekt der Klasse `Person` erzeugt wird. Hierzu kann man den Konstruktor der Oberklasse aufrufen. Dieses geschieht mit dem Schlüsselwort `super`. `super` ruft den Konstruktor der Oberklasse auf:

Student.java
```
13    Student(String name,String adresse,int nr){
14      super(name,adresse);
15      matrikelNummer = nr;
16    }
17  }
```

In unserem Beispiel bekommt der Konstruktor der Klasse `Student` alle Daten, die benötigt werden, um ein Personenobjekt und ein Studentenobjekt zu erzeugen. Als erstes wird im Rumpf des Studentenkonstruktors der Konstruktor der Klasse `Person` aufgerufen. Anschließend wird das zusätzliche Feld der Klasse `Student` mit entsprechenden Daten initialisiert.

Ein Objekt der Klasse `Student` kann wie gewohnt konstruiert werden:

TestStudent.java
```
1  class TestStudent {
2    public static void main(String [] _){
3      Student s
4       = new Student("Martin Müller","Hauptstraße 2",755423);
5      System.out.println(s);
6    }
7  }
```

2.3.4 Zuweisungskompatibilität

Objekte einer Klasse sind auch ebenso Objekte ihrer Oberklasse. Daher können sie benutzt werden wie die Objekte ihrer Oberklasse, insbesondere bei einer Zuweisung. Da in unserem Beispiel die Objekte der Klasse `Student` auch Objekte der Klasse `Person` sind, dürfen diese auch Feldern des Typs `Person` zugewiesen werden:

TestStudent1.java
```
1  class TestStudent1{
2    public static void main(String [] args){
3      Person2 p
4       = new Student("Martin Müller","Hauptstraße",7463456);
5    }
6  }
```

Alle Studenten sind auch Personen.

Hingegen die andere Richtung ist nicht möglich: nicht alle Personen sind Studenten. Folgendes Programm wird von Java mit einem Fehler zurückgewiesen:

```
1  class StudentError1{
2    public static void main(String [] args){
3      Student s
4        = new Person2("Martin Müller","Hauptstraße");
5    }
6  }
```

Die Kompilierung dieser Klasse führt zu folgender Fehlermeldung:

```
StudentError1.java:3: incompatible types
found   : Person
required: Student
    Student s = new Person2("Martin Müller","Hauptstraße");
                ^
1 error
```

Java weist diese Klasse zurück, weil eine Person nicht ein Student ist.

Gleiches gilt für den Typ von Methodenparametern. Wenn die Methode einen Parameter vom Typ `Person` verlangt, so kann man ihm auch Objekte eines spezielleren Typs geben, in unserem Fall der Klasse `Student`.

TestStudent2.java

```
1  class TestStudent2 {
2
3    static void printPerson(Person2 p){
4      System.out.println(p.toString());
5    }
6
7    public static void main(String [] args){
8       Student s
9         = new Student("Martin Müller","Hauptstraße",754545);
10      printPerson(s);
11   }
12 }
```

Der umgekehrte Fall ist wiederum nicht möglich. Methoden, die als Parameter Objekte der Klasse `Student` verlangen, dürfen nicht mit Objekten einer allgemeineren Klasse aufgerufen werden:

```
1  class StudentError2{
2
3    static void printStudent(Student s){
4      System.out.println(s.toString());
5    }
6
7    public static void main(String [] args){
8       Person2 p = new Person2("Martin Müller","Hauptstraße");
9       printStudent(p);
10   }
11 }
```

Auch hier führt die Kompilierung zu einer entsprechenden Fehlermeldung:

```
StudentError2.java:9: printStudent(Student) in StudentError2
                      cannot be applied to (Person2)
     printStudent(p);
     ^
1 error
```

2.3.5 Späte Bindung (late binding)

Wir haben gesehen, dass wir Methoden überschreiben können. Interessant ist, wann welche Methode ausgeführt wird. In unserem Beispiel gibt es je eine Methode `toString` in der Oberklasse `Person` als auch in der Unterklasse `Student`.

Welche dieser zwei Methoden wird wann ausgeführt? Wir können dieser Frage experimentell nachgehen:

TestLateBinding.java

```
1  class TestLateBinding {
2
3    public static void main(String [] args){
4      Student s = new Student("Martin Müller","Hauptstraße",756456);
5      Person2 p1 = new Person2("Harald Schmidt","Marktplatz");
```

```
6
7       System.out.println(s.toString());
8       System.out.println(p1.toString());
9
10      Person2 p2 = new Student("Martin Müller","Hauptstraße",756456);
11      System.out.println(p2.toString());
12    }
13  }
```

Dieses Programm erzeugt folgende Ausgabe:

```
sep@swe10:~/fh/> java TestLateBinding
Martin Müller, Hauptstraße Matrikel-Nr.: 756456
Harald Schmidt, Marktplatz
Martin Müller, Hauptstraße Matrikel-Nr.: 756456
```

Die ersten beiden Ausgaben entsprechen sicherlich den Erwartungen: es wird eine Student und anschließend eine Person ausgegeben. Die dritte Ausgabe ist interessant. Obwohl der Befehl:

```
1 System.out.println(p2.toString());
```

die Methode `toString` auf einem Feld vom Typ `Person` ausführt, wird die Methode `toString` aus der Klasse `Student` ausgeführt. Dieser Effekt entsteht, weil das Objekt, das im Feld `p2` gespeichert wurde, als Student und nicht als Person erzeugt wurde. Die Idee der Objektorientierung ist, dass die Objekte die Methoden in sich enthalten. In unserem Fall enthält das Objekt im Feld `p2` seine eigene `toString`-Methode. Diese wird ausgeführt. Der Ausdruck `p2.toString()` ist also zu lesen als:

> Objekt, das in Feld `p2` gespeichert ist, führe bitte deine Methode `toString` aus.

Da dieses Objekt, auch wenn wir es dem Feld nicht ansehen, ein Objekt der Klasse `Student` ist, führt es die entsprechende Methode der Klasse `Student` und nicht der Klasse `Person` aus.

Dieses in Java realisierte Prinzip wird als *late binding* bezeichnet.[7]

Aufgabe 11 In dieser Aufgabe sollen Sie eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben.

Gegeben seien die folgenden Javaklassen, wobei Sie die Klasse `Dialogue` nicht zu analysieren oder zu verstehen brauchen:

- ButtonLogic.java

```
1 class ButtonLogic {
2   String getDescription(){
3     return "in Großbuchstaben umwandeln";
4   }
5   String eval(String x){return x.toUpperCase();}
6 }
```

- Dialogue.java

```
1 import javax.swing.*;
2 import java.awt.event.*;
3 import java.awt.*;
4 class Dialogue extends JFrame{
5
6   final ButtonLogic logic;
7
8   final JButton button;
```

[7] Achtung: *late binding* funktioniert in Java nur bei Methoden, nicht bei Feldern.

```
9   final JTextField inputField  = new JTextField(20) ;
10  final JTextField outputField = new JTextField(20) ;
11  final JPanel p = new JPanel();
12
13  Dialogue(ButtonLogic l){
14    logic = l;
15    button=new JButton(logic.getDescription());
16    button.addActionListener
17     (new ActionListener(){
18        public void actionPerformed(ActionEvent _){
19          outputField.setText
20              (logic.eval(inputField.getText().trim()));
21        }
22      });
23    p.setLayout(new BorderLayout());
24    p.add(inputField,BorderLayout.NORTH);
25    p.add(button,BorderLayout.CENTER);
26    p.add(outputField,BorderLayout.SOUTH);
27    getContentPane().add(p);
28    pack();
29    setVisible(true);
30  }
31 }
```

TestDialogue.java

```
1  class TestDialogue {
2    public static void main(String [] _){
3      new Dialogue(new ButtonLogic());
4    }
5  }
```

a) Übersetzen Sie die drei Klassen und starten Sie das Programm.

b) Schreiben Sie eine Unterklasse der Klasse `ButtonLogic`. Sie sollen dabei die Methoden `getDescription` und `eval` so überschreiben, dass der Eingabestring in Kleinbuchstaben umgewandelt wird. Schreiben Sie eine Hauptmethode, in der Sie ein Objekt der Klasse `Dialogue` mit einem Objekt Ihrer Unterklasse von `ButtonLogic` erzeugen.

c) Schreiben Sie jetzt Applikationen, die die Umwandlung von Groß- in Kleinbzw. Klein- in Großbuchstaben entsprechend einer Türkischen Lokalisierung durchführen.

d) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so dass Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, das für den im Eingabefeld eingegebenen String die Länge im Ausgabefeld anzeigt.

e) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so dass Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, das für den im Eingabefeld eingegebenen String anzeigt, ob dieser einen Teilstring `"depp"` enthält.

Methoden als Daten

Mit dem Prinzip der späten Methodenbindung können wir unsere ursprüngliche Arbeitshypothese, dass Daten und Programme zwei unterschiedliche Konzepte sind, etwas aufweichen. Objekte enthalten in ihren Feldern die Daten und mit ihren Methoden Unterprogramme. Wenn in einem Methodenaufruf ein Objekt übergeben wird, übergeben wir somit in dieser Methode nicht nur spezifische Daten, sondern auch spezifische Unterprogramme. Dieses haben wir uns in

der letzten Aufgabe zunutze gemacht, um Funktionalität an eine graphische Benutzeroberfläche zu übergeben. Hierzu betrachten wir den Konstruktor und die Benutzung der Klasse `Dialogue`, die in einer der letzten Aufgaben vorgegeben war:

```
1  Dialogue(ButtonLogic l){
2    logic = l;
3    button=new JButton(logic.getDescription());
```

Diese Klasse hat einen Konstruktor, der als Argument ein Objekt des Typs `ButtonLogic` erwartet. In dieser Klasse gibt es eine Methode `String eval(String x)`, die offensichtlich benutzt wird, um die Funktionalität der graphischen Benutzerschnittstelle (GUI) zu bestimmen. Ebenso enthält sie eine Methode `String getDescription()`, die festlegt, was für eine Beschriftung für den Knopf benutzt werden soll. Wir übergeben also Funktionalität in Form von Methoden an die Klasse `Dialogue`. Je nachdem, wie in unserer konkreten Unterklasse von `ButtonLogic` diese beiden Methoden überschrieben sind, verhält sich das Guiprogramm.

Die in diesem Abschnitt gezeigte Technik ist eine typische Javatechnik. Bestehende Programme können erweitert und mit eigener Funktionalität benutzt werden, ohne dass man die bestehenden Klassen zu ändern braucht. Wir haben neue Guiprogramme schreiben können, ohne an der Klasse `Dialogue` etwas ändern zu müssen, ohne sogar irgendetwas über Guiprogrammierung zu wissen. Durch die Objektorientierung lassen sich in Java hervorragend verschiedene Aufgaben trennen und im Team bearbeiten sowie Softwarekomponenten wiederverwenden.

Aufgabe 12 In dieser Aufgabe sollen Sie Klassen zur Beschreibung geometrischer Objekte im 2-dimensionalen Raum entwickeln. Wir beginnen mit der Klasse für Punkte im zweidimensionalen Raum. Schreiben Sie für die Klasse geeignete Konstruktoren und eine Methode `toString`.

a) Schreiben Sie eine Klasse `Vertex`, die Punkte im zweidimensionalen Raum darstellt. Benutzen Sie für die Koordinaten dabei den primitiven Typ `double`.

b) Schreiben Sie einen geeigneten Konstruktor für die Klasse `Vertex`.

c) Schreiben Sie für die Klasse `Vertex` eine Methode `toString`.

d) Überschreiben Sie für die Klasse `Vertex` Methode `equals`.

e) Definieren Sie Methoden zur Addition und Subtraktion von Punkten, sowie zur Multiplikation mit einem Skalar mit folgenden Signaturen.

```
1 Vertex add(Vertex that);
2 Vertex sub(Vertex that);
3 Vertex mult(double skalar);
```

Diese Methoden sollen das Objekt unverändert lassen und ein neues Vertex-Objekt als Ergebnis zurückliefern.

f) Testen Sie die Klasse `Vertex` in einer `main`-Methode.

Aufgabe 13 Sie sollen auf diesem Übungsblatt die Aufgaben der vorherigen Aufgabe weiterführen und die dort programmierte Klassen benutze und gegenfalls verändern.

a) Schreiben Sie eine Klasse `GeometricObject`. Eine geometrische Figur soll als Eigenschaften die Weite und Höhe eines gedachten die Figur umschließenden Rechtecks *(bounding box)* haben, sowie die Position der linken oberen Ecke dieses Rechtecks im 2-dimensionalen Raum. (Wobei das Koordinatensystem von oben nach unten und links nach rechts aufsteigend verläuft.)

b) Schreiben Sie geeignete Konstruktoren für die Klasse `GeometricObject` und eine aussagekräftige Methode `toString`.

c) Überschreiben Sie für die Klasse `GeometricObject` Methode `equals`.

d) Implementieren Sie die folgende Methoden für Ihre Klasse `GeometricObject`:

- `void moveTo(Vertex p)`, die den Eckpunkt der Figur auf einen neuen Wert setzt.
- `void move(Vertex v)`, die den Eckpunkt um die Kordinatenwerte des Arguments verschiebt.
- Eine Methode `area`, die den Flächeninhalt der *bounding box* berechnet.
- `boolean hasWithin(Vertex p)`, die wahr ist, wenn der Punkt `p` innerhalb der *bounding box* der geometrischen Figur liegt.

Testen Sie diese Methoden in einer Methode `main`.

Aufgabe 14 Sie sollen auf diesem Übungsblatt die Aufgaben der vorherigen Aufgabe weiterführen und die dort programmierte Klassen benutzen und gegenfalls verändern.

a) Implementieren Sie eine Unterklasse `Ellipse` der Klasse `GeometricObject`, die Ellipsen darstellt. Schreiben Sie geeignete Konstruktoren und überschreiben Sie die Methode `toString` und die Methode `area`, so dass jetzt der Flächeninhalt des Kreises und nicht der bounding box berechnet wird.

2.3.6 Zugriff auf Methoden der Oberklasse

Vergleichen wir die Methoden `toString` der Klassen `Person` und `Student`, so sehen wir, dass in der Klasse `Student` Code der Oberklasse verdoppelt wurde:

```
1  public String toString(){
2    return    name + ", " + address
3            + " Matrikel-Nr.: " + matrikelNummer;
4  }
```

Der Ausdruck

```
1 name + ", " + address
```

wiederholt die Berechnung der `toString`-Methode aus der Klasse `Person`. Es wäre schön, wenn an dieser Stelle die entsprechende Methode aus der Oberklasse benutzt werden könnte. Auch dieses ist in Java möglich. Ähnlich, wie der Konstruktor der Oberklasse explizit aufgerufen werden kann, können auch Methoden der Oberklasse explizit aufgerufen werden. Auch in diesem Fall ist das Schlüsselwort `super` zu benutzen, allerdings nicht in der Weise, als sei `super` eine Methode, sondern als sei es ein Feld, das ein Objekt enthält, also ohne Argumentklammern. Dieses Feld erlaubt es, direkt auf die Eigenschaften der Oberklasse zuzugreifen. Somit läßt sich die `toString`-Methode der Klasse `Student` auch wie folgt schreiben:

```
1  public String toString(){
2    return      //call toString of super class
3                super.toString()
4                //add the Matrikelnummer
5            + " Matrikel-Nr.: " + matrikelNummer;
6  }
```

2.3.7 Die Klasse Object

Eine berechtigte Frage ist, welche Klasse die Oberklasse für eine Klasse ist, wenn es keine `extends`-Klausel gibt. Bisher haben wir nie eine entsprechende Oberklasse angegeben.

Java hat in diesem Fall eine Standardklasse: `Object`. Wenn nicht explizit eine Oberklasse angegeben wird, so ist die Klasse `Object` die direkte Oberklasse. Weil die `extends`-Relation transitiv ist, ist schließlich jede Klasse eine Unterklasse der Klasse `Object`. Insgesamt bilden alle Klassen, die in Java existieren, eine Baumstruktur, deren Wurzel die Klasse `Object` ist.

Es bewahrheitet sich die Vermutung über objektorientierte Programmierung, dass alles als Objekt betrachtet wird.[8] Es folgt insbesondere, dass jedes Objekt die Eigenschaften hat, die in der Klasse `Object` definiert wurden. Ein Blick in die Java API Documentation zeigt, dass zu diesen Eigenschaften auch die Methode `toString` gehört, wie wir sie bereits einige mal geschrieben haben. Jetzt erkennen wir, dass wir diese Methode dann überschrieben haben. Auch wenn wir für eine selbstgeschriebene Klasse die Methode `toString` nicht definiert haben, existiert eine solche Methode. Allerdings ist deren Verhalten selten ein für unsere Zwecke geeignetes.

Die Eigenschaften, die alle Objekte haben, weil sie in der Klasse `Object` definiert sind, sind äußerst allgemein. Sobald wir von einem Object nur noch wissen, dass es vom Typ `Object` ist, können wir kaum noch spezifische Dinge mit ihm anfangen.

Die Methode equals

Eine weitere Methode, die in der Klasse `Object` definiert ist, ist die Methode `equals`. Sie hat folgende Signatur:

```
1 public boolean equals(Object other)
```

Wenn man diese Methode überschreibt, so kann definiert werden, wann zwei Objekte einer Klasse als gleich angesehen werden sollen. Für Personen würden wir gerne definieren, dass zwei Objekte dieser Klasse gleich sind, wenn sie ein und denselben Namen und ein und dieselbe Adresse haben. Mit unseren derzeitigen Mitteln läßt sich dieses leider nicht ausdrücken. Wir würden gerne die `equals`-Methode wie folgt überschreiben:

```
1 public boolean equals(Object other){
2   return        this.name.equals(other.name)
3               && this.adresse.equals(other.adresse);
4 }
```

Dieses ist aber nicht möglich, weil für das Objekt `other`, von dem wir nur wissen, dass es vom Typ `Object` ist, keine Felder `name` und `adresse` existieren.

Um dieses Problem zu umgehen, sind Konstrukte notwendig, die von allgemeineren Typen wieder zu spezielleren Typen führen. Ein solches Konstrukt lernen wir in den folgenden Abschnitten kennen.

2.3.8 Klassentest

Wie wir oben gesehen haben, können wir zu wenige Informationen über den Typen eines Objektes haben. Objekte wissen aber selbst, von welcher Klasse sie einmal erzeugt wurden. Java stellt einen binären Operator zur Verfügung, der erlaubt, abzufragen, ob ein Objekt zu einer Klasse

[8] Java kennt acht eingebaute primitive Typen für Zahlen, Wahrheitswerte und Buchstaben. Diese sind zwar keine Objekte, werden notfalls von Java aber in entsprechende Objektklassen automatisch konvertiert.

gehört. Dieser Operator heißt `instanceof`. Er hat links ein Objekt und rechts einen Klassennamen. Das Ergebnis ist ein bool'scher Wert, der genau dann wahr ist, wenn das Objekt eine Instanz der Klasse ist.

InstanceOfTest.java

```
1  class InstanceOfTest {
2    public static void main(String [] str){
3       Person2 p1 = new Person2("Strindberg","Skandinavien");
4       Person2 p2 = new Student("Ibsen","Skandinavien",789565);
5       if (p1 instanceof Student)
6         System.out.println("p1 ist ein  Student.");
7       if (p2 instanceof Student)
8         System.out.println("p2 ist einStudent.");
9       if (p1 instanceof Person2)
10        System.out.println("p1 ist eine Person.");
11      if (p2 instanceof Person2)
12        System.out.println("p2 ist eine Person.");
13   }
14 }
```

An der Ausgabe dieses Programms kann man erkennen, dass ein `instanceof`-Ausdruck wahr wird, wenn das Objekt ein Objekt der Klasse oder aber einer Unterklasse der Klasse des zweiten Operanden ist.

```
sep@swe10:~/fh> java InstanceOfTest
p2 ist einStudent.
p1 ist eine Person.
p2 ist eine Person.
```

2.3.9 Typzusicherung (Cast)

Im letzten Abschnitt haben wir eine Möglichkeit kennengelernt, zu fragen, ob ein Objekt zu einer bestimmten Klasse gehört. Um ein Objekt dann auch wieder so benutzen zu können, dass es zu dieser Klasse gehört, müssen wir diesem Objekt diesen Typ erst wieder zusichern. Im obigen Beispiel haben wir zwar erfragen können, dass das in Feld `p2` gespeicherte Objekt nicht nur eine Person, sondern ein Student ist; trotzdem können wir noch nicht `p2` nach seiner Matrikelnummer fragen. Hierzu müssen wir erst zusichern, dass das Objekt den Typ `Student` hat.

Eine Typzusicherung in Java wird gemacht, indem dem entsprechenden Objekt in Klammer der Typ vorangestellt wird, den wir ihm zusichern wollen:

CastTest.java

```
1  class CastTest {
2    public static void main(String [] str){
3      Person2 p = new Student("Ibsen","Skandinavien",789565);
4
5      if (p instanceof Student){
6        Student s = (Student)p;
7        System.out.println(s.matrikelNummer);
8      }
9    }
10 }
```

Die Zeile `s = (Student)p;` sichert erst dem Objekt im Feld `p` zu, dass es ein Objekt des Typs `Student` ist, so dass es dann als Student benutzt werden kann. Wir haben den Weg zurück vom Allgemeinen ins Spezifischere gefunden. Allerdings ist dieser Weg gefährlich. Eine Typzusicherung kann fehlschlagen:

```
1 class CastError {
2   public static void main(String [] str){
3     Person2 p = new Person2("Strindberg","Skandinavien");
4     Student s = (Student)p;
5     System.out.println(s.matrikelNr);
6   }
7 }
```

Dieses Programm macht eine Typzusicherung des Typs `Student` auf ein Objekt, das nicht von diesem Typ ist. Es kommt in diesem Fall zu einen Laufzeitfehler:

```
sep@swe10:~/fh> java CastError
Exception in thread "main" java.lang.ClassCastException: Person2
        at CastError.main(CastError.java:4)
```

Die Fehlermeldung sagt, dass wir in Zeile 4 des Programms eine Typzusicherung auf ein Objekt des Typs `Person` vornehmen, die fehlschlägt. Will man solche Laufzeitfehler verhindern, so ist man auf der sicheren Seite, wenn eine Typzusicherung nur dann gemacht wird, nachdem man sich mit einem `instanceof`-Ausdruck davon überzeugt hat, dass das Objekt wirklich von dem Typ ist, den man ihm zusichern will.

Mit den jetzt vorgestellten Konstrukten können wir eine Lösung der Methode `equals` für die Klasse `Person` mit der erwarteten Funktionalität schreiben:

```
1 public boolean equals(Object other){
2   boolean erg = false;
3   if (other instanceof Person2){
4      Person2 p = (Person2) other;
5      erg = this.name.equals(p.name)
6              && this.adresse.equals(p.adresse);
7   }
8   return erg;
9 }
```

Nur, wenn das zu vergleichende Objekt auch vom Typ `Person` ist und den gleichen Namen und die gleiche Adresse hat, dann sind zwei Personen gleich.

Kapitel 3

Funktionale und imperative Konzepte

Achtung: Dieses Kapitel ist noch nicht fertig.

Im letzten Abschnitt wurde ein erster Einstieg in die objektorientierte Programmierung gegeben. Wie zu sehen war, ermöglicht die objektorientierte Programmierung, das zu lösende Problem in logische Untereinheiten zu unterteilen, die direkt mit den Teilen der zu modellierenden Problemwelt korrespondieren.

Die Methodenrümpfe, die die eigentlichen Befehle enthalten, in denen etwas berechnet werden soll, waren bisher recht kurz. In diesem Kapitel werden wir Konstrukte kennenlernen, die es ermöglichen, in den Methodenrümpfen komplexe Berechnungen vorzunehmen. Die in diesem Abschnitt vorgestellten Konstrukte sind herkömmliche Konstrukte der imperativen Programmierung und in ähnlicher Weise auch in Programmiersprachen wie C zu finden.[1]

3.1 Primitive Typen, die klassenlose Gesellschaft

Bisher haben wir noch überhaupt keine Berechnungen im klassischen Sinne als das Rechnen mit Zahlen kennengelernt. Java stellt Typen zur Repräsentation von Zahlen zur Verfügung. Leider sind diese Typen keine Klassen; d.h. insbesondere, dass auf diesen Typen keine Felder und Methoden existieren, auf die mit einem Punkt zugegriffen werden kann.

Die im Folgenden vorgestellten Typen nennt man primitive Typen. Sie sind fest von Java vorgegeben. Im Gegensatz zu Klassen, die der Programmierer selbst definieren kann, können keine neuen primitiven Typen definiert werden. Um primitive Typnamen von Klassennamen leicht textuell unterscheiden zu können, sind sie in Kleinschreibung definiert worden.

Ansonsten werden primitive Typen genauso behandelt wie Klassen. Felder können primitive Typen als Typ haben und ebenso können Parametertypen und Rückgabetypen von Methoden primitive Typen sein.

Um Daten der primitiven Typen aufschreiben zu können, gibt es jeweils Literale für die Werte dieser Typen.

3.1.1 Zahlenmengen in der Mathematik

In der Mathematik sind wir gewohnt, mit verschiedenen Mengen von Zahlen zu arbeiten:

- **natürliche Zahlen** $\mathbb{N}$: Eine induktiv definierbare Menge mit einer kleinsten Zahl, so dass es für jede Zahl eine eindeutige Nachfolgerzahl gibt.

[1] Gerade in diesem Bereich wollten die Entwickler von Java einen leichten Umstieg von der C-Programmierung nach Java ermöglichen. Leider hat Java in dieser Hinsicht auch ein C-Erbe und ist nicht in allen Punkte so sauber entworfen, wie es ohne diese Designvorgabe wäre.

- **ganze Zahlen** $\mathbb{Z}$: Die natürlichen Zahlen erweitert um die mit einem negativen Vorzeichen behafteten Zahlen, die sich ergeben, wenn man eine größere Zahl von einer natürlichen Zahl abzieht.
- **rationale Zahlen** $\mathbb{Q}$: Die ganzen Zahlen erweitert um Brüche, die sich ergeben, wenn man eine Zahl durch eine Zahl teilt, von der sie kein Vielfaches ist.
- **reelle Zahlen** $\mathbb{R}$: Die ganzen Zahlen erweitert um irrationale Zahlen, die sich z.B. aus der Quadratwurzel von Zahlen ergeben, die nicht das Quadrat einer rationalen Zahl sind.
- **komplexe Zahlen** $\mathbb{C}$: Die reellen Zahlen erweitert um imaginäre Zahlen, wie sie benötigt werden, um einen Wurzelwert für negative Zahlen darzustellen.

Es gilt folgende Mengeninklusion zwischen diesen Mengen:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

Da bereits $\mathbb{N}$ nicht endlich ist, ist keine dieser Mengen endlich.

3.1.2 Zahlenmengen im Rechner

Da wir nur von einer endlich großen Speicherkapazität ausgehen können, lassen sich für keine der aus der Mathematik bekannten Zahlenmengen alle Werte in einem Rechner darstellen. Wir können also schon einmal nur Teilmengen der Zahlenmengen darstellen.

Von der Hardwareseite stellt sich heute zumeist die folgende Situation dar: Der Computer hat einen linearen Speicher, der in Speicheradressen unterteilt ist. Eine Speicheradresse bezeichnet einen Bereich von 32 Bit. Wir bezeichnen diese als ein Wort. Die Einheit von 8 Bit wird als Byte bezeichnet [2]. Heutige Rechner verwalten also in der Regel Dateneinheiten von 32 Bit. Hieraus ergibt sich die Kardinalität der Zahlenmengen, mit denen ein Rechner als primitive Typen rechnen kann. Soll mit größeren Zahlenmengen gerechnet werden, so muss hierzu eine Softwarelösung genutzt werden.

3.1.3 natürliche Zahlen

Natürliche Zahlen werden in der Regel durch Zeichenketten von Symbolen, den Ziffern, eines endlichen Alphabets dargestellt. Die Größe dieser Symbolmenge wird als die Basis b der Zahlendarstellung bezeichnet. Für die Basis b gilt: $b > 1$. Die Ziffern bezeichnen die natürlichen Zahlen von 0 bis $b - 1$.

Der Wert der Zahl einer Zeichenkette $a_{n-1} \dots a_0$ berechnet sich für die Basis b nach folgender Formel:

$$\sum_{i=0}^{n-1} a_i * b^i = a_0 * b^0 + \cdots + a_{n-1} * b^{n-1}$$

Gebräuchliche Basen sind:

- 2: Dualsystem
- 8: Oktalsystem
- 10: Dezimalsystem

[2] ein anderes selten gebrauchtes Wort aus dem Französischen ist: Oktett

- 16: Hexadezimalsystem[3]

Zur Unterscheidung wird im Zweifelsfalle die Basis einer Zahlendarstellung als Index mit angegeben.

Beispiel 3.1.1 *Die Zahl 10 in Bezug auf unterschiedliche Basen dargestellt:*

$$(10)_{10} = (A)_{16} = (12)_8 = (1010)_2$$

Darüberhinaus gibt es auch Zahlendarstellungen, die nicht in Bezug auf eine Basis definiert sind, wie z.B. die römischen Zahlen.

Betrachten wir wieder unsere Hardware, die in Einheiten von 32 Bit agiert, so lassen sich in einer Speicheradresse durch direkte Anwendung der Darstellung im Dualsystem die natürlichen Zahlen von 0 bis $2^{32} - 1$ darstellen.

3.1.4 ganze Zahlen

In vielen Programmiersprachen wie z.B. Java gibt es keinen primitiven Typen, der eine Teilmenge der natürlichen Zahlen darstellt.[4] Zumeist wird der Typ `int` zur Darstellung ganzer Zahlen benutzt. Hierzu bedarf es einer Darstellung vorzeichenbehafteter Zahlen in den Speicherzellen des Rechners. Es gibt mehrere Verfahren, wie in einem dualen System vorzeichenbehaftete Zahlen dargestellt werden können.

Vorzeichen und Betrag

Die einfachste Methode ist, von den n für die Zahlendarstellung zur Verfügung stehenden Bit eines zur Darstellung des Vorzeichens und die übrigen $n - 1$ Bit für eine Darstellung im Dualsystem zu nutzen.

Beispiel 3.1.2 *In der Darstellung durch Vorzeichen und Betrag werden bei einer Wortlänge von 8 Bit die Zahlen* 10 *und* -10 *durch folgende Bitmuster repräsentiert:* 00001010 *und* 10001010.

Wenn das linkeste Bit das Vorzeichen bezeichnet, ergibt sich daraus, dass es zwei Bitmuster für die Darstellung der Zahl 0 gibt: 10000000 und 00000000.

In dieser Darstellung lassen sich bei einer Wortlänge n die Zahlen von $-2^{n-1} - 1$ bis $2^{n-1} - 1$ darstellen.

Die Lösung der Darstellung mit Vorzeichen und Betrag erschwert das Rechnen. Wir müssen zwei Verfahren bereitstellen: eines zum Addieren und eines zum Subtrahieren (so wie wir in der Schule schriftliches Addieren und Subtrahieren getrennt gelernt haben).

Beispiel 3.1.3 *Versuchen wir, das gängige Additionsverfahren für* 10 *und* -10 *in der Vorzeichendarstellung anzuwenden, so erhalten wir:*

$$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

Das Ergebnis stellt keinenfalls die Zahl 0 *dar, sondern die Zahl* -20.

Es läßt sich kein einheitlicher Algorithmus für die Addition in dieser Darstellung finden.

[3] Eine etwas unglückliche Namensgebung aus der Mischung eines griechischen mit einem lateinischen Wort.
[4] Wobei wir großzügig den Typ `char` ignorieren.

Einerkomplement

Ausgehend von der Idee, dass man eine Zahlendarstellung sucht, in der allein durch das bekannte Additionsverfahren auch mit negativen Zahlen korrekt gerechnet wird, kann man das Verfahren des Einerkomplements wählen. Die Idee des Einerkomplements ist, dass für jede Zahl die entsprechende negative Zahl so dargestellt wird, indem jedes Bit gerade andersherum gesetzt ist.

Beispiel 3.1.4 *Bei einer Wortlänge von 8 Bit werden die Zahlen* 10 *und* -10 *durch folgende Bitmuster dargestellt:* 00001010 *und* 11110101*.*

Jetzt können auch negative Zahlen mit dem gängigen Additionsverfahren addiert werden, also kann die Subtraktion durch ein Additionsverfahren durchgeführt werden.

$$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$$

Das errechnete Bitmuster stellt die negative *Null dar.*

In der Einerkomplementdarstellung läßt sich zwar fein rechnen, wir haben aber immer noch zwei Bitmuster zur Darstellung der 0. Für eine Wortlänge n lassen sich auch wieder die Zahlen von $-2^{n-1}-1$ bis $2^{n-1}-1$ darstellen..

Ebenso wie in der Darstellung mit Vorzeichen und Betrag erkennt man in der Einerkomplementdarstellung am linkesten Bit, ob es sich um eine negative oder um eine positive Zahl handelt.

Zweierkomplement

Die Zweierkomplementdarstellung verfeinert die Einerkomplementdarstellung, so dass es nur noch ein Bitmuster für die Null gibt. Im Zweierkomplement wird für eine Zahl die negative Zahl gebildet, indem zu ihrer Einerkomplementdarstellung noch 1 hinzuaddiert wird.

Beispiel 3.1.5 *Bei einer Wortlänge von 8 Bit werden die Zahlen* 10 *und* -10 *durch folgende Bitmuster dargestellt:* 00001010 *und* 11110110*.*

Jetzt können weiterhin auch negative Zahlen mit dem gängigen Additionsverfahren addiert werden, also kann die Subtraktion durch ein Additionsverfahren durchgeführt werden.

$$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Das errechnete Bitmuster stellt die Null dar.

Die *negative* Null aus dem Einerkomplement stellt im Zweierkomplement keine Null dar, sondern die Zahl -1, wie man sich vergegenwärtigen kann, wenn man von der 1 das Zweierkomplement bildet.

Das Zweierkomplement ist die in heutigen Rechenanlagen gebräuchlichste Form der Zahlendarstellung für ganze Zahlen. In modernen Programmiersprachen spiegelt sich dieses in den Wertebereichen primitiver Zahlentypen wieder. So kennt Java 4 Typen zur Darstellung ganzer Zahlen, die sich lediglich in der Anzahl der Ziffern unterscheiden. Die Zahlen werden intern als Zweierkomplement dargestellt.

Typname	Länge	Wertebereich
`byte`	8 Bit	$-128 = -2^7$ bis $127 = 2^7 - 1$
`short`	16 Bit	$-32768 = -2^{15}$ bis $32767 = 2^{15} - 1$
`int`	32 Bit	$-2147483648 = -2^{31}$ bis $2147483647 = 2^{32} - 1$
`long`	64 Bit	-9223372036854775808 bis 9223372036854775807

In der Programmiersprache Java sind die konkreten Wertebereiche für die einzelnen primitiven Typen in der Spezifikation festgelegt. In anderen Programmiersprachen wie z.B. C ist dies nicht der Fall. Hier hängt es vom Compiler und dem konkreten Rechner ab, welchen Wertebereich die entsprechenden Typen haben.

Es gibt Programmiersprachen wie z.B. Haskell[PJ03], in denen es einen Typ gibt, der potentiell ganze Zahlen von beliebiger Größe darstellen kann.

Aufgabe 15 Starten Sie folgendes Javaprogramm:

TestInteger.java

```
1 class TestInteger {
2   public static void main(String [] _){
3     System.out.println(2147483647+1);
4     System.out.println(-2147483648-1);
5   }
6 }
```

Erklären Sie die Ausgabe.

Frage eines Softwaremenschen an die Hardwarebauer

Gängige Spezifikationen moderner Programmiersprachen sehen einen ausgezeichneten Wert *nan* für *not a number* in der Arithmetik vor, der ausdrücken soll, dass es sich bei dem Wert nicht mehr um eine darstellbare Zahl handelt. In der Arithmetik moderne Prozessoren ist ein solcher zusätzlicher Wert nicht vorgesehen. Warum eigentlich nicht?! Es sollte doch möglich sein, einen Prozessor zu bauen, der beim Überlauf des Wertebereichs nicht stillschweigend das durch den Überlauf entstandene Bitmuster wieder als Zahl interpretiert, sondern den ausgezeichneten Wert *nan* als Ergebnis hat. Ein Programmierer könnte dann entscheiden, wie er in diesem Fall verfahren möchte. Eine große Fehlerquelle wäre behoben. Warum ist eine solche Arithmetik noch nicht in handelsüblichen Prozessoren verwirklicht?

3.1.5 Kommazahlen

Wollen wir Kommazahlen, also Zahlen aus den Mengen $\mathbb{Q}$ und $\mathbb{R}$, im Rechner darstellen, so stoßen wir auf ein zusätzliches Problem: es gilt dann nämlich nicht mehr, dass ein Intervall nur endlich viele Werte enthält, wie es für ganze Zahlen noch der Fall ist. Bei ganzen Zahlen konnten wir immerhin wenigstens alle Zahlen eines bestimmten Intervalls darstellen. Wir können also nur endlich viele dieser unendlich vielen Werte in einem Intervall darstellen. Wir werden also das Intervall *diskretisieren*.

Festkommazahlen

Vernachlässigen wir für einen Moment jetzt einmal wieder negative Zahlen. Eine einfache und naheliegende Idee ist, bei einer Anzahl von n Bit, die für die Darstellung der Kommazahlen zur Verfügung stehen, einen Teil davon für den Anteil vor dem Komma und den Rest für den Anteil nach dem Komma zu benutzen. Liegt das Komma in der Mitte, so können wir eine Zahl aus n Ziffern durch folgende Formel ausdrücken:

Der Wert der Zahl einer Zeichenkette $a_{n-1} \dots a_{n/2}, a_{n/2-1} \dots a_0$ berechnet sich für die Basis b nach folgender Formel:

$$\sum_{i=0}^{n-1} a_i^{i-n/2} = a_0 * b^{-n/2} + \dots + a_{n-1} * b^{n-1-n/2}$$

Wir kennen diese Darstellung aus dem im Alltag gebräuchlichen Zehnersystem. Für Festkommazahlen ergibt sich ein überraschendes Phänomen. Zahlen, die sich bezüglich einer Basis darstellen lassen, sind bezüglich einer anderen Basis nicht darstellbar. So läßt sich schon die sehr einfach zur Basis 10 darstellbare Zahl $0,1$ nicht zur Basis 2 als Festkommazahl darstellen, und umgekehrt können Sie einfach zur Basis 2 als Festkommazahl darstellbare Zahlen nicht zur Basis 10 darstellen.

Dem Leser wird natürlich nicht entgangen sein, dass wir keine irrationale Zahl über die Festkommadarstellung darstellen können.

Festkommazahlen spielen in der Rechnerarchitektur kaum eine Rolle. Ein sehr verbreitetes Anwendungsgebiet für Festkommazahlen sind Währungsbeträge. Hier interessieren in der Regel nur die ersten zwei oder drei Dezimalstellen nach dem Komma.

Fließkommazahlen

Eine Alternative zu der Festkommadarstellung von Zahlen ist die Fließkommadarstellung. Während die Festkommadarstellung einen Zahlenbereich der rationalen Zahlen in einem festen Intervall durch diskrete, äquidistant verteilte Werte darstellen kann, sind die diskreten Werte in der Fließkommadarstellung nicht gleich verteilt.

In der Fließkommadarstellung wird eine Zahl durch zwei Zahlen charakterisiert und ist bezüglich einer Basis b:

- die Mantisse für die darstellbaren Ziffern. Die Mantisse charakterisiert die Genauigkeit der Fließkommazahl.
- der Exponent, der angibt, wie weit die Mantisse hinter bzw. vor dem Komma liegt.

Aus Mantisse m, Basis b und Exponent exp ergibt sich die dargestellte Zahl durch folgende Formel:

$$z = m * b^{exp}$$

Damit lassen sich mit Fließkommazahlen sehr große und sehr kleine Zahlen darstellen. Je größer jedoch die Zahlen werden, desto weiter liegen sie von der nächsten Zahl entfernt.

Für die Fließkommadarstellung gibt es in Java zwei Zahlentypen, die nach der Spezifikation des IEEE 754-1985 gebildet werden:

- **float**: 32 Bit Fließkommazahl nach IEEE 754. Kleinste positive Zahl: 2^{-149}. Größte positive Zahl: $(2 - 2^{-23})*^{127}$
- **double**: 64 Bit Fließkommazahl nach IEEE 754. Kleinste positive Zahl: 2^{-1074}. Größte positive Zahl: $(2 - 2^{-52})*^{1023}$

Im Format für `double` steht das erste Bit für das Vorzeichen, die nächsten 11 Bit markieren den Exponenten und die restlichen 52 Bit kodieren die Mantisse.

Im Format für `float` steht das erste Bit für das Vorzeichen, die nächsten 8 Bit markieren den Exponenten und die restlichen 23 Bit kodieren die Mantisse.

Bestimmte Bitmuster charakterisieren einen Wert für negative und positive unbeschränkte Werte (unendlich) sowie Zahlen, Bitmuster, die charakterisieren, dass es sich nicht mehr um eine Zahl handelt.

Beispiel 3.1.6 *Der folgende Test zeigt, dass bei einer Addition von zwei Fließkommazahlen die kleinere Zahl das Nachsehen hat:*

DoubleTest.java

```
class DoubleTest{
  public static void main(String [] args){
    double x = 325e200;
    double y = 325e-200;
    System.out.println(x);
    System.out.println(y);
    System.out.println(x+y);
    System.out.println(x+100000);
  }
}
```

Wie man an der Ausgabe erkennen kann: selbst die Addition der Zahl `100000` *bewirkt keine Veränderung auf einer großen Fließkommazahl:*

```
sep@linux:~/fh/prog3/examples/src> java DoubleTest
3.25E202
3.25E-198
3.25E202
3.25E202
sep@linux:~/fh/prog3/examples/src>
```

Beispiel 3.1.7 *Das folgende kleine Beispiel zeigt, inwieweit und für den Benutzer oft auf überraschende Weise die Fließkommadarstellung zu Rundungen führt:*

Rounded.java

```
class Rounded {
  public static void main(String [] args){
    System.out.println(8f);
    System.out.println(88f);
    System.out.println(8888f);
    System.out.println(88888f);
    System.out.println(888888f);
    System.out.println(8888888f);
    System.out.println(88888888f);
    System.out.println(888888888f);
    System.out.println(8888888888f);
    System.out.println(88888888888f);
    System.out.println(888888888888f);

    System.out.println(1f+1000000000000f-1000000000000f);
  }
}

```

Das Programm hat die folgende Ausgabe. Insbesondere in der letzten Zeile fällt auf, dass Addition und anschließende Subtraktion ein und derselben Zahl nicht die Identität ist. Für Fließkommazahlen gilt nicht: $x + y - y = x$.

```
sep@linux:~/fh/prog3/examples/src> java Rounded
8.0
88.0
8888.0
88888.0
888888.0
8888888.0
8.8888888E7
8.888889E8
8.8888893E9
8.8888885E10
8.8888889E11
8.8888889E12
0.0
sep@linux:~/fh/prog3/examples/src>
```

3.1.6 Der Typ: boolean

Eine in der Informatik häufig gebrauchte Unterscheidung ist, ob etwas wahr oder falsch ist, also eine Unterscheidung zwischen genau zwei Werten.[5] Hierzu bedient man sich der Wahrheitswerte aus der formalen Logik. Java bietet einen primitiven Typ an, der genau zwei Werte annehmen kann, den Typ: `boolean`. Die Bezeichnung ist nach dem englischen Mathematiker und Logiker George Boole (1815–1869) gewählt worden.

Entsprechend der zwei möglichen Werte für diesen Typ stellt Java auch zwei Literale zur Verfügung: `true` und `false`.

Bool'sche Daten lassen sich ebenso wie numerische Daten benutzen. Felder und Methoden können diesen Typ verwenden.

Testbool.java

```
1  class Testbool{
2    boolean boolFeld;
3
4    Testbool(boolean b){
5      boolFeld = b;
6    }
7
8    boolean getBoolFeld(){
9      return this.boolFeld;
10   }
11
12   public static void main(String [] args){
13     Testbool t = new Testbool(true);
14     System.out.println(t.getBoolFeld());
15     t = new Testbool(false);
16     System.out.println(t.getBoolFeld());
17   }
18 }
```

[5] Wahr oder falsch, eine dritte Möglikchkeit gibt es nicht. Logiker postulieren dieses als *tertium non datur*.

3.1.7 Boxen für primitive Typen

Javas Typsystem ist etwas zweigeteilt. Es gibt Objekttypen und primitive Typen. Die Daten der primitiven Typen stellen keine Objekte dar. Für jeden primitiven Typen gibt es allerdings im Paket `java.lang` eine Klasse, die es erlaubt, Daten eines primitiven Typs als Objekt zu speichern. Dieses sind die Klassen `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` und `Character`. Objekte dieser Klassen sind nicht modifizierbar. Einmal ein Integer-Objekt mit einer bestimmten Zahl erzeugt, läßt sich die in diesem Objekt erzeugte Zahl nicht mehr verändern.

Wollte man in bisherigen Klassen ein Datum eines primitiven Typen in einer Variablen speichern, die nur Objekte speichern kann, so musste man es in einem Objekt der entsprechenden Klasse kapseln. Man spricht von *boxing*. Es kam zu Konstruktoraufrufen dieser Klassen. Sollte später mit Operatoren auf den Zahlen, die durch solche gekapselten Objekte ausgedrückt wurden, gerechnet werden, so war der primitive Wer mit einem Methodenaufruf aus dem Objekt wieder zu extrahieren, dem sogenannten *unboxing*. Es kam zu Aufrufen von Methoden wie `intValue()` im Code.

Beispiel 3.1.8 *In diesem Beispiel sieht man das manuelle Verpacken und Auspacken primitiver Daten.*

ManualBoxing.java

```
1  package name.panitz.boxing;
2  public class ManualBoxing{
3    public static void main(String [] _){
4      int i1 = 42;
5      Object o = new Integer(i1);
6      System.out.println(o);
7      Integer i2 = new Integer(17);
8      Integer i3 = new Integer(4);
9      int i4 = 21;
10     System.out.println((i2.intValue()+i3.intValue())*i4);
11   }
12 }
```

In Java können die primitiven Typen mit ihren entsprechenden Klassen synonym verwendet werden. Nach außen hin werden die primitiven Typen auch zu Objekttypen. Der Übersetzer nimmt fügt die notwendigen *boxing*- und *unboxing*-Operationen vor.

Beispiel 3.1.9 *Jetzt das vorherige kleine Programm ohne explizite* boxing- *und* unboxing-*Aufrufe.*

AutomaticBoxing.java

```
1  package name.panitz.boxing;
2  public class AutomaticBoxing{
3    public static void main(String [] _){
4      int i1 = 42;
5      Object o = i1;
6      System.out.println(o);
7      Integer i2 = 17;
8      Integer i3 = 4;
9      int i4 = 21;
10     System.out.println((i2+i3)*i4);
11   }
12 }
```

3.2 Ausdrücke

Wir haben jetzt gesehen, was Java uns für Typen zur Darstellung von Zahlen zur Verfügung stellt. Jetzt wollen wir mit diesen Zahlen nach Möglichkeit auch noch rechnen können. Hierzu stellt Java eine feste Anzahl von Operatoren wie `*, -, /` etc. zur Verfügung. Prinzipell gibt es in der Informatik für Operatoren drei mögliche Schreibweisen:

- **Präfix:** Der Operator wird vor den Operanden geschrieben, also z.B. `(* 2 21)`. Im ursprünglichen Lisp gab es die Prefixnotation für Operatoren.
- **Postfix:** Der Operator folgt den Operanden, also z.B. `(21 2 *)`. Forth und Postscript sind Beispiele von Sprachen mit Postfixnotation.
- **Infix:** Der Operator steht zwischen den Operanden. Dieses ist die gängige Schreibweise in der Mathematik und für das Auge die gewohnteste. Aus diesem Grunde bedient sich Java der Infixnotation: `42 * 2`.

3.2.1 Die Grundrechenarten

Java stellt für Zahlen die vier Grundrechenarten zur Verfügung.

Bei der Infixnotation gelten für die vier Grundrechenarten die üblichen Regeln der Bindung, nämlich Punktrechnung vor Strichrechnung. Möchte man diese Regel durchbrechen, so sind Unterausdrücke in Klammern zu setzen. Folgende kleine Klasse demonstriert den Unterschied:

PunktVorStrich.java

```
1 class PunktVorStrich{
2   public static void main(String [] args){
3     System.out.println(17 + 4 * 2);
4     System.out.println((17 + 4) * 2);
5   }
6 }
```

Wir können nun also Methoden schreiben, die Rechnungen vornehmen. In der folgenden Klasse definieren wir z.B. eine Methode zum Berechnen der Quadratzahl der Eingabe:

Square.java

```
1 class Square{
2   static int square(int i){
3     return i*i;
4   }
5 }
```

Division To e done!

3.2.2 Vergleichsoperatoren

Obige Operatoren rechnen jeweils auf zwei Zahlen und ergeben wieder eine Zahl als Ergebnis. Vergleichsoperatoren vergleichen zwei Zahlen und geben einen bool'schen Wert, der angibt, ob der Vergleich wahr oder falsch ist. Java stellt die folgenden Vergleichsoperatoren zur Verfügung: `<, <=, >, >=, !=, ==`. Für die Gleichheit ist in Java das doppelte Gleichheitszeichen `==` zu schreiben, denn das einfache Gleichheitszeichen ist bereits für den Zuweisungsbefehl vergeben. Die Ungleichheit wird mit `!=` bezeichnet.

Folgende Tests demonstrieren die Benutzung der Vergleichsoperatoren:

Vergleich.java
```
class Vergleich{

  public static void main(String[] args){
    System.out.println(1+1 < 42);
    System.out.println(1+1 <= 42);
    System.out.println(1+1 >  42);
    System.out.println(1+1 >= 42);
    System.out.println(1+1 == 42);
    System.out.println(1+1 != 42);
  }
}
```

3.2.3 Bool'sche Operatoren

In der bool'schen Logik gibt es eine ganze Reihe von binären Operatoren für logische Ausdrücke. Für zwei davon stellt Java auch Operatoren bereit: `&&` für das logische und ($\wedge$) und `||` für das logische oder ($\vee$).

Zusätzlich kennt Java noch den unären Operator der logischen Negation $\neg$. Er wird in Java mit `!` bezeichnet.

Wie man im folgenden Test sehen kann, gibt es auch unter den bool'schen Operatoren eine Bindungspräzedenz, ähnlich wie bei der Regel Punktrechnung vor Strichrechnung. Der Operator `&&` bindet stärker als der Operator `||`:

TestboolOperator.java
```
class TestboolOperator{
  public static void main(String [] args){
    System.out.println(true && false);
    System.out.println(true || false);
    System.out.println(!true || false);
    System.out.println(true || true && false);
  }
}
```

In der formalen Logik kennt man noch weitere Operatoren, z.B. die Implikation $\rightarrow$. Diese Operatoren lassen sich aber durch die in Java zur Verfügung stehenden Operatoren ausdrücken. $A \rightarrow B$ entspricht $\neg A \vee B$. Wir können somit eine Methode schreiben, die die logische Implikation testet:

TestboolOperator2.java
```
class TestboolOperator2{
  static boolean implication(boolean a, boolean b){
    return !a || b;
  }

  public static void main(String [] args){
    System.out.println(implication(true, false));
  }
}
```

3.2.4 Der Bedingungsoperator

Java kennt auch einen Operator mit drei Operanden. Er besteht aus zwei einzelnen Zeichen, die als Trenner zwischen den Operanden stehen. Zunächst kommt der erste Operand, dann das

Zeichen `?`, dann der zweite Operand, gefolgt vom Zeichen `:`, dem der dritte Operand folgt. Schematisch sehen die Ausdrücke dieses Operators wie folgt aus:

$cond\ ?\ alt_1\ :\ alt_2$

Das Ergebnis dieses Operators wird wie folgt berechnet: Der erste Operand wird zu einem Wahrheitswert ausgewertet. Wenn dieser *wahr* ist, so wird der zweite Operand als Ergebnis ausgewertet, wenn er *falsch* ist, wird der dritte Operand als Ergebnis ausgewertet.

SoOderSo.java

```
class SoOderSo{
  public static void  main(String[] args){
    System.out.println((17+4)*2==42?"tatsaechlich gleich":"unterschiedlich");
  }
}
```

Der Bedingungsoperator ist unser erstes Konstrukt, um Verzweigungen auszudrücken. Da der Bedingungsoperator auch einen Wert errechnet, können wir diesen benutzen, um mit ihm weiterzurechenen. Der Bedingungsoperator kann also tatsächlich ineinandergesteckt werden:

Signum1.java

```
class Signum1{
  public static void  main(String[] args){
    System.out.println("signum(42) = "+((42>0)?1:((42<0)?-1:0)));
  }
}
```

Hier wird zunächst geschaut, ob die Zahl 42 größer als 0 ist. Ist dieses der Fall wird die Zahl 1 ausgegeben, ansonsten wird weitergeschaut, ob die Zahl 42 kleiner 1 ist. Hier wird im Erfolgsfall die Zahl -1 ausgegeben. Wenn beides nicht der Fall war, wird die Zahl 0 ausgegeben.

Zugegebener Maßen ist dieser Ausdruck schon schwer zu lesen. Wir werden später bessere Konstrukte kennenlernen, um verschiedene Fälle zu unterscheiden.

3.2.5 Funktionen

Im letzten Abschnitt haben wir die wichtigsten Operatoren in Java kennengelernt. Mit diesen lassen sich im Prinzip schon komplexe Berechnungen durchführen. Allerdings können uns die theoretischen Informatiker beweisen, dass wir mit den Operatorausdrücken allein noch nicht mächtig genug sind, um alle erdenklichen berechenbaren mathematischen Funktionen berechnen zu können. Hierzu Bedarf es noch dem Konzept der Funktionen. Betrachten wir hierzu eine Funktion im klassischen mathematischen Sinn: eine Funktion hat eine Reihe von Argumenten und berechnet für die Argumente ein Ergebnis. Wichtig ist, dass eine Funktion immer für die gleichen Argument ein und dasselbe Ergebnis auf deterministische Weise errechnet. Die Anzahl der Argumente einer Funktion wird als ihre Stelligkeit bezeichnet.

Im folgenden Beispiel wird die Funktion `quadrat` definiert. Sie hat ein Argument. Als Ergebnis gibt sie das Quadrat dieser Zahl zurück. Nachdem diese Funktion definiert wurde, benutzen wir sie in der Funktion `main` mehrfach, jeweils mit anderen Argumenten.

Quadrat.java

```
class Quadrat{
  static int quadrat(int x){
    return x*x;
  }

  public static void main(String[] args){
    System.out.println(quadrat(5));
    System.out.println(quadrat(10));
    System.out.println(quadrat(0));
```

```
10      System.out.println(quadrat(-12));
11    }
12  }
```

Wichtig bei der Definition einer Funktion ist, der `return`-Befehl. Eine Funktion berechnet ein Ergebnis. Dieses Ergebnis ist der Ausdruck, der hinter dem Befehlswort `return` steht.

Java wertet jetzt jedesmal wenn eine Funktion benutzt wird, diesen Aufruf aus und ersetzt ihn quasi durch sein Ergebnis, nämlich den Ausdruck, der hinter dem Befehlswort `return` steht.

In Programmiersprachen werden die Argumente einer Funktion als Parameter bezeichnet. Bei der Definition der Funktion spricht man von den formalen Parametern, in unserem Falle das `x`. Bei der Definition der formalen Parameter ist vor seinem Namen, sein Typ zu schreiben.

Beim Aufruf der Funktion `quadrat` wird nun das `x` durch einen konkreten Ausdruck, z.B. die Zahl `5` ersetzt.

In der Wahl des Namens für einen formalen Parameter ist der Programmierer weitgehendst frei. Dieser Name existiert nur innerhalb des Funktionsrumpfes. Verschiedene Funktionen können Parameter gleichen Namens haben. Diese Parameter sind dann trotzdem vollkommen unabhängig.

Wir können in Java beliebig viele Funktionen definieren, Wir können, nachdem eine Funktion einmal definiert ist, sie beliebig oft und an fast beliebig vielen Stellen aufrufen, sogar mehrfach verschachtelt:

QuadratUndDoppel.java

```
1  class QuadratUndDoppel{
2
3    static int quadrat(int x){
4      return x*x;
5    }
6    static int doppelt(int x){
7      return x*x;
8    }
9
10   public static void main(String[] args){
11     System.out.println(doppelt(quadrat(5)));
12   }
13 }
```

Hier wird in der Funktion `main` zunächst die Funktion `quadrat` mit der Zahl `5` aufgerufen, das Ergebnis (die `25`) wird dann als Argument der Funktion `doppelt` benutzt.

Innerhalb einer Funktion können auch andere Funktionen aufgerufen werden.

SummeDerQuadrate.java

```
1  class SummeDerQuadrate{
2    static int quadrat(int x){
3      return x*x;
4    }
5
6    static int summeDerQuadrate(int x,int y){
7      return quadrat(x)+quadrat(y);
8    }
9
10   public static void main(String[] args){
11     System.out.println(summeDerQuadrate(3,4));
12   }
13 }
```

In diesem Programm ruft die zweistellige Funktion `summeDerQuadrate` für jedes ihrer Argumente die Funktion `quadrat` auf, und addiert anschließend die Ergebnisse.

Die erste Zeile (bis zur öffnenden Klammer) einer Funktion wird als Signatur bezeichnet. Sie beinhaltet den Rückgabetyp, den Namen und die Argumente der Funktion. Alles innerhalb der geschweiften Klammern wird als Rumpf bezeichnet.

Die Berechnungen, die eine Funktion vornimmt, können beliebig komplexe Ausdrücke sein:

Signum2.java

```
class Signum2{
  static int signum(int i){
    return (i>0)?1:((i<0)?-1:0);
  }
  public static void main(String[] args){
    System.out.println("signum(42) = "+signum(42));
    System.out.println("signum(-42) = "+signum(-42));
  }
}
```

Aufgabe 16 Schreiben Sie eine Funktion, mit zwei ganzzahligen Parametern. Die Rückgabe soll die größere der beiden Zahlen sein. Testen Sie die Funktionen in einer Hauptfunktion mit mindesten drei Aufrufen.

3.2.6 Rekursive Funktionen

Sobald wir die Signatur einer Funktion oder Prozedur definiert haben, dürfen wir sie benutzen, sprich aufrufen. Damit ergibt sich eine sehr mächtige Möglichkeit der Programmierung. Wir können Funktionen bereits in ihren eigenen Rumpf aufrufen. Solche Funktionen werden rekursiv genannt. *Recurrere* ist das lateinische Wort für zurücklaufen. Eine rekursive Funktion läuft während ihrer Auswertung wieder zu sich selbst zurück.

Damit lassen sich wiederholt Programmteile ausführen. Das folgende Programm wird z.B. nicht müde, uns mit dem Wort `hallo` zu erfreuen.

HalloNerver.java

```
class HalloNerver{
  static void halloNerver(){
    System.out.println("hallo");
    halloNerver();
  }

  public static void main(String[] args){
    halloNerver();
  }
}
```

Die Funktion `main` ruft die Funktion `halloNerver` auf. Diese druckt einmal das Wort `hallo` auf die Konsole und ruft sich dann selbst wieder auf. Dadurch wird wieder `hallo` auf die Konsole geschrieben und so weiter. Wir haben ein endlos laufendes Programm. Tatsächlich endlos? Lassen sie es mal möglichst lange auf Ihrem Rechner laufen.

Was zunächst wie eine Spielerei anmutet, kann verfeinert werden, indem mit Hilfe eines Arguments mitgezählt wird, wie oft die Prozedur bereits rekursiv aufgerufen wurde:

HalloZaehler.java

```
class HalloZaehler{
  static void halloZaehler(int i){
    System.out.println("hallo "+i);
```

```
4     halloZaehler(i+1);
5   }
6
7   public static void main(String[] args){
8     halloZaehler(1);
9   }
10 }
```

Auch dieses Programm läuft endlos.

Mit dem Bedingungsoperator haben wir aber ein Werkzeug, um zu verhindern, dass rekursive Programme endlos laufen. Wir können den Bedingungsoperator nämlich benutzen, um zu unterscheiden, ob wir ein weiteres Mal einen rekursiven Aufruf vornehmen wollen, oder nicht mehr.

Hierzu folgt ein kleines Programm, in dem die rekursive Prozedur eine Zahl als Parameter enthält. Zunächst druckt die Prozedur den Wert des Parameters auf der Kommandozeile aus, dann unterscheidet die Prozedur. Wenn der Parameter den Wert `0` hat, dann wird nur noch das Wort `ende` ausgedruckt, ansonsten kommt es zu einen rekursiven Aufruf. Für den rekursiven Aufruf wird der Parameter eins kleiner genommen, als beim Originalaufruf.

CountDown.java

```
1  class CountDown{
2    static String countDown(int i){
3      System.out.println("hallo "+i);
4      System.out.print(i==0?"":countDown(i-1));
5      return "";
6    }
7    public static void main(String[] args){
8      countDown(10);
9    }
10 }
```

Damit verringert sich in jedem rekursiven Aufruf der Parameter um eins. Wenn er anfänglich positiv war, wird er schließlich irgendwann einmal `0` sein. Dann verhindert der Bedingungsoperator einen weiteren rekursiven Aufruf. Die Auswertung endet. Man sagt, das Programm terminiert.

In diesem Zusammenhang gibt es mehrere interessante Ergebnisse aus der theoretischen Informatik. Mit der Möglichkeit rekursive Funktionen zu schreiben und mit der Möglichkeit über den Bedingungsoperator zu entscheiden, wie weiter ausgewertet wird, hat eine Programmiersprache die Mächtigkeit, dass mit ihr alle berechenbaren mathematischen Funktionen programmiert werden können.

Allerdings haben wir dafür auch einen Preis zu zahlen: wir können jetzt Programme schreiben, die nicht terminieren. Und das muss auch so sein, denn nach Ergebnissen der Theoretiker, muss es in einer Programmiersprache, die alle berechenbaren mathematischen Funktionen ausdrücken kann, auch Programme geben, die nicht terminieren.

Nach soviel esoterisch anmutenden Ausflügen in die Theorie, wollen wir nun auch eine rekursive Funktion schreiben, die etwas interessantes berechnet. Hierzu schreiben wir einmal die Fakultätsfunktion, die mathematisch definiert ist als:

$$fac(n) = \begin{cases} 1 & \text{für } n = 0 \\ n * fac(n-1) & \text{für } n > 1 \end{cases}$$

Diese Definition lässt sich direkt in ein Java Programm umsetzen:

Fakultaet.java

```
1 class Fakultaet{
2   static int fac(int i){
3     return i==0?1:i*fac(i-1);
4   }
5   public static void main(String[] args){
6     System.out.println("fac(5) = "+fac(5));
7   }
8 }
```

Wir können dieses Programm von Hand ausführen, indem wir den Methodenaufruf für `fac` für einen konkreten Parameter `i` durch die für diesen Wert zutreffende Alternative der Bedingungsabfrage ersetzen. Wir kennzeichnen einen solchen Ersetzungsschritt durch einen Pfeil $\rightarrow$:

```
fac(4)
→4*fac(4-1)
→4*fac(3)
→4*(3*fac(3-1))
→4*(3*fac(2))
→4*(3*(2*fac(2-1)))
→4*(3*(2*fac(1)))
→4*(3*(2*(1*fac(1-1))))
→4*(3*(2*(1*fac(0))))
→4*(3*(2*(1*1)))
→4*(3*(2*1))
→4*(3*2)
→4*6
→24
```

Aufgabe 17 Nehmen Sie das Programm zur Berechnung der Fakultät und testen sie es mit verschiedenen Argumenten.

Aufgabe 18 Im Bestseller *Sakrileg (der Da Vinci Code)* spielen die Fibonaccizahlen eine Rolle. Für eine natürliche Zahl n ist ihre Fibonaccizahl definiert durch:

$$f(n) = \begin{cases} n & \text{für } n \leq 1 \\ f(n-1) + f(n-2) & \text{für } n > 1 \end{cases}$$

Programmieren Sie eine Funktion `int fib(int n)`, die für eine Zahl, die entsprechende Fibonaccizahl zurückgibt.

Geben Sie in der Hauptfunktion die ersten 20 Fibonaccizahlen aus.

3.3 Zusammengesetzte Befehle

Streng genommen kennen wir bisher nur einen Befehl, den Zuweisungsbefehl, der einem Feld einen neuen Wert zuweist.[6] In diesem Abschnitt lernen wir weitere Befehle kennen. Diese Befehle sind in dem Sinne zusammengesetzt, dass sie andere Befehle als Unterbefehle haben.

[6] Konstrukte mit Operatoren nennt man dagegen Ausdrücke.

3.3.1 Bedingungsabfrage mit: if

Ein häufig benötigtes Konstrukt ist, dass ein Programm abhängig von einer bool'schen Bedingung sich verschieden verhält. Hierzu stellt Java die **if**-Bedingung zur Verfügung. Dem Schlüsselwort `if` folgt in Klammern eine bool'sche Bedingung, anschließend kommen in geschweiften Klammern die Befehle, die auszuführen sind, wenn die Bedingung wahr ist. Anschließend kann optional das Schlüsselwort `else` folgen mit den Befehlen, die andernfalls auszuführen sind:

FirstIf.java

```
1  class FirstIf {
2
3    static void firstIf(boolean bedingung){
4      if (bedingung) {
5        System.out.println("Bedingung ist wahr");
6      } else {
7        System.out.println("Bedingung ist falsch");
8      }
9    }
10
11   public static void main(String [] args){
12      firstIf(true || false);
13   }
14
15 }
```

Das `if`-Konstrukt erlaubt es uns also, Fallunterscheidungen zu treffen. Wenn in den Alternativen nur ein Befehl steht, so können die geschweiften Klammern auch fortgelassen werden. Unser Beispiel läßt sich also auch schreiben als:

FirstIf2.java

```
1  class FirstIf2 {
2
3    static void firstIf(boolean bedingung){
4      if (bedingung) System.out.println("Bedingung ist wahr");
5      else System.out.println("Bedingung ist falsch");
6    }
7
8    public static void main(String [] args){
9       firstIf(true || false);
10   }
11
12 }
```

Eine Folge von mehreren `if`-Konstrukten läßt sich auch direkt hintereinanderschreiben, so dass eine Kette von `if`- und `else`-Klauseln entsteht:

ElseIf.java

```
1  class ElseIf {
2
3    static String lessOrEq(int i,int j){
4      if (i<10)  return "i kleiner zehn";
5      else if (i>10)  return "i größer zehn";
6      else if (j>10)  return "j größer zehn";
7      else if (j<10)  return "j kleiner zehn";
8      else return "j=i=10";
9    }
10
11   public static void main(String [] args){
12      System.out.println(lessOrEq(10,9));
```

```
13    }
14
15 }
```

Wenn zuviele `if`-Bedingungen in einem Programm einander folgen und ineinander verschachtelt sind, dann wird das Programm schnell unübersichtlich. Man spricht auch von *Spaghetti-code*. In der Regel empfiehlt es sich, in solchen Fällen noch einmal über das Design nachzudenken, ob die abgefragten Bedingungen sich nicht durch verschiedene Klassen mit eigenen Methoden darstellen lassen.

Mit der Möglichkeit, in dem Programm abhängig von einer Bedingung unterschiedlich weiterzurechnen, haben wir theoretisch die Möglichkeit, alle durch ein Computerprogramm berechenbaren mathematischen Funktionen zu programmieren. So können wir z.B. eine Methode schreiben, die für eine Zahl `i` die Summe von `1` bis `n` berechnet:

Summe.java

```
1 class Summe{
2   static int summe(int i){
3     if (i==1)  {
4       return 1;
5     }else {
6       return summe(i-1) + i;
7     }
8   }
9 }
```

Wir können dieses Programm von Hand ausführen, indem wir den Methodenaufruf für `summe` für einen konkreten Parameter `i` durch die für diesen Wert zutreffende Alternative der Bedingungsabfrage ersetzen. Wir kennzeichnen einen solchen Ersetzungsschritt durch einen Pfeil $\rightarrow$:

```
summe(4)
→summe(4-1)+4
→summe(3)+4
→summe(3-1)+3+4
→summe(2)+3+4
→summe(2-1)+2+3+4
→summe(1)+2+3+4
→1+2+3+4
→3+3+4
→6+4
→10
```

Wie man sieht, wird für `i=4` die Methode `summe` genau viermal wieder aufgerufen, bis schließlich die Alternative mit dem konstanten Rückgabewert `1` zutrifft. Unser Trick war, im Methodenrumpf die Methode, die wir gerade definieren, bereits zu benutzen. Diesen Trick nennt man in der Informatik *Rekursion*. Mit diesem Trick ist es uns möglich, ein Programm zu schreiben, bei dessen Ausführung ein bestimmter Teil des Programms mehrfach durchlaufen wird.

Das wiederholte Durchlaufen von einem Programmteil ist das A und O der Programmierung. Daher stellen Programmiersprachen in der Regel Konstrukte zur Verfügung, mit denen man dieses direkt ausdrücken kann. Die Rekursion ist lediglich ein feiner Trick, dieses zu bewerkstelligen. In den folgenden Abschnitten lernen wir die zusammengesetzten Befehle von Java kennen, die es erlauben auszudrücken, dass ein Programmteil mehrfach zu durchlaufen ist.

Aufgabe 19 Ergänzen Sie ihre Klasse `Ausleihe` um eine Methode `void verlaengereEinenMonat()`, die den Rückgabetermin des Buches um einen Monat

erhöht.

Aufgabe 20 Modellieren und schreiben Sie eine Klasse `Counter`, die einen Zähler darstellt. Objekte dieser Klasse sollen folgende Funktionalität bereitsstellen:

- Eine Methode `click()`, die den internen Zähler um eins erhöht.
- Eine Methode `reset()`, die den Zähler wieder auf den Wert `0` setzt.
- Eine Methode, die den aktuellen Wert des Zählers ausgibt.

Testen Sie Ihre Klasse.

Aufgabe 21 Schreiben Sie mit den bisher vorgestellten Konzepten ein Programm, das unendlich oft das Wort `Hallo` auf den Bildschirm ausgibt. Was beobachten Sie, wenn sie das Programm lange laufen lassen?

Aufgabe 22 Schreiben Sie eine Methode, die für eine ganze Zahl die Fakultät dieser Zahl berechnet. Testen Sie die Methode zunächst mit kleinen Zahlen, anschließend mit großen Zahlen. Was stellen Sie fest?

Aufgabe 23 Modellieren und schreiben Sie eine Klasse, die ein Bankkonto darstellt. Auf das Bankkonto sollen Einzahlungen und Auszahlungen vorgenommen werden können. Es gibt einen maximalen Kreditrahmen. Das Konto soll also nicht beliebig viel in die Miese gehen können. Schließlich muss es eine Möglichkeit geben, Zinsen zu berechnen und dem Konto gutzuschreiben.

3.3.2 Iteration

Die im letzten Abschnitt kennengelernte Programmierung der Programmwiederholung durch Rekursion kommt ohne zusätzliche zusammengesetzte Befehle von Java aus. Da Rekursionen von der virtuellen Maschine Javas nur bis zu einem gewissen Maße unterstützt werden, bietet Java spezielle Befehle an, die es erlauben, einen Programmteil kontrolliert mehrfach zu durchlaufen. Die entsprechenden zusammengesetzten Befehle heißen Iterationsbefehle. Java kennt drei unterschiedliche Iterationsbefehle.

Schleifen mit: while

Ziel der Iterationsbefehle ist es, einen bestimmten Programmteil mehrfach zu durchlaufen. Hierzu ist es notwendig, eine Bedingung anzugeben, für wie lange eine Schleife zu durchlaufen ist. `while`-Schleifen in Java haben somit genau zwei Teile:

- die Bedingung
- und den Schleifenrumpf.

Java unterscheidet zwei Arten von `while`-Schleifen: Schleifen, für die vor dem Durchlaufen der Befehle des Rumpfes die Bedingung geprüft wird, und Schleifen, für die nach Durchlaufen des Rumpfes die Bedingung geprüft wird.

Vorgeprüfte Schleifen Die vorgeprüften Schleifen haben folgendes Schema in Java:

`while` (*pred*){*body*}

pred ist hierbei ein Ausdruck, der zu einem bool'schen Wert auswertet. *body* ist eine Folge von Befehlen. Java arbeitet die vorgeprüfte Schleife ab, indem erst die Bedingung *pred* ausgewertet wird. Ist das Ergebnis `true`, dann wird der Rumpf *(body)* der Schleife durchlaufen. Anschließend wird wieder die Bedingung geprüft. Dieses wiederholt sich so lange, bis die Bedingung zu `false` auswertet.

Ein simples Beispiel einer vorgeprüften Schleife ist folgendes Programm, das die Zahlen von 0 bis 9 auf dem Bildschirm ausgibt:

WhileTest.java

```
1 class WhileTest {
2   public static void main(String [] args){
3     int i = 0;
4     while (i < 10){
5       i = i+1;
6       System.out.println(i);
7     }
8   }
9 }
```

Mit diesen Mitteln können wir jetzt versuchen, die im letzten Abschnitt rekursiv geschriebene Methode `summe` iterativ zu schreiben:

Summe2.java

```
1  class Summe2 {
2    public static int summe(int n){
3
4      int erg = 0 ;              // Feld für Ergebnis.
5      int j   = n ;              // Feld zur Schleifenkontrolle.
6
7      while (j>0){               // j läuft von n bis 1.
8        erg = erg + j;           // akkumuliere das Ergebnis.
9        j = j-1;                 // verringere Laufzähler.
10     }
11
12     return erg;
13   }
14 }
```

Wie man an beiden Beispielen oben sieht, gibt es oft ein Feld, das zur Steuerung der Schleife benutzt wird. Dieses Feld verändert innerhalb des Schleifenrumpfes seinen Wert. Abhängig von diesem Wert wird die Schleifenbedingung beim nächsten Bedingungstest wieder wahr oder falsch.

Schleifen haben die unangenehme Eigenschaft, dass sie eventuell nie verlassen werden. Eine solche Schleife läßt sich minimal wie folgt schreiben:

Bottom.java

```
1 class Bottom {
2    static public void bottom(){
3      while (wahr()){};
4    }
5
6    static boolean wahr(){return true;}
7
```

```
8     public static void main(String [] _){bottom();}
9  }
```

Ein Aufruf der Methode `bottom` startet eine nicht endende Berechnung.

Häufige Programmierfehler sind inkorrekte Schleifenbedingungen oder falsch kontrollierte Schleifenvariablen. Das Programm terminiert dann mitunter nicht. Solche Fehler sind in komplexen Programmen oft schwer zu finden.

Nachgeprüfte Schleifen In der zweiten Variante der `while`-Schleife steht die Schleifenbedingung syntaktisch nach dem Schleifenrumpf:

`do` {*body*} `while` (*pred*)

Bei der Abarbeitung einer solchen Schleife wird entsprechend der Notation, die Bedingung erst nach der Ausführung des Schleifenrumpfes geprüft. Am Ende wird also geprüft, ob die Schleife ein weiteres Mal zu durchlaufen ist. Das impliziert insbesondere, dass der Rumpf mindestens einmal durchlaufen wird.

Die erste Schleife, die wir für die vorgeprüfte Schleife geschrieben haben, hat folgende nachgeprüfte Variante:

DoTest.java

```
1  class DoTest {
2    public static void main(String [] args){
3      int i = 0;
4      do {
5        System.out.println(i);
6        i = i+1;
7      } while (i < 10);
8    }
9  }
```

Man kann sich leicht davon vergewissern, dass die nachgeprüfte Schleife mindestens einmal durchlaufen[7] wird:

VorUndNach.java

```
1  class VorUndNach {
2
3    public static void main(String [] args){
4
5        while (falsch())
6           {System.out.println("vorgeprüfte Schleife");};
7
8        do {System.out.println("nachgeprüfte Schleife");}
9        while (false);
10   }
11
12   public static boolean falsch(){return false;}
13 }
```

[7] Der Javaübersetzer macht kleine Prüfungen auf konstanten Werten, ob Schleifen jeweils durchlaufen werden oder nicht terminieren. Deshalb brauchen wir die Hilfsmethode `falsch()`.

Schleifen mit: for

Das syntaktisch aufwendigste Schleifenkonstrukt in Java ist die *for*-Schleife.

Wer sich die obigen Schleifen anschaut, sieht, dass sie an drei verschiedenen Stellen im Programmtext Code haben, der kontrolliert, wie oft die Schleife zu durchlaufen ist. Oft legen wir ein spezielles Feld an, dessen Wert die Schleife kontrollieren soll. Dann gibt es im Schleifenrumpf einen Zuweisungsbefehl, der den Wert dieses Feldes verändert. Schließlich wird der Wert dieses Feldes in der Schleifenbedingung abgefragt.

Die Idee der *for*-Schleife ist, diesen Code, der kontrolliert, wie oft die Schleife durchlaufen werden soll, im Kopf der Schleife zu bündeln. Solche Daten sind oft Zähler vom Typ `int`, die bis zu einem bestimmten Wert herunter oder hoch gezählt werden. Später werden wir noch die Standardklasse `Iterator` kennenlernen, die benutzt wird, um durch Listenelemente durchzuiterieren.

Eine *for*-Schleife hat im Kopf

- eine Initialisierung der relevanten Schleifensteuerungsvariablen *(init)*,
- ein Prädikat als Schleifenbedingung *(pred)*
- und einen Befehl, der die Schleifensteuerungsvariable weiterschaltet *(step)*.

`for (`*init, pred, step*`){`*body*`}`

Entsprechend sieht unsere jeweilige erste Schleife (die Ausgabe der Zahlen von 0 bis 9) in der `for`-Schleifenversion wie folgt aus:

ForTest.java

```
1 class ForTest {
2   public static void main(String [] args){
3
4     for (int i=0; i<10; i=i+1){
5       System.out.println(i);
6     }
7
8   }
9 }
```

Die Reihenfolge, in der die verschiedenen Teile der `for`-Schleife durchlaufen werden, wirkt erst etwas verwirrend, ergibt sich aber natürlich aus der Herleitung der `for`-Schleife aus der vorgeprüften `while`-Schleife:

Als erstes wird genau einmal die Initialisierung der Schleifenvariablen ausgeführt. Anschließend wird die Bedingung geprüft. Abhängig davon wird der Schleifenrumpf ausgeführt. Als letztes wird die Weiterschaltung ausgeführt, bevor wieder die Bedingung geprüft wird.

Die nun schon hinlänglich bekannte Methode `summe` stellt sich in der Version mit der `for`-Schleife wie folgt dar:

Summe3.java

```
1 class Summe3 {
2   public static int summe(int n){
3
4     int erg = 0 ;                    // Feld für Ergebnis
5
6     for (int j = n;j>0;j=j-1){       // j läuft von n bis 1
7       erg = erg + j;                 // akkumuliere das Ergebnis
8     }
9
```

```
10      return erg;
11    }
12  }
```

Beim Vergleich mit der `while`-Version erkennt man, wie sich die Schleifensteuerung im Kopf der `for`-Schleife nun gebündelt an einer syntaktischen Stelle befindet.

Die drei Teile des Kopfes einer `for`-Schleife können auch leer sein. Dann wird in der Regel an einer anderen Stelle der Schleife entsprechender Code zu finden sein. So können wir die Summe auch mit Hilfe der `for`-Schleife so schreiben, dass die Schleifeninitialisierung und Weiterschaltung vor der Schleife bzw. im Rumpf durchgeführt wird:

Summe4.java

```
1  class Summe4 {
2    public static int summe(int n){
3
4      int erg = 0 ;              // Feld für Ergebnis.
5      int j   = n;              // Feld zur Schleifenkontrolle
6
7      for (;j>0;){               // j läuft von n bis 1
8        erg = erg + j;           // akkumuliere das Ergebnis.
9        j = j-1;                 // verringere Laufzähler
10     }
11
12     return erg;
13   }
14 }
```

Wie man jetzt sieht, ist die `while`-Schleife nur ein besonderer Fall der `for`-Schleife. Obiges Programm ist ein schlechter Programmierstil. Hier wird ohne Not die Schleifensteuerung mit der eigentlichen Anwendungslogik vermischt.

Vorzeitiges Beenden von Schleifen

Java bietet innerhalb des Rumpfes seiner Schleifen zwei Befehle an, die die eigentliche Steuerung der Schleife durchbrechen. Entgegen der im letzten Abschnitt vorgestellten Abarbeitung der Schleifenkonstrukte, führen diese Befehle zum plötzlichen Abbruch des aktuellen Schleifendurchlaufs.

Verlassen der Schleife Der Befehl, um eine Schleife komplett zu verlassen, heißt `break`. Der `break` führt zum sofortigen Abbruch der nächsten äußeren Schleife.

Der `break`-Befehl wird in der Regel mit einer `if`-Bedingung auftreten.

Mit diesem Befehl läßt sich die Schleifenbedingung auch im Rumpf der Schleife ausdrücken. Das Programm der Zahlen 0 bis 9 läßt sich entsprechend unschön auch mit Hilfe des `break`-Befehls wie folgt schreiben.

BreakTest.java

```
1  class BreakTest {
2    public static void main(String [] args){
3      int i = 0;
4      while (true){
5        if (i>9) {break;};
6        i = i+1;
7        System.out.println(i);
8      }
```

```
9     }
10 }
```

Gleichfalls läßt sich der `break`-Befehl in der `for`-Schleife anwenden. Dann wird der Kopf der `for`-Schleife vollkommen leer:

ForBreak.java

```
1  class ForBreak {
2    public static void main(String [] args){
3      int i = 0;
4      for (;;){
5          if (i>9) break;
6          System.out.println(i);
7          i=i+1;
8      }
9    }
10 }
```

In der Praxis wird der `break`-Befehl gerne für besondere Situationen inmitten einer längeren Schleife benutzt, z.B. für externe Signale.

Verlassen des Schleifenrumpfes Die zweite Möglichkeit, den Schleifendurchlauf zu unterbrechen, ist der Befehl `continue`. Diese Anweisung bricht nicht die Schleife komplett ab, sondern nur den aktuellen Durchlauf. Es wird zum nächsten Durchlauf gesprungen.

Folgendes kleines Programm druckt mit Hilfe des `continue`-Befehls die Zahlen aus, die durch 17 oder 19 teilbar sind:

ContTest.java

```
1  class ContTest{
2    public static void main(String [] args){
3       for (int i=1; i<1000;i=i+1){
4          if (!(i % 17 == 0 || i % 19 == 0) )
5     //wenn nicht die Zahl durch 17 oder 19 ohne Rest teilbar ist
6             continue;
7          System.out.println(i);
8       }
9    }
10 }
```

Wie man an der Ausgabe dieses Programms sieht, wird mit dem Befehl `continue` der Schleifenrumpf verlassen und die Schleife im Kopf weiter abgearbeitet. Für die `for`-Schleife heißt das insbesondere, dass die Schleifenweiterschaltung der nächste Ausführungsschritt ist.

Aufgabe 24 Schreiben Sie jetzt die Methode zur Berechnung der Fakultät, indem Sie eine Iteration und nicht eine Rekursion benutzen.

Aufgabe 25 Schreiben Sie eine Methode
`static String darstellungZurBasis(int x,int b)`, die als Parameter eine Zahl x und eine zweite Zahl b erhält. Sie dürfen annehmen, dass $x > 0$ und $1 < b < 11$. Das Ergebnis soll eine Zeichenkette vom Typ `String` sein, in der die Zahl x zur Basis b dargestellt ist. Testen Sie ihre Methode mit unterschiedlichen Basen.

Hinweis: Der zweistellige Operator `%` berechnet den ganzzahligen Rest einer Division. Bei einem geschickten Umgang mit den Operatoren `%, /` und `+` und einer `while`-Schleife kommen Sie mit sechs Zeilen im Rumpf der Methode aus.

Aufgabe 26 Schreiben Sie eine Methode `static int readIntBase10(String str)`. Diese Methode soll einen String, der nur aus Ziffern besteht, in die von ihm repräsentierte Zahl umwandeln. Benutzen sie hierzu die Methode `charAt` der `String`-Klasse, die es erlaubt, einzelne Buchstaben einer Zeichenkette zu selektieren.

3.3.3 die switch Anweisung

Aus C erbt Java eine sehr spezielle zusammengesetzte Anweisung, die **switch**-Anweisung. Es ist eine Anweisung für eine Fallunterscheidung mit mehreren Fällen, die `switch`-Anweisung. Die Idee dieser Anweisung ist, eine Kette von mehreren `if-then`-Anweisungen zu vermeiden. Leider ist die `switch`-Anweisung in seiner Anwendungsbereite recht begrenzt und in Form und Semantik ziemlich veraltet.

Schematisch hat die `switch`-Anweisung die folgende Form:

```
switch (expr){
  case const: stats
    ...
  case const: stats
  default: stats
  case const: stats
    ...
  case const: stats
}
```

Dem Schlüsselwort `switch` folgt ein Ausdruck, nach dessen Wert eine Fallunterscheidung getroffen werden soll. In geshweiften Klammern folgen die verschiedenen Fälle. Ein Fall beginnt mit dem Schüsselwort `case` gefolgt von einer Konstante. Diese Konstante ist von einem ganzzahligen Typ und darf kein Ausdruck sein, der erst während der Laufzeit berechnet wird. Es muss hier eine Zahl stehen. Die Konstante muss während der Übersetzungszeit des Programms feststehen. Der Konstante folgt ein Doppelpunkt, dem dann die Anweisungen für diesen Fall folgen.

Ein besonderer Fall ist der `default`-Fall. Dieses ist der Standardfall. Er wird immer ausgeführt, egal was für einen Wert der Ausdruck nach dem die `switch`-Anweisung unterscheidet hat.

Beispiel 3.3.1 *Ein kleines Beispiel soll die operationale Semantik dieser Anweisung verdeutlichen.*

Switch.java

```
1  class Switch {
2
3    public static void main (String [] args){
4      switch (4*new Integer(args[0]).intValue()){
5        case 42 : System.out.println(42);
6        case 52 : System.out.println(52);
7        case 32 : System.out.println(32);
8        case 22 : System.out.println(22);
9        case 12 : System.out.println(12);
10       default : System.out.println("default");
11     }
12   }
13 }
```

Starten wir das Programm mit dem Wert `13`*, so dass der Ausdruck, nach dem wir die Fallunterscheidung durchführen zu* `52` *auswertet, so bekommen wir folgende Ausgabe:*

```
sep@swe10:~/fh/internal/beispiele> java Switch 13
52
32
22
12
default
sep@swe10:~/fh/internal/beispiele>
```

Wie man sieht, springt die `switch`*-Anweisung zum Fall für den Wert* `52`*, führt aber nicht nur dessen Anweisungen aus, sondern alle folgenden Anweisungen.*

Das oben beobachtete Verhalten ist verwirrend. Zumeist will man in einer Fallunterscheidung, dass nur die entsprechenden Anweisung für den vorliegenden Fall ausgeführt werden und nicht auch für alle folgenden Fälle. Um dieses zu erreichen, gibt es die `break`-Anweisung, wie wir sie auch schon von den Schleifenanweisungen kennen. Endet man jeden Fall mit der `break`-Anweisung, dann erhälz man das meistens erwünschte Verhalten.

Beispiel 3.3.2 *Das obige Beispiel läßt sich durch Hinzufügen der* `break`*-Anweisung so ändern, dass immer nur ein Fall ausgeführt wird.*

```
1  class Switch2 {
2
3    public static void main (String [] args){
4      switch (4*new Integer(args[0]).intValue()){
5        case 42 : System.out.println(42);break;
6        case 52 : System.out.println(52);break;
7        case 32 : System.out.println(32);break;
8        case 22 : System.out.println(22);break;
9        case 12 : System.out.println(12);break;
10       default : System.out.println("default");
11     }
12   }
13 }
```

An der Ausgabe sehen wir, dass zu einem Fall gesprungen wird und am Ende dieses Falls die Anweisung verlassen wird.

```
sep@swe10:~/fh/internal/beispiele> java Switch2 13
52
sep@swe10:~/fh/internal/beispiele>
```

3.3.4 Rekursion und Iteration

Wir kennen zwei Möglichkeiten, um einen Programmteil wiederholt auszuführen: Iteration und Rekursion. Während die Rekursion kein zusätzliches syntaktisches Konstrukt benötigt, sondern lediglich auf den Aufruf einer Methode in ihrem eigenen Rumpf beruht, benötigte die Iteration spezielle syntaktische Konstrukte, die wir lernen mussten.

Javas virtuelle Maschine ist nicht darauf ausgerichtet, Programme mit hoher Rekursionstiefe auszuführen. Für jeden Methodenaufruf fordert Java intern einen bestimmten Speicherbereich an, der erst wieder freigegeben wird, wenn die Methode vollständig beendet wurde. Dieser Speicherbereich wird als der *stack* bezeichnet. Er kann relativ schnell ausgehen. Javas Maschine hat keinen Mechanismus, um zu erkennen, wann dieser Speicher für den Methodenaufruf eingespart werden kann.

Folgendes Programm illustriert, wie für eine Iteration über viele Schleifendurchläufe gerechnet werden kann, die Rekursion hingegen zu einen Programmabbruch führt, weil nicht genug Speicherplatz auf dem *stack* vorhanden ist.

StackTest.java

```
1  class StackTest {
2    public static void main(String [] args){
3      System.out.println(count1(0));
4      System.out.println(count2(0));
5    }
6
7    public static int count1(int k){
8      int j = 2;
9      for (int i= 0;i<1000000000;i=i+1){
10       j=j+1;
11     }
12     return j;
13   }
14
15   public static int count2(int i){
16       if (i<1000000000) return count2(i+1) +1; else return 2;
17   }
18
19 }
20
```

Aufgabe 27 Im Bestseller *Sakrileg (der Da Vinci Code)* spielen die Fibonaccizahlen eine Rolle. Für eine natürliche Zahl n ist ihre Fibonaccizahl definiert durch:

$$f(n) = \begin{cases} n & \text{für } n \leq 1 \\ f(n-1) + f(n-2) & \text{für } n > 1 \end{cases}$$

Programmieren Sie eine Funktion `static int fib(int n)`, die für eine Zahl, die entsprechende Fibonaccizahl zurückgibt.

Geben Sie in der Hauptfunktion die ersten 20 Fibonaccizahlen aus.

Durchlaufen Sie das Programm auch einmal schrittweise mit dem Debugger in Eclipse.

Aufgabe 28 In dieser Aufgabe sollen Sie die Klasse `Datum` des ersten Übungsblattes weiterentwickeln und für diese die folgenden nützlichen Methoden schreiben:

a) Schreiben Sie in der Klasse `Datum` eine Methode
`boolean isEarlier(Datum that)`
die genau dann wahr ergibt, wenn das `this`.Datumsobjekt früher liegt, als das Datum übergebene Datumsobjekt.

b) Schreiben Sie eine Methode
`boolean schaltjahr()`,
die angibt, ob es sich bei dem Jahr des Datums um ein Schaltjahr handelt oder nicht.

c) Schreiben Sie eine Methode
`void toNextDay()`
die das Datumsobjekt um einen Tag weiterschaltet.

d) Schreiben Sie eine Methode `Wochentag wasFuerEinTag()`, die angibt, um was für einen Wochentag es sich bei dem betreffenden Datum handelt.
Eine algorithmische Beschreibung hierzu finden sie auf:

de.wikipedia.org/wiki/Wochentagsberechnung.

Der Ergebnistyp `Wochentag` sei ein Objekt der folgenden Aufzählungsklasse:

Wochentag.java
```
1 public enum Wochentag {
2   montag,dienstag,mittwoch,donnerstag
3  ,freitag,sonnabend,sonntag;
4 }
```

Aufgabe 29 In dieser Aufgabe sollen Sie wie auf dem dritten Übungsblatt für das kleine Dialog-Gui eine Anwendungslogik schreiben. Die Eingabe soll dabei auf Zahlen beschränkt sein. Hierzu sei folgende Unterklasse der Klasse `ButtonLogic` gegeben.

ButtonLogicInt.java
```
1 class ButtonLogicInt  extends ButtonLogic {
2   String getDescription(){
3     return "Zahl Umwandeln";
4   }
5   String eval(String x){return eval(new Integer(x.trim()));}
6
7   String eval(int x){return x+"";}
8 }
```

Schreiben Sie jetzt eine Anwendung, bei der die Zahlen, die im Eingabefeld eingegeben wird als Dualzahl im Ausgabefeld angegeben wird. Schreiben Sie hierzu eine Unterklasse von `ButtonLogicInt`.

3.4 Ausdrücke und Befehle

Wir kennen mitlerweile eine große Anzahl mächtiger Konstrukte Javas. Dem aufmerksamen Leser wird aufgefallen sein, dass sich diese Konstrukte in zwei große Gruppen einteilen lassen: Ausdrücke und Befehle[8] (englisch: *expressions* und *statements*).

Ausdrücke berechnen direkt einen Objekt oder ein Datum eines primitiven Typs. Ausdrücke haben also immer einen Wert. Befehle hingegen sind Konstrukte, die Felder verändern oder Ausgaben auf dem Bildschirm erzeugen.

Ausdrücke	**Befehle**
Literale: `1, "fgj"`	Zuweisung: `x=1;`
Operatorausdrücke: `1*42`	Felddeklaration: `int i;`
Feldzugriffe: `obj.myField`	zusammengesetzte Befehle (if, while, for)
Methodenaufrufe mit Methodenergebnis: `obj.toString()`	Methodenaufrufe ohne Methodenergebnis: `System.out.println("hallo")`
Erzeugung neuer Objekte `new MyClass(56)`	Ablaufsteuerungsbefehle wie `break, continue, return`

Ausdrücke können sich aus Unterausdrücken zusammensetzen. So hat ein binärer Operatorausdruck zwei Unterausdrücke als Operanden. Hingegen kann ein Ausdruck niemals einen Befehl enthalten.

Beispiele für Befehle, die Unterbefehle und Unterausdrücke enthalten, haben wir bereits zu genüge gesehen. An bestimmten Stellen von zusammengesetzten Befehle müssen Ausdrücke

[8] In der Literatur findet man mitunter auch den Ausdruck *Anweisung* für das, was wir als Befehl bezeichnen. *Befehl* bezeichnet dann den Oberbegriff für Anweisungen und Ausdrücke.

eines bestimmten Typs stehen: so muss z.B. die Schleifenbedingung ein Ausdruck des Typs `boolean` sein.

3.4.1 Bedingungen als Befehl oder als Ausdruck

Die Bedingung kennen wir bisher nur als Befehl in Form des `if`-Befehls. Java kennt auch einen Ausdruck, der eine Unterscheidung auf Grund einer Bedingung macht. Dieser Ausdruck hat drei Teile: die bool'sche Bedingung und jeweils die positive und negative Alternative. Syntaktisch wird die Bedingung durch ein Fragezeichen von den Alternativen und die Alternativen werden mit einem Doppelpunkt voneinander getrennt:

$$pred ? alt1 : alt2$$

Im Gegensatz zum `if`-Befehl, in dem die `else`-Klausel fehlen kann, müssen im Bedingungsausdruck stets beide Alternativen vorhanden sein (weil ja ein Ausdruck per Definition einen Ergebniswert braucht). Die beiden Alternativen brauchen den gleichen Typ. Dieser Typ ist der Typ des Gesamtausdrucks.

Folgender Code:

CondExpr.java

```
1 class CondExpr {
2   public static void main(String [] _){
3     System.out.println(true?1:2);
4     System.out.println(false?3:4);
5   }
6 }
```

druckt erst die Zahl 1 und dann die Zahl 4.

```
sep@linux:~/fh/prog1/examples/classes> java CondExpr
1
4
sep@linux:~/fh/prog1/examples/classes>
```

3.4.2 Auswertungsreihenfolge und Seiteneffekte von Ausdrücken

Wir haben uns bisher wenig Gedanken darüber gemacht, in welcher Reihenfolge Unterausdrücke von der Javamaschine ausgewertet werden. Für Befehle war die Reihenfolge, in der sie von Java abgearbeitet werden, intuitiv sehr naheliegend. Eine Folge von Befehlen wird in der Reihenfolge ihres textuellen Auftretens abgearbeitet, d.h. von links nach rechts und von oben nach unten. Zusammengesetze Befehle wie Schleifen und Bedingungen geben darüberhinaus einen expliziten Programmablauf vor.

Für Ausdrücke ist eine solche explizite Reihenfolge nicht unbedingt ersichtlich. Primär interessiert das Ergebnis eines Ausdrucks, z.B. für den Ausdruck `(23-1)*(4-2)` interessiert nur das Ergebnis `42`. Es ist im Prinzip egal, ob erst `23-1` oder erst `4-2` gerechnet wird. Allerdings können Ausdrücke in Java nicht nur einen Wert berechnen, sondern gleichzeitig auch noch zusätzliche Befehle sozusagen unter der Hand ausführen. In diesem Fall sprechen wir von Seiteneffekten.

Reihenfolge der Operanden

Wir können Seiteneffekte, die eine Ausgabe auf den Bildschirm drucken, dazu nutzen, zu testen, in welcher Reihenfolge Unterausdrücke ausgewertet werden. Hierzu schreiben wir eine Subtraktionsmethode mit einem derartigen Seiteneffekt:

Minus.java
```
1  class Minus {
2
3    static public int sub(int x,int y){
4      int erg = x-y;
5      System.out.println("eine Subtraktion mit Ergebnis: "
6                         +erg+" wurde durchgeführt.");
7      return erg;
8    }
9  }
```

Obige Methode `sub` berechnet nicht nur die Differenz zweier Zahlen und gibt diese als Ergebnis zurück, sondern zusätzlich druckt sie das Ergebnis auch noch auf dem Bildschirm. Dieses Drucken ist bei einem Audruck, der einen Aufruf der Methode `sub` enthält, ein Seiteneffekt. Mit Hilfe solcher Seiteneffekte, die eine Ausgabe erzeugen, können wir testen, wann ein Ausdruck ausgewertet wird:

Operatorauswertung.java
```
1  class Operatorauswertung {
2
3     public static void main(String [] args){
4       System.out.println(Minus.sub(23,1)*Minus.sub(4,2));
5     }
6  }
```

Anhand dieses Testprogramms kann man erkennen, dass Java die Operanden eines Operatorausdrucks von links nach rechts auswertet:

```
sep@swe10:~/fh/prog1/beispiele> java Operatorauswertung
eine Subtraktion mit Ergebnis: 22 wurde durchgeführt.
eine Subtraktion mit Ergebnis: 2 wurde durchgeführt.
44
sep@swe10:~/fh/prog1/beispiele>
```

Auswertung der Methodenargumente

Ebenso wie für die Operanden eines Operatorausdrucks müssen wir für die Argumente eines Methodenaufrufs untersuchen, ob, wann und in welcher Reihenfolge diese ausgewertet werden. Hierzu schreiben wir eine Methode, die eines ihrer Argumente ignoriert und das zweite als Ergebnis zurückgibt:

Reihenfolge.java
```
1  public class Reihenfolge{
2
3      /**
4         Returns second argument.
5         @param x ignored  argument.
6         @param y returned argument.
7         @return projection on second argument.
8       **/
9    public static int snd(int x,int y){return y;}
10
```

Desweiteren eine Methode, die einen Seiteneffekt hat und einen konstanten Wert zurückgibt:

Reihenfolge.java
```
11     /**
12        Returns the constants 1 and prints its argument.
13        @param  str The String that is printed as side effect.
```

```
14        @return constantly 1.
15      **/
16    public static int eins(String str){
17      System.out.println(str); return 1;
18    }
```

Damit läßt sich überprüfen, dass die Methode `snd` tatsächlich beide Argumente auswertet, und zwar auch von links nach rechts. Obwohl ja streng genommen die Auswertung des ersten Arguments zum Berechnen des Ergebnisses nicht notwendig wäre:

Reihenfolge.java

```
19  public static void main(String [] args){
20    //<bluev>test, in which order arguments are evaluated.</bluev>
21    snd(eins("a"),eins("b"));
22  }}
```

Terminierung

Spannender noch wird die Frage, ob und wann die Argumente ausgewertet werden , wenn bestimmte Argumente nicht terminieren. Auch dieses läßt sich experimentell untersuchen. Wir schreiben eine Methode, die nie terminiert:

TerminationTest.java

```
1  class TerminationTest {
2      /**
3        Nonterminating method.
4        @return never returns anything.
5       **/
6    public static int bottom( ){while(wahr()){};return 1; }
7
8      /**
9         Returns the constants 1 and prints its argument.
10        @param  str The String that is printed as side effect.
11        @return constantly 1.
12     **/
13   public static int eins(String str){
14     System.out.println(str); return 1;
15   }
16
17     /**
18        Constantly true method.
19        @return constantly the value true.
20     **/
21   public static boolean wahr( ){return true; }
```

Den Bedingungsausdruck können wir für einen festen Typen der beiden Alternativen als Methode schreiben:

TerminationTest.java

```
22     /**
23        Method mimicking conditional expression.
24        @param pre The boolean condition.
25        @param a1 The positive alternative.
26        @param a2 The negative alternative.
27        @return in case of pre the second otherwise third argument
28      **/
29   public static int wenn(boolean pre,int a1,int a2 ){
30     return pre?a1:a2; }
```

Ein Test kann uns leicht davon überzeugen, dass der Methodenaufruf mit dem entsprechenden Bedingungsausdruck im Terminierungsverhalten nicht äquivalent ist.

TerminationTest.java

```
31   public static void main(String [] args){
32
33     //test, whether both alternatives of conditional
34     //expression are evaluated or just one.
35
36     //this will terminate:
37     System.out.println(      false?bottom():eins("b")   );
38
39     //this won't terminate:
40     System.out.println(wenn( false,bottom(),eins("b") )  );
41   }
42 }
```

Man sieht also, dass eine äquivalente Methode zum Bedingungsausdruck in Java nicht geschrieben werden kann.

So natürlich uns die in diesem Abschnitt festgestellten Auswertungsreihenfolgen von Java erscheinen, so sind sie doch nicht selbstverständlich. Es gibt funktionale Programmiersprachen, die nicht wie Java vor einem Methodenaufruf erst alle Argumente des Aufrufs auswerten. Diese Auswertungsstrategie wird dann als *nicht strikt* bezeichnet, während man die Auswertungsstrategie von Java als *strikt* bezeichnet.

3.4.3 Auswertung der bool'schen Operatoren

Wir haben die beiden zweistelligen Infixoperatoren für das logische *und* `&&` und das logische *oder* `||` kennengelernt. Aus der formalen Logik ist bekannt:

$true \vee A = true$ und $false \wedge A = false$

In bestimmten Fällen läßt sich das Ergebnis einer bool'schen Operation bereits bestimmen, ohne den zweiten Operanden anzuschauen. Die Frage ist, ob auch Java in diesen Fällen sich zunächst den linken Operanden anschaut und nur bei Bedarf noch den rechten Operanden betrachtet. Hierzu können wir ein Testprogramm schreiben. Wir bedienen uns dabei wieder einer Methode, die nicht nur einen bool'schen Wert als Ergebnis, sondern auch einen Seiteneffekt in Form einer Ausgabe auf den Bildschirm hat:

LazyBool.java

```
1  class LazyBool {
2
3    static boolean booleanExpr(){
4      System.out.println("in booleanExpr");
5      return true;
6    }
7
8    static public void main(String [] _){
9      System.out.println(true || booleanExpr());
10     System.out.println(false && booleanExpr());
11   }
12
13 }
```

An der Ausgabe dieses Programms läßt sich schließlich erkennen, ob Java bei der Auswertung auch noch den rechten Operanden betrachtet , obwohl durch den linken Operanden der Wert des Gesamtausdrucks bereits ermittelt werden kann:

```
sep@linux:~/fh/prog1/examples/classes> java LazyBool
true
false
sep@linux:~/fh/prog1/examples/classes>
```

Und tatsächlich bekommen wir keine Ausgabe aus der Methode `booleanExpr`. Java wertet also den linken Operanden in diesem Fall nicht mehr aus. Die beiden bool'schen Operatoren `&&` und `||` werden in Java nicht strikt ausgewertet.

strikte bool'sche Operatoren

Zu den beiden logischen binären Operatoren `&&` und `||` gibt es zwei Versionen, in denen das entsprechende Zeichen nicht doppelt steht: `&` und `|`. Sie stehen auch für das logische *und* bzw. *oder*. Die einfachen Versionen der bool'schen Operatoren haben eine andere Strategie, wie sie das Ergebnis berechnen. Sie werten zunächst beide Operanden aus, um dann das Ergebnis aus deren Werten zu berechnen.

Die Strategie, immer erst alle Operanden einer Operation (entsprechend die Parameter eines Methodenaufrufs) komplett anzuschauen und auszuwerten, nennt man *strikt*. Die gegenteilige Strategie, nur das Notwendigste von den Operanden auszuwerten, entsprechend nicht-strikt.[9]

Wenn wir jetzt im obigen Testprogramm statt der nicht-strikten Operatoren die beiden strikten Operatoren benutzen, beobachten wir den Seiteneffekt aus der Methode `booleanExpr`:

StrictBool.java

```
1  class StrictBool {
2
3    static boolean booleanExpr(){
4      System.out.println("in booleanExpr");
5      return true;
6    }
7
8    static public void main(String [] _){
9      System.out.println(true | booleanExpr());
10     System.out.println(false & booleanExpr());
11   }
12
13 }
```

Und tatsächlich bekommen wir jetzt zusätzliche Ausgaben auf dem Bildschirm:

```
sep@linux:~/fh/prog1/examples/classes> java StrictBool
in booleanExpr
true
in booleanExpr
false
sep@linux:~/fh/prog1/examples/classes>
```

3.5 Reihungen

Java kennt, wie fast alle Programmiersprachen, ein weiteres Konzept von Sammlungen: Reihungen (eng. arrays).[10] Reihungen stellen im Gegensatz zu Listen oder Mengen eine Menge

[9] Kommt zur nicht-strikten Strategie noch eine Strategie hinzu, die verhindert, dass Ausdrücke doppelt ausgewertet werden, so spricht man von einer faulen *(lazy)* Auswertung.

[10] In der deutschen Literatur findet man oft den Ausdruck *Datenfeld* für Reihungen. Wir haben uns gegen diesen Ausdruck entschieden, um nicht mit Feldern einer Klasse durcheinander zu kommen.

von Daten gleichen Typs mit fester Anzahl dar. Jedes Element einer Reihung hat einen festen Index, über den es direkt angesprochen werden kann.

3.5.1 Deklaration von Reihungen

Eine Reihung hat im Gegensatz zu den Sammlungsklassen[11] einen festen Elementtyp.

Ein Reihungstyps wird deklariert, indem dem Elementtyp ein eckiges Klammernpaar nachgestellt wird, z.B. ist `String []` eine Reihung von Stringelementen. Die Elemente einer Reihung können sowohl von einem Objekttyp als auch von einem primitiven Typ sein, also gibt es auch den Typ `int []` oder z.B. `boolean []`.

Reihungen sind Objekte. Sie sind zuweisungskompatibel für Objektfelder, es lassen sich Typzusicherungen auf Reihungen durchführen und Reihungen haben ein Feld `length`vom Typ `int`, das die feste Länge einer Reihung angibt.

ObjectArray.java

```
1 class ObjectArray {
2   public static void main(String [] args){
3     Object as = args;
4     System.out.println(((String [])as).length);
5   }
6 }
```

3.5.2 Erzeugen von Reihungen

Es gibt zwei Verfahren, um Reihungen zu erzeugen: indem die Elemente der Reihung aufgezählt werden oder indem die Länge der Reihung angegeben wird. Eine Mischform, in der sowohl Länge als auch die einzelnen Elemente angegeben werden, gibt es nicht.

Aufzählung der Reihung

Die einfachste Art, um eine Reihung zu erzeugen, ist, die Elemente aufzuzählen. Hierzu sind die Elemente in geschweiften Klammern mit Komma getrennt aufzuzählen:

FirstArray.java

```
1  class FirstArray {
2
3   static String [] komponisten
4    = {"carcassi","carulli","giuliani"
5      ,"molino","monzino","paganini","sor"};
6
7    public static void main(String [] args){
8      System.out.println(komponisten.length);
9      System.out.println(komponisten.toString());
10   }
11 }
```

Wie man beim Starten dieser kleinen Klasse erkennen kann, ist für Reihungen keine eigene Methode `toString` in Java implementiert worden.

[11] Dies wird sich mit den generischen Typen ab Java 1.5 ändern.

Uninitialisierte Reihungen

Eine weitere Methode zur Erzeugung von Reihungen ist, noch nicht die einzelnen Elemente der Reihung anzugeben, sondern nur die Anzahl der Elemente:

SecondArray.java

```
1 class SecondArray {
2
3   static int [] zahlenReihung = new int[10];
4
5   public static void main(String [] args){
6     System.out.println(zahlenReihung);
7   }
8 }
```

3.5.3 Zugriff auf Elemente

Die einzelnen Elemente einer Reihung können über einen Index angesprochen werden. Das erste Element einer Reihung hat den Index `0`, das letzte Element den Index `length-1`.

Als Syntax benutzt Java die auch aus anderen Programmiersprachen bekannte Schreibweise mit eckigen Klammern:

```
1 String [] stra = {"hallo","welt"};
2 String str = stra[1];
```

Typischer Weise wird mit einer `for`-Schleife über den Index einer Reihung iteriert. So läßt sich z.B. eine Methode, die eine Stringdarstellung für Reihungen erzeugt, wie folgt schreiben:

ArrayToString.java

```
1 public class ArrayToString{
2   static public String arrayToString(String [] obja){
3     StringBuffer result = new StringBuffer("{");
4     for (int i=0;i<obja.length;i=i+1){
5       if (i>0) result.append(",");
6       result.append(obja[i].toString());
7     }
8     result.append("}");
9     return result.toString();
10  }
11 }
```

3.5.4 Ändern von Elementen

Eine Reihung kann als ein Komplex von vielen einzelnen Feldern gesehen werden. Die Felder haben keine eigenen Namen, sondern werden über den Namen der Reihung zusammen mit ihrem Index angesprochen. Mit diesem Bild ergibt sich automatisch, wie nun einzelnen Reihungselementen neue Objekte zugewiesen werden können:

```
1 String [] stra = {"hello","world"};
2 stra[0]="hallo";
3 stra[1]="welt";
```

3.5.5 Das Kovarianzproblem

Beim Entwurf von Java wurde in Bezug auf Reihungen eine Entwurfsentscheidung getroffen, die zu Typfehlern während der Laufzeit führen können.

Seien gegeben zwei Typen `A` und `B`, so dass gilt: `B extends A`. Dann können Objekte des Typs `B []` Feldern des Typs `A []` zugewiesen werden.

Eine solche Zuweisung ist gefährlich.[12] Diese Zuweisung führt dazu, dass Information verloren geht. Hierzu betrachte man folgendes kleine Beispiel:

Kovarianz.java

```
1  class Kovarianz {
2
3    static String [] stra = {"hallo","welt"};
4
5    //gefährliche Zuweisung
6    static Object [] obja = stra;
7
8    public static void main(String [] args) {
9      //dieses ändert auch stra. Laufzeitfehler!
10     obja[0]=new Integer(42);
11
12     String str = stra[0];
13     System.out.println(str);
14   }
15 }
```

Die Zuweisung `obja = stra;` führt dazu, dass `stra` und `obja` auf ein und dieselbe Reihung verweisen. Das Feld `stra` erwartet, dass nur `String`-Objekte in der Reihung sind, das Feld `obja` ist weniger streng und läßt Objekte beliebigen Typs als Elemente der Reihung zu. Aus diesem Grund läßt der statische Typcheck die Zuweisung `obja[0]=new Integer(42);` zu. In der Reihung, die über das Feld `obja` zugreift, wird ein Objekt des Typs `Integer` gespeichert. Der statische Typcheck läßt dieses zu, weil beliebige Elemente in dieser Reihung stehen dürfen. Erst bei der Ausführung der Klasse kommt es bei dieser Zuweisung zu einem Laufzeitfehler:

```
sep@swe10:~/fh/beispiele> java Kovarianz
Exception in thread "main" java.lang.ArrayStoreException
        at Kovarianz.main(Kovarianz.java:10)
```

Die Zuweisung hat dazu geführt, dass in der Reihung des Feldes `stra` ein Element eingefügt wird, dass kein `String`-Objekt ist. Dieses führt zu obiger `ArrayStoreException`. Ansonsten würde der Reihungszugriff `stra[0];` in der nächsten Zeile kein `String`-Objekt ergeben.

Dem Kovarianzproblem kann man entgehen, wenn man solche Zuweisungen wie `obja = stra;` nicht durchführt, also nur Reihungen einander zuweist, die exakt ein und denselben Elementtyp haben. In der virtuellen Maschine wird es trotzdem bei jeder Zuweisung auf ein Reihungselement zu einen dynamischen Typcheck kommen, der zu Lasten der Effizienz geht.

Die Entwurfsentscheidung, die zum Kovarianzproblem führt, ist nicht vollkommen unmotiviert. Wäre die entsprechende Zuweisung, die zu diesem Problem führt, nicht erlaubt, so ließe sich nicht die Methode `ArrayToString.arrayToString` so schreiben, dass sie für Reihungen mit beliebigen Elementtypen[13] angewendet werden könnte.

[12] Und daher in generischen Klassen in Java 1.5 nicht zulässig, sprich `List<Object>` darf z.B. kein Objekt des Typs `List<String>` zugewiesen werden.

[13] Primitive Typen sind dabei ausgenommen.

3.5.6 Weitere Methoden für Reihungen

Im Paket `java.util` steht eine Klasse `Arrays` zur Verfügung, die eine Vielzahl statischer Methoden zur Manipulation von Reihungen beinhaltet.

In der Klasse `java.lang.System` gibt es eine Methode

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

zum Kopieren bestimmter Reihungsabschnitte in eine zweite Reihung.

3.5.7 Reihungen von Reihungen

Da Reihungen ganz normale Objekttypen sind, lassen sich, ebenso wie es Listen von Listen geben kann, auch Reihungen von Reihungen erzeugen. Es entstehen mehrdimensionale Räume. Hierzu ist nichts neues zu lernen, sondern lediglich die bisher gelernte Technik von Reihungen anwendbar. Angenommen, es gibt eine Klasse `Schachfigur`, die die 14 verschiedenen Schachfiguren modelliert, dann läßt sich ein Schachbrett wie folgt über eine zweidimensionale Reihenstruktur darstellen:

SchachFeld.java

```
class SchachFeld {
  Schachfigur [][] spielFeld
    =  {new Schachfigur[8]
       ,new Schachfigur[8]
       ,new Schachfigur[8]
       ,new Schachfigur[8]
       ,new Schachfigur[8]
       ,new Schachfigur[8]
       ,new Schachfigur[8]
       ,new Schachfigur[8]
       };

  public void zug(int vonX,int vonY,int nachX,int nachY){
    spielFeld[nachX][nachY] = spielFeld[vonX][vonY];
    spielFeld[vonX][vonY] = null;
  }
}
class Schachfigur {}
```

3.5.8 Iteration durch Reihungen

Zum Iterieren über einen Zahlenbereich können wir eine entsprechende Klasse vorsehen.

FromTo.java

```
package name.panitz.crempel.util;

import java.util.Iterator;

public class FromTo implements Iterable<Integer>,Iterator<Integer>{
  private final int to;
  private int from;
  public FromTo(int f,int t){to=t;from=f;}
```

```
9   public boolean hasNext(){return from<=to;}
10  public Integer next(){int result = from;from=from+1;return result;}
11  public Iterator<Integer> iterator(){return this;}
12  public void remove(){new UnsupportedOperationException();}
13 }
```

3.5.9 Blubbersortierung

Der Name *bubble sort* leitet sich davon ab, dass Elemente wie die Luftblasen in einem Mineralwasserglass innerhalb der Liste aufsteigen, wenn sie laut der Ordnung an ein späteres Ende gehören. Ein vielleicht phonetisch auch ähnlicher klingender deutscher Name wäre *Blubbersortierung*. Dieser Name ist jedoch nicht in der deutschen Terminologie etabliert und es wird in der Regel der englische Name genommen.

Die Idee des *bubble sort* ist, jeweils nur zwei benachbarte Elemente einer Liste zu betrachten und diese gegebenenfalls in ihrer Reihenfolge zu vertauschen. Eine Liste wird also von vorne bis hinten durchlaufen, immer zwei benachbarte Elemente betrachtet und diese, falls das vordere nicht kleiner ist als das hintere, getauscht. Wenn die Liste in dieser Weise einmal durchgegangen wurde, ist sie entweder fertig sortiert oder muss in gleicher Weise nocheinmal durchgegangen werden, solange, bis keine Vertauschungen mehr vorzunehmen sind.

Die Liste `("z","b","c","a")` wird durch den *bubble sort*-Algorithmus in folgender Weise sortiert:

1. Bubble-Durchlauf
`("z","b","c","a")`
`("b","z","c","a")`
`("b","c","z","a")`
`("b","c","a","z")`

Das Element `"z"` ist in diesem Durchlauf an das Ende der Liste geblubbert.

2. Bubble-Durchlauf
`("b","c","a","z")`
`("b","a","c","z")`

In diesem Durchlauf ist das Element `"c"` um einen Platz nach hinten geblubbert.

3. Bubble-Durchlauf
`("b","a","c","z")`
`("a","b","c","z")`

Im letzten Schritt ist das Element `"b"` auf seine endgültige Stelle geblubbert.

Blubbersortierung läßt sich relativ leicht für Reihungen implementieren. Eine Reihung ist dabei nur mehrfach zu durchlaufen und nebeneinanderstehende Elemente sind eventuell in der Reihenfolge zu vertauschen. Dieses wird so lange gemacht, bis keine Vertauschung mehr stattfindet.

BubbleSort.java

```
1  class BubbleSort {
2    static void bubbleSort(Comparable [] obja){
3      boolean toBubble = true;
4      while (toBubble){ toBubble = bubble(obja);}
5    }
6
7    static boolean bubble(Comparable [] obja){
8      boolean result = false;
9      for (int i=0;i<obja.length;i=i+1){
10       try {
```

```
11          if (obja[i].compareTo(obja[i+1])>0){
12            Comparable o = obja[i];
13            obja[i]=obja[i+1];
14            obja[i+1]=o;
15            result=true;
16          }
17        }catch (ArrayIndexOutOfBoundsException _){}
18      }
19      return result;
20    }
21
22    static public void main(String [] args){
23      String [] stra = {"a","zs","za","bb","aa","aa","y"};
24      System.out.println(ArrayToString.arrayToString(stra));
25      bubbleSort(stra);
26      System.out.println(ArrayToString.arrayToString(stra));
27    }
28
29  }
```

Die Umsetzung von *quicksort* auf Reihungen ist hingegen eine schwierige und selbst für erfahrene Programmierer komplizierte Aufgabe.

3.6 Aufzählungstypen

Häufig möchte man in einem Programm mit einer endlichen Menge von Werten rechnen. Java bot bis Version 1.4 kein ausgezeichnetes Konstrukt an, um dieses auszudrücken. Man war gezwungen in diesem Fall sich des Typs `int` zu bedienen und statische Konstanten dieses Typs zu deklarieren. Dieses sieht man auch häufig in Bibliotheken von Java umgesetzt. Etwas mächtiger und weniger primitiv ist, eine Klasse zu schreiben, in der es entsprechende Konstanten dieser Klasse gibt, die durchnummeriert sind. Dieses ist ein Programmiermuster, das Aufzählungsmuster.

Mit Java 1.5 ist ein expliziter Aufzählungstyp in Java integriert worden. Syntaktisch erscheint dieser wie eine Klasse, die statt des Schlüsselworts `class` das Schlüsselwort `enum` hat. Es folgt als erstes in diesen Aufzählungsklassen die Aufzählung der einzelnen Werte.

Beispiel 3.6.1 *Ein erster Aufzählungstyp für die Wochentage.*

Wochentage.java

```
1 package name.panitz.enums;
2 public enum Wochentage {
3   montag,dienstag,mittwoch,donnerstag
4  ,freitag,sonnabend,sonntag;
5 }
```

Auch dieses neue Konstrukt wird von Javaübersetzer in eine herkömmlige Javaklasse übersetzt. Wir können uns davon überzeugen, indem wir uns einmal den Inhalt der erzeugten Klassendatei mit `javap` *wieder anzeigen lassen:*

```
sep@linux:fh/> javap name.panitz.enums.Wochentage
Compiled from "Wochentage.java"
public class name.panitz.enums.Wochentage extends java.lang.Enum{
    public static final name.panitz.enums.Wochentage montag;
    public static final name.panitz.enums.Wochentage dienstag;
    public static final name.panitz.enums.Wochentage mittwoch;
```

```
    public static final name.panitz.enums.Wochentage donnerstag;
    public static final name.panitz.enums.Wochentage freitag;
    public static final name.panitz.enums.Wochentage sonnabend;
    public static final name.panitz.enums.Wochentage sonntag;
    public static final name.panitz.enums.Wochentage[] values();
    public static name.panitz.enums.Wochentage valueOf(java.lang.String);
    public name.panitz.enums.Wochentage(java.lang.String, int);
    public int compareTo(java.lang.Enum);
    public int compareTo(java.lang.Object);
    static {};
}
```

Eine der schönen Eigenschaften der Aufzählungstypen ist, dass sie in einer `switch`-Anweisung benutzt werden können.

Beispiel 3.6.2 *Wir fügen der Aufzählungsklasse eine Methode zu, um zu testen ob der Tag ein Werktag ist. Hierbei läßt sich eine* `switch`*-Anweisung benutzen.*

Tage.java

```
package name.panitz.enums;
public enum Tage {
  montag,dienstag,mittwoch,donnerstag
 ,freitag,sonnabend,sonntag;

  public boolean isWerktag(){
    switch (this){
      case sonntag    :
      case sonnabend  :return false;
      default         :return true;
    }
  }

  public static void main(String [] _){
    Tage tag = freitag;
    System.out.println(tag);
    System.out.println(tag.ordinal());
    System.out.println(tag.isWerktag());
    System.out.println(sonntag.isWerktag());
  }
}
```

Das Programm gibt die erwartete Ausgabe:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.Tage
freitag
4
true
false
sep@linux:~/fh/java1.5/examples>
```

Eine angenehme Eigenschaft der Aufzählungsklassen ist, dass sie in einer Reihung alle Werte der Aufzählung enthalten, so dass mit der neuen `for`-Schleife bequem über diese iteriert werden kann.

Beispiel 3.6.3 *Wir iterieren in diesem Beispiel einmal über alle Wochentage.*

IterTage.java

```
package name.panitz.enums;
public class IterTage {
  public static void main(String [] _){
```

```
4       for (Tage tag:Tage.values())
5         System.out.println(tag.ordinal()+": "+tag);
6     }
7   }
```

Die erwarttete Ausgabe ist:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.IterTage
0: montag
1: dienstag
2: mittwoch
3: donnerstag
4: freitag
5: sonnabend
6: sonntag
sep@linux:~/fh/java1.5/examples>
```

Schließlich kann man den einzelnen Konstanten einer Aufzählung noch Werte übergeben.

Beispiel 3.6.4 *Wir schreiben eine Aufzählung für die Euroscheine. Jeder Scheinkonstante wird noch eine ganze Zahl mit übergeben. Es muss hierfür ein allgemeiner Konstruktor geschrieben werden, der diesen Parameter übergeben bekommt.*

Euroschein.java

```
1  package name.panitz.enums;
2  public enum Euroschein {
3     fünf(5),zehn(10),zwanzig(20),fünfzig(50),hundert(100)
4    ,zweihundert(200),tausend(1000);
5    private int value;
6    Euroschein(int v){value=v;}
7    public int value(){return value();}
8
9    public static void main(String [] _){
10     for (Euroschein schein:Euroschein.values())
11       System.out.println
12         (schein.ordinal()+": "+schein+" -> "+schein.value);
13   }
14 }
```

Das Programm hat die folgende Ausgabe:

```
sep@linux:~/fh/java1.5/examples> java -classpath classes/ name.panitz.enums.Euroschein
0: fünf -> 5
1: zehn -> 10
2: zwanzig -> 20
3: fünfzig -> 50
4: hundert -> 100
5: zweihundert -> 200
6: tausend -> 1000
sep@linux:~/fh/java1.5/examples>
```

3.7 Klassen für logische Formeln

Java hat wie die meisten Programmiersprachen eine eingebauten Typ für Wahrheitswerte und kennt kennt ein paar logische Operatoren, wie die Konjunktion (das logische Und) die Disjunktion (das logische Oder) und die Negation (das logische Nicht). Die logischen Ausdrücke werden in Java als Ausdrücke ausgewertet zu einen der beiden Wahrheitswerte `true` oder `false`.

Im parallel gehaltenen Modul ≫Diskrete Strukturen≪ werden aussagenlogische Formeln nicht als Ausdrücke einer Programmiersprache betrachtet, sondern als eine Formelsprache. Über die Formeln werden dort Äquivalenzen gezeigt und Umformungen auf Formeln durchgeführt.

In diesem Abschnitt soll mit den bisherigen vorgestellten Javamitteln eine Klassenbibliothek entwickelt werden, die aussagenlogische Formeln darstellen können. So erhalten wir Objekte, die aussagenlogische Formeln repräsentieren. Diese Objekte können dann verarbeitet werden.

Beginnen wir mit einer Klasse, die allgemein für eine aussagenlogische Formel stehen soll.

Formel.java
```
1 public class Formel{
```

Tatsächlich ist etwas fraglich, was für Felder und Methoden diese Klasse bekommen soll. Es gibt ja sehr viel unterschiedliche Arten aussagenlogischer Formeln, je nachdem mit welchem logischen Junktor die Formel gebildet wurde. Diese unterschiedlichen Arten lassen sich sicher gut als Unterklassen der Klasse Formel darstellen. Wir können in der allgemeinen Klasse zunächst einmal sammeln, welche Methoden wir auf alle Formeln anwenden wollen.

Ziel unserer Übung soll sein, dass wir in der Lage sind schrittweise eine Formel in eine Normalform umzuwandeln. Diese Normalform soll die sogenannte konjunktive Normalform sein, auch Klauselform genannt. Diese zeichnet sich dadurch aus, dass nur noch die logischen Junktoren der Negation, Konjunktion und Disjunktion in der Formel auftauchen, Negationen nur direkt vor einer aussagenlogischen Variablen stehen und innerhalb einer Disjunktion (Oderverknüpfung) keine Konjunktion mehr auftritt. Um eine Formel in die Klauselform sind also schrittweise Umformungen vorzunehmen.

- Pfeilsymbole (Implikation und Äquivalenz) sind aus der Formel zu eliminieren.
- Die Negationen sind in die Formel hinein zu schieben.
- Eine Disjunktion innerhalb derer sich noch eine Konjunktion befindet ist auszumultiplizieren.

Damit muss es für diese Schritte für eine Formel die Möglichkeit geben, diese Umformungen durchzuführen. Wir sehen entsprechende Methoden vor.

Formel.java
```
2   Formel eliminierePfeil(){return this;}
3   Formel negatNachInnen(){return this;}
4   Formel undUeberOder(){return this;}
```

Die Rückgabe des `this`-Objekts ist in diesem Fall noch etwas unbeholfen. Die Formel soll umgeformt werden. Da wir nicht wissen, um was für eine Formelart es sich handelt, können wir auch nicht sagen, wie sie in den einzelnen Schritten umzuformen ist. So lange wir keine bessere Möglichkeit über abstrakte Klassen oder Schnittstellen kennen, behelfen wir uns in diesem Fall mit der Implementierung, die die Formel nicht verändert.

Die drei Umformungen hintereinander durchgeführt, sollen die Klauselform ergeben.

Formel.java
```
5   Formel klauselForm(){
6     return this.eliminierePfeil().negatNachInnen().undUeberOder();
7   }
```

Um unsere kleine aussagenlogische Bibliothek gut zu verwenden, sollen Formeln sich im Format des Textverarbeitungsprogramms LaTeX umwandeln lassen, d.h. in einen String, der von LaTeX in ein druckbares Format (heutzutage zumeist PDF) übersetzt wird.

Formel.java
```
8   String toLatex(){return "";}
```

Auch hier greifen wir vorerst zu dem Behelf, dass im allgemeinen der leere String als Ergebnis zurück gegeben wird.

Mit diesen Methoden lässt sich eine kleine Methode schreiben, die den Normalisierungsprozess einer Formel zur Klauselform komplett als LaTeX-Dokument darstellt:

Formel.java

```
9   String normalisationAsLatex() {
10    Formel o1 = this;
11    Formel o2 = o1.eliminierePfeil();
12    Formel o3 = o2.negatNachInnen();
13    Formel o4 = o3.undUeberOder();
14    return "\\section*{Umwandlung in Klauselform}"
15     +"originale Formel\\\\\n"
16     +"$"+o1.toLatex()+"$\n\n"
17     +"ohne Implikationen und Äquivalenzen\\\\\n"
18     +"$"+o2.toLatex()+"$\n\n"
19     +"Negationen nach innen\\\\\n"
20     +"$"+o3.toLatex()+"$\n\n"
21     +"Klauselform\\\\\n"
22     +"$"+o4.toLatex()+"$\\\\\n";
23  }
```

Mit Hilfe einer statischen Methode in der im nächsten Abschnitt angegebenen Klasse `LatexUtils`, kann der Normalisierungsprozess als ein LaTeX-Dokument gesetzt werden und in einem PDF-Anzeigeprogramm geöffnet werde.

Formel.java

```
24  void displayNormalisation(){
25    LatexUtils.displayLatex(normalisationAsLatex());
26  }
27 }
```

3.7.1 LaTeX Hilfsklasse

Die folgende Klasse sei hier unerklärt eingefügt. Sie bitten die in der Klasse Formel benutzte Funktionalität, dass ein LaTeX-Dokument gesetzt und angezeigt wird. Dabei wird davon ausgegangen, dass das Programm `pdflatex` zum setzen einer Dokuments in LaTeX installiert ist, sowie das Programm `evince` zum Betrachten eines PDF-Dokuments. Durch Ändern der Konstante `LATEX_VIEWER` kann auch ein anderes Programm als Betrachter von PDF-Dokumenten gewählt werden.

das Programm bedient sich der Möglichkeit in Java in einem Prozess andere Programme zu starten. Desweiteren benutzt das Programm die Standardklassen für Eingabe und Ausgabe in Java.

LatexUtils.java

```
1  import java.io.BufferedReader;
2
3  import java.io.FileOutputStream;
4  import java.io.InputStreamReader;
5  import java.io.OutputStreamWriter;
6  import java.io.Writer;
7  import java.io.File;
8
9  public class LatexUtils {
10   final static String LATEX_START
11     = "\\documentclass{article}\n"+
12       "\\usepackage[latin1]{inputenc}\n" +
```

```
13        "\\setlength\\parindent{0pt} \n" +
14        "\\begin{document}\n";
15
16    final static String LATEX_END = "\\end{document}";
17
18    final static String FILE_NAME = "output";
19    final static String LATEX_FILE = FILE_NAME+".tex";
20    final static String PDF_FILE = FILE_NAME+".pdf";
21
22    final static String LATEX_PROCESSOR = "pdflatex";
23    final static String LATEX_VIEWER = "evince";
24
25    static public void displayLatex(String latex){
26      try{
27        Writer w
28          = new OutputStreamWriter(new FileOutputStream(LATEX_FILE)
29                                   ,"iso-8859-1");
30        w.write(LATEX_START);
31        w.write(latex);
32        w.write(LATEX_END);
33        w.close();
34
35        String[] args
36         ={LATEX_PROCESSOR,"-interaction=nonstopmode",LATEX_FILE};
37        Process p = Runtime.getRuntime().exec(args,null,new File("."));
38        BufferedReader input
39          = new BufferedReader(new InputStreamReader(p.getInputStream()));
40        int  c = input.read();
41        while (c > -1) {
42           System.out.print((char)c);
43           c = input.read();
44        }
45        p.waitFor();
46
47        String[] args2 = {LATEX_VIEWER,PDF_FILE};
48        Process p2 =Runtime.getRuntime().exec(args2,null,new File("."));
49        BufferedReader input2
50         = new BufferedReader(new InputStreamReader(p2.getErrorStream()));
51        int  c2 = input2.read();
52        while (c2 > -1) {
53          System.out.print((char)c2);
54          c2 = input2.read();
55        }
56      }catch (Exception e){
57        e.printStackTrace();
58      }
59    }
60  }
```

3.7.2 Spezifische Formelklassen

Jetzt geht es an Darstellung der unterschiedlichen Formelarten, wie Disjunktionen, Implikationen usw. Hierfür wird jeweils eine Unterklasse von `Formel` definiert. In den einzelnen Unterklassen sind die Methoden zur Umformung gegebenenfalls zu überschreiben.

Aussagenlogische Variablen

Die einfachste Formel besteht nur aus einer einzigen aussagenlogischen Variablen. Diese hat einen Namen, der als String dargestellt werden kann. Wir können zur Darstellung aussagenlogischer Variablen entsprechend eine einfache Klasse vorsehen:

Var.java
```
1  class Var extends Formel {
2    String name;
3
4    public Var(String name) {
5      this.name = name;
6    }
```

Die Methode `toString` soll gerade den Namen der Variablen zurück geben, ebenso die Methode `toLatex`.

Var.java
```
7    @Override
8    public String toString() {
9      return name;
10   }
11   @Override
12   String toLatex() {
13     return name;
14   }
15 }
```

Bei einer aussagenlogischen Variablen sind keine Umformungen vorzunehmen. Daher brauchen die drei Methoden zur Formelumformung nicht überschrieben werden.

Implikation

Als nächstes soll eine Klasse definiert werden, die Implikationsformeln darstellen kann. Eine Implikation ist ein binärer Operator, d.h. es werden zwei Formeln mit diesem verknüpft. Eine linke und eine rechte. Somit sehen wir für diese zwei Teilformeln der Implikation Felder in der Klasse vor. Ein Feld `lhs` für die linke Seite der Implikation, ein Feld `rhs` für die rechte Seite:

Imply.java
```
1  public class Imply extends Formel {
2    Formel lhs;
3    Formel rhs;
4    public Imply(Formel lhs, Formel rhs) {
5      this.lhs = lhs;
6      this.rhs = rhs;
7    }
```

In der Stringdarstellung einer Implikation sei der Implikationsfall durch die Zeichenkombination `->` ausgedrückt.

Imply.java
```
8    @Override
9    public String toString() {
10     return "("+lhs +" -> "+rhs+")";
11   }
```

Zur Darstellung einer Formel in LaTeX-Quelltext sind jeweils linke und rechte Seite als LaTeX-Quelltext zu erzeugen und der entsprechende Befehl eine nach rechts weisenden Pfeils in LaTeX zu setzen.

Imply.java
```
12   @Override
13   String toLatex() {
14     return "("+lhs.toLatex()+" \\rightarrow "+rhs.toLatex()+")";
15   }
```

Schließlich sind für die drei Umformungen die Methoden zu implementieren. Da es sich bei dieser Formel nicht um eine Negation handelt, ist auf der obersten Ebene der Formel auch keine Negation eventuell nach innen zu schieben. Allerdings kann es in den Teilformeln für die linke und rechte Seite notwendig sein Umformungen vorzunehmen, um alle Negationen direkt vor aussagenlogische Variablen zu schieben. Daher wird für das Ergebnis der Methode `negatNachInnen` ein neues Implikationsobjekt erzeugt. Die linke und rechte Teilformel dieser neuen Implikation sollen dadurch entstehen, dass die alten linken und rechten Teilformeln entsprechend umgeformt werden.

Imply.java
```
16   @Override
17   Formel negatNachInnen() {
18     return new Imply(lhs.negatNachInnen(), rhs.negatNachInnen());
19   }
```

Mit der gleichen Begründung ist die Methode `undUeberOder` auf analoge Weise zu implementieren:

Imply.java
```
20   @Override
21   Formel undUeberOder() {
22     return new Imply(lhs.undUeberOder(), rhs.undUeberOder());
23   }
```

Tatsächlich ist für Implikationsformeln in der Methode `eliminierePfeil` eine Umformung vorzunehmen. Implikationen lassen sich durch eine äquivalente Formel mit Negation und Disjunktion ausdrücken. Hierzu ist folgende Umformung vorzunehmen:
Ersetze $A \rightarrow B$ durch $\neg A \vee B$.

Hierzu brauchen wir natürlich Klassen um die Disjunktion und die Negation auszudrücken. Diese werden in den folgenden Abschnitten implementiert. Der obige Ersetzungsschritt, lässt sich dann in folgender Methode ausdrücken:

Imply.java
```
24   @Override
25   Formel eliminierePfeil() {
26     return new Or(new Not(lhs.eliminierePfeil()),rhs.eliminierePfeil());
27   }
28 }
```

Zu beachten ist, dass nicht nur allein das `this`-Objekt direkt umzuformen ist, sondern auch wieder die Teilformeln auf der linken und rechten Seite der Formel.

Äquivalenz

Als nächstes sei eine Klasse definiert, um logische Äquivalenzformeln auszudrücken. Diese Klasse ist sehr ähnlich zur Klasse für Implikationen. Es gibt wieder eine linke und eine rechte Seite. Die Methoden für die Umwandlungen in einen String und in LaTeX-Quelltext sind in analoger Weise zu implementieren:

Equiv.java
```
1  public class Equiv extends Formel {
2    Formel lhs;
```

```
3   Formel rhs;
4   public Equiv(Formel lhs, Formel rhs) {
5     this.lhs = lhs;
6     this.rhs = rhs;
7   }
8
9   @Override
10  public String toString() {
11    return "("+lhs +" <-> "+rhs+")";
12  }
13  @Override
14  String toLatex() {
15    return "("+lhs.toLatex()+" \\leftrightarrow "+rhs.toLatex()+")";
16  }
```

Die beiden Umformungen, die die Negation bzw. Disjunktion und Konjunktion betreffen, sind auf analoge Wiese zu implementieren, wie für die Implikation. Sie betreffen die aktuellen `this`-Formel nicht direkt, müssen aber in den linken und rechten Teilformeln weitergeführt werden.

Equiv.java

```
17  @Override
18  Formel negatNachInnen() {
19    return new Equiv(lhs.negatNachInnen(), rhs.negatNachInnen());
20  }
21  @Override
22  Formel undUeberOder() {
23    return new Equiv(lhs.undUeberOder(), rhs.undUeberOder());
24  }
```

Der Äquivalenzpfeil soll eliminiert werden und durch andere Junktoren ausgedrückt werden. Hierzu soll folgende Umformung vorgenommen werden:
Ersetze $A \leftrightarrow B$ durch $(A \wedge B \vee (\neg A \wedge \neg B)$.

Und auch diese Umformung ist auf die linke bzw. rechte Teilformel durchzuführen. Der entsprechende Javacode benutzt die im folgenden Abschnitt definierte Klasse zur Konjunktion `And`.

Equiv.java

```
25  @Override
26  Formel eliminierePfeil() {
27    Formel lhsN = lhs.eliminierePfeil();
28    Formel rhsN = rhs.eliminierePfeil();
29    return new Or(new And(lhsN,rhsN),new And(new Not(lhsN),new Not(rhsN)));
30  }
31
32 }
```

Die Konjunktion

Auch die Konjunktion besteht aus einer linken und rechten Teilformel. Somit ergeben sich die analogen Definitionen zu den beiden bereits gesehenen Klassen für binäre Operatoren:

And.java

```
1  public class And extends Formel {
2    Formel lhs;
3    Formel rhs;
4    public And(Formel lhs, Formel rhs) {
5      this.lhs = lhs;
6      this.rhs = rhs;
```

```
7   }
8   @Override
9   public String toString() {
10    return "("+lhs +" /\\ "+rhs+")";
11  }
12  @Override
13  String toLatex() {
14    return "("+lhs.toLatex()+" \\wedge "+rhs.toLatex()+")";
15  }
```

Für alle drei Umformungen ist für eine Konjunktionsformel direkt nichts durchzuführen. Es sind jeweils nur die linken und rechten Teilformeln zu transformieren. Somit wird jeweils mit den transformierten linken und rechten Teilformeln eine neue Konjunktion erzeugt.

And.java

```
16  @Override
17  Formel eliminierePfeil() {
18    return new And(lhs.eliminierePfeil(), rhs.eliminierePfeil());
19  }
20  @Override
21  Formel negatNachInnen() {
22    return new And(lhs.negatNachInnen(), rhs.negatNachInnen());
23  }
24  @Override
25  Formel undUeberOder() {
26    return new And(lhs.undUeberOder(), rhs.undUeberOder());
27  }
28 }
```

Disjunktion

Auch Disjunktionen gehen sehr analog zu Konjunktionen. Es gibt wieder linke und rechte Teilformeln

Or.java

```
1  public class Or extends Formel {
2    Formel lhs;
3    Formel rhs;
4
5    public Or(Formel lhs, Formel rhs) {
6      this.lhs = lhs;
7      this.rhs = rhs;
8    }
9
10   @Override
11   public String toString() {
12     return "("+lhs +" \\/ "+rhs+")";
13   }
14   @Override
15   String toLatex() {
16     return "("+lhs.toLatex()+" \\vee "+rhs.toLatex()+")";
17   }
```

Die beiden Umformungen, die sich mit der Negation bez. Elimination von Pfeilen beschäftigen müssen auch in disem Fall nur für die linke und rechte Teilformel durchgeführt werden.

Or.java

```
18  @Override
19  Formel eliminierePfeil() {
```

```
20      return new Or(lhs.eliminierePfeil(), rhs.eliminierePfeil());
21    }
22    @Override
23    Formel negatNachInnen() {
24      return new Or(lhs.negatNachInnen(), rhs.negatNachInnen());
25    }
```

Aufgabe 30 Implementieren Sie für die Klasse `Or` die Methode `undUeberOder`, so dass sie die folgende Umformung realisiert:

- Ersetze $(A \vee (B \wedge C))$ durch $(A \vee B) \wedge (A \vee C)$
- und ersetze $((B \wedge C) \vee A)$ durch $(B \vee A) \wedge (C \vee A)$.

Hinweis: Unterscheiden Sie mit dem Operator `instanceof` ob es sich bei dem linken oder rechten Teilformeln um eine Konjunktion handelt.

Mit der obigen Aufgabe ist dann auch die Klasse `Or` vollständig.

Or.java

```
26  }
```

Negation

Als letzte aussagenlogische Formel ist jetzt noch die Negation zu betrachten.

Die Negation ist ein unärer Operator. Er hat nur eine Teilformel. Diese steht nach dem Operator, also auf seiner rechten Seite. Daher sehen wir nur ein Feld für die Negationsklasse `Not` vor:

Not.java

```
1  public class Not extends Formel {
2    Formel rhs;
3
4    public Not(Formel rhs) {
5      this.rhs = rhs;
6    }
7     @Override
8    public String toString() {
9      return "|-"+rhs;
10   }
11   @Override
12   String toLatex() {
13     return "\\neg "+rhs.toLatex();
14   }
```

Und auch hier ist wieder nicht viel bei der Umformung, die Pfeiloperatoren eliminiert zu tun. Es ist die Umformung lediglich auf die negierte Teilformel anzuwenden.

Not.java

```
15   @Override
16   Formel eliminierePfeil() {
17     return new Not(rhs.eliminierePfeil());
18   }
```

Das Gleiche gilt für die Umformung, die Konjunktion und Disjunktion betrifft:

Not.java

```
19   @Override
20   Formel undUeberOder() {
21     return new Not(rhs.undUeberOder());
22   }
```

Es bleibt der interessanteste Fall. Die Negationssymbole so weit in die Formel wie möglich zu schieben. Dieser sei in einer Aufgabe realisiert:

Aufgabe 31 Implementieren Sie die Methode `negatNachInnen` für die Klasse `Not`. Sie soll folgende Umformungsschritte realsieren:

Ersetze sukzessive Teilformeln nach den folgenden Regeln:

- $\neg\neg A$ durch A,
- $\neg(A \vee B)$ durch $(\neg A \wedge \neg B)$,
- $\neg(A \wedge B)$ durch $\neg A \vee \neg B$

Mit der obigen Aufgabe ist dann auch die Klasse `Not` vollständig.

Not.java

```
23 }
```

3.7.3 Ein kleiner Testaufruf

TestFormel.java

```
1  public class TestFormel {
2
3    public static void main(String[] args) {
4      Formel x = new Var("x");
5      Formel y = new Var("y");
6      Formel imp = new Imply(x, y);
7      Formel nx = new Not(x);
8      Formel or = new Or(nx, y);
9      Formel reithB = new Equiv(imp, or);
10     System.out.println(reithB.toLatex());
11     System.out.println(reithB.eliminierePfeil().toLatex());
12     System.out.println(reithB.klauselForm().toLatex());
13     reithB.displayNormalisation();
14   }
15 }
```

Aufgabe 32 Testen Sie Ihre Bibliothek für logische Formeln und die Umwandlung in Klauselform an unterschiedlichen Beispielen. Erzeugen Sie hierzu ein paar LaTeX-Dokumente.

Kapitel 4

Weiterführende Konzepte

4.1 Pakete

Java bietet die Möglichkeit, Klassen in Paketen zu sammeln. Die Klassen eines Paketes bilden zumeist eine funktional logische Einheit. Pakete sind hierarchisch strukturiert, d.h. Pakete können Unterpakete haben. Damit entsprechen Pakete Ordnern im Dateisystem. Pakete ermöglichen verschiedene Klassen gleichen Namens, die unterschiedlichen Paketen zugeordnet sind.

4.1.1 Paketdeklaration

Zu Beginn einer Klassendefinition kann eine Paketzugehörigkeit für die Klasse definiert werden. Dieses geschieht mit dem Schlüsselwort `package` gefolgt von dem gewünschten Paket. Die Paketdeklaration schließt mit einem Semikolon.

Folgende Klasse definiert sie dem Paket `testPackage` zugehörig:

MyClass.java

```
1 package testPackage;
2 class MyClass {
3 }
```

Unterpakete werden von Paketen mit Punkten abgetrennt. Folgende Klasse wird dem Paket `testPackages` zugeordnet, das ein Unterpaket des Pakets `panitz` ist, welches wiederum ein Unterpaket des Pakets `name` ist:

TestPaket.java

```
1 package name.panitz.testPackages;
2 class TestPaket {
3   public static void main(String [] args){
4     System.out.println("hello from package \'testpackages\'");
5   }
6 }
```

Paketnamen werden per Konvention in lateinischer Schrift immer mit Kleinbuchstaben als erstem Buchstaben geschrieben.

Wie man sieht, kann man eine weltweite Eindeutigkeit seiner Paketnamen erreichen, wenn man die eigene Webadresse hierzu benutzt.[1] Dabei wird die Webadresse rückwärts verwendet.

Paketname und Klassenname zusammen identifizieren eine Klasse eindeutig. Jeder Programmierer schreibt sicherlich eine Vielzahl von Klassen `Test`, es gibt aber in der Regel nur einen Programmierer, der diese für das Paket `name.panitz.testPackages` schreibt. Paket- und Klassenname zusammen durch einen Punkt getrennt werden der *vollqualifizierte Name* der Klasse

[1] Leider ist es in Deutschland weit verbreitet, einen Bindestrich in Webadressen zu verwenden. Der Bindestrich ist leider eines der wenigen Zeichen, die Java in Klassen- und Paketnamen nicht zuläßt.

genannt, im obigen Beispiel ist entsprechend der vollqualifizierte Name:
`name.panitz.testPackages.Test`

Der Name einer Klasse ohne die Paketnennung heißt unqualifiziert.

4.1.2 Übersetzen von Paketen

Bei größeren Projekten ist es zu empfehlen, die Quelltexte der Javaklassen in Dateien zu speichern, die im Dateisystem in einer Ordnerstruktur, die der Paketstruktur entspricht, liegen. Dieses ist allerdings nicht unbedingt zwingend notwendig. Hingegen zwingend notwendig ist es, die erzeugten Klassendateien in Ordnern entsprechend der Paketstruktur zu speichern.

Der Javainterpreter `java` sucht nach Klassen in den Ordnern entsprechend ihrer Paketstruktur. `java` erwartet also, dass die obige Klasse `Test` in einem Ordner `testPackages` steht, der ein Unterordner des Ordners `panitz` ist, der ein Unterordner des Ordners `tfhberlin` ist. usw. `java` sucht diese Ordnerstruktur von einem oder mehreren Startordnern ausgehend. Die Startordner werden in einer Umgebungsvariablen `CLASSPATH` des Betriebssystems und über den Kommandozeilenparameter `-classpath` festgelegt.

Der Javaübersetzer `javac` hat eine Option, mit der gesteuert wird, dass `javac` für seine `.class`-Dateien die notwendige Ordnerstruktur erzeugt und die Klassen in die ihren Paketen entsprechenden Ordner schreibt. Die Option heißt `-d`. Dem `-d` ist nachgestellt, von welchem Startordner aus die Paketordner erzeugt werden sollen. Memotechnisch steht das `-d` für *destination*.

Wir können die obige Klasse z.B. übersetzen mit folgendem Befehl auf der Kommandozeile:
`javac -d . Test.java`

Damit wird ausgehend vom aktuellem Verzeichnis[2] ein Ordner `de` mit Unterordner `tfhberlin` etc. erzeugt.

4.1.3 Starten von Klassen in Paketen

Um Klassen vom Javainterpreter zu starten, reicht es nicht, ihren Namen anzugeben, sondern der vollqualifizierte Name ist anzugeben. Unsere obige kleine Testklasse wird also wie folgt gestartet:

```
sep@swe10:~/> java name.panitz.testPackages.Test
hello from package 'testpackages'
sep@swe10:~/>
```

Jetzt erkennt man auch, warum dem Javainterpreter nicht die Dateiendung `.class` mit angegeben wird. Der Punkt separiert Paket- und Klassennamen.

Aufmerksame Leser werden bemerkt haben, dass der Punkt in Java durchaus konsistent mit einer Bedeutung verwendet wird: hierzu lese man ihn als *'enthält ein'*. Der Ausdruck:
`name.panitz.testPackages.Test.main(args)`
liest sich so als: das Paket `de` enthält ein Unterpaket `tfhberlin`, das ein Unterpaket `panitz` enthält, das ein Unterpaket `testpackages` enthält, das eine Klasse `Test` enthält, die eine Methode `main` enthält.

[2] Der Punkt steht in den meisten Betriebssystemen für den aktuellen Ordner, in dem gerade ein Befehl ausgeführt wird.

4.1.4 Das Java Standardpaket

Die mit Java mitgelieferten Klassen sind auch in Paketen gruppiert. Die Standardklassen wie z.B. `String` und `System` und natürlich auch `Object` liegen im Java-Standardpaket `java.lang`. Java hat aber noch eine ganze Reihe weitere Pakete, so z.B. `java.util`, in dem sich Listenklassen befinden, `java.applet`, in dem Klassen zur Programmierung von Applets auf HTML-Seiten liegen, oder `java.io`, welches Klassen für Eingaben und Ausgaben enthält.

4.1.5 Benutzung von Klassen in anderen Paketen

Um Klassen benutzen zu können, die in anderen Paketen liegen, müssen diese eindeutig über ihr Paket identifiziert werden. Dieses kann dadurch geschehen, dass die Klassen immer vollqualifiziert angegeben werden. Im folgenden Beispiel benutzen wir die Standardklasse `ArrayList`[3] aus dem Paket `java.util`.

TestArrayList.java

```
1  package name.panitz.utilTest;
2  class TestArrayList {
3    public static void main(String [] args){
4      java.util.ArrayList<String> xs = new java.util.ArrayList<String>();
5      xs.add("friends");
6      xs.add("romans");
7      xs.add("countrymen");
8      System.out.println(xs);
9    }
10 }
```

Wie man sieht, ist der Klassenname auch beim Aufruf des Konstruktors vollqualifiziert anzugeben.

4.1.6 Importieren von Paketen und Klassen

Importieren von Klassen

Vollqualifizierte Namen können sehr lang werden. Wenn Klassen, die in einem anderen Paket als die eigene Klasse liegen, unqualifiziert benutzt werden sollen, dann kann dieses zuvor angegeben werden. Dieses geschieht zu Beginn einer Klasse in einer Importanweisung. Nur die Klassen aus dem Standardpaket `java.lang` brauchen nicht explizit durch eine Importanweisung bekannt gemacht zu werden.

Unsere Testklasse aus dem letzten Abschnitt kann mit Hilfe einer Importanweisung so geschrieben werden, dass die Klasse `ArrayList` unqualifiziert[4] benutzt werden kann:

TestImport.java

```
1  package name.panitz.utilTest;
2
3  import java.util.ArrayList;
4
5  class TestImport {
6    public static void main(String [] args){
7      ArrayList<String> xs = new ArrayList<String>();
8      xs.add("friends");
9      xs.add("romans");
```

[3] `ArrayList` ist eine generische Klasse, ein Konzept, das wir erst in einem späteren Kapitel kennenlernen werden.
[4] Aus historischen Gründen wird in diesem Kapitel als Beispiel bereits mit den generischen Klassen `ArrayList` und `Vector` ein Konzept benutzt, das erst im nächsten Kapitel erklärt wird.

```
10      xs.add("countrymen");
11      System.out.println(xs);
12    }
13  }
```

Es können mehrere Importanweisungen in einer Klasse stehen. So können wir z.B. zusätzlich die Klasse `Vector` importieren:

TestImport2.java

```
1  package name.panitz.utilTest;
2
3  import java.util.ArrayList;
4  import java.util.Vector;
5
6  class TestImport2 {
7    public static void main(String [] args){
8      ArrayList<String> xs = new ArrayList<String>();
9      xs.add("friends");
10     xs.add("romans");
11     xs.add("countrymen");
12     System.out.println(xs);
13
14     Vector<String> ys = new Vector<String>();
15     ys.add("friends");
16     ys.add("romans");
17     ys.add("countrymen");
18     System.out.println(ys);
19   }
20 }
```

Importieren von Paketen

Wenn in einem Programm viele Klassen eines Paketes benutzt werden, so können mit einer Importanweisung auch alle Klassen dieses Paketes importiert werden. Hierzu gibt man in der Importanweisung einfach statt des Klassennamens ein `*` an.

TestImport3.java

```
1  package name.panitz.utilTest;
2
3  import java.util.*;
4
5  class TestImport3 {
6    public static void main(String [] args){
7      List<String> xs = new ArrayList<String>();
8      xs.add("friends");
9      System.out.println(xs);
10
11     Vector<String> ys = new Vector<String>();
12     ys.add("romans");
13     System.out.println(ys);
14   }
15 }
```

Ebenso wie mehrere Klassen können auch mehrere komplette Pakete importiert werden. Es können auch gemischt einzelne Klassen und ganze Pakete importiert werden.

4.1.7 Statische Imports

Statische Eigenschaften einer Klasse werden in Java dadurch angesprochen, dass dem Namen der Klasse mit Punkt getrennt die gewünschte Eigenschaft folgt. Werden in einer Klasse sehr oft statische Eigenschaften einer anderen Klasse benutzt, so ist der Code mit deren Klassennamen durchsetzt. Die Javaentwickler haben mit Java 1.5 ein Einsehen. Man kann jetzt für eine Klasse alle ihre statischen Eigenschaften importieren, so dass diese unqualifiziert benutzt werden kann. Die `import`-Anweisung sieht aus wie ein gewohntes Paktimport, nur dass das Schlüsselwort `static` eingefügt ist und erst dem klassennamen der Stern folgt, der in diesen Fall für alle statischen Eigenschaften steht.

Beispiel 4.1.1 *Wir schreiben eine Hilfsklasse zum Arbeiten mit Strings, in der wir eine Methode zum umdrehen eines Strings vorsehen:*

StringUtil.java

```
1 package name.panitz.staticImport;
2 public class StringUtil {
3   static public String reverse(String arg) {
4     StringBuffer result = new StringBuffer();
5     for (char c:arg.toCharArray()) result.insert(0,c);
6     return result.toString();
7   }
8 }
```

Die Methode `reverse` *wollen wir in einer anderen Klasse benutzen. Importieren wir die statischen Eigenschaften von* `StringUtil`*, so können wir auf die Qualifizierung des Namens der Methode* `reverse` *verzichten:*

UseStringUtil.java

```
1 package name.panitz.staticImport;
2 import static name.panitz.staticImport.StringUtil.*;
3 public class UseStringUtil {
4   static public void main(String [] args) {
5     for (String arg:args)
6      System.out.println(reverse(arg));
7   }
8 }
```

Die Ausgabe dieses programms:

```
sep@linux:fh> java -classpath classes/ name.panitz.staticImport.UseStringUtil hallo welt
ollah
tlew
sep@linux:~/fh/java1.5/examples>
```

4.2 Sichtbarkeitsattribute

Sichtbarkeiten[5] erlauben es, zu kontrollieren, wer auf Klassen und ihre Eigenschaften zugreifen kann. Das *wer* bezieht sich hierbei auf andere Klassen und Pakete.

4.2.1 Sichtbarkeitsattribute für Klassen

Für Klassen gibt es zwei Möglichkeiten der Sichtbarkeit. Entweder darf von überall aus eine Klasse benutzt werden oder nur von Klassen im gleichen Paket. Syntaktisch wird dieses dadurch

[5] Man findet in der Literatur auch den Ausdruck *Erreichbarkeiten*.

ausgedrückt, dass der Klassendefinition entweder das Schlüsselwort `public` vorangestellt ist oder aber kein solches Attribut voransteht:

MyPublicClass.java

```
1 package name.panitz.p1;
2 public class MyPublicClass {
3 }
```

MyNonPublicClass.java

```
1 package name.panitz.p1;
2 class MyNonPublicClass {
3 }
```

In einem anderen Paket dürfen wir nur die als öffentlich deklarierte Klasse benutzen. Folgende Klasse übersetzt fehlerfrei:

UsePublic.java

```
1 package name.panitz.p2;
2
3 import name.panitz.p1.*;
4
5 class UsePublic {
6   public static void main(String [] args){
7     System.out.println(new MyPublicClass());
8   }
9 }
```

Der Versuch, eine nicht öffentliche Klasse aus einem anderen Paket heraus zu benutzen, gibt hingegen einen Übersetzungsfehler:

```
1 package name.panitz.p2;
2
3 import name.panitz.p1.*;
4
5 class UseNonPublic {
6   public static void main(String [] args){
7     System.out.println(new MyNonPublicClass());
8   }
9 }
```

Java gibt bei der Übersetzung eine entsprechende gut verständliche Fehlermeldung:

```
sep@swe10:~> javac -d . UseNonPublic.java
UseNonPublic.java:7: name.panitz.p1.MyNonPublicClass is not
public in name.panitz.pantitz.p1;
cannot be accessed from outside package
       System.out.println(new MyNonPublicClass());
                          ^
UseNonPublic.java:7: MyNonPublicClass() is not
public in name.panitz.p1.MyNonPublicClass;
cannot be accessed from outside package
       System.out.println(new MyNonPublicClass());
                          ^
2 errors
sep@swe10:~>
```

Damit stellt Java eine Technik zur Verfügung, die es erlaubt, bestimmte Klassen eines Softwarepaketes als rein interne Klassen zu schreiben, die von außerhalb des Pakets nicht benutzt werden können.

4.2.2 Sichtbarkeitsattribute für Eigenschaften

Java stellt in Punkto Sichtbarkeiten eine noch feinere Granularität zur Verfügung. Es können nicht nur ganze Klassen als nicht-öffentlich deklariert, sondern für einzelne Eigenschaften von Klassen unterschiedliche Sichtbarkeiten deklariert werden.

Für Eigenschaften gibt es vier verschiedene Sichtbarkeiten:
`public`, `protected`, kein Attribut, `private`

Sichbarkeiten hängen zum einem von den Paketen ab, in denen sich die Klassen befinden, darüberhinaus unterscheiden sich Sichtbarkeiten auch darin, ob Klassen Unterklassen voneinander sind. Folgende Tabelle gibt eine Übersicht über die vier verschiedenen Sichtbarkeiten:

Attribut	Sichtbarkeit
`public`	Die Eigenschaft darf von jeder Klasse aus benutzt werden.
`protected`	Die Eigenschaft darf für jede Unterklasse und jede Klasse im gleichen Paket benutzt werden.
kein Attribut	Die Eigenschaft darf nur von Klassen im gleichen Paket benutzt werden.
`private`	Die Eigenschaft darf nur von der Klasse, in der sie definiert ist, benutzt werden.

Damit kann in einer Klasse auf Eigenschaften mit jeder dieser vier Sichtbarkeiten zugegriffen werden. Wir können die Fälle einmal systematisch durchprobieren. In einer öffentlichen Klasse eines Pakets `p1` definieren wir hierzu vier Felder mit den vier unterschiedlichen Sichtbarkeiten:

VisibilityOfFeatures.java

```
1  package name.panitz.p1;
2
3  public class VisibilityOfFeatures{
4    private   String s1 = "private";
5              String s2 = "package";
6    protected String s3 = "protected";
7    public    String s4 = "private";
8
9    public static void main(String [] args){
10     VisibilityOfFeatures v = new VisibilityOfFeatures();
11     System.out.println(v.s1);
12     System.out.println(v.s2);
13     System.out.println(v.s3);
14     System.out.println(v.s4);
15   }
16 }
```

In der Klasse selbst können wir auf alle vier Felder zugreifen.

In einer anderen Klasse, die im gleichen Paket ist, können private Eigenschaften nicht mehr benutzt werden:

PrivateTest.java

```
1  package name.panitz.p1;
2
3  public class PrivateTest
4  {
5
6    public static void main(String [] args){
7      VisibilityOfFeatures v = new VisibilityOfFeatures();
8      //s1 is private and cannot be accessed;
9      //we are in a  different class.
```

```
10      //System.out.println(v.s1);
11      System.out.println(v.s2);
12      System.out.println(v.s3);
13      System.out.println(v.s4);
14    }
15  }
```

Von einer Unterklasse können unabhängig von ihrem Paket die *geschützten* Eigenschaften benutzt werden. Ist die Unterklasse in einem anderen Paket, können Eigenschaften mit der Sichtbarkeit `package` nict mehr benutzt werden:

PackageTest.java

```
1  package name.panitz.p2;
2  import name.panitz.p1.VisibilityOfFeatures;
3
4  public class  PackageTest extends VisibilityOfFeatures{
5
6    public static void main(String [] args){
7      PackageTest v = new PackageTest();
8      //s1 is private and cannot be accessed
9      // System.out.println(v.s1);
10
11     //s2 is package visible and cannot be accessed;
12     //we are in a different package.
13     //System.out.println(v.s2);
14
15     System.out.println(v.s3);
16     System.out.println(v.s4);
17   }
18 }
```

Von einer Klasse, die weder im gleichen Paket noch eine Unterklasse ist, können nur noch öffentliche Eigenschaften benutzt werden:

ProtectedTest.java

```
1  package name.panitz.p2;
2  import name.panitz.p1.VisibilityOfFeatures;
3
4  public class ProtectedTest  {
5
6    public static void main(String [] args){
7      VisibilityOfFeatures v = new VisibilityOfFeatures();
8      //s1 is private and cannot be accessed
9      // System.out.println(v.s1);
10
11     //s2 is package visible and cannot be accessed. We are
12     //in a different package
13     //System.out.println(v.s2);
14
15     //s2 is protected and cannot be accessed.
16     //We are not a subclass
17     //System.out.println(v.s3);
18
19     System.out.println(v.s4);
20   }
21 }
```

Java wird in seinem Sichtbarkeitskonzept oft kritisiert, und das von zwei Seiten. Einerseits ist es mit den vier Sichtbarkeiten schon relativ unübersichtlich; die verschiedenen Konzepte

der Vererbung und der Pakete spielen bei Sichtbarkeiten eine Rolle. Andererseits ist es nicht vollständig genug und kann verschiedene denkbare Sichtbarkeiten nicht ausdrücken.

In der Praxis fällt die Entscheidung zwischen privaten und öffentlichen Eigenschaften leicht. Geschützte Eigenschaften sind hingegen selten. Das Gros der Eigenschaften hat die Standardsichtbarkeit der Paketsichtbarkeit.

4.2.3 Private Felder mit get- und set-Methoden

Der direkte Feldzufriff ist in bestimmten Anwendungen nicht immer wünschenswert, so z.B. wenn ein Javaprogramm verteilt auf mehreren Rechner ausgeführt wird oder wenn der Wert eines Feldes in einer Datenbank abgespeichert liegt. Dann ist es sinnvoll, den Zugriff auf ein Feld durch zwei Methoden zu kapseln: eine Methode, um den Wert des Feldes abzufragen, und eine Methode, um das Feld mit einem neuen Wert zu belegen. Solche Methoden heißen get- bzw. set-Methoden. Um technisch zu verhindern, dass direkt auf das Feld zugegriffen wird, wird das Feld hierzu als *private* attributiert und nur die get- und set-Methoden werden als öffentlich attributiert. Dabei ist die gängige Namenskonvention, dass zu einem Feld mit Namen `name` die get- und set-Methoden `getName` bzw. `setName` heißen.

Beispiel 4.2.1 *Eine kleine Klasse mit Kapselung eines privaten Feldes:*

```
1 package name.panitz.sep.skript;
2
3 public class GetSetMethod {
4   private int value=0;
5
6   public void setValue(int newValue) {value=newValue;}
7   public int  getvalue(){return value;}
8 }
```

4.2.4 Überschriebene Methodensichtbarkeiten

Beim Überschreiben einer Methode darf ihr Sichtbarkeitsattribut nicht enger gemacht werden. Man darf also eine öffentliche Methode aus der Oberklasse nicht mit einer privaten, geschützten oder paketsichtbaren Methode überschreiben. Der Javaübersetzer weist solche Versuche, eine Methode zu überschreiben, zurück:

```
1 class OverrideToString {
2   String toString(){return "Objekt der Klasse OverrideToString";}
3 }
```

Der Versuch, diese Klasse zu übersetzen, führt zu folgender Fehlermeldung:

```
ep@linux:~/fh/prog1/examples/src> javac OverrideToString.java
OverrideToString.java:2:
toString() in OverrideToString cannot override toString() in java.lang.Object;
attempting to assign weaker access privileges;
was public
  String toString(){return "Objekt der Klasse OverrideToString";}
         ^
1 error
sep@linux:~/fh/prog1/examples/src>
```

Die Oberklasse `Object` enthält eine öffentliche Methode `toString`. Wollen wir diese Methode überschreiben, muss sie mindestens so sichtbar sein wie in der Oberklassen.

4.3 Schnittstellen (Interfaces) und abstrakte Klassen

Wir haben schon Situationen kennengelernt, in denen wir eine Klasse geschrieben haben, von der nie ein Objekt konstruiert werden sollte, sondern für die wir nur Unterklassen definiert und instanziiert haben. Die Methoden in diesen Klassen hatten eine möglichst einfache Implementierung; sie sollten ja nie benutzt werden, sondern die überschreibenden Methoden in den Unterklassen. Ein Beispiel für eine solche Klassen war die Klasse `ButtonLogic`, mit der die Funktionalität eines GUIs definiert wurde.

Java bietet ein weiteres Konzept an, mit dem Methoden ohne eigentliche Implementierung deklariert werden können, die Schnittstellen.

4.3.1 Schnittstellen

Schnittstellendeklaration

Eine Schnittstelle sieht einer Klasse sehr ähnlich. Die syntaktischen Unterschiede sind:

- statt des Schlüsselworts `class` steht das Schlüsselwort `interface`.
- die Methoden haben keine Rümpfe, sondern nur eine Signatur.

So läßt sich für unsere Klasse `ButtonLogic` eine entsprechende Schnittstelle schreiben:

DialogueLogic.java

```
1 package name.panitz.dialoguegui;
2
3 public interface DialogueLogic {
4   public String getDescription();
5   public String eval(String input);
6 }
```

Schnittstellen sind ebenso wie Klassen mit dem Javaübersetzer zu übersetzen. Für Schnittstellen werden auch Klassendateien mit der Endung `.class` erzeugt.

Im Gegensatz zu Klassen haben Schnittstellen keinen Konstruktor. Das bedeutet insbesondere, dass mit einer Schnittstelle kein Objekt erzeugt werden kann. Was hätte ein solches Objekt auch für ein Verhalten? Die Methoden haben ja gar keinen Code, den sie ausführen könnten. Eine Schnittstelle ist vielmehr ein Versprechen, dass Objekte Methoden mit den in der Schnittstelle definierten Signaturen enthalten. Objekte können aber immer nur über Klassen erzeugt werden.

Implementierung von Schnittstellen

Objekte, die die Funktionalität einer Schnittstelle enthalten, können nur mit Klassen erzeugt werden, die diese Schnittstelle implementieren. Hierzu gibt es zusätzlich zur `extends`-Klausel in Klassen auch noch die Möglichkeit, eine `implements`-Klausel anzugeben.

To Do: implements vs extends erklärem

Eine mögliche Implementierung der obigen Schnittstelle ist:

ToUpperCase.java

```
1 package name.panitz.dialoguegui;
2
3 public class ToUpperCase implements DialogueLogic{
4   protected String result;
5
6   public String getDescription(){
7     return "convert into upper cases";
```

```
8   }
9   public String eval(String input){
10    result = input.toUpperCase();
11    return result;
12  }
13 }
```

Die Klausel `implements DialogueLogic` verspricht, dass in dieser Klasse für alle Methoden aus der Schnittstelle eine Implementierung existiert. In unserem Beispiel waren zwei Methoden zu implementieren, die Methode `eval` und `getDescription()`.

Im Gegensatz zur `extends`-Klausel von Klassen können in einer `implements`-Klausel auch mehrere Schnittstellen angegeben werden, die implementiert werden.

Definieren wir zum Beispiel ein zweite Schnittstelle:

ToHTMLString.java

```
1 package name.panitz.html;
2
3 public interface ToHTMLString {
4   public String toHTMLString();
5 }
```

Diese Schnittstelle verlangt, dass implementierende Klassen eine Methode haben, die für das Objekt eine Darstellung als HTML erzeugen können.

Jetzt können wir eine Klasse schreiben, die die beiden Schnittstellen implementiert.

ToUpper.java

```
1  package name.panitz.dialoguegui;
2
3  import name.panitz.html.*;
4
5  public class ToUpper extends    ToUpperCase
6                       implements ToHTMLString, DialogueLogic {
7
8    public String toHTMLString(){
9       return   "<html><head><title>"+getDescription()
10             + "</title></head>"
11             + "<body><b>Small Gui application</b>"
12             + " for convertion of "
13             + " a <b>String</b> into <em>upper</em>"
14             + " case letters.<br></br>"
15             + "The result of your query was: <p>"
16             + "<span style=\"font-family: monospace;\">"
17             + result
18             + "</span></p></body></html>";
19   }
20 }
```

Schnittstellen können auch einander erweitern. Dieses geschieht dadurch, dass Schnittstellen auch eine `extends`-Klausel haben. Wir können also auch eine Schnittstelle definieren, die die beiden obigen Schnittstellen zusammenfaßt:

DialogueLogics.java

```
1 package name.panitz.dialoguegui;
2
3 import name.panitz.html.*;
4
5 public interface DialogueLogics
6         extends ToHTMLString, DialogueLogic {}
```

Ebenso können wir jetzt eine Klasse ableiten, die diese Schnittstelle implementiert:

UpperConversion.java

```
1 package name.panitz.dialoguegui;
2
3 class UpperConversion extends ToUpper
4                       implements DialogueLogics{}
```

Benutzung von Schnittstellen

Schnittstellen sind genauso Typen wie Klassen. Wir kennen jetzt also drei Arten von Typen:

- primitive Typen
- Klassen
- Schnittstellen

Parameter können vom Typ einer Schnittstellen sein, ebenso wie Felder oder Rückgabetypen von Methoden. Die Zuweisungkompatibilität nutzt nicht nur die Unterklassenbeziehung, sondern auch die Implementierungsbeziehung. Ein Objekt der Klasse `C` darf einem Feld des Typs der Schnittstelle `I` zugewiesen werden, wenn `C` die Schnittstelle `I` implementiert.

Im Folgenden eine kleine Gui-Anwendung, die wir im einzelnen noch nicht verstehen müssen. Man beachte, dass der Typ `DialogueLogics` an mehreren Stellen benutzt wird wie ein ganz normaler Klassentyp. Nur einen Konstruktoraufruf mit `new` können wir für diesen Typ nicht machen.

HtmlDialogue.java

```
1  package name.panitz.dialoguegui;
2  import java.awt.event.*;
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.plaf.basic.*;
6  import javax.swing.text.*;
7  import javax.swing.text.html.*;
8
9  public class HtmlDialogue extends JFrame{
10   final DialogueLogics logic;
11   final JButton button;
12   final JTextField inputField = new JTextField(20) ;
13   final JTextPane outputField = new JTextPane();
14   final JPanel p = new JPanel();
15
16   public HtmlDialogue(DialogueLogics l){
17     outputField.setEditorKit(new HTMLEditorKit());
18     logic = l;
19     button=new JButton(logic.getDescription());
20     button.addActionListener
21      (new ActionListener(){
22         public void actionPerformed(ActionEvent _){
23           logic.eval(inputField.getText().trim());
24           outputField.setText(logic.toHTMLString());
25           pack();
26         }
27       });
28
29     p.setLayout(new BorderLayout());
30     p.add(inputField,BorderLayout.NORTH);
31     p.add(button,BorderLayout.CENTER);
```

```
32      p.add(outputField,BorderLayout.SOUTH);
33      getContentPane().add(p);
34      pack();
35      setVisible(true);
36    }
37  }
```

Schließlich können wir ein Objekt der Klasse `UpperConversion`, die die Schnittstelle `DialogueLogics` implementiert, konstruieren und der Gui-Anwendung übergeben:

HtmlDialogueTest.java

```
1  package name.panitz.dialoguegui;
2  public class HtmlDialogueTest {
3    public static void main(String [] args){
4      new HtmlDialogue(new UpperConversion());
5    }
6  }
```

Die Anwendung in voller Aktion kann in Abbildung 4.1 bewundert werden.

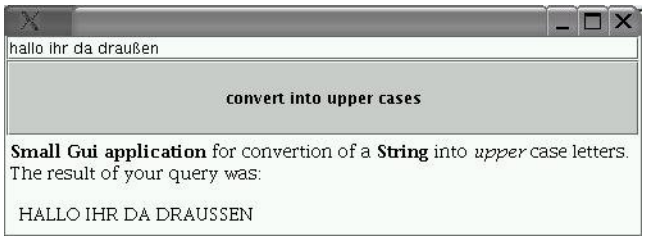

Abbildung 4.1: Ein Gui-Dialog mit Html-Ausgabe.

Semantische Einschränkungen für Schnittstellen

Es gibt einige semantische Einschränkungen, die über die syntaktischen Einschränkungen hinausgehen:

- Schnittstellen können nur Schnittstellen, nicht aber Klassen erweitern.
- Jede Methode einer Schnittstelle muss öffentlich sein, braucht also das Attribut `public`. Wenn dieses für eine Methode nicht deklariert ist, so wird Java dieses von selbst hinzufügen. Trotzdem müssen implementierende Klassen diese Methode dann als öffentlich deklarieren. Daher ist es besser, das Attribut `public` auch hinzuschreiben.
- Es gibt keine statischen Methoden in Schnittstellen.
- Jede Methode ist abstrakt, d.h. hat keinen Rumpf. Man kann dieses noch zusätzlich deutlich machen, indem man das Attribut `abstract` für die Methode mit angibt.
- Felder einer Schnittstelle sind immer statisch, brauchen also das Attribut `static` und zusätzlich noch das Attribut `final`.

Aufgabe 33 Legen Sie die Klassen aus Übungsblatt 3 in ein Paket, das Sie eindeutig charakterisiert.

Aufgabe 34 Lassen Sie jetzt zusätzlich die Klasse `GeometricObject` folgende Schnittstelle implementieren:

GameObject.java

```
1  package name.panitz;
2  import java.awt.Graphics;
3  public interface GameObject{
4    public boolean touches(GameObject that);
5    public Vertex getPosition();
6    public int getWidth();
7    public int getHeight();
8    public void paintMeTo(Graphics g);
9    public void move();
10   public void reverse();
11   public double weight();
12 }
```

Die müssen hierbei noch in der Schnittstelle Ihre Klasse Vertex importieren.

a) `touches` soll prüfen, ob die umgebenen Vierecke zweier Spielobjekte gemeinsame Punkte haben.

b) `paintMeTo` soll das Objekt auf eine Leinwand zeichnen.

c) `move` soll das Objekt einer Bahn einen Schritt weiterbewegen.

d) `reverse` soll die Bewegungsrichtung des Objekts umdrehen.

e) weight soll die Schwere des Objektes ausdrücken. Diese soll möglichst mit dem Flächeninhalt ausgedrückt sein.

f) Schreiben Sie JUnit-Test für die Methode `touches`.

Aufgabe 35 Erzeugen Sie in einer Testmethode ein Objekt der folgenden Klasse:

GameWindow.java

```
1  package name.panitz;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.util.ArrayList;
5  import java.util.List;
6  import javax.swing.*;
7
8  public class GameWindow extends JFrame{
9    JPanel p = new JPanel(){
10     public Dimension getPreferredSize() {
11       return new Dimension(800,600);
12     };
13     protected void paintComponent(Graphics g) {
14       for (GameObject o:os) o.paintMeTo(g);
15     };
16   };
17   final List<GameObject> os=new ArrayList<GameObject>();
18   public GameWindow() {
19     add(p);
20     pack();
21     setVisible(true);
22     Timer t = new Timer(100,new ActionListener(){
23        public void actionPerformed(ActionEvent e){
24          for (GameObject o:os) o.move();
25          repaint();
```

```
26          }
27        }
28        );
29        t.start();
30      }
31      public void addGameObject(GameObject o){
32        os.add(o);
33        p.repaint();
34      }
35  }
```

Fügen Sie dem Objekt dabei jeweils unterschiedliche Objekte der Klassen, die die Schnittstelle `GameObject` implementieren, mit der Methode `addGameObject` hinzu.

Aufgabe 36 Implementieren Sie eine Unterklasse `EquilateralTriangle` der Klasse `GeometricObject`, die gleichseitige Dreiecke darstellt. Schreiben Sie geeignete Konstruktoren und überschreiben Sie alle Methoden auf eine sinnvolle Art und Weise. Der Konstruktor soll die Seitenlänge des gleichseitigen Dreiecks erhalten.

Kapitel 5

Generische Programmierung

5.1 Generische Typen

Generische Typen wurden im JSR014 definiert. In der Expertengruppe des JSR014 war der Autor dieses Skripts zeitweilig als Stellvertreter der Software AG Mitglied. Die Software AG hatte mit der Programmiersprache Bolero bereits einen Compiler für generische Typen implementiert[Pan00]. Der Bolero Compiler generiert auch Java Byte Code. Von dem ersten Wunsch nach Generizität bis zur nun vorliegenden Javaversion 1.5 sind viele Jahre vergangen. Andere wichtige JSRs, die in Java 1.5 integriert werden, tragen bereits die Nummern 175 und 201. Hieran kann man schon erkennen, wie lange es gedauert hat, bis generische Typen in Java integriert wurden.

Interessierten Programmierern steht schon seit Mitte der 90er Jahre eine Javaerweiterung mit generischen Typen zur Verfügung. Unter den Namen Pizza [OW97] existiert eine Javaerweiterung, die nicht nur generische Typen, sondern auch algebraische Datentypen mit *pattern matching* und Funktionsobjekten zu Java hinzufügte. Unter den Namen *GJ* für *Generic Java* wurde eine allein auf generische Typen abgespeckte Version von *Pizza* publiziert. *GJ* ist tatsächlich der direkte Prototyp für Javas generische Typen. Die Expertenrunde des JSR014 hat *GJ* als Grundlage für die Spezifikation genommen und an den grundlegenden Prinzipien auch nichts mehr geändert.

5.1.1 Generische Klassen

Die Idee für generische Typen ist, eine Klasse zu schreiben, die für verschiedene Typen als Inhalt zu benutzen ist. Das geht bisher in Java, allerdings mit einem kleinen Nachteil. Versuchen wir einmal, in traditionellem Java eine Klasse zu schreiben, in der wir beliebige Objekte speichern können. Um beliebige Objekte speichern zu können, brauchen wir ein Feld, in dem Objekte jeden Typs gespeichert werden können. Dieses Feld muss daher den Typ `Object` erhalten:

OldBox.java

```
1 class OldBox {
2   Object contents;
3   OldBox(Object contents){this.contents=contents;}
4 }
```

Der Typ `Object` ist ein sehr unschöner Typ; denn mit ihm verlieren wir jegliche statische Typinformation. Wenn wir die Objekte der Klasse `OldBox` benutzen wollen, so verlieren wir sämtliche Typinformation über das in dieser Klasse abgespeicherte Objekt. Wenn wir auf das Feld `contents` zugreifen, so haben wir über das darin gespeicherte Objekte keine spezifische Information mehr. Um das Objekt weiter sinnvoll nutzen zu können, ist eine dynamische Typzusicherung durchzuführen:

UseOldBox.java

```
1 class UseOldBox{
2   public static void main(String [] _){
```

```
3     OldBox b = new OldBox("hello");
4     String s = (String)b.contents;
5     System.out.println(s.toUpperCase());
6     System.out.println(((String) s).toUpperCase());
7   }
8 }
```

Wann immer wir mit dem Inhalt des Felds `contents` arbeiten wollen, ist die Typzusicherung während der Laufzeit durchzuführen. Die dynamische Typzusicherung kann zu einem Laufzeitfehler führen. So übersetzt das folgende Programm fehlerfrei, ergibt aber einen Laufzeitfehler:

UseOldBoxError.java

```
1 class UseOldBoxError{
2   public static void main(String [] _){
3     OldBox b = new OldBox(new Integer(42));
4     String s = (String)b.contents;
5     System.out.println(s.toUpperCase());
6   }
7 }
```

```
sep@linux:~/fh/java1.5/examples/src> javac UseOldBoxError.java
sep@linux:~/fh/java1.5/examples/src> java UseOldBoxError
Exception in thread "main" java.lang.ClassCastException
        at UseOldBoxError.main(UseOldBoxError.java:4)
sep@linux:~/fh/java1.5/examples/src>
```

Wie man sieht, verlieren wir Typsicherheit, sobald der Typ `Object` benutzt wird. Bestimmte Typfehler können nicht mehr statisch zur Übersetzungszeit, sondern erst dynamisch zur Laufzeit entdeckt werden.

Der Wunsch ist, Klassen zu schreiben, die genauso allgemein benutzbar sind wie die Klasse `OldBox` oben, aber trotzdem die statische Typsicherheit garantieren, indem sie nicht mit dem allgemeinen Typ `Object` arbeiten. Genau dieses leisten generische Klassen. Hierzu ersetzen wir in der obigen Klasse jedes Auftreten des Typs `Object` durch einen Variablennamen. Diese Variable ist eine Typvariable. Sie steht für einen beliebigen Typen. Dem Klassennamen fügen wir zusätzlich in der Klassendefinition in spitzen Klammern eingeschlossen hinzu, dass diese Klasse eine Typvariable benutzt. Wir erhalten somit aus der obigen Klasse `OldBox` folgende generische Klasse `Box`.

Box.java

```
1 class Box<elementType> {
2   elementType contents;
3   Box(elementType contents){this.contents=contents;}
4 }
```

Die Typvariable `elementType` ist als allquantifiziert zu verstehen. Für jeden Typ `elementType` können wir die Klasse `Box` benutzen. Man kann sich unsere Klasse `Box` analog zu einer realen Schachtel vorstellen: Beliebige Dinge können in die Schachtel gelegt werden. Betrachten wir dann allein die Schachtel von außen, können wir nicht mehr wissen, was für ein Objekt darin enthalten ist. Wenn wir viele Dinge in Schachteln packen, dann schreiben wir auf die Schachtel jeweils drauf, was in der entsprechenden Schachtel enthalten ist. Ansonsten würden wir schnell die Übersicht verlieren. Und genau das ermöglichen generische Klassen. Sobald wir ein konkretes Objekt der Klasse `Box` erzeugen wollen, müssen wir entscheiden, für welchen Inhalt wir eine Box brauchen. Dieses geschieht, indem in spitzen Klammern dem Klassennamen `Box` ein entsprechender Typ für den Inhalt angehängt wird. Wir erhalten dann z.B. den Typ `Box<String>`, um Strings in der Schachtel zu speichern, oder `Box<Integer>`, um Integerobjekte darin zu speichern:

UseBox.java

```
1  class UseBox{
2    public static void main(String [] _){
3      Box<String> b1 = new Box<String>("hello");
4      String s = b1.contents;
5      System.out.println(s.toUpperCase());
6      System.out.println(b1.contents.toUpperCase());
7
8      Box<Integer> b2 = new Box<Integer>(new Integer(42));
9
10     System.out.println(b2.contents.intValue());
11   }
12 }
```

Wie man im obigen Beispiel sieht, fallen jetzt die dynamischen Typzusicherungen weg. Die Variablen `b1` und `b2` sind jetzt nicht einfach vom Typ `Box`, sondern vom Typ `Box<String>` respektive `Box<Integer>`.

Da wir mit generischen Typen keine Typzusicherungen mehr vorzunehmen brauchen, bekommen wir auch keine dynamischen Typfehler mehr. Der Laufzeitfehler, wie wir ihn ohne die generische Box hatten, wird jetzt bereits zur Übersetzungszeit entdeckt. Hierzu betrachte man das analoge Programm:

```
1  class UseBoxError{
2    public static void main(String [] _){
3      Box<String> b = new Box<String>(new Integer(42));
4      String s = b.contents;
5      System.out.println(s.toUpperCase());
6    }
7  }
```

Die Übersetzung dieses Programms führt jetzt bereits zu einen statischen Typfehler:

```
sep@linux:~/fh/java1.5/examples/src> javac UseBoxError.java
UseBoxError.java:3: cannot find symbol
symbol  : constructor Box(java.lang.Integer)
location: class Box<java.lang.String>
    Box<String> b = new Box<String>(new Integer(42));
                    ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

Vererbung

Generische Typen sind ein Konzept, das orthogonal zur Objektorientierung ist. Von generischen Klassen lassen sich in gewohnter Weise Unterklassen definieren. Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein. So können wir unsere einfache Schachtelklasse erweitern, so dass wir zwei Objekte speichern können:

GPair.java

```
1  class GPair<at,bt> extends Box<at>{
2    GPair(at x,bt y){
3      super(x);
4      snd = y;
5    }
6
7    bt snd;
8
```

```
9   public String toString(){
10    return "("+contents+","+snd+")";
11  }
12 }
```

Die Klasse `GPair` hat zwei Typvariablen. Instanzen von `GPair` müssen angeben von welchem Typ die beiden zu speichernden Objekte sein sollen.

UsePair.java

```
1  class UsePair{
2    public static void main(String [] _){
3      GPair<String,Integer> p
4       = new GPair<String,Integer>("hallo",new  Integer(40));
5
6      System.out.println(p);
7      System.out.println(p.contents.toUpperCase());
8      System.out.println(p.snd.intValue()+2);
9    }
10 }
```

Wie man sieht kommen wir wieder ohne Typzusicherung aus. Es gibt keinen dynamischen Typcheck, der im Zweifelsfall zu einer Ausnahme führen könnte.

```
sep@linux:~/fh/java1.5/examples/classes> java UsePair
(hallo,40)
HALLO
42
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können auch eine Unterklasse bilden, indem wir mehrere Typvariablen zusammenfassen. Wenn wir uniforme Paare haben wollen, die zwei Objekte gleichen Typs speichern, können wir hierfür eine spezielle Paarklasse definieren.

UniPair.java

```
1  class UniPair<at> extends GPair<at,at>{
2    UniPair(at x,at y){super(x,y);}
3    void swap(){
4      final at z = snd;
5      snd = contents;
6      contents = z;
7    }
8  }
```

Da beide gespeicherten Objekte jeweils vom gleichen Typ sind, konnten wir jetzt eine Methode schreiben, in der diese beiden Objekte ihren Platz tauschen. Wie man sieht, sind Typvariablen ebenso wie unsere bisherigen Typen zu benutzen. Sie können als Typ für lokale Variablen oder Parameter genutzt werden.

UseUniPair.java

```
1  class UseUniPair{
2    public static void main(String [] _){
3      UniPair<String> p
4       = new UniPair<String>("welt","hallo");
5
6      System.out.println(p);
7      p.swap();
8      System.out.println(p);
9    }
10 }
```

Wie man bei der Benutzung der uniformen Paare sieht, gibt man jetzt natürlich nur noch einen konkreten Typ für die Typvariablen an. Die Klasse `UniPair` hat ja nur eine Typvariable.

```
sep@linux:~/fh/java1.5/examples/classes> java UseUniPair
(welt,hallo)
(hallo,welt)
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können aber auch Unterklassen einer generischen Klasse bilden, die nicht mehr generisch ist. Dann leiten wir für eine ganz spezifische Instanz der Oberklasse ab. So läßt sich z.B. die Klasse `Box` zu einer Klasse erweitern, in der nur noch Stringobjekte verpackt werden können:

StringBox.java

```
1 class StringBox extends Box<String>{
2   StringBox(String x){super(x);}
3 }
```

Diese Klasse kann nun vollkommen ohne spitze Klammern benutzt werden:

UseStringBox.java

```
1 class UseStringBox{
2   public static void main(String [] _){
3     StringBox b = new StringBox("hallo");
4     System.out.println(b.contents.length());
5   }
6 }
```

Einschränken der Typvariablen

Bisher standen in allen Beispielen die Typvariablen einer generischen Klasse für jeden beliebigen Objekttypen. Hier erlaubt Java uns, Einschränkungen zu machen. Es kann eingeschränkt werden, dass eine Typvariable nicht für alle Typen ersetzt werden darf, sondern nur für bestimmte Typen.

Versuchen wir einmal, eine Klasse zu schreiben, die auch wieder der Klasse `Box` entspricht, zusätzlich aber eine `set`-Methode hat und nur den neuen Wert in das entsprechende Objekt speichert, wenn es größer ist als das bereits gespeicherte Objekt. Hierzu müssen die zu speichernden Objekte in einer Ordnungsrelation vergleichbar sein, was in Java über die Implementierung der Schnittstelle `Comparable` ausgedrückt wird. Im herkömmlichen Java würden wir die Klasse wie folgt schreiben:

CollectMaxOld.java

```
1  class CollectMaxOld{
2    private Comparable value;
3
4    CollectMaxOld(Comparable x){value=x;}
5
6    void setValue(Comparable x){
7      if (value.compareTo(x)<0) value=x;
8    }
9
10   Comparable getValue(){return value;}
11 }
```

Die Klasse `CollectMaxOld` ist in der Lage, beliebige Objekte, die die Schnittstelle `Comparable` implementieren, zu speichern. Wir haben wieder dasselbe Problem wie in der Klasse `OldBox`: Greifen wir auf das gespeicherte Objekt mit der Methode `getValue` erneut zu, wissen wir nicht

mehr den genauen Typ dieses Objekts und müssen eventuell eine dynamische Typzusicherung durchführen, die zu Laufzeitfehlern führen kann.

Javas generische Typen können dieses Problem beheben. In gleicher Weise, wie wir die Klasse `Box` aus der Klasse `OldBox` erhalten haben, indem wir den allgemeinen Typ `Object` durch eine Typvariable ersetzt haben, ersetzen wir jetzt den Typ `Comparable` durch eine Typvariable, geben aber zusätzlich an, dass diese Variable für alle Typen steht, die die Untertypen der Schnittstelle `Comparable` sind. Dieses wird durch eine zusätzliche `extends`-Klausel für die Typvariable angegeben. Wir erhalten somit eine generische Klasse `CollectMax`:

CollectMax.java

```
1  class CollectMax <elementType extends Comparable>{
2    private elementType value;
3
4    CollectMax(elementType x){value=x;}
5
6    void setValue(elementType x){
7      if (value.compareTo(x)<0) value=x;
8    }
9
10   elementType getValue(){return value;}
11 }
```

Für die Benutzung diese Klasse ist jetzt für jede konkrete Instanz der konkrete Typ des gespeicherten Objekts anzugeben. Die Methode `getValue` liefert als Rückgabetyp nicht ein allgemeines Objekt des Typs `Comparable`, sondern exakt ein Objekt des Instanzstyps.

UseCollectMax.java

```
1  class UseCollectMax {
2    public static void main(String [] _){
3      CollectMax<String> cm = new CollectMax<String>("Brecht");
4      cm.setValue("Calderon");
5      cm.setValue("Horvath");
6      cm.setValue("Shakespeare");
7      cm.setValue("Schimmelpfennig");
8      System.out.println(cm.getValue().toUpperCase());
9    }
10 }
```

Wie man in der letzten Zeile sieht, entfällt wieder die dynamische Typzusicherung.

5.1.2 Generische Schnittstellen

Generische Typen erlauben es, den Typ `Object` in Typsignaturen zu eleminieren. Der Typ `Object` ist als schlecht anzusehen, denn er ist gleichbedeutend damit, dass keine Information über einen konkreten Typ während der Übersetzungszeit zur Verfügung steht. Im herkömmlichen Java ist in APIs von Bibliotheken der Typ `Object` allgegenwärtig. Sogar in der Klasse `Object` selbst begegnet er uns in Signaturen. Die Methode `equals` hat einen Parameter vom Typ `Object`, d.h. prinzipiell kann ein Objekt mit Objekten jeden beliebigen Typs verglichen werden. Zumeist will man aber nur gleiche Typen miteinander vergleichen. In diesem Abschnitt werden wir sehen, dass generische Typen es uns erlauben, allgemein eine Gleichheitsmethode zu definieren, in der nur Objekte gleichen Typs miteinander verglichen werden können. Hierzu werden wir eine generische Schnittstelle definieren.

Generische Typen erweitern sich ohne Umstände auf Schnittstellen. Im Vergleich zu generischen Klassen ist nichts Neues zu lernen. Syntax und Benutzung funktionieren auf die gleiche Weise.

Äpfel mit Birnen vergleichen

Um zu realisieren, dass nur noch Objekte gleichen Typs miteinander verglichen werden können, definieren wir eine Gleichheitsschnitstelle. In ihr wird eine Methode spezifiziert, die für die Gleichheit stehen soll. Die Schnittstelle ist generisch über den Typen, mit dem vergleichen werden soll.

EQ.java

```
1 interface EQ<otherType> {
2   public boolean eq(otherType other);
3 }
```

Jetzt können wir für jede Klasse nicht nur bestimmen, dass sie die Gleichheit implementieren soll, sondern auch, mit welchen Typen Objekte unserer Klasse verglichen werden sollen. Schreiben wir hierzu eine Klasse `Apfel`. Die Klasse Apfel soll die Gleichheit auf sich selbst implementieren. Wir wollen nur Äpfel mit Äpfeln vergleichen können. Daher definieren wir in der `implements`-Klausel, dass wir `EQ<Apfel>` implementieren wollen. Dann müssen wir auch die Methode `eq` implementieren, und zwar mit dem Typ `Apfel` als Parametertyp:

Apfel.java

```
1  class Apfel implements EQ<Apfel>{
2    String typ;
3
4    Apfel(String typ){
5      this.typ=typ;}
6
7    public boolean eq(Apfel other){
8      return this.typ.equals(other.typ);
9    }
10 }
```

Jetzt können wir Äpfel mit Äpfeln vergleichen:

TestEQ.java

```
1 class TestEq{
2   public static void main(String []_){
3     Apfel a1 = new Apfel("Golden Delicious");
4     Apfel a2 = new Apfel("Macintosh");
5     System.out.println(a1.eq(a2));
6     System.out.println(a1.eq(a1));
7   }
8 }
```

Schreiben wir als nächstes eine Klasse die Birnen darstellen soll. Auch diese implementiere die Schnittstelle `EQ`, und zwar dieses Mal für Birnen:

Birne.java

```
1  class Birne implements EQ<Birne>{
2    String typ;
3
4    Birne(String typ){
5      this.typ=typ;}
6
7    public boolean eq(Birne other){
8      return this.typ.equals(other.typ);
9    }
10 }
```

Während des statischen Typchecks wird überprüft, ob wir nur Äpfel mit Äpfeln und Birnen mit Birnen vergleichen. Der Versuch, Äpfel mit Birnen zu vergleichen, führt zu einem Typfehler:

```
1 class TesteEqError{
2   public static void main(String []_){
3     Apfel a = new Apfel("Golden Delicious");
4     Birne b = new Birne("williams");
5     System.out.println(a.equals(b));
6     System.out.println(a.eq(b));
7   }
8 }
```

Wir bekommen die verständliche Fehlermeldung, dass die Gleichheit auf Äpfel nicht für einen Birnenparameter aufgerufen werden kann.

```
./TestEQError.java:6: eq(Apfel) in Apfel cannot be applied to (Birne)
    System.out.println(a.eq(b));
                       ^
1 error
```

Wahrscheinlich ist es jedem erfahrenden Javaprogrammierer schon einmal passiert, dass er zwei Objekte verglichen hat, die er gar nicht vergleichen wollte. Da der statische Typcheck solche Fehler nicht erkennen kann, denn die Methode `equals` läßt jedes Objekt als Parameter zu, sind solche Fehler mitunter schwer zu lokalisieren.

Der statische Typcheck stellt auch sicher, dass eine generische Schnittstelle mit der korrekten Signatur implementiert wird. Der Versuch, eine Birneklasse zu schreiben, die eine Gleichheit mit Äpfeln implementieren soll, dann aber die Methode `eq` mit dem Parametertyp `Birne` zu implementieren, führt ebenfalls zu einer Fehlermeldung:

```
1 class BirneError implements EQ<Apfel>{
2   String typ;
3
4   BirneError(String typ){
5     this.typ=typ;}
6
7   public boolean eq(Birne other){
8     return this.typ.equals(other.typ);
9   }
10 }
```

Wir bekommen folgende Fehlermeldung:

```
sep@linux:~/fh/java1.5/examples/src> javac BirneError.java
BirneError.java:1: BirneError is not abstract and does not override abstract method eq(Apfel) in EQ
class BirneError implements EQ<Apfel>{
^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

5.1.3 Kovarianz gegen Kontravarianz

Gegeben seien zwei Typen `A` und `B`. Der Typ `A` soll Untertyp des Typs `B` sein, also entweder ist `A` eine Unterklasse der Klasse `B` oder `A` implementiert die Schnittstelle `B` oder die Schnittstelle `A` erweitert die Schnittstelle `B`. Für diese Subtyprelation schreiben wir das Relationssymbol $\sqsubseteq$. Es gelte also $A \sqsubseteq B$. Gilt damit auch für einen generischen Typ `C`: `C<A>`$\sqsubseteq$`C<B>`?

Man mag geneigt sein, zu sagen ja. Probieren wir dieses einmal aus:

```
1 class Kontra{
2   public static void main(String []_){
3     Box<Object> b  = new Box<String>("hello");
4   }
5 }
```

Der Javaübersetzer weist dieses Programm zurück:

```
sep@linux:~/fh/java1.5/examples/src> javac Kontra.java
Kontra.java:4: incompatible types
found   : Box<java.lang.String>
required: Box<java.lang.Object>
    Box<Object> b  = new Box<String>("hello");
                     ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

Eine `Box<String>` ist keine `Box<Object>`. Der Grund für diese Entwurfsentscheidung liegt darin, dass bestimmte Laufzeitfehler vermieden werden sollen. Betrachtet man ein Objekt des Typs `Box<String>` über eine Referenz des Typs `Box<Object>`, dann können in dem Feld `contents` beliebige Objekte gespeichert werden. Die Referenz über den Typ `Box<String>` geht aber davon aus, dass in `contents` nur Stringobjekte gespeichert werden.

Man vergegenwärtige sich nochmals, dass Reihungen in Java sich hier anders verhalten. Bei Reihungen ist die entsprechende Zuweisung erlaubt. Eine Reihung von Stringobjekten darf einer Reihung beliebiger Objekte zugewiesen werden. Dann kann es bei der Benutzung der Reihung von Objekten zu einen Laufzeitfehler kommen.

Ko.java

```
1 class Ko{
2   public static void main(String []_){
3     String [] x = {"hello"};
4     Object [] b  = x;
5     b[0]=new Integer(42);
6     x[0].toUpperCase();
7   }
8 }
```

Das obige Programm führt zu folgendem Laufzeitfehler:

```
sep@linux:~/fh/java1.5/examples/classes> java Ko
Exception in thread "main" java.lang.ArrayStoreException
        at Ko.main(Ko.java:5)
sep@linux:~/fh/java1.5/examples/classes>
```

Für generische Typen wurde ein solcher Fehler durch die Strenge des statischen Typchecks bereits ausgeschlossen.

5.1.4 Sammlungsklassen

Die Paradeanwendung für generische Typen sind natürlich Sammlungsklassen, also die Klassen für Listen und Mengen, wie sie im Paket `java.util` definiert sind. Mit der Version 1.5 von Java finden sich generische Versionen der bekannten Sammlungsklassen. Jetzt kann man angeben, was für einen Typ die Elemente einer Sammlung genau haben sollen.

ListTest.java
```
1  import java.util.*;
2  import java.util.List;
3  class ListTest{
4    public static void main(String [] _){
5      List<String> xs = new ArrayList<String>();
6      xs.add("Schimmelpfennig");
7      xs.add("Shakespeare");
8      xs.add("Horvath");
9      xs.add("Brecht");
10     String x2 = xs.get(1);
11     System.out.println(xs);
12   }
13 }
```

Aus Kompatibilitätsgründen mit bestehendem Code können generische Klassen auch weiterhin ohne konkrete Angabe des Typparameters benutzt werden. Während der Übersetzung wird in diesen Fällen eine Warnung ausgegeben.

WarnList.java
```
1  import java.util.*;
2  import java.util.List;
3  class WarnTest{
4    public static void main(String [] _){
5      List xs = new ArrayList<String>();
6      xs.add("Schimmelpfennig");
7      xs.add("Shakespeare");
8      xs.add("Horvath");
9      xs.add("Brecht");
10     String x2 = (String)xs.get(1);
11     System.out.println(xs);
12   }
13 }
```

Obiges Programm übersetzt mit folgender Warnung:

```
sep@linux:~/fh/java1.5/examples/src> javac WarnList.java
Note: WarnList.java uses unchecked or unsafe operations.
Note: Recompile with -warnunchecked for details.
sep@linux:~/fh/java1.5/examples/src>
```

5.1.5 Generische Methoden

Bisher haben wir generische Typen für Klassen und Schnittstellen betrachtet. Generische Typen sind aber nicht an einen objektorientierten Kontext gebunden, sondern basieren ganz im Gegenteil auf dem Milner-Typsystem, das funktionale Sprachen, die nicht objektorientiert sind, benutzen. In Java verläßt man den objektorientierten Kontext in statischen Methoden. Statische Methoden sind nicht an ein Objekt gebunden. Auch statische Methoden lassen sich generisch in Java definieren. Hierzu ist vor der Methodensignatur in spitzen Klammern eine Liste der für die statische Methode benutzten Typvariablen anzugeben.

Eine sehr einfache statische generische Methode ist eine *trace*-Methode, die ein beliebiges Objekt erhält, dieses Objekt auf der Konsole ausgibt und als Ergebnis genau das erhaltene Objekt unverändert wieder zurückgibt. Diese Methode `trace` hat für alle Typen den gleichen Code und kann daher entsprechend generisch geschrieben werden:

Trace.java

```
class Trace {

  static <elementType>  elementType trace(elementType x){
    System.out.println(x);
    return x;
  }

  public static void main(String [] _){
    String x = trace ((trace ("hallo")
                      +trace( " welt")).toUpperCase());

    Integer y = trace (new Integer(40+2));
  }
}
```

In diesem Beispiel ist zu erkennen, dass der Typchecker eine kleine Typinferenz vornimmt. Bei der Anwendung der Methode `trace` ist nicht anzugeben, mit welchen Typ die Typvariable `elementType` zu instanziieren ist. Diese Information inferriert der Typchecker automatisch aus dem Typ des Arguments.

5.2 Iteration

Typischer Weise wird in einem Programm über die Elemente eines Sammlungstyp iteriert oder über alle Elemente einer Reihung. Hierzu kennt Java verschiedene Schleifenkonstrukte. Leider kannte Java bisher kein eigenes Schleifenkonstrukt, das bequem eine Iteration über die Elemente einer Sammlung ausdrücken konnte. Die Schleifensteuerung musste bisher immer explizit ausprogrammiert werden. Hierbei können Programmierfehler auftreten, die insbesondere dazu führen können, dass eine Schleife nicht terminiert. Ein Schleifenkonstrukt, das garantiert terminiert, kannte Java bisher nicht.

Beispiel 5.2.1 *In diesen Beispiel finden sich die zwei wahrscheinlich am häufigsten programmierten Schleifentypen. Einmal iterieren wir über alle Elemente einer Reihung und einmal iterieren wir mittels eines Iteratorobjekts über alle Elemente eines Sammlungsobjekts:*

OldIteration.java

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

class OldIteration{

  public static void main(String [] _){
    String [] ar
      = {"Brecht","Horvath","Shakespeare","Schimmelpfennig"};
    List xs = new ArrayList();

    for (int i= 0;i<ar.length;i++){
      final String s = ar[i];
      xs.add(s);
    }

    for (Iterator it=xs.iterator();it.hasNext();){
      final String s = (String)it.next();
      System.out.println(s.toUpperCase());
```

```
20        }
21     }
22  }
```

Die Codemenge zur Schleifensteuerung ist gewaltig und übersteigt hier sogar die eigentliche Anwendungslogik.

Mit Java 1.5 gibt es endlich eine Möglichkeit, zu sagen, mache für alle Elemente im nachfolgenden Sammlungsobjekt etwas. Eine solche Syntax ist jetzt in Java integriert. Sie hat die Form:

`for` (*Type identifier* : *expr*){*body*}

Zu lesen ist dieses Kosntrukt als: für jedes *identifier* des Typs *Type* in *expr* führe *body* aus.[1]

Beispiel 5.2.2 *Damit lassen sich jetzt die Iterationen der letzten beiden Schleifen wesentlich eleganter ausdrücken.*

NewIteration.java

```
1  import java.util.List;
2  import java.util.ArrayList;
3
4  class NewIteration{
5
6    public static void main(String [] _){
7      String [] ar
8       = {"Brecht","Horvath","Shakespeare","Schimmelpfennig"};
9      List<String> xs = new ArrayList<String>();
10
11     for (String s:ar) xs.add(s);
12     for (String s:xs) System.out.println(s.toUpperCase());
13   }
14 }
```

Der gesamte Code zur Schleifensteuerung ist entfallen. Zusätzlich ist garantiert, dass für endliche Sammlungsiteratoren auch die Schleife terminiert.

Wie man sieht, ergänzen sich generische Typen und die neue `for`-Schleife.

Beispiel 5.2.3 *Ein abschließendes kleines Beispiel für generische Sammlungsklassen und die neue* `for`*-Schleife. Die folgende Klasse stellt Methoden zur Verfügung, um einen String in eine Liste von Wörtern zu spalten und umgekehrt aus einer Liste von Wörtern wieder einen String zu bilden:*

TextUtils.java

```
1  import java.util.*;
2  import java.util.List;
3
4  class TextUtils {
5
6    static List<String> words (String s){
7      final List<String> result = new ArrayList<String>();
8
```

[1] Ein noch sprechenderes Konstrukt wäre gewesen, wenn man statt des Doppelpunkts das Schlüsselwort `in` benutzt hätte. Aus Aufwärtskompatibilitätsgründen wird jedoch darauf verzichtet, neue Schlüsselwörter in Java einzuführen.

```
9     StringBuffer currentWord = new StringBuffer();
10
11    for (char c:s.toCharArray()){
12      if (Character.isWhitespace(c)){
13        final String newWord = currentWord.toString().trim();
14        if(newWord.length()>0){
15          result.add(newWord);
16          currentWord=new StringBuffer();
17        }
18      }else{currentWord.append(c);}
19    }
20    return result;
21  }
22
23  static String unwords(List<String> xs){
24    StringBuffer result=new StringBuffer();
25    for (String x:xs) result.append(" "+x);
26    return result.toString().trim();
27  }
28
29  public static void main(String []_){
30    List<String> xs = words("  the   world  is my Oyster  ");
31
32    for (String x:xs) System.out.println(x);
33
34    System.out.println(unwords(xs));
35  }
36 }
```

Kapitel 6

Datentypen und Algorithmen

Die bisher kennengelernten Javakonstrukte bilden einen soliden Kern, der genügend programmiertechnische Mittel zur Verfügung stellt, um die gängigsten Konzepte der Informatik umzusetzen. In diesem Kapitel werden wir keine neuen Javakonstrukte kennenlernen, sondern mit den bisher bekannten Mitteln die häufigsten in der Informatik gebräuchlichen Datentypen und auf ihnen anzuwendende Algorithmen erkunden.

6.1 Listen

Eine der häufigsten Datenstrukturen in der Programmierung sind Sammlungstypen. In fast jedem nichttrivialen Programm wird es Punkte geben, an denen eine Sammlung mehrerer Daten gleichen Typs anzulegen sind. Eine der einfachsten Strukturen, um Sammlungen anzulegen, sind Listen. Da Sammlungstypen oft gebraucht werden, stellt Java entsprechende Klassen als Standardklassen zur Verfügung. Bevor wir uns aber diesen bereits vorhandenen Klassen zuwenden, wollen wir in diesem Kapitel Listen selbst spezifizieren und programmieren.

6.1.1 Formale Spezifikation

Wir werden Listen als abstrakten Datentyp formal spezifizieren. Ein abstrakter Datentyp (ADT) wird spezifiziert über eine endliche Menge von Methoden. Hierzu wird spezifiziert, auf welche Weise Daten eines ADT konstruiert werden können. Dazu werden entsprechende Konstruktormethoden spezifiziert. Dann wird eine Menge von Funktionen definiert, die wieder Teile aus den konstruierten Daten selektieren können. Schließlich werden noch Testmethoden spezifiziert, die angeben, mit welchem Konstruktor ein Datum erzeugt wurde.

Der Zusammenhang zwischen Konstruktoren und Selektoren sowie zwischen den Konstruktoren und den Testmethoden wird in Form von Gleichungen spezifiziert.

Der Trick, der angewendet wird, um abstrakte Datentypen wie Listen zu spezifizieren, ist die Rekursion. Das Hinzufügen eines weiteren Elements zu einer Liste wird dabei als das Konstruieren einer neuen Liste aus der Ursprungsliste und einem weiteren Element betrachtet. Mit dieser Betrachtungsweise haben Listen eine rekursive Struktur: eine Liste besteht aus dem zuletzt vorne angehängten neuen Element, dem sogenannten Kopf der Liste, und aus der alten Teilliste, an die dieses Element angehängt wurde, dem sogenannten Schwanz der Liste. Wie bei jeder rekursiven Struktur bedarf es eines Anfangs der Definition. Im Falle von Listen wird dieses durch die Konstruktion einer leeren Liste spezifiziert.[1]

[1] Man vergleiche es mit der Definition der natürlichen Zahlen: die 0 entspricht der leeren Liste, der Schritt von n nach $n+1$ dem Hinzufügen eines neuen Elements zu einer Liste.

Konstruktoren

Abstrakte Datentypen wie Listen lassen sich durch ihre Konstruktoren spezifizieren. Die Konstruktoren geben an, wie Daten des entsprechenden Typs konstruiert werden können. In dem Fall von Listen bedarf es nach den obigen Überlegungen zweier Konstruktoren:

- einem Konstruktor für neue Listen, die noch leer sind.
- einem Konstruktor, der aus einem Element und einer bereits bestehenden Liste eine neue Liste konstruiert, indem an die Ursprungsliste das Element angehängt wird.

Wir benutzen in der Spezifikation eine mathematische Notation der Typen von Konstruktoren.[2] Dem Namen des Konstruktors folgt dabei mit einem Doppelpunkt abgetrennt der Typ. Der Ergebnistyp wird von den Parametertypen mit einem Pfeil getrennt.

Somit lassen sich die Typen der zwei Konstruktoren für Listen wie folgt spezifizieren:

- Empty: () →**List**
- Cons: (**Object**,**List**) →**List**

Selektoren

Die Selektoren können wieder auf die einzelnen Bestandteile der Konstruktion zurückgreifen. Der Konstruktor **Cons** hat zwei Parameter. Für **Cons**-Listen werden zwei Selektoren spezifiziert, die jeweils einen dieser beiden Parameter wieder aus der Liste selektieren. Die Namen dieser beiden Selektoren sind traditioneller Weise *head* und *tail*.

- head: **(List)→Object**
- tail: **(List)→List**

Der funktionale Zusammenhang von Selektoren und Konstruktoren läßt sich durch folgende Gleichungen spezifizieren:

$$\begin{aligned} head(Cons(x, xs)) &= x \\ tail(Cons(x, xs)) &= xs \end{aligned}$$

Testmethoden

Um für Listen Algorithmen umzusetzen, ist es notwendig, unterscheiden zu können, welche Art der beiden Listen vorliegt: die leere Liste oder eine **Cons**-Liste. Hierzu bedarf es noch einer Testmethode, die mit einem bool'schen Wert als Ergebnis angibt, ob es sich bei der Eingabeliste um die leere Liste handelte oder nicht. Wir wollen diese Testmethode *isEmpty* nennen. Sie hat folgenden Typ:

- isEmpty: **List→boolean**

Das funktionale Verhalten der Testmethode läßt sich durch folgende zwei Gleichungen spezifizieren:

$$\begin{aligned} isEmpty(Empty()) &= true \\ isEmpty(Cons(x, xs)) &= false \end{aligned}$$

Somit ist alles spezifiziert, was eine Listenstruktur ausmacht. Listen können konstruiert werden, die Bestandteile einer Liste wieder einzeln selektiert und Listen können nach der Art ihrer Konstruktion unterschieden werden.

[2] Entgegen der Notation in Java, in der der Rückgabetyp kurioser Weise vor den Namen der Methode geschrieben wird.

Listenalgorithmen

Allein diese fünf Funktionen beschreiben den ADT der Listen. Wir können aufgrund dieser Spezifikation Algorithmen für Listen schreiben.

Länge Und ebenso läßt sich durch zwei Gleichungen spezifizieren, was die Länge einer Liste ist:

$$\begin{aligned} length(Empty()) &= 0 \\ length(Cons(x, xs)) &= 1 + length(xs) \end{aligned}$$

Mit Hilfe dieser Gleichungen läßt sich jetzt schrittweise die Berechnung einer Listenlänge auf Listen durchführen. Hierzu benutzen wir die Gleichungen als Ersetzungsregeln. Wenn ein Unterausdruck in der Form der linken Seite einer Gleichung gefunden wird, so kann diese durch die entsprechende rechte Seite ersetzt werden. Man spricht bei so einem Ersetzungsschritt von einem Reduktionsschritt.

Beispiel 6.1.1 *Wir errechnen in diesem Beispiel die Länge einer Liste, indem wir die obigen Gleichungen zum Reduzieren auf die Liste anwenden:*

$$\begin{aligned} & length(\mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\ \rightarrow\ & 1 + length(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + length(\mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + (1 + length(\mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + (1 + 0)) \\ \rightarrow\ & 1 + (1 + 1) \\ \rightarrow\ & 1 + 2 \\ \rightarrow\ & 3 \end{aligned}$$

Letztes Listenelement Wir können mit einfachen Gleichungen spezifizieren, was wir unter dem letzten Element einer Liste verstehen.

$$\begin{aligned} last(Cons(x, Empty())) &= x \\ last(Cons(x, xs)) &= last(xs) \end{aligned}$$

Auch die Funktion `last` können wir von Hand auf einer Beispielliste einmal per Reduktion ausprobieren:

$$\begin{aligned} & last(\mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\ \rightarrow\ & last(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & last(\mathbf{Cons}(c, \mathbf{Empty}())) \\ \rightarrow\ & c \end{aligned}$$

Listenkonkatenation Die folgenden Gleichungen spezifizieren, wie zwei Listen aneinandergehängt werden:

$$\begin{aligned} concat(Empty(), ys) &= ys \\ concat(Cons(x, xs), ys) &= Cons(x, concat(xs, ys)) \end{aligned}$$

Auch diese Funktion läßt sich beispielhaft mit der Reduktion einmal durchrechnen:

$$
\begin{aligned}
& \mathit{concat}(\mathbf{Cons}(i, \mathbf{Cons}(j, \mathbf{Empty}())), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\
\rightarrow\ & \mathbf{Cons}(i, \mathit{concat}(\mathbf{Cons}(j, \mathbf{Empty}()), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))))) \\
\rightarrow\ & \mathbf{Cons}(i, \mathbf{Cons}(j, \mathit{concat}(\mathbf{Empty}(), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))))) \\
\rightarrow\ & \mathbf{Cons}(i, \mathbf{Cons}(j, \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))))
\end{aligned}
$$

Schachtel- und Zeiger-Darstellung

Listen lassen sich auch sehr schön graphisch visualisieren. Hierzu wird jede Liste durch eine Schachtel mit zwei Feldern dargestellt. Von diesen beiden Feldern gehen Pfeile aus. Der erste Pfeil zeigt auf das erste Element der Liste, dem `head`, der zweite Pfeil zeigt auf die Schachtel, die für den Restliste steht dem *tail*. Wenn eine Liste leer ist, so gehen keine Pfeile von der Schachtel aus, die sie repräsentiert.

Die Liste *Cons(a,Cons(b,Cons(c,Empty())))* hat somit die Schachtel- und Zeigerr- Darstellung aus Abbildung 6.1.

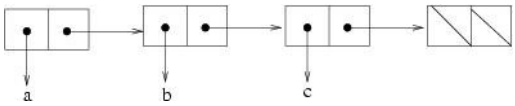

Abbildung 6.1: Schachtel Zeiger Darstellung einer dreielementigen Liste.

In der Schachtel- und Zeiger-Darstellung läßt sich sehr gut verfolgen, wie bestimmte Algorithmen auf Listen dynamisch arbeiten. Wir können die schrittweise Reduktion der Methode `concat` in der Schachtel- und Zeiger-Darstellung gut nachvollziehen:

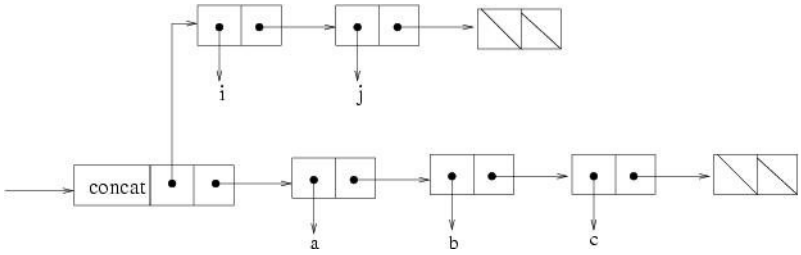

Abbildung 6.2: Schachtel Zeiger Darstellung der Funktionsanwendung von concat auf zwei Listen.

Abbildung 6.2 zeigt die Ausgangssituation. Zwei Listen sind dargestellt. Von einer Schachtel, die wir als die Schachtel der Funktionsanwendung von `concat` markiert haben, gehen zwei Zeiger aus. Der erste auf das erste Argument, der zweite auf das zweite Argument der Funktionsanwendung.

Abbildung 6.3 zeigt die Situation, nachdem die Funktion `concat` einmal reduziert wurde. Ein neuer Listenknoten wurde erzeugt. Dieser zeigt auf das erste Element der ursprünglich ersten Argumentliste. Der zweite zeigt auf den rekursiven Aufruf der Funktion `concat`, diesmal mit der Schwanzliste des ursprünglich ersten Arguments.

Abbildung 6.4 zeigt die Situation nach dem zweiten Reduktionsschritt. Ein weiterer neuer Listenknoten ist entstanden und ein neuer Knoten für den rekursiven Aufruf ist entstanden.

Abbildung 6.5 zeigt die endgültige Situation. Der letzte rekursive Aufruf von `concat` hatte als erstes Argument eine leere Liste. Deshalb wurde kein neuer Listenknoten erzeugt, sondern lediglich der Knoten für die Funktionsanwendung gelöscht. Man beachte, dass die beiden

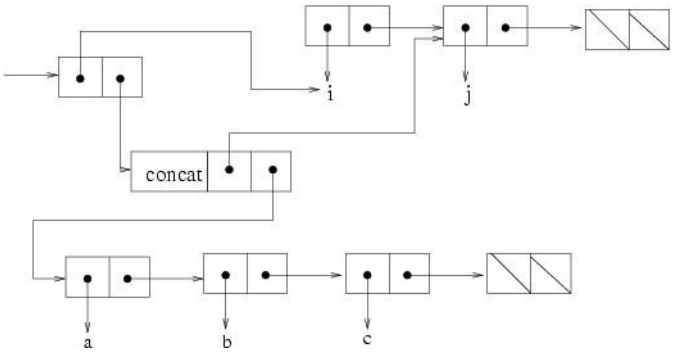

Abbildung 6.3: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

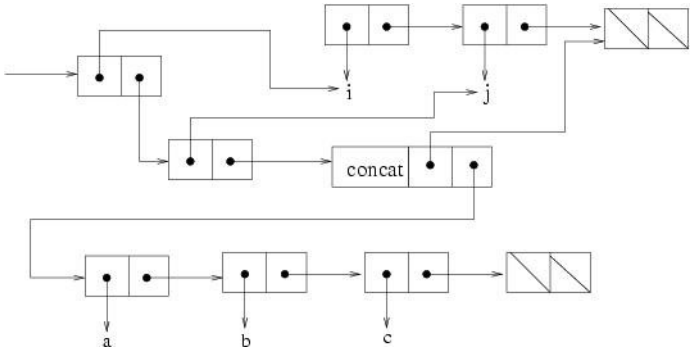

Abbildung 6.4: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

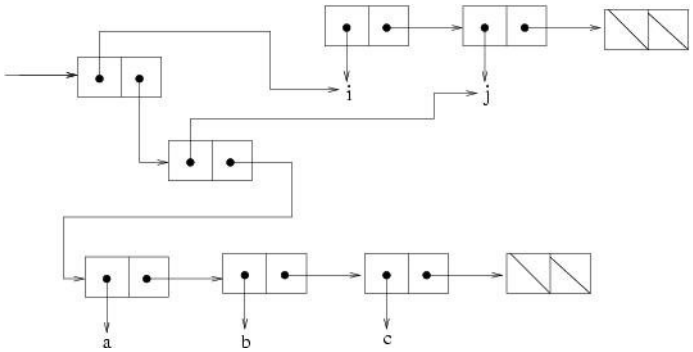

Abbildung 6.5: Schachtel Zeiger Darstellung des Ergebnisses nach der Reduktion.

ursprünglichen Listen noch vollständig erhalten sind. Sie wurden nicht gelöscht. Die erste Argumentliste wurde quasi kopiert. Die zweite Argumentliste teilen sich gewisser Maßen die neue Ergebnisliste der Funktionsanwendung und die zweite ursprüngliche Argumentliste.

6.1.2 Modellierung

Java kennt keine direkte Unterstützung für ADTs, die nach obigen Prinzip spezifiziert werden. Es gibt jedoch eine Javaerweiterung namens *Pizza*[OW97], die eine solche Unterstützung eingebaut hat. Da wir aber nicht auf *Pizza* zurückgreifen wollen, bleibt uns nichts anderes, als Listen in Java zu implementieren.

Nachdem wir im letzten Abschnitt formal spezifiziert haben, wie Listen konstruiert werden, wollen wir in diesem Abschnitt betrachten, wie diese Spezifikation geeignet mit unseren programmiersprachlichen Mitteln modelliert werden kann, d.h. wie viele und was für Klassen werden benötigt. Wir werden in den folgenden zwei Abschnitten zwei alternative Modellierungen der Listenstruktur angeben.

Modellierung als Klassenhierarchie

Laut Spezifikation gibt es zwei Arten von Listen: leere Listen und Listen mit einem Kopfelement und einer Schwanzliste. Es ist naheliegend, für diese zwei Arten von Listen je eine eigene Klasse bereitzustellen, jeweils eine Klasse für leere Listen und eine für **Cons**-Listen. Da beide Klassen zusammen einen Datentyp Liste bilden sollen, sehen wir eine gemeinsame Oberklasse dieser zwei Klassen vor. Wir erhalten die Klassenhierarchie in Abbildung: 6.6.

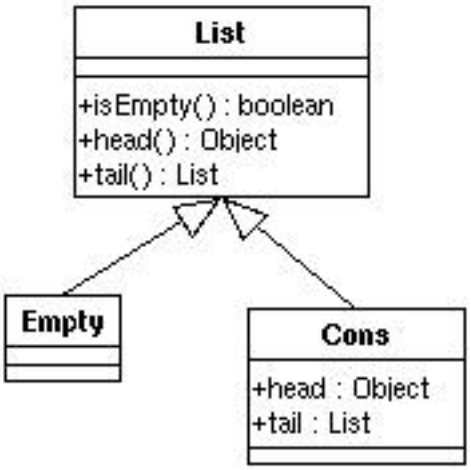

Abbildung 6.6: Modellierung von Listen mit drei Klassen.

Wir haben uns in diesem Klassendiagramm dazu entschieden, dass die Selektormethoden für alle Listen auch für leere Listen zur Verfügung stehen. Es ist in der formalen Spezifikation nicht angegeben worden, was in dem Fall der Methoden *head* und *tail* auf leere Listen als Ergebnis erwartet wird. Wir können hier Fehler geben oder aber bestimmte ausgezeichnete Werte als Ergebnis zurückgeben.

In einer Klasse

Eine alternative Modellierung der Listenstruktur besteht aus nur genau einer Klasse. Diese Klasse braucht hierzu aber zwei Konstruktoren. Wie in vielen objektorientierten Sprachen ist es in Java auch möglich, für eine Klasse mehrere Konstruktoren mit unterschiedlichen Parametertypen zu schreiben. Wir können also eine Klasse `List` modellieren, die zwei Konstruktoren hat: einen mit keinem Parameter und einen mit zwei Parametern.

6.1.3 Codierung

Die obigen beiden Modellierungen der Listen lassen sich jetzt direkt in Javacode umsetzen.

Implementierung als Klassenhierarchie

Die Oberklasse List Die zusammenfassende Oberklasse der Listenstruktur `List` stellt für die Listenmethoden `head`, `tail` und `isEmpty` jeweils eine prototypische Implementierung zur Verfügung. Die Methode `isEmpty` setzen wir in unserer Umsetzung standardmäßig auf `false`.

List.java

```
1 package name.panitz.data.list;
2 class List<a>{
3  boolean isEmpty(){return false;}
4  a head(){return null;}
5  List<a> tail(){return null;}
```

Die Methoden `tail` und `head` können nur für nichtleere Listen Objekte zurückgeben. Daher haben wir uns für die prototypischen Implementierung in der Klasse `List` dazu entschieden, den Wert `null` zurückzugeben. `null` steht in Java für das

Fehlen eines Objektes. Der Versuch, auf Felder und Methoden von `null` zuzugreifen, führt in Java zu einem Fehler.

Für die Klasse `List` schreiben wir keinen Konstruktor. Wir wollen diese Klasse nie direkt instanziieren, d.h. nie einen Konstruktor für die Klasse `List` aufrufen.[3]

Die Klasse Empty Die Klasse, die die leere Liste darstellt, ist relativ leicht abzuleiten. Wir stellen einen Konstruktor zur Verfügung und überschreiben die Methode `isEmpty`.

Empty.java

```
1 package name.panitz.data.list;
2 class Empty<a> extends List<a> {
3   Empty(){}
4   boolean isEmpty(){return true;}
5 }
```

Die Klasse Cons Schließlich ist noch die Klasse `Cons` zu codieren. Diese Klasse benötigt nach unserer Modellierung zwei Felder, in denen Kopf und Schwanz der Liste abgespeichert werden können. Die Methoden `head` und `tail` werden so überschrieben, dass sie entsprechend den Wert eines dieser Felder zurückgeben. Der Konstruktor initialisiert diese beiden Felder.

Cons.java

```
1 package name.panitz.data.list;
2 class Cons<a> extends List<a>{
3
4   a hd;
5   List<a> tl ;
6
7   Cons(a x,List<a> xs){
8     hd = x;
9     tl = xs;
10  }
11
12  a head(){
13    return hd;
14  }
15
16  List<a> tail(){
```

[3] In diesem Fall generiert Java automatisch einen Konstruktor ohne Argumente, doch dazu mehr an anderer Stelle.

```
17      return tl;
18    }
```

Damit ist die Listenstruktur gemäß unserer formalen Spezifikation vollständig implementiert.

Einfache Tests Für Algorithmen auf Listen werden wir nur die zwei Konstruktoren, die zwei Selektoren und die Testmethode `isEmpty` benutzen. Folgendes kleine Testprogramm konstruiert eine Liste mit drei Elementen des Typs `String`. Anschließend folgen einige Tests für die Selektormethoden:

TestFirstList.java

```
1  package name.panitz.data.list;
2  class TestFirstList {
3    public static void main(String [] args){
4
5      //Konstruktion einer Testliste
6      List xs = new Cons<String>("friends",
7                new Cons<String>("romans",
8                new Cons<String>("countrymen",
9                new Empty<String>())));
10
11     //Zugriffe auf einzelne Elemente
12     System.out.println("1. Element: "+xs.head());
13     System.out.println("2. Element: "+xs.tail().head());
14     System.out.println("3. Element: "+xs.tail().tail().head());
15
16     //Test für die Methode isEmpty()
17     if (xs.tail().tail().tail().isEmpty()){
18       System.out.println("leere Liste nach drittem Element.");
19     }
20
21     //Ausgabe eines null Wertes
22     System.out.println(xs.tail().tail().tail().head());
23   }
24 }
```

Tatsächlich bekommen wir für unsere Tests die erwartete Ausgabe:

```
sep@linux:~/fh/prog1/examples/classes> java name.panitz.data.list.TestFirstList
1. Element: friends
2. Element: romans
3. Element: countrymen
leere Liste nach drittem Element.
null
sep@linux:~/fh/prog1/examples/classes>
```

Implementierung als eine Klasse

In der zweiten Modellierung haben wir auf eine Klassenhierarchie verzichtet. Was wir in der ersten Modellierung durch die verschiedenen Klassen mit verschieden überschriebenen Methoden ausgedrückt haben, muss in der Modellierung in einer Klasse über ein bool'sches Feld vom Typ `boolean` ausgedrückt werden. Die beiden unterschiedlichen Konstruktoren setzen ein bool'sches Feld `empty`, um für das neu konstruierte Objekt zu markieren, mit welchem Konstruktor es konstruiert wurde. Die Konstruktoren setzen entsprechend die Felder, so dass die Selektoren diese nur auszugeben brauchen.

Li.java

```
1  package name.panitz.data.list;
2  class Li<a> implements Iterable<a>{
3    boolean empty = true;
4    a  hd;
5    Li<a> tl;
6
7    Li(){}
8
9    Li(a x,Li<a> xs){
10    hd = x;
11    tl = xs;
12    empty = false;
13   }
14
15   boolean isEmpty() {return empty;}
16   a  head(){return hd;}
17   Li<a>      tail(){return tl;}
18
19   public java.util.Iterator<a> iterator(){
20     final Li<a>  itout = this;
21     return new java.util.Iterator<a>(){
22       Li<a>  it = itout;
23       public boolean hasNext(){return !it.isEmpty();}
24       public a next(){a result=it.head();it=it.tail();return result;}
25       public void remove(){throw new UnsupportedOperationException();}
26     };
27   }
```

Zum Testen dieser Listenimplementierung sind nur die Konstruktoraufrufe bei der Listenkonstruktion zu ändern. Da wir nur noch eine Klasse haben, gibt es keine zwei Konstruktoren mit unterschiedlichen Namen, sondern beide haben denselben Namen:

TestFirstLi.java

```
1
2  package name.panitz.data.list;
3  class TestFirstLi {
4    public static void main(String [] args){
5
6      //Konstruktion einer Testliste
7      Li<String> xs = new Li<String>("friends",
8                      new Li<String>("romans",
9                      new Li<String>("countrymen",
10                     new Li<String>())));
11
12     //Zugriffe auf einzelne Elemente
13     System.out.println("1. Element: "+xs.head());
14     System.out.println("2. Element: "+xs.tail().head());
15     System.out.println("3. Element: "+xs.tail().tail().head());
16
17     //Test für die Methode isEmpty()
18     if (xs.tail().tail().tail().isEmpty()){
19       System.out.println("leere Liste nach drittem Element.");
20     }
21
22     //Ausgabe eines null Wertes
23     System.out.println(xs.tail().tail().tail().head());
24   }
```

```
25
26 }
```

Wie man aber sieht, ändert sich an der Benutzung von Listen nichts im Vergleich zu der Modellierung mittels einer Klassenhierarchie. Die Ausgabe ist ein und dieselbe für beide Implementierungen:

```
sep@linux:~/fh/prog1/examples/classes> java TestFirstLi
1. Element: friends
2. Element: romans
3. Element: countrymen
leere Liste nach drittem Element.
null
sep@linux:~/fh/prog1/examples/classes>
```

Konstruktion langer Listen Die Konstruktion der dreielementigen Liste oben war schon recht umständlich. Jede Teilliste musste explizit durch einen Konstruktoraufruf generiert werden. Die variable Argumentanzahl für Methoden erlaubt es, einen Konstruktor zu definieren, der die einzelnen Elemente der zu erzeugenen Liste als Argumente erhält.

Li.java

```
27   public Li(a... xs){
28     this(0,xs);
29   }
30
31   private Li(int i,a... xs){
32     if (i<xs.length) {
33       empty=false;
34       hd=xs[i];
35       tl = new Li<a>(i+1,xs);
36     }
37   }
```

Jetzt lassen sich bequem auch längere Listen durch einen einzigen Konstruktoraufruf erzeugen:

TestFirstLi2.java

```
1
2  package name.panitz.data.list;
3  class TestFirstLi2 {
4    public static void main(String [] args){
5
6      //Konstruktion einer Testliste
7      Li<String> xs
8       = new Li<String>
9            ("Friends", "Romans", "countrymen", "lend", "me","your", "ears"
10           ,"I","come","to","bury","Caesar","not","to","praise","him");
11
12     //Zugriffe auf einzelne Elemente
13     System.out.println("1. Element: "+xs.head());
14     System.out.println("2. Element: "+xs.tail().head());
15     System.out.println("3. Element: "+xs.tail().tail().head());
16
17     //Test für die Methode isEmpty()
18     if (!xs.tail().tail().tail().isEmpty()){
19       System.out.println("keine leere Liste nach drittem Element.");
20     }
21
22     //Ausgabe 4. Wertes
23     System.out.println(xs.tail().tail().tail().head());
```

```
24    }
25
26  }
```

6.1.4 Methoden für Listen

Mit der formalen Spezifikation und schließlich Implementierung von Listen haben wir eine Abstraktionsebene eingeführt. Listen sind für uns Objekte, für die wir genau die zwei Konstruktormethoden, zwei Selektormethoden und eine Testmethode zur Verfügung haben. Unter Benutzung dieser fünf Eigenschaften der Listenklasse können wir jetzt beliebige Algorithmen auf Listen definieren und umsetzen.

Länge

Eine interessante Frage bezüglich Listen ist die nach ihrer Länge. Wir können die Länge einer Liste berechnen, indem wir durchzählen, aus wievielen **Cons**-Listen eine Liste besteht. Wir haben bereits die Funktion *length* durch folgende zwei Gleichungen spezifiziert.

$$\begin{aligned} length(Empty()) &= 0 \\ length(Cons(x, xs)) &= 1 + length(xs) \end{aligned}$$

Diese beiden Gleichungen lassen sich direkt in Javacode umsetzen. Die zwei Gleichungen ergeben genau zwei durch eine `if`-Bedingung zu unterscheidende Fälle in der Implementierung. Wir können die Klassen `List` und `Li` um folgende Methode `length` ergänzen:

Li.java

```
1   int length(){
2     if (isEmpty())return 0;
3     return 1+tail().length();
4   }
```

In den beiden Testklassen läßt sich ausprobieren, ob die Methode `length` entsprechend der Spezifikation funktioniert. Wir können in der `main`-Methode einen Befehl einfügen, der die Methode `length` aufruft:

TestLength.java

```
1  package name.panitz.data.list;
2  class TestLength {
3    static  Li<String> XS
4     =  new Li<String>("friends"
5       ,new Li<String>("romans"
6       ,new Li<String>("countrymen"
7       ,new Li<String>())));
8
9    public static void main(String [] args){
10     System.out.println(XS.length());
11   }
12 }
```

Und wie wir bereits schon durch Reduktion dieser Methode von Hand ausgerechnet haben, errechnet die Methode `length` tatsächlich die Elementanzahl der Liste:

```
sep@linux:~/fh/prog1/examples/classes> java TestLength
3
sep@linux:~/fh/prog1/examples/classes>
```

Alternative Implementierung im Fall der Klassenhierarchie Im Fall der Klassenhierarchie können wir auch die `if`-Bedingung der Methode `length` dadurch ausdrücken, dass die beiden Fälle sich in den unterschiedlichen Klassen für die beiden unterschiedlichen Listenarten befinden. In der Klasse `List` kann folgende prototypische Implementierung eingefügt werden:

List.java
```
1   int length(){return 0;}
2 }
```

In der Klasse `Cons`, die ja Listen mit einer Länge größer 0 darstellt, ist dann die Methode entsprechend zu überschreiben:

Cons.java
```
1   int length(){return 1+tail().length();}
2 }
```

Auch diese Längenimplementierung können wir testen:

TestListLength.java
```
1 package name.panitz.data.list;
2 class TestListLength {
3   static  List<String> XS
4     = new Cons<String>("friends"
5      ,new Cons<String>("romans"
6      ,new Cons<String>("countrymen"
7      ,new Empty<String>())));
8
9   public static void main(String [] args){
10    System.out.println(XS.length());
11  }
12 }
```

An diesem Beispiel ist gut zu sehen, wie durch die Aufsplittung in verschiedene Unterklassen beim Schreiben von Methoden `if`-Abfragen verhindert werden können. Die verschiedenen Fälle einer `if`-Abfrage finden sich dann in den unterschiedlichen Klassen realisiert. Der Algorithmus ist in diesem Fall auf verschiedene Klassen aufgeteilt.

Iterative Lösung Die beiden obigen Implementierungen der Methode `length` sind rekursiv. Im Rumpf der Methode wurde sie selbst gerade wieder aufgerufen. Natürlich kann die Methode mit den entsprechenden zusammengesetzten Schleifenbefehlen in Java auch iterativ gelöst werden.

Wir wollen zunächst versuchen, die Methode mit Hilfe einer `while`-Schleife zu realisieren. Der Gedanke ist naheliegend. Solange es sich noch nicht um die leere Liste handelt, wird ein Zähler, der die Elemente zählt, hochgezählt:

Li.java
```
1   int lengthWhile(){
2     int erg=0;
3     Li<a> xs = this;
4     while (!xs.isEmpty()){
5       erg= erg +1;
6       xs = xs.tail();
7     }
8     return erg;
9   }
```

Schaut man sich diese Lösung genauer an, so sieht man, dass sie die klassischen drei Bestandteile einer `for`-Schleife enthält:

- die Initialisierung einer Laufvariablen: `List xs = this;`

- eine bool'sche Bedingung zur Schleifensteuerung: `!xs.isEmpty()`
- eine Weiterschaltung der Schleifenvariablen: `xs = xs.tail();`

Damit ist die Schleife, die über die Elemente einer Liste iteriert, ein guter Kandidat für eine `for`-Schleife. Wir können das Programm entsprechend umschreiben:

Li.java
```
1  int lengthFor(){
2    int erg=0;
3    for (Li<a> xs=this;!xs.isEmpty();xs=xs.tail()){
4      erg = erg +1;
5    }
6    return erg;
7  }
```

Eine Schleifenvariable ist also nicht unbedingt eine Zahl, die hoch oder herunter gezählt wird, sondern kann auch ein Objekt sein, von dessen Eigenschaften abhängt, ob die Schleife ein weiteres Mal zu durchlaufen ist.

In solchen Fällen ist die Variante mit der `for`-Schleife der Variante mit der `while`-Schleife vorzuziehen, weil somit die Befehle, die die Schleife steuern, gebündelt zu Beginn der Schleife stehen.

toString

Eine sinnvolle Methode `toString` für Listen erzeugt einen String, in dem die Listenelemente durch Kommas getrennt sind.

Li.java
```
1  public String toString(){
2    return "("+toStringAux()+")";
3  }
4
5  private String toStringAux(){
6    if (isEmpty()) return "";
7    else if (tail().isEmpty()) return head().toString();
8    else return head().toString()+","+tail().toStringAux();
9  }
```

Aufgabe 37 Nehmen Sie beide der in diesem Kapitel entwickelten Umsetzungen von Listen und fügen Sie ihrer Listenklassen folgende Methoden hinzu. Führen Sie Tests für diese Methoden durch.

a) `a last()`: gibt das letzte Element der Liste aus.

b) `List<a> concat(List<a> other)` bzw.: `Li<a> concat(Li<a> other)`: erzeugt eine neue Liste, die erst die Elemente der `this`-Liste und dann der `other`-Liste hat, es sollen also zwei Listen aneinander gehängt werden.

c) `a elementAt(int i)`: gibt das Element an einer bestimmten Indexstelle der Liste zurück. Spezifikation:

$$\begin{aligned} elementAt(Cons(x, xs), 1) &= x \\ elementAt(Cons(x, xs), n+1) &= elementAt(xs, n) \end{aligned}$$

6.1.5 Sortierung

Eine sehr häufig benötigte Eigenschaft von Listen ist, sie nach einer bestimmten Größenrelation sortieren zu können. Wir wollen in diesem Kapitel Objekte des Typs `String` sortieren. Über die Methode `compareTo` der Klasse `String` läßt sich eine kleiner-gleich-Relation auf Zeichenketten definieren:

StringOrdering.java
```
1 package name.panitz.data.list;
2 class StringOrdering{
3   static boolean lessEqual(String x,String y){
4     return x.compareTo(y)<=0;
5   }
6 }
```

Diese statische Methode werden wir zum Sortieren benutzen.

In den folgenden Abschnitte werden wir drei verschiedene Verfahren der Sortierung kennenlernen.

Sortieren durch Einfügen

Die einfachste Methode einer Sortierung ist, neue Elemente in einer Liste immer so einzufügen, dass nach dem Einfügen eine sortierte Liste entsteht. Wir definieren also zunächst eine Klasse, die es erlaubt, Elemente so einzufügen, dass alle vorhergehenden Elemente kleiner und alle nachfolgenden Elemente größer sind. Diese Klasse braucht unsere entsprechenden Konstruktoren und einen neuen Selektor, der den spezialisierteren Rückgabetyp hat:

SortStringLi.java
```
1 package name.panitz.data.list;
2 class SortStringLi extends Li<String> {
3
4   SortStringLi(){super();}
5   SortStringLi(String x,SortStringLi xs){super(x,xs);}
6
7   SortStringLi sortTail(){return (SortStringLi)tail();}
```

Die entscheidende neue Methode für diese Klasse ist die Einfügemethode `insertSorted`. Sie erzeugt eine neue Liste, in die das neue Element eingefügt wird:

SortStringLi.java
```
8   SortStringLi insertSorted(String x){
```

Im Falle einer leeren Liste wird die einelementige Liste zurückgegeben:

SortStringLi.java
```
9     if (isEmpty()) {return new SortStringLi(x,this);}
```

Andernfalls wird unterschieden, ob das einzufügende Element kleiner als das erste Listenelement ist. Ist das der Fall, so wird das neue Element in die Schwanzliste sortiert eingefügt:

SortStringLi.java
```
10    else if (StringOrdering.lessEqual((String)head(),x)){
11      return new SortStringLi
12                  ((String)head()
13                  ,((SortStringLi)tail()).insertSorted(x));
```

Anderfalls wird das neue Element mit dem Konstruktor vorne eingefügt:

SortStringLi.java
```
14    } else return new SortStringLi(x,this);
15  }//method insertSorted
```

Die eigentliche Sortiermethode erzeugt eine leere Ergebnisliste, in die nacheinander die Listenelemente sortiert eingefügt werden:

SortStringLi.java

```
16   SortStringLi getSorted(){
17     SortStringLi result = new SortStringLi();
18
19     for (Li<String> xs= this;!xs.isEmpty();xs = xs.tail()){
20       result = result.insertSorted(xs.head());
21     }
22
23     return result;
24   }//method sort
```

Somit hat die Klasse `SortStringLi` eine Sortiermethode, die wir in einer Hauptmethode testen können:

SortStringLi.java

```
25   public static void main(String [] args){
26     SortStringLi xs
27       = new SortStringLi("zz"
28        ,new SortStringLi("ab"
29        ,new SortStringLi("aaa"
30        ,new SortStringLi("aaa"
31        ,new SortStringLi("aaz"
32        ,new SortStringLi("aya"
33        ,new SortStringLi()))))));
34
35
36     System.out.println("Die unsortierte Liste:");
37     System.out.println(xs);
38
39     Li<String> ys = xs.getSorted();
40     System.out.println("Die sortierte Liste:");
41     System.out.println(ys);
42   }
43 }//class SortStringLi
```

Die Ausgabe unseres Testprogramms zeigt, dass tatsächlich die Liste sortiert wird:

```
sep@swe10:~/fh/prog1/Listen> java SortStringLi
Die unsortierte Liste:
(zz,ab,aaa,aaa,aaz,aya)
Die sortierte Liste:
(aaa,aaa,aaz,ab,aya,zz)
sep@swe10:~/fh/prog1/Listen>
```

Quick Sort

Der Namen *quick sort* hat sich für eine Sortiermethode durchgesetzt, die sich das Prinzip des Teilens des Problems zu eigen macht, bis die durch Teilen erhaltenen Subprobleme trivial zu lösen sind.[4]

[4] Der Name *quick sort* ist insofern nicht immer berechtigt, weil in bestimmten Fällen das Verfahren nicht sehr schnell im Vergleich zu anderen Verfahren ist.

formale Spezifikation Mathematisch läßt sich das Verfahren wie durch folgende Gleichungen beschreiben:

$$\begin{aligned} quicksort(Empty()) &= Empty() \\ quicksort(Cons(x, xs)) &= quicksort(\{y | y \in xs, y <= x\}) \\ &\quad ++Cons(x, quicksort(\{y | y \in xs, y > x\})) \end{aligned}$$

Die erste Gleichung spezifiziert, dass das Ergebnis der Sortierung einer leeren Liste eine leere Liste zum Ergebnis hat.

Die zweite Gleichung spezifiziert den Algorithmus für nichtleere Listen. Der in der Gleichung benutzte Operator `++` steht für die Konkatenation zweier Listen mit der in der letzten Aufgabe geschriebenen Methode `concat`.

Die Gleichung ist zu lesen als:

> Um eine nichtleere Liste zu sortieren, filtere alle Elemente aus der Schwanzliste, die kleiner sind als der Kopf der Liste. Sortiere diese Teilliste. Mache dasselbe mit der Teilliste aus den Elementen des Schwanzes, die größer als das Kopfelement sind. Hänge schließlich diese beiden sortierten Teillisten aneinander und das Kopfelement dazwischen.

Modellierung Anders als in unserer obigen Sortierung durch Einfügen in eine neue Liste, für die wir eine Unterklasse der Klasse `Li` geschrieben haben, wollen wir die Methode `quicksort` in der Klasse `Li` direkt implementieren, d.h. allgemein für alle Listen zur Verfügung stellen.

Aus der Spezifikation geht hervor, dass wir als zentrales Hilfsmittel eine Methode brauchen, die nach einer bestimmten Bedingung Elemente aus einer Liste filtert. Wenn wir diesen Mechanismus haben, so ist der Rest des Algorithmus mit Hilfe der Methode `concat` trivial direkt aus der Spezifikation ableitbar. Um die Methode `filter` möglichst allgemein zu halten, können wir sie so schreiben, dass sie ein Objekt bekommt, in dem eine Methode die Bedingung, nach der zu filtern ist, angibt. Eine solche Klasse sieht allgemein wie folgt aus:

FilterCondition.java

```
1 package name.panitz.data.list;
2 class FilterCondition<a> {
3   boolean condition(a testMe){
4     return true;
5   }
6 }
```

Für bestimmte Bedingungen können für eine solche Klasse Unterklassen definiert werden, die die Methode `condition` entsprechend überschreiben. Für unsere Sortierung brauchen wir zwei Bedingungen: einmal wird ein Objekt getestet, ob es größer ist als ein vorgegebenes Objekt, ein anderes Mal, ob es kleiner ist. Wir erhalten also folgende kleine Klassenhierarchie aus Abbildung 6.7:

Die beiden Unterklassen brauchen jeweils ein Feld, in dem das Objekt gespeichert ist, mit dem das Element im Größenvergleich getestet wird.

Die für den Sortieralgorithmus benötigte Methode `filter` kann entsprechend ein solches Objekt also Argument bekommen:

```
1 Li<a> filter(Condition<a> cond);
```

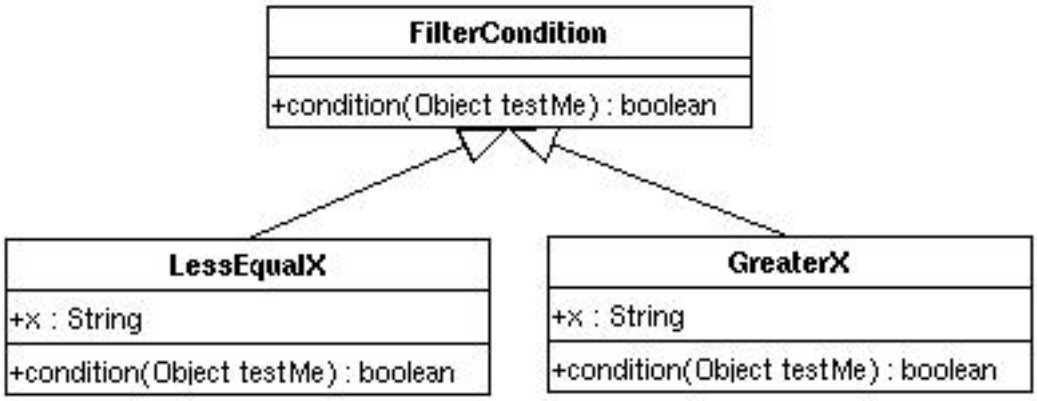

Abbildung 6.7: Modellierung der Filterbedingungen.

Codierung Die entscheidende Methode für den Sortieralgorithmus ist `filter`. Mit dieser Methode werden entsprechend einer Filterbedingung bestimmte Elemente aus einer Liste selektiert.

In der Klasse `Li` kann nun die Methode `filter` eingefügt werden:

Li.java

```
1   Li<a> filter(FilterCondition<a> cond){
2     Li<a> result = new Li<a>();
3
4     //test all elements of this list
5     for (Li<a> xs=this;!xs.isEmpty();xs=xs.tail()){
6
7       //in case that the condition is true for the element
8       if (cond.condition(xs.head())) {
9         //then add it to the result
10        result = new Li<a>(xs.head(),result);
11      }
12    }
13    return result;
14  }
```

Hiermit ist die Hauptarbeit für den *quick sort*-Algorithmus getan.

Beispiel 6.1.2 *Bevor wir die Methode* `quicksort` *implementieren, wollen wir ein paar Tests für unsere Methode* `filter` *schreiben. Hierzu schreiben wir Klassen für die Bedingungen, nach denen wir Elemente aus einer Liste filtern wollen:*

Zunächst eine Bedingung, die Stringobjekte mit einer Länge größer als 10 selektiert:

LongString.java

```
1  package name.panitz.data.list;
2  class LongString extends FilterCondition<String>{
3    boolean condition(String testMe){
4      return testMe.length()>10;
5    }
6  }
```

Eine weitere Bedingung soll testen, ob ein Stringobjekt mit einem Großbuchstaben 'A' beginnt:

StringStartsWithA.java
```
1 package name.panitz.data.list;
2 class StringStartsWithA extends FilterCondition<String>{
3   boolean condition(String testMe){
4     return testMe.charAt(0)=='A';
5   }
6 }
```

Und eine dritte Bedingung, die wahr wird für Stringobjekte, die kein großes 'A' enthalten:

ContainsNoA.java
```
1 package name.panitz.data.list;
2 class ContainsNoA extends FilterCondition<String>{
3   boolean condition(String testMe){
4     for (int i= 0;i<testMe.length();i=i+1){
5       final char c = testMe.charAt(i);
6
7       if (c=='A' || c=='a') return false;
8     }
9     return true;
10  }
11 }
```

Probeweise filtern wir jetzt einmal eine Liste nach diesen drei Bedingungen:

TestFilter.java
```
1 package name.panitz.data.list;
2 class TestFilter {
3
4   static
5      Li<String> XS = new Li<String>("Shakespeare",
6              new Li<String>("Brecht",
7              new Li<String>("Achternbusch",
8              new Li<String>("Calderon",
9              new Li<String>("Moliere",
10             new Li<String>("Sorokin",
11             new Li<String>("Schimmelpfennig",
12             new Li<String>("Kane",
13             new Li<String>("Wilde",
14             new Li<String>()))))))))));
15
16  public static void main(String [] _){
17     System.out.println(XS);
18     System.out.println(XS.filter(new ContainsNoA()));
19     System.out.println(XS.filter(new StringStartsWithA()));
20     System.out.println(XS.filter(new LongString()));
21  }
22 }
```

In der Ausgabe können wir uns vom korrekten Lauf der Methode `filter` *überzeugen:*

```
sep@linux:~/fh/prog1/examples/classes> java TestFilter
(Shakespeare,Brecht,Achternbusch,Calderon,Moliere,Sorokin,Schimmelpfennig,Kane,Wilde)
(Wilde,Schimmelpfennig,Sorokin,Moliere,Brecht)
(Achternbusch)
(Schimmelpfennig,Achternbusch,Shakespeare)
sep@linux:~/fh/prog1/examples/classes>
```

Interessant zu beobachten mag sein, dass unsere Methode `filter` *die Reihenfolge der Elemente der Liste umdreht.*

Zurück zu unserer eigentlichen Aufgabe, dem *quick sort*-Verfahren. Hier wollen wir die Eingabeliste einmal nach allen Elementen, die kleiner als das erste Element sind, filtern; und einmal nach allen Elementen, die größer als dieses sind. Hierzu brauchen wir zwei Filterbedingungen. Diese hängen beide von einem bestimmten Element, nämlich dem ersten Element der Liste, ab. Die beiden Klassen für die Bedingung lassen sich relativ einfach aus der Modellierung ableiten:

LessEqualX.java

```
1  package name.panitz.data.list;
2  class LessEqualX<a> extends FilterCondition<a>{
3    Comparable<a> x;
4
5    LessEqualX(Comparable<a> x){
6      this.x=x;
7    }
8
9    boolean condition(a testMe){
10     return x.compareTo(testMe)>0;
11   }
12 }
```

Entsprechend für die größer-Relation:

GreaterX.java

```
1  package name.panitz.data.list;
2  class GreaterX<a> extends FilterCondition<a> {
3    Comparable<a> x;
4
5    GreaterX(Comparable<a> x){
6      this.x=x;
7    }
8
9    boolean condition(a testMe){
10     return x.compareTo(testMe)<=0;
11   }
12 }
```

Die Methode `quicksort` läßt sich direkt aus der formalen Spezifikation ableiten:

Li.java

```
1    static public <a extends Comparable<a>> Li<a> quicksort(Li<a> xs){
2      Li<a> result = new Li<a>();
3      if (!xs.isEmpty()){
4        result
5         = //filter the smaller elements out of the tail
6           //and sort these
7           quicksort(xs.tail().filter(new LessEqualX<a>(xs.head())))
8            //concatenate it with the sorted
9            //sublist of greater elements
10           .append(new Li<a>(xs.head()
11                  ,quicksort(xs.tail().filter(new GreaterX<a>(xs.head()))))
12             ));
13     }
14     return result;
15   }
```

Obige Umsetzung des *quick sort*-Algorithmus ist allgemeiner als der zuvor entwickelte Algorithmus zur Sortierung durch Einfügen. Die entscheidende Methode `filter` ist parameterisiert über die Bedingung, nach der gefiltert werden soll. Damit läßt sich schnell eine *quick sort*-Methode schreiben, deren Filter nicht auf der größer-Relation von `String` Objekten basiert. Hierzu sind

nur entsprechende Unterklassen der Klasse `FilterCondition` zu schreiben und in der Sortiermethode zu benutzen. Wieder einmal haben wir unsere strenge Trennung aus der anfänglichen Arbeitshypothese durchbrochen: Die Objekte der Klasse `FilterCondition` stellen nicht primär Daten dar, sondern eine Methode, die wir als Argument einer anderen Methode (der Methode `filter`) übergeben.

Sortieren für beliebige Relationen Es ist naheliegend, die Parameterisierung über die eigentliche Ordnungsrelation der Sortiermethode mitzugeben, also eine Sortiermethode zu schreiben, die einen Parameter hat, der angibt, nach welchem Kriterium zu sortieren ist:

```
1 Li<a> sortBy(Relation rel)
```

Hierzu brauchen wir eine Klasse `Relation`, die eine Methode hat, in der entschieden wird, ob zwei Objekte in einer Relation stehen:

Relation.java

```
1 package name.panitz.data.list;
2 class Relation<a,b>{
3   boolean lessEqual(a x,b y){
4     return true;
5   }
6 }
```

Je nachdem, was wir sortieren wollen, können wir eine Subklasse der Klasse `Relation` definieren, die uns sagt, wann zwei Objekte in der kleiner-Relation stehen. Für Objekte des Typs `String` bietet folgende Klasse eine adäquate Umsetzung:

StringLessEqual.java

```
1 package name.panitz.data.list;
2 class StringLessEqual extends Relation<String,String> {
3   boolean lessEqual(String x,String y){
4     return x.compareTo(y)<=0;
5   }
6 }
```

Um den *quick sort*-Algorithmus anzuwenden, benötigen wir nun noch eine Möglichkeit, aus einer Relation die beiden Bedingungen für die Methode `filter` generieren. Wir schreiben zwei neue Subklassen der Klasse `FilterCondition`, die für eine Relation jeweils kleinere bzw. größere Objekte als ein vorgegebenes Objekt filtern.

OrderingCondition.java

```
1  package name.panitz.data.list;
2  class OrderingCondition<a> extends FilterCondition<a>{
3    a x;
4    Relation<a,a> rel;
5
6    OrderingCondition(a x,Relation<a,a> rel){
7      this.x=x;
8      this.rel = rel;
9    }
10
11   boolean condition(a y){
12     return rel.lessEqual(y,x);
13   }
14 }
```

Entsprechend für die negierte Relation:

NegativeOrderingCondition.java

```
package name.panitz.data.list;
class NegativeOrderingCondition<a>  extends FilterCondition<a>{
  a x;
  Relation<a,a> rel;

  NegativeOrderingCondition(a x,Relation<a,a> rel){
    this.x=x;
    this.rel = rel;
  }

  boolean condition(a y){
    return !rel.lessEqual(y,x);
  }
}
```

Damit haben wir alle Bausteine zur Hand, mit denen ein über die Ordungsrelation parameterisierte Sortiermethode geschrieben werden kann:

Li.java

```
  Li<a> sortBy(Relation<a,a>  rel){
    Li<a> result = new Li<a>();
    if (!isEmpty()){
      FilterCondition<a> le
       = new OrderingCondition<a>(head(),rel);
      FilterCondition <a> gr
       = new NegativeOrderingCondition<a>(head(),rel);
      result = tail()
                .filter(le)
                .sortBy(rel)
                .append(new Li<a>(head()
                                  ,tail()
                                  .filter(gr)
                                  .sortBy(rel)));
    }
    return result;
  }
}
```

Beim Aufruf der Methode `sortBy` ist ein Objekt mitzugeben, das die Relation angibt, nach der sortiert werden soll.

Beispiel 6.1.3 *Im folgenden Beispiel werden Strings einmal nach ihrer lexikographischen Ordnung, einmal nach ihrer Länge sortiert:*

StringLengthLessEqual.java

```
package name.panitz.data.list;
class StringLengthLessEqual extends Relation<String,String> {
  boolean lessEqual(String x,String y){
    return x.length()<= y.length();
  }
}
```

In der Testmethode können wir die Methode `sortBy` *jeweils mit einer der Bedingungen aufrufen:*

TestSortBy.java

```
package name.panitz.data.list;
class TestSortBy{
  public static void main(String [] args){
    Li<String> xs = TestFilter.XS;
```

```
5
6       System.out.println("Die unsortierte Liste:");
7       System.out.println(xs);
8       System.out.println("Die alphabetisch sortierte Liste:");
9       System.out.println(xs.sortBy(new StringLessEqual()));
10      System.out.println("Die nach der Länge sortierte Liste:");
11      System.out.println(xs.sortBy(new StringLengthLessEqual()));
12    }
13  }
```

Und tatsächlich können wir jetzt die Sortiermethode benutzen, um nach unterschiedlichen Kriterien zu sortieren:

```
sep@linux:~/fh/prog1/examples/classes> java TestSortBy
Die unsortierte Liste:
(Shakespeare,Brecht,Achternbusch,Calderon,Moliere,Sorokin,Schimmelpfennig,Kane,Wilde)
Die alphabetisch sortierte Liste:
(Achternbusch,Brecht,Calderon,Kane,Moliere,Schimmelpfennig,Shakespeare,Sorokin,Wilde)
Die nach der Länge sortierte Liste:
(Kane,Wilde,Brecht,Moliere,Sorokin,Calderon,Shakespeare,Achternbusch,Schimmelpfennig)
sep@linux:~/fh/prog1/examples/classes>
```

Aufgabe 38 Verfolgen Sie schrittweise mit Papier und Beistift, wie der *quicksort* Algorithmus die folgenden zwei Listen sortiert:

- `("a","b","c","d","e")`
- `("c","a","b","d","e")`

Aufgabe 39 Diese Aufgabe soll mir helfen, Listen für Ihre Leistungsbewertung zu erzeugen.

a) Implementieren Sie für Ihre Listenklasse eine Methode `String toHtmlTable()`, die für Listen Html-Code für eine Tabelle erzeugt, z.B:

```
1 <table>
2   <tr>erstes Listenelement</tr>
3   <tr>zweites Listenelement</tr>
4   <tr>drittes Listenelement</tr>
5 </table>
```

b) Nehmen Sie die Klasse `Student`, die Felder für Namen, Vornamen und Matrikelnummer hat. Implementieren Sie für diese Klasse eine Methode `String toTableRow()`, die für Studenten eine Zeile einer Html-Tabelle erzeugt:

```
1 Student s1 = new Student("Müller","Hans",167857);
2 System.out.println(s1.toTableRow());
```

soll folgende Ausgabe ergeben:

```
1 <td>Müller</td><td>Hans</td><td>167857</td>
```

Ändern Sie die Methode `toString` so, dass sie dasselbe Ergebnis wie die neue Methode `toTableRow` hat.

c) Legen Sie eine Liste von Studenten an, sortieren Sie diese mit Hilfe der Methode `sortBy`nach Nachnamen und Vornamen und erzeugen Sie eine Html-Seite, die die sortierte Liste anzeigt.

Sie können zum Testen die folgende Klasse benutzen:

HtmlView.java

```
1  package name.panitz.data.list;
2  import java.awt.*;
3  import java.awt.event.*;
4  import javax.swing.*;
5  import javax.swing.plaf.basic.*;
6  import javax.swing.text.*;
7  import javax.swing.text.html.*;
8
9  public class HtmlView extends JPanel {
10
11   //example invocation
12   public static void main(String s[]) {
13     HtmlView view = new HtmlView();
14     view.run();
15     view.setText("<h1>hallo</h1>");
16   }
17
18   JFrame frame;
19   JTextPane ausgabe = new JTextPane();
20
21   public HtmlView() {
22     ausgabe.setEditorKit(new HTMLEditorKit());
23     add(ausgabe);
24   }
25
26   void setText(String htmlString){
27     ausgabe.setText(htmlString);
28     frame.pack();
29     ausgabe.repaint();
30   }
31
32   void run(){
33     frame = new JFrame("HtmlView");
34     frame.getContentPane().add(this);
35     frame.pack();
36     frame.setVisible(true);
37   }
38 }
```

6.1.6 Formale Beweise über Listenalgorithmen

Im ersten Kapitel haben wir unter den Disziplinen der Programmierung auch die formale Verifikation aufgezählt. Die formale Verifikation erlaubt es, Eigenschaften von Programmen allgemein mathematisch zu beweisen. Anders als durch Testfälle, die nur Aussagen über ein Programm für ausgewählte Fälle machen können, können über die Verifikation allgemeine Aussagen bewiesen werden, die sich auf alle möglichen Argumente für ein Programm beziehen können. Formale Verifikation ist seit Jahrzehnten ein weites Feld der Forschung, die zumeist im Gebiet der KI (künstlichen Intelligenz) angesiedelt ist.

Voraussetzung für eine formale Verifikation ist, dass sowohl die Datentypen als auch die programmierten Algorithmen in einer formalen Weise spezifiziert und notiert wurden. Für unsere Listentypen haben wir das bisher getan. Unsere Listen sind daher bestens geeignet, um formale Beweise zu führen.

Vollständige Induktion

Das aus der Mathematik bekannte Verfahren der vollständigen Induktion über die natürlichen Zahlen ist ein gängiges Verfahren zur Verifikation von Algorithmen. Ein Induktionsbeweis geht dabei in zwei Schritten. Zunächst wird im sogenannten Induktionsanfang die Aussage für das kleinste Datum geführt, bei natürlichen Zahlen also für die Zahl 0. Im zweiten Schritt, dem sogenannten Induktionsschritt, wird angenommen, dass die Aussage bereits für alle Werte kleiner eines bestimmten Wertes n bewiesen wurde. Dann wird versucht, unter dieser Annahme die Aussage auch für n zu beweisen. Sind beide Beweisschritte gelungen, so ist die Aussage für alle endlichen Werte bewiesen.

Induktion über Listen

Das Beweisverfahren der Induktion läßt sich auf rekursiv definierte Datentypen, so wie unsere Listen, anwenden. Der Basisfall, also entsprechend der 0 bei den natürlichen Zahlen, sind die Daten, die durch einen nicht rekursiven Konstruktor erzeugt werden. Für Listen entsprechend sind dieses die Listen, die mit dem Konstruktor für leere Listen erzeugt wurden. Im Induktionsschritt wird angenommen, die zu beweisende Aussage sei bereits für Daten, die mit weniger Konstruktoren erzeugt wurden, bewiesen. Unter dieser Annahme wird versucht zu zeigen, dass die Aussage auch für mit einem weiteren Konstruktor erzeugte Daten gilt. Für Listen bedeutet das, man versucht, die Annahme für die Liste der Form *Cons(x,xs)* zu beweisen unter der Annahme, dass die Aussage für die Liste *xs* bereits bewiesen wurde.

Beispiel 6.1.4 *Als Beispiel wollen wir eine Eigenschaft über Listen im Zusammenhang mit den Funktionen* concat *und* length *beweisen. Wir wollen beweisen, dass die Länge der Konkatenation zweier Listen gleich der Summe der Längen der beiden Listen ist, also dass für alle Listen* xs *und* ys *gilt:*

$$length(concat(xs, ys)) \quad = \quad length(xs) + length(ys)$$

Dabei seien die beiden Funktionen wieder spezifiziert als:

$$\begin{aligned} length(Empty()) &= 0 \\ length(Cons(x, xs)) &= 1 + length(xs) \end{aligned}$$

und

$$\begin{aligned} concat(Empty(), ys) &= ys \\ concat(Cons(x, xs), ys) &= Cons(x, concat(xs, ys)) \end{aligned}$$

Wir werden die Aussage beweisen mit einer Induktion über das erste Argument der Funktion concat*, also dem* xs*:*

- **Induktionsanfang:** *Wir versuchen, die Aussage zu beweisen mit* xs=Empty()*. Wir erhalten die folgende Aussage:*

  $$length(concat(Empty(), ys)) \quad = \quad length(Empty()) + length(ys)$$

  Wir können jetzt auf beiden Seiten der Gleichung durch Reduktion mit den Gleichungen aus der Spezifikation die Gleichung vereinfachen:

  $$\begin{aligned} length(concat(Empty(), ys)) &= length(Empty()) + length(ys) \\ length(ys) &= length(Empty()) + length(ys) \\ length(ys) &= 0 + length(ys) \\ length(ys) &= length(ys) \end{aligned}$$

Wir haben die Aussage auf eine Tautologie reduziert. Für xs *als leere Liste ist damit unsere Aussage bereits bewiesen.*

- **Induktionsschritt:** *Jetzt wollen wir die Aussage für* xs=Cons(x',xs') *beweisen, wobei wir als Induktionsvoraussetzung annehmen, dass sie für* xs' *bereits wahr ist, dass also gilt:*

$$length(concat(xs', ys)) = length(xs') + length(ys)$$

Hierzu stellen wir die zu beweisende Gleichug auf und reduzieren sie auf beiden Seiten:

$$\begin{aligned} length(concat(Cons(x', xs'), ys)) &= length(Cons(x', xs')) + length(ys) \\ length(Cons(x', concat(xs', ys))) &= length(Cons(x', xs')) + length(ys) \\ 1 + length(concat(xs', ys)) &= 1 + length(xs') + length(ys) \\ length(concat(xs', ys)) &= length(xs') + length(ys) \end{aligned}$$

Die letzte Gleichung ist gerade die Induktionsvoraussetzung, von der wir angenommen haben, dass diese bereits wahr ist. Wir haben unsere Aussage für alle endlichen Listen bewiesen.

Aufgabe 40 Gegeben sei folgende Schnittstelle:

List.java

```
1  package name.panitz.util;
2
3  public interface List<A> {
4
5    A getElement();
6    List<A> getNext();
7
8    boolean isEmpty();
9    void add(A a);
10
11   int size();
12   boolean contains(A a);
13   A get(int i);
14   A last();
15   List<A> reverse();
16 }
```

Schreiben Sie eine Klasse mit dem vollqualifizierten Namen
`name.panitz.util.solution.LinkedList`,
die die obige Schnittstelle implementiert. Die Klasse soll zusätzlich einen öffentlichen Standardkonstruktor haben der eine leere Liste erzeugt.

Die Methoden sind im einzelnen wie folgt zu implementieren:

a) `getElement`: gibt das vorderste Element, das in der Liste gespeichert ist zurück. Ist die Liste leer, so wird `null` zurück gegeben.

b) `getNext`: gibt die Restliste ohne das ersten Element der Liste, zurück. Ist die Liste leer, so wird `null` zurück gegeben.

c) `isEmpty`: gibt an, ob es sich um eine leere Liste handelt.

d) `add`: modifiziert die Liste, indem es vorne an die Liste ein neues Element einfügt.

e) `size`: gibt die Anzahl der Elemente in der Liste zurück.

f) `contains`: gibt an, ob das übergebene Objekt in der Liste enthalten ist. Es wird für den Test die `equals`-Methode verwendet.

g) `get`: gibt das Element an dem als Parameter übergebenen Index zurück. Das erste Element habe den Index 0. Für negativen oder für zu großen Index wird `null` zurück gegeben.

h) `last`: gibt das letzte in der Liste gespeicherte Element zurück.

i) `reverse`: erzeugt eine vollkommen neue Liste, die aus den Elementen dieser Liste in umgekehrter Reihenfolge besteht.

Auf der Webseite `http://swtsrv01.cs.hs-rm.de/ksteb001` existiert ein allererster Prototyp einer Webapplikation, auf der Lösungen zu Aufgaben hochgeladen werden können. Die Webapplikation lässt dann JUnit-Tests über die hochgeladene Lösung laufen und zeigt das Ergebnis dieser Tests an.

Aufgabe 41 Schreiben Sie eine Klasse `CompoundObject`, die eine Unterklasse von `GeometricObject` darstellt. Ein `CompoundObject` soll eine Liste weiterer Geometrische Objekte enthalten, die zusammen ein größeres Objekt darstellen sollen.

Achten Sie darauf, dass die *Bounding-Box* entsprechend korrekt gesetzt ist.

Sie sollen dabei Ihre eigene Listenimplementierung aus der vorherigen Aufgabe benutzen und nicht auf eine der Standardlisten zurückgreifen.

Testen Sie ein paar Objekt der Klasse `CompoundObject` in Ihrer kleinen GUI-Anwendung aus dem letzten Aufgabenblatt.

Anhang A

Javawerkzeuge

A.1 Klassenpfad und Java-Archive

Bisher haben wir Javaklassen einzeln übersetzt und jeweils pro Klasse eine Datei `.class` erhalten. Ein Javaprogramm als solches läßt sich damit gar nicht genau eingrenzen. Ein Javaprogramm ist eine Klasse mit einer Hauptmethode zusammen mit der Menge aller Klassen, die von der Hauptmethode zur Ausführung benötigt werden. Selbst Klassen, die zwar im gleichen Paket wie die Hauptklasse mit der Hauptmethode liegen, werden nicht unbedingt vom Programm benötigt und daher auch nicht geladen. Ein Programm liegt also in vielen verschiedenen Dateien verteilt. Das kann unhandlich werden, wenn wir unser Programm anderen Benutzern bereitstellen wollen. Wir wollen wohl kaum mehrere hundert Dateien auf einen Server legen, die ein potentieller Kunde dann herunterlädt.

Aus diesem Grunde bietet Java eine Möglichkeit an, die Klassen eines Programms oder einer Bibliothek zusammen in einer Datei zu bündeln. Hierzu gibt es `.jar`-Dateien. `Jar` steht für *Java archive*. Die Struktur und Benutzung der `.jar`-Dateien leitet sich von den seit alters her in Unix bekannten `.tar`-Dateien ab, wobei `tar` für *tape archive* steht und ursprünglich dazu gedacht war, Dateien gemeinsam auf einen Tonband-Datenträger abzuspeichern.

In einem Javaarchiv können nicht nur Klassendateien gespeichert werden, sondern auch Bilder und Sounddateien, die das Programm benötigt, und zusätzlich Versionsinformationen.

A.1.1 Benutzung von jar-Dateien

`jar` ist zunächst einmal ein Programm zum Verpacken von Dateien und Ordnern in eine gemeinsame Datei, ähnlich wie es auch das Programm `zip` macht. Das Programm `jar` wird in einer Javainstallation mitgeliefert und kann von der Kommandozeile gestartet werden. Ein Aufruf ohne Argumente führt zu einer Hilfemeldung:

```
sep@swe10:~/fh/prog2> jar
Syntax: jar {ctxu}[vfm0Mi] [JAR-Datei] [Manifest-Datei] [-C dir] Dateien ...
Optionen:
    -c  neues Archiv erstellen
    -t  Inhaltsverzeichnis für Archiv auflisten
    -x  benannte (oder alle) Dateien aus dem Archiv extrahieren
    -u  vorhandenes Archiv aktualisieren
    -v  ausführliche Ausgabe für Standardausgabe generieren
    -f  Namen der Archivdatei angeben
    -m  Manifestinformationen aus angegebener Manifest-Datei einbeziehen
    -0  nur speichern; keine ZIP-Komprimierung verwenden
    -M  keine Manifest-Datei für die Einträge erstellen
    -i  Indexinformationen für die angegebenen JAR-Dateien generieren
    -C  ins angegebene Verzeichnis wechseln und folgende Datei einbeziehen
Falls eine Datei ein Verzeichnis ist, wird sie rekursiv verarbeitet.
Der Name der Manifest-Datei und der Name der Archivdatei müssen
```

```
in der gleichen Reihenfolge wie die Flags ''m'' und ''f'' angegeben werden.

Beispiel 1: Archivieren von zwei Klassendateien in einem Archiv
mit dem Namen classes.jar:
       jar cvf classes.jar Foo.class Bar.class
Beispiel 2: Verwenden der vorhandenen Manifest-Datei ''meinmanifest''
und Archivieren aller
           Dateien im Verzeichnis foo/ in ''classes.jar'':
       jar cvfm classes.jar meinmanifest -C foo/ .

sep@swe10:~/fh/prog2>
```

Wie man sieht, gibt es eine Reihe von Optionen, um mit `jar` die Dateien zu erzeugen, ihren Inhalt anzuzeigen oder sie wieder auszupacken.

Erzeugen von Jar-Dateien

Das Kommando zum Erzeugen einer Jar-Datei hat folgende Form:

```
jar cf jar-file input-file(s)
```

Die Befehlsoptionen im einzelnen:

- Das `c` steht dafür, das eine Jar-Datei erzeugt werden soll.
- Das `f` steht dafür, dass der Dateiname der zu erzeugenden Jar-Datei folgt.
- `jar-file` ist der Name, den die zu erzeugende Datei haben soll. Hier nimmt man üblicherweise die Erweiterung `.jar` für den Dateinamen.
- `input-file(s)` ist eine Liste beliebiger Dateien und Ordner. Hier kann auch die aus der Kommandozeile bekannte `*`-Notation benutzt werden.

Beispiel A.1.1 *Der Befehl*

```
jar cf myProg.jar *.class
```

verpackt alle Dateien mit der Erweiterung `.class` *in eine* `Jar`*-Datei namens* `myProg.jar`*.*

Anzeige des Inhalts einer Jar-Datei

Das Kommando zur Anzeige der in einer Jar-Datei gespeicherten Dateien hat folgende Form:

```
jar tf jar-file
```

Die Befehlsoptionen im einzelnen:

- Das `t` steht dafür, das eine Auflistung der enthaltenen Dateien angezeigt werden[1] soll.
- Das `f` steht dafür, dass der Dateiname der benötigten Jar-Datei folgt.
- `jar-file` ist der Name der existierenden Jar-Datei.

Extrahieren des Inhalts der Jar-Datei

Der Befehl zum Extrahieren des Inhalts einer Jar-Datei hat folgende schematische Form:

```
jar xf jar-file [archived-file(s)]
```

Die Befehlsoptionen im einzelnen:

[1] `t` steht dabei für *table*.

- Die `x`-Option gibt an, dass Dateien aus einer Jar-Datei extrahiert werden sollen.
- Das `f` gibt wieder an, dass der Name einer JAR-Datei folgt.
- `jar-file` ist der Name einer Jar-Datei, aus der die entsprechenden Dateien zu extrahieren sind.
- Optional kann noch eine Liste von Dateinamen folgen, die angibt, welche Dateien extrahiert werden sollen. Wird diese Liste weggelassen, so werden alle Dateien extrahiert.

Bei der Extraktion der Dateien werden die Dateien in dem aktuellen Verzeichnis der Kommandozeile gespeichert. Hierbei wird die originale Ordnerstruktur der Dateien verwendet, sprich Unterordner, wie sie in der Jar-Datei verpackt wurden, werden auch als solche Unterordner wieder ausgepackt.

Ändern einer Jar Datei

Schließlich gibt es eine Option, die es erlaubt, eine bestehende Jar-Datei in ihrem Inhalt zu verändern. Der entsprechende Befehl hat folgendes Format:

```
jar uf jar-file input-file(s)
```

Die Befehlsoptionen im einzelnen:

- Die Option `u` gibt an, dass eine bestehende Jar-Datei geändert werden soll.
- Das `f` gibt wie üblich an, dass der Name einer Jar-Datei folgt.
- Eine Liste von Dateinamen gibt an, dass diese zur Jar-Datei hinzuzufügen sind.

Dateien, die bereits in der Jar-Datei enthalten sind, werden durch diesen Befehl mit der neuen Version überschrieben.

A.1.2 Der Klassenpfad

Jar-Dateien sind nicht nur dazu gedacht, dass damit Javaanwendungen einfacher ausgetauscht werden können, sondern der Javainterpreter kann Klassen direkt aus der Jar-Datei starten. Eine Jar-Datei braucht also nicht ausgepackt zu werden, um eine Klasse darin auszuführen. Es ist dem Javainterpreter lediglich mitzuteilen, dass er auch in einer Jar-Datei nach den entsprechenden Klassen suchen soll.

Angabe des Klassenpfades als Option

Der Javainterpreter hat eine Option, mit der ihm angegeben werden kann, wo er die Klassen suchen soll, die er zur Ausführung der Anwendung benötigt. Die entsprechende Option des Javainterpreters heißt `-cp` oder auch `-classpath`, wie die Hilfmeldung des Javainterpreters auch angibt:

```
sep@swe10:~/fh/prog2> java -help -cp
Usage: java [-options] class [args...]
           (to execute a class)
   or  java -jar [-options] jarfile [args...]
           (to execute a jar file)

where options include:

    -cp -classpath <directories and zip/jar files separated by :>
                set search path for application classes and resources
```

Es läßt sich nicht nur eine Jar-Datei für den Klassenpfad angeben, sondern mehrere, und darüber hinaus auch Ordner im Dateisystem, in denen sich die Paketstruktur der Javaanwendung bis hin zu deren Klassendateien befindet. Die verschiedenen Einträge des Klassenpfades werden bei der Suche einer Klasse von vorne nach hinten benutzt, bis die Klasse im Dateisystem oder in einer Jar-Datei gefunden wurde. Die verschiedenen Einträge des Klassenpfades werden durch einen Doppelpunkt getrennt.

Angabe des Klassenpfades als Umgebungsvariable

Implizit existiert immer ein Klassenpfad in Ihrem Betriebssystem als eine sogenannte Umgebungsvariable. Sie können den Wert dieser Umgebungsvariable abfragen und ändern. Die Umgebungsvariable, die Java benutzt, um den Klassenpfad zu speichern, heißt `CLASSPATH`. Deren Wert benutzt Java, wenn kein Klassenpfad per Option angegeben wird.

Windows und Unix unterscheiden sich leicht in der Benutzung von Umgebungsvariablen. In Unix wird der Wert einer Umgebungsvariable durch ein vorangestelltes Dollarzeichen bezeichnet, also `$CLASSPATH`, in Windows wird sie durch Prozentzeichen eingeschlossen also, `%CLASSPATH%`.

Abfrage des Klassenpfades In der Kommandozeile kann man sich über den aktuellen Wert einer Umgebungsvariablen informieren.

Unix In Unix geschieht dieses leicht mit Hilfe des Befehls `echo`, dem die Variable in der Dollarnotation folgt:

```
sep@swe10:~/fh/prog2> echo $CLASSPATH
.:/home/sep/jarfiles/log4j-1.2.8.jar:/home/sep/jarfiles:
sep@swe10:~/fh/prog2>
```

Windows In Windows hingegen benutzt man den Konsolenbefehl `set`, dem der Name der Umgebungsvariablen folgt.

```
set CLASSPATH
```

Setzen des Klassenpfades Innerhalb einer Eingabeaufforderung kann für diese Eingabeaufforderung der Wert einer Umgebungsvariablen geändert werden. Auch hierin unterscheiden sich Unix und Windows marginal:

Unix

Beispiel A.1.2 *Wir fügen dem Klassenpfad eine weitere Jar-Datei an ihrem Beginn an:*

```
sep@swe10:~/fh/prog2> export CLASSPATH=~/jarfiles/jugs.jar:$CLASSPATH
sep@swe10:~/fh/prog2> echo $CLASSPATH
/home/sep/jarfiles/jugs.jar:.:/home/sep/jarfiles/log4j-1.2.8.jar:/home/sep/jarfiles:
```

Windows In Windows werden Umgebungsvariablen auch mit dem `set`-Befehl geändert. Hierbei folgt dem Umgebungsvariablennamen mit einem Gleichheitszeichen getrennt der neue Wert.

Beispiel A.1.3 *Der entsprechende Befehl des letzten Beispiels ist in Windows:*
`set CLASSPATH=~\jarfiles\jugs.jar:%CLASSPATH%`

Anhang B

Fragen und Antworten als Lernhilfe

- **Ist Java eine kompilierte oder interpretierte Sprache?**

  Im Prinzip beides: Ein Compiler übersetzt den Quelltext in .class-Dateien, die Code für eine virtuelle Maschine enthalten. Ein Interpreter ist in der Lage diesen Code auszuführen.

- **Wie ist ein Javaprogramm strukturiert?**

  Pro Datei eine Klasse (oder Schnittstelle oder Aufzählung), die den Namen der Datei trägt.

- **Gibt es einen Präprozessor?**

  Nein.

- **Gibt es einen Linker?**

  Nein, aber die virtuelle Maschine hat einen Klassenlader, der den Code der einzelnen Klassen bei Bedarf lädt.

- **Welche Namenskonventionen gibt es?**

  Klassennamen beginnen mit einem Großbuchstaben, globale Konstanten sind komplett in Großbuchstaben geschrieben. Pakete, Parameter, Variablen beginnen mit einem Kleinbuchstaben. Bei Bezeichnern aus mehreren Wörter fängt das nächste Wort immer mit einem Großbuchstaben an.

- **Ist Java statisch getypt?**

  Ja, der Compiler überprüft, ob das Programm korrekt getypt ist.

- **Gibt es auch dynamische Typüberprüfung zur Laufzeit?**

  Ja, zum einem verifiziert die virtuelle Maschine noch einmal den Code, zum anderen bewirkt eine Typzusicherung (cast) eine Laufzeittypüberprüfung.

- **Kann ich sicherstellen, dass die Typzusicherung während der Laufzeit nicht fehlschlägt.**

  Ja, indem vor der Typzusicherung der Test mit `instanceof` gemacht wird.

- **Wie schreibt man Unterprogramme oder Funktionen?**

  Immer innerhalb einer Klasse. Sie werden als Methoden bezeichnet.

- **Was sind die Eigenschaften einer Klasse?**

  Zum einen die Felder (auch als Attribute) bezeichnet, in denen Referenzen auf Objekte abgelegt werden können. Zum anderen die Methoden und Konstruktoren.

- **Was ist das `this`-Objekt?**

  Damit wird das Objekt bezeichnet, in dem sich eine Eigenschaft befindet.

- **Was ist Vererbung?**

  Jede Klasse hat genau eine Oberklasse, die in der `extends`-Klausel angegeben wird. Objekte können alle Eigenschaften die in ihrer Oberklasse zur Verfügung stehen benutzen.

- **Was ist wenn ich keine Oberklasse angebe?**

  Dann ist automatisch die Klasse `Object` die Oberklasse.

- **Was sind Konstruktoren?**

  Konstruktoren sind der Code einer Klasse, der beim Erzeugen von neuen Objekten ausgeführt wird und in der Regel die Felder des Objektes initialisiert.

- **Wie werden Konstruktoren definiert?**

  Ähnlich wie Methoden. Sie haben den Namen der Klasse und keinen Rückgabetyp.

- **Wie werden Konstuktoren aufgerufen?**

  Durch das Schlüsselwort `new` gefolgt von dem Klassennamen.

- **Hat jede Klasse einen Konstruktor?**

  Ja.

- **Und wenn ich keinen Konstruktor für meine Klasse schreibe?**

  Dann fügt Java einen leeren Konstruktor ohne Parameter ein.

- **Was ist der Unterschied zwischen statischen und nicht statischen Eigenschaften einer Klasse?**

  Eine statische Eigenschaft ist nicht an spezielle Objekte gebunden. Sie hat daher auch kein `this`-Objekt. Eine statische Eigenschaft existiert nur einmal für alle Objekte einer Klasse. Nicht-statische Methoden werden auch als Objektmethoden bezeichnet.

- **Gibt es auch statische Konstruktoren?**

  Ja, pro Klasse genau einen, der keine Parameter hat.

- **Kann ich den statischen Konstruktor auch selbst definieren?**

  Ja, mit dem Schlüssenwort `static` gefolgt von in geschweiften Klammern eingeschlossenen Code.

- **Was bedeutet Überschreiben von Methoden?**

  Methoden, die es in der Oberklasse bereits gibt in einer Unterklasse neu zu definieren.

- **Kann ich auch Konstruktoren überschreiben?**

  Nein.

- **Was bezeichnet man als Polymorphie?**

  Wenn eine Methode in verschiedenen Unterklassen einer Klasse überschrieben wird.

- **Kann ich in einer überschreibenen Methode, die überschriebene Methode aufrufen?**

  Ja, indem man das Schlüsselwort `super` benutzt und mit einem Punkt abgetrennt den eigentlichen Methodenaufruf folgen läßt.

- **Kann ich Konstruktoren der Oberklasse aufrufen.**

  Ja, aber nur im Konstruktor als erste Anweisung. Hier muss sogar der Aufruf eines Konstruktors der Oberklasse stehen. Dieser Aufruf wird durch das Schlüsselwort `super` gefolgt von der Parameterliste gemacht.

- **Was ist, wenn ich im Konstruktor keinen Aufruf an einen Konstruktor der Oberklasse schreiben.**

  Dann generiert Java den Aufruf eines Konstruktors der Oberklasse ohne Parameter als erste Anweisung in den Konstruktor. Sollte so ein parameterloser Konstruktor nicht existieren, dann gibt es allerdings einen Folgefehler.

- **Was ist späte Bindung (late binding)?**

  Beim Aufruf von Objektmethoden wird immer der Methodencode ausgeführt, der in der Klasse implementiert wurde, von der das Objekt, auf dem diese Methode aufgerufen wurde, erzeugt wurde. Es wird also immer die überschreibende Version einer Methode benutzt.

- **Was sind virtuelle Methoden und gibt es sie in Java?**

  In C++ werden Methoden, für die späte Bindung gewünscht ist als virtuell markiert. In Java funktioniert die späte Bindung für alle Objektmethoden.

- **Funktioniert späte Bindung auch für Felder?**

  Nein.

- **Funktioniert späte Bindung auch für statische Methoden?**

  Nein.

- **Funktioniert späte Bindung auch für Konstruktoren?**

  Nein.

- **Was sind überladene Methoden?**

  Methoden gleichen Namens in einer Klasse, die sich in Typ/Anzahl der Parameter unterscheiden.

- **Können auch Konstruktoren überladen werden?**

  Ja.

- **Gibt es das Prinzip von später Bindung auch für die verschiedenen überladenen Versionen einer Methode?**

  Nein! Die Auflösung, welche der überladenen Versionen einer Methode ausgeführt wird, wird bereits statisch vom Compiler vorgenommen und nicht dynamisch während der laufzeit.

- **Was sind abstrakte Klassen?**

  Klassen, die als `abstract` deklariert sind. Nur abstrakte Klassen können abstrakte Eigenschaften enthalten.

- **Und was sind abstrakte Eigenschaften?**

  Das sind Methoden, die keinen Methodenrumpf haben.

- **Können abstrakte Klassen Konstruktoren haben?**

  Ja, allerdings können von abstrakten Klassen keine Objekte mit `new` erzeugt werden.

- **Wie kann ich dann Objekte einer abstrakten Klassen erzeugen und wozu haben die dann Konstruktoren?**

  Indem Objekte einer nicht abstrakten Unterklasse mit `new` erzeugt werden. Im Konstrukor der Unterklasse wird ein Konstruktor der abstrakten Oberklasse aufgerufen.

- **Kann eine Klasse mehrere abstrakte Oberklassen haben?**

  Nein, auch abstrakte Klassen sind Klassen und es gilt die Regel: jede Klasse hat genau eine Oberklasse.

- **Kann ich in einer abstrakten Klasse abstrakte Methoden aufrufen?**

  Ja, im Rumpf einer nicht-abstrakten Methode können bereits abstrakte Klassen aufgerufen werden.

- **Was sind Schnittstellen?**

  Man kann Schnittstellen als abstrakte Klassen ansehen, in denen jeder Methode abstrakt ist.

- **Warum gibt es dann Schnittstellen?**

  Schnittstellen gelten nicht als Klassen. Eine Klasse kann nur eine Oberklasse haben, aber zusätzlich mehrere Schnittstellen implementieren.

- **Wie wird deklariert, dass eine Klasse eine Schnitstelle implementiert?**

  Durch die `implements`-Klausel in der Klassendeklaration. In ihr können mehrere Komma getrennte Schnittstellen angegeben werden.

- **Kann eine Schnittstelle eine Oberklasse haben?**

  Nein.

- **Kann eine Schnittstelle weitere Oberschnittstellen haben.**

  Ja, diese werden in der `extends`-Klausel angegeben.

- **Muss eine Klasse die eine Schnittstelle implementiert alle Methoden der Schnittstelle implementieren?**

  Im Prinzip ja, jedoch nicht, wenn die Klasse selbst abstrakt ist.

- **Haben Schnittstellen auch Objektfelder?**

  Nein! Nur Methoden und statische Felder.

- **Gibt es Zeiger?**

  Explizit nicht. Implizit wird außer bei Werten primitiver Typen immer mit Referenzen auf Objekte gearbeitet.

- **Sind alle Daten in Java Objekte?**

  Nein, es gibt 8 primitive Typen, deren Daten keine Objekte sind. Es gibt aber zu jeden dieser 8 primitiven Typen eine Klasse, die die Daten entsprechend als Objekt speichern.

- **Welches sind die primitiven Typen?**

  byte, short, int, long, double, float, char, boolean

- **Sind Strings Objekte?**

  Ja, die Klasse String ist eine Klasse wie Du und ich.

- **Sind Reihungen (arrays) Objekte?**

  Ja, und sie haben sogar ein Attribut, das ihre Länge speichert.

- **Was sind generische Typen?**

  Klassen oder Schnittstellen, in denen ein oder mehrere Typen variabel gehalten sind.

- **Wie erzeugt man Objekte einer generischen Klasse?**

  Indem beim Konstruktoraufruf in spitzen Klammern konkrete Typen für die Typvariablen angegeben werden.

- **Was passiert, wenn ich für generische Typen die spitzen Klammern bei der Benutzung weglasse?**

  Dann gibt der Compiler eine Warnung und nimmt den allgemeinsten Typ für die Typvariablen an. Meistens ist das der Typ `Object`.

- **Was sind generische Methoden?**

  Methoden in denen ein oder mehrere Parametertypen variabel gehalten sind.

- **Muss ich bei der Benutzung von generischen Methoden die konkreten Typen für die Typvariablen mit angeben?**

  Anders als beim Konstruktoraufruf für generische Klassen nicht.

- **Was passiert, wenn ich sie nicht angebe?**

  Dann inferriert der Typchecker die konkreten Typen, sofern das möglich ist.

- **Was sind typische Beispiel für generische Typen?**

  Alle Sammlungsklassen und Abbildungsklassen im Paket `java.util`, zB die Klassen `ArrayList`,`LinkedList`, `HashSet`, `Vector` oder die Schnittstellen `List`, `Map`, `Set`.

- **Wo wir gerade dabei sind. Was sollte ich bei der Klasse `Vector` beachten?**

  In 90% der Fälle ist ein Objekt der Klasse `ArrayList` einem Objekt der Klasse `Vector` vorzuziehen. `Vector` ist eine sehr alte Klasse. Ihre Objekte sind synchronisiert, die anderen Sammlungsklassen nicht, es lassen sich von den anderen Sammlungsklassen allerdings synchronisierte Kopien machen. Am besten `Vector` gar nie benutzen.

- **Was ist automatisches Boxing und Unboxing?**

  Die Konvertierung von Daten primitiver Typen in Objekte der korrespondierenden Klassen und umgekehrt wird automatisch vorgenommen.

- **Welche zusammengesetzten Befehle gibt es?**

  if, while, for, switch

- **Was hat es mit der besonderen for-Schleife auf sich?**

  Es handelt sich um eine sogenannte for-each-Schleife. Syntaktisch trennt hier ein Doppelpunkt die lokale Schleifenvariable das Sammlungsobjekt.

- **Für welche Objekte kann die for-each-Schleife benutzt werden?**

  Für alle Objekte, die die Schnittstelle `Iterable` implementieren und für Reihungen.

- **Bedeutet das, ich kann die for-each Schleife auch für meine Klassen benutzen?**

  Ja, genau, man muss nur die Schnittstelle `Iterable` hierzu implementieren.

- **Können Operatoren überladen oder neu definiert werden?**

  Nein.

- **Was sind Ausnahme?**

  Objekte von Unterklassen der Klasse `Exception`. Zusätzlich gibt es die Klasse `Error` und die gemeinsame Klasse`Throwable`.

- **Wozu sind Ausnahmen gut?**

  Um in bestimmten Situationen mit einem `throw` Befehl den Programmfluß abzubrechen um eine besondere Ausnahmesituation zu signalisieren.

- **Was für Objekte dürfen in eine `throw` geworfen werden?**

  Nur Objekte einer Unterklasse von `Throwable`.

- **Wie werden geworfene Ausnahme behandelt?**

  Indem man sie innerhalb eines `try-catch`-Konstruktes wieder abfängt.

- **Kann man auf unterschiedliche Ausnahmen in einem `catch` unterschiedlich reagieren?**

  Ja, einfach mehrere `catch` untereinanderschreiben. Das als erstes zutreffende `catch` ist dann aktiv.

- **Was ist das mit dem `finally` beim `try-catch`?**

  Hier kann Code angegeben werden, der immer ausgeführt werden soll, egal ob und welche Ausnahme aufgetreten ist. Dieses ist sinnvoll um eventuell externe Verbindungen und ähnliches immer sauber zu schließen.

- **Darf man beliebig Ausnahmen werfen?**

  Ausnahmen, die nicht in einem `catch` abgefangen werden müssen in der `throws`-Klausel einer Methode deklariert werden.

- **Alle?**

  Nein, Ausnahmeobjekte von `RuntimeException` dürfen auch geworfen werden, wenn sie nicht in der `throws`-Klausel einer Methode stehen.

- **Wozu sind Pakete da?**

  Unter anderen damit man nicht Milliarden von Klassen in einem Ordner hat und dabei die Übersicht verliert.

- **Ist mir egal. Pakete sind mir zu umständlich.**

  Das ist aber dumm, wenn man Klassen gleichen Namens aus verschiedenen Bibliotheken benutzen will. Die kann man nur unterscheiden, wenn sie in verschiedenen Paketen sind.

- **OK, ich sehs ein. Wie nenne ich mein Paket?**

  Am besten die eigene Webdomain rückwärsts nehmen also für uns an der FH: `de.fhwiesbaden.informatik.panitz.meineApplikation`

- **Da fehlt doch der Bindestrich.**

  Der ist kein gültiges Zeichen innerhalb eines Bezeichners in Java (es ist ja der Minusoperator). Bindestriche in Webadressen sind eine recht deutsche Krankheit.

- **Muss ich imports machen um Klassen aus anderen Paketen zu benutzen?**

  Nein. Das macht die Sache nur bewuemer, weil sonst der Klassenname immer und überall komplett mit seinem Paket angegeben werden muss, auch beim Konstruktoraufruf oder z.B. `instanceof`.

- **Machen Import-Anweisungen das Programm langsamer oder größer?**

  Nein! Sie haben insbesondere nichts mit includes in C gemein. Sie entsprechen eher dem `using namespace` aus C++.

- **Da gibt es doch auch noch `public`, `private` und `protected`.**

  Jaja, die Sichtbarkeiten. Hinzu kommt, wenn man keine Sichtbarkeit hinschreibt. `public` heißt von überall aufrufbar, `protected` heißt in Unterklassen und gleichem Paket aufrufbar, `package` das ist wenn man nichts hinschreibt, heißt nur im gleichen Paket sichtbar, und `private` nur in der Klasse.

- **Und dann war da noch das `final`.**

  Das hat zwei Bedeutungen: bei Variablen und Feldern, dass sie nur einmal einen Wert zugewiesen bekommen, bei Klassen, dass keine Unterklassen von der Klasse definiert werden

- **Ich will GUIs programmieren.**

  Wunderbar, da gibt es zwei Bibliotheken: Swing und AWT.

- **Na toll, warum zwei?**

  Historisch.

- **Und welche sol ich jetzt benutzen?**

  Swing!

- **Also kann ich alles über AWT vergessen, insbesondere das Paket `java.awt`?**

  Nein. Swing benutzt große Teile aus AWT. Insbesondere hat jede Swingkomponente eine Oberklasse aus AWT. Aber auch die Ereignisbehandlungsklassen werden vollständig aus AWT benutzt.

- **Und woran erkenne ich jetzt, wann ich Komponente aus AWT bnutzen muss, obwohl ich in Swing programmiere?**

  Alle graphischen AWT-Komponenten haben ein Swing Pendant, das mit dem dem Buchstaben `J` beginnt, z.B. `javax.swing.JButton` im Gegensatz zu `java.awt.Button`.

- **Sonst noch paar Tipps, zum benutzen von Swing?**

  Keine eigenen Threads schreiben! Nicht die Methode `paint` sondern `paintComponent` überschreiben.

- **Toll, wenn ich keine Threads schreiben soll, wie mache ich dann z.B. zeitgesteuerte Ereignisse in Swing?**

  Da gibt es die Klasse `javax.swing.Timer` für.

- **Ich will XML verarbeiten.**

  Kein Problem, alles vorhanden, soll es das DOM-Api oder SAX sein?

- **Weiß nicht? Was ist der Unterschied?**

  In DOM wird eine Baumstruktur für ein XML-Dokument im Speicher aufgebaut. Mit der kann man alles machen, was man mit einem Baum machen kann.

- **Super. Wozu ist dann SAX?**

  Oft braucht man nie die ganze Baumstruktur, sondern muss das Dokument nur einmal von vorne nach hinten durcharbeiten. Dabei hilft SAX

- **Ist dann effizienter schätze ich?**

  Ganz genau, insbesondere wird viel weniger Speicher benötigt.

- **Na ich weiß nicht, ob ich jetzt genügend weiß für die Klausur?**

  Das weiß ich auch nicht, über alles haben wir hier nicht gesprochen. Ich denke aber es reicht an Wissen aus.

Anhang C

Wörterliste

In dieser Mitschrift habe ich mich bemüht, soweit existent oder naheliegend, deutsche Ausdrücke für die vielen in der Informatik auftretenden englischen Fachbegriffe zu benutzen. Dieses ist nicht aus einem nationalen Chauvinismus heraus, sondern für eine flüssige Lesbarkeit des Textes geschehen. Ein mit sehr vielen englischen Wörtern durchsetzter Text ist schwerer zu lesen, insbesondere auch für Menschen, deren Muttersprache nicht deutsch ist.

Es folgen hier Tabellen der verwendeten deutschen Ausdrücke mit ihrer englischen Entsprechung.

Deutsch	**Englisch**
Abbildung	map
Attribut	modifier
Aufrufkeller	stack trace
Ausdruck	expression
Ausnahme	exception
Auswertung	evaluation
Bedingung	condition
Befehl	statement
Behälter	container
Blubbersortierung	bubble sort
Datei	file
Deklaration	declaration
Eigenschaft	feature
Erreichbarkeit	visibility
Fabrikmethode	factory method
faul	lazy
Feld	field
geschützt	protected
Hauruckverfahren	bruteforce
Implementierung	implementation
Interpreter	interpreter
Keller	stack
Klasse	class
Kommandozeile	command line
Menge	set
Methode	method
Modellierung	design
nebenläufig	concurrent
Oberklasse	superclass
Ordner	folder
Paket	package
privat	private
Puffer	buffer
Reihung	array
Rumpf	body
Rückgabetyp	return type
Sammlung	collection
Schleife	loop
Schnittstelle	interface
Sichtbarkeit	visibility

Steuerfaden	thread
strikt	strict
Strom	stream
Typ	type
Typzusicherung	type cast
Unterklasse	subclass
Verklemmung	dead lock
Verzeichnis	directory
vollqualifizierter Name	fully qualified name
Zeichen	character
Zeichenkette	string
Zuweisung	assignment
öffentlich	public
Übersetzer	compiler

Englisch	**Deutsch**
array	Reihung
assignment	Zuweisung
body	Rumpf
bruteforce	Hauruckverfahren
bubble sort	Blubbersortierung
buffer	Puffer
character	Zeichen
class	Klasse
collection	Sammlung
command line	Kommandozeile
compiler	Übersetzer
concurrent	nebenläufig
condition	Bedingung
container	Behälter
dead lock	Verklemmung
declaration	Deklaration
design	Modellierung
directory	Verzeichnis
evaluation	Auswertung
exception	Ausnahme
expression	Ausdruck
factory method	Fabrikmethode
feature	Eigenschaft
field	Feld
file	Datei
folder	Ordner
fully qualified name	vollqualifizierter Name
implementation	Implementierung
interface	Schnittstelle
interpreter	Interpreter
lazy	faul
loop	Schleife
map	Abbildung
method	Methode
modifier	Attribut
package	Paket
private	privat
protected	geschützt
public	öffentlich
return type	Rückgabetyp
set	Menge
stack	Keller
stack trace	Aufrufkeller
statement	Befehl
stream	Strom
strict	strikt

string	Zeichenkette
subclass	Unterklasse
superclass	Oberklasse
thread	Steuerfaden
type	Typ
type cast	Typzusicherung
visibility	Sichtbarkeit
visibility	Erreichbarkeit

Anhang D

Gesammelte Aufgaben

Aufgabe 1 Öffnen Sie die Kommandozeile ihres Betriebssystems. Machen Sie sich mit rudimentären Befehlen der Kommandozeile vertraut. Erzeugen Sie mit dem Befehl `mkdir` einen neuen Ordner. Wechseln Sie mit `cd` in diesen Ordner.

Aufgabe 2 Schreiben Sie das Programm `FirstProgram.java` des Skriptes mit einem Texteditor ihrer Wahl. Speichern Sie es als `FirstProgram.java` in dem auf der Kommandozeile neu erstellten Ordner ab.

Aufgabe 3 Lassen Sie sich das Programm, in der Kommandozeile mit dem Befehl `more` einmal anzeigen.

Aufgabe 4 Übersetzen Sie Das Programm auf der Kommandozeile mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `FirstProgram.class`. Lassen Sie sich mit dem Befehl `ls -l` anzeigen, ob die Datei `FirstProgram.class` tatsächlich erzeugt wurde.

Führen Sie nun das Programm mit dem Javainterpreter `java` aus.

Sofern Ihnen unterschiedliche Betriebssysteme zur Verfügung stehen, führen Sie dieses sowohl einmal auf Linux als auch einmal unter Windows durch.

Aufgabe 5 Experimentieren Sie ein wenig mit den ersten kleinen Javaprogramm herum.

Aufgabe 6 Von ihrem heimischen Rechner loggen Sie sich mit dem Befehl `ssh` von daheim auf den Server an der Hochschule auf der Kommandozeile ein. Dafür müssen Sie sich mit Ihrer Informatik-Benutzerkennung einloggen:
`ssh ihreKennung@login1.cs.hs-rm.de`

Starten Sie nun das Programm mit dem Javainterpreter auf dem Server.

Aufgabe 7 Loggen Sie sich von zu Hause mit dem Befehl `sftp` auf dem Server `login1.cs.hs-rm.de` ein. Navigieren Sie dann mit dem Befehl `cd` in den Ordner, in dem Ihr Programm liegt. Holen Sie es dann mit dem Befehl `get FirstProgram.java` auf Ihren heimischen Rechner.

Aufgabe 8 Schreiben Sie Klassen, die die Objekte des Bibliotheksystems repräsentieren können:

- Personen mit Namen, Vornamen, Straße, Ort und Postleitzahl.
- Bücher mit Titel und Autor.

- Datum mit Tag, Monat und Jahr.
- Buchausleihe mit Ausleiher, Buch und Datum.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben Sie für jede dieser Klassen eine Methode `public String toString()` mit dem Ergebnistyp `String`. Das Ergebnis soll eine gute textuelle Beschreibung des Objektes sein.[1]

c) Schreiben Sie eine Hauptmethode in einer Klasse `Main`, in der Sie Objekte für jede der obigen Klassen erzeugen und die Ergebnisse der `toString`-Methode auf den Bildschirm ausgeben.

Aufgabe 9 Suchen Sie auf Ihrer lokalen Javainstallation oder im Netz auf den Seiten von Sun (`http://www.javasoft.com`) nach der Dokumentation der Standardklassen von Java. Suchen Sie die Dokumentation der Klasse `String`. Testen Sie einige der für die Klasse `String` definierten Methoden.

Aufgabe 10 Ergänzen Sie jetzt die Klasse `Person` aus der letzten Aufgabe um ein statisches Feld `letzerVorname` mit einer Zeichenkette, die angeben soll, welchen Vornamen das zuletzt erzeugte Objekt vom Typ `Person` hatte. Hierzu müssen Sie im Konstruktor der Klasse Person dafür sorgen, dass nach der Zuweisung der Objektfelder auch noch das Feld `letzerVorname` verändert wird. Testen Sie in einer Testklasse, dass sich tatsächlich nach jeder Erzeugung einer neuen Person dieses Feld verändert hat.

Aufgabe 11 In dieser Aufgabe sollen Sie eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben.

Gegeben seien die folgenden Javaklassen, wobei Sie die Klasse `Dialogue` nicht zu analysieren oder zu verstehen brauchen:

- ButtonLogic.java

```
1  class ButtonLogic {
2    String getDescription(){
3      return "in Großbuchstaben umwandeln";
4    }
5    String eval(String x){return x.toUpperCase();}
6  }
```

- Dialogue.java

```
1  import javax.swing.*;
2  import java.awt.event.*;
3  import java.awt.*;
4  class Dialogue extends JFrame{
5
6    final ButtonLogic logic;
7
8    final JButton button;
9    final JTextField inputField  = new JTextField(20) ;
10   final JTextField outputField = new JTextField(20) ;
11   final JPanel p = new JPanel();
12
13   Dialogue(ButtonLogic l){
14     logic = l;
15     button=new JButton(logic.getDescription());
16     button.addActionListener
```

[1] Sie brauchen noch nicht zu verstehen, warum vor dem Rückgabetyp noch ein Attribut `public` steht.

```
17      (new ActionListener(){
18         public void actionPerformed(ActionEvent _){
19           outputField.setText
20              (logic.eval(inputField.getText().trim()));
21         }
22       });
23     p.setLayout(new BorderLayout());
24     p.add(inputField,BorderLayout.NORTH);
25     p.add(button,BorderLayout.CENTER);
26     p.add(outputField,BorderLayout.SOUTH);
27     getContentPane().add(p);
28     pack();
29     setVisible(true);
30   }
31 }
```

TestDialogue.java

```
1 class TestDialogue {
2   public static void main(String [] _){
3     new Dialogue(new ButtonLogic());
4   }
5 }
```

a) Übersetzen Sie die drei Klassen und starten Sie das Programm.

b) Schreiben Sie eine Unterklasse der Klasse `ButtonLogic`. Sie sollen dabei die Methoden `getDescription` und `eval` so überschreiben, dass der Eingabestring in Kleinbuchstaben umgewandelt wird. Schreiben Sie eine Hauptmethode, in der Sie ein Objekt der Klasse `Dialogue` mit einem Objekt Ihrer Unterklasse von `ButtonLogic` erzeugen.

c) Schreiben Sie jetzt Applikationen, die die Umwandlung von Groß- in Klein- bzw. Klein- in Großbuchstaben entsprechend einer Türkischen Lokalisierung durchführen.

d) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so dass Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, das für den im Eingabefeld eingegebenen String die Länge im Ausgabefeld anzeigt.

e) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so dass Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, das für den im Eingabefeld eingegebenen String anzeigt, ob dieser einen Teilstring `"depp"` enthält.

Aufgabe 12 In dieser Aufgabe sollen Sie Klassen zur Beschreibung geometrischer Objekte im 2-dimensionalen Raum entwickeln. Wir beginnen mit der Klasse für Punkte im zweidimensionalen Raum. Schreiben Sie für die Klasse geeignete Konstruktoren und eine Methode `toString`.

a) Schreiben Sie eine Klasse `Vertex`, die Punkte im zweidimensionalen Raum darstellt. Benutzen Sie für die Koordinaten dabei den primitiven Typ `double`.

b) Schreiben Sie einen geeigneten Konstruktor für die Klasse `Vertex`.

c) Schreiben Sie für die Klasse `Vertex` eine Methode `toString`.

d) Überschreiben Sie für die Klasse `Vertex` Methode `equals`.

e) Definieren Sie Methoden zur Addition und Subtraktion von Punkten, sowie zur Multiplikation mit einem Skalar mit folgenden Signaturen.

```
1 Vertex add(Vertex that);
2 Vertex sub(Vertex that);
3 Vertex mult(double skalar);
```

Diese Methoden sollen das Objekt unverändert lassen und ein neues Vertex-Objekt als Ergebnis zurückliefern.

f) Testen Sie die Klasse `Vertex` in einer `main`-Methode.

Aufgabe 13 Sie sollen auf diesem Übungsblatt die Aufgaben der vorherigen Aufgabe weiterführen und die dort programmierte Klassen benutze und gegenfalls verändern.

a) Schreiben Sie eine Klasse `GeometricObject`. Eine geometrische Figur soll als Eigenschaften die Weite und Höhe eines gedachten die Figur umschließenden Rechtecks *(bounding box)* haben, sowie die Position der linken oberen Ecke dieses Rechtecks im 2-dimensionalen Raum. (Wobei das Koordinatensystem von oben nach unten und links nach rechts aufsteigend verläuft.)

b) Schreiben Sie geeignete Konstruktoren für die Klasse `GeometricObject` und eine aussagekräftige Methode `toString`.

c) Überschreiben Sie für die Klasse `GeometricObject` Methode `equals`.

d) Implementieren Sie die folgende Methoden für Ihre Klasse `GeometricObject`:

- `void moveTo(Vertex p)`, die den Eckpunkt der Figur auf einen neuen Wert setzt.
- `void move(Vertex v)`, die den Eckpunkt um die Kordinatenwerte des Arguments verschiebt.
- Eine Methode `area`, die den Flächeninhalt der *bounding box* berechnet.
- `boolean hasWithin(Vertex p)`, die wahr ist, wenn der Punkt `p` innerhalb der *bounding box* der geometrischen Figur liegt.

Testen Sie diese Methoden in einer Methode `main`.

Aufgabe 14 Sie sollen auf diesem Übungsblatt die Aufgaben der vorherigen Aufgabe weiterführen und die dort programmierte Klassen benutzen und gegenfalls verändern.

a) Implementieren Sie eine Unterklasse `Ellipse` der Klasse `GeometricObject`, die Ellipsen darstellt. Schreiben Sie geeignete Konstruktoren und überschreiben Sie die Methode `toString` und die Methode `area`, so dass jetzt der Flächeninhalt des Kreises und nicht der bounding box berechnet wird.

Aufgabe 15 Starten Sie folgendes Javaprogramm:

TestInteger.java

```
1 class TestInteger {
2   public static void main(String [] _){
3     System.out.println(2147483647+1);
4     System.out.println(-2147483648-1);
5   }
6 }
```

Erklären Sie die Ausgabe.

Aufgabe 16 Schreiben Sie eine Funktion, mit zwei ganzzahligen Parametern. Die Rückgabe soll die größere der beiden Zahlen sein. Testen Sie die Funktionen in einer Hauptfunktion mit mindesten drei Aufrufen.

Aufgabe 17 Nehmen Sie das Programm zur Berechnung der Fakultät und testen sie es mit verschiedenen Argumenten.

Aufgabe 18 Im Bestseller *Sakrileg (der Da Vinci Code)* spielen die Fibonaccizahlen eine Rolle. Für eine natürliche Zahl n ist ihre Fibonaccizahl definiert durch:

$$f(n) = \begin{cases} n & \text{für } n \leq 1 \\ f(n-1) + f(n-2) & \text{für } n > 1 \end{cases}$$

Programmieren Sie eine Funktion `int fib(int n)`, die für eine Zahl, die entsprechende Fibonaccizahl zurückgibt.

Geben Sie in der Hauptfunktion die ersten 20 Fibonaccizahlen aus.

Aufgabe 19 Ergänzen Sie ihre Klasse `Ausleihe` um eine Methode `void verlaengereEinenMonat()`, die den Rückgabetermin des Buches um einen Monat erhöht.

Aufgabe 20 Modellieren und schreiben Sie eine Klasse `Counter`, die einen Zähler darstellt. Objekte dieser Klasse sollen folgende Funktionalität bereitsstellen:

- Eine Methode `click()`, die den internen Zähler um eins erhöht.
- Eine Methode `reset()`, die den Zähler wieder auf den Wert `0` setzt.
- Eine Methode, die den aktuellen Wert des Zählers ausgibt.

Testen Sie Ihre Klasse.

Aufgabe 21 Schreiben Sie mit den bisher vorgestellten Konzepten ein Programm, das unendlich oft das Wort `Hallo` auf den Bildschirm ausgibt. Was beobachten Sie, wenn sie das Programm lange laufen lassen?

Aufgabe 22 Schreiben Sie eine Methode, die für eine ganze Zahl die Fakultät dieser Zahl berechnet. Testen Sie die Methode zunächst mit kleinen Zahlen, anschließend mit großen Zahlen. Was stellen Sie fest?

Aufgabe 23 Modellieren und schreiben Sie eine Klasse, die ein Bankkonto darstellt. Auf das Bankkonto sollen Einzahlungen und Auszahlungen vorgenommen werden können. Es gibt einen maximalen Kreditrahmen. Das Konto soll also nicht beliebig viel in die Miese gehen können. Schließlich muss es eine Möglichkeit geben, Zinsen zu berechnen und dem Konto gutzuschreiben.

Aufgabe 24 Schreiben Sie jetzt die Methode zur Berechnung der Fakultät, indem Sie eine Iteration und nicht eine Rekursion benutzen.

Aufgabe 25 Schreiben Sie eine Methode
`static String darstellungZurBasis(int x,int b)`, die als Parameter eine Zahl x und eine zweite Zahl b erhält. Sie dürfen annehmen, dass $x > 0$ und $1 < b < 11$. Das Ergebnis soll eine Zeichenkette vom Typ `String` sein, in der die Zahl x zur Basis b dargestellt ist. Testen Sie ihre Methode mit unterschiedlichen Basen.

Hinweis: Der zweistellige Operator `%` berechnet den ganzzahligen Rest einer Division. Bei einem geschickten Umgang mit den Operatoren `%, /` und `+` und einer `while`-Schleife kommen Sie mit sechs Zeilen im Rumpf der Methode aus.

Aufgabe 26 Schreiben Sie eine Methode `static int readIntBase10(String str)`. Diese Methode soll einen String, der nur aus Ziffern besteht, in die von ihm repräsentierte Zahl umwandeln. Benutzen sie hierzu die Methode `charAt` der `String`-Klasse, die es erlaubt, einzelne Buchstaben einer Zeichenkette zu selektieren.

Aufgabe 27 Im Bestseller *Sakrileg (der Da Vinci Code)* spielen die Fibonaccizahlen eine Rolle. Für eine natürliche Zahl n ist ihre Fibonaccizahl definiert durch:

$$f(n) = \begin{cases} n & \text{für } n \leq 1 \\ f(n-1) + f(n-2) & \text{für } n > 1 \end{cases}$$

Programmieren Sie eine Funktion `static int fib(int n)`, die für eine Zahl, die entsprechende Fibonaccizahl zurückgibt.

Geben Sie in der Hauptfunktion die ersten 20 Fibonaccizahlen aus.

Durchlaufen Sie das Programm auch einmal schrittweise mit dem Debugger in Eclipse.

Aufgabe 28 In dieser Aufgabe sollen Sie die Klasse `Datum` des ersten Übungsblattes weiterentwickeln und für diese die folgenden nützlichen Methoden schreiben:

a) Schreiben Sie in der Klasse `Datum` eine Methode
`boolean isEarlier(Datum that)`
die genau dann wahr ergibt, wenn das `this`.Datumsobjekt früher liegt, als das Datum übergebene Datumsobjekt.

b) Schreiben Sie eine Methode
`boolean schaltjahr()`,
die angibt, ob es sich bei dem Jahr des Datums um ein Schaltjahr handelt oder nicht.

c) Schreiben Sie eine Methode
`void toNextDay()`
die das Datumsobjekt um einen Tag weiterschaltet.

d) Schreiben Sie eine Methode `Wochentag wasFuerEinTag()`, die angibt, um was für einen Wochentag es sich bei dem betreffenden Datum handelt.
Eine algorithmische Beschreibung hierzu finden sie auf:
`de.wikipedia.org/wiki/Wochentagsberechnung`.

Der Ergebnistyp `Wochentag` sei ein Objekt der folgenden Aufzählungsklasse:

Wochentag.java

```
1 public enum Wochentag {
2   montag,dienstag,mittwoch,donnerstag
3  ,freitag,sonnabend,sonntag;
4 }
```

Aufgabe 29 In dieser Aufgabe sollen Sie wie auf dem dritten Übungsblatt für das kleine Dialog-Gui eine Anwendungslogik schreiben. Die Eingabe soll dabei auf Zahlen beschränkt sein. Hierzu sei folgende Unterklasse der Klasse `ButtonLogic` gegeben.

ButtonLogicInt.java

```
1 class ButtonLogicInt  extends ButtonLogic {
2   String getDescription(){
3     return "Zahl Umwandeln";
4   }
5   String eval(String x){return eval(new Integer(x.trim()));}
6
7   String eval(int x){return x+"";}
8 }
```

Schreiben Sie jetzt eine Anwendung, bei der die Zahlen, die im Eingabefeld eingegeben wird als Dualzahl im Ausgabefeld angegeben wird. Schreiben Sie hierzu eine Unterklasse von `ButtonLogicInt`.

Aufgabe 30 Implementieren Sie für die Klasse `Or` die Methode `undUeberOder`, so dass sie die folgende Umformung realisiert:

- Ersetze $(A \vee (B \wedge C))$ durch $(A \vee B) \wedge (A \vee C)$
- und ersetze $((B \wedge C) \vee A)$ durch $(B \vee A) \wedge (C \vee A)$.

Hinweis: Unterscheiden Sie mit dem Operator `instanceof` ob es sich bei dem linken oder rechten Teilformeln um eine Konjunktion handelt.

Aufgabe 31 Implementieren Sie die Methode `negatNachInnen` für die Klasse `Not`. Sie soll folgende Umformungsschritte realsieren:

Ersetze sukzessive Teilformeln nach den folgenden Regeln:

- $\neg\neg A$ durch A,
- $\neg(A \vee B)$ durch $(\neg A \wedge \neg B)$,
- $\neg(A \wedge B)$ durch $\neg A \vee \neg B$

Aufgabe 32 Testen Sie Ihre Bibliothek für logische Formeln und die Umwandlung in Klauselform an unterschiedlichen Beispielen. Erzeugen Sie hierzu ein paar LaTeX-Dokumente.

Aufgabe 33 Legen Sie die Klassen aus Übungsblatt 3 in ein Paket, das Sie eindeutig charakterisiert.

Aufgabe 34 Lassen Sie jetzt zusätzlich die Klasse `GeometricObject` folgende Schnittstelle implementieren:

GameObject.java

```
1 package name.panitz;
2 import java.awt.Graphics;
3 public interface GameObject{
4   public boolean touches(GameObject that);
5   public Vertex getPosition();
6   public int getWidth();
7   public int getHeight();
8   public void paintMeTo(Graphics g);
9   public void move();
```

```
10   public void reverse();
11   public double weight();
12 }
```

Die müssen hierbei noch in der Schnittstelle Ihre Klasse Vertex importieren.

a) `touches` soll prüfen, ob die umgebenen Vierecke zweier Spielobjekte gemeinsame Punkte haben.

b) `paintMeTo` soll das Objekt auf eine Leinwand zeichnen.

c) `move` soll das Objekt einer Bahn einen Schritt weiterbewegen.

d) `reverse` soll die Bewegungsrichtung des Objekts umdrehen.

e) weight soll die Schwere des Objektes ausdrücken. Diese soll möglichst mit dem Flächeninhalt ausgedrückt sein.

f) Schreiben Sie JUnit-Test für die Methode `touches`.

Aufgabe 35 Erzeugen Sie in einer Testmethode ein Objekt der folgenden Klasse:

GameWindow.java

```
1  package name.panitz;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.util.ArrayList;
5  import java.util.List;
6  import javax.swing.*;
7
8  public class GameWindow extends JFrame{
9    JPanel p = new JPanel(){
10     public Dimension getPreferredSize() {
11       return new Dimension(800,600);
12     };
13     protected void paintComponent(Graphics g) {
14       for (GameObject o:os) o.paintMeTo(g);
15     };
16   };
17   final List<GameObject> os=new ArrayList<GameObject>();
18   public GameWindow() {
19     add(p);
20     pack();
21     setVisible(true);
22     Timer t = new Timer(100,new ActionListener(){
23        public void actionPerformed(ActionEvent e){
24          for (GameObject o:os) o.move();
25          repaint();
26        }
27     }
28     );
29     t.start();
30   }
31   public void addGameObject(GameObject o){
32     os.add(o);
33     p.repaint();
34   }
35 }
```

Fügen Sie dem Objekt dabei jeweils unterschiedliche Objekte der Klassen, die die Schnittstelle `GameObject` implementieren, mit der Methode `addGameObject` hinzu.

Aufgabe 36 Implementieren Sie eine Unterklasse `EquilateralTriangle` der Klasse `GeometricObject`, die gleichseitige Dreiecke darstellt. Schreiben Sie geeignete Konstruktoren und überschreiben Sie alle Methoden auf eine sinnvolle Art und Weise. Der Konstruktor soll die Seitenlänge des gleichseitigen Dreiecks erhalten.

Aufgabe 37 Nehmen Sie beide der in diesem Kapitel entwickelten Umsetzungen von Listen und fügen Sie ihrer Listenklassen folgende Methoden hinzu. Führen Sie Tests für diese Methoden durch.

a) `a last()`: gibt das letzte Element der Liste aus.

b) `List<a> concat(List<a> other)` bzw.: `Li<a> concat(Li<a> other)`: erzeugt eine neue Liste, die erst die Elemente der `this`-Liste und dann der `other`-Liste hat, es sollen also zwei Listen aneinander gehängt werden.

c) `a elementAt(int i)`: gibt das Element an einer bestimmten Indexstelle der Liste zurück. Spezifikation:

$$\begin{aligned} elementAt(Cons(x, xs), 1) &= x \\ elementAt(Cons(x, xs), n+1) &= elementAt(xs, n) \end{aligned}$$

Aufgabe 38 Verfolgen Sie schrittweise mit Papier und Beistift, wie der *quicksort* Algorithmus die folgenden zwei Listen sortiert:

- `("a","b","c","d","e")`
- `("c","a","b","d","e")`

Aufgabe 39 Diese Aufgabe soll mir helfen, Listen für Ihre Leistungsbewertung zu erzeugen.

a) Implementieren Sie für Ihre Listenklasse eine Methode `String toHtmlTable()`, die für Listen Html-Code für eine Tabelle erzeugt, z.B:

```
<table>
  <tr>erstes Listenelement</tr>
  <tr>zweites Listenelement</tr>
  <tr>drittes Listenelement</tr>
</table>
```

b) Nehmen Sie die Klasse `Student`, die Felder für Namen, Vornamen und Matrikelnummer hat. Implementieren Sie für diese Klasse eine Methode `String toTableRow()`, die für Studenten eine Zeile einer Html-Tabelle erzeugt:

```
Student s1 = new Student("Müller","Hans",167857);
System.out.println(s1.toTableRow());
```

soll folgende Ausgabe ergeben:

```
<td>Müller</td><td>Hans</td><td>167857</td>
```

Ändern Sie die Methode `toString` so, dass sie dasselbe Ergebnis wie die neue Methode `toTableRow` hat.

c) Legen Sie eine Liste von Studenten an, sortieren Sie diese mit Hilfe der Methode `sortBy`nach Nachnamen und Vornamen und erzeugen Sie eine Html-Seite, die die sortierte Liste anzeigt.

Sie können zum Testen die folgende Klasse benutzen:

HtmlView.java

```
1  package name.panitz.data.list;
2  import java.awt.*;
3  import java.awt.event.*;
4  import javax.swing.*;
5  import javax.swing.plaf.basic.*;
6  import javax.swing.text.*;
7  import javax.swing.text.html.*;
8
9  public class HtmlView extends JPanel {
10
11   //example invocation
12   public static void main(String s[]) {
13     HtmlView view = new HtmlView();
14     view.run();
15     view.setText("<h1>hallo</h1>");
16   }
17
18   JFrame frame;
19   JTextPane ausgabe = new JTextPane();
20
21   public HtmlView() {
22     ausgabe.setEditorKit(new HTMLEditorKit());
23     add(ausgabe);
24   }
25
26   void setText(String htmlString){
27     ausgabe.setText(htmlString);
28     frame.pack();
29     ausgabe.repaint();
30   }
31
32   void run(){
33     frame = new JFrame("HtmlView");
34     frame.getContentPane().add(this);
35     frame.pack();
36     frame.setVisible(true);
37   }
38 }
```

Aufgabe 40 Gegeben sei folgende Schnittstelle:

List.java

```
1  package name.panitz.util;
2
3  public interface List<A> {
4
5    A getElement();
6    List<A> getNext();
7
8    boolean isEmpty();
9    void add(A a);
10
11   int size();
12   boolean contains(A a);
13   A get(int i);
```

```
14    A last();
15    List<A> reverse();
16  }
```

Schreiben Sie eine Klasse mit dem vollqualifizierten Namen
`name.panitz.util.solution.LinkedList`,
die die obige Schnittstelle implementiert. Die Klasse soll zusätzlich einen öffentlichen Standardkonstruktor haben der eine leere Liste erzeugt.

Die Methoden sind im einzelnen wie folgt zu implementieren:

a) `getElement`: gibt das vorderste Element, das in der Liste gespeichert ist zurück. Ist die Liste leer, so wird `null` zurück gegeben.

b) `getNext`: gibt die Restliste ohne das ersten Element der Liste, zurück. Ist die Liste leer, so wird `null` zurück gegeben.

c) `isEmpty`: gibt an, ob es sich um eine leere Liste handelt.

d) `add`: modifiziert die Liste, indem es vorne an die Liste ein neues Element einfügt.

e) `size`: gibt die Anzahl der Elemente in der Liste zurück.

f) `contains`: gibt an, ob das übergebene Objekt in der Liste enthalten ist. Es wird für den Test die `equals`-Methode verwendet.

g) `get`: gibt das Element an dem als Parameter übergebenen Index zurück. Das erste Element habe den Index 0. Für negativen oder für zu großen Index wird `null` zurück gegeben.

h) `last`: gibt das letzte in der Liste gespeicherte Element zurück.

i) `reverse`: erzeugt eine vollkommen neue Liste, die aus den Elementen dieser Liste in umgekehrter Reihenfolge besteht.

Auf der Webseite `http://swtsrv01.cs.hs-rm.de/ksteb001` existiert ein allererster Prototyp einer Webapplikation, auf der Lösungen zu Aufgaben hochgeladen werden können. Die Webapplikation lässt dann JUnit-Tests über die hochgeladene Lösung laufen und zeigt das Ergebnis dieser Tests an.

Aufgabe 41 Schreiben Sie eine Klasse `CompoundObject`, die eine Unterklasse von `GeometricObject` darstellt. Ein `CompoundObject` soll eine Liste weiterer Geometrische Objekte enthalten, die zusammen ein größeres Objekt darstellen sollen.

Achten Sie darauf, dass die *Bounding-Box* entsprechend korrekt gesetzt ist.

Sie sollen dabei Ihre eigene Listenimplementierung aus der vorherigen Aufgabe benutzen und nicht auf eine der Standardlisten zurückgreifen.

Testen Sie ein paar Objekt der Klasse `CompoundObject` in Ihrer kleinen GUI-Anwendung aus dem letzten Aufgabenblatt.

Klassenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.

[Pan00] Sven Eric Panitz. Generische Typen in Bolero. *Javamagazin*, 4 2000.

[PJ03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.