Programmieren I

WS 03/04

Sven Eric Panitz
TFH Berlin

Version 17. März 2004

Dieses Skript entstand begleitend zur Vorlesung des WS 02/03 vollkommen neu und stellte somit eine Mitschrift der Vorlesung dar. Naturgemäß ist es in Aufbau und Qualität nicht mit einem Buch vergleichbar. Flüchtigkeits- und Tippfehler ließen sich im Eifer des Gefechtes nicht vermeiden und traten in nicht geringem Maße auf. Ich bin natürlich stets dankbar, wenn ich auf solche aufmerksam gemacht werde und diese korrigieren kann.

Diese Version des Skriptes ist eine Neuauflage für das WS03/04, in der einige Umstrukturierungen und Korrekturen vorgenommen wurden.

Besonderen Dank gebührt Christian Achter, der gewissenhaft dieses Skript auf Flüchtigkeits und Rechtschreibfehler durchgelesen hat.

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 werden. Der XML-Quelltext verweist direkt auf ein XSLT-Skript, das eine HTML-Darstellung erzeugt, so daß ein entsprechender Browser mit XSLT-Prozessor die XML-Datei direkt als HTML-Seite darstellen kann.

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

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.

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

- **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.
- **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:

```
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:


```
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

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

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

Aufgabe 1 Schreiben Sie das obige Programm mit einen Texteditor ihrer Wahl. Speichern Sie es als `FirstProgram.java` ab. Übersetzen Sie es mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `FirstProgram.class`. Führen Sie das Programm mit dem Javainterpreter `java` aus. 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

```
class Answer {
  public static void main(String [] args){
    System.out.println(42);
  }
}
```

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 aufstellen:

> 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 mit Java

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.

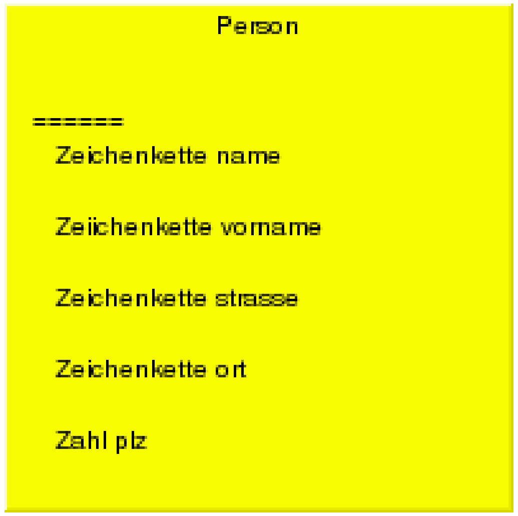

Abbildung 2.1: Modellierung einer Person.

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

[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.

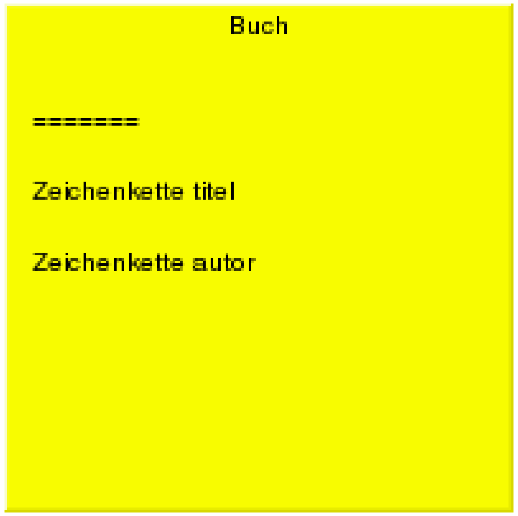

Abbildung 2.2: Modellierung eines Buches.

Abbildung 2.3: Modellierung eines Datums.

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.

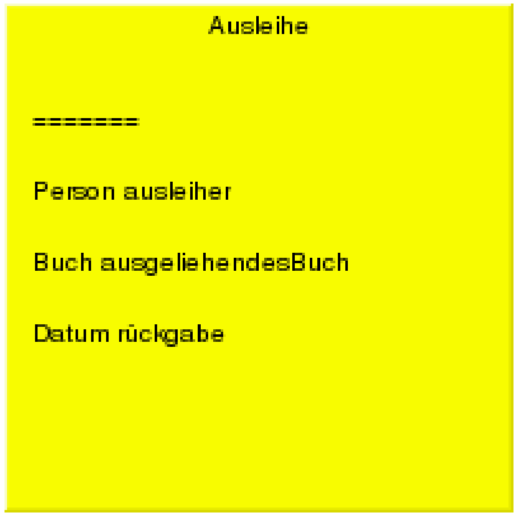

Abbildung 2.4: Modellierung eines Ausleihvorgangs.

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 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.

Minimal.java

```
class Minimal {
}
```

> **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();
```

Aufgabe 2 Schreiben sie ein Programm, das ein Objekt der Klasse `Minimal` erzeugt und auf dem Bildschirm ausgibt. Hierzu ersetzen sie einfach im Programm `Answer` die `42` durch den Ausdruck `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:

```
"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:
Im Folgenden schreiben wir eine Klasse mit zwei Feldern:

ErsteFelder.java
```
class ErsteFelder {
  Minimal meinFeld;
  String  meinTextFeld;
}
```

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.

Aufgabe 3 Schreiben Sie für die vier Karteikarten in der Modellierung eines Bibliotheksystems entsprechende Klassen mit den entsprechenden Feldern.

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:
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
```
class ErsteFelder2 {
  Minimal meinFeld = new Minimal();
  String  meinTextFeld = "hallo";
}
```

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:
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
```
class StringUtilMethod {
  String addStrings(String leftText, String rightText){
    return leftText+rightText;
  }
}
```

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:
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:

[3]Der Ausdruck für *Methoden* kommt speziell aus der objektorientierten Prorammierung. 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.

PersonExample1.java

```java
class PersonExample1 {
  String vorname;
  String nachname;

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

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

PersonExample2.java

```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

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

```



```java
  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:**
> Wir erzeugen ein Personenobjekt mit dem für die entsprechende Klasse geschriebenen Konstruktor:

TestePerson1.java

```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

```java
class FirstLocalFields{
  public static void main(String [] args){
    String str1 = "hello";
    String str2 = "world";
    System.out.println(str1);
    System.out.println(str2);
    String str3 = str1+" "+str2;
    System.out.println(str3);
  }
}
```

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
```
class GetPersonName{
  public static void main (String [] args){
    Person1 p = new Person1("August","Strindberg");

    String name = p.vorname;
    System.out.println(name);
  }
}
```

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
```
class CallFullNameMethod{
  public static void main (String [] args){
    Person1 p = new Person1("August","Strindberg");

    String name = p.getFullName();
    System.out.println(name);
  }
}
```

Aufgabe 4 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.

Hinweis: der Typ, der ganze Zahlen in Java bezeichnet, heißt `int`.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben Sie für jede dieser Klassen eine Methode `public String toString()` mit dem Ergebnistyp. 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 5 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
```
class UseThis {
  String aField ;
  UseThis(String aField){
    this.aField = aField;
  }
}
```

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.

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

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

```
  }
}
```

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

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

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 6 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.

Kapitel 3

Imperative Konzepte

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.

[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.

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.
- **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 $b-1$.

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

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

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

Gebräuchliche Basen sind:

- 2: Dualsystem
- 8: Oktalsystem
- 10: Dezimalsystem
- 16: Hexadezimalsystem[3]

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

> **Beispiel:**
> 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.

[3] Eine etwas unglückliche Namensgebung aus der Mischung eines griechischen mit einem lateinischen Wort.

[4] Wobei wir großzügig den Typ `char` ignorieren.

Beispiel:
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:
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.

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:
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:
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[**?**], in denen es einen Typ gibt, der potentiell ganze Zahlen von beliebiger Größe darstellen kann.

Aufgabe 7 Starten Sie folgendes Javaprogramm:

TestInteger.java

```
class TestInteger {
  public static void main(String [] _){
    System.out.println(2147483647+1);
```

```
    System.out.println(-2147483648-1);
  }
}
```

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:

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

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:
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:
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(8888888888888f);

    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 Bool (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
```
class Testbool{
  boolean boolFeld;

  Testbool(boolean b){
    boolFeld = b;
  }

  boolean getBoolFeld(){
    return this.boolFeld;
  }

  public static void main(String [] args){
    Testbool t = new Testbool(true);
    System.out.println(t.getBoolFeld());
    t = new Testbool(false);
    System.out.println(t.getBoolFeld());
  }
}
```

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

3.2 Operatoren

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`.

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

```java
class PunktVorStrich{
  public static void main(String [] args){
    System.out.println(2 + 20 * 2);
    System.out.println((2 + 20) * 2);
  }
}
```

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

```java
class Square{
  static int square(int i){
    return i*i;
  }
}
```

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

```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);
  }
}
```

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

```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

```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));
  }
}
```

3.3 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.

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

```java
class FirstIf {

  static void firstIf(boolean bedingung){
    if (bedingung) {
      System.out.println("Bedingung ist wahr");
    } else {
      System.out.println("Bedingung ist falsch");
    }
  }

  public static void main(String [] args){
     firstIf(true || false);
  }

}
```

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

```java
class FirstIf2 {

  static void firstIf(boolean bedingung){
    if (bedingung) System.out.println("Bedingung ist wahr");
```

[6] Konstrukte mit Operatoren nennt man dagegen Ausdrücke.


```java
    else System.out.println("Bedingung ist falsch");
  }

  public static void main(String [] args){
     firstIf(true || false);
  }

}
```

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

ElseIf.java

```java
class ElseIf {

  static String lessOrEq(int i,int j){
    if (i<10)  return "i kleiner zehn";
    else if (i>10)  return "i größer zehn";
    else if (j>10)  return "j größer zehn";
    else if (j<10)  return "j kleiner zehn";
    else return "j=i=10";
  }

  public static void main(String [] args){
     System.out.println(lessOrEq(10,9));
  }

}
```

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

```java
class Summe{
  static int summe(int i){
    if (i==1)  {
      return 1;
    }else {
      return summe(i-1) + i;
    }
  }
}
```

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 →:

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

Aufgabe 9 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 10 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 11 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 12 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:

$$\boxed{\texttt{while}\ (\textit{pred})\{\textit{body}\}}$$

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

```
class WhileTest {
  public static void main(String [] args){
    int i = 0;
    while (i < 10){
      i = i+1;
      System.out.println(i);
    }
  }
}
```

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

Summe2.java

```
class Summe2 {
  public static int summe(int n){

    int erg = 0 ;              // Feld für Ergebnis.
    int j   = n ;              // Feld zur Schleifenkontrolle.

    while (j>0){               // j läuft von n bis 1.
      erg = erg + j;           // akkumuliere das Ergebnis.
      j = j-1;                 // verringere Laufzähler.
    }

    return erg;
  }
}
```

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

```
class Bottom {
   static public void bottom(){
     while (wahr()){};
   }

   static boolean wahr(){return true;}

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

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

```
class DoTest {
  public static void main(String [] args){
    int i = 0;
    do {
      System.out.println(i);
      i = i+1;
    } while (i < 10);
  }
}
```

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

VorUndNach.java

```
class VorUndNach {

  public static void main(String [] args){

      while (falsch())
         {System.out.println("vorgeprüfte Schleife");};

      do {System.out.println("nachgeprüfte Schleife");}
      while (false);
  }

  public static boolean falsch(){return false;}
}
```

[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
```java
class ForTest {
  public static void main(String [] args){

    for (int i=0; i<10; i=i+1){
      System.out.println(i);
    }

  }
}
```

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
```java
class Summe3 {
  public static int summe(int n){

    int erg = 0 ;                     // Feld für Ergebnis

    for (int j = n;j>0;j=j-1){        // j läuft von n bis 1
      erg = erg + j;                  // akkumuliere das Ergebnis
    }

    return erg;
  }
}
```

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
```java
class Summe4 {
  public static int summe(int n){

    int erg = 0 ;                 // Feld für Ergebnis.
    int j   = n;                 // Feld zur Schleifenkontrolle

    for (;j>0;){                  // j läuft von n bis 1
      erg = erg + j;              // akkumuliere das Ergebnis.
      j = j-1;                    // verringere Laufzähler
    }

    return erg;
  }
}
```

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

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

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

ForBreak.java

```java
class ForBreak {
  public static void main(String [] args){
    int i = 0;
    for (;;){
        if (i>9) break;
        System.out.println(i);
        i=i+1;
    }
  }
}
```

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

```java
class ContTest{
  public static void main(String [] args){
     for (int i=1; i<1000;i=i+1){
        if (!(i % 17 == 0 || i % 19 == 0) )
      //wenn nicht die Zahl durch 17 oder 19 ohne Rest teilbar ist
```



```java
           continue;
        System.out.println(i);
     }
  }
}
```

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.

Kritik an Javas Schleifenkonstrukten

Die Entwickler von Java waren sehr konservativ im Entwurf der Schleifenkonstrukte. Hier schlägt sich deutlich die Designvorgabe nieder, nicht zu sehr von C entfernt zu sein. So kennt Java zwar drei Schleifenkonstrukte, doch sind diese relativ primitiv. Alle Schleifensteuerung muß vom Programmierer selbst codiert werden. Damit mischt sich Anwendungslogik mit Steuerungslogik. Es gibt auch keine Schleifenkonstrukte, deren Schleifen leichter als terminierend zu erkennen sind oder die garantiert terminieren. Man kann durchaus kritisieren, daß dieses nicht der aktuellste Stand der Technik ist. Im Folgenden ein paar kleine Beispiele, wie in anderen Programmiersprachen Schleifen ausgedrückt werden können.

For-Schleifen in Bolero Die Programmiersprache Bolero der Software AG, die ebenso wie Java Code für Javas virtuelle Maschine erzeugt, lehnt sich eher an die aus Datenbankanfragesprachen bekannte `select`-Anweisung an.

Die `for`-Schleife läuft hier über alle Elemente einer Liste oder eines Iteratorobjektes. Für Zahlen gibt es zusätzlich einfache syntaktische Konstrukte, um Iteratoren über diese zu erzeugen. Das Programm, das die Zahlen 0 bis 9 ausgibt, sieht in Bolero wie folgt aus:

```
for x in 0 to 9 do
  System.out.println(x)
end for
```

For-Schleifen in XQuery XQuery ist die Anfragesprache für XML-Daten, die derzeit vom W3C definiert wird. Hier geht man ähnliche Wege wie in Bolero:

```
for $x in 0 to 9
return <zahl>{$x}</zahl>
```

Da XQuery sich insbesondere mit Sequenzen von XML-Unterdokumenten beschäftigt und mit Hilfe von XPath-Ausdrücken diese selektieren kann, lassen sich komplexe Selektionen relativ elegant ausdrücken:

```
<autoren>{
  for $autor in $buchliste/buch/autor
  return $autor
  sortBy (./text())
}</autoren>
```

Mengenschreibweise in funktionalen Sprachen In funktionalen Sprachen spielen Listen eine fundamentale Rolle. Moderne funktionale Sprachen wie ML, Clean oder Haskell bieten Konstrukte an, die Listen erzeugen oder filtern. Hierzu bedient man sich einer Notation, die der mathematischen Mengenschreibweise entlehnt ist. Diese Technik wird *list comprehensions* bezeichnet. So gibt es einfache Literale, die Listen erzeugen:

- `[1,2..10]` erzeugt die Liste der Zahlen 1 bis 10
- `[2,4..100]` erzeugt die Liste der geraden Zahlen bis 100
- `[x*x | x<-[1,2..]]` erzeugt die unendliche Liste der Quadratzahlen entsprechend dem mathematischen Ausdruck: $\{x^2 | x \in \mathbb{N}\}$

Mit diesen Konstrukten läßt sich ein Ausdruck, der die Liste aller Primzahlen berechnet, in einer Zeile schreiben:

```
let sieb (x:xs)=(x:sieb [y|y<-xs,mod y x /= 0]) in sieb [2,3..]
```

Mit Javaversion 1.5 (Codename Tiger), deren erste Testversion für Ende 2003 öffentlich gemacht werden soll, wird es eine neue Variante der `for`-Schleife geben, die die in diesem Abschnitt vorgestellten Konzepte aus anderen Programmiersprachen zu weiten Teilen realisiert.

Aufgabe 13 Schreiben Sie jetzt die Methode zur Berechnung der Fakultät, indem Sie eine Iteration und nicht eine Rekursion benutzen.

Aufgabe 14 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 15 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 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

```java
class StackTest {
  public static void main(String [] args){
    System.out.println(count1(0));
    System.out.println(count2(0));
  }

  public static int count1(int k){
    int j = 2;
    for (int i= 0;i<1000000000;i=i+1){
      j=j+1;
    }
    return j;
  }

  public static int count2(int i){
      if (i<1000000000) return count2(i+1) +1; else return 2;
  }

}
```

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)

[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.

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 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:

$$\boxed{pred?\,alt1 : alt2}$$

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

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
```java
class Minus {

  static public int sub(int x,int y){
    int erg = x-y;
    System.out.println("eine Subtraktion mit Ergebnis: "
                       +erg+" wurde durchgeführt.");
    return erg;
  }
}
```

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
```java
class Operatorauswertung {

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

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
```java
public class Reihenfolge{

    /**
       Returns second argument.
       @param x ignored  argument.
       @param y returned argument.
       @return projection on second argument.
     **/
  public static int snd(int x,int y){return y;}

```

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

Reihenfolge.java
```java
    /**
       Returns the constants 1 and prints its argument.
       @param  str The String that is printed as side effect.
       @return constantly 1.
    **/
  public static int eins(String str){
    System.out.println(str); return 1;
  }
```

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


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
```java
class TerminationTest {
    /**
      Nonterminating method.
      @return never returns anything.
     **/
  public static int bottom( ){while(wahr()){};return 1; }

    /**
       Returns the constants 1 and prints its argument.
       @param  str The String that is printed as side effect.
       @return constantly 1.
    **/
  public static int eins(String str){
    System.out.println(str); return 1;
  }

    /**
       Constantly true method.
       @return constantly the value true.
    **/
  public static boolean wahr( ){return true; }
```

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

TerminationTest.java
```java
    /**
       Method mimicking conditional expression.
       @param pre The boolean condition.
       @param a1 The positive alternative.
       @param a2 The negative alternative.
       @return in case of pre the second otherwise third argument
     **/
  public static int wenn(boolean pre,int a1,int a2 ){
    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
```java
  public static void main(String [] args){

    //test, whether both alternatives of conditional
```

```
    //expression are evaluated or just one.

    //this will terminate:
    System.out.println(      false?bottom():eins("b")    );

    //this won't terminate:
    System.out.println(wenn( false,bottom(),eins("b") )  );
  }
}
```

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:

$$true \vee A = true \text{ und } false \wedge A = false$$

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

```
class LazyBool {

  static boolean booleanExpr(){
    System.out.println("in booleanExpr");
    return true;
  }

  static public void main(String [] _){
    System.out.println(true || booleanExpr());
    System.out.println(false && booleanExpr());
  }

}
```


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

```
class StrictBool {

  static boolean booleanExpr(){
    System.out.println("in booleanExpr");
    return true;
  }

  static public void main(String [] _){
    System.out.println(true | booleanExpr());
    System.out.println(false & booleanExpr());
  }

}
```

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>
```

[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.

Aufgabe 16 Für die Lösung dieser Aufgabe gibt es 3 Punkte, die auf die Klausur angerechnet werden. Voraussetzung hierzu ist, daß die Lösung mir in der Übung gezeigt und erklärt werden kann.

In dieser Aufgabe sollen Sie eine Klasse für römische Zahlen entwickeln.

a) Schreiben Sie eine Klasse `Roman`. Diese Klasse soll eine natürliche Zahl darstellen.

b) Schreiben Sie für Ihre Klasse `Roman` einen Konstruktor, der ein Stringobjekt als Parameter hat. Dieser Stringparameter soll eine römische Zahl darstellen. Der Konstruktor soll diese Zahl lesen und in einem Feld des Typs `int` abspeichern.

c) Implementieren Sie die Methode `public String toString()` für Ihre Klasse `Roman`, die die intern gespeicherte Zahl als römische Zahl dargestellt zurückibt.

d) Fügen Sie ihrer Klasse `Roman` die folgenden Methoden für arithmetische Rechnungen hinzu.

- `Roman add(Roman other)`
- `Roman sub(Roman other)`
- `Roman mul(Roman other)`
- `Roman div(Roman other)`

e) Testen Sie Ihre Klasse `Roman`.

Kapitel 4

Vererbung

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:

Person.java
```
class Person {

  String name ;
  String address;

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

  public String toString(){
    return name+", "+address;
  }
}
```

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
```
class StudentOhneVererbung {

  String name ;
  String address;
  int matrikelNummer;
```

```

  StudentOhneVererbung(String name, String address,int nr){
    this.name = name;
    this.address = address;
    matrikelNummer = nr;
  }

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

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
```
class Student extends Person {
```

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.[1] Es gibt immer maximal eine `extends`-Klausel in einer Klassendefinition.

4.1 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:

[1] Dieses ist z.B. in C++ möglich.

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

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

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

4.2 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
```
  public String toString(){
    return    name + ", " + address
            + " Matrikel-Nr.: " + matrikelNummer;
  }
```

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.

4.3 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
```
  Student(String name,String adresse,int nr){
    super(name,adresse);
    matrikelNummer = nr;
  }
}
```

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

4.4 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
```
class TestStudent1{
  public static void main(String [] args){
    Person p
     = new Student("Martin Müller","Hauptstraße",7463456);
  }
}
```

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:

```
class StudentError1{
  public static void main(String [] args){
```



```
    Student s
      = new Person("Martin Müller","Hauptstraße");
  }
}
```

Die Kompilierung dieser Klasse führt zu folgender Fehlermeldung:

```
StudentError1.java:3: incompatible types
found   : Person
required: Student
    Student s = new Person("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
```
class TestStudent2 {

  static void printPerson(Person p){
    System.out.println(p.toString());
  }

  public static void main(String [] args){
     Student s
       = new Student("Martin Müller","Hauptstraße",754545);
     printPerson(s);
  }
}
```

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:

```
class StudentError2{

  static void printStudent(Student s){
    System.out.println(s.toString());
  }

  public static void main(String [] args){
     Person p = new Person("Martin Müller","Hauptstraße");
     printStudent(p);
  }
}
```

Auch hier führt die Kompilierung zu einer entsprechenden Fehlermeldung:

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

4.5 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

```java
class TestLateBinding {

  public static void main(String [] args){
    Student s = new Student("Martin Müller","Hauptstraße",756456);
    Person p1 = new Person("Harald Schmidt","Marktplatz");

    System.out.println(s.toString());
    System.out.println(p1.toString());

    Person p2 = new Student("Martin Müller","Hauptstraße",756456);
    System.out.println(p2.toString());
  }
}
```

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:

```java
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.[2]

Aufgabe 17 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

```java
class ButtonLogic {
  String getDescription(){
    return "in Großbuchstaben umwandeln";
  }
  String eval(String x){return x.toUpperCase();}
}
```

Dialogue.java

```java
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
class Dialogue extends JFrame{

  final ButtonLogic logic;

  final JButton button;
  final JTextField inputField  = new JTextField(20) ;
  final JTextField outputField = new JTextField(20) ;
  final JPanel p = new JPanel();

  Dialogue(ButtonLogic l){
    logic = l;
    button=new JButton(logic.getDescription());
    button.addActionListener
     (new ActionListener(){
        public void actionPerformed(ActionEvent _){
          outputField.setText
             (logic.eval(inputField.getText().trim()));
        }
      });
    p.setLayout(new BorderLayout());
    p.add(inputField,BorderLayout.NORTH);
    p.add(button,BorderLayout.CENTER);
    p.add(outputField,BorderLayout.SOUTH);
    getContentPane().add(p);
```

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

```
    pack();
    setVisible(true);
  }
}
```

TestDialogue.java

```
class TestDialogue {
  public static void main(String [] _){
    new Dialogue(new ButtonLogic());
  }
}
```

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 eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie römische Zahlen in arabische Zahlen umwandeln können. Testen Sie Ihr Programm.

d) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie arabische Zahlen in römische Zahlen umwandeln können. Testen Sie Ihr Programm.

e) Schreiben Sie jetzt ein Guiprogramm, daß eine Zahl aus ihrer Darstellung zur Basis 10 in eine Darstellung zur Basis 2 umwandelt. Testen Sie.

4.5.1 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:

```
  Dialogue(ButtonLogic l){
    logic = l;
    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.

4.6 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:

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

Der Ausdruck

```
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:

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

4.7 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.[3] 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.

Aufgrund einer Schwäche des Typsystems von Java ist man in Java oft gezwungen, in Methodensignaturen den Typ `Object` zu verwenden. Unglücklicher Weise ist die Information, daß ein Objekt vom Typ `Object` ist, wertlos. Dieses gilt ja für jedes Objekt. In kommenden Versionen von Java wird diese Schwäche behoben sein.

4.7.1 Die Methode equals

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

```
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:

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

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.

[3] Wobei wir nicht vergessen wollen, daß Daten der primitiven Typen keine Objekte sind.

4.8 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 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

```
class InstanceOfTest {
  public static void main(String [] str){
     Person p1 = new Person("Strindberg","Skandinavien");
     Person p2 = new Student("Ibsen","Skandinavien",789565);
     if (p1 instanceof Student)
       System.out.println("p1 ist ein  Student.");
     if (p2 instanceof Student)
       System.out.println("p2 ist einStudent.");
     if (p1 instanceof Person)
       System.out.println("p1 ist eine Person.");
     if (p2 instanceof Person)
       System.out.println("p2 ist eine Person.");
  }
}
```

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.
```

4.9 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

```
class CastTest {
  public static void main(String [] str){
```

```
    Person p = new Student("Ibsen","Skandinavien",789565);

    if (p instanceof Student){
      Student s = (Student)p;
      System.out.println(s.matrikelNummer);
    }
  }
}
```

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     Person p = new Person("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: Person
        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:

```
public boolean equals(Object other){
  boolean erg = false;
  if (other instanceof Person){
     Person p = (Person) other;
     erg = this.name.equals(p.name)
             && this.adresse.equals(p.adresse);
  }
  return erg;
}
```

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 5

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.

5.1 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.

5.1.1 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]

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:

$$
\begin{aligned}
head(Cons(x,xs)) &= x \\
tail(Cons(x,xs)) &= xs
\end{aligned}
$$

[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.

[2] Entgegen der Notation in Java, in der der Rückgabetyp kurioser Weise vor den Namen der Methode geschrieben wird.

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.

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:
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:

$$\begin{aligned} & concat(\mathbf{Cons}(i, \mathbf{Cons}(j, \mathbf{Empty}())), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))) \\ \rightarrow\ & \mathbf{Cons}(i, concat(\mathbf{Cons}(j, \mathbf{Empty}()), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))))) \\ \rightarrow\ & \mathbf{Cons}(i, \mathbf{Cons}(j, concat(\mathbf{Empty}(), \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}())))))) \\ \rightarrow\ & \mathbf{Cons}(i, \mathbf{Cons}(j, \mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))))) \end{aligned}$$

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 5.1.

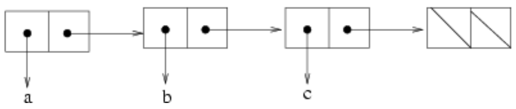

Abbildung 5.1: Schachtel Zeiger Darstellung einer dreielementigen Liste.

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:

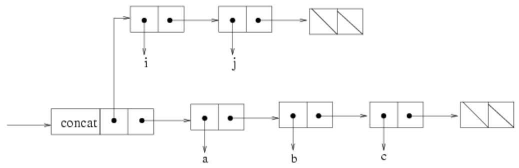

Abbildung 5.2: Schachtel Zeiger Darstellung der Funktionsanwendung von concat auf zwei Listen.

Abbildung 5.2 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.

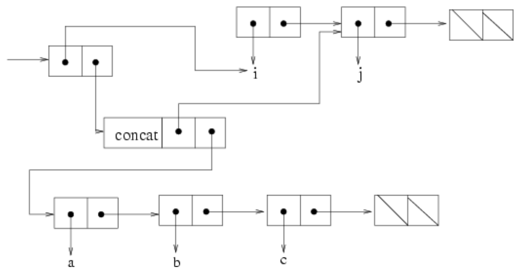

Abbildung 5.3: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

Abbildung 5.3 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.

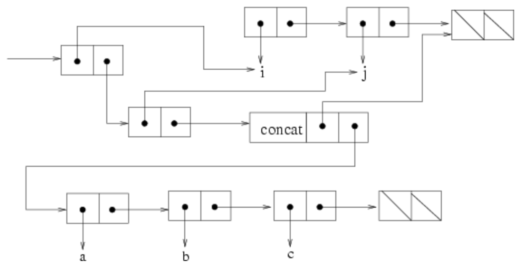

Abbildung 5.4: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

Abbildung 5.4 zeigt die Situation nach dem zweiten Reduktionsschritt. Ein weiterer neuer Listenknoten ist entstanden und ein neuer Knoten für den rekursiven Aufruf ist entstanden.

Abbildung 5.5 zeigt die endgültige Situation. Der letzte rekursive Aufruf von `concat` hatte als

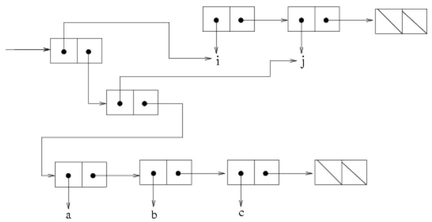

Abbildung 5.5: Schachtel Zeiger Darstellung des Ergebnisses nach der Reduktion.

erstes Argument eine leere Liste. Deshalb wurde kein neuer Listenknoten erzeugt, sondern lediglich der Knoten für die Funktionsanwendung gelöscht. Man beachte, daß die beiden 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.

5.1.2 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: 5.6.

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

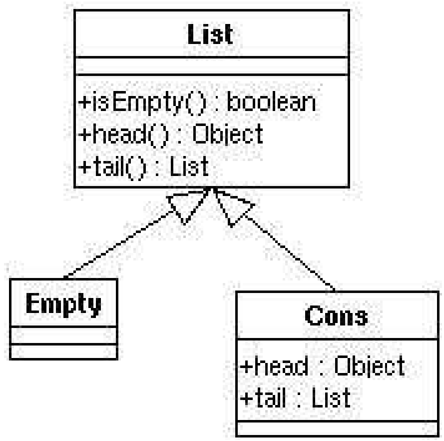

Abbildung 5.6: Modellierung von Listen mit drei Klassen.

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.

5.1.3 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

```
class List {
 boolean isEmpty(){return false;}
 Object head(){return null;}
 List 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

```
class Empty extends List {
  Empty(){}
  boolean isEmpty(){return true;}
}
```

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

```
class Cons extends List{

  Object hd;
  List tl ;

  Cons(Object x,List xs){
    hd = x;
    tl = xs;
  }

  Object head(){
    return hd;
  }

  List tail(){
    return tl;
  }
```

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

```
class TestFirstList {
  public static void main(String [] args){
```

[3] In diesem Fall generiert Java automatisch einen Konstruktor ohne Argumente, doch dazu mehr an anderer Stelle.


```

    //Konstruktion einer Testliste
    List xs = new Cons("friends",
              new Cons("romans",
              new Cons("countrymen",
              new Empty())));

    //Zugriffe auf einzelne Elemente
    System.out.println("1. Element: "+xs.head());
    System.out.println("2. Element: "+xs.tail().head());
    System.out.println("3. Element: "+xs.tail().tail().head());

    //Test für die Methode isEmpty()
    if (xs.tail().tail().tail().isEmpty()){
      System.out.println("leere Liste nach drittem Element.");
    }

    //Ausgabe eines null Wertes
    System.out.println(xs.tail().tail().tail().head());
  }
}
```

Tatsächlich bekommen wir für unsere Tests die erwartete Ausgabe:

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

```
class Li {
  boolean empty = true;
  Object  hd;
  Li      tl;

  Li  (){}

```

```
  Li (Object x,Li xs){
   hd = x;
   tl = xs;
   empty = false;
  }

  boolean isEmpty() {return empty;}
  Object  head(){return hd;}
  Li      tail(){return tl;}
```

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

```java
class TestFirstLi {
  public static void main(String [] args){

    //Konstruktion einer Testliste
    Li xs = new Li("friends",
            new Li("romans",
            new Li("countrymen",
            new Li())));

    //Zugriffe auf einzelne Elemente
    System.out.println("1. Element: "+xs.head());
    System.out.println("2. Element: "+xs.tail().head());
    System.out.println("3. Element: "+xs.tail().tail().head());

    //Test für die Methode isEmpty()
    if (xs.tail().tail().tail().isEmpty()){
      System.out.println("leere Liste nach drittem Element.");
    }

    //Ausgabe eines null Wertes
    System.out.println(xs.tail().tail().tail().head());
  }

}
```

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>
```


5.1.4 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.

$$\begin{aligned} \mathit{length}(\mathit{Empty}()) &= 0 \\ \mathit{length}(\mathit{Cons}(x, xs)) &= 1 + \mathit{length}(xs) \end{aligned}$$

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

```java
  int length(){
    if (isEmpty())return 0;
    return 1+tail().length();
  }
```

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

```java
class TestLength {
  static  Li XS = new Li("friends",new Li("romans",
                  new Li("countrymen",new Li())));

  public static void main(String [] args){
    System.out.println(XS.length());
  }
}
```

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
```
  int length(){return 0;}
}
```

In der Klasse `Cons`, die ja Listen mit einer Länge größer 0 darstellt, ist dann die Methode entsprechend zu überschreiben:

Cons.java
```
  int length(){return 1+tail().length();}
}
```

Auch diese Längenimplementierung können wir testen:

TestListLength.java
```
class TestListLength {
  static  List XS = new Cons("friends",new Cons("romans",
                    new Cons("countrymen",new Empty())));

  public static void main(String [] args){
    System.out.println(XS.length());
  }
}
```

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
```
  int lengthWhile(){
    int erg=0;
    Li xs = this;
    while (!xs.isEmpty()){
      erg= erg +1;
      xs = xs.tail();
```



```
    }
    return erg;
  }
```

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
```
  int lengthFor(){
    int erg=0;
    for (Li xs=this;!xs.isEmpty();xs=xs.tail()){
      erg = erg +1;
    }
    return erg;
  }
```

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
```
  public String toString(){
    return "("+toStringAux()+")";
  }

  private String toStringAux(){
    if (isEmpty()) return "";
    else if (tail().isEmpty()) return head().toString();
    else return head().toString()+","+tail().toStringAux();
  }
```

Aufgabe 18 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) `Object last()`: gibt das letzte Element der Liste aus.

b) `List concat(List other)` bzw.: `Li concat(Li 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) `Object elementAt(int i)`: gibt das Element an einer bestimmten Indexstelle der Liste zurück. Spezifikation:

$$\begin{aligned} elementAt(Cons(x, xs), 1) &= x \\ elementAt(Cons(x, xs), n+1) &= elementAt(xs, n) \end{aligned}$$

5.1.5 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
```java
class StringOrdering{
  static boolean lessEqual(String x,String y){
    return x.compareTo(y)<=0;
  }
}
```

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
```java
class SortStringLi extends Li {

  SortStringLi(){super();}
  SortStringLi(String x,SortStringLi xs){super(x,xs);}

  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
```java
  SortStringLi insertSorted(String x){
```

Im Falle einer leeren Liste wird die einelementige Liste zurückgegeben:

SortStringLi.java
```java
    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
```java
    else if (StringOrdering.lessEqual((String)head(),x)){
      return new SortStringLi
                  ((String)head()
                  ,((SortStringLi)tail()).insertSorted(x));
```

Anderfalls wird das neue Element mit dem Konstruktor vorne eingefügt:

SortStringLi.java
```java
    } else return new SortStringLi(x,this);
  }//method insertSorted
```

Die eigentliche Sortiermethode erzeugt eine leere Ergebnisliste, in die nacheinander die Listenelemente sortiert eingefügt werden:

SortStringLi.java
```java
  SortStringLi getSorted(){
    SortStringLi result = new SortStringLi();

    for (Li xs= this;!xs.isEmpty();xs = xs.tail()){
      result = result.insertSorted((String)xs.head());
    }

    return result;
  }//method sort
```

Somit hat die Klasse `SortStringLi` eine Sortiermethode, die wir in einer Hauptmethode testen können:

SortStringLi.java
```java
  public static void main(String [] args){
    SortStringLi xs
      = new SortStringLi("zz"
       ,new SortStringLi("ab"
       ,new SortStringLi("aaa"
       ,new SortStringLi("aaa"
```

```
      ,new SortStringLi("aaz"
      ,new SortStringLi("aya"
      ,new SortStringLi())))))));


    System.out.println("Die unsortierte Liste:");
    System.out.println(xs);

    Li ys = xs.getSorted();
    System.out.println("Die sortierte Liste:");
    System.out.println(ys);
  }
}//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]

formale Spezifikation Mathematisch läßt sich das Verfahren wie durch folgende Gleichungen beschreiben:

$$\begin{aligned} quicksort(Empty()) &= Empty() \\ quicksort(Cons(x, xs)) &= quicksort(\{y \mid y \in xs, y <= x\}) \\ &\quad ++Cons(x, quicksort(\{y \mid y \in xs, y > x\})) \end{aligned}$$

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.

[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.

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
```
class FilterCondition {
  boolean condition(Object testMe){
    return true;
  }
}
```

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 5.7:

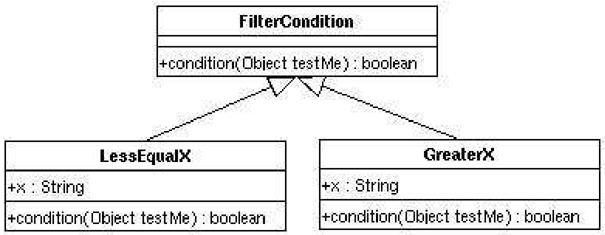

Abbildung 5.7: Modellierung der Filterbedingungen.

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:

```
Li filter(Condition cond);
```

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
```java
Li filter(FilterCondition cond){
  Li result = new Li();

  //test all elements of this list
  for (Li xs=this;!xs.isEmpty();xs=xs.tail()){

    //in case that the condition is true for the element
    if (cond.condition(xs.head())) {
      //then add it to the result
      result = new Li(xs.head(),result);
    }
  }
  return result;
}
```

Hiermit ist die Hauptarbeit für den *quick sort*-Algorithmus getan.

Beispiel:
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
```java
class LongString extends FilterCondition{
  boolean condition(Object testMe){
    return ((String)testMe).length()>10;
  }
}
```

Eine weitere Bedingung soll testen, ob ein Stringobjekt mit einem Großbuchstaben 'A' beginnt:

StringStartsWithA.java
```java
class StringStartsWithA extends FilterCondition{
  boolean condition(Object testMe){
    return ((String)testMe).charAt(0)=='A';
  }
}
```

Und eine dritte Bedingung, die wahr wird für Stringobjekte, die kein großes 'A' enthalten:

ContainsNoA.java
```java
class ContainsNoA extends FilterCondition{
  boolean condition(Object testMe){
    for (int i= 0;i<((String)testMe).length();i=i+1){
      final char c = ((String)testMe).charAt(i);
```



```java

      if (c=='A' || c=='a') return false;
    }
    return true;
  }
}
```

Probeweise filtern wir jetzt einmal eine Liste nach diesen drei Bedingungen:

TestFilter.java
```java
class TestFilter {

  static
    Li XS = new Li("Shakespeare",
            new Li("Brecht",
            new Li("Achternbusch",
            new Li("Calderon",
            new Li("Moliere",
            new Li("Sorokin",
            new Li("Schimmelpfennig",
            new Li("Kane",
            new Li("Wilde",
            new Li())))))))));

  public static void main(String [] _){
    System.out.println(XS);
    System.out.println(XS.filter(new ContainsNoA()));
    System.out.println(XS.filter(new StringStartsWithA()));
    System.out.println(XS.filter(new LongString()));
  }
}
```

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
```java
class LessEqualX extends FilterCondition{
  Comparable x;

  LessEqualX(Comparable x){
    this.x=x;
  }

  boolean condition(Object testMe){
    return x.compareTo(testMe)>0;
  }
}
```

Entsprechend für die größer-Relation:

GreaterX.java
```java
class GreaterX extends FilterCondition {
  Comparable x;

  GreaterX(Comparable x){
    this.x=x;
  }

  boolean condition(Object testMe){
    return x.compareTo(testMe)<=0;
  }
}
```

Die Methode `quicksort` läßt sich direkt aus der formalen Spezifikation ableiten:

Li.java
```java
  Li quicksort(){
    Li result = new Li();
    if (!isEmpty()){
      result
       =   //filter the smaller elements out of the tail
           tail().filter(new LessEqualX((String)head()))
            //sort these
            .quicksort()
            //concatenate it with the sorted
            //sublist of greater elements
            .append(new Li(head()
                   ,tail().filter(new GreaterX((String)head()))
                    .quicksort()
              ));
    }
    return result;
  }
```


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:

```java
Li 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
```java
class Relation {
  boolean lessEqual(Object x,Object y){
    return true;
  }
}
```

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
```java
class StringLessEqual extends Relation {
  boolean lessEqual(Object x,Object y){
    return ((String)x).compareTo((String)y)<=0;
  }
}
```

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
```java
class OrderingCondition extends FilterCondition{
  Object x;
  Relation rel;

  OrderingCondition(Object x,Relation rel){
    this.x=x;
```

```
    this.rel = rel;
  }

  boolean condition(Object y){
    return rel.lessEqual(y,x);
  }
}
```

Entsprechend für die negierte Relation:

NegativeOrderingCondition.java
```

class NegativeOrderingCondition extends FilterCondition{
  Object x;
  Relation rel;

  NegativeOrderingCondition(Object x,Relation rel){
    this.x=x;
    this.rel = rel;
  }

  boolean condition(Object 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 sortBy(Relation rel){
    Li result = new Li();
    if (!isEmpty()){
      FilterCondition le
       = new OrderingCondition(head(),rel);
      FilterCondition gr
       = new NegativeOrderingCondition(head(),rel);
      result = tail()
                .filter(le)
                .sortBy(rel)
                .append(new Li(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:
Im folgenden Beispiel werden Strings einmal nach ihrer lexikographischen Ordnung, einmal nach ihrer Länge sortiert:

StringLengthLessEqual.java
```
class StringLengthLessEqual extends Relation {
  boolean lessEqual(Object x,Object y){
    return ((String)x).length()<= ((String)y).length();
  }
}
```

In der Testmethode können wir die Methode `sortBy` jeweils mit einer der Bedingungen aufrufen:

TestSortBy.java
```
class TestSortBy{
  public static void main(String [] args){
    Li xs = TestFilter.XS;

    System.out.println("Die unsortierte Liste:");
    System.out.println(xs);
    System.out.println("Die alphabetisch sortierte Liste:");
    System.out.println(xs.sortBy(new StringLessEqual()));
    System.out.println("Die nach der Länge sortierte Liste:");
    System.out.println(xs.sortBy(new StringLengthLessEqual()));
  }
}
```

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 19 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 20 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

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.basic.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

public class HtmlView extends JPanel {

  //example invocation
  public static void main(String s[]) {
    HtmlView view = new HtmlView();
    view.run();
    view.setText("<h1>hallo</h1>");
  }

  JFrame frame;
  JTextPane ausgabe = new JTextPane();
```



```

  public HtmlView() {
    ausgabe.setEditorKit(new HTMLEditorKit());
    add(ausgabe);
  }

  void setText(String htmlString){
    ausgabe.setText(htmlString);
    frame.pack();
    ausgabe.repaint();
  }

  void run(){
    frame = new JFrame("HtmlView");
    frame.getContentPane().add(this);
    frame.pack();
    frame.setVisible(true);
  }
}
```

Bubble Sort

Der sogenannte *bubble sort*-Algorithmus ist nicht unbedingt geeignet für Listen mit dynamischer Länge, sondern für Reihungen mit fester Länge *(arrays)*, die wir in einem späteren Kapitel kennenlernen werden. Daher werden wir diesen Algorithmus erst an späterer Stelle betrachten.

5.1.6 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:
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:

$$length(concat(xs, ys)) = length(xs) + length(ys)$$

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:

  $$length(concat(Empty(), ys)) = length(Empty()) + length(ys)$$

  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.

Kapitel 6

Weiterführende programmiersprachliche Konzepte

6.1 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.

6.1.1 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:

```
package testPackage;
class MyClass {
}
```

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

TestPaket.java

```
package de.tfhberlin.panitz.testPackages;
class TestPaket {
  public static void main(String [] args){
```



```
    System.out.println("hello from package \'testpackages\'");
  }
}
```

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 `de.tfhberlin.panitz.testPackages` schreibt. Paket- und Klassenname zusammen durch einen Punkt getrennt werden der *vollqualifizierte Name* der Klasse genannt, im obigen Beispiel ist entsprechend der vollqualifizierte Name:
`de.tfhberlin.panitz.testPackages.Test`

Der Name einer Klasse ohne die Paketnennung heißt unqualifiziert.

6.1.2 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.

[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.

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

6.1.3 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 de.tfhberlin.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:
`de.tfhberlin.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.

6.1.4 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.

6.1.5 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` aus dem Paket `java.util`.

TestArrayList.java

```
package de.tfhberlin.panitz.utilTest;
class TestArrayList {
  public static void main(String [] args){
    java.util.ArrayList xs = new java.util.ArrayList();
    xs.add("friends");
    xs.add("romans");
    xs.add("countrymen");
    System.out.println(xs);
  }
}
```

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

6.1.6 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 benutzt werden kann:

TestImport.java

```
package de.tfhberlin.panitz.utilTest;

import java.util.ArrayList;

class TestImport {
  public static void main(String [] args){
    ArrayList xs = new ArrayList();
    xs.add("friends");
    xs.add("romans");
    xs.add("countrymen");
    System.out.println(xs);
  }
}
```

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

TestImport2.java

```
package de.tfhberlin.panitz.utilTest;

import java.util.ArrayList;
import java.util.Vector;

class TestImport2 {
  public static void main(String [] args){
    ArrayList xs = new ArrayList();
    xs.add("friends");
    xs.add("romans");
    xs.add("countrymen");
    System.out.println(xs);

    Vector ys = new Vector();
    ys.add("friends");
    ys.add("romans");
    ys.add("countrymen");
    System.out.println(ys);
  }
}
```

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
```
package de.tfhberlin.panitz.utilTest;

import java.util.*;

class TestImport3 {
  public static void main(String [] args){
    List xs = new ArrayList();
    xs.add("friends");
    System.out.println(xs);

    Vector ys = new Vector();
    ys.add("romans");
    System.out.println(ys);
  }
}
```

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

6.2 Sichtbarkeitsattribute

Sichtbarkeiten[3] erlauben es, zu kontrollieren, wer auf Klassen und ihre Eigenschaften zugreifen kann. Das *wer* bezieht sich hierbei auf andere Klassen und Pakete.

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

MyPublicClass.java
```
package de.tfhberlin.panitz.p1;
public class MyPublicClass {
}
```

MyNonPublicClass.java
```
package de.tfhberlin.panitz.p1;
class MyNonPublicClass {
}
```

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

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

UsePublic.java
```
package de.tfhberlin.panitz.p2;

import de.tfhberlin.panitz.p1.*;

class UsePublic {
  public static void main(String [] args){
    System.out.println(new MyPublicClass());
  }
}
```

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

```
package de.tfhberlin.panitz.p2;

import de.tfhberlin.panitz.p1.*;

class UseNonPublic {
  public static void main(String [] args){
    System.out.println(new MyNonPublicClass());
  }
}
```

Java gibt bei der Übersetzung eine entsprechende gut verständliche Fehlermeldung:

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

6.2.2 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

```java
package de.tfhberlin.panitz.p1;

public class VisibilityOfFeatures{
  private   String s1 = "private";
            String s2 = "package";
  protected String s3 = "protected";
  public    String s4 = "private";

  public static void main(String [] args){
    VisibilityOfFeatures v = new VisibilityOfFeatures();
    System.out.println(v.s1);
    System.out.println(v.s2);
    System.out.println(v.s3);
    System.out.println(v.s4);
  }
}
```

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

```java
package de.tfhberlin.panitz.p1;

public class PrivateTest
```



```java
{

  public static void main(String [] args){
    VisibilityOfFeatures v = new VisibilityOfFeatures();
    //s1 is private and cannot be accessed;
    //we are in a  different class.
    //System.out.println(v.s1);
    System.out.println(v.s2);
    System.out.println(v.s3);
    System.out.println(v.s4);
  }
}
```

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` nicht mehr benutzt werden:

PackageTest.java

```java
package de.tfhberlin.panitz.p2;
import de.tfhberlin.panitz.p1.VisibilityOfFeatures;

public class  PackageTest extends VisibilityOfFeatures{

  public static void main(String [] args){
    PackageTest v = new PackageTest();
    //s1 is private and cannot be accessed
    // System.out.println(v.s1);

    //s2 is package visible and cannot be accessed;
    //we are in a different package.
    //System.out.println(v.s2);

    System.out.println(v.s3);
    System.out.println(v.s4);
  }
}
```

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

ProtectedTest.java

```java
package de.tfhberlin.panitz.p2;
import de.tfhberlin.panitz.p1.VisibilityOfFeatures;

public class ProtectedTest  {

  public static void main(String [] args){
    VisibilityOfFeatures v = new VisibilityOfFeatures();
    //s1 is private and cannot be accessed
    // System.out.println(v.s1);
```

```

    //s2 is package visible and cannot be accessed. We are
    //in a different package
    //System.out.println(v.s2);

    //s2 is protected and cannot be accessed.
    //We are not a subclass
    //System.out.println(v.s3);

    System.out.println(v.s4);
  }
}
```

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.

6.2.3 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. get-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:
Eine kleine Klasse mit Kapselung eines privaten Feldes:

```
package de.tfhberlin.sep.skript;

public class GetSetMethod {
  private int value=0;

  public void setValue(int newValue) {value=newValue;}
  public int  getvalue(){return value;}
}
```


6.2.4 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:

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

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.

6.2.5 Unveränderbare Listen

Mit den Sichtbarkeitsattributen können wir jetzt auch sicherstellen, daß für unsere Listenimplementierung einmal erzeugte Listen unveränderbar bleiben. Hierzu setzen wir die internen drei Felder der Listen einfach als privat. Insgesamt erhalten wir dadurch die folgende Klasse für Listen:

InmutableList.java

```
class InmutableList {
  private boolean empty = true;
  private Object  hd;
  private InmutableList tl;

  public InmutableList(){}
  public InmutableList(Object x,InmutableList xs){
   hd = x;
   tl = xs;
   empty = false;
  }

  public boolean isEmpty() {return empty;}
  public Object  head(){return hd;}
  public InmutableList tail(){return tl;}

```

```
  public int length(){
    if (isEmpty())return 0;
    return 1+tail().length();
  }

  public String toString(){
    return "("+toStringAux()+")";
  }
  private String toStringAux(){
    if (isEmpty()) return "";
    else if (tail().isEmpty()) return head().toString();
    else return head().toString()+","+tail().toStringAux();
  }

  public InmutableList append(InmutableList ys){
    if (isEmpty()) return ys;
    return new InmutableList(head(),tail().append(ys));
  }

  public InmutableList filter(FilterCondition cond){
    InmutableList result = new InmutableList();

    for (InmutableList xs=this;!xs.isEmpty();xs=xs.tail()){
      if (cond.condition(xs.head())) {
        result = new InmutableList(xs.head(),result);
      }
    }
    return result;
  }

  public InmutableList sortBy(Relation rel){
    if (!isEmpty()){
      final FilterCondition le
       = new OrderingCondition(head(),rel);
      final FilterCondition gr
       = new NegativeOrderingCondition(head(),rel);
      final InmutableList smaller = tail().filter(le).sortBy(rel);
      final InmutableList greater = tail().filter(gr).sortBy(rel);
      return smaller.append(new InmutableList(head(),greater));
    }
    return new InmutableList();
  }
}
```

6.3 Schnittstellen (Interfaces) und abstrakte Klassen

Wir haben schon einige 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. Beispiele für solche Klassen waren die Sortierrelationen `Relation` in den Sortieralgorithmen oder auch 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.

6.3.1 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

```
package de.tfhberlin.panitz.dialoguegui;

public interface DialogueLogic {
  public String getDescription();
  public String eval(String input);
}
```

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.

Eine mögliche Implementierung der obigen Schnittstelle ist:

ToUpperCase.java

```
package de.tfhberlin.panitz.dialoguegui;

public class ToUpperCase implements DialogueLogic{
```

```
  protected String result;

  public String getDescription(){
    return "convert into upper cases";
  }
  public String eval(String input){
    result = input.toUpperCase();
    return result;
  }
}
```

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
```
package de.tfhberlin.panitz.html;

public interface ToHTMLString {
  public String toHTMLString();
}
```

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
```
package de.tfhberlin.panitz.dialoguegui;

import de.tfhberlin.panitz.html.*;

public class ToUpper extends    ToUpperCase
                     implements ToHTMLString, DialogueLogic {

  public String toHTMLString(){
     return   "<html><head><title>"+getDescription()
            + "</title></head>"
            + "<body><b>Small Gui application</b>"
            + " for convertion of "
            + " a <b>String</b> into <em>upper</em>"
            + " case letters.<br></br>"
            + "The result of your query was: <p>"
            + "<span style=\"font-family: monospace;\">"
            + result
            + "</span></p></body></html>";
  }
}
```


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
```
package de.tfhberlin.panitz.dialoguegui;

import de.tfhberlin.panitz.html.*;

public interface DialogueLogics
        extends ToHTMLString, DialogueLogic {}
```

Ebenso können wir jetzt eine Klasse ableiten, die diese Schnittstelle implementiert:

UpperConversion.java
```
package de.tfhberlin.panitz.dialoguegui;

class UpperConversion extends ToUpper
                      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
```
package de.tfhberlin.panitz.dialoguegui;
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.plaf.basic.*;
import javax.swing.text.*;
```

```
import javax.swing.text.html.*;

public class HtmlDialogue extends JFrame{
  final DialogueLogics logic;
  final JButton button;
  final JTextField inputField = new JTextField(20) ;
  final JTextPane outputField = new JTextPane();
  final JPanel p = new JPanel();

  public HtmlDialogue(DialogueLogics l){
    outputField.setEditorKit(new HTMLEditorKit());
    logic = l;
    button=new JButton(logic.getDescription());
    button.addActionListener
     (new ActionListener(){
        public void actionPerformed(ActionEvent _){
          logic.eval(inputField.getText().trim());
          outputField.setText(logic.toHTMLString());
          pack();
        }
      });

    p.setLayout(new BorderLayout());
    p.add(inputField,BorderLayout.NORTH);
    p.add(button,BorderLayout.CENTER);
    p.add(outputField,BorderLayout.SOUTH);
    getContentPane().add(p);
    pack();
    setVisible(true);
  }
}
```

Schließlich können wir ein Objekt der Klasse `UpperConversion`, die die Schnittstelle `DialogueLogics` implementiert, konstruieren und der Gui-Anwendung übergeben:

HtmlDialogueTest.java
```
package de.tfhberlin.panitz.dialoguegui;
public class HtmlDialogueTest {
  public static void main(String [] args){
    new HtmlDialogue(new UpperConversion());
  }
}
```

Die Anwendung in voller Aktion kann in Abbildung 6.1 bewundert werden.

Semantische Einschränkungen für Schnittstellen

Es gibt einige semantische Einschränkungen, die über die syntaktischen Einschränkungen hinausgehen:

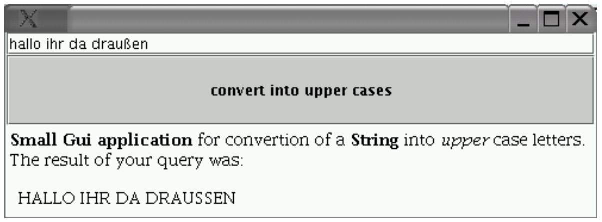

Abbildung 6.1: Ein Gui-Dialog mit Html-Ausgabe.

- 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`.

Iteratorschnittstellen

Wir kennen bereits das Programmierprinzip der Iteration, das wir benutzen, indem wir mit einem Schleifenkonstrukt für die Werte einer Liste einen bestimmten Codeblock wiederholt ausführen. In Java werden für die Iteration mit einer `for`-Schleife häufig bestimmte Iteratorobjekte benutzt.[4] Ein Iteratorobjekt kennzeichnet sich durch zwei Methoden:

- einer Methode `next`, die jeweils das nächste Element der Iteration zurückgibt.
- einer Methode `hasNext`, die in einem bool'schen Rückgabewert angibt, ob es weitere Elemente gibt, die über `next` erfragt werden könnten.

Die Funktionalität eines Iteratorobjekts läßt sich über eine Schnittstellendefinition gut beschreiben.[5]

Wir sehen einen Iterator vor, dessen Elemente vom primitiven Typ `int` sind:

IntIterator.java
```
package de.tfhberlin.panitz.iterator;
public interface IntIterator{
  public int next();
  public boolean hasNext();
}
```

[4] Mit der Version 1.5 von Java werden diese Iteratorobjekte im Zusammenhang mit der `for`-Schleife nocheinmal gesteigerte Bedeutung bekommen.
[5] Und das wird auch tatsächlich in Javas Standardklassen so gemacht.

Iteratoren sind also Objekte, von denen so lange, bis die Methode `hasNext()` einmal `false` zurückgibt, nach einen nächsten Element gefragt werden kann:

PrintIntIteratorElements.java
```java
package de.tfhberlin.panitz.iterator;
public class PrintIntIteratorElements{
  static void print(IntIterator it){
    while (it.hasNext()) System.out.print(it.next()+" ");
  }
}
```

Jetzt können wir verschiedene Iteratoren für `int`-Zahlen schreiben. Ein typischer Iterator soll uns, angefangen von einem Anfangswert bis zu einem Endwert, nacheinander Zahlen geben, und zwar in einer bestimmten Schrittweite:

FromToStep.java
```java
package de.tfhberlin.panitz.iterator;
public class FromToStep implements IntIterator{
  private int from;
  private int to;
  private int step;

  public FromToStep(int from,int to,int step){
    this.from=from;this.to=to;this.step=step;}

  public boolean hasNext(){return from<=to;}
  public int next(){
    int result = from;
    from=from+step;
    return result;
  }
}
```

Wollen wir immer nur um eins weiterzählen, so können wir eine Subklasse schreiben, die die Schrittweite auf `1` setzt:

FromTo.java
```java
package de.tfhberlin.panitz.iterator;
public class FromTo extends FromToStep{
  public FromTo(int from,int to){super(from,to,1);}
}
```

Wir können einmal versuchen, diesen Iterator mit unserer `print`-Methode auszugeben:

TestFromTo.java
```java
package de.tfhberlin.panitz.iterator;
public class TestFromTo {
  public static void main(String []_){
    PrintIntIteratorElements.print(new FromTo(17,42));
  }
}
```



```
sep@linux:~/fh/prog1/examples/src> java de.tfhberlin.panitz.iterator.TestFromTo
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
sep@linux:~/fh/prog1/examples/src>
```

Aufgabe 21 In dieser Aufgabe sollen Sie ein Programm schreiben, das nach und nach die Primzahlen ausgibt.

a) Schreiben sie eine Unterklasse `From` der Klasse `FromTo`, die von einem Startwert an in Einerschritten ganze Zahlen zurückgibt und für die `hasNext` immer wahr ist.

b) Schreiben Sie eine Klasse `Sieb`, die die Schnittstelle `IntIterator` implementiert. Die Klasse soll zwei Felder haben: eine ganze Zahl und ein weiteres Objekt, das die Schnittstelle `IntIterator` implementiert. Der Konstruktor habe die Signatur:
`public Sieb(int i,IntIterator it)`
Die Methode `next` soll das nächste Element des inneren Iterators `it` zurückgeben, das nicht durch die Zahl `i` teilbar ist.

Beispiel: `PrintIntIteratorElements.print(new Sieb(3,new From(1)))` gibt alle nicht durch 3 teilbaren natürlichen Zahlen auf dem Bildschirm aus.

c) Übersetzen Sie ihren Code mit folgender Klasse:

```java
package de.tfhberlin.panitz.iterator;
class PrimIterator implements IntIterator{
  private IntIterator sieb = new From(2);
  public boolean hasNext(){return sieb.hasNext();}
  public int next(){
    int result = sieb.next();
    sieb= new Sieb(result,sieb);
    return result;
  }
}
```

Lassen Sie sich alle Werte von `PrimIterator` mit der Methode `PrintIntIteratorElements.print` ausgeben.

6.3.2 Abstrakte Klassen

Im vorangegangenen Abschnitt haben wir zusätzlich zu Klassen noch das Konzept der Schnittstellen kennengelernt. Klassen enthalten Methoden und Implementierungen für die Methoden. Jede Methode hat einen Rumpf. Schnittstellen enthalten nur Methodensignaturen. Keine Methode einer Schnittstelle hat eine Implementierung. Es gibt keine Methodenrümpfe. Java kennt noch eine Mischform zwischen Klassen und Schnittstellen: abstrakte Klassen.

Definition abstrakter Klassen

Eine Klasse wird als abstrakt deklariert, indem dem Schlüssenwort `class` das Schlüsselwort `abstract` vorangestellt wird. Eine abstrakte Klasse kann nun Methoden mit und Methoden ohne Rumpf enthalten. Methoden, die in einer abstrakten Klassen keine Implementierung enthalten, sind mit dem Attribut `abstract` zu kennzeichnen. Für abstrakte Klassen gilt also, daß bestimmte Methoden bereits implementiert sind und andere Methoden in Unterklassen zu implementieren sind.

Als Beispiel können wir Listen mit Hilfe einer abstrakten Klasse implementieren:

AbstractList.java

```java
package de.tfhberlin.sep.abstractList;
abstract class AbstractList  {
  abstract public boolean isEmpty();

  abstract public Object head();
  abstract public AbstractList tail();

  abstract public AbstractList empty();
  abstract public AbstractList cons(Object x,AbstractList xs);

  public int length(){
    if (isEmpty()) return 0;
    return 1+tail().length();
  }

  public AbstractList concat(AbstractList other){
    if (isEmpty()) return this;
    return cons(head(),tail().concat(other));
  }
}
```

Die abstrakte Klasse für Listen definiert die Signaturen für die fünf Methoden, die in der Spezifikation des Listendatentyps spezifiziert wurden: die beiden Selektormethoden `head` und `tail`, die Testmethode `isEmpty` und die zwei Konstruktoren `cons` und `tail`.

Da abstrakte Klassen nicht für alle ihre Eigenschaften Implementierungen haben, gilt für abstrakte Klassen ebenso wie für Schnittstellen, daß sie keinen Konstruktor haben, d.h. sie nie direkt konstruiert werden können, sondern immer nur durch eine Unterklasse, durch die die fehlenden Implementierungen ergänzt werden, instanziiert werden können.

Da der Name eines Konstruktors an den Namen einer Klasse gebunden ist, können wir in der abstrakten Klasse keine abstrakten Konstruktoren schreiben. Wir definieren daher zwei Methoden, die für den Konstruktor der entsprechenden Unterklassen stehen sollen. Dieses hat den Vorteil, daß wir für alle Unterklassen Konstruktormethoden mit gleichen Namen vorschreiben können. Solche Methoden, die quasi einen Konstruktor beschreiben, nennt man Fabrikmethoden. Im obigen Beispiel haben wir gemäß der Listenspezifikation zwei abstrakte Methoden zur Konstruktion von Listen definiert: `cons` und `empty`.

Für Listen sind in der abstrakten Listenklasse bereits zwei Methoden implementiert: die Methoden `length` und `concat`.

In diesen Methoden können die abstrakten Methoden für Listen bereits benutzt werden, obwohl diese in der abstrakten Klassen noch nicht implementiert wurden. Im Falle eines konkreten Aufrufs dieser Methoden, wird dieses auf einem Objekt einer konkreten, nicht abstrakten Klasse geschehen und dann liegen Implementierungen der abstrakten Methoden vor. Somit können wir also die Methoden `isEmpty`, `head`, `tail` und sogar die Fabrikmethoden `empty` und `cons` in der Implementierung der Methoden `length` und `concat` benutzen.

Implementierende Unterklassen abstrakter Klassen

In der Ableitungshierarchie verhalten sich abstrakte Klassen wie jede andere Klasse auch, d.h. sie können mittels der`extends`-Klausel durch andere Klassen erweitert werden und selbst mit `implements`- und `extends`-Klauseln erweitern. Weiterhin gilt, daß eine Klasse maximal von einer Klasse direkt ableiten darf, auch wenn diese Klasse abstrakt ist.

Entsprechend können wir eine Unterklasse der abstrakten Listenklasse implementieren, die nicht mehr abstrakt ist. Hierzu sind für die fünf abstrakten Methoden konkrete Implementierungen zu schreiben:

LinkedList.java

```java
package de.tfhberlin.sep.abstractList;
public class LinkedList extends AbstractList {

  private Object hd=null;
  private AbstractList tl=null;
  private boolean empty;

  public LinkedList(){empty=true;}
  public LinkedList(Object x,AbstractList xs){
    hd=x;tl=xs;empty=false;
  }

  public boolean isEmpty(){return empty;}

  public Object head(){return hd;}
  public AbstractList tail(){return tl;}

  public AbstractList empty(){return new LinkedList();}
  public AbstractList cons(Object x,AbstractList xs){
     return new LinkedList(x,xs);
   }

}
```

Die Implementierungen entsprechen der Implementierung, die wir bereits in der Klasse `Li` gewählt hatten. Zusätzlich waren wir gezwungen, die beiden Fabrikmethoden zu implementieren. Wie man sieht, bilden die Fabrikmethoden direkt auf die beiden Konstruktoren der Klasse ab.

Abstrakte Klassen und Schnittstellen

Abstrakte Klassen können auch Schnittstellen implementieren. Entgegen konkreter Klassen brauchen sie aber nicht alle Eigenschaften einer Schnittstelle zu implementieren. Erst eine konkrete Klasse muß alle abstrakten Eigenschaften, die sie auf irgendeinem Weg erbt, implementieren. Ein typisches Szenario für das Zusammenspiel von Schnittstellen und abstrakten Klassen können wir für unsere Listenimplementierung angeben. Eine Schnittstelle beschreibt zunächst die für Listen relevanten Methoden:

```
package de.tfhberlin.sep.listen;
public interface List  {
  boolean isEmpty();

  Object head();
  AbstractList tail();

  public int length();
  public AbstractList concat(AbstractList other);
}
```

Die abstrakte Klasse dient als eine Implementierung all der Methoden, die für die Implementierung von Listen einheitlich ausgedrückt werden können. Eine solche abstrakte Klasse ist als Hilfsklasse zu sehen, die schon einmal zusammenfasst, was alle implementierenden Listenklassen gemeinsam haben:

```
package de.tfhberlin.sep.listen;
public abstract class AbstractList  implements List{

  abstract public AbstractList empty();
  abstract public AbstractList cons(Object x,AbstractList xs);

  public int length(){
    if (isEmpty()) return 0;
    return 1+tail().length();
  }

  public AbstractList concat(AbstractList other){
    if (isEmpty()) return this;
    return cons(head(),tail().concat(other));
  }
}
```

Wie man sieht, brauchen die Methoden aus der Schnittstelle `List` nicht implementiert zu werden. Ausprogrammiert sind bereits die Methoden, die allein auf den fünf spezifizierten Listenmethoden basieren.

In konkreten Klassen schließlich findet sich eine eigentliche technische Umsetzung der Listenstruktur, wie wir es oben an der Klasse `LinkedList` exemplarisch programmiert haben.

Aufgabe 22 (4 Punkte)
In dieser Aufgabe wird ein kleines Programm, das einen Psychoanalytiker simuliert, vervollständigt.

a) Laden Sie sich hierzu das Archiv
`Eliza.zip` (`http://www.tfh-berlin.de/~panitz/prog1/load/Eliza.zip`)
vom Netz. Entpacken Sie es. Machen Sie sich mit den einzelnen Klassen vertraut.

b) In der Klasse `MyList` sind nicht alle abstrakten Methoden der Klasse `Li` implementiert. Ergänzen Sie `MyList` um die Methoden:
`reverse`, `words`, `unwords`, `drop`, `tails`, `isPrefixIgnoreCaseOf`. Implementieren Sie diese Methoden entsprechend ihrer Dokumentation in der abstrakten Klasse `Li` und schreiben Sie Tests für jede Methode.

Wenn Ihre Tests erfolgreich sind, übersetzen Sie alle Klassen und starten Sie die `main`-Methode der Klasse `Main`.

c) Erfinden Sie eigene Einträge für die Liste `respMsgs` in der Klasse `Data`.

d) Erklären Sie, was die Methode `rotate` von den anderen Methoden der Klasse `Li` fundamental unterscheidet. Demonstrieren Sie dieses anhand eines Tests.

6.4 Ausnahme- und Fehlerbehandlung

Es gibt während des Ablaufs eines Programmes Situationen, die als Ausnahmen zum eigentlichen Programmablauf betrachtet werden können. Eine typische solche Situation könnte z.B. die Methode `head` auf Listen sein. Im Normalfall gibt diese Methode das vorderste Listenelement zurück. Eine Ausnahmefall ist, wenn dieses Element nicht existiert, weil die Liste leer ist. Java hält ein Konzept bereit, das die Behandlung von Ausnahmen abseits der eigentlichen Programmlogik erlaubt.

6.4.1 Ausnahme- und Fehlerklassen

Java stellt Standardklassen zur Verfügung, deren Objekte einen bestimmten Ausnahme- oder Fehlerfall ausdrücken. Die gemeinsame Oberklasse aller Klassen, die Fehler- oder Ausnahmefälle ausdrücken, ist `java.lang.Throwable`. Diese Klasse hat zwei Unterklassen, nämlich:

- `java.lang.Error`: alle Objekte dieser Klasse drücken aus, daß ein ernsthafter Fehlerfall aufgetreten ist, der in der Regel von dem Programm selbst nicht zu beheben ist.
- `java.lang.Exception`: alle Objekte dieser Klasse stellen Ausnahmesituationen dar. Im Programm kann eventuell beschrieben sein, wie bei einer solchen Ausnahmesituation weiter zu verfahren ist. Eine Unterklasse von `Exception` ist die Klasse `java.lang.RuntimeException`.

6.4.2 Werfen von Ausnahmen

Ein Objekt vom Typ `Throwable` allein zeigt noch nicht an, daß ein Fehler aufgetreten ist. Hierzu gibt es einen speziellen Befehl, der im Programmablauf dieses kennzeichnet, der Befehl `throw`.

`throw` ist ein Schlüsselwort, dem ein Objekt des Typs `Throwable` folgt. Bei einem `throw`-Befehl verläßt Java die eigentliche Ausführungsreihenfolge des Programms und unterrichtet die virtuelle Maschine davon, daß eine Ausnahme aufgetreten ist. Z.B. können wir für die Fakultätsmethoden bei einem Aufruf mit einer negativen Zahl eine Ausnahme werfen:

FirstThrow.java
```
1  package de.tfhberlin.panitz.exceptions;
2  public class FirstThrow {
3
4    public static int fakultät(int n){
5      if (n==0) return 1;
6      if (n<0) throw new RuntimeException();
7      return n*fakultät(n-1);
8    }
9
10   public static void main(String [] args){
11     System.out.println(fakultät(5));
12     System.out.println(fakultät(-3));
13     System.out.println(fakultät(4));
14   }
15 }
```

Wenn wir dieses Programm starten, dann sehen wir, daß zunächst die Fakultät für die Zahl 5 korrekt berechnet und ausgegeben wird, dann der Fehlerfall auftritt, was dazu führt, daß der Fehler auf der Kommandozeile ausgegeben wird und das Programm sofort beendet wird. Die Berechnung der Fakultät von 4 wird nicht mehr durchgeführt. Es kommt zu folgender Ausgabe:

```
swe10:~> java de.tfhberlin.panitz.exceptions.FirstThrow
120
Exception in thread "main" java.lang.RuntimeException
  at de.tfhberlin.panitz.exceptions.FirstThrow.fakultät(FirstThrow.java:6)
  at de.tfhberlin.panitz.exceptions.FirstThrow.main(FirstThrow.java:12)
swe10:~>
```

Wie man sieht, unterrichtet uns Java in der ersten Zeile davon, daß eine Ausnahme des Typs `RuntimeException` geworfen wurde. In der zweiten Zeile erfahren wir, daß dieses bei der Ausführung der Methode `fakultät` in Zeile 6 der Klasse `FirstThrow` geschehen ist. Anschließend, in den Zeilen weiter unten, gibt Java jeweils an, in welcher Methode der Aufruf der in der drüberliegenden Methode stattfand.

Die Ausgabe gibt also an, durch welchen verschachtelten Methodenaufruf es an die Stelle kam, in der die Ausnahme geworfen wurde. Diese Aufrufstruktur wird als Aufrufkeller (*stack trace*) bezeichnet.

Das Erzeugen eines Ausnahmeobjekts allein bedeutet noch keinen Fehlerfall. Wenn wir das obige Programm minimal ändern, so daß wir das Schlüsselwort `throw` weglassen, so wird der Sonderfall für negative Eingaben nicht gesondert behandelt.

NonThrow.java
```
package de.tfhberlin.panitz.exceptions;
public class NonThrow {

  public static int fakultät(int n){
    if (n==0) return 1;
    if (n<0) new RuntimeException();
    return n*fakultät(n-1);
```



```
  }

  public static void main(String [] args){
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Wenn wir dieses Programm starten, so wird es nicht terminieren und je nach benutzter Javamaschine schließlich abbrechen:

```
swe10:~> java de.tfhberlin.panitz.exceptions.NonThrow
120

An irrecoverable stack overflow has occurred.
```

Es reicht also nicht aus, ein Fehlerobjekt zu erzeugen, sondern es muß dieses auch mit einem `throw`-Befehl geworfen werden. Geworfen werden können alle Objekte einer Unterklasse von `Throwable`. Versucht man hingegen, andere Objekte zu werfen, so führt dies schon zu einem Übersetzungsfehler.

Folgende Klasse:

```
package de.tfhberlin.panitz.exceptions;
public class NotThrowable {

  public static void main(String [] args){
      throw "i am not throwable";
  }
}
```

führt zu einem Übersetzungsfehler:

```
swe10:~> javac -d . NotThrowable.java
NotThrowable.java:5: incompatible types
found   : java.lang.String
required: java.lang.Throwable
      throw "i am not throwable";
            ^
1 error
swe10:~>
```

Ausnahmen können natürlich nicht nur auftreten, wenn wir sie selbst explizit geworfen haben, sondern auch von Methoden aus Klassen, die wir selbst benutzen, geworfen werden. So kann z.B. die Benutzung der Methode `charAt` aus der Klasse `String` dazu führen, daß eine Ausnahme geworfen wird.

ThrowIndex.java
```
package de.tfhberlin.panitz.exceptions;
public class ThrowIndex {
```

```

  public static void main(String [] args){
      "i am too short".charAt(120);
  }
}
```

Starten wir dieses Programm, so wird auch eine Ausnahme geworfen:

```
swe10:~> java de.tfhberlin.panitz.exceptions.ThrowIndex
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
                                    String index out of range: 120
  at java.lang.String.charAt(String.java:516)
  at de.tfhberlin.panitz.exceptions.ThrowIndex.main(ThrowIndex.java:5)
swe10:~>
```

Wie man an diesem Beispiel sieht, gibt Java nicht nur die Klasse der Ausnahme, die geworfen wurde, aus (`java.lang.StringIndexOutOfBoundsException:`), sondern auch noch eine zusätzliche Erklärung. Die Objekte der Unterklassen von `Throwable` haben in der Regel einen Konstruktor, der erlaubt noch eine zusätzliche Information, die den Fehler erklärt, mit anzugeben. Das können wir auch in unserem Beispielprogramm nutzen:

SecondThrow.java

```
package de.tfhberlin.panitz.exceptions;

public class SecondThrow {
  public static int fakultät(int n){
    if (n==0) return 1;
    if (n<0)
     throw
      new RuntimeException
           ("negative Zahl für Fakultätsberechnung");
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Damit erhalten wir folgende Ausgabe:

```
swe10:~> java de.tfhberlin.panitz.exceptions.SecondThrow
120
Exception in thread "main" java.lang.RuntimeException:
                                    negative Zahl für Fakultätsberechnung
  at de.tfhberlin.panitz.exceptions.SecondThrow.fakultät(SecondThrow.java:6)
  at de.tfhberlin.panitz.exceptions.SecondThrow.main(SecondThrow.java:12)
swe10:~>
```


6.4.3 Deklaration von geworfenen Ausnahmen

Um sich auf Ausnahmefälle einzustellen, ist notwendig, daß einer Methode angesehen werden kann, ob sie bei der Ausführung eventuell eine Ausnahme werfen wird. Java bietet an, dieses in der Signatur der Methoden zu schreiben. Java bietet dieses nicht nur an, sondern schreibt sogar zwingend vor, daß alle Ausnahmeobjekte, die in einer Methode geworfen werden, auch in der Signatur der Methode angegeben sind. Einzig davon ausgenommen sind Objekte des Typs `RuntimeException`. Wollen wir in unserem obigen Programm eine andere Ausnahme werfen als eine `RuntimeException`, so können wir das zunächst nicht:

```
package de.tfhberlin.panitz.exceptions;
public class ThirdThrow {

  public static int fakultät(int n){
    if (n==0) return 1;
    if (n<0) throw new Exception
             ("negative Zahl für Fakultätsberechnung");
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Bei der Übersetzung kommt es zu folgendem Fehler:

```
swe10:~> javac -d . ThirdThrow.java
ThirdThrow.java:6: unreported exception java.lang.Exception;
                                    must be caught or declared to be thrown
    if (n<0) throw new Exception("negative Zahl für Fakultätsberechnung");
             ^
1 error
```

Java verlangt, daß wir für die Methode `fakultät` in der Signatur angeben, daß die Methode eine Ausnahme wirft. Dieses geschieht durch eine `throws`-Klausel zwischen Signatur und Rumpf der Methode. Dem Schlüsselwort `throws` folgen dabei durch Kommas getrennt die Ausnahmen, die durch die Methode geworfen werden können.

In unserem Beispiel müssen wir für beide Methoden angeben, daß eine `Exception` auftreten kann, denn in der Methode `main` können ja die Ausnahmen der Methode `fakultät` auftreten:

FourthThrow.java

```
package de.tfhberlin.panitz.exceptions;
public class FourthThrow {

  public static int fakultät(int n) throws Exception{
    if (n==0) return 1;
```

```
    if (n<0)
     throw
      new Exception("negative Zahl für Fakultätsberechnung");
    return n*fakultät(n-1);
  }

  public static void main(String [] args) throws Exception{
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Somit stellt Java sicher, daß über die möglichen Ausnahmefälle Buch geführt wird.

6.4.4 Eigene Ausnahmeklassen

Man ist bei der Programmierung nicht auf die von Java in Standardklassen ausgedrückten Ausnahmeklassen eingeschränkt. Es können eigene Klassen, die von der Klasse `Exception` ableiten, geschrieben und ebenso wie die Standardausnahmen geworfen werden:

NegativeNumberException.java
```
package de.tfhberlin.panitz.exceptions;
public class NegativeNumberException extends Exception {
}
```

So kann unser Beispielprogramm jetzt unsere eigene Ausnahme werfen :

FifthThrow.java
```
package de.tfhberlin.panitz.exceptions;
public class FifthThrow {

  public static int fakultät(int n) throws Exception{
    if (n==0) return 1;
    if (n<0) throw new NegativeNumberException();
    return n*fakultät(n-1);
  }

  public static void main(String [] args) throws Exception{
    System.out.println(fakultät(5));
    System.out.println(fakultät(-3));
    System.out.println(fakultät(4));
  }
}
```

Bei der Ausführung dieses Programms sehen wir jetzt unsere eigene Ausnahme:


```
sep@swe10:~/fh/beispiele> java de.tfhberlin.panitz.exceptions.FifthThrow
120
Exception in thread "main" de.tfhberlin.panitz.exceptions.NegativeNumberException
        at de.tfhberlin.panitz.exceptions.FifthThrow.fakultät(FifthThrow.java:6)
        at de.tfhberlin.panitz.exceptions.FifthThrow.main(FifthThrow.java:12)
sep@swe10:~/fh/beispiele>
```

6.4.5 Fangen von Ausnahmen

Zu einem vollständigen Konzept zur Ausnahmebehandlung gehört nicht nur, daß über Ausnahmezustände beim Programmabbruch berichtet wird, sondern auch, daß auch angegeben werden kann, wie im Falle einer aufgetretenen Ausnahme weiter zu verfahren ist.

Syntax

Java stellt hierzu das `try`-und-`catch` Konstrukt zur Verfügung. Es hat folgende Struktur:

`try` {*stats*} `catch` (*ExceptionName ident*){*stats*}

Der `try`-Block umschließt in diesem Konstrukt den Code, der bei der Ausführung auf das Auftreten eventueller Ausnahmen abgeprüft werden soll. Der `catch`-Block (von dem es auch mehrere geben kann) beschreibt, was für Code im Falle des Auftretens einer Ausnahme zur Ausnahmebehandlung auszuführen ist. Jetzt können wir programmieren, wie im Falle einer Ausnahme zu verfahren ist:

Catch1.java
```
package de.tfhberlin.panitz.exceptions;
public class Catch1 {

  public static int fakultät(int n) throws Exception{
    if (n==0) return 1;
    if (n<0) throw new NegativeNumberException();
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    try {
      System.out.println(fakultät(5));
      System.out.println(fakultät(-3));
      System.out.println(fakultät(4));
    }catch (Exception e){
      System.out.println("Ausnahme aufgetreten: "+e);
    }
  }
}
```

Wie man sieht, braucht jetzt die Methode `main` nicht mehr zu deklarieren, daß sie eine Ausnahme wirft, denn sie fängt ja alle Ausnahmen, die eventuell während ihrer Auswertung geworfen wurden, ab. Das Programm erzeugt folgende Ausgabe:

```
swe10:~> java de.tfhberlin.panitz.exceptions.Catch1
120
Ausnahme aufgetreten: de.tfhberlin.panitz.exceptions.NegativeNumberException
swe10:~>
```

Das Programm berechnet zunächst korrekt die Fakultät für 5, es kommt zu einer Ausnahme bei der Berechnung der Fakultät von -3. Das Programm verläßt den normalen Programmablauf und wird erst in der `catch`-Klausel wieder abgefangen. Der Code dieser `catch`-Klausel wird ausgeführt. Innerhalb der `catch`-Klausel hat das Programm Zugriff auf das Ausnahmeobjekt, das geworfen wurde. In unserem Fall benutzen wir dieses, um es auf dem Bildschirm auszugeben.

Granularität des Abfangens

Die Granularität, für welche Programmteile eine Ausnahmebehandlung ausgeführt werden soll, steht in unserem Belieben. Wir können z.B. auch für jeden Aufruf der Methode `fakultät` einzeln eine Ausnahmebehandlung vornehmen:

Catch2.java

```java
package de.tfhberlin.panitz.exceptions;
public class Catch2 {

  public static int fakultät(int n) throws Exception{
    if (n==0) return 1;
    if (n<0) throw new NegativeNumberException();
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    try {
      System.out.println(fakultät(5));
    }catch (Exception _){
      System.out.println("Ausnahme für Fakultät von 5");
    }
    try {
      System.out.println(fakultät(-3));
    }catch (Exception _){
      System.out.println("Ausnahme für Fakultät von -3");
    }
    try {
      System.out.println(fakultät(4));
    }catch (Exception _){
      System.out.println("Ausnahme für Fakultät von 4");
    }
  }
}
```

Dieses Programm[6] erzeugt folgende Ausgabe auf dem Bildschirm:

[6] Eine Konvention, die ich in funktionalen Programmiersprachen kennengelernt habe, benutzt für nichtgebrauchte Variablen den Unterstrich als Bezeichner. Da mich in den einzelnen `catch`-Klauseln das Ausnahmeobjekt nicht interessiert, benutze ich jeweils den Unterstrich als Bezeichner dafür.


```
swe10:~/fh/beispiele> java de.tfhberlin.panitz.exceptions.Catch2
120
Ausnahme für Fakultät von -3
24
swe10:~/fh/beispiele>
```

Abfangen spezifischer Ausnahmen

In allen unseren bisherigen Beispielen fangen wir in der `catch`-Klausel allgemein die Fehlerobjekte des Typs `Exception` ab. Ebenso deklarieren wir allgemein in der `throws`-Klausel der Methode `fakultät`, daß ein Ausnahmeobjekt des Typs `Exception` geworfen wird. Hier können wir spezifischer sein und jeweils exakt die Unterklasse von `Exception` angeben, deren Objekte tatsächlich geworfen werden:

Catch3.java

```java
package de.tfhberlin.panitz.exceptions;
public class Catch3 {

  public static int fakultät(int n)
                      throws NegativeNumberException{
    if (n==0) return 1;
    if (n<0) throw new NegativeNumberException();
    return n*fakultät(n-1);
  }

  public static void main(String [] args){
    try {
      System.out.println(fakultät(5));
      System.out.println(fakultät(-3));
      System.out.println(fakultät(4));
    }catch (NegativeNumberException e){
      System.out.println("Ausnahme aufgetreten: "+e);
    }
  }
}
```

Abfangen mehrerer Ausnahmen

Wir können nun nicht nur eine spezifische Ausnahme abfangen, sondern für unterschiedliche Ausnahmen auch unterschiedliche Ausnahmebehandlungen vorsehen. Dieses geschieht einfach, dadurch, daß mehrere `catch`-Klauseln untereinander stehen.

Hierzu definieren wir uns zunächst eine weitere Ausnahmeklasse:

NumberTooLargeException.java

```java
package de.tfhberlin.panitz.exceptions;
public class NumberTooLargeException extends Exception {
}
```

Jetzt können wir unterschiedliche Ausnahmen werfen und wieder fangen:

Catch4.java

```java
package de.tfhberlin.panitz.exceptions;
public class Catch4 {

  public static int fakultät(int n)
        throws NegativeNumberException, NumberTooLargeException{
    if (n==0) return 1;
    if (n<0) throw new NegativeNumberException();
    if (n>20) throw new NumberTooLargeException();
    return n*fakultät(n-1);
  }

  public static void printFakultät(int i){
    try {
      System.out.println(fakultät(i));
    }catch (NegativeNumberException _){
      System.out.println("Fakultät von negativer Zahl");
    }
    catch (NumberTooLargeException _){
      System.out.println("Fakultät von zu großer Zahl");
    }
  }

  public static void main(String [] args){
    printFakultät(30);
    printFakultät(-3);
    printFakultät(4);
  }
}
```

Dieses Programm führt zu folgender Ausgabe:

```
swe10:~/fh/beispiele> java de.tfhberlin.panitz.exceptions.Catch4
Fakultät von zu großer Zahl
Fakultät von negativer Zahl
24
swe10:~/fh/beispiele>
```

Zusammenspiel mit Rückgabewerten

Für Methoden mit einem Rückgabewert ist es beim Abfangen von Ausnahmen wichtig, darauf zu achten, daß in sämtlichen Fällen von abgefangenen Ausnahmen trotzdem ein Rückgabewert zurückgegeben wird. Ebenso ist auch zu berücksichtigen, daß jede benutzte Variable, bevor sie benutzt wird, auch einen Wert zugewiesen bekommen hat. Folgendes Programm wird aus diesem Grund vom Javaübersetzer mit einer Fehlermeldung zurückgewiesen:


```java
package de.tfhberlin.panitz.exceptions;
public class WrongCatch {

  public static int fakultät(int n)
        throws NegativeNumberException, NumberTooLargeException{
    if (n==0) return 1;
    if (n<0) throw new NegativeNumberException();
    if (n>20) throw new NumberTooLargeException();
    return n*fakultät(n-1);
  }

  public static int checkFakultät(int i){
    try {
      return fakultät(i);
    }catch (Exception e){
      System.out.println("Ausnahme "+e+" aufgetreten");
    }
  }

  public static void main(String [] args){
    System.out.println(checkFakultät(30));
    System.out.println(checkFakultät(-3));
    System.out.println(checkFakultät(4));
  }
}
```

Die Übersetzung führt zu folgender Fehlermeldung:

```
sep@swe10:~/fh/beispiele> javac -d . WrongCatch.java
WrongCatch.java:12: missing return statement
  public static int checkFakultät(int i){
                                         ^
1 error
sep@swe10:~/fh/beispiele>
```

Für die im `try`-Block stehenden Befehle ist nicht garantiert, daß sie tatsächlich ausgeführt werden. Tritt eine Ausnahme auf, so wird der `try`-Block verlassen, bevor der `return`-Befehl ausgeführt wurde, und die Methode gibt keinen Rückgabewert zurück, was aber ihre Signatur verlangt. Wir müssen dafür sorgen, daß auch in Ausnahmefällen ein Rückgabewert existiert. Dieses kann durch einen `return`-Befehl im `catch`-Block geschehen:

Catch5.java

```java
package de.tfhberlin.panitz.exceptions;
public class Catch5 {

  public static int fakultät(int n)
        throws NegativeNumberException, NumberTooLargeException{
    if (n==0) return 1;
    if (n<0) throw new NegativeNumberException();
```

```
  if (n>20) throw new NumberTooLargeException();
  return n*fakultät(n-1);
 }

 public static int checkFakultät(int i){
  try {
   return fakultät(i);
  }catch (Exception e){
   System.out.println("Ausnahme "+e+" aufgetreten");
   return 0;
  }
 }

 public static void main(String [] args){
  System.out.println(checkFakultät(30));
  System.out.println(checkFakultät(-3));
  System.out.println(checkFakultät(4));
 }
}
```

Dieses Programm läßt sich wieder fehlerfrei übersetzen und ergibt folgende Ausgabe:

```
[sep@swe10:~/fh/beispiele> java de.tfhberlin.panitz.exceptions.Catch5
Ausnahme de.tfhberlin.panitz.exceptions.NumberTooLargeException aufgetreten
0
Ausnahme de.tfhberlin.panitz.exceptions.NegativeNumberException aufgetreten
0
24
sep@swe10:~/fh/beispiele>
```

6.4.6 Der Aufrufkeller

Wir haben schon gesehen, daß Javas Ausnahmeobjekte wissen, wie die Aufrufreihenfolge war, die zu der Programmstelle führt, in der die Ausnahme erzeugt wurde. Java hat einen internen Aufrufkeller, in dem alle Methoden übereinander stehen, die aufgerufen wurden, aber deren Aufruf noch nicht beendet wurde. Bei der Fehlersuche kann es oft sehr hilfreich sein, an bestimmten Stellen zu erfahren, durch welche Aufrufhierarchie der Methoden an diese Stelle gelangt wurde. Ausnahmeobjekte enthalten die Methode `printStackTrace`, die genau diese Information auf den Bildschirm ausgibt. Wir können diese Methode zur Fehlersuche nutzen, indem wir ein Ausnahmeobjekt erzeugen, dieses aber nicht werfen, sondern lediglich die Methode `printStackTrace` darauf aufrufen. Die normale Ausführungsreihenfolge wird hierdurch nicht berührt, es erscheint lediglich eine weitere Ausgabe auf dem Bildschirm:

StackTrace.java

```
1 package de.tfhberlin.panitz.exceptions;
2 public class StackTrace {
3
4   public static int fakultät(int n) {
```



```
5   if (n==0) {
6     new Exception().printStackTrace();
7    return 1;
8   }
9   return n*fakultät(n-1);
10  }
11
12  public static void main(String [] args){
13   System.out.println(fakultät(4));
14  }
15 }
16
```

Dieses Programm erzeugt folgende recht informative Ausgabe auf dem Bildschirm:

```
sep@swe10:~/fh/beispiele> java de.tfhberlin.panitz.exceptions.StackTrace
java.lang.Exception
        at de.tfhberlin.panitz.exceptions.StackTrace.fakultät(StackTrace.java:6)
        at de.tfhberlin.panitz.exceptions.StackTrace.fakultät(StackTrace.java:9)
        at de.tfhberlin.panitz.exceptions.StackTrace.fakultät(StackTrace.java:9)
        at de.tfhberlin.panitz.exceptions.StackTrace.fakultät(StackTrace.java:9)
        at de.tfhberlin.panitz.exceptions.StackTrace.fakultät(StackTrace.java:9)
        at de.tfhberlin.panitz.exceptions.StackTrace.main(StackTrace.java:13)
24
sep@swe10:~/fh/beispiele>
```

Es läßt sich hier sehr schön die rekursive Struktur der Fakultätsmethode nachverfolgen: Wenn von der Methode die Zeile 6 ausgeführt wird, so wurde sie schon viermal in Zeile 9 ausgeführt und all diese Aufrufe der Methode sind noch nicht beendet. Man kann den Aufrufkeller auch als Liste der noch zu erledigenden Methoden betrachten. Die Ausführung aller Methoden, die dort verzeichnet sind, wurde bereits angefangen, aber noch nicht beendet.

6.4.7 Schließlich und finally

Java erlaubt, am Ende eines `try`-und-`catch`-Konstruktes noch eine `finally`-Klausel hinzuzufügen. Diese besteht aus dem Schlüsselwort `finally`, gefolgt von Anweisungen. Die Anweisungen einer `finally`-Klausel werden immer ausgeführt, unabhängig davon, ob eine Ausnahme abgefangen wurde oder nicht. Im folgenden Programm wird in beiden Fällen der Text in der `finally`-Klausel ausgegeben:

Finally.java

```
class Finally {

 static void m(int i) throws Exception{
  if (i>0) throw new Exception();
 }

 public static void main(String [] args){
  try {m(1);}
  catch (Exception _){System.out.println("Ausnahme gefangen");}
```

```
    finally {System.out.println("erster Test");}

    try {m(-1);}
    catch (Exception _){System.out.println("Ausnahme gefangen");}
    finally {System.out.println("zweiter Test");}
  }
}
```

Die `finally`-Klausel wird nicht nur ausgeführt, wenn keine Ausnahme geworfen oder eine Ausnahme gefangen wurde, sondern auch, wenn eine Ausnahme geworfen wurde, für die es keine `catch`-Klausel gibt[7].

So wird in der folgenden Klasse die Ausgabe der `finally`-Klausel sogar gemacht, obwohl eine Ausnahme auftritt, die nicht abgefangen wird:

MoreFinally.java

```
class MoreFinally {

    static void m(int i){
        if (i<0) throw new NullPointerException();
    }

    public static void main(String [] args){
        try {m(-1);}
        catch (IndexOutOfBoundsException _){}
        finally
        {System.out.println("wird trotzdem ausgegeben");}
   }

}
```

Wie man an dem Programmlauf sieht, tritt eine nichtabgefangene Ausnahme auf, und trotzdem wird noch der `finally`-Code ausgeführt.

Die `finally`-Klausel ist dazu da, um Code zu einer Ausnahmebehandlung hinzuzufügen, der sowohl nach dem Auftreten von jeder abgefangenen Ausnahme auszuführen ist als auch , wenn keine Ausnahme abgefangen wurde. Typisch für solchen Code sind Verwaltungen von externen Komponenten. Wenn in einer Methode eine Datenbankverbindung geöffnet wird, so ist diese Datenbankverbindung sowohl im erfolgreichen Fall als auch, wenn irgendeine Ausnahme auftritt, wieder zu schließen.

6.4.8 Anwendungslogik per Ausnahmen

Es gibt die Möglichkeit, Ausnahmen nicht nur für eigentliche Ausnahmefälle zu benutzen, sondern die normale Anwendungslogik per Ausnahmebehandlung zu programmieren. Hierbei ersetzt ein `try-catch`-Block eine `if`-Bedingung. Einer der Fälle, die normaler Weise

[7]Ansonsten wäre die *finally*-Klausel ein überflüssiges Konstrukt, könnte man seinen Code ja direkt anschließend an das *try-catch*-Konstrukt anhängen.

über eine Bedingung angefragt werden, wird als Ausnahmefall behandelt. In unserer Listenimplementierung wurde für die leere Liste von den Methoden `head` und `tail` jeweils der Wert `null` als Ergebnis zurückgegeben. Bei der Programmierung der Methode `length` kann man sich zunutze machen, daß der Aufruf einer Methode auf den Wert `null` zu einer `NullPointerException` führt. Der Fall einer leeren Liste kann über eine abgefangene Ausnahme behandelt werden:

```
  public int length(){

    try {
      return 1+tail().length();
    }catch (NullPointerException _){
      return 0;
    }
  }
```

In diesem Programmierstil wird auf eine anfängliche Unterscheidung von leeren und nichtleeren Listen verzichtet. Die eine entsprechende `if`-Bedingung entfällt. Erst, wenn die Auswertung von `tail().length()` zu einer Ausnahme führt, wird darauf rückgeschlossen, daß wohl eine leere Liste vorgelegen haben muß.

Kapitel 7

Java Standardklassen

Java kommt mit einer sehr großen und von Version zu Version wachsenden Bibliothek von Standardklassen daher. In diesem Kapitel soll ein Überblick über die wichtigsten Standardklassen gegeben werden. Naturgemäß können nicht alle Standardklassen vorgestellt werden. Man informiere sich gegebenenfalls durch die entsprechenden Dokumentationen. Das Hauptpaket für Javastandardklassen ist das Paket `java`. Es hat folgende wichtigen Unterpakete:

- `java.lang`: das automatisch importierte Standardpaket. Hier finden sich die allerwichtigsten und permanent gebrauchten Klassen, wie z.B. `String`, `System` und vor allem auch `Object`. Auch die wichtigsten Ausnahmeklassen liegen in diesem Paket.
- `java.util`: hier befinden sich primär die Sammlungsklassen und Schnittstellen für die Sammlungsklassen. Desweiteren finden sich hier Klassen zum Umgang mit Kalenderdaten, Zeiten oder Währungen.
- `java.io`: Klassen für dateibasierte Ein-/Ausgabe-Operationen.
- `java.applet`: Klassen zum Schreiben von Applets.
- `java.awt`: Klassen für graphische Komponenten.
- `javax.swing`: weitere Klassen für graphische Komponenten. Neuere Implementierung zu `java.awt`, die versucht, noch generischer und plattformunabhängiger zu sein.

In Java gibt es noch eine Vielzahl weiterer Standardpakete mit unterschiedlichster spezieller Funktionalität, z.B. Klassen für den Umgang mit XML-Daten oder für die Anbindung an Datenbanken. Hier empfiehlt es sich im entsprechenden Fall, die Dokumentation zu lesen und nach einem Tutorial auf den Sun-Webseiten zu suchen.

7.1 Die Klasse Objekt

Die Klasse `java.lang.Object` ist die Wurzel der Klassenhierarchie in Java. Alle Objekte erben die Methoden, die in dieser Klasse definiert sind. Trotzdem ist bei fast allen Methoden nötig, daß sie durch spezialisierte Versionen in den entsprechenden Unterklassen überschrieben werden. Dieses haben wir im Laufe des Skripts schon an der Methode `toString` nachverfolgen können.

7.1.1 Die Methoden der Klasse Object

equals

`public boolean equals(Object obj)` ist die Methode zum Testen auf die Gleichheit von zwei Objekten. Dabei ist die Gleichheit nicht zu verwechseln mit der Identität, die mit dem Operator `==` getestet wird. Der Unterschied zwischen Identität und Gleichheit läßt sich sehr schön an Strings demonstrieren:

EqualVsIdentical.java

```
class EqualVsIdentical {

  public static void main(String [] args){
    String x = "hallo".toUpperCase();
    String y = "hallo".toUpperCase();

    System.out.println("x: "+x);
    System.out.println("y: "+y);

    System.out.println("x==x         -> "+(x==x));
    System.out.println("x==y         -> "+(x==y));
    System.out.println("x.equals(x) -> "+(x.equals(x)));
    System.out.println("x.equals(y) -> "+(x.equals(y)));
  }

}
```

Die Ausgabe dieses Tests ist:

```
sep@swe10:~/fh/beispiele> java EqualVsIdentical
x: HALLO
y: HALLO
x==x        -> true
x==y        ->false
x.equals(x) ->true
x.equals(y) ->true
sep@swe10:~/fh/beispiele>
```

Obwohl die beiden Objekte `x` und `y` die gleichen Texte darstellen, sind es zwei unabhängige Objekte; sie sind nicht identisch, aber gleich.

Sofern die Methode `equals` für eine Klasse nicht überschrieben wird, wird die entsprechende Methode aus der Klasse `Object` benutzt. Diese überprüft aber keine inhaltliche Gleichheit. Es ist also zu empfehlen, die Methode `equals` für alle eigenen Klassen, die zur Datenhaltung geschrieben wurden, zu überschreiben. Dabei sollte die Methode immer folgender Spezifikation genügen:

- **Reflexivität**: es sollte immer gelten: `x.equals(x)`
- **Symmetrie**: wenn `x.equals(y)` dann auch `y.equals(x)`
- **Transitivität**: wenn `x.equals(y)` und `y.equals(z)` dann gilt auch `x.equals(z)`

- **Konsistenz**: wiederholte Aufrufe von `equals` auf dieselben Objekte liefern dasselbe Ergebnis, sofern die Objekte nicht verändert wurden.
- nichts gleicht `null`: `x.equals(null)` ist immer falsch.

hashCode

`public int hashCode()` ist eine Methode, die für das Objekt eine beliebige ganze Zahl, seinen *hashcode*, angibt[1]. Eine solche Zahl wird von vielen Algorithmen verwendet, so daß die Javaentwickler sie für so wichtig hielten, daß sie eine Methode in der Klasse `Object` hierfür vorgesehen haben.

Ein *Hash*-Verfahren funktioniert wie folgt. Stellen Sie sich vor, Sie haben ein großes Warenlager mit 1000 durchnummerierten Regalen. Wann immer Sie ein Objekt in diesem Lager lagern wollen, fragen Sie das Objekt nach seinem *Hashcode*. Diese Zahl nehmen Sie modulo 1000. Das Ergebnis dieser Rechnung ist eine Zahl zwischen 0 und 999. Jetzt legen Sie das Objekt in das Regal mit der entsprechenden Nummer. Suchen Sie jetzt ein bestimmtes Objekt in ihrem Warenlager, so können Sie wiederum den *Hashcode* des Objektes nehmen und wissen, es kann nur in genau einem der 1000 Regale liegen. Sie brauchen also nur in einem und nicht in 1000 Regalen nach Ihrem Objekt zu suchen.

Damit ein solches Verfahren funktioniert, muß folgende Spezifikation für die Methode `hashCode` erfüllt sein:

- wenn `x.equals(y)` dann folgt `x.hashCode()==y.hashCode()`
- bleibt ein Objekt unverändert, so ändert sich auch nicht sein *Hashcode*

Beachten Sie, daß ungleiche Objekte nicht unbedingt ungleiche *Hashwerte* haben.

clone

`protected Object clone() throws CloneNotSupportedException` ist eine Methode, die dazu dient, ein neues Objekt zu erzeugen, daß inhaltlich gleich, aber nicht identisch zum `this`-Objekt ist. Dieses läßt sich formaler durch folgende Gleichungen ausdrücken[2]:

- `x.clone() != x`
- `x.clone().equals(x)`

Die Methode `clone` in der Klasse Objekt hat in der Regel schon eine Funktionalität, die wir erwarten. Sie nimmt das Objekt und erstellt eine Kopie von diesem Objekt. Leider gibt es in Java einen verwirrenden Effekt. Die Methode `clone` verhält sich unterschiedlich, abhängig davon, ob die Klasse des gecloneten Objektes die Schnittstelle `Cloneable` implementiert:

[1] Hierbei handelt es sich in keinster Weise um irgendwelche unerlaubten Substanzen. Mit *hash* wird im Englischen das Nummersymbol # des Gatters bezeichnet.

[2] Diese Gleichungen verdeutlichen noch einmal sehr den Unterschied, auf den auch Pedanten der deutschen Sprache gerne hinweisen: Das geclonete Objekt gleicht dem Original, ist aber nicht dasselbe Objekt.

NoClone.java

```java
class NoClone {
  String x = "hallo";
  public static void main(String [] args)
    throws CloneNotSupportedException{
    NoClone nc = new NoClone();
    NoClone cc = (NoClone)nc.clone();
    System.out.println(cc.x);
  }
}
```

Dieses Programm wirft eine Ausnahme:

```
sep@swe10:~/fh/beispiele> java NoClone
Exception in thread "main" java.lang.CloneNotSupportedException: NoClone
        at java.lang.Object.clone(Native Method)
        at NoClone.main(NoClone.java:5)
sep@swe10:~/fh/beispiele>
```

Wenn wir hingegen das Programm so ändern, daß es die Schnittstelle `Cloneable` implementiert, dann wird diese Ausnahme nicht geworfen:

Clone.java

```java
class Clone implements Cloneable {
  String x = "hallo";

  public static void main(String [] args)
      throws CloneNotSupportedException{
    Clone nc = new Clone();
    Clone cc = (Clone)nc.clone();
    System.out.println(cc.x);
  }
}
```

Dieses Programm führt zu einem Programmdurchlauf ohne Ausnahme:

```
sep@swe10:~/fh/beispiele> java Clone
hallo
sep@swe10:~/fh/beispiele>
```

Besonders verwirrend scheint dieses Verhalten zu sein, weil die Schnittstelle `Cloneable` überhaupt gar keine Methoden enthält, die zu implementieren sind.

finalize

`protected void finalize() throws Throwable` ist eine Methode, die ein ähnliches Anwendungsgebiet wie die `finally`-Klausel in einer Ausnahmebehandlung hat. Wenn Java ein Objekt in seinem Speicher findet, von dem Java zeigen kann, daß es von keinem laufenden

Programmteil mehr genutzt wird, dann löscht Java dieses Objekt aus dem Speicher. Bevor das Objekt vollkommen gelöscht wird, hat das Objekt noch einen letzten Wunsch frei, indem nämlich seine Methode `finalize` aufgerufen wird. Die meisten Objekte sind genügsam und lassen sich ohne weitere Aktion aus dem Speicher löschen. Sobald ein Objekt aber eine externe Komponente darstellt, sei es ein Datenbankeintrag oder ein externes Programm, so ist es vielleicht notwendig, für diese externe Komponente noch Aktionen durchzuführen: die externe Komponente ist zu schließen oder z.B. ein Datenbankeintrag zu verändern. Wenn Objekte solcherlei Aktionen benötigen, bevor sie endgültig aus dem Speicher gelöscht werden, so ist für ihre Klasse die Methode `finalize` zu überschreiben. Java gibt keinerlei Garantie darüber, wann die Methode `finalize` ausgeführt wird. Es kann relativ zufällig sein, wie lange sich ein Objekt im Speicher befindet.

Beispiel:
Wir können einmal testen, wie Java die Objekte willkürlich aus dem Speicher löscht. Hierzu schreiben wir eine Klasse, die in der Methode `finalize` eine Aufgabe über das Objekt auf der Kommandozeile ausgibt.

Finalize.java
```
class Finalize {
   int i;
   Finalize(int i){this.i=i;}
   protected void finalize(){
     System.out.print(i+",");
   }
```

In einer Hauptmethode erzeugen wir einmal nacheinander viele Objekte dieser Klasse, ohne daß wir sie irgendwie weiter benutzen wollen:

Finalize.java
```
   public static void main(String [] _){
     for (int i=0;i<100000;i=i+1){
       new Finalize(i);
       System.out.print("*");
     }
   }
}
```

Aufgabe 23 Übersetzen und starten Sie die Klasse `Finalize` und beobachten Sie, wie und wann Java Objekte aus dem Speicher löscht.

toString

`public String toString()` ist eine Methode, die wir schon oft benutzt oder überschrieben haben. Wenn wir sie nicht überschreiben, sondern aus der Klasse `Object` erben, so bekommen wir eine textuelle Darstellung, die sich wie folgt berechnet:
`getClass().getName() + '@' + Integer.toHexString(hashCode())`

weitere Methoden

Es gibt noch einige weitere Methoden in der Klasse Objekt. Diese beschäftigen sich allerdings alle mit Steuerfäden oder der Reflektion, zwei Konzepten, die wir noch nicht kennengelernt haben.

7.2 Behälterklassen für primitive Typen

Da Daten von primitiven Typen keine Objekte sind, für die Methoden aufgerufen oder die einem Feld des Typs `Object` zugewiesen werden können, stellt Java für jeden primitiven Typen eine Klasse zur Verfügung, die es erlaubt, primitive Daten als Objekte zu betrachten. In der Regel haben die Klassen einen Konstruktor, der einen Wert des entsprechenden primitiven Typen als Parameter hat, und einen Konstruktor, der einen String als einen bestimmten Wert interpretiert. Objekte dieser Klasse sind unveränderbar. Der eigentlich dargestellte Wert des primitiven Typs ist durch eine Zugriffsmethode zu erhalten.

Die Klassen sind: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`.

In den Klassen gibt es zumeist noch Konstanten und Methoden, die etwas über den primitiven Typen aussagen, z.B. das Feld `MAX_VALUE` in der Klasse `Integer`.

7.3 String und Stringbuffer

Die Klasse `String` mit einer Reihe von Methoden haben wir des öfteren bereits kennengelernt. Zusätzlich gibt es noch die Klasse `StringBuffer`. Sie unterscheidet sich primär von `String` darin, daß ihre Objekte veränderbar sind. Der Operator `+` für `String`-Objekte erzeugt ein neues Objekt. In der Klasse `StringBuffer` existiert die Methode `append`, die ein Objekt verändert, indem es weitere Zeichen an die Zeichenkette anhängt.

Der Unterschied ist leicht durch das folgende Programm zu verdeutlichen:

StringVsStringBuffer.java
```
class StringVsStringBuffer {

  public static void main(String [] args){
    String a = "hello";
    String b = a+" world";
    System.out.println(a);

    StringBuffer ab = new StringBuffer("hello");
    StringBuffer bb = ab.append(" world");
    System.out.println(ab);
  }
}
```

Programme, die nach und nach einen Ergebnisstring berechnen, werden in der Regel ein Objekt des Typs `StringBuffer` erzeugen, auf das mehrmals die Methode `append` ausgeführt wird. Schließlich wird von diesem Objekt die `toString` Methode benutzt, um den Ergebnisstring zu erhalten:

ReturnString.java
```java
class ReturnString {

  static public String toRoman(int i){
    StringBuffer result = new StringBuffer();
    while (i>=1000) {i=i-1000;result.append("M");}
    while (i>=900)  {i=i-900 ;result.append("CM");}
    while (i>=500)  {i=i-500 ;result.append("D");}
    while (i>=400)  {i=i-400 ;result.append("CD");}
    while (i>=100)  {i=i-100 ;result.append("C");}
    while (i>=90)   {i=i-90  ;result.append("XC");}
    while (i>=50)   {i=i-50  ;result.append("L");}
    while (i>=40)   {i=i-40  ;result.append("XL");}
    while (i>=10)   {i=i-10  ;result.append("X");}
    while (i>=9)    {i=i-9   ;result.append("IX");}
    while (i>=5)    {i=i-5   ;result.append("V");}
    while (i>=4)    {i=i-4   ;result.append("IV");}
    while (i>=1)    {i=i-1   ;result.append("I");}
    return result.toString();
  }

  public static void main(String [] args){
    System.out.println(toRoman(1999));
    System.out.println(toRoman(494));
  }

}
```

7.4 Sammlungsklassen

Wir haben in diesem Skript ausführlich eigene Listenklassen auf unterschiedliche Weisen spezifiziert, modelliert und implementiert. Java stellt im Paket `java.util` Implementierungen von Sammlungsklassen zur Verfügung.

Die Sammlungsklassen sind über verschiedene Schnittstellen definiert. Die Oberschnittstelle für Sammlungsklassen ist: `java.util.Collection`. Ihre Hauptunterschnittstellen sind: `List` und `Set` für die Darstellung von Listen bzw. Mengen.

7.4.1 Listen

Im Gegensatz zu unseren eigenen Listen sind Listenobjekte in Java zunächst einmal veränderbare Objekte. In unserer Modellierung gab es keine Methode wie `add`, die in einer Liste ein Objekt eingefügt hat. Bei uns wurde stets eine neue Liste mit einem zusätzlich vorn angehängten Element erzeugt. Javas Standardlisten haben Methoden, die den Inhalt eines Listenobjektes verändern:

- `add` zum Hinzufügen von Elementen.

- `clear` zum Löschen aller Elemente.
- `remove` zum Löschen einzelner Elemente.

Der Unterschied zwischen unseren eigenen Listen und den Listen der Javabibliothek ist ungefähr derselbe wie der zwischen der Klasse `String` und `StringBuffer`.

Listenklassen

Listen sind mit Hilfe einer abstrakten Klasse implementiert. Die eigentlichen konkreten Klassen, die Listen implementieren, sind: `ArrayList`, `LinkedList` und `Vector`. Dabei ist `ArrayList` die gebräuchlichste Implementierung. `Vector` ist eine ältere Implementierung, die als Nachteil hat, daß sie stets eine Synchronisation für nebenläufige Steuerfäden vornimmt, die in der Regel nicht unbedingt benötigt wird.

Folgende Klasse zeigt, wie eine Liste erzeugt wird und ihr nach und nach Elemente hinzugefügt werden:

ListUsage.java
```java
import java.util.List;
import java.util.ArrayList;

class ListUsage{

  public static void main(String [] args){
    List xs = new ArrayList();
    xs.add("hallo");
    xs.add("welt");
    xs.add("wie");
    xs.add("geht");
    xs.add("es");
    xs.add("dir");
    System.out.println(xs);
  }

}
```

Das Programm hat folgende Ausgabe:

```
sep@swe10:~/fh/beispiele> java ListUsage
[hallo , welt , wie , geht , es , dir ]
sep@swe10:~/fh/beispiele>
```

Wie man sieht, fügt die Methode `add` Objekte am Ende einer Liste an.

7.4.2 Mengen

Im Gegensatz zu Listen können Mengenobjekte keine Elemente doppelt enthalten. Hierbei werden zwei Objekte als gleich betrachtet, wenn der Aufruf der Methode `equals` für sie den Wert `true` ergibt. Daraus folgt insbesondere, daß es nur sinnvoll ist, Elemente zu einer Menge zuzufügen, für die auch eine sinnvolle Überschreibung der Methode `equals` existiert.

Die Klasse `HashSet` ist eine Implementierung der Schnittstelle `Set`. Sie basiert darauf, daß die Objekte, die in der Menge gespeichert werden sollen, eine möglichst gute Implementierung der Methode `hashCode` haben. Möglichst gut bedeutet in diesem Fall, daß in möglichst vielen Fällen gilt:

Aus `!x.equals(y)` folgt `x.hashCode != y.hashCode()`

7.4.3 Iteratoren

Java stellt eine Schnittstelle `Iterator` bereit. Ihr einziger Zweck ist, zu beschreiben, daß durch eine Sammlung von Elementen durchiteriert werden kann.

Ein Iterator kann nacheinander gebeten werden, sein nächstes Element auszugeben, und ein Iterator kann gefragt werden, ob es noch weitere Elemente gibt.

Hierzu gibt es die entsprechenden Methoden:

- `Object next()`, die den Iterator um ein Element weiterschaltet und das aktuelle Element als Ergebnis hat.
- `boolean hasNext()`, die `true` als Ergebnis hat, wenn die Methode `next` ein weiteres Element liefert.

Iteratoren sind zum einmaligen Gebrauch gedacht. Sie werden benutzt, um einmal durch die Elemente einer Sammlung durchzuiterieren.

Iterator für Li

Am einfachsten können wir das Prinzip eines Iteratorobjekts verstehen, wenn wir eine entsprechende Iteratorklasse für unsere eigenen Listen schreiben:

LiIterator.java

```
public class LiIterator implements java.util.Iterator{

  public LiIterator(Li xs){this.xs=xs;}

  private Li xs;

  public boolean hasNext() {return !xs.isEmpty();}

  public Object next(){
    Object result = xs.head();
    xs = xs.tail();
    return result;
  }

  public void remove(){
   throw new UnsupportedOperationException ();
  }
}
```


Der Klasse `Li` können wir eine Methode hinzufügen, die den entsprechenden Iterator für die Liste zurückgibt:

```
  public java.util.Iterator iterator(){
    return new LiIterator(this);
  }
```

Benutzung von Iteratoren

Iteratoren sind dafür gedacht, über die Elemente von Sammlungsklassen zu iterieren. Hierzu bedient man sich normaler Weise der `for`-Schleife. Sie wird hierzu mit einem Iterator initialisiert und hat als Bedingung, daß der Iterator noch ein weiteres Element liefert. Die Weiterschaltung wird in der Regel leer gelassen, weil Iteratoren mit dem Aufruf der Methode `next` beides machen: das nächste Element ausgeben und den Iterator weiterschalten.

Wir können als kleines Beispiel eine kleine Methode schreiben, die die Elemente einer Sammlung als `String` aneinanderhängt:

Iterate.java

```
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;

class Iterate {

  static public String unwords(Collection cs){
    String result = "";

    for (Iterator it=cs.iterator();it.hasNext();){
      result = result+" "+it.next();
    }
    return result;
  }

  static public void main(String [] _){
    Collection cs = new ArrayList();
    cs.add("lebt");
    cs.add("wohl");
    cs.add("gott");
    cs.add("weiß");
    cs.add("wann");
    cs.add("wir");
    cs.add("uns");
    cs.add("wiedersehen");

    System.out.println(unwords(cs));
```

```
  }
}
```

Eine kleine Warnunug: Es passiert leicht, daß man in einer entsprechenden Schleife, die einen Iterator benutzt, zweimal die Methode `next` aufruft. Das ist dann in der Regel nicht das gewünschte Verhalten. Statt z.B.:

```
    for (Iterator it=cs.iterator();it.hasNext();){
      result = result+" "+it.next();
      System.out.println(it.next())
    }
```

war wahrscheinlich das gewünschte Verhalten:

```
    for (Iterator it=cs.iterator();it.hasNext();){
      Object theNext = it.next();
      result = result+" "+theNext;
      System.out.println(theNext);
    }
```

7.4.4 Abbildungen

Abbildungen assoziieren Elemente einer Art mit Elementen einer anderen Art. Sie sind vergleichbar mit Wörterbüchern. In einem deutsch/englischen Wörterbuch werden die deutschen Wörter auf die englischen Wörter mit der entsprechenden Bedeutung abgebildet.

Wir benutzen eine Abbildung, indem wir für einen Schlüssel (dem deutschen Wort) den entsprechenden Werteeintrag suchen (das englische Wort).

Abbildungen als Liste von Paaren

Da eine Abbildung jeweils ein Element mit einem anderem assoziiert, ist eine Umsetzung als Liste von Paaren naheliegend.[3] Ein Paar besteht dabei aus einem Schlüssel und dem assoziierten Wert.

Paar.java

```
public class Paar{
  public Object fst;
  public Object snd;
  public Paar(Object f,Object s){fst=f;snd=s;}
}
```

So ist eine einfache Umsetzung von Abbildungen durch Erweitern unserer Listenklassen zu bekommen:

[3] Das Programm `Eliza` aus einer der vorangegangenen Aufgaben hat genau so Abbildungen umgesetzt.

Abbildung.java

```
class Abbildung extends Li{
  public Abbildung(Object schlüssel,Object wert,Abbildung xs){
      super(new Paar(schlüssel,wert),xs);
  }

  public Abbildung(){
  }

  Object lookup(Object schluessel){
    Object result = null;

    for (Abbildung xs = this
        ;!this.isEmpty()
        ;xs=(Abbildung)xs.tail()){
      Paar p = (Paar)xs.head();
      if (p.fst.equals(schluessel)){
        return p.snd;
      }
    }
    return result;
  }

}
```

Die Methode `lookup` iteriert über die Liste, bis sie ein Listenelement gefunden hat, dessen Schlüssel dem gesuchten Schlüssel gleich kommt.

Dictionary.java

```
class Dictionary {
  static final Abbildung dic =
     new Abbildung("Menge","set"
    ,new Abbildung("Abbildung","map"
    ,new Abbildung("Liste","list"
    ,new Abbildung("Iterator","iterator"
    ,new Abbildung("Schnittstelle","interface"
    ,new Abbildung("Klasse","class"
    ,new Abbildung())))))));

  static String english(String deutsch){
    return (String)dic.lookup(deutsch);
  }

  public static void main(String [] _){
    System.out.println(english("Schnittstelle"));
    System.out.println(english("Abbildung"));
  }
}
```

Standardklasse für Abbildungen

Java stellt eine Schnittstelle zur Verwirklichung von Abbildungen zur Verfügung, die Schnittstelle `java.util.Map`.

Die wichtigsten zwei Methoden dieser Schnittstelle sind:

- `void put(Object key, Object value)`: ein neues Schlüssel-/Wertpaar wird der Abbildung hinzugefügt.
- `Object get(Object key)`: für einen bestimmten Schlüssel wird ein bestimmter Wert nachgeschlagen. Gibt es für diesen Schlüssel keinen Eintrag in der Abbildung, so ist das Ergebnis dieser Methode `null`.

Eine Schnittstelle, die Abbildungen implementiert, ist die Klasse `HashMap`. Ihre Benutzung funktioniert ähnlich wie die unserer Klasse `Abbildung`, mit dem bereits bekannten Unterschied, daß unsere Abbildungen unveränderbare Objekte sind, während die Objekte der entsprechenden Javaklasse durch die `put`-Methode modifiziert werden:

Dic.java

```java
import java.util.*;

class Dic{

  static Map map = new HashMap();

  static {
    map.put("Menge","set");
    map.put("Abbildung","map");
    map.put("Liste","list");
    map.put("Iterator","iterator");
    map.put("Schnittstelle","interface");
    map.put("Klasse","class");
  }

  static String english(String deutsch){
   return (String)map.get(deutsch);
  }

  public static void main(String [] _){
    System.out.println(english("Schnittstelle"));
    System.out.println(english("Abbildung"));
  }

}
```

7.4.5 Weitere Sammlungsmethoden

In der Klasse `java.util.Collections` befinden sich statische Methoden zur Manipulation von Listen, wie z.B. zum Rotieren, Rückwärtsdrehen oder Sortieren.

7.4.6 Vergleiche

Ähnlich, wie wir für unsere Sortiermethode `sortBy` eine Klasse `Relation` als Parameter übergeben haben, in der es eine Methode gab, die prüfte, ob zwei Elemente in der kleiner-Relation stehen, gibt es in Java die Schnittstelle `java.util.Comparator`. In dieser Klasse gibt es die Methode `int compare(Object o1, Object o2)`, deren Ergebnis eine negative Zahl, 0 oder eine positive Zahl ist, je nachdem, ob das erste Argument kleiner, gleich oder größer als das zweite Argument ist.

Wir können einen Comparator für Stringobjekte schreiben, der zwei Strings vergleicht, indem er sie erst umdreht und erst dann die Standardvergleiche für Strings anwendet:

ReverseComparator.java

```java
import java.util.Comparator;
class ReverseComparator implements Comparator {

  public int compare(Object o1, Object o2) {
    return reverse((String) o1).compareTo(reverse((String) o2));
  }

  static private String reverse(String str){
    StringBuffer result = new StringBuffer();
    for (int i=str.length()-1;i>=0;i=i-1){
      result.append(str.charAt(i));
    }
    return result.toString();
  }

}
```

Mit diesem Comparator läßt sich jetzt eine Liste von Strings sortieren. Das Ergebnis ist dann eine Liste, in der sich sich reimende Wörter hintereinander befinden:

Rhyme.java

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

class Rhyme {
  public static List/*String*/ rhyme(List/*String*/ xs){
    Collections.sort(xs,new ReverseComparator());
    return xs;
  }

  public static void main(String [] args){
      List xs = new ArrayList();
      xs.add("maus");
      xs.add("rasieren");
      xs.add("bauhaus");
      xs.add("verlieren");
      xs.add("regen");
```

```
      xs.add("laus");
      xs.add("segen");
      xs.add("kapitulieren");
      xs.add("monieren");
      xs.add("existieren");
      xs.add("kraus");
      rhyme(xs);
      System.out.println(xs);
  }

}
```

Als eine zweite Schnittstelle stellt Java die Standardklasse `java.lang.Comparable` zur Verfügung. In dieser Schnittstelle gibt es eine Methode `int compareTo(Object o)`. Die Sortiermethoden auf Sammlungsklassen erwarten, daß die Elemente der Sammlung die Schnittstelle `Comparable` implementieren. Ansonsten kommt es in der Sammlungsklasse beim Sortieren zu einer `ClassCastException`.

7.5 Reihungen

Java kennt, wie fast alle Programmiersprachen, ein weiteres Konzept von Sammlungen: Reihungen (eng. arrays).[4] Reihungen stellen im Gegensatz zu Listen oder Mengen eine Menge von Daten gleichen Typs mit fester Anzahl dar. Jedes Element einer Reihung hat einen festen Index, über den es direkt angesprochen werden kann.

7.5.1 Deklaration von Reihungen

Eine Reihung hat im Gegensatz zu den Sammlungsklassen[5] 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

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

[4]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.

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

7.5.2 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

```
class FirstArray {

 static String [] komponisten
  = {"carcassi","carulli","giuliani"
    ,"molino","monzino","paganini","sor"};

  public static void main(String [] args){
    System.out.println(komponisten.length);
    System.out.println(komponisten.toString());
  }
}
```

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

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

```
class SecondArray {

  static int [] zahlenReihung = new int[10];

  public static void main(String [] args){
    System.out.println(zahlenReihung);
  }
}
```

7.5.3 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:

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

```
public class ArrayToString{
  static public String arrayToString(String [] obja){
    StringBuffer result = new StringBuffer("{");
    for (int i=0;i<obja.length;i=i+1){
      if (i>0) result.append(",");
      result.append(obja[i].toString());
    }
    result.append("}");
    return result.toString();
  }
}
```

7.5.4 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:

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

7.5.5 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.[6] Diese Zuweisung führt dazu, daß Information verloren geht. Hierzu betrachte man folgendes kleine Beispiel:

Kovarianz.java

```
class Kovarianz {

  static String [] stra = {"hallo","welt"};

```

[6] 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.


```
  //gefährliche Zuweisung
  static Object [] obja = stra;

  public static void main(String [] args) {
    //dieses ändert auch stra. Laufzeitfehler!
    obja[0]=new Integer(42);

    String str = stra[0];
    System.out.println(str);
  }
}
```

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[7] angewendet werden könnte.

7.5.6 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

[7] Primitive Typen sind dabei ausgenommen.

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

zum Kopieren bestimmter Reihungsabschnitte in eine zweite Reihung.

7.5.7 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 {}
```

7.5.8 Blubbersortierung

Wir haben bisher zwei Sortierverfahren im Zusammenhang mit Listen kennengelernt. Ein weiteres Verfahren, das für Listen denkbar ungeeignet ist, ist das sogenannte *bubble sort*.

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
```
class BubbleSort {

  static void bubbleSort(Comparable [] obja){
    boolean toBubble = true;
    while (toBubble){
      toBubble = bubble(obja);
    }
  }

  static boolean bubble(Comparable [] obja){
    boolean result = false;
    for (int i=0;i<obja.length;i=i+1){
      try {
        if (obja[i].compareTo(obja[i+1])>0){
          Comparable o = obja[i];
          obja[i]=obja[i+1];
          obja[i+1]=o;
          result=true;
        }
```

```
      }catch (ArrayIndexOutOfBoundsException _){}
    }
    return result;
  }

  static public void main(String [] args){
    String [] stra = {"a","zs","za","bb","aa","aa","y"};
    System.out.println(ArrayToString.arrayToString(stra));
    bubbleSort(stra);
    System.out.println(ArrayToString.arrayToString(stra));
  }

}
```

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

Aufgabe 24 (3 Punkte)

In dieser Aufgabe soll ein Spielbrett für das Spiel Vier gewinnt implementiert werden. Laden Sie hierzu die Datei
vier.zip (`http://www.tfh-berlin.de/~panitz/prog1/load/vier.zip`)

a) Schreiben Sie eine Klasse `VierImplementierung`, die die Schnittstelle `VierLogik` entsprechend der Dokumentation implementiert.

b) Schreiben Sie folgende Hauptmethode und starten Sie diese. Sie sollten jetzt in der Lage sein, über die Eingabekonsole Vier gewinnt zu spielen.

```
public static void main(String[] args) {
  new VierKonsole().spiel(new VierImplementierung());
}
```

Suchen Sie sich einen Spielpartner und versuchen Sie, gegen ihn zu gewinnen.

7.6 Ein- und Ausgabe

Im Paket `java.io` befinden sich eine Reihe von Klassen, die Ein- und Ausgabe von Daten auf externen Datenquellen erlauben. In den häufigsten Fällen handelt es sich hierbei um Dateien.

Ein-/Ausgabeoperationen werden in Java auf sogenannten Datenströmen ausgeführt. Datenströme verbinden das Programm mit einer externen Datenquelle bzw. Senke. Auf diesen Strömen stehen Methoden zum Senden (Schreiben) von Daten an die Senke bzw. Empfangen (Lesen) von Daten aus der Quelle. Typischer Weise wird ein Datenstrom angelegt, geöffnet, dann werden darauf Daten gelesen und geschrieben und schließlich der Strom wieder geschlossen.

Ein Datenstrom charakterisiert sich dadurch, daß er streng sequentiell arbeitet. Daten werden also auf Strömen immer von vorne nach hinten gelesen und geschrieben.

Java unterscheidet zwei fundamentale unterschiedliche Arten von Datenströmen:

- **Zeichenenströme**: Die Grundeinheit der Datenkommunikation sind Buchstaben und andere Schriftzeichen. Hierbei kann es sich um beliebige Zeichen aus dem Unicode handeln, also Buchstaben so gut wie jeder bekannten Schrift, von lateinischer über kyrillische, griechische, arabische, chinesische bis hin zu exotischen Schriften wie der keltischen Keilschrift. Dabei ist entscheidend, in welcher Codierurung die Buchstaben in der Datenquelle vorliegen. Die Codierung wird in der Regel beim Konstruieren eines Datenstroms festgelegt. Geschieht dieses nicht, so wird die Standardcodierung des Systems, auf dem das Programm läuft, benutzt.
- **Byteströme (Oktettströme)**: Hierbei ist die Grundeinheit immer ein Byte, das als Zahl verstanden wird.

7.6.1 Zeichenströme

Zwei Hauptoberklassen für Zeichenenströme stehen zur Verfügung:

- `Reader`: für Ströme, aus denen gelesen werden soll.
- `Writer`: für Ströme, in die geschrieben werden soll.

Entsprechende Unterklassen stehen zur Verfügung für die verschiedenen Arten von Datenquellen, so z.B. `FileInputStream` und `FileOutputStream`.

7.6.2 Byteströme

Entsprechend gibt es auch zwei Hauptklassen zum Lesen bzw. Schreiben von Byteströmen:

- `InputStream`
- `OutputStream`

Byteströme sind dafür gedacht, binäre Dateien zu lesen und zu schreiben, wie z.B. Bilddateien.

Auch hier stehen verschiedene Unterklassen für verschiedene Arten von Datenquellen zur Verfügung. Für Dateien sind dieses entsprechend die Klassen `FileInputStream` und `FileOutputStream`.

7.6.3 Ausnahmen

Ein-/Ausgabeoperationen sind fehleranfällig. Externe Datenquellen können fehlerhaft sein, nicht existieren oder sich unerwartet verhalten. Daher kann es bei fast allen Methoden auf Datenströmen zu Ausnahmen kommen, die alle von der Oberklasse `IOException` ableiten.

7.6.4 Schreiben und Lesen

Sowohl für Zeichenströme als auch für Byteströme stehen Methoden `int read()` bzw. `int write(int c)` zur Verfügung. Die Methode `read` liefert entweder einen positiven Wert oder den Wert `-1`, der anzeigt, daß keine weitere Daten im Datenstrom vorhanden sind.

7.6.5 Datenströme

Zum Schreiben und Lesen in Dateien haben die entsprechenden Stromklassen Konstruktoren, die einen Datenstrom für einen Dateinamen erzeugen. Mit der Konstruktion wird der Strom geöffnet. Java schließt den Strom automatisch wieder, wenn das Objekt nicht mehr benutzt wird; da dieser Zeitpunkt abhängig davon ist, wann zufällig die Speicherverwaltung ein Objekt aus dem Speicher löscht, ist es zu empfehlen, immer explizit die Methode `close` für solche Datenströme aufzurufen, die nicht mehr benutzt werden.

Textdateien lesen

Zum Lesen einer Textdatei läßt sich bequem die Klasse `FileReader` benutzen. Im folgenden Programm wird eine über die Kommandozeile spezifizierte Datei gelesen und jedes darin enthaltene Zeichen zusammen mit seiner Unicodenummer auf dem Bildschirm ausgegeben:

ShowFile.java

```
import java.io.*;
class ShowFile{
  public static void main(String [] args) throws Exception{
    Reader reader = new FileReader(args[0]);
    while (true){
      final int next = reader.read();
      if (next==-1) break;
      System.out.print((char)next);
      System.out.print(next);
    }
  }
}
```

Textdateien schreiben

Entsprechend gibt es auch eine Klasse, die zum Schreiben einer Textdatei bequem benutzt werden kann: `FileWriter`. So läßt sich ein Programm zum Kopieren einer Datei wie folgt schreiben:

Copy.java

```
import java.io.*;

public class Copy {
  public static void main(String[] args) throws IOException {
    FileReader in = new FileReader(args[0]);
    FileWriter out = new FileWriter(args[1]);
```



```
    int c;

    while ((c = in.read()) != -1){
      out.write(c);
    }
  }
}
```

7.6.6 Vom InputStream zum Reader, vom OutputStream zum Writer

`Reader` und `Writer` sind praktische Klassen zur Verarbeitung von Zeichenströmen. Primär sind aber auch Textdateien lediglich eine Folge von Bytes.

InputStreamReader und OutputStreamWriter

Mit den Klassen `InputStreamReader` und `OutputStreamWriter` lassen sich Objekte vom Typ `IntputStream` bzw. `OutputStream` zu `Reader`- bzw. `Writer`-Objekten machen.

> **Beispiel:**
> Statt die vorgefertigte Klasse `FileWriter` zum Schreiben einer Textdatei zu benutzen, erzeugt die folgende Version zum Kopieren von Dateien einen über einen `FileOutputStream` erzeugten `Writer` bzw. einen über einen `FileInputStream` erzeugten `Reader`:

Copy2.java

```
import java.io.*;

class Copy2 {
  static public void main(String [] args)
     throws Exception {
    Reader reader = new FileReader(args[0]);
    Writer writer
     = new OutputStreamWriter(new FileOutputStream(args[1]));

    int c;
    while ((c = reader.read()) != -1){
      writer.write(c);
    }
    writer.close();
  }
}
```

Zeichenkodierungen

Java-Strings sind Zeichenketten, die nicht auf eine Kultur mit einer bestimmten Schrift beschränkt , sondern in der Lage sind, alle im Unicode erfassten Zeichen darzustellen; seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines Strings kann potentiell eines dieser mehreren zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige, kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen, die acht Bit im Speicher belegen, ein sogenanntes Byte. Ein Byte ist in der Lage, 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments) darzustellen. Für ein chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Codierung* eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem ß zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit zwei Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet.

Man sieht, daß es Codierungen geben muß, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüberhinaus auch eine Reihe Mischformen; manche Zeichen werden durch ein Byte, andere durch zwei oder sogar durch zwei Byte dargestellt.

Die Klasse `OutputStreamWriter` sieht einen Konstruktor vor, dem man zusätzlich zum `OutputStream`, in den geschrieben werden soll, als zweites Element auch die Codierung angeben kann, in der die Buchstaben abgespeichert werden sollen. Wenn diese Codierung nicht explizit angegeben wird, so benutzt Java die standardmäßig auf dem Betriebssystem benutzte Codierung.

Beispiel:
In dieser Version der Kopierung einer Textdatei wird für den `Writer` ein Objekt der Klasse `OutputStreamWriter` benutzt, in der als Zeichenkodierung utf-16 benutzt wird.

EncodedCopy.java

```java
import java.nio.charset.Charset;
import java.io.*;

class EncodedCopy {
  static public void main(String [] args)
     throws Exception {
    Reader reader = new FileReader(args[0]);
    Writer writer = new OutputStreamWriter
                         (new FileOutputStream(args[1])
                         ,Charset.forName("UTF-16"));

    int c;
```



```java
    while ((c = reader.read()) != -1){
      writer.write(c);
    }
    writer.close();
  }
}
```

Betrachtet man die Größe der geschriebenen Datei, so wird man feststellen, daß sie mehr als doppelt so groß ist wie die Ursprungsdatei.

```
sep@linux:~/fh/prog1/> java EncodedCopy EncodedCopy.java EncodedCopyUTF16.java
sep@linux:~/fh/prog1/> ls -l EncodedCopy.java
-rw-r--r--    1 sep      users         443 2004-01-07 19:12 EncodedCopy.java
sep@linux:~/fh/prog1/> ls -l EncodedCopyUTF16.java
-rw-r--r--    1 sep      users         888 2004-01-07 19:13 EncodedCopyUTF16.java
sep@linux:~/fh/prog1/>
```

Aufgabe 25 Schreiben Sie ein Programm `FileConvert` zum Konvertieren von Textdateien in eine andere Kodierung. Dem Programm sollen über die Kommandozeilenparameter der Name der Eingabedatei, der Name der Ausgabedatei und der Name der benutzten Codierung übergeben werden. Ein möglicher Aufruf wäre also:

```
linux:~/>java FileConvert test.txt konvertiertTest.txt utf-8
```

Lassen Sie eine deutsche Textdatei mit Umlauten in eine Datei mit der Codierung utf-8 konvertieren. Betrachten Sie die Ergebnisdatei. Was stellen Sie fest?

Gepufferte Ströme

Die bisher betrachteten Ströme arbeiten immer exakt zeichenweise, bzw. byteweise. Damit wird bei jedem `read` und bei jedem `write` direkt von der Quelle bzw. an die Senke ein Zeichen übertragen. Für Dateien heißt das, es wird über das Betriebssystem auf die Datei auf der Festplatte zugegriffen. Handelt es sich bei Quelle/Senke um eine teure und aufwändige Netzwerkverbindung, so ist für jedes einzelne Zeichen über diese Netzwerkverbindung zu kommunizieren. Da in der Regel nicht nur einzelne Zeichen über einen Strom übertragen werden sollen, ist es effizienter, wenn technisch gleich eine Menge von Zeichen übertragen wird. Um dieses zu bewerkstelligen, bietet java an, Ströme in gepufferte Ströme umzuwandeln.

Ein gepufferter Strom hat einen internen Speicher. Bei einer Datenübertragung wird für schreibende Ströme erst eine Anzahl von Zeichen in diesem Zwischenspeicher abgelegt, bis dieser seine Kapazität erreicht hat, um dann alle Zeichen aus dem Zwischenspeicher *en bloc* zu übertragen. Für lesende Ströme wird entsprechend für ein `read` gleich eine ganze Anzahl von Zeichen von der Datenquelle geholt und im Zwischenspeicher abgelegt. Weitere `read`-Operationen holen dann die Zeichen nicht mehr direkt aus der Datenquelle, sondern aus dem Zwischenspeicher, bis dieser komplett ausgelesen wurde und von der Datenquelle wieder zu füllen ist.

Die entsprechenden Klassen, die Ströme in gepufferte Ströme verpacken, heißen:
`BufferedInputStream`, `BufferedOutputStream` und
entsprechend `BufferedReader`, `BufferedWriter`.

Beispiel:
Jetzt ergänzen wir zur Effizienzsteigerung noch das Kopierprogramm, so daß der benutzte `Writer` gepuffert ist:

BufferedCopy.java

```java
import java.io.*;
import java.nio.charset.Charset;

class BufferedCopy{
  static public void main(String [] args)
     throws Exception {

    Reader reader = new FileReader(args[0]);
    Writer writer = new BufferedWriter
                       (new OutputStreamWriter
                           (new FileOutputStream(args[1])
                           ,Charset.forName("UTF-16")));

    int c;
    while ((c = reader.read()) != -1){
      writer.write(c);
    }
    writer.close();
  }
}
```

7.6.7 Ströme für Objekte

Bisher haben wir uns darauf beschränkt, Zeichenketten über Ströme zu lesen und zu schreiben. Java bietet darüberhinaus die Möglichkeit an, beliebige Objekte über Ströme zu schreiben und zu lesen. Hierzu können mit den Klassen `ObjectOutputStream` und `ObjectInputStream` beliebige `OutputStream`- bzw. `InputStream`-Objekte zu Strömen für Objekte gemacht werden.

In diesen Klassen stehen Methoden zum Lesen und Schreiben von Objekten zur Verfügung. Allerdings können über diese Ströme nur Objekte von Klassen geschickt werden, die die Schnittstelle `java.io.Serializable` implementieren. Die meisten Standardklassen implementieren diese Schnittstelle. `Serializable` enthält keine Methoden, es reicht also zum Implementieren aus, die Klausel `implements Serializable` für eine Klasse zu benutzen, damit Objekte der Klasse über Objektströme geschickt werden können.

Objektströme haben zusätzlich Methoden zum Lesen und Schreiben von primitiven Typen.

Beispiel:
Folgendes Testprogramm schreibt eine Zahl und ein Listenobjekt in eine Datei, um diese anschließend wieder aus der Datei auszulesen.

WriteReadObject.java

```java
import java.io.*;
import java.util.List;
import java.util.ArrayList;

```



```java
public class WriteReadObject {
  public static void main(String [] args) throws Exception{
    FileOutputStream fos = new FileOutputStream("t.tmp");
    ObjectOutputStream oos = new ObjectOutputStream(fos);

    List xs = new ArrayList();
    xs.add("the");
    xs.add("world");
    xs.add("is");
    xs.add("my");
    xs.add("oyster");

    oos.writeInt(12345);
    oos.writeObject(xs);
    oos.close();

    FileInputStream fis = new FileInputStream("t.tmp");
    ObjectInputStream ois = new ObjectInputStream(fis);

    int i = ois.readInt();
    List ys = (List) ois.readObject();

    ois.close();

    System.out.println(i);
    System.out.println(ys);
  }
}
```

Aufgabe 26 Erweitern Sie die Klasse `Li`, so daß Sie Ihre Listenobjekte in Dateien schreiben und wieder aus Dateien lesen können. Testen Sie ihre Implementierung.

7.6.8 Stromloses IO

Die Ein- und Ausgabe mit Stromklasse zwingt immer, eine Datenquelle vom Anfang an durchzulesen, bis man zu den Daten der Quelle kommt, die einen interessieren. Oft interessiert man sich aber nur für bestimmte Daten irgendwo in der Mitte einer Datei. Ein typisches Beispiel wäre ein Dateiformat eines Textdokumentes. Am Anfang der Datei ist eine Tabelle, die angibt, auf welcher Position in der Datei sich die einzelnen Seiten befinden. Ist man nur an einer bestimmten Seite des Dokuments interessiert, so kann man ihren Index in der Datei der Tabelle am Anfang entnehmen und dann direkt diese Seite lesen, ohne alle vorherigen Seiten betrachten zu müssen. Für eine solche Arbeitsweise stellt Java die Klasse `java.io.RandomAccessFile` zur Verfügung.

Aufgabe 27 Studieren Sie die Dokumentation von `java.io.RandomAccessFile` und schreiben Sie einige Testbeispiele zur Benutzung dieser Klasse.

Kapitel 8

Erweiterungen in Java 1.5

8.1 Einführung

Mit Java 1.5 werden einige neue Konzepte in Java eingeführt, die viele Programmierer bisher schmerzlich vermisst haben. Obwohl die ursprünglichen Designer über diese neuen Konstrukte von Anfang an nachgedacht haben und sie auch gerne in die Sprache integriert hätten, schaffen diese Konstrukte es erst jetzt nach vielen Jahren in die Sprache Java. Hierfür kann man zwei große Gründe angeben:

- Beim Entwurf und der Entwicklung von Java waren die Entwickler bei Sun unter Zeitdruck.
- Die Sprache Java wurde in ihren programmiersprachlichen Konstrukten sehr konservativ entworfen. Die Syntax wurde von C übernommen. Die Sprache sollte möglichst wenige aber mächtige Eigenschaften haben und kein Sammelsurium verschiedenster Techniken sein.

Seitdem Java eine solch starke Bedeutung als Programmiersprache erlangt hat, gibt es einen definierten Prozess, wie neue Eigenschaften der Sprache hinzugefügt werden, den *Java community process (JPC)*. Hier können fundierte Verbesserungs- und Erweiterungsvorschläge gemacht werden. Wenn solche von genügend Leuten unterstützt werden, kann ein sogenannter *Java specification request (JSR)* aufgemacht werden. Hier wird eine Expertenrunde gegründet, die die Spezifikation und einen Prototypen für die Javaerweiterung erstellt. Die Spezifikation und prototypische Implementierung werden schließlich öffentlich zur Diskussion gestellt. Wenn es keine Einwände von irgendwelcher Seite mehr gibt, wird die neue Eigenschaft in eine der nächsten Javaversionen integriert.

Mit Java 1.5 findet das Ergebnis einer Vielzahl JSRs Einzug in die Sprache. DieProgrammiersprache wird in einer Weise erweitert, wie es schon lange nicht mehrder Fall war. Den größten Einschnitt stellt sicherlich die Erweiterung des Typsystems auf generische Typen dar. Aber auch die anderen neuen Kosntrukte, die verbesserte `for`-Schleife, Aufzählungstypen, statisches Importieren und automatisches Boxen, sind keine esoterischen Eigenschaften, sondern werden das alltägliche Arbeiten mit Java beeinflussen. In den folgenden Abschnitten werfen wir einen Blick auf die neuen Javaeigenschaften im Einzelnen.

8.2 Generische Typen

Generische Typen wurden im JSR014 definiert. In der Expertengruppe des JSR014 war der Autor dieses Skriptszeitweilig 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 bald 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 *patternmatching* 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 Expertenrundedes JSR014 hat *GJ* als Grundlage für die Spezifikation genommen und anden grundlegenden Prinzipien auch nichts mehr geändert.

8.2.1 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

```
class OldBox {
  Object contents;
  OldBox(Object contents){this.contents=contents;}
}
```

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 sinnvollnutzen zu können, ist eine dynamische Typzusicherung durchzuführen:

UseOldBox.java

```
class UseOldBox{
  public static void main(String [] _){
    OldBox b = new OldBox("hello");
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
    System.out.println(((String) s).toUpperCase());
  }
}
```

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

```java
class UseOldBoxError{
  public static void main(String [] _){
    OldBox b = new OldBox(new Integer(42));
    String s = (String)b.contents;
    System.out.println(s.toUpperCase());
  }
}
```

```
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 mehrstatisch 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. Hierzuersetzen wir in der obigen Klasse jedes Auftreten des Typs `Object` durch einen Variablennamen. Diese Variable ist eineTypvariable. 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

```java
class Box<elementType> {
  elementType contents;
  Box(elementType contents){this.contents=contents;}
}
```

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 darinenthalten 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 angehängtwird. Wir erhalten dann z.B. den Typ `Box<String>`, um Stringsin der Schachtel zu speichern, oder `Box<Integer>`, um Integerobjekte darin zu speichern:

UseBox.java

```java
class UseBox{
  public static void main(String [] _){
    Box<String> b1 = new Box<String>("hello");
    String s = b1.contents;
    System.out.println(s.toUpperCase());
    System.out.println(b1.contents.toUpperCase());

    Box<Integer> b2 = new Box<Integer>(new Integer(42));

    System.out.println(b2.contents.intValue());
  }
}
```

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:

```java
class UseBoxError{
  public static void main(String [] _){
    Box<String> b = new Box<String>(new Integer(42));
    String s = b.contents;
    System.out.println(s.toUpperCase());
  }
}
```

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 Objektorientierungist. Von generischen Klassen lassen sich in gewohnter Weise Unterklassendefinieren. Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein. So können wir unsere einfache Schachtelklasse erweitern, so daß wir zwei Objekte speichern können:

Pair.java
```
class Pair<at,bt> extends Box<at>{
  Pair(at x,bt y){
    super(x);
    snd = y;
  }

  bt snd;

  public String toString(){
    return "("+contents+","+snd+")";
  }
}
```

Die Klasse `Pair` hat zwei Typvariablen. Instanzen von `Pair` müssen angeben von welchem Typ die beiden zu speichernden Objekte sein sollen.

UsePair.java
```
class UsePair{
  public static void main(String [] _){
    Pair<String,Integer> p
     = new Pair<String,Integer>("hallo",new  Integer(40));

    System.out.println(p);
    System.out.println(p.contents.toUpperCase());
    System.out.println(p.snd.intValue()+2);
  }
}
```

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
```
class UniPair<at> extends Pair<at,at>{
  UniPair(at x,at y){super(x,y);}
  void swap(){
    final at z = snd;
    snd = contents;
    contents = z;
  }
}
```


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
```
class UseUniPair{
  public static void main(String [] _){
    UniPair<String> p
     = new UniPair<String>("welt","hallo");

    System.out.println(p);
    p.swap();
    System.out.println(p);
  }
}
```

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 Klasseerweitern, in der nur noch Stringobjekte verpackt werden können:

StringBox.java
```
class StringBox extends Box<String>{
  StringBox(String x){super(x);}
}
```

Diese Klasse kann nun vollkommen ohne spitze Klammern benutzt werden:

UseStringBox.java
```
class UseStringBox{
  public static void main(String [] _){
    StringBox b = new StringBox("hallo");
    System.out.println(b.contents.length());
  }
}
```

Einschränken der Typvariablen

Bisher standen in allen Beispielen die Typvariablen einer generischen Klassefür 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 größerist 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

```
class CollectMaxOld{
  private Comparable value;

  CollectMaxOld(Comparable x){value=x;}

  void setValue(Comparable x){
    if (value.compareTo(x)<0) value=x;
  }

  Comparable getValue(){return value;}
}
```

Die Klasse `CollectMaxOld` ist in der Lage, beliebige Objekte, die die Schnittstelle `Comparable` implementieren, zu speichern. Wir habenwieder dasselbe Problem wie in der Klasse `OldBox`: Greifen wir auf dasgespeicherte 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. Wirerhalten somit eine generische Klasse `CollectMax`:

CollectMax.java

```
class CollectMax <elementType extends Comparable>{
  private elementType value;

  CollectMax(elementType x){value=x;}

  void setValue(elementType x){
    if (value.compareTo(x)<0) value=x;
  }

  elementType getValue(){return value;}
}
```

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 Objektdes Typs `Comparable`, sondern exakt ein Objekt des Instanztyps.

UseCollectMax.java

```
class UseCollectMax {
  public static void main(String [] _){
    CollectMax<String> cm = new CollectMax<String>("Brecht");
    cm.setValue("Calderon");
    cm.setValue("Horvath");
    cm.setValue("Shakespeare");
    cm.setValue("Schimmelpfennig");
    System.out.println(cm.getValue().toUpperCase());
  }
}
```

Wie man in der letzten Zeile sieht, entfällt wieder die dynamische Typzusicherung.

8.2.2 Generische Schnittstellen

Generische Typen erlauben es, den Typ `Object` in Typsignaturen zu eleminieren. Der Typ `Object` ist als schlecht anzusehen, denn er istgleichbedeutend damit, daß keine Information über einen konkreten Typ währenddder 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 miteinandervergleichen. In diesem Abschnitt werden wir sehen, daß generische Typen es uns erlauben, allgemein eine Gleichheitsmethode zu definieren, in der nur Objektegleichen Typs miteinander verglichen werden können. Hierzu werden wir eine generische Schnittstelle definieren.

Generische Typen erweitern sich ohne Umstände auf Schnittstellen. Im Vergleichzu 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

```
interface EQ<otherType> {
  public boolean eq(otherType other);
}
```

Jetzt können wir für jede Klasse nicht nur bestimmen, daß sie die Gleichheitimplementieren 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 dieMethode `eq` implementieren, und zwar mit dem Typ `Apfel` als Parametertyp:

Apfel.java
```
class Apfel implements EQ<Apfel>{
  String typ;

  Apfel(String typ){
    this.typ=typ;}

  public boolean eq(Apfel other){
    return this.typ.equals(other.typ);
  }
}
```

Jetzt können wir Äpfel mit Äpfeln vergleichen:

TestEQ.java
```
class TestEq{
  public static void main(String []_){
    Apfel a1 = new Apfel("Golden Delicious");
    Apfel a2 = new Apfel("Macintosh");
    System.out.println(a1.eq(a2));
    System.out.println(a1.eq(a1));
  }
}
```

Schreiben wir als nächstes eine Klasse die Birnen darstellen soll. Auch dieseimplementiere die Schnittstelle `EQ`, und zwar dieses Mal für Birnen:

Birne.java
```
class Birne implements EQ<Birne>{
  String typ;

  Birne(String typ){
    this.typ=typ;}

  public boolean eq(Birne other){
    return this.typ.equals(other.typ);
  }
}
```

Während des statischen Typchecks wird überprüft, ob wir nur Äpfel mit Äpfelnund Birnen mit Birnen vergleichen. Der Versuch, Äpfel mit Birnen zuvergleichen, führt zu einem Typfehler:

```
class TesteEqError{
  public static void main(String []_){
    Apfel a = new Apfel("Golden Delicious");
    Birne b = new Birne("williams");
    System.out.println(a.equals(b));
    System.out.println(a.eq(b));
```



```
  }
}
```

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 einmalpassiert, daß er zwei Objekte verglichen hat, die er gar nicht vergleichen wollte. Da der statische Typcheck solche Fehler nicht erkennen kann, denn dieMethode `equals` läßt jedes Objekt als Parameter zu, sind solche Fehler mitunter schwer zu lokalisieren.

Der statische Typcheck stellt auch sicher, daß eine generische Schnittstellemit 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` zuimplementieren, führt ebenfalls zu einer Fehlermeldung:

```
class BirneError implements EQ<Apfel>{
  String typ;

  BirneError(String typ){
    this.typ=typ;}

  public boolean eq(Birne other){
    return this.typ.equals(other.typ);
  }
}
```

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>
```

8.2.3 Kovarianz gegen Kontravarianz

Gegeben seien zwei Typen `A` und `B`. Der Typ `A` soll Untertyp des Typs `B` sein, also entweder ist `A` eineUnterklasse der Klasse `B` oder `A` implementiert die Schnittstelle `B` oder die Schnittstelle `A` erweitert dieSchnittstelle `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:

```
class Kontra{
  public static void main(String []_){
    Box<Object> b  = new Box<String>("hello");
  }
}
```

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>`. DerGrund 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

```
class Ko{
  public static void main(String []_){
    String [] x = {"hello"};
    Object [] b  = x;
    b[0]=new Integer(42);
    x[0].toUpperCase();
  }
}
```

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.

8.2.4 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

```
import java.util.*;
import java.util.List;
class ListTest{
  public static void main(String [] _){
    List<String> xs = new ArrayList<String>();
    xs.add("Schimmelpfennig");
    xs.add("Shakespeare");
    xs.add("Horvath");
    xs.add("Brecht");
    String x2 = xs.get(1);
    System.out.println(xs);
  }
}
```

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

```
import java.util.*;
import java.util.List;
class WarnTest{
  public static void main(String [] _){
    List xs = new ArrayList<String>();
    xs.add("Schimmelpfennig");
    xs.add("Shakespeare");
    xs.add("Horvath");
    xs.add("Brecht");
    String x2 = (String)xs.get(1);
    System.out.println(xs);
  }
}
```

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>
```

8.2.5 Generische Methoden

Bisher haben wir generische Typen für Klassen und Schnittstellen betrachtet. Generische Typen sind aber nicht an einen objektorientiertenKontext 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 Methodebenutzten 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.

8.3 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 eigenesSchleifenkonstrukt, 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:
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());
    }
  }
}
```

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:
Damit lassen sich jetzt die Iterationen der letzten beiden Schleifen wesentlich eleganter ausdrücken.

NewIteration.java
```
import java.util.List;
import java.util.ArrayList;

class NewIteration{

```

[1]Ein noch sprechenderes Konstrukt wäre gewesen, wenn man statt des Doppelpunkts das Schlüsselwort `in` benutzt hätte. AusAufwärtskompatibilitätsgründen wird jedoch darauf verzichtet, neue Schlüsselwörter in Java einzuführen.

```
  public static void main(String [] _){
    String [] ar
     = {"Brecht","Horvath","Shakespeare","Schimmelpfennig"};
    List<String> xs = new ArrayList<String>();

    for (String s:ar) xs.add(s);
    for (String s:xs) System.out.println(s.toUpperCase());
  }
}
```

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.

8.3.1 Die neuen Schnittstellen Iterable und SimpleIterator

Beispiel:

ReaderIterator.java

```
package sep.util.io;

import java.io.Reader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Iterator;


public class ReaderIterator
              implements Iterable<Character>
                        ,Iterator<Character>{
  private Reader reader;
  private int n;
  public ReaderIterator(Reader r){
    reader=new BufferedReader(r);
    try{n=reader.read();
    }catch(IOException _){n=-1;}
  }
  public Character next(){
    Character result = new Character((char)n);
    try{n=reader.read();
    }catch(IOException _){n=-1;}
    return result;
  }

  public boolean hasNext(){
    return n!=-1;
  }

```



```
  public void remove(){
    throw new UnsupportedOperationException();
  }

  public Iterator<Character> iterator(){return this;}
}
```

TestReaderIterator.java

```
import sep.util.io.ReaderIterator;
import java.io.FileReader;

class TestReaderIterator {
  public static void main(String [] args) throws Exception{
      Iterable<Character> it
        =new ReaderIterator(new FileReader(args[0]));
      for (Character c:it){
      System.out.print(c);
    }
  }
}
```

Beispiel:
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 einerListe von Wörtern wieder einen String zu bilden:

TextUtils.java

```
import java.util.*;
import java.util.List;

class TextUtils {

  static List<String> words (String s){
    final List<String> result = new ArrayList<String>();

    StringBuffer currentWord = new StringBuffer();

    for (char c:s.toCharArray()){
      if (Character.isWhitespace(c)){
        final String newWord = currentWord.toString().trim();
        if(newWord.length()>0){
          result.add(newWord);
          currentWord=new StringBuffer();
        }
      }else{currentWord.append(c);}
    }
    return result;
  }

  static String unwords(List<String> xs){
```

```
    StringBuffer result=new StringBuffer();
    for (String x:xs) result.append(" "+x);
    return result.toString().trim();
  }

  public static void main(String []_){
    List<String> xs = words("  the   world  is my Oyster  ");

    for (String x:xs) System.out.println(x);

    System.out.println(unwords(xs));
  }
}
```

8.4 Automatisches Boxen

8.5 Aufzählungstypen

8.6 Ein paar Beispielklassen

UnaryFunction.java

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

public interface UnaryFunction<arg,result>{
  public result eval(arg a);
}
```

Tuple1.java

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

public class Tuple1<t1> {
  public t1 e1;
  public Tuple1(t1 a1){e1=a1;}
  String parenthes(Object o){return "("+o+")";}
  String simpleToString(){return e1.toString();}
  public String toString(){return parenthes(simpleToString());}
  public boolean equals(Object other){
    if (! (other instanceof Tuple1)) return false;
    return e1.equals(((Tuple1)other).e1);
  }
}
```

Tuple2.java

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

public class Tuple2<t1,t2> extends Tuple1<t1>{
  public t2 e2;
  public Tuple2(t1 a1,t2 a2){super(a1);e2=a2;}
```



```
  String simpleToString(){
    return super.simpleToString()+","+e2.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple2)) return false;
    return super.equals(other)&& e2.equals(((Tuple2)other).e2);
  }
}
```

Tuple3.java

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

public class Tuple3<t1,t2,t3> extends Tuple2<t1,t2>{
  public t3 e3;
  public Tuple3(t1 a1,t2 a2,t3 a3){super(a1,a2);e3=a3;}
  String simpleToString(){
    return super.simpleToString()+","+e3.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple3)) return false;
    return super.equals(other)&& e3.equals(((Tuple3)other).e3);
  }
}
```

Tuple4.java

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

public class Tuple4<t1,t2,t3,t4> extends Tuple3<t1,t2,t3>{
  public t4 e4;
  public Tuple4(t1 a1,t2 a2,t3 a3,t4 a4){super(a1,a2,a3);e4=a4;}
  String simpleToString(){
    return super.simpleToString()+","+e4.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple4)) return false;
    return super.equals(other)&& e4.equals(((Tuple4)other).e4);
  }
}
```

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

HsMaybe.hs
```
data Maybe a = Nothing|Just a
```

Maybe.java
```
package name.panitz.crempel.util;
public interface Maybe<a> {}
```

Nothing.java
```
package name.panitz.crempel.util;
public class Nothing<a> implements Maybe<a>{

  public String toString(){return "Nothing("+")";}
  public boolean equals(Object other){
    return (other instanceof Nothing);
  }
}
```

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

public class Just<a> implements Maybe<a>{
  private a just;

  public Just(a just){this.just = just;}
  public a getJust(){return just;}

  public String toString(){return "Just("+just+")";}
  public boolean equals(Object other){
    if (!(other instanceof Just)) return false;
    final Just o= (Just) other;
    return just.equals(o.just);
  }
}
```

CrempelTool.java
```
package name.panitz.crempel.tool;
public interface CrempelTool{String getDescription();void startUp();}
```

Kapitel 9

Steuerfäden

Bisher sind wir nur in der Lage, Programme zu beschreiben, deren Abarbeitung streng sequentiell nacheinander im Programmtext festgelegt ist. Insbesondere wenn man graphische Komponenten programmiert, möchte man Progammteile haben, die nebeneinander her arbeiten. In zwei Fenstern können graphisch animierte Objekte stehen, und wir wollen nicht, daß erst das eine und dann, wenn dieses beendet ist, das andere Fenster seine Animation spielt.

Trotzdem sollen die beiden Fenster nicht zu unterschiedlichen Programmen gehören, sondern Bestandteil eines Programms sein, und auch in der Lage sein, mit denselben Objekten zu arbeiten.

Um nebenläufige Programmteile zu programmieren, stellt Java das Konzept der Steuerfäden zur Verfügung. Steuerfäden sind zu unterscheiden von Prozessen im Betriebssystem. Die Prozesse eines Betriebssystems haben eine eigene Speicherumgebung, während die Steuerfäden in Java, alle in derselben virtuellen Maschine laufen und sich einen Speichbereich für ihre Daten teilen und sich darin auch Daten teilen können.

9.1 Schreiben von Steuerfäden

Es gibt zwei Arten, wie in Java Steuerfäden umgesetzt werden können:

- Durch Ableiten von der Klasse `Thread` und überschreiben der Methode `run`.
- Durch Implementierung der Schnittstelle `Runnable`, die Methode `run` enthält.

In der Regel wird man die Klasse `Thread` überschreiben und nur die Schnittstelle `Runnable` selbst implementieren, wenn das Verbot der mehrfachen Erbung, verhindert, daß von der Klasse `Thread` abgelitten werden kann.

Java vergibt bei der Erzeugung eines Steuerfadens jedem Steuerfaden einen Namen, der mit der Methode `getName()` abgefragt werden kann. Dieser Name kann aber auch im Konstruktor gesetzt werden.

Für ein Objekt der Klasse `Thread` kann nach seiner Erzeugung die Methode `start()` aufgerufen werden, die dafür sorgt, daß der Code der Methode `run()` ausgeführt wird. Wurden mehrere Steuerfäden gestartet, so wird Java mehr oder weniger zufällig dafür sorgen, daß beide Steuerfäden Gelegenheit bekommen, ihre `run`-Methode auszuführen. Es gibt keinerlei Garantie dafür, wie fair und in welcher Weise Java dafür sorgt, daß alle Steuerfäden ihre Methode `run` ausführen können. Insbesondere können Programme, die Steuerfäden benutzen, auf unterschiedlichen Betriebssystem ganz unterschiedliche Verhalten aufweisen.

Beispiel:
Folgendes Programm definiert einen simplen Steuerfaden, der unendlich oft seinen Namen auf den Bildschirm ausgibt. In der `main`-Methode werden zwei Instanzen dieses Steuerfadens erzeugt und gestartet. Anhand der Ausgabe des Programms kann verfolgt werden, wie Java zwischen den zwei Steuerfäden umschaltet:

SimpleThread.java

```
class SimpleThread extends Thread {

  public void run(){
    while (true){
      System.out.print(getName());
    }
  }

  public static void main(String [] args){
    new SimpleThread().start();
    new SimpleThread().start();
  }
}
```

9.1.1 Schlafenlegen von Prozessen

In der Klasse `Thread` gibt es eine statische Methode `void sleep(long millis)`. Der Aufruf dieser Methode bewirkt, daß der aktuell laufende Steuerfaden entsprechend seines Parameters für einen bestimmten Zeitraum nicht weiterarbeitet. Die Einheit für den Parameter sind tausendstel Sekunden.

Beispiel:
Unseren ersten Steuerfaden aus dem letzten Beispiel erweitern wir jetzt darum, daß er sich jeweils eine bestimmte Zeit zur Ruhe legt, nachdem er seinen Namen ausgedruckt hat:

SleepingThread.java

```
class SleepingThread extends Thread {
  public SleepingThread(int s){this.s=s;}
  private int s;

  public void run(){
    while (true){
      System.out.print(getName());
      try {Thread.sleep(s);}catch (InterruptedException _){}
    }
  }
```



```

  public static void main(String [] args){
    new SleepingThread(1000).start();
    new SleepingThread(300).start();
  }
}
```

Wenn das Programm läuft, sieht man, daß der Steuerfaden, der kürzer schläft, häufiger seinen Namen ausgibt:

```
sep@swe10:~/fh/internal/beispiele> java SleepingThread
Thread-1Thread-0Thread-1Thread-1Thread-1Thread-0Thread-1Thread-1Thread-1
Thread-0Thread-1Thread-1Thread-1Thread-1Thread-0Thread-1Thread-1Thread-1
Thread-0Thread-1Thread-1Thread-1Thread-0Thread-1Thread-1Thread-1Thread-0
Thread-1Thread-1Thread-1Thread-0Thread-1Thread-1Thread-1Thread-1Thread-0
Thread-1Thread-1Thread-1Thread-1Thread-0Thread-1Thread-1Thread-0Thread-1
Thread-1Thread-1Thread-0Thread-1Thread-1Thread-1Thread-1Thread-0Thread-1
Thread-1Thread-1Thread-0Thread-1Thread-1Thread-1Thread-0
sep@swe10:~/fh/internal/beispiele>
```

Man beachte, daß die Methode `sleep` eine statische Methode und keine Objektmethode für die Klasse `Thread` ist. Sie wird also nicht auf den Steuerfadenobjekten ausgeführt, sondern global und gilt für den gerade aktiven Steuerfaden. Das wird dann jeweils der Steuerfaden sein, in dessen `run`-Methode der Aufruf von `sleep` steht.

9.2 Koordination nebenläufiger Steuerfäden

Wenn zwei Steuerfäden sich Daten teilen, dann kann es zu Problemen kommen.

9.2.1 Benutzung gemeinsamer Objekte

Wir schreiben ein kleines Objekt, das zwei Zahlen enthält und eine Methode `swap`, die die beiden Zahlen in dem Objekt vertauscht. Wir machen die Methode `swap` durch eine `sleep`-Anweisung künstlich langsam:

TwoNumbers.java

```
class TwoNumbers{
   int x;
   int y;

   TwoNumbers(int x, int y) {this.x=x; this.y=y;}

   void swap(){
     int z=x;
     x=y;
     try {Thread.sleep(100);}catch (InterruptedException _){}
     y=z;
   }

```

```
  public String toString(){return "("+x+","+y+")";}
}
```

Jetzt können wir zwei Steuerfäden schreiben. Der eine ruft immer wieder die Methode `swap` auf ein `TwoNumbers`-Objekt auf:

Swap.java
```
class Swap extends Thread{

  public Swap(TwoNumbers twoN){this.twoN=twoN;}

  private TwoNumbers twoN;

  public void run(){
    while (true){twoN.swap();}
  }
}
```

Der andere druckt immer wieder ein `TwoNumbers`-Objekt aus:

PrintTwoNumbers.java
```
class PrintTwoNumbers extends Thread {
  public PrintTwoNumbers(TwoNumbers twoN){this.twoN=twoN;}

  private TwoNumbers twoN;

  public void run(){
    while (true){
      System.out.println(twoN);
      try {Thread.sleep(1000);}catch (InterruptedException _){}
    }
  }

  public static void main(String [] _){
    TwoNumbers twoN = new TwoNumbers(1,2);
    new Swap(twoN).start();
    new PrintTwoNumbers(twoN).start();
  }
}
```

Wenn wir dieses Programm starten, dann sehen wir, daß wir so gut wie nie die beiden Zahlen ausdrucken, sondern immer ein Paar zwei gleicher Zahlen:

```
sep@swe10:~/fh/internal/beispiele> java PrintTwoNumbers
(1,2)
(1,1)
(2,2)
(1,1)
(2,2)
(1,1)
```



```
(1,1)
(2,2)
(1,1)
(2,2)
(1,1)
(1,1)
(2,2)

sep@swe10:~/fh/internal/beispiele>
```

Hier wird das Objekt der Klasse `TwoNumbers` immer nur ausgedruckt, wenn es inmitten der Ausführung der Methode `swap` ist. In diesem Zwischenzustand sind gerade im Zuge des Vertauschungsprozesses beide Zahlen gleich. Will man verhindern, daß andere Steuerfäden einen Zwischenstand einer Operation sehen, also das Objekt in einem Zustand, den es eigentlich nicht haben soll, so kann man in Java das Mittel der Synchronisation anwenden.

9.2.2 Synchronisation

Um bestimmte Programmteile für Steuerfäden zu reservieren, können diese Programmteile synchronisiert werden.

Synchronisation von Methoden

Java kennt für Methoden das zusätzliche Attribut `synchronized`. Wenn eine Methode das Attribut `synchronized` hat, bedeutet das, daß für ein bestimmtes Objekt von der Klasse, in der die Methode ist, immer nur genau einmal zur Zeit eine synchronisierte Methode ausgeführt werden darf. Soll gerade eine zweite synchronisierte Methode ausgeführt werden, so muß diese warten, bis der andere Aufruf der synchronisierten Methode beendet ist. Wollen wir den obigen Effekt in der Klasse `TwoNumbers` verhindern, nämlich daß die Methode `toString` einen temporären Zwischenzustand der Methode `swap` sieht, so können wir die beiden Methoden synchronisieren. Wir erhalten die folgende Klasse:

SafeTwoNumbers.java
```
class SafeTwoNumbers{
  int x;
  int y;

  SafeTwoNumbers(int x, int y) {this.x=x; this.y=y;}

  synchronized void swap(){
    int z=x;
    x=y;
    try {Thread.sleep(400);}catch (InterruptedException _){}
    y=z;
  }

  synchronized public String toString(){
    return "("+x+","+y+")";
  }
}
```

Wenn wir jetzt das Programm `PrintTwoNumbers` so abändern, daß es die synchronisierten Objekte des Typs `SafeTwoNumbers` benutzt, so drucken wir tatsächlich immer Objekte, in denen beide Zahlen unterschiedlich sind:

```
sep@swe10:~/fh/internal/beispiele> java PrintTwoNumbers
(1,2)
(2,1)
(1,2)
(2,1)
(1,2)
(2,1)
(1,2)
(2,1)

sep@swe10:~/fh/internal/beispiele>
```

Synchronisation von Blöcken

Java bietet zusätzlich an, daß nicht ganze Methoden, sondern nur bestimmte Programmsegmente als zu synchronisierender Abschnitt zu markieren sind. Solche Blöcke von Anweisungen werden hierzu in geschweifte Klammern gesetzt. Zusätzlich ist noch anzugeben, über welches Objekt synchronisiert wird. Insgesamt hat eine `synchronized`-Anweisung die folgende syntaktische Form:

`synchronized (`*obj*`){`*stats*`}`

Statt also die Synchronisation an den Methoden vorzunehmen, können wir auch erst bei ihrem Aufruf verlangen, daß dieser synchronisiert ist.

Hierzu synchronisieren wir über das Objekt `twoN` den Aufruf der Methode `swap`:

SafeSwap.java

```java
class SafeSwap extends Thread{

  public SafeSwap(TwoNumbers twoN){this.twoN=twoN;}

  private TwoNumbers twoN;

  public void run(){
    while (true){
      synchronized(twoN){twoN.swap();}
    }
  }
}
```

Und wir synchronisieren entsprechend auch den Aufruf, der dafür sorgt, daß das Objekt ausgedruckt wird:

SafePrintTwoNumbers.java

```java
class SafePrintTwoNumbers extends Thread {
  public SafePrintTwoNumbers(TwoNumbers twoN){this.twoN=twoN;}

```



```java
  private TwoNumbers twoN;

  public void run(){
    while (true){
      synchronized (twoN){System.out.println(twoN);}
      try {Thread.sleep(1000);}catch (InterruptedException _){}
    }
  }

  public static void main(String [] _){
    TwoNumbers twoN = new TwoNumbers(1,2);
    new SafeSwap(twoN).start();
    new SafePrintTwoNumbers(twoN).start();
  }
}
```

Auch in dieser Version drucken wir immer nur konsistente Zustände des Objekts `twoN`. Es wird kein temporärer Zwischenzustand der Methode `swap` ausgegeben.

9.2.3 Verklemmungen

Mit der Synchronisation von Operationen über verschiedene Objekte handelt man sich leider ein Problem ein, das unter dem Namen Verklemmung (eng. deadlock)[1] bekannt ist. Wenn es zu einer Verklemmung in einem Programm kommt, friert das Programm ein. Es rechnet nichts mehr, sondern wartet intern darauf, daß andere Programmteile mit der Berechnung eines kritischen Abschnitts fertig werden.

Damit es zu einer Verklemmung kommt, braucht es mindestens zwei Steuerfäden und zwei Objekte, über die diese Steuerfäden synchronisieren.

Verklemmung bei der Buchausleihe

Es gibt eine schöne Analogie für Verklemmungen aus dem Alltag. Hierzu denke man als Steuerfäden zwei Studenten (*Hans* und *Lisa*) und als Objekte zwei Bücher in der Bibliothek (*Programmieren* und *Softwaretechnik*). Hans leiht sich das Buch Programmieren und Lisa das Buch Softwaretechnik aus. Beide haben sich damit exklusiv je ein Objekt gesichert (es gibt nicht mehrere Exemplare des gleichen Buches in der Bibliothek.). Nun steht im Buch Programmieren eine Referenz auf das Buch Softwaretechnik und umgekehrt. Um das Buch Programmieren zu Ende durchzustudieren, braucht Hans jetzt auch das Buch Softwaretechnik, und für Lisa gilt das entsprechende umgekehrt. Beide wollen sich jetzt das andere Buch in der Bibliothek ausleihen, stellen aber fest, daß es verliehen ist. Jetzt warten sie darauf, daß es zurückgegeben wird. Beide geben ihr Buch nicht zurück, weil sie ja mit beiden Büchern arbeiten wollen. Somit warten beide aufeinander. Dieses Phänomen nennt man Verklemmung.

[1] Im Skript von Herrn Grude[Gru03] findet sich die etwas drastischere deutsche Bezeichnung *tödliche Umklammerung*.

Das 5-Philosophenproblem

Eine zweite Illustration von Verklemmungen ist das sogenannte 5-Philosophen-Problem. Fünf Philosophen sitzen um einen runden Tisch. Zwischen ihnen liegen Gabeln. Ein Philosoph kann entweder denken oder essen. Zum Essen braucht er zwei Gabeln, denken kann er nur, wenn er keine Gabel in der Hand hat. Nun greifen sich die Philosophen entweder Gabeln und essen oder lassen die Gabeln liegen und denken. Problematisch wird es, wenn sich jeder Philosoph erst die Gabel zu seiner Rechten greift, um dann festzustellen, daß auf der linken Seite keine Gabel liegt. Wenn jetzt alle darauf warten, daß links wieder eine Gabel hingelegt wird, und die Gabel in ihrer rechten Hand nicht zurückgeben, so kommt es zu einer Verklemmung. Keiner ißt und keiner denkt.

Verklemmung im Straßenverkehr

Auch im Straßenverkehr kann man mitunter Verklemmungen beobachten. Zwei in entgegengesetzte Richtung fahrende Autos fahren auf eine Kreuzung zu und wollen dort beide links abbiegen. Sie fahren soweit in die Kreuzung ein, daß sie dem entgegenkommenden Fahrzeug jeweils die Fläche auf der Fahrbahn, die es zum Linksabbiegen braucht, blockieren. Beide warten jetzt darauf, daß der andere weiterfährt. Der Verkehr steht.

Verklemmung in Java

Mit Steuerfäden und Synchronisation haben wir in Java die Mechanismen zur Verfügung, die zu einer klassischen Verklemmung führen können. Wir schreiben eine Steuerfadenklasse, in der es zwei Objekte gibt. In der Methode `run` wird erst über das eine Objekt synchronisiert und dann über das andere. Wir erzeugen in der Hauptmethode zwei Objekte dieser Steuerfadenklasse jeweils mit vertauschten Objekten:

Deadlock.java

```java
class Deadlock extends Thread{

  Object a;
  Object b;

  Deadlock (Object a,Object b){this.a=a;this.b=b;}

  public void run(){
    synchronized (a){
      System.out.println(
        getName()+"jetzt habe ich das erste Objekt: "+a);
      try {Thread.sleep(1000);}catch (InterruptedException _){}
      synchronized (b){
          System.out.println(getName()+"jetzt habe ich beide");
      }
    }
  }

  public static void main(String [] _){
    String s1 = "hallo";
```



```java
    String s2 = "welt";
    new Deadlock(s1,s2).start();
    new Deadlock(s2,s1).start();
  }
}
```

Dieses Programm hat zur Folge, daß sich der erste Stuerfaden zunächst das Objekt `s1` reserviert und der zweite Steuerfaden das Objekt `s2` reserviert. Wenn sie jetzt jeweils das andere Objekt anfordern, müssen sie darauf warten, bis der andere Steuerfaden dieses wieder frei gibt. Das Programm ist verklemmt, nichts geht mehr:

```
sep@swe10:~/fh/internal/beispiele> java Deadlock
Thread-0jetzt habe ich das erste Objekt: hallo
Thread-1jetzt habe ich das erste Objekt: welt
```

9.2.4 Warten und Benachrichtigen

Über die Synchronisation haben wir sichergestellt, daß bestimmte Operationen nur ausgeführt werden, wenn man bestimmte Objekte exklusiv hat. Falls ein anderer Steuerfaden dieses Objekt gesperrt hat, so muß der Steuerfaden warten, bis das Objekt für ihn zur Verfügung steht.

Darüberhinaus kann es erwünscht sein, daß ein Steuerfaden warten soll, bis ein anderer Steuerfaden bestimmte Operationen durchgeführt hat. Hierzu gibt es in Java die Methoden `wait` und `notifyAll`.

Stellen wir uns vor, wir haben ein Objekt, in dem eine Zahl gespeichert wird. Es gibt eine Methode, diese Zahl zu setzen, und eine Methode, um diese auszulesen. Zwei Steuerfäden sollen über dieses Objekt kommunizieren. Der eine schreibt immer Zahlen hinein, der andere liest diese wieder aus. Nun soll der lesende Steuerfaden immer erst so lange warten, bis der schreibende eine neue Zahl gespeichert hat, und der speichernde Steuerfaden soll immer so lange warten, bis der vorherige Wert ausgelesen wurde, damit er nicht überschrieben wird.

Ohne Vorkehrungen dafür, daß abwechselnd gelesen und geschrieben wird, sieht die Klasse wie folgt aus:

GetSet1.java

```java
class GetSet1 {
  private int i;
  GetSet1(int i){this.i=i;}

  synchronized int get(){return i;}
  synchronized void set(int i){this.i=i;}
}
```

Unsere beiden Steuerfäden, die auf einem solchen Objekt lesen und schreiben, können wir schreiben als:

Set.java

```java
class Set extends Thread{
  GetSet1 o;
  Set(GetSet1 o){this.o=o;}
  public void run(){
    for (int i=1;i<=1000;i=i+1){
      o.set(i);
    }
  }
}
```

Der lesende Steuerfaden mit entsprechender Hauptmethode:

Get.java

```java
class Get extends Thread{
  GetSet1 o;
  Get(GetSet1 o){this.o=o;}
  public void run(){
    for (int i=1;i<=1000;i=i+1){
      System.out.print(o.get()+" ");
    }
  }

  public static void main(String [] _){
    GetSet1 gss = new GetSet1(0);
    new Get(gss).start();
    new Set(gss).start();
  }
}
```

Starten wir dieses Programm, so werden nicht die Zahlen 0 bis 1000 auf dem Bildschirm ausgeben. Es ist nicht garantiert, daß erst nach einem Schreiben ein Lesen stattfindet. Wir können dieses durch einen internen Merker signalisieren, der angibt, ob eine neue Zahl verfügbar ist. Bevor wir eine neue Zahl lesen, können wir uns immer wieder schlafenlegen und dann schauen, ob jetzt die neue Zahl verfügbar ist:

GetSet2.java

```java
class GetSet2 {
  private int i;
  private boolean available= true;
  GetSet2(int i){this.i=i;}

  synchronized int get(){
    while (!available){
      try{Thread.sleep(1000);}catch (Exception _){}
    }
    available=false;
    return i;
  }

```



```java
  synchronized void set(int i){
    while (available){
      try{Thread.sleep(1000);}catch (Exception _){}
    }
    this.i=i;
    available=true;
  }
}
```

Starten wir damit unser Testprogramm, so kommt es zu einer Verklemmung. Um dieses zu umgehen, hat Java das Prinzip von `wait` und `notifyAll`. Statt `sleep` rufen wir die Methode `wait` auf und erlauben damit anderen Steuerfäden, obwohl wir in einer synchronisierten Methode sind, auf demselben Objekt aktiv zu werden. Die `wait`-Anweisung wird aufgehoben, wenn irgendwo eine `notifyAll`-Anweisung aufgerufen wird. Dann konkurrieren wieder alle Steuerfäden um die Objekte, über die sie synchronisieren.

Unser Programm sieht schließlich wie folgt aus:

GetSet3.java

```java
class GetSet3 {
  private int i;
  private boolean available= true;
  GetSet3(int i){this.i=i;}

  synchronized int get(){
    while (!available){
      try{wait();}catch (Exception _){}
    }
    available=false;
    notifyAll();

    return i;
  }

  synchronized void set(int i){
    while (available){
      try{wait();}catch (Exception _){}
    }
    this.i=i;
    available=true;
    notifyAll();
  }
}
```

Damit ist gewährleistet, daß tatsächlich nur abwechselnd gelesen und geschrieben wird. Wie man sieht, kann es relativ kompliziert werden, über synchronisierte Methoden und der `wait`-Anweisung zu programmieren, und Programme mit mehreren Steuerfäden, die untereinander kommunizieren, sind schwer auf Korrektheit zu prüfen.

Kapitel 10

Javawerkzeuge

10.1 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.

10.1.1 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}[vfmOMi] [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:
  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:

- 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.

[1] `t` steht dabei für *table*.

Aufgabe 28 Nehmen Sie eines Ihrer Javaprojekte des letzten Semesters (z.B. Eliza oder VierGewinnt) und verpacken die `.class`-Dateien des Projektes in eine Jar-Datei. Berücksichtigen Sie dabei die Paketstruktur, die sich in der Ordnerhierarchie der `.class` Dateien wiederspiegelt.

10.1.2 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.

Aufgabe 29 Starten Sie die Anwendung, die Sie in der letzten Aufgabe als Jar-Datei verpackt haben, mit Hilfe der `java`-Option: `-cp`.

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:**
> 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:**
> Der entsprechende Befehl des letzten Beispiels ist in Windows:
> `set CLASSPATH=~\jarfiles\jugs.jar:%CLASSPATH%`

Aufgabe 30 Laden Sie die Jar-Datei:
jugs.jar (`http://www.tfh-berlin.de/~panitz//prog2/load/jugs.jar`) .
In diesem Archiv liegt eine Javaanwendung, die eine interaktive Javaumgebung bereitstellt. Javaausdrücke und Befehle können eingegeben und direkt ausgeführt werden. Das Archiv enthält zwei Klassen mit einer Hauptmethode:

- `Jugs`: ein Kommandozeilen-basierter Javainterpreter.
- `JugsGui`: eine graphische interaktive Javaumgebung.

Um diese Anwendung laufen zu lassen, wird ein zweites Javaarchive benötigt: die JAR-Datei `tools.jar`. Diese befindet sich in der von Sun gelieferten Entwicklungsumgebung.

Setzen Sie den Klassenpfad (einmal per Javaoption, einmal durch neues Setzen der Umgebungsvariablen `CLASSPATH`) auf die beiden benötigten JAR-Dateien und starten Sie eine der zwei Hauptklassen. Lassen Sie folgende Ausdrücke in *Jugs* auswerten.

- `2*21`
- `"hello world".toUpperCase().substring(2,5)`
- `System.getProperties()`
- `System.getProperty("user.name")`

10.2 Java Dokumentation mit javadoc

10.3 Klassendateien analysieren mit javap

Anhang A

Beispielaufgaben für die Klausur

Aufgabe 1 Führen Sie die folgende Klasse von Hand aus und schreiben Sie auf, was auf dem Bildschirm ausgegeben wird:

```
class Aufgabe1{
  public static void main(String [] args){
    int i = 42;
    for (int j = i; j>=i%15;j=j-5){
      System.out.println(j);
    }
  }
}
```

Aufgabe 2 Die folgenden Javaprogramme enthalten Fehler. Beschreiben Sie die Fehler und korrigieren Sie sie:

a)
```
class Aufgabe2a {
   static int dividiere(int z, int n){
     if (n!=0) return z/n;
   }
}
```

b)
```
class Aufgabe2b {
   static int dividiere(int z, int n){
     if (n!=0) return z/n;
     throw new Exception("division durch 0");
   }
}
```

c)
```
class Aufgabe2c {
  String x;
```



```
  String y;

  Aufgabe2c(String x, StringBuffer y){
    this.x = x;
    this.y = y;
  }
}
```

d)
```
class Aufgabe2d{
  public void printTwiceAsUpperCase(String s){
    Object doppelS = s+s;
    System.out.println(doppelS.toUpperCase());
  }
}
```

e)
```
class Aufgabe2e_1{
  private String s = "hallo";
  public String getS(){return s;}
}
```

```
class Aufgabe2e_2{
  public static String getS(Aufgabe2e_1 e1){return e1.s;}
}
```

f)
```
class Aufgabe2f{
  public void printString(String s){
    System.out.println(s);
  }

  public static void main(String [] args){
    printString("hallo");
  }
}
```

Aufgabe 3 Gegeben sei die folgende abstrakte Klasse für Listen, gemäß unserer Spezifikation aus der Vorlesung:

```
abstract class AbList{
    abstract public AbList empty();
    abstract public AbList cons(Object x, AbList xs);
    abstract public boolean isEmpty();
    abstract public Object head();
    abstract public AbList tail();
}
```

Schreiben Sie für die Klasse folgende Methoden, die ihr hinzugefügt werden können:

a) `public AbList take(int i);`
`take` soll eine Teilliste der ersten i Elemente zurückgeben.

b) `public AbList drop(int i);`
`drop` soll eine Teilliste zurückgeben, in der die ersten i Elemente fehlen.

c) `public AbList sublist(int beginIndex,int laenge);`
`sublist` soll die Teilliste zurückgeben, vom Element an der Stelle `beginIndex` anfängt und die nachfolgenden `laenge` Elemente enthält. Sie dürfen die Methoden `take` und `drop` benutzen.

d) `public AbList twiceElements();`
`twiceElements` baut eine Liste, in der jedes Element doppelt eingetragen wurde.
Beispiel: aus der Liste `("a","b","c")` erzeugt `twiceElements` die Liste: `("a","a","b","b","c","c")`.

Aufgabe 4 Schreiben Sie eine Methode
`public static int wievielC(char c, String str)`
die zählt, wie oft der Buchstabe `c` im String `str` ist.

Aufgabe 5 Betrachten Sie sich folgende Klassen:

```
class Aufgabe5_1{
    public String getInfo(){return "5_1";}
}
```

```
class Aufgabe5_2 extends Aufgabe5_1{
    public String getInfo(){return "5_2";}
}
```

```
class Aufgabe5_3 extends Aufgabe5_2{
    public String getInfo(){return "5_3";}
}
```

```
class Aufgabe5 {
  static String getInfo(Aufgabe5_1 o){return o.getInfo();}

  public static void main(String [] args){
    Aufgabe5_1 a1 = new Aufgabe5_3();
    Aufgabe5_1 a2 = new Aufgabe5_2();
    Aufgabe5_1 a3 = new Aufgabe5_1();
    System.out.println(a3.getInfo());
    System.out.println(getInfo(a3));
    System.out.println(getInfo(a1));
    System.out.println(a2.getInfo());
  }
}
```

Führen Sie die Methode `main` von Hand aus. Was wird auf dem Bildschirm ausgegeben?

Aufgabe 6 Betrachten Sie die folgende Klasse:

```
class Aufgabe6{
  String g1(){
    new NullPointerException();
    return "g1";
  }

  String g2()throws Exception{
    try {
      return g1();
    }catch (NullPointerException _){
      throw new Exception();
    }
  }

  String g3(){
    String result="";
    try {
      result=g2();
    }catch (NullPointerException _){
      return "null pointer";
    }catch (Exception _){
      return "exception";
    }
    return result;
  }
  public static void main(String [] args){
    System.out.println(new Aufgabe6().g3());
  }
}
```

Führen Sie das Programm von Hand aus. Was wird auf dem Bildschirm ausgegeben? Erklären Sie wie es zu dieser Ausgabe kommt.

Anhang B

Gesammelte Aufgaben

Aufgabe 1 Schreiben Sie das obige Programm mit einen Texteditor ihrer Wahl. Speichern Sie es als `FirstProgram.java` ab. Übersetzen Sie es mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `FirstProgram.class`. Führen Sie das Programm mit dem Javainterpreter `java` aus. Führen Sie dieses sowohl einmal auf Linux als auch einmal unter Windows durch.

Aufgabe 2 Schreiben sie ein Programm, das ein Objekt der Klasse `Minimal` erzeugt und auf dem Bildschirm ausgibt. Hierzu ersetzen sie einfach im Programm `Answer` die `42` durch den Ausdruck `new Minimal()`

Aufgabe 3 Schreiben Sie für die vier Karteikarten in der Modellierung eines Bibliotheksystems entsprechende Klassen mit den entsprechenden Feldern.

Aufgabe 4 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.

Hinweis: der Typ, der ganze Zahlen in Java bezeichnet, heißt `int`.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben Sie für jede dieser Klassen eine Methode `public String toString()` mit dem Ergebnistyp. 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.

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

Aufgabe 5 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 6 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 7 Starten Sie folgendes Javaprogramm:

TestInteger.java

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

Erklären Sie die Ausgabe.

Aufgabe 8 Ergänzen Sie ihre Klasse `Ausleihe` um eine Methode `void verlaengereEinenMonat()`, die den Rückgabetermin des Buches um einen Monat erhöht.

Aufgabe 9 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 10 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 11 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 12 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 13 Schreiben Sie jetzt die Methode zur Berechnung der Fakultät, indem Sie eine Iteration und nicht eine Rekursion benutzen.

Aufgabe 14 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 15 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 16 Für die Lösung dieser Aufgabe gibt es 3 Punkte, die auf die Klausur angerechnet werden. Voraussetzung hierzu ist, daß die Lösung mir in der Übung gezeigt und erklärt werden kann.

In dieser Aufgabe sollen Sie eine Klasse für römische Zahlen entwickeln.

a) Schreiben Sie eine Klasse `Roman`. Diese Klasse soll eine natürliche Zahl darstellen.

b) Schreiben Sie für Ihre Klasse `Roman` einen Konstruktor, der ein Stringobjekt als Parameter hat. Dieser Stringparameter soll eine römische Zahl darstellen. Der Konstruktor soll diese Zahl lesen und in einem Feld des Typs `int` abspeichern.

c) Implementieren Sie die Methode `public String toString()` für Ihre Klasse `Roman`, die die intern gespeicherte Zahl als römische Zahl dargestellt zurückibt.

d) Fügen Sie ihrer Klasse `Roman` die folgenden Methoden für arithmetische Rechnungen hinzu.

- `Roman add(Roman other)`
- `Roman sub(Roman other)`
- `Roman mul(Roman other)`
- `Roman div(Roman other)`

e) Testen Sie Ihre Klasse `Roman`.

Aufgabe 17 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

```java
class ButtonLogic {
  String getDescription(){
    return "in Großbuchstaben umwandeln";
  }
  String eval(String x){return x.toUpperCase();}
}
```

Dialogue.java

```java
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
class Dialogue extends JFrame{

  final ButtonLogic logic;

  final JButton button;
  final JTextField inputField  = new JTextField(20) ;
  final JTextField outputField = new JTextField(20) ;
  final JPanel p = new JPanel();

  Dialogue(ButtonLogic l){
    logic = l;
    button=new JButton(logic.getDescription());
    button.addActionListener
     (new ActionListener(){
        public void actionPerformed(ActionEvent _){
          outputField.setText
             (logic.eval(inputField.getText().trim()));
        }
      });
    p.setLayout(new BorderLayout());
    p.add(inputField,BorderLayout.NORTH);
    p.add(button,BorderLayout.CENTER);
    p.add(outputField,BorderLayout.SOUTH);
    getContentPane().add(p);
    pack();
    setVisible(true);
  }
}
```

TestDialogue.java

```java
class TestDialogue {
  public static void main(String [] _){
    new Dialogue(new ButtonLogic());
  }
}
```

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 eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie römische Zahlen in arabische Zahlen umwandeln können. Testen Sie Ihr Programm.

d) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie arabische Zahlen in römische Zahlen umwandeln können. Testen Sie Ihr Programm.

e) Schreiben Sie jetzt ein Guiprogramm, daß eine Zahl aus ihrer Darstellung zur Basis 10 in eine Darstellung zur Basis 2 umwandelt. Testen Sie.

Aufgabe 18 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) `Object last()`: gibt das letzte Element der Liste aus.

b) `List concat(List other)` bzw.: `Li concat(Li 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) `Object elementAt(int i)`: gibt das Element an einer bestimmten Indexstelle der Liste zurück. Spezifikation:

$$\begin{aligned} \mathit{elementAt}(\mathit{Cons}(x, xs), 1) &= x \\ \mathit{elementAt}(\mathit{Cons}(x, xs), n+1) &= \mathit{elementAt}(xs, n) \end{aligned}$$

Aufgabe 19 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 20 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

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.basic.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

public class HtmlView extends JPanel {

  //example invocation
  public static void main(String s[]) {
    HtmlView view = new HtmlView();
    view.run();
    view.setText("<h1>hallo</h1>");
  }

  JFrame frame;
  JTextPane ausgabe = new JTextPane();

  public HtmlView() {
    ausgabe.setEditorKit(new HTMLEditorKit());
    add(ausgabe);
  }

  void setText(String htmlString){
    ausgabe.setText(htmlString);
    frame.pack();
```

```
    ausgabe.repaint();
  }

  void run(){
    frame = new JFrame("HtmlView");
    frame.getContentPane().add(this);
    frame.pack();
    frame.setVisible(true);
  }
}
```

Aufgabe 21 In dieser Aufgabe sollen Sie ein Programm schreiben, das nach und nach die Primzahlen ausgibt.

a) Schreiben sie eine Unterklasse `From` der Klasse `FromTo`, die von einem Startwert an in Einerschritten ganze Zahlen zurückgibt und für die `hasNext` immer wahr ist.

b) Schreiben Sie eine Klasse `Sieb`, die die Schnittstelle `IntIterator` implementiert. Die Klasse soll zwei Felder haben: eine ganze Zahl und ein weiteres Objekt, das die Schnittstelle `IntIterator` implementiert. Der Konstruktor habe die Signatur:
`public Sieb(int i,IntIterator it)`
Die Methode `next` soll das nächste Element des inneren Iterators `it` zurückgeben, das nicht durch die Zahl `i` teilbar ist.

Beispiel: `PrintIntIteratorElements.print(new Sieb(3,new From(1)))` gibt alle nicht durch 3 teilbaren natürlichen Zahlen auf dem Bildschirm aus.

c) Übersetzen Sie ihren Code mit folgender Klasse:

```
package de.tfhberlin.panitz.iterator;
class PrimIterator implements IntIterator{
  private IntIterator sieb = new From(2);
  public boolean hasNext(){return sieb.hasNext();}
  public int next(){
    int result = sieb.next();
    sieb= new Sieb(result,sieb);
    return result;
  }
}
```

Lassen Sie sich alle Werte von `PrimIterator` mit der Methode `PrintIntIteratorElements.print` ausgeben.

Aufgabe 22 (4 Punkte)
In dieser Aufgabe wird ein kleines Programm, das einen Psychoanalytiker simuliert, vervollständigt.

a) Laden Sie sich hierzu das Archiv
`Eliza.zip` (`http://www.tfh-berlin.de/~panitz/prog1/load/Eliza.zip`)
vom Netz. Entpacken Sie es. Machen Sie sich mit den einzelnen Klassen vertraut.

b) In der Klasse `MyList` sind nicht alle abstrakten Methoden der Klasse `Li` implementiert. Ergänzen Sie `MyList` um die Methoden:
`reverse`, `words`, `unwords`, `drop`, `tails`, `isPrefixIgnoreCaseOf`. Implementieren Sie diese Methoden entsprechend ihrer Dokumentation in der abstrakten Klasse `Li` und schreiben Sie Tests für jede Methode.

Wenn Ihre Tests erfolgreich sind, übersetzen Sie alle Klassen und starten Sie die `main`-Methode der Klasse `Main`.

c) Erfinden Sie eigene Einträge für die Liste `respMsgs` in der Klasse `Data`.

d) Erklären Sie, was die Methode `rotate` von den anderen Methoden der Klasse `Li` fundamental unterscheidet. Demonstrieren Sie dieses anhand eines Tests.

Aufgabe 23 Übersetzen und starten Sie die Klasse `Finalize` und beobachten Sie, wie und wann Java Objekte aus dem Speicher löscht.

Aufgabe 24 (3 Punkte)
In dieser Aufgabe soll ein Spielbrett für das Spiel Vier gewinnt implementiert werden. Laden Sie hierzu die Datei
vier.zip (`http://www.tfh-berlin.de/~panitz/prog1/load/vier.zip`)

a) Schreiben Sie eine Klasse `VierImplementierung`, die die Schnittstelle `VierLogik` entsprechend der Dokumentation implementiert.

b) Schreiben Sie folgende Hauptmethode und starten Sie diese. Sie sollten jetzt in der Lage sein, über die Eingabekonsole Vier gewinnt zu spielen.

```
public static void main(String[] args) {
  new VierKonsole().spiel(new VierImplementierung());
}
```

Suchen Sie sich einen Spielpartner und versuchen Sie, gegen ihn zu gewinnen.

Aufgabe 25 Schreiben Sie ein Programm `FileConvert` zum Konvertieren von Textdateien in eine andere Kodierung. Dem Programm sollen über die Kommandozeilenparameter der Name der Eingabedatei, der Name der Ausgabedatei und der Name der benutzten Codierung übergeben werden. Ein möglicher Aufruf wäre also:

```
linux:~/>java FileConvert test.txt konvertiertTest.txt utf-8
```

Lassen Sie eine deutsche Textdatei mit Umlauten in eine Datei mit der Codierung utf-8 konvertieren. Betrachten Sie die Ergebnisdatei. Was stellen Sie fest?

Aufgabe 26 Erweitern Sie die Klasse `Li`, so daß Sie Ihre Listenobjekte in Dateien schreiben und wieder aus Dateien lesen können. Testen Sie ihre Implementierung.

Aufgabe 27 Studieren Sie die Dokumentation von `java.io.RandomAccessFile` und schreiben Sie einige Testbeispiele zur Benutzung dieser Klasse.

Aufgabe 28 Nehmen Sie eines Ihrer Javaprojekte des letzten Semesters (z.B. Eliza oder VierGewinnt) und verpacken die `.class`-Dateien des Projektes in eine Jar-Datei. Berücksichtigen Sie dabei die Paketstruktur, die sich in der Ordnerhierarchie der `.class` Dateien wiederspiegelt.

Aufgabe 29 Starten Sie die Anwendung, die Sie in der letzten Aufgabe als Jar-Datei verpackt haben, mit Hilfe der `java`-Option: `-cp`.

Aufgabe 30 Laden Sie die Jar-Datei:
jugs.jar (`http://www.tfh-berlin.de/~panitz//prog2/load/jugs.jar`) .
In diesem Archiv liegt eine Javaanwendung, die eine interaktive Javaumgebung bereitstellt. Javaausdrücke und Befehle können eingegeben und direkt ausgeführt werden. Das Archiv enthält zwei Klassen mit einer Hauptmethode:

- `Jugs`: ein Kommandozeilen-basierter Javainterpreter.
- `JugsGui`: eine graphische interaktive Javaumgebung.

Um diese Anwendung laufen zu lassen, wird ein zweites Javaarchive benötigt: die JAR-Datei `tools.jar`. Diese befindet sich in der von Sun gelieferten Entwicklungsumgebung.

Setzen Sie den Klassenpfad (einmal per Javaoption, einmal durch neues Setzen der Umgebungsvariablen `CLASSPATH`) auf die beiden benötigten JAR-Dateien und starten Sie eine der zwei Hauptklassen. Lassen Sie folgende Ausdrücke in *Jugs* auswerten.

- `2*21`
- `"hello world".toUpperCase().substring(2,5)`
- `System.getProperties()`
- `System.getProperty("user.name")`

Anhang C

Java Syntax

Im folgenden ist eine kontextfreie Grammatik der Javasyntax angegeben. Terminalsymbole sind mit `Schreibmaschinentyp` gesetzt, Nichtterminale in *kursiver Proportionsschrift*. Alternativen sind durch einen Längsstrich | getrennt, optionale Teile in eckigen Klammern [] und 0- bis n-fache Wiederholung in geschweiften Klammern {} angegeben.

Identifier ::=

`IDENTIFIER`

QualifiedIdentifier ::=

Identifier { `.` *Identifier* }

Literal ::=

```
 IntegerLiteral
|FloatingPointLiteral
|CharacterLiteral
|StringLiteral
|BooleanLiteral
|NullLiteral
```

Expression ::=

Expression1 [AssignmentOperator Expression1]

AssignmentOperator ::=

```
 =
|+=
|-=
|*=
|/=
|&=
||=
|^=
|%=
|<<=
|>>=
|>>>=
```

Type ::=

Identifier { . *Identifier* } *BracketsOpt*
|*BasicType*

StatementExpression ::=

Expression

ConstantExpression ::=

Expression

Expression1 ::=

Expression2 [Expression1Rest]

Expression1Rest ::=

[? *Expression* : *Expression1]*

Expression2 ::=

Expression3 [Expression2Rest]

Expression2Rest ::=

{*Infixop Expression3*} *Expression3* `instanceof` *Type*

Infixop ::=

```
 ||
|&&
||
|^
|&
|==
|!=
|<
|>
|<=
|>=
|<<
|>>
|>>>
|+
|-
|*
|/
|%
```

Expression3 ::=

PrefixOp Expression3
|(*Expr* | *Type*) *Expression3*
|*Primary* {*Selector*}{*PostfixOp*}

Primary ::=

(*Expression*)
|`this` *[Arguments]*
|`super` *SuperSuffix*
|*Literal*
|`new` *Creator*
|*Identifier* { . *Identifier* }*[IdentifierSuffix]*
|*BasicType BracketsOpt* `.class`
|`void.class`

IdentifierSuffix ::=

[*(*] *BracketsOpt* `.class`| *Expression*] *)*
|*Arguments*
|. *(* `class`|`this`|`super` *Arguments* |`new` *InnerCreator)*

PrefixOp ::=

```
 ++
|--
|!
|~
|+
|-
```

PostfixOp ::=

`++`
|`--`

Selector ::=

`.` *Identifier [Arguments]*
|`.this`
|`.super` *SuperSuffix*
|`.new` *InnerCreator*
|`[` *Expression* `]`

SuperSuffix ::=

Arguments
|`.` *Identifier [Arguments]*

BasicType ::=

`byte`
|`short`
|`char`
|`int`
|`long`
|`float`
|`double`
|`boolean`

ArgumentsOpt ::=

[Arguments]

Arguments ::=

`(` *[Expression { , Expression }]* `)`

BracketsOpt ::=

{`[]`}

Creator ::=

QualifiedIdentifier (ArrayCreatorRest | ClassCreatorRest)

InnerCreator ::=

Identifier ClassCreatorRest

ArrayCreatorRest ::=

`[` *(* `]` *BracketsOpt ArrayInitializer | Expression* `]`{`[` *Expression* `]`} *BracketsOpt)*

ClassCreatorRest ::=

Arguments [ClassBody]

ArrayInitializer ::=

`{` *[VariableInitializer* {`,` *VariableInitializer*} *[*`,`*]]* `}`

VariableInitializer ::=

ArrayInitializer
| *Expression*

ParExpression ::=

`(` *Expression* `)`

Block ::=

`{` *BlockStatements* `}`

BlockStatements ::=

{ *BlockStatement* }

BlockStatement ::=

LocalVariableDeclarationStatement
|*ClassOrInterfaceDeclaration*
|*[Identifier :] Statement*

LocalVariableDeclarationStatement ::=

[`final`*] Type VariableDeclarators ;*

Statement ::=

Block
|`if` *ParExpression Statement [*`else` *Statement]*
|`for(` *ForInitOpt* `;` *[Expression]* `;` *ForUpdateOpt* `)` *Statement*
|`while` *ParExpression Statement*
|`do` *Statement* `while` *ParExpression* `;`
|`try` *Block (Catches | [Catches]* `finally` *Block)*
|`switch` *ParExpression* `{` *SwitchBlockStatementGroups* `}`
|`synchronized` *ParExpression Block*
|`return` *[Expression] ;*
|`throw` *Expression ;*
|`break` *[Identifier]*
|`continue` *[Identifier]*
|`;`
|*ExpressionStatement*
|*Identifier* `:` *Statement*

Catches ::=

CatchClause {*CatchClause*}

CatchClause ::=

`catch(` *FormalParameter* `)` *Block*

SwitchBlockStatementGroups ::=

{ *SwitchBlockStatementGroup* }

SwitchBlockStatementGroup ::=

SwitchLabel BlockStatements

SwitchLabel ::=

`case` *ConstantExpression* :
|`default`

MoreStatementExpressions ::=

{ , *StatementExpression* }

ForInit ::=

StatementExpression MoreStatementExpressions
|[`final`] *Type VariableDeclarators*

ForUpdate ::=

StatementExpression MoreStatementExpressions

ModifiersOpt ::=

{ *Modifier* }

Modifier ::=

`public`
|`protected`
|`private`
|`static`
|`abstract`
|`final`
|`native`
|`synchronized`
|`transient`
|`volatile`
|`strictfp`

VariableDeclarators ::=

VariableDeclarator { , *VariableDeclarator* }

VariableDeclaratorsRest ::=

VariableDeclaratorRest { , *VariableDeclarator* }

ConstantDeclaratorsRest ::=

ConstantDeclaratorRest { , *ConstantDeclarator* }

VariableDeclarator ::=

Identifier VariableDeclaratorRest

ConstantDeclarator ::=

Identifier ConstantDeclaratorRest

VariableDeclaratorRest ::=

BracketsOpt [= *VariableInitializer*]

ConstantDeclaratorRest ::=

BracketsOpt = *VariableInitializer*

VariableDeclaratorId ::=

Identifier BracketsOpt

CompilationUnit ::=

[`package` *QualifiedIdentifier* ;] {*ImportDeclaration*}{*TypeDeclaration*}

ImportDeclaration ::=

`import` *Identifier* { . *Identifier* } [. *] ;

TypeDeclaration ::=

ClassOrInterfaceDeclaration
|;

ClassOrInterfaceDeclaration ::=

ModifiersOpt (*ClassDeclaration* | *InterfaceDeclaration*)

ClassDeclaration ::=

`class` *Identifier* [`extends` *Type*] [`implements` *TypeList*] *ClassBody*

InterfaceDeclaration ::=

`interface` *Identifier* [`extends` *TypeList*] *InterfaceBody*

TypeList ::=

Type { , *Type*}

ClassBody ::=

{{ *ClassBodyDeclaration* }}

InterfaceBody ::=

{{ *InterfaceBodyDeclaration*}}

ClassBodyDeclaration ::=

;
|[`static`] *Block*
|*ModifiersOpt MemberDecl*

MemberDecl ::=

MethodOrFieldDecl
|`void` *Identifier MethodDeclaratorRest*
|*Identifier ConstructorDeclaratorRest*
|*ClassOrInterfaceDeclaration*

MethodOrFieldDecl ::=

Type Identifier MethodOrFieldRest

MethodOrFieldRest ::=

VariableDeclaratorRest
|*MethodDeclaratorRest*

InterfaceBodyDeclaration ::=

;
|*ModifiersOpt InterfaceMemberDecl*

InterfaceMemberDecl ::=

InterfaceMethodOrFieldDecl
|`void` *Identifier VoidInterfaceMethodDeclaratorRest*
|*ClassOrInterfaceDeclaration*

InterfaceMethodOrFieldDecl ::=

Type Identifier InterfaceMethodOrFieldRest

InterfaceMethodOrFieldRest ::=

ConstantDeclaratorsRest ;
|*InterfaceMethodDeclaratorRest*

MethodDeclaratorRest ::=

FormalParameters BracketsOpt [`throws` *QualifiedIdentifierList*] (*MethodBody*|;
)

VoidMethodDeclaratorRest ::=

FormalParameters [`throws` *QualifiedIdentifierList*] (*MethodBody* | ;)

InterfaceMethodDeclaratorRest ::=

FormalParameters BracketsOpt [`throws` *QualifiedIdentifierList*] ;

VoidInterfaceMethodDeclaratorRest ::=

FormalParameters [`throws` *QualifiedIdentifierList*] ;

ConstructorDeclaratorRest ::=

FormalParameters [`throws` *QualifiedIdentifierList*] *MethodBody*

QualifiedIdentifierList ::=

QualifiedIdentifier { , *QualifiedIdentifier*}

FormalParameters ::=

([*FormalParameter* { , *FormalParameter*}])

FormalParameter ::=

[`final`] *Type VariableDeclaratorId*

MethodBody ::=

Block

Anhang D

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

Verzeichnis der Klassen

Abbildungsverzeichnis

Literaturverzeichnis

[Gru03] Ulrich Grude. Einführung in die Programmierung mit Java. `www.tfh-berlin.de/~grude/SkriptJava1SS03.pdf`, 2003. Skript, TFH Berlin.

[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.