Programmieren I

Version 20. September 2003

WS 02/03

Prof. Dr. Sven Eric Panitz
TFH Berlin

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

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird. 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

1 Einführung **1-1**

1.1 Ziel der Vorlesung 1-1

1.2 Programmieren 1-1

1.2.1 Disziplinen der Programmierung 1-1

1.2.2 Was ist ein Programm 1-4

1.2.3 Klassifizierung von Programmiersprachen 1-4

1.2.4 Arbeitshypothese 1-6

2 Objektorientierte Programmierung mit Java **2-1**

2.1 Grundkonzepte der Objektorientierung 2-1

2.1.1 Außer der Reihe: die Hauptmethode 2-1

2.1.2 Objekte und Klassen 2-1

2.1.3 Felder und Methoden 2-3

3 Imperative Konzepte **3-1**

3.1 Primitive Typen, die klassenlose Gesellschaft 3-1

3.1.1 Der Typ: int 3-2

3.1.2 Der Typ: boolean 3-2

3.1.3 weitere primitive Typen 3-3

3.2 Operatoren 3-3

3.3 Zusammengesetzte Befehle 3-5

3.3.1 Bedingungsabfrage mit: if 3-5

3.3.2 Iteration 3-8

3.3.3 Rekursion und Iteration 3-15

3.4 Ausdrücke und Befehle 3-17

3.4.1 Bedingungen als Befehl oder als Ausdruck 3-18

3.4.2 Auswertungsreihenfolge und Seiteneffekte von Ausdrücken 3-18

4 Vererbung **4-1**

4.1 Hinzufügen neuer Eigenschaften 4-2
4.2 Überschreiben bestehender Eigenschaften 4-3
4.3 Konstruktion 4-3
4.4 Zuweisungskompatibilität 4-4
4.5 Späte Bindung (late binding) 4-6
4.5.1 Methoden als Daten 4-7
4.6 Zugriff auf Methoden der Oberklasse 4-8
4.7 Die Klasse Object 4-9
4.7.1 Die Methode equals 4-10
4.8 Klassentest 4-10
4.9 Typzusicherung (Cast) 4-11

5 Datentypen und Algorithmen **5-1**

5.1 Listen 5-1
5.1.1 Formale Spezifikation 5-1
5.1.2 Modellierung 5-3
5.1.3 Codierung 5-4
5.1.4 Methoden für Listen 5-7
5.1.5 Spezialisierte Listen 5-9
5.1.6 Sortierung 5-11

6 Weiterführende programmiersprachliche Konzepte **6-1**

6.1 Pakete 6-1
6.1.1 Paketdeklaration 6-1
6.1.2 Übersetzen von Paketen 6-2
6.1.3 Starten von Klassen in Paketen 6-2
6.1.4 Das Java Standardpaket 6-3
6.1.5 Benutzung von Klassen in anderen Paketen 6-3
6.1.6 Importieren von Paketen und Klassen 6-3
6.2 Sichtbarkeitsattribute 6-5
6.2.1 Sichtbarkeitsattribute für Klassen 6-5
6.2.2 Sichtbarkeitsattribute für Eigenschaften 6-6
6.3 Schnittstellen (Interfaces) und abstrakte Klassen 6-9
6.3.1 Schnittstellen 6-9
6.3.2 Abstrakte Klassen 6-14

6.4 Ausnahme- und Fehlerbehandlung 6-17
6.4.1 Ausnahme- und Fehlerklassen 6-18
6.4.2 Werfen von Ausnahmen 6-18
6.4.3 Deklaration von geworfenen Ausnahmen 6-21
6.4.4 Eigene Ausnahmeklassen 6-22
6.4.5 Fangen von Ausnahmen 6-23
6.4.6 Der Aufrufkeller 6-29
6.4.7 Schließlich und finally 6-30
6.4.8 Anwendungslogik per Ausnahmen 6-31

7 Java Standardklassen **7-1**
7.1 Die Klasse Objekt 7-1
7.1.1 Die Methoden der Klasse Object 7-2
7.2 Behälterklassen für primitive Typen 7-5
7.3 String und Stringbuffer 7-5
7.4 Sammlungsklassen 7-7
7.4.1 Listen 7-7
7.4.2 Mengen 7-8
7.4.3 Iteratoren 7-8
7.4.4 Abbildungen 7-10
7.4.5 Weitere Sammlungsmethoden 7-13
7.4.6 Vergleiche 7-13
7.5 Reihungen 7-14
7.5.1 Deklaration von Reihungen 7-14
7.5.2 Erzeugen von Reihungen 7-15
7.5.3 Zugriff auf Elemente 7-16
7.5.4 Ändern von Elementen 7-16
7.5.5 Das Kovarianzproblem 7-17
7.5.6 Weitere Methoden für Reihungen 7-18
7.5.7 Reihungen von Reihungen 7-18
7.5.8 Blubbersortierung 7-19
7.6 Ein- und Ausgabe 7-20
7.6.1 Zeichenströme 7-21
7.6.2 Byteströme 7-21
7.6.3 Schreiben und Lesen 7-21
7.6.4 Dateiströme 7-21
7.6.5 Gepufferte Ströme 7-23
7.6.6 Stromloses IO 7-24

8 Steuerfäden **8-1**

8.1 Schreiben von Steuerfäden 8-1

8.1.1 Schlafenlegen von Prozessen 8-2

8.2 Koordination nebenläufiger Steuerfäden 8-3

8.2.1 Benutzung gemeinsamer Objekte 8-3

8.2.2 Synchronisation 8-5

8.2.3 Verklemmungen 8-7

8.2.4 Warten und Benachrichtigen 8-9

9 Graphische Benutzeroberflächen **9-1**

9.1 Gui Komponenten 9-2

9.1.1 Top-Level Komponenten 9-2

9.1.2 Zwischenkomponenten 9-3

9.1.3 Atomare Komponenten 9-4

9.2 Gruppierung 9-5

9.2.1 Flow Layout 9-5

9.2.2 Border Layout 9-6

9.2.3 Grid Layout 9-7

9.3 Ereignissbehandlung 9-8

9.3.1 Action Listener 9-8

9.3.2 Mausereignisse 9-10

9.3.3 Fensterereignisse 9-11

9.4 Innere und anonyme Klassen 9-12

9.4.1 Innere Klassen 9-12

9.4.2 Anonyme Klassen 9-13

10 Schlußkapitel **10-1**

10.1 Überbleibsel 10-1

10.1.1 die switch Anweisung 10-1

10.1.2 Das final Attribut 10-3

10.1.3 String Konstanten 10-3

10.1.4 Operatoren 10-4

10.2 Ausblick 10-6

10.3 Persönliches Schlußwort 7

A Beispielaufgaben **A-1**

B Gesammelte Aufgaben **B-1**

C Java Syntax **C-1**

D Wörterliste **D-1**

Verzeichnis der Klassen D-4

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 allgeinen 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 der Wartung bestehender Software. Hierzu

gehört die Anpassung des Programms an neue Versionen benutzter Bibliotheken, oder neuer Versionen 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 den 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 unterschiedliche 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, daß 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 Benutzng 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.

mathematische Antwort

Eine Funktion, die deterministisch für Eingabewerte eine Ausgabewert berechnet.

sprachwissenschaftliche Antwort

Ein Satz einer durch einer 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 mitlerweile mehr Programmiersprachen als natürlich 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:

```
1 javac Test.java
```

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

```
1 java Test
```

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 Außer der Reihe: die Hauptmethode

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.

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

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

Aufgabe 1 Schreiben Sie das obige Programm mit einem Texteditor oder einer Javaprogrammierumgebung ihrer Wahl und lassen Sie es laufen. Machen Sie diese Übung sowohl auf einem Windowsbetriebssystem als auch auf einem Unix-artigen Betriebssystem.

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.

2.1.2 Objekte und Klassen

Die Grundidee der objektorientierten Programmierung ist, Daten mit den Programmteilen, durch die sie manipuliert werden können, organisatorisch in Objekten zu bündeln. Ein Ob-

jekt 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.

Klassen

Eine Klasse beschreibt eine Menge von Objekten gleicher Art. Die Klassendefinition ist eine Beschreibung der möglichen Objekte. In ihr ist definiert, was für Daten zu den Objekten gehören und welche Operationen auf diesen Daten angewendet werden können. Klassendefinitionen sind die eigentlichen Programmtexte, die der Programmierer schreibt. In Java steht genau eine Klassendefinition[1] in genau einer Datei. Die Datei hat den Namen der Klasse.

In Java wird eine Klasse durch das Schlüsselwort `class` gefolgt von den 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:

Minimal.java

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

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

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

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. Eine Objekt der obigen minimalen Javaklasse läßt sich entsprechend durch folgenden Ausdruck erzeugen:

```
1 new Minimal();
```

Aufgabe 2 Schreiben sie ein Programm, das ein Objekt der Klasse `Minimal` erzeugt und auf dem Bildschirm ausgibt.

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

[1] Auch hier werden wir Ausnahmen kennenlernen.

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

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

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

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

2.1.3 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 ausgeführt werden konnten. 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.

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

ErsteFelder.java

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

Wir haben hiermit eine erste Klasse, die zwei Felder enthält, definiert. Das Feld soll dabei einmal ein Objekt unserer minimalen Klasse sein und das andere Mal eine Zeichenkette.

Feldnamen werden per Konvention immer klein geschrieben.

Zuweisung In Feldern können Objekte gespeichert werden. Hierzu muß dem Feld ein Objekt zugewisen 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.

```
1 meinFeld = new Minimal();
2 meinTextFeld = "hallo";
```

Nach einer Zuweisung repräsentiert das Feld das ihm zugewiesene Objekt.

Methoden

Methoden[2] 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.

Als Beispiel definieren wir eine Methode `addString`, die den Ergebnistyp `String` und zwei Parameter vom Typ `String` hat.

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

Methoden und Parameternamen werden per Konvention immer klein geschrieben.

Zugriff auf Felder im Methodenrumpf In einer Methode stehen die Felder der Klasse zur Verfügung[3].

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:

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

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

[2] 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*.

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

Person.java

```
class Person {
  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

Wie 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 Namen 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.

Person.java

```
class Person {
  String vorname;
  String nachname;

  Person(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:

```
new Person("Nicolo","Paganini");
new Person("William","Shakespeare");
```

lokale Felder

Wir kennen bisher die Felder einer Klassen. 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.

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

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

Zugriff auf Eigenschaften eines Objektes

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

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

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

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

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 einzelen Objekten gehören, können sie nicht auf dynamische Felder und Methoden der Objekte zugreifen.

StaticTest.java

```
1 class StaticTest {
2   static void printThisText(String text){
3     System.out.println(text);
4   }
5 }
```

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

Auf statische Eigenschaften wird zugegriffen, indem dem Klassennamen per Punkt getrennt das Eigenschaften aufgerufen wird:

```
1 class Main {
2   public static void main(String [] args){
3     StaticText.printThisText("hello");
4   }
5 }
```

Ebenso wie statische Methoden, gibt es auch statische Felder. Im Unterschied zu dynamischen Felder, 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.

der this-Bezeichner

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

UseThis.java

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

Die Benutzung von `this` in einer statischen Methode sollte zu einem Übersetzungsfehler führen, weil es für statische Eigenschaften kein konkretes Objekt gibt, auf dem sie ausgeführt werden.

Mit Ende diesen Kapitels sind wir in der Lage, erste Klassen mit Methoden und Feldern zu definieren und auf Feldern und Methoden zuzugreifen.

Aufgabe 3 Schreiben Sie Klassen, die je eines der folgenden Objektarten repräsentieren können:

- Personen mit Namen und Adresse.
- Bücher, die einen Titel und einen Autor haben.
- Buchausleihe, in der vermerkt ist, daß eine Person ein bestimmtes Buch ausgeliehen hat.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben sie für jede dieser Klassen eine Methode `toString` mit dem Ergibnistyp `String`. Das Ergebnis soll eine gute textuelle Beschreibung des Objektes sein.

c) Fügen Sie der Klasse `Person` ein statisches Feld zu, in dem stets der Name der zuletzt erzeugten Person gespeichert ist.

d) 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 Bildschrim ausgeben. Zeigen Sie in diesen Hauptmethode, daß das statische Feld der Klasse `Person` wie spezifiziert funktioniert.

e) Testen Sie, ob sich statische Eigenschaften auch über die Objekte einer Klasse und nicht nur über den Klassennamen zugreifen lassen.

Kapitel 3

Imperative Konzepte

Im letzten Abschnitt wurde ein erster Einstieg in die objektorientierte Programmierung gegeben. Wie zu sehen war, ermöglicht es 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 sind 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, die mit einem Punkt zugegriffen werden können.

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 dieser Designvorgabe wäre.

3.1.1 Der Typ: int

Java hält eine Anzahl numerische primitiver Datentypen bereit. Ein einfacher Datentyp zur Darstellung ganzer Zahlen ist der Typ: `int`. Daten dieses Typs lassen sich durch eine einfache Ziffernfolge schreiben. Die Literale diesen Typs sind entsprechend also einfache Ziffernfolgen, denen ein Negationszeichen vorangestellt sein kann, z.B. `0, 42, 1967, -1000`.

In der folgenden Klassen sehen wir Beispiele, wie `int` als Typ eines Feldes als Rückgabe- und Parametertyp einer Methode benutzt wird, sowie ein Beispiel für ein Literal.

Testint.java

```
1  class Testint{
2    int eineGanzeZahl;
3
4    Testint(int i){
5      eineGanzeZahl = i;
6    }
7
8    int getEineGanzeZahl(){
9      return this.eineGanzeZahl;
10   }
11
12   public static void main(String [] args){
13     Testint t = new Testint(42);
14     System.out.println(t.getEineGanzeZahl());
15   }
16 }
```

3.1.2 Der Typ: boolean

Ein in der Informatik häufig gebrauchte Unterscheidung ist, ob etwas wahr oder falsch ist, also eine Unterscheidung zwischen genau zwei Werten.[2] 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

```
1  class Testbool{
2    boolean boolFeld;
3
4    Testbool(boolean b){
5      boolFeld = b;
6    }
7
```

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

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

3.1.3 weitere primitive Typen

Java kennt neben `int` und `boolean` noch weitere primitive Typen: `byte` , `char`, `short`, `long`, `double`, `float`. Wir wollen uns mit ihren Details vorerst nicht belasten.

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

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

```
5   }
6 }
```

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

Square.java

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

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

```
1  class Vergleich{
2
3    public static void main(String[] args){
4      System.out.println(1+1 < 42);
5      System.out.println(1+1 <= 42);
6      System.out.println(1+1 >  42);
7      System.out.println(1+1 >= 42);
8      System.out.println(1+1 == 42);
9      System.out.println(1+1 != 42);
10   }
11 }
```

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

```
1 class TestboolOperator{
2   public static void main(String [] args){
3     System.out.println(true && false);
4     System.out.println(true || false);
5     System.out.println(!true || false);
6     System.out.println(true || true && false);
7   }
8 }
```

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

```
1 class TestboolOperator2{
2   static boolean implication(boolean a, boolean b){
3     return !a || b;
4   }
5
6   public static void main(String [] args){
7     System.out.println(implication(true, false));
8   }
9 }
```

3.3 Zusammengesetzte Befehle

Streng genommen kennen wir bisher nur einen Befehl, den Zuweisungsbefehl, der einem Feld einen neuen Wert zuweist.[3] 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

```
1 class FirstIf {
2
3   static void firstIf(boolean bedingung){
4     if (bedingung) {
```

[3] Konstrukte mit Operatoren nennt man dagegen Ausdrücke.

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

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

FirstIf2.java

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

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

ElseIf.java

```
1  class ElseIf {
2
3    static String lessOrEq(int i,int j){
4      if (i<10)  return "i kleiner zehn";
5      else if (i>10)  return "i größer zehn";
6      else if (j>10)  return "j größer zehn";
7      else if (j<10)  return "j kleiner zehn";
8      else return "j=i=10";
9    }
10
11   public static void main(String [] args){
12      System.out.println(lessOrEq(10,9));
13   }
14
15 }
```

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

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

Summe.java

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

Wir können diese 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 4 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 eininge der für die Klasse `String` definierten Methoden.

Aufgabe 5 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 6 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 7 Modellieren und schreiben Sie eine Klasse, die ein Bankkonto darstellt. Auf das Bankkonto sollen Einzahlungen und Auszahlungen vorgenommen werden können. Es gibt eine 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 der Rekursion kommt ohne zusätzliche zusammengesetzte Befehle von Java aus. Da Rekursionen von der virtuellen Maschine Javas nur bis zu einen 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 es Rumpfes die Bedingung geprüft wird.

Vorgeprüfte Schleifen Die vorgeprüfte Schleifen haben folgendes Schema in Java:

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

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

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

WhileTest.java

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

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

Summe2.java

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

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

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

Bottom.java

```
1 class Bottom {
2    static public void bottom(){
3      while (true){};
4    }
5 }
```

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 die nachgeprüfte Variante:

DoTest.java

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

Man kann sich leicht davon vergewissern, daß die nachgeprüfte Schleife mindestens einmal durchlaufen[4] wird:

VorUndNach.java

```
1 class VorUndNach {
2
3   public static void main(String [] args){
4
5       while (falsch())
6          {System.out.println("vorgeprüfte Schleife");};
```

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

```
7
8         do {System.out.println("nachgeprüfte Schleife");}
9         while (false);
10    }
11
12    public static boolean falsch(){return false;}
13  }
```

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, der Ausgabe der Zahlen von 0 bis 9 in der `for`-Schleifenversion wie folgt aus.

ForTest.java

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

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

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

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

Summe3.java

```
1  class Summe3 {
2    public static int summe(int n){
3
4      int erg = 0 ;                    // Feld für Ergebnis
5
6      for (int j = n;j>0;j=j-1){       // j läuft von n bis 1
7        erg = erg + j;                 // akkumuliere das Ergebnis
8      }
9
10     return erg;
11   }
12 }
```

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

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

Summe4.java

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

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

Vorzeitiges Beenden von Schleifen

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

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

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

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

BreakTest.java

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

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

ForBreak.java

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

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

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

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

ContTest.java

```
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
            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 kann.

For Schleifen in Bolero Die Programmiersprache Bolero der Software AG, die ebenso wie Java Code für die Java virtuelle Maschine erzeugt, lehnt sich eher an die aus Datenbankanfragesprachen bekannten `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 Selektion 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. Sie 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 den mathematischen Ausdruck: $< lpar/ > x^2 | x \in I\!N < rpar/ >$

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..]
```

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 zu erkennen, wann diese 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

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

```

```
7   public static int count1(int k){
8     int j = 2;
9     for (int i= 0;i<1000000000;i=i+1){
10      j=j+1;
11    }
12    return j;
13  }
14
15  public static int count2(int i){
16      if (i<1000000000) return count2(i+1) +1; else return 2;
17  }
18
19 }
20
```

Aufgabe 8 In dieser Aufgabe können Sie zum ersten mal eine Funktionalität schreiben, die durch eine graphisches Benutzeroberfläche bedient wird.

a) Kopieren Sie sich die Klassen

- `Dialogue` (`http://www.tfh-berlin.de/~panitz/prog1/../Dialogue.java`) und
- `String2String` (`http://www.tfh-berlin.de/~panitz/prog1/../String2String.java`)

Die Klasse `Dialogue` definiert eine kleine graphische Benutzerschnittstelle. Zwei Textflächen und ein Schaltknopf. Beim Drücken des Schaltknopfs wird der Text aus der oberen Textfläche ausgelesen, mit dem Text die Methode `string2string` des Objektes der Klasse `String2String` aufgerufen und das Ergebnis in die untere Textfläche geschrieben.

Übersetzen Sie die Klassen und starten Sie die `main`-Methode.

b) Ändern sie die Methode `string2string` so, daß das Ergebnis alle Buchstaben in entsprechende Großbuchstaben umwandelt.

Aufgabe 9 Schreiben Sie eine Methode, `int readInt(String str)`. Diese Methode soll einen String, der nur aus Ziffern besteht in die von ihm repräsentierte Zahl umwandeln. Benutzen sie hierzu Methoden der `String`-Klasse, die es erlauben, einzelne Buchstaben einer Zeichenkette zu selektieren. Behandeln sie den primitiven Typen `char` dabei, als sei es der Typ `int`.

Aufgabe 10 **Hinweis:** Die Aufgabe wird fortgesetzt werden.

a) Schreiben Sie eine Klasse `From`. Objekte dieser Klasse sollen die Methode `int next()` haben. Ausgehend von einem initialen Anfangswert, soll diese Methode stets eine um eins höhere Zahl bei jedem Aufruf ausgeben.

Beispiel: Für ein Objekt `frm`, das mit `new From(42)` konstruiert wurde, sollen die Aufrufe

```
1 System.out.println(frm.next());
2 System.out.println(frm.next());
3 System.out.println(frm.next());
4 System.out.println(frm.next());
```

die Zahlen `42, 43, 44, 45` ausgegeben.

b) Schreiben Sie eine Klasse `Sieb`. Objekte dieser Klasse sollen ein Objekt der Klasse `From` sowie eine Zahl `n` enthalten. Auch in dieser Klasse schreiben sie eine Methode `int next()`. Das Ergebnis soll die nächste Zahl die das Objekt vom Typ `From` ausgibt sein, die nicht durch `n` teilbar ist.

Hinweis: der Operator `%` berechnet den ganzzahligen Rest einer Division.

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[5] (englisch: *expressions* und *statements*).

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

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

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

Beispiele für Befehle sie 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.

[5] 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.

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 mit einem Doppelpunkt getrennt:

pred`?` *alt1*`:` *alt2*

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

Folgender Code:

```
1 System.out.println(true?1:2);
2 System.out.println(false?3:4);
```

druckt erst die Zahl 1 und dann die Zahl 4.

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 denen sie von Java abgearbeitet werden inruitiv 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 könne Seiteneffekte, die eine Ausgabe auf den Bildschirm drucken, dazu nutzen, zu testen, in welcher Reihenfolge Unterausdrücke ausgewertet werden. Hierzu schreiben wir eine Subtraktionsmethode mit einem derartigen Seiteneffekt.

Minus.java

```
1 class Minus {
2
3   static public int sub(int x,int y){
4     int erg = x-y;
```

```
5       System.out.println("eine Subtraktion mit Ergebnis: "
6                          +erg+" wurde durchgeführt.");
7       return erg;
8     }
9   }
```

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

Operatorauswertung.java

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

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

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

Auswertung der Methodenargumente

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

Reihenfolge.java

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

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

```
    /**
       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.

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

Terminierung

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

TerminationTest.java

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

```
    /**
       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
```

```
7       **/
8     public static int wenn(boolean pre,int a1,int a2 ){
9       return pre?a1:a2; }
```

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

```
1   public static void main(String [] args){
2
3     //test, whether both alternatives of conditional
4     //expression are evaluated or just one.
5
6     //this will terminate:
7     System.out.println(      false?bottom():eins("b")   );
8
9     //this won't terminate:
10    System.out.println(wenn( false,bottom(),eins("b") )  );
11  }
12}
```

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 auswerteten. Diese Auswertungsstrategie wird dann als *nicht strikt* bezeichnet, während man die Auswertungsstrategie von Java als *strikt* bezeichnet.

Kapitel 4

Vererbung

Eine der grundlegendsten Ziele der objektorientierung Programmierung ist die Möglichkeit, bestehende Programme um weitere 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

```
1  class Person {
2
3    String name ;
4    String address;
5
6    Person(String name, String address){
7      this.name = name;
8      this.address = address;
9    }
10
11   public String toString(){
12     return name+", "+address;
13   }
14 }
```

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

StudentOhneVererbung.java

```
1  class StudentOhneVererbung {
2
3    String name ;
4    String address;
5    int matrikelNummer;
```

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

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

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

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

```
1 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 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 Vererbung 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 mit neuen Eigenschaften zu erweiteren. 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.

```
class Student extends Person{

  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.

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

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

```
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:

```
  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:

```
1  class Student extends Person{
2     int matrikelNummer;
3
4     Student(String name,String adresse,int nr){
5       super(name,adresse);
6       matrikelNummer = nr;
7     }
8  }
```

In unserem Beispiel bekommt der Konstruktor der Klasse `Student` alle Daten die benötigt werden 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:

```
1  Student s
2   = new Student("Martin Müller","Hauptstraße 2",755423);
```

Wir erhalten insgesamt folgende Klasse `Student`:

Student.java

```
1  class Student extends Person{
2     int matrikelNummer;
3
4     Student(String name,String adresse,int nr){
5       super(name,adresse);
6       matrikelNummer = nr;
7     }
8
9    public String toString(){
10     return    name + ", " + address
11             + " Matrikel-Nr.: " + matrikelNummer;
12   }
13
14 }
```

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, so dürfen diese auch Feldern des Typs `Person` zugewiesen werden:

TestStudent1.java

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

Alle Studenten sind auch Personen.

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

StuedentError1.java

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

Die Kompilierung dieser Klasse führt zu folgender Fehlermeldung:

```
StudentError1.java:3: incompatible types
found   : Person
required: Student
    Student s = new 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 ihn auch Objekte eines spezielleren Typs geben, in unserem Fall der Klasse `Student`.

TestStudent2.java

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

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

StuedentError2.java

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

Auch hier führt die Kompilierung zu einer entsprechenden Fehlermeldung:

```
StudentError2.java:9: printStudent(Student) in StudentError2
                      cannot be applied to (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

```
1  class TestLateBinding {
2
3    public static void main(String [] args){
4      Student s = new Student("Martin Müller","Hauptstraße",756456);
5      Person p1 = new Person("Harald Schmidt","Marktplatz");
6
7      System.out.println(s.toString());
8      System.out.println(p1.toString());
9
10     Person p2 = new Student("Martin Müller","Hauptstraße",756456);
11     System.out.println(p2.toString());
12   }
13 }
```

Dieses Programm erzeugt folgende Ausgabe:

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

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

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

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

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

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

Dieses in Java realisierte Prinzip wird als *late binding* bezeichnet.[2]

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 können wir uns zunutze machen, um Funktionalität zu übergeben. Hierzu betrachten wir die den Konstruktor und die Benutzung der Klasse `Dialogue`, die in einer der letzten Aufgaben vorgegeben war.

Dialogue.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class Dialogue extends JPanel {
6
7      JFrame frame;
8      JButton button            = new JButton("ausführen");;
9      JTextArea eingabe         = new JTextArea(20,40);
10     JTextArea ausgabe         = new JTextArea(20,40);
11     ButtonListener myListener = new ButtonListener();
12     String2String funktion;
13
14     public Dialogue(String2String funktionalität) {
15       setLayout(new BorderLayout());
```

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

```
16        button.addActionListener(myListener);
17        funktion = funktionalität;
18        add("North",eingabe);
19        add(button);
20        add("South",ausgabe);
21      }
22
23     ......
24
25      public static void main(String [] args) {
26        new Dialogue(new String2String()).run();
27      }
28  }
```

Diese Klasse hat einen Konstruktor, der als Argument ein Objekt des Typs `String2String` erwartet. In dieser Klasse gibt es eine Methode `String string2string(String inp)`, die offensichtlich benutzt wird, um die Funktionalität der graphischen Benutzerschnittstelle (GUI) zu bestimmen. In der Aufgabe sollte die Methode `string2string` geändert werden, damit das GUI eine neue Funktionalität bekommt. Jetzt können wir über die Vererbung eine Unterklasse der Klasse `String2String` mit einer entsprechend neuen Funktionalität schreiben und mit dieser den Konstruktor der Klasse `Dialogue` aufrufen:

DoubleTheString.java

```
1  class DoubleTheString extends String2String{
2      String string2string(String inp){
3        return inp+inp;
4      }
5
6      public static void main(String [] args) {
7        new Dialogue(new DoubleTheString()).run();
8      }
9  }
```

Die `main`-Methode der Klasse `DoubleTheString` öffnet nun das GUI der Klasse `Dialogue` mit einer neuen Funktionalität. Ohne die Klassen `Dialogue` und `String2String` zu verändern, haben wir sie in modifizierter Weise benutzt.

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.

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:

```
1    public String toString(){
2      return    name + ", " + address
```

```
3             + " Matrikel-Nr.: " + matrikelNummer;
4   }
```

Der Ausdruck

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

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

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

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, so 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 objektorientierten 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 Dokumentation zeigt, daß zu diesen Eigenschaften auch die Methode `toString` gehört, wie wir sie bereits schon 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 äußert allgemein. Sobald wir von einem Object nur noch wissen, daß es vom Typ `Object` ist, können wir kaum noch spezifische Dinge mit ihm anfangen.

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

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 Object 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:

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

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

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

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

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

4.8 Klassentest

Wie wir oben gesehen haben, können wir zu wenig 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

```
1 class InstanceOfTest {
2   public static void main(String [] str){
3      Person p1 = new Person("Strindberg","Skandinavien");
4      Person p2 = new Student("Ibsen","Skandinavien",789565);
5      if (p1 instanceof Student)
6        System.out.println("p1 ist ein  Student.");
7      if (p2 instanceof Student)
```

```
8         System.out.println("p2 ist einStudent.");
9       if (p1 instanceof Person)
10        System.out.println("p1 ist eine Person.");
11      if (p2 instanceof Person)
12        System.out.println("p2 ist eine Person.");
13    }
14  }
```

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

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

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, das es zu dieser Klasse gehört, müssem 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, dem wir ihm zusichern wollen:

CastTest.java

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

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

CastError.java

```
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, dem man ihm zusichern will.

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

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

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

Aufgabe 11 Mit dem Prinzip der Vererbung können wir die Klassen `From` und `Sieb` aus der letzten Aufgabe so ändern, daß wir mit ihnen ein Programm schreiben können, das alle Primzahlen erzeugen kann.

a) Ändern sie die Klasse `Sieb` der letzten Aufgabe so ab, daß Sie eine Unterklasse der Klasse `From` erhalten.

b) Sofern sie in der ersten Unteraufgabe eine korrekte Lösung haben, druckt folgendes Programm alle Primzahlen. Testen sie dieses.

PrintPrim.java

```
1 class PrintPrim{
2   /**
3     Prints prime numbers on screen.
4   **/
```

```
5     static public void main(String [] args){
6       //initial sieve: starts with all numbers greater 1
7       From sieb = new From(2);
8
9       while (true){
10        //get the first number of new sieve
11        int i = sieb.next();
12
13        //print it
14        System.out.println(i);
15
16        //create a new sieve, which deletes all multiples of i
17        sieb = new Sieb(sieb,i);
18      }//while
19    }
20  }
```

Aufgabe 12 Lösen Sie die Aufgabe 8 jetzt, indem Sie eine Unterklasse der Klasse `String2String` schreiben.

Aufgabe 13 In dieser Aufgabe sollen Sie die Klasse `Counter` aus einer der vorhergehenden Aufgaben in ein Gui einbinden. Laden Sie sich hierzu die Klasse
`CounterGUI.java` (`http://www.tfh-berlin.de/~panitz/prog1/../CounterGUI.java`)
vom Netz und kompilieren Sie diese mit Ihrer Klasse `Counter`. Passen Sie gegebenenfalls Ihre Klasse `Counter` an.

Aufgabe 14 Für die Lösung dieser Aufgabe gibt es 3 Punkte, die auf die Klausur angerechnet werden. Voraussetzung hierzu ist, daß die Lösung mir spätestens in der Übung am 21.11.2002 gezeigt und erklärt werden kann.

In dieser Aufgabe sollen Sie römische Zahlen in arabische Zahlen umwandeln.

a) Laden Sie sich hierzu die Klasse
`Roman.java` (`http://www.tfh-berlin.de/~panitz/prog1/../Roman.java`)
vom Netz und kompilieren Sie diese mit der Klasse `Dialogue`.
Die Methode `readRoman` ist noch nicht ausprogrammiert. Ergänzen Sie den Rumpf dieser Methode, so daß die Methode entsprechend der Dokumentation
`Roman.html` (`http://www.tfh-berlin.de/~panitz/prog1/../Roman.html`)
funktioniert.

b) Schreiben Sie eine Unterklasse der Klasse `Roman`, in der die Methode `toString` so überschrieben ist, so daß sie eine Repräsentation als römische und nicht als arabische Zahl ausgibt.

c) Fügen Sie zur Klasse `Roman` Methoden `add,sub,mul,div` hinzu, mit den Signaturen: `Roman add(Roman other)`,`Roman sub(Roman other)`,....

Diese Methoden sollen die vier Grundrechenarten auf römische Zahlen realisieren.

Kapitel 5

Datentypen und Algorithmen

Die bisher kennengelernten Javakonstrukten bilden einen soliden Kern, der genügend programmiertechnische Mittel zur Verfügung stellt, um die gängisten Konzepte der Informatik umzusetzen. In diesem Kapitel werden wir keine neuen Javakonstrukte kennen lernen, sondern mit den bisher bekannten Mitteln, die häufigsten in der Informatik gebräuchlich Datentypen und auf ihnen anzuwendenen 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 Samlung 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 zuwenden, wollen wir in diesem Kapitel Listen selbst spezifizieren und programmieren.

5.1.1 Formale Spezifikation

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 weiterem Element betrachtet. Mit dieser Betrachtungsweise haben Listen eine rekursive Struktur: eine Liste besteht aus dem zuletzt vorne angehängten neuem Element, dem sogenannten Kopf der Liste, und aus der alten Teilliste, an die dieses Element angehängt wurde, dem sogenannten Schwanz der Liste. Wie jede rekursive Struktur bedarf es einen Anfang der Definition. Im Falle von Listen wird dieses durch die Konstruktion einer leeren Liste spezifziert.[1]

Konstruktoren

Abstrakte Datentypen wie Listen, lassen sich durch ihre Konstruktoren spezifizieren. Die Konstruktoren geben an, wie Daten des entsprechenden Typs konstruiert werden können.

[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.

In dem Fall von Listen bedarf 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 den Selektoren und Konstruktoren läßt sich durch folgende Gleichungen spezifizieren:

$head(Cons(x,xs)) = x$
$tail(Cons(x,xs)) = xs$

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:

$isEmpty(Empty) = true$
$isEmpty(Cons(x,xs)) = false$

Somit ist alles spezifiziert, was eine Listenstruktur ausmacht. Listen können konstruiert werden, die Bestandteile einer Liste wieder einzeln selektiert werden und Listen können nach der Art ihrer Konstruktion unterschieden werden.

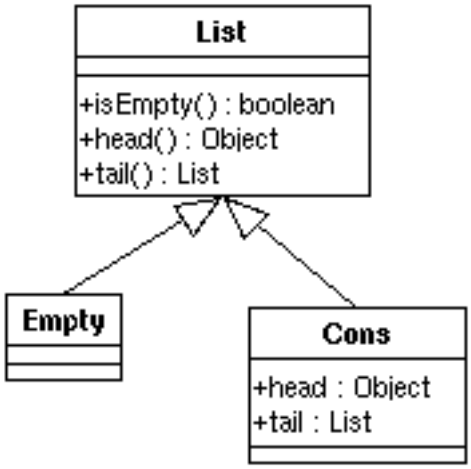
[2]Entgegen der Notation in Java, in der der Rückgabetyp kurioser Weise vor den Namen der Methode geschrieben wird.

5.1.2 Modellierung

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 Datentypen Liste bilden sollen, stehen wir eine gemeinsame Oberklasse dieser zwei Klassen vor. Wir erhalten folgende Klassenhierarchie:



Wir haben uns in diesem Klassendiagramm dazu entschieden, daß die Selektormethoden für alle Listen auch für leere Listen zur Verfügung stehen. Es ist in der formalen Spezifikation nicht angegeben worden, was in dem Fall der Methoden *head* und *tail* auf leere Listen als Ergebnis erwartet wird. Wir können hier Fehler geben, oder aber bestimmte ausgezeichnete Werte als Ergebnis zurückgeben.

In einer Klasse

Eine alternative Modellierung der Listenstruktur besteht aus nur genau einer Klasse. Diese Klasse braucht hierzu aber zwei Konstruktoren. Wie in vielen objektorientierten Sprachen ist es in Java auch möglich für eine Klasse mehrere Konstruktoren mit unterschiedlichen Parametertypen zu schreiben. Wir können also eine Klasse `List` modellieren, die zwei Konstruktoren hat. Einen mit keinem Parameter und einen mit zwei Parametern.

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

```
1 class List {
2  boolean isEmpty(){return false;}
3  Object head(){return null;}
4  List tail(){return null;}
5 }
```

Die Methoden `tail` und `head` können nur für nichtleere Listen Objekte zurückgeben. Daher haben wir uns für die prototypischen Implementierung in der Klasse `List` dazu entschieden, den Wert `null` zurückzugeben. `null` steht in Java für das Fehlen eines Objektes. Der Versuch auf Felder und Methoden von `null` zuzugreifen führt in Java zu einem Fehler.

Für die Klasse `List` schreiben wir keinen Konstruktor. Wir wollen diese Klasse nie direkt instanziieren, d.h. nie einen Konstruktor für die Klasse `List` aufrufen.[3]

Die Klasse Empty Die Klasse, die die leere Liste darstellt ist relativ leicht abzuleiten. Wir stellen einen Konstruktor zur Verfügung und überschreiben die Methode `isEmpty`.

Empty.java

```
1 class Empty extends List {
2   Empty(){}
3   boolean isEmpty(){return true;}
4 }
```

Die Klasse Cons Schließlich ist noch die Klasse `Cons` zu codieren. Diese Klasse benötigt nach unserer Modellierung zwei Felder, in denen Kopf und Schwanz der Liste abgespeichert werden können. Die Methoden `head` und `tail` werden so überschrieben, daß sie entsprechend den Wert eines dieser Felder zurückgeben. Der Konstruktor initialisiert diese beiden Felder.

Cons.java

```
1 class Cons extends List{
2
3   Object hd;
4   List tl ;
5
6   Cons(Object x,List xs){
```

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

```
7       hd = x;
8       tl = xs;
9     }
10
11    Object head(){
12      return hd;
13    }
14
15    List tail(){
16      return tl;
17    }
18  }
```

Damit ist die Listenstruktur gemäß unserer formalen Spezifikation vollständig implementiert.

Einfache Tests Für Algorithmen auf Listen werden wir nur die zwei Konstruktoren, die zwei Selektoren und die Testmethode `isEmpty` benutzen. Folgendes kleine Testprogramm konstruiert eine Liste mit drei Elementen des Typs `String`. Anschließend folgen einige Tests für die Selektormethoden.

TestFirstList.java

```
1   class TestFirstList {
2     public static void main(String [] args){
3
4       //Konstruktion einer Testliste
5       List xs = new Cons("friends",
6                 new Cons("romans",
7                 new Cons("countrymen",
8                 new Empty())));
9
10      //Zugriffe auf einzelne Elemente
11      System.out.println("1. Element: "+xs.head());
12      System.out.println("2. Element: "+xs.tail().head());
13      System.out.println("3. Element: "+xs.tail().tail().head());
14
15      //Test für die Methode isEmpty()
16      if (xs.tail().tail().tail().isEmpty()){
17        System.out.println("leere Liste nach drittem Element.");
18      }
19
20      //Ausgabe eines null Wertes
21      System.out.println(xs.tail().tail().tail().head());
22    }
23
24  }
```

Implementierung als eine Klasse

In der zweiten Modellierung haben wir auf eine Klassenhierarchie verzichtet. Was in der ersten Modellierung 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 das bool'sche Feld, um für das neu konstruierte Objekt zu markieren, mit welchem Konstruktor es konstruiert wurde. Die Konstruktoren setzen entsprechend die Felder, so daß die Selektoren diese nur auszugeben brauchen.

Li.java

```
1  class Li {
2    boolean empty = true;
3    Object  hd = null;
4    Li      tl = null;
5
6    Li (){}
7
8    Li (Object x,Li xs){
9     hd = x;
10    tl = xs;
11    empty = false;
12   }
13
14   boolean isEmpty() {return empty;}
15   Object  head(){return hd;}
16   Li      tail(){return tl;}
17 }
```

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 unterscvhiedlichen Namen, sondern beide haben denselben Namen.

TestFirstLi.java

```
1  class TestFirstLi {
2    public static void main(String [] args){
3
4      //Konstruktion einer Testliste
5      Li xs = new Li("friends",
6              new Li("romans",
7              new Li("countrymen",
8              new Li())));
9
10     //Zugriffe auf einzelne Elemente
11     System.out.println("1. Element: "+xs.head());
12     System.out.println("2. Element: "+xs.tail().head());
13     System.out.println("3. Element: "+xs.tail().tail().head());
14
15     //Test für die Methode isEmpty()
16     if (xs.tail().tail().tail().isEmpty()){
```

```
17        System.out.println("leere Liste nach drittem Element.");
18      }
19
20      //Ausgabe eines null Wertes
21      System.out.println(xs.tail().tail().tail().head());
22    }
23
24  }
```

Wie man aber sieht, ändert sich an der Benutzung von Listen nichts im Vergleich zu der Modellierung mittels einer Klassenhierarchie.

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 5 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 wieviel **Cons**-Listen eine Liste besteht. Formal können wir die Bedeutung der Methode *length* mit den folgenden zwei Gleichungen spezifizieren:

$length(Empty()) = 0$
$length(Cons(x, xs)) = 1 + length(xs)$

Diese beiden Gleichungen lassen sich direkt in Javacode umsetzen. Wir können die Klassen `List` und `Li` um folgende Methode `length` ergänzen:

```
1   int length(){
2     if (isEmpty()){ return 0;}
3     else{ return 1+tail().length();}
4   }
```

In den beiden Testklassen läßt sich ausprobieren, ob die Methode `length` entsprechend der Spezifikation funktioniert. Wir können in der `main`-Methode einen Befehl einfügen, der die Methode `length` aufruft:

```
1  System.out.println(xs.length());
```

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:

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

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

```
1  int length(){return 1+tail().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 befinden sich dann in den unterschiedlichen Klassen realisiert. Der Algorithmus ist in diesem Fall auf verschiedene Klassen aufgeteilt.

Iterartive 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 Geadanke ist naheliegend. Solange es sich noch nicht um die leere Liste handelt, wird ein Zähler, der die Elemente zählt, hochgezählt:

```
1  int length(){
2    int erg=0;
3    List xs = this;
4    while (!xs.isEmpty()){
5      erg= erg +1;
6      xs = xs.tail();
7    }
8    return erg;
9  }
```

Schaut man sich diese Lösung genauer an, so sieht man, daß sie die klassischen drei Bestandteile einer `for`-Schleife enthält:

- die Initialisierung einer Laufvariablen: `List xs = this;`
- eine bool'sche Bedingung zur Schleifensteuerung: `!xs.isEmpty()`
- eine Weiterschaltung der Schleifenvariablen: `xs = xs.tail();`

Damit ist die Schleife, die über die Elemente einer Liste iteriert ein guter Kandidat für eine `for`-Schleife. Wir können das Programm entsprechend umschreiben:

```
1 int length(){
2   int erg=0;
3   for (List xs=this;!xs.isEmpty();xs=xs.tail()){
4     erg = erg +1;
5   }
6   return erg;
7 }
```

Eine Schleifenvariable ist also nicht unbedingt eine Zahl, die hoch oder herunter gezählt wird, sondern kann auch ein Objekt sein, von dessen Eigenschaften abhängt, ob die Schleife ein weiteres mal zu durchlaufen ist.

In solchen Fällen ist die Variante mit der `for`-Schleife der Variante mit der `while`-Schleife vorzuziehen, weil somit die Befehle, die die Schleife steuern gebündelt zu Beginn der Schleife stehen.

toString

Eine sinnvolle Methode `toString` für Listen erzeugt einen String, in dem die Listenelemente durch Kommas getrennt sind und die in Klammern eingeschlossen ist.

```
1 public String toString(){
2   return "("+toStringAux();
3 }
4
5 private String toStringAux(){
6   if (isEmpty()) return ")";
7   else if (tail().isEmpty()) return head().toString()+")";
8   else return head().toString()+","+tail().toStringAux();
9 }
```

Aufgabe 15 Nehmen Sie eine der in diesem Kapitel vorgestellten Umsetzungen von Listen und fügen Sie ihrer Listenklasse folgende Methoden zu. Führen Sie Tests für diese Methoden durch.

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

b) `List concat(List 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 Stelle der Liste zurück.

5.1.5 Spezialisierte Listen

Unsere Umsetzung der Listen hat eine unschöne Schwäche. Die Elemente einer Liste sind lediglich vom Typ `Object`. Das hat zwar den Vorteil, daß wir beliebige Objekte in einer Liste

speichern können, aber, wenn wir z.B. mit der Methode `elementAt` auf ein Element der Liste zugreifen, wissen wir über den Typ dieses Elements nicht konkretes mehr. Dieses liegt an einer Schäche des Typssystems von Java. Um Listen so zu gestallten, daß Objekte beliebigen Typs gespeichert werden konnten, mußten wir für den ersten Parameter des Konstruktors **Cons** den Typ `Object` wählen. Daraus ergibt sich aber auch der Rückgabetyp `Object` für den Selektor `head`, der nach unserer Spezifkation die einzige Möglichkeit ist, wieder auf Listenelemente zuzugreifen. Java fehlt derzeit eine Möglichkeit, in einer Klassendeklaration von einen beliebigen aber festen Typen zu sprechen.[4] Benutzen wir die oben definierte Listenklasse `Li`, so muß nach einer Selektion eine Typzusicherung gemacht werden:

ListTypeCast.java

```
1  class ListTypeCast
2    public static void main(String [] args){
3      Li xs = new Li("friends",new Li ("romans",new Li()));
4      System.out.println(((String)xs.tail().head()).toUpperCase());
5    }
6  }
```

Eine Typzusicherung vorzunehmen, ist immer eine unschöne Lösung. Die Typzusicherung kann bei falscher Programmierung zu Laufzeitfehlern führen. Java ist nicht in der Lage bereits zur Übersetzungszeit solche Fehler abzufangen. Wenn wir eine derartige Typzusicherung nach der Selektion eines Elements aus einer Liste verhindern wollen, so können wir eine spezialisierte Liste schreiben, die nur Elemente eines bestimmten Typs speichern kann. Hierzu schreiben wir eine Unterklasse der Klasse `Li`, deren Konstruktor und Selektor nicht den Typ `Object` als Parameter bzw. Ergebnistyp hat, sondern für unser Beispiel den Typ `String`.

StringLi.java

```
1  class StringLi extends Li{
2    StringLi(String x,StringLi xs){
3      super(x,xs);
4    }
5    StringLi(){
6      super();
7    }
8    String stringHead(){return (String)head();}
9    StringLi stringTail(){return (StringLi)tail();}
10 }
```

Java erlaubt uns nicht die Methoden `head` und `tail` aus der Oberklasse mit anderen, spezialisierten Rückgabetypen zu überschreiben. Daher sind die neuen Methoden `stringHead` und `stringTail` definiert worden. Dieser neue Unterklasse stellt nur noch Listen mit `String`-Elementen dar. Werden nun Listenelemente selektiert, so ist ihr Typ `String`, die Typzusicherung entfällt.

TestLi.java

```
1  class TestLi {
2   public static void main(String [] args){
3    StringLi xs
```

[4] Dieses wird sich mit den *generischen* Typen in einer der kommenden Javaversionen ändern.

```
4       =  new StringLi("friends"
5         ,new StringLi("romans"
6         ,new StringLi()));
7     System.out.println(xs.stringTail().stringHead().toUpperCase());
8     System.out.println(xs.length());
9    }
10  }
```

Wie man an den Aufruf der Methode `length` sieht, lassen sich die geerbten Methoden auf Objekten des Typs `StringLi` weiterhin benutzen.

Die Definition einer für einen bestimmten Elementtyp spezialisierten Liste lohnt sich besonders, wenn in einem Projekt besonders viele Listen mit diesem spezialisierten Typ vorkommen.

5.1.6 Sortierung

Eine sehr häufig benötigte Eigenschaft von Listen ist, sie nach einer bestimmten Größenrelation sortieren zu können. Wir wollen in diesem Kapitel Objekte des Typs `String` sortieren. Über die Methode `compareTo` der Klasse `String` läßt sich eine kleiner gleich Relation auf Zeichenketten definieren:

StringOrdering.java

```
1  class StringOrdering{
2    static boolean lessEqual(String x,String y){
3      return x.compareTo(y)<=0;
4    }
5  }
```

Diese statische Methode werden wir zum Sortieren benutzen.

In den folgenden Abschnitte werden wir drei verschiedene Verfahren der Sortierung kennenlernen.

Sortieren durch Einfügen

Die einfachste Methode einer Sortierung ist, neue Elemente in einer Liste immer so einzufügen, daß nach dem Einfügen eine Sortierte Liste entsteht. Wir definieren also zunächst eine Klasse, die es erlaubt, Elemente so einzufügen, daß alle vorhergehenden Elemente kleiner und alle nachfolgenden Elemente größer sind. Diese Klasse braucht unsere entsprechenden Konstruktoren und einen neuen Selektor, der den spezialisierteren Rückgabetyp hat:

SortStringLi.java

```
1  class SortStringLi extends StringLi {
2
3    SortStringLi(){super();}
4    SortStringLi(String x,SortStringLi xs){super(x,xs);}
5    SortStringLi sortTail(){return (SortStringLi)tail();}
```

Die entscheidene neue Methode für diese Klasse ist die Einfügemethode `insertSorted`. Sie erzeugt eine neue Liste, in der das neue Element eingefügt wird:

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

Im Falle einer leeren Liste wird die einelementige Liste zurückgegeben:

```
1    if (isEmpty()) {return new SortStringLi(x,this);}
```

Andernfalls wird unterschieden, ob das einzufügene Element kleiner als das erste Listenelement ist. Ist das der Fall, so wird das neue Element in die Schwanzliste sortiert eingefügt:

```
1    else if (StringOrdering.lessEqual(stringHead(),x)){
2      return new
3      SortStringLi(stringHead(),sortTail().insertSorted(x));
```

Anderfalls wird das neue Element mit dem Konstruktor vorne eingefügt:

```
1    } else return new SortStringLi(x,this);
2  }//method insertSorted
```

Die eigentliche Sortiermethode erzeugt eine leere Ergebnisliste, in die nacheinander die Listenelement sortiert eingefügt werden.

```
1  SortStringLi sort(){
2    SortStringLi result = new SortStringLi();
3
4    for (StringLi xs= this;!xs.isEmpty();xs = xs.stringTail()){
5      result = result.insertSorted(xs.stringHead());
6    }
7
8    return result;
9  }//method sort
```

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

```
1  public static void main(String [] args){
2    SortStringLi xs
3        = new SortStringLi("zz"
4         ,new SortStringLi("ab"
5         ,new SortStringLi("aaa"
6         ,new SortStringLi("aaa"
7         ,new SortStringLi("aaz"
8         ,new SortStringLi("aya"
9         ,new SortStringLi()))))));
```

```
10
11          StringLi ys = xs.sort();
12          System.out.println("Die unsortierte Liste:");
13          System.out.println(xs);
14          System.out.println("Die sortierte Liste:");
15          System.out.println(ys);
16    }
17  }//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 dem Prinzip des Teilens des Problems zu eigen macht; bis die durch Teilen erhaltenen Subprobleme trivial zu lösen sind.[5]

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

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

Die erste Gleichung spezifiziert, daß das Ergebnis der Sortierung einer leeren Liste, eine leere Liste zum Ergebnis hat.

Die zweite Gleichung spezifiziert den Algorithmus für nichtleere Listen. Der in der Gleichung benutzte Operator `++` steht für 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.

[5] Der Name *quick sort* ist insofern nicht immer berechtigt, weil in bestimmten Fällen ist das Verfahren nicht sehr schnell im Vergleich zu anderen Verfahren ist.

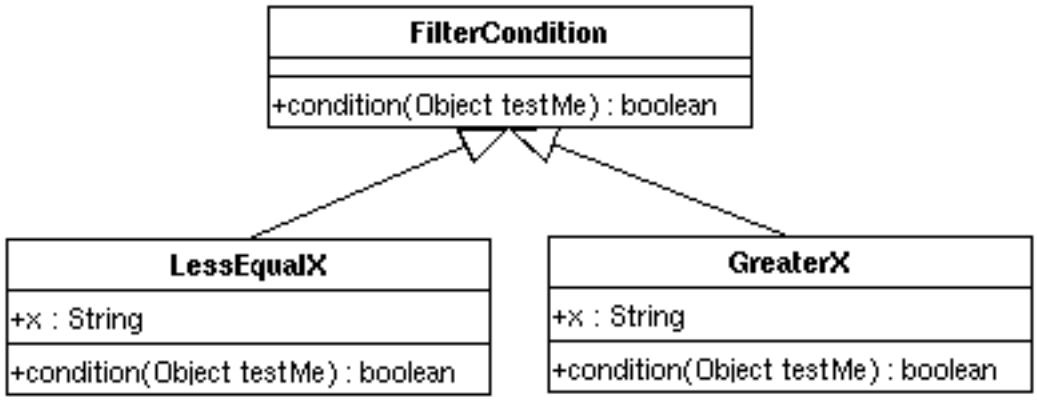
Modellierung Wir wollen die Methode `quicksort` in der Klasse `Li` implementieren, d.h. allgemein für alle Liste zur Verfügung stellen.

Aus der Spezifikation geht hervor, daß wir als zentrales Hilfsmittel eine Methode brauchen, die nach einer bestimmten Bedingung Elemente aus einer Liste filtert. Wenn wir diesen Mechanismus haben, so ist der Rest des Algorithmus mit Hilfe der Methode `concat` trivial direkt aus der Spezifikation ableitbar. Um die Methode `filter` möglichst allgemein zu halten, können wir sie so schreiben, daß sie ein Objekt bekommt, in dem eine Methode die Bedingung, nach der zu filtern ist, angibt. Eine solche Klasse sieht allgemein wie folgt aus:

FilterCondition.java

```
1 class FilterCondition {
2   boolean condition(Object testMe){
3     return true;
4   }
5 }
```

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:



Die beiden Unterklassen brauchen jeweils ein Feld, in dem das Objekt gespeichert ist, mit dem das Elemente im Gröenvergleich getestet wird.

Die für den Sortieralgorithmus benötigte Methode `filter` kann entsprechend ein solches Objekt also Argument bekommen:

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

Codierung Die beiden Klassen für die Bedingung lassen sich relativ einfach aus der Modellierung ableiten.

LessEqualX.java

```
class LessEqualX extends FilterCondition{
  String x;

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

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

Entsprechend für die größer-Relation:

GreaterX.java

```
class GreaterX extends FilterCondition {
  String x;

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

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

In der Klasse `Li` kann nun die Methode `filter` eingeführt werden.

```
class Li {

   ...

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

```
class Li {

   ...

  Li quicksort(){
    Li result = new Li();
    if (!isEmpty()){
      result
       =   //filter the smaller elements out of the tail
           tail().filter(new LessEqualX((String)head()))
            //sort this
            .quicksort()
            //concatenate the sublist of greater elements
            .concat(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 entscheidene Methode `filter` ist parameterisiert über die Bedingung, nach der gefiltert werden soll. Damit läßt sich schnell eine *quicksort* 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 des 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 diese 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:

```
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

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

```
4       }
5   }
```

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 adequate Umsetzung:

StringLessEqual.java

```
1   class StringLessEqual extends Relation {
2       boolean lessEqual(Object x,Object y){
3         return ((String)x).compareTo(y)>0;
4       }
5   }
```

Um den *quicksort* 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 filtert.

OrderingCondition.java

```
1   class OrderingCondition extends FilterCondition{
2     Object x;
3     Relation rel;
4
5     OrderingCondition(Object x,Relation rel){
6       this.x=x;
7       this.rel = rel;
8     }
9
10    boolean condition(Object y){
11      return rel.lessEqual(x,y);
12    }
13  }
```

Entsprechend für die negierte Relation:

NegativeOrderingCondition.java

```
1
2   class NegativeOrderingCondition extends FilterCondition{
3     Object x;
4     Relation rel;
5
6     NegativeOrderingCondition(Obhect x,Relation rel){
7       this.x=x;
8       this.rel = rel;
9     }
10
11    boolean condition(Object y){
12      return !rel.lessEqual(x,y);
13    }
14  }
```

Damit haben wir alle Bausteine zur Hand, mit denen ein über die Ordungsrelation parameterisierte Sortiermethode geschrieben werden kann:

```
class Li {
   ...

  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)
                .concat(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. Im folgenden Beispiel werden Strings nach ihrer lexikographischen Ordnung sortiert.

```
  public static void main(String [] args){
    Li xs
            = new Li("zz"
             ,new Li("ab"
             ,new Li("aaa"
             ,new Li("aaa"
             ,new Li("aaz"
             ,new Li("aya"
             ,new Li()))))));


    Li ys = xs.sortBy(new StringLessEqual());
    System.out.println("Die unsortierte Liste:");
    System.out.println(xs);
    System.out.println("Die sortierte Liste:");
    System.out.println(ys);
  }
```

Aufgabe 16 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 17 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) Schreiben Sie eine Klasse `Student`. Ein Student soll Felder für Namen, Vornamen und Matrikelnummer haben. Implementieren Sie für Studenten eine Methode `String toTableRow()`, die für Studenten eine Zeile eine 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>
```

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 Klasse:
`HtmlView` (`http://www.tfh-berlin.de/~panitz/prog1/../HtmlView.java`)
benutzen.

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. Wir stellen den Algorithmus trotzdem bereits in diesem Kapitel für Listen vor und werden ihn später auch noch einmal für Reihungen implementieren.

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 entgültige Stelle geblubbert.

Implementierung auf Listen Die fundamentale Methode der Blubbersortierung ist das eigentliche Blubbern, in dem die Liste einmal durchlaufen wird, und das Ergebnis eine Liste ist, in der unter Umständen Elemente getauscht wurden. Das Ergebnis eines Blubberdurchgangs ist entsprechend nicht nur die geblubberte Liste, sondern auch bool'scher Wert, der angibt, ob Elemente bei dem Blubberdurchgang vertauscht wurden. Das Ergebnis ist also ein Paar von zwei Objekten. Um mit Methoden zwei Ergebnisswerte zurückgeben zu können, definieren wir uns zunächst die Klasse `Pair`, die zwei Objekte beliebigen Typs speichern kann.

Pair.java
```
1 class Pair {
2   Object fst;
3   Object snd;
4   Pair(Object fst,Object snd){
5     this.fst=fst;
6     this.snd=snd;
7   }
8 }
```

Mit den Feldern `fst` und `snd` kann jeweils auf eines der beiden Objekte zugegriffen werden.

Die Methode `bubble` kann jetzt so definiert werden, daß sie ein Objekt der Klasse `Pair` als Ergebnis hat. Im Feld `fst` dieses Objektes wird ein Objekt des Typs `Boolean` stehen, der angibt, ob Elemente vertauscht wurden. Im Feld `snd` steht die eigentliche Liste. Wir implementieren die Blubbersortierung für die Klasse `Li`.

```
1  Pair/*Boolean,Li*/ bubble (Relation rel){
```

Zunächst legen wir lokale Felder an für die zu bearbeitende Liste, für das Ergebnis und ein bool'sches Feld, welches angibt, ob wir zwei Elemente vertauscht haben.

```
1    Li bubbleMe = this;
2    Li bubbleResult = new Li();
3    boolean bubbleMade = false;
```

Im einfachsten Fall ist die Liste leer:

```
1    if (bubbleMe.isEmpty()) {bubbleMade= false;
2    }
```

auch für einelementige Liste ist die Lösung einfach:

```
1    else if (bubbleMe.tail().isEmpty()) {
2      bubbleResult = bubbleMe;bubbleMade= false;
3    }
```

Im Falle von Listen mit mindestens zwei Elementen, können die ersten beiden Elemente verglichen werden. Entweder sind diese bereits in der richtigen Reihenfolge, dann kann auf der Restliste rekursiv weitergearbeitet werden:

```
1    else if(rel.lessEqual(bubbleMe.head()
2                              ,bubbleMe.tail().head()))
3    {
4      Pair furtherBubbleRes = bubbleMe.tail().bubble(rel);
5      bubbleResult
6        = new Li(bubbleMe.head(),(Li)furtherBubbleRes.snd);
7      bubbleMade
8        = ((Boolean)furtherBubbleRes.fst).booleanValue();
9    }
```

Sind sie nicht in der richtigen Reihenfolge, so werden der Kopf und der Kopf des Schwanzes getauscht und die Methode für diesen neuen Schwannz rekursiv aufgerufen:

```
1    else {
2      bubbleMade = true;
3      Li furtherBubble
4       = (Li)new Li(bubbleMe.head(),bubbleMe.tail().tail())
5             .bubble(rel).snd;
6      bubbleResult
7       =new Li(bubbleMe.tail().head(),furtherBubble);
8    }
```

Schließlich wird das Ergebnisobjekt erzeugt und zurückgegeben.

```
    return new Pair(new Boolean(bubbleMade),bubbleResult);
  }
```

Die Methode `bubble` leistet den Hauptteil des Sortieralgorithmus. Sie braucht nun nur noch so oft in einer Schleife auf eine Liste aufgerufen werden, bis die Liste sortiert vorliegt:

```

  Li bubbleSortBy(Relation rel){
    Li bubbleMe = this;
    boolean doBubble = true;

    while (doBubble){
      Pair furtherBubbleRes = bubbleMe.bubble(rel);
      doBubble = ((Boolean)furtherBubbleRes.fst).booleanValue();
      bubbleMe = (Li)furtherBubbleRes.snd;
    }
    return bubbleMe;
  }
```

Kapitel 6

Weiterführende programmiersprachliche Konzepte

6.1 Pakete

Java bietet die Möglichkeit, Klassen in Pakete 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:

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

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

TestPaket.java

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

```
4       System.out.println("hello from package \'testpackages\'");
5     }
6   }
```

Paketnamen werden per Konvention in lateinischer Schrift immer mit Kleinbuchstaben als ersten 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, wird der*vollqualifizierte Name* der Klasse bezeichnet, in 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

Der Javainterpreter `java` sucht nach Klassen in den Ordnern entsprechend ihrer Paketstruktur. `java` erwartet also, daß die obige Klasse `Test` in einem Ordner `testPackages`, der ein Unterordner des Ordners `panitz`, der ein Unterordner des Ordners `tfhberlin` usw. ist. Ausgehend sucht `java` diese Ordner von einem oder mehreren Startordnern, in denen Pakete und Klassen liegen.

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 ihrer Pakete entsprechenden Ordner schreibt. Die Option heißt `-d`. Dem `-d` ist nachgestellt, von welchem Startordner 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 ein Ordner `de` mit Unterordner `tfhberlin` etc. erzeugt.

6.1.3 Starten von Klassen in Paketen

Um Klassen vom Javainterpreter zu starten, reicht es nicht ihren Namen anzugeben, sondern dr 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:~/>
```

[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.

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

6.1.4 Das Java Standardpaket

Die mit Java mitgelieferten Klassen sind auch in Pakete 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 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

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

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

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

```
1  package de.tfhberlin.panitz.utilTest;
2
```

```
3  import java.util.ArrayList;
4
5  class TestImport {
6    public static void main(String [] args){
7      ArrayList xs = new ArrayList();
8      xs.add("friends");
9      xs.add("romans");
10     xs.add("countrymen");
11     System.out.println(xs);
12   }
13 }
```

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

TestImport2.java

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

Importieren von Paketen

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

TestImport3.java

```
1  package de.tfhberlin.panitz.utilTest;
2
3  import java.util.*;
4
5  class TestImport3 {
```

```
6     public static void main(String [] args){
7       ArrayList xs = new ArrayList();
8       xs.add("friends");
9       System.out.println(xs);
10
11      Vector ys = new Vector();
12      ys.add("romans");
13      System.out.println(ys);
14    }
15  }
```

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

Eine Importanweisung für Pakete importiert nicht die Klassen aus Unterpaketen dieses Pakets.

6.2 Sichtbarkeitsattribute

Sichtbarkeiten[2] 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 eine Klasse benutzt werden, oder nur von Klasse im gleichem Paket. Syntaktisch wird dieses dadurch ausgedrückt, daß der Klassendefinition entweder das Schlüsselwort `public` vorangestellt ist, oder aber kein solches Attribut voransteht:

MyPublicClass.java

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

MyNonPublicClass.java

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

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

UsePublic.java

```
1  package de.tfhberlin.panitz.p2;
2
3  import de.tfhberlin.panitz.p1.*;
```

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

```
4
5  class UsePublic {
6      public static void main(String [] args){
7        System.out.println(new MyPublicClass());
8      }
9  }
```

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

UseNonPublic.java

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

Java gibt bei der Übersetzung eine entsprechende gut verständliche Fehlermeldung:

```
sep@swe10:~> javac -d . UseNonPublic.java
UseNonPublic.java:7: 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 erlaubbt 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 werden, 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, in dem sich die Klassen befinden ab, hinzu 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 Untelasse benutzt werden.
kein Attribut	Die Eigenschaft darf nur von Klassen im gleichem Paket benutzt werden.
`private`	Die Eigenschaft darf nur von der Klasse, in der es definiert ist, benutzt werden.

Damit kann in einer Klasse auf Eigenschaften mit jeder dieser vier Sichtbarkeiten zugegriffen werden:

VisibilityOfFeatures.java

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

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

PrivateTest.java

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

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

PackageTest.java

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

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

ProtectedTest.java

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

Java wird in seinem Sichtbarkeitskonzept oft kritisiert und das von zwei Seiten. Einerseits ist es mit den vier Sichtbarkeiten schon relativ unübersichtlich. Die verschiedenen Konzepte der Vererbung und der Pakete spielen bei Sichtbarkeiten eine Rolle; andererseits ist es nicht vollständig genug und kann evrschiedene 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 Standardsichbarkeit der Paketsichtbarkeit.

6.3 Schnittstellen (Interfaces) und abstrakte Klassen

Wir haben schon einige Situationen kennen gelernt, 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 sollte ja nie benutzt werden, sondern die überschreibenden Methoden in den Unterklassen. Beispiele für solche Klassen waren die Sortierrelationen in den Sortieralgorithmen, oder auch die Klasse `String2String`, 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 `String2String` eine entsprechende Schnittstelle schreiben:

DialogueFunction.java

```
1 package de.tfhberlin.panitz.dialoguegui;
2
3 public interface DialogueFunction {
4
5   public String execute(String input);
6
7 }
```

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:

UpperCaseFunction.java

```
1  package de.tfhberlin.panitz.dialoguegui;
2
3  public class UpperCaseFunction implements DialogueFunction{
4
5    public String execute(String input){
6      return input.toUpperCase();
7    }
8
9  }
```

Die Klausel `implements DialogueFunction` verspricht, daß in dieser Klasse für alle Methoden aus der Schnittstelle eine Implementierung existiert. In unserem Beispiel brauchte nur eine Methode implementiert werden, die Methode `execute`.

Im Gegensatz zur `extends`-Klausel von Klassen können in einer `implements`-Klausel auch mehrere Schnittstellen angegeben werden, die implementiert werden.

Definieren wir zum Beispiel ein zweite Schnittstelle:

ToHTMLString.java

```
1  package de.tfhberlin.panitz.html;
2
3  public interface ToHTMLString {
4
5    public String toHTMLString();
6
7  }
```

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 beide Schnittstellen implementiert.

Test.java

```
1  package de.tfhberlin.panitz.interfaceTests;
2
3  import de.tfhberlin.panitz.html.*;
4  import de.tfhberlin.panitz.dialoguegui.*;
5
6  public class Test implements ToHTMLString, DialogueFunction {
7
8    String überschrift;
9      String body;
10
```

```
11    public Test(String überschrift,String body){
12      this.überschrift = überschrift;
13      this.body = body;
14    }
15
16    public String toHTMLString(){
17       return "<html><title>"+überschrift+"</title>"
18              +"<body>"+body+"</body>";
19    }
20
21    public String execute(String input){
22      body=input;
23      return toHTMLString();
24    }
25  }
```

Schnittstellen können auch einander erweitern. Dieses geschieht dadurch, daß Schnittstellen auch eine `extends`-Klausel haben.

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.

Als etwas ausführlicheres Beispiel folgt die Klasse `Dialogue`, die ein kleines GUI definiert, angegeben. Wir haben diese Klasse bereits in Teilen zuvor gesehen. Für die Funktionalität des GUI wird jetzt keine Klasse angegeben, sondern nur die Schnittstelle `DialogueFunction`.

In der Klasse werden innere Klassen definiert und benutzt, was wir noch nicht kennen gelernt haben; trotzdem sollte das Programm zu lesen und zu verstehen sein.

Dialogue.java

```
1  package de.tfhberlin.panitz.dialoguegui;
2
3  //import some of Java's gui tool kit packages
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class Dialogue extends JPanel {
9
```

```
10  //first we define object for the graphical components
11
12  /** a window frame **/
13  private JFrame frame              = new JFrame("Dialogue");
14  /** a button **/
15  private JButton button            = new JButton("ausführen");;
16  /** to text areas **/
17  private JTextArea eingabe         = new JTextArea(10,40);
18  private JTextArea ausgabe         = new JTextArea(10,40);
19
20
21  /** two objects, which control the behaviour of th GUI**/
22  ButtonListener myListener = new ButtonListener();
23  DialogueFunction funktion;
24
25  /** construction of a Dailogue involves adding its graphical
26      components to it.
27
28      We use the grid layout.
29
30      param funktionalität defines the function, which is
31         excecuted, when the button is pressed.
32  **/
33  public Dialogue(DialogueFunction funktionalität) {
34    //initialize the function field
35    funktion = funktionalität;
36
37    //add functionality to the button
38    button.addActionListener(myListener);
39
40    //set the layout and add the gui components
41    setLayout(new GridLayout(3,1));
42    add(eingabe);
43    add(button);
44    add(ausgabe);
45  }
46
47  /** this internally defined class implements, what happens,
48      when the button is pushed.
49  **/
50  class ButtonListener implements ActionListener {
51    public void actionPerformed(ActionEvent e) {
52      ausgabe.setText(funktion.execute(eingabe.getText()));
53    }
54  }
55
56  void run(){
57
58    // this will exit the programm when someone closes the window
59    frame.addWindowListener(new WindowAdapter() {
```

```
60        public void windowClosing(WindowEvent e) {System.exit(0);}
61      });
62
63      // add this gui component to the window
64      frame.getContentPane().add(this);
65      frame.pack();
66      frame.setVisible(true);
67    }
68  }
```

In der Klasse `Dialogue` wird noch keine konkrete Implementierung der Schnittstelle `DialogueFunction` angegeben. Erst wenn wir sie benutzen wollen, brauchen wir ein Objekt, welches diese Schnittstelle implementiert:

DialogueTest.java

```
1  package de.tfhberlin.panitz.dialoguegui;
2
3  public class DialogueTest {
4
5    public static void main(String [] args){
6      new Dialogue(new UpperCaseFunction()).run();
7    }
8  }
```

Der Konstruktor der Klasse `Dialogue` kann mit einem Objekt der Klasse `UpperCaseFunction` aufgerufen werden, weil die Klasse `UpperCaseFunction` die Schnittstelle `DialogueFunction` implementiert.

Semantische Einschränkungen für Schnittstellen

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

- Schnittstellen können nur Schnittstellen, nicht aber Klassen erweitern.
- Jede Methode einer Schnittstelle muß öffentlich sein, braucht also das Attribut `public`. Wenn dieses für eine Methode nicht deklariert ist, so wird Java dieses von selbst hinzufügen. Trotzdem müssen implementierende Klasse 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:

DialogueFunction.java

```
1  package de.tfhberlin.panitz.dialoguegui;
2
3  public interface DialogueFunction
4
```

```
5   abstract public String execute(String input);
6
7   }
```

- Felder einer Schnittstelle sind immer statisch brauchen also das Attribut `static` und brauchen zusätzlich noch das Attribut `final`.

Aufgabe 18 Für die Lösung dieser Aufgabe gibt es 3 Punkte, die auf die Klausur angerechnet werden. Voraussetzung hierzu ist, daß die Lösung mir spätestens in der Übung am 19.12.2002 gezeigt und erklärt werden kann.

In dieser Aufgabe wird ein kleines Programm, das einen Psychoanalytiker simuliert, vervollständigt.

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

b) In der Klasse `Li`, sind die Methoden:
`reverse`, `words`, `unwords`, `stripPunctuation`, `drop`,
`tails`, `isPrefixOf`, `isPrefixIgnoreCaseOf`
noch nicht ausprogrammiert. Implementieren Sie diese Methoden und schreiben Sie Tests für jede Methode.

Wenn Ihre Tests erfolgreich sind, starten Sie die `main`-Methode der Klasse `Eliza`.

c) Ergänzen sie in der Klasse `Eliza` die Liste im Feld `respMsgs` um die auskommentierten Einträge.

d) Erfinden Sie eigene Einträge für die Liste `respMsgs`.

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

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

```
1  package de.tfhberlin.sep.abstractList;
2  abstract class AbstractList  {
3    abstract public boolean isEmpty();
4
5    abstract public Object head();
6    abstract public AbstractList tail();
7
8    abstract public AbstractList empty();
9    abstract public AbstractList cons(Object x,AbstractList xs);
10
11   public int length(){
12     if (isEmpty()) return 0;
13     return 1+tail().length();
14   }
15
16   public AbstractList concat(AbstractList other){
17     if (isEmpty()) return this;
18     return cons(head(),tail().concat(other));
19   }
20 }
```

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 kann.

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 durch die `extends`-Klausel durch andere Klassen erweitert werden und selbst `implements-` und `extends-`Klauseln erweitert werden. 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

```
1  package de.tfhberlin.sep.abstractList;
2  public class LinkedList extends AbstractList {
3
4    private Object hd=null;
5    private AbstractList tl=null;
6    private boolean empty;
7
8    public LinkedList(){empty=true;}
9    public LinkedList(Object x,AbstractList xs){
10     hd=x;tl=xs;empty=false;
11   }
12
13   public boolean isEmpty(){return empty;}
14
15   public Object head(){return hd;}
16   public AbstractList tail(){return tl;}
17
18   public AbstractList empty(){return new LinkedList();}
19   public AbstractList cons(Object x,AbstractList xs){
20      return new LinkedList(x,xs);
21    }
22
23 }
```

Die Implementierungen entsprechen der Implementierung die wir bereits schon 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 all abstrakten Eigenschaften, die sie auf irgendeinen Weg erbt, implementieren. Ein typisches Szenario für für das Zusammenspiel von Schnittstellen und abstrakten Klassen können wir für unsere Listenimplementierung angeben. Eine Schnittstelle beschreibt zunächst die für Listen relevanten Methoden:

List.java

```
1  package de.tfhberlin.sep.listen;
2  public interface List  {
3    boolean isEmpty();
4
5    Object head();
6    AbstractList tail();
7
8    public int length();
9    public AbstractList concat(AbstractList other);
10 }
```

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

```
1  package de.tfhberlin.sep.listen;
2  public abstract class AbstractList  implements List{
3
4    abstract public AbstractList empty();
5    abstract public AbstractList cons(Object x,AbstractList xs);
6
7    public int length(){
8      if (isEmpty()) return 0;
9      return 1+tail().length();
10   }
11
12   public AbstractList concat(AbstractList other){
13     if (isEmpty()) return this;
14     return cons(head(),tail().concat(other));
15   }
16 }
```

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.

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 Befahl, 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 und unterrichtet die virtuell 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 fakultaet(int n){
5      if (i==0) return 1;
6      if (i<0) throw new RuntimeException();
7      return n*fakultät(n-1);
8    }
9
10   public static void main(String [] args){
11     System.out.println(fakultaet(5));
12     System.out.println(fakultaet(-3));
13     System.out.println(fakultaet(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 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 unterreichtet und Java in der ersten Zeile davon, daß eine Ausnahme des Typs `RuntimeException` geworfen wurde. In der zweiten Zeile erfahren wir, daß dieses in 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

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

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 führ dieses schon zu einen Übersetzungsfehler.

Folgende Klasse:

NotThrowable.java

```
1 package de.tfhberlin.panitz.exceptions;
2 public class NotThrowable {
3
4   public static void main(String [] args){
5       throw "i am not throwable";
6   }
7 }
```

führt zu einen 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 von Methoden aus Klassen, die wir selbst benutzen, geworfen werden. So kann z.B. die Benutzung der Methode `charAt` aus Klasse `String` dazu führen, daß eine Ausnahme geworfen wird.

ThrowIndex.java

```
1 package de.tfhberlin.panitz.exceptions;
2 public class ThrowIndex {
3
4   public static void main(String [] args){
5       "i am too short".charAt(120);
6   }
7 }
```

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: package de.tfhberlin.panitz.exceptions;

SecondThrow.java

```
1  public class SecondThrow {
2
3    public static int fakultät(int n){
4      if (n==0) return 1;
5      if (n<0)
6       throw
7        new RuntimeException
8             ("negative Zahl für Fakultätsberechnung");
9      return n*fakultät(n-1);
10   }
11
12   public static void main(String [] args){
13     System.out.println(fakultät(5));
14     System.out.println(fakultät(-3));
15     System.out.println(fakultät(4));
16   }
17 }
```

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:

ThirdThrow.java

```
1  package de.tfhberlin.panitz.exceptions;
2  public class ThirdThrow {
3
4    public static int fakultät(int n){
5      if (n==0) return 1;
6      if (n<0) throw new Exception
7                ("negative Zahl für Fakultätsberechnung");
8      return n*fakultät(n-1);
9    }
10
```

```
11    public static void main(String [] args){
12      System.out.println(fakultät(5));
13      System.out.println(fakultät(-3));
14      System.out.println(fakultät(4));
15    }
16  }
```

Bei der Übersetzung kommt es zu folgenden 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 dr 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

```
1   package de.tfhberlin.panitz.exceptions;
2   public class FourthThrow {
3
4     public static int fakultät(int n) throws Exception{
5       if (n==0) return 1;
6       if (n<0)
7        throw
8         new Exception("negative Zahl für Fakultätsberechnung");
9       return n*fakultät(n-1);
10    }
11
12    public static void main(String [] args) throws Exception{
13      System.out.println(fakultät(5));
14      System.out.println(fakultät(-3));
15      System.out.println(fakultät(4));
16    }
17  }
```

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 werden, und ebenso wie die Standardausnahmen geworfen werden:

NegativeNumberException.java

```
1  package de.tfhberlin.panitz.exceptions;
2  public class NegativeNumberException extends Exception {
3  }
```

So daß unser Beispielprogramm jetzt unsere eigene Ausnahme werfen kann:

FifthThrow.java

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

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, 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 im Falle des Auftretens einer Ausnahme für Code zur Ausnahmebehandlung auszuführen ist. Jetzt können wir Programmieren, qie im Falle einer Ausnahme zu verfahren ist:

Catch1.java

```
1  package de.tfhberlin.panitz.exceptions;
2  public class Catch1 {
3
4    public static int fakultät(int n) throws Exception{
5      if (n==0) return 1;
6      if (n<0) throw new NegativeNumberException();
7      return n*fakultät(n-1);
8    }
9
10   public static void main(String [] args){
11     try {
12       System.out.println(fakultät(5));
13       System.out.println(fakultät(-3));
14       System.out.println(fakultät(4));
15     }catch (Exception e){
16       System.out.println("Ausnahme aufgetreten: "+e);
17     }
18   }
19 }
```

Wie man sieht, braucht jetzt die Methode `main` nicht mehr zu deklarieren, daß sie eine Ausnahme wirft, denn sie fängt ja alls 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, da 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

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

```
3
4     public static int fakultät(int n) throws Exception{
5       if (n==0) return 1;
6       if (n<0) throw new NegativeNumberException();
7       return n*fakultät(n-1);
8     }
9
10    public static void main(String [] args){
11      try {
12        System.out.println(fakultät(5));
13      }catch (Exception _){
14        System.out.println("Ausnahme für Fakultät von 5");
15      }
16      try {
17        System.out.println(fakultät(-3));
18      }catch (Exception _){
19        System.out.println("Ausnahme für Fakultät von -3");
20      }
21      try {
22        System.out.println(fakultät(4));
23      }catch (Exception _){
24        System.out.println("Ausnahme für Fakultät von 4");
25      }
26    }
27  }
```

Dieses Programm[3] erzeugt folgende Ausgabe auf dem Bildschirm:

```
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

```
1  package de.tfhberlin.panitz.exceptions;
2  public class Catch3 {
3
4    public static int fakultät(int n)
```

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

```
5                          throws NegativeNumberException{
6     if (n==0) return 1;
7     if (n<0) throw new NegativeNumberException();
8     return n*fakultät(n-1);
9   }
10
11  public static void main(String [] args){
12    try {
13      System.out.println(fakultät(5));
14      System.out.println(fakultät(-3));
15      System.out.println(fakultät(4));
16    }catch (NegativeNumberException e){
17      System.out.println("Ausnahme aufgetreten: "+e);
18    }
19  }
20 }
```

Abfangen mehrerer Ausnahmen

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

Hierzu definieren wir uns zunächst eine weitere Ausnahmeklasse:

NumberTooLargeException.java

```
1 package de.tfhberlin.panitz.exceptions;
2 public class NumberTooLargeException extends Exception {
3 }
```

Jetzt können wir unterschiedliche Ausnahmen werfen und wieder fangen:

Catch4.java

```
1  package de.tfhberlin.panitz.exceptions;
2  public class Catch4 {
3
4    public static int fakultät(int n)
5          throws NegativeNumberException, NumberTooLargeException{
6      if (n==0) return 1;
7      if (n<0) throw new NegativeNumberException();
8      if (n>20) throw new NumberTooLargeException();
9      return n*fakultät(n-1);
10   }
11
12   public static void printFakultät(int i){
13     try {
14       System.out.println(fakultät(i));
15     }catch (NegativeNumberException _){
16       System.out.println("Fakultät von negativer Zahl");
17     }
```

```
18      catch (NumberTooLargeException _){
19        System.out.println("Fakultät von zu großer Zahl");
20      }
21    }
22
23    public static void main(String [] args){
24      printFakultät(30);
25      printFakultät(-3);
26      printFakultät(4);
27    }
28  }
29
```

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 Ausnahme wichtig darauf zu achten, daß auf 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:

WrongCatch.java

```
1   package de.tfhberlin.panitz.exceptions;
2   public class WrongCatch {
3
4     public static int fakultät(int n)
5           throws NegativeNumberException, NumberTooLargeException{
6       if (n==0) return 1;
7       if (n<0) throw new NegativeNumberException();
8       if (n>20) throw new NumberTooLargeException();
9       return n*fakultät(n-1);
10    }
11
12    public static int checkFakultät(int i){
13      try {
14        return fakultät(i);
15      }catch (Exception e){
16        System.out.println("Ausnahme "+e+" aufgetreten");
17      }
18    }
19
```

```
20    public static void main(String [] args){
21      System.out.println(checkFakultät(30));
22      System.out.println(checkFakultät(-3));
23      System.out.println(checkFakultät(4));
24    }
25  }
```

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 Befehlen 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

```
1   package de.tfhberlin.panitz.exceptions;
2   public class Catch5 {
3
4     public static int fakultät(int n)
5           throws NegativeNumberException, NumberTooLargeException{
6       if (n==0) return 1;
7       if (n<0) throw new NegativeNumberException();
8       if (n>20) throw new NumberTooLargeException();
9       return n*fakultät(n-1);
10    }
11
12    public static int checkFakultät(int i){
13      try {
14        return fakultät(i);
15      }catch (Exception e){
16        System.out.println("Ausnahme "+e+" aufgetreten");
17        return 0;
18      }
19    }
20
21    public static void main(String [] args){
22      System.out.println(checkFakultät(30));
23      System.out.println(checkFakultät(-3));
24      System.out.println(checkFakultät(4));
25    }
26  }
27
```

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 4 mal 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. Alle Methoden, die dort verzeichnet sind, wurden bereits angefangen auszuführen, ihre Ausführung ist 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

```
1  class Finally {
2
3    static void m(int i) throws Exception{
4      if (i>0) throw new Exception();
5    }
6
7    public static void main(String [] args){
8      try {m(1);}
9      catch (Exception _){System.out.println("Ausnahme gefangen");}
10     finally {System.out.println("erster Test");}
11
12     try {m(-1);}
13     catch (Exception _){System.out.println("Ausnahme gefangen");}
14     finally {System.out.println("zweiter Test");}
15   }
16 }
```

Die `finally`-Klausel wird nicht nur ausgeführt, wenn keine Ausnahme geworfen wurde oder eine Ausnahme gefangen wurde, sondern auch, wenn eine Ausnahme geworfen wurde, für die es keine `catch`-Klausel gibt[4].

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

MoreFinally.java

```
1  class MoreFinally {
2
3      static void m(int i){
```

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

```
4           if (i<0) throw new NullPointerException();
5       }
6
7       public static void main(String [] args){
8           try {m(-1);}
9           catch (IndexOutOfBoundsException _){}
10          finally
11          {System.out.println("wird trotzdem ausgegeben");}
12      }
13
14  }
```

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, also auch auszuführen ist, 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 über eine Bedingung angefragt wird, wird als Ausnahmefall behandelt. In unserer Listenimplementierung wurde für die leere Liste von den Mathoden `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.

```
1   public int length(){
2
3     try {
4       return 1+tail().length();
5     }catch (NullPointerException _){
6       return 0;
7     }
8   }
```

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 Statndardklassen daher. In diesen 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 Platformunabhängiger 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

```
1  class EqualVsIdentical {
2
3    public static void main(String [] args){
4      String x = "hallo".toUpperCase();
5      String y = "hallo".toUpperCase();
6
7      System.out.println("x: "+x);
8      System.out.println("y: "+y);
9
10     System.out.println("x==x        -> "+(x==x));
11     System.out.println("x==y        -> "+(x==y));
12     System.out.println("x.equals(x) -> "+(x.equals(x)));
13     System.out.println("x.equals(y) -> "+(x.equals(y)));
14   }
15
16 }
```

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)` undy`.equals(z)` dann gilt auch `x.equals(z)`

- **Konsistenz**: wiederholte Aufrufe von `equals` aus dieselben Objekte, ergibt 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 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 desn *Hashcode* des Objektes nehmen, und wissen, es kann nur in einen der 1000 Regale liegen. Sie brauchen also nur, in einen und nicht in 1000 Regalen nach ihren 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. Er 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.

NoClone.java

```
1 class NoClone {
2   String x = "hallo";
```

[1] Hierbei handelt es sich in keinster Weise um irgendwelche unerlaubten Substanzen.

[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.

```
3   public static void main(String [] args)
4     throws CloneNotSupportedException{
5     NoClone nc = new NoClone();
6     NoClone cc = nc.clone();
7     System.out.println(cc.x);
8   }
9 }
```

Diese 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

```
1 class Clone implements Cloneable {
2   String x = "hallo";
3
4   public static void main(String [] args)
5       throws CloneNotSupportedException{
6     Clone nc = new Clone();
7     Clone cc = (Clone)nc.clone();
8     System.out.println(cc.x);
9   }
10 }
```

Führt jetzt zu einen 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 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 entgültig aus dem Speicher gelöscht werden, sie ist für ihre Klasse die Methode `finalize` zu überschreiben. Java keinerlei Garantie darüber, wann die Methode `finalize` ausgeführt wird. Es kann relativ zufällig sein, wie lange sich ein Objekt im Speicher befindet.

toString

`public String toString()` ist eine Methode, die wir schon oft benutzt 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 werden können, 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, die einen Wert des entsprechenden primitiven Typen als Parameter haben und einen Konstruktor, der einen String als einen bestimmten Wert interpretiert. Objekte dieser Klassel 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 berechnet, 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

```
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 Methoden wie `add`, die in einer Liste ein Objekt zu eingefügt hat. Bei uns wurde stets eine neue Liste mit einem zusätzlich vorn angehängten Element erzeugt. Javas Standardlisten haben Methoden, die den Inhalt eines Listenobjetes 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 ungefährt 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

```
1  import java.util.List;
2  import java.util.ArrayList
3
4  class ListUsage{
5
6    public static void main(String [] args){
7      List xs = new ArrayList();
8      xs.add("hallo");
9      xs.add("welt");
10     xs.add("wie");
11     xs.add("geht");
12     xs.add("es");
```

```
13      xs.add("dir");
14      System.out.println(xs);
15    }
16
17  }
```

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 für 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.hshCode()`

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 eigene Listen schreiben:

LiIterator.java

```
1  package de.tfhberlin.panitz.list;
2
3  public class LiIterator implements java.util.Iterator{
4
5    public LiIterator(Li xs){this.xs=xs;}
6
7    private Li xs;
8
9    public boolean hasNext() {return !xs.isEmpty();}
10
11   public Object next(){
12     Object result = xs.head();
13     xs = xs.tail();
14     return result;
15   }
16
17   public void remove(){
18    throw new UnsupportedOperationException ();
19   }
20 }
```

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

```
1    public java.util.Iterator iterator(){
2      return new LiIterator(this);
3    }
```

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 heirzu 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

```
1  class Iterate {
2
3    static public String unwords(Collection cs){
4      String result = "";
```

```
5
6       for (Iterator it=cs.iterator();it.hasNext();){
7         result = result+" "+it.next();
8       }
9     }
10
11    static public void main(String [] _){
12      Collection cs = new ArrayList();
13      cs.add("lebt");
14      cs.add("wohl");
15      cs.add("gott");
16      cs.add("weiß");
17      cs.add("wann");
18      cs.add("wir");
19      cs.add("uns");
20      cs.add("wiedersehen");
21
22      System.out.println(unwords(cs));
23    }
24  }
```

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

```
1       for (Iterator it=cs.iterator();it.hasNext();){
2         result = result+" "+it.next();
3         System.out.println(it.next())
4       }
```

War wahrscheinlich das gewünschte Verhalten:

```
1       for (Iterator it=cs.iterator();it.hasNext();){
2         Object theNext = it.next();
3         result = result+" "+theNext;
4         System.out.println(theNext);
5       }
```

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

Abbildung.java

```
1  package de.tfhberlin.panitz.list;
2
3  class Abbildung extends Li{
4    public Abbildung(Object schlüssel,Object wert,Abbildung xs){
5        super(new Pair(schluessel,wert),xs);
6    }
7
8    public Abbildung(){
9    }
10
11   Object lookup(Object schluessel){
12     Object result = null;
13
14     for (Abbildung xs = this
15         ;!this.isEmpty()
16         ;xs=(Abbildung)xs.tail()){
17       Pair p = (Pair)xs.head();
18       if (p.fst.equals(schluessel)){
19         return p.snd;
20       }
21
22     }
23     return result;
24   }
25
26 }
```

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

Dictionary.java

```
1  package de.tfhberlin.panitz.list;
2
3  class Dictionary {
4    static final Abbildung dic =
5       new Abbildung("Menge","set"
6      ,new Abbildung("Abbildung","map"
7      ,new Abbildung("Liste","list"
8      ,new Abbildung("Iterator","iterator"
9      ,new Abbildung("Schnittstelle","interface"
10     ,new Abbildung("Klasse","class"
```

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

```
11      ,new Abbildung()))))));
12
13    static String english(String deutsch){
14      return (String)dic.lookup(deutsch);
15    }
16
17    public static void main(String [] _){
18      System.out.println(english("Schnittstelle"));
19      System.out.println(english("Abbildung"));
20    }
21  }
```

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

```
1   import java.util.*;
2
3   class Dic{
4
5     static Map map = new HashMap();
6
7     static {
8       map.put("Menge","set");
9       map.put("Abbildung","map");
10      map.put("Liste","list");
11      map.put("Iterator","iterator");
12      map.put("Schnittstelle","interface");
13      map.put("Klasse","class");
14    }
15
16    static String english(String deutsch){
17     return (String)dic.get(deutsch);
18    }
```

```
19
20    public static void main(String [] _){
21      System.out.println(english("Schnittstelle"));
22      System.out.println(english("Abbildung"));
23    }
24
25  }
```

7.4.5 Weitere Sammlungsmethoden

In der Klasse `java.util.Collections` befinden sich statische Methoden zur Maniputaion 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, ob zwei Elemente in der kleiner Relation stehen, so gibt es in Java die Schnitstelle `java.util.Comparator`. In dieser Klasse gibt es die Methode `int compare(Object o1, Object o2)`, deren Ergebnis eine negative Zahl, o, 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 Standardvergleich für Strings anwendet:

ReverseComporator.java

```
1  import java.util.Comparator;
2  class ReverseComparator implements Comparator {
3
4    public int compare(Object o1, Object o2) {
5      return reverse((String) o1).compareTo(reverse((String) o2));
6    }
7
8    static private String reverse(String str){
9      StringBuffer result = new StringBuffer();
10     for (int i=str.length()-1;i>=0;i=i-1){
11       result.append(str.charAt(i));
12     }
13     return result.toString();
14   }
15
16 }
```

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

Rhyme.java

```
1  import java.util.*;
2
```

```
3  class Rhyme {
4
5    public static List/*String*/ rhyme(List/*String*/ xs){
6      Collections.sort(xs,new ReverseComparator());
7      return xs;
8    }
9
10   public static void main(String [] args){
11       List xs = new ArrayList();
12       xs.add("maus");
13       xs.add("rasieren");
14       xs.add("bauhaus");
15       xs.add("verlieren");
16       xs.add("regen");
17       xs.add("laus");
18       xs.add("segen");
19       xs.add("kapitulieren");
20       xs.add("monieren");
21       xs.add("existieren");
22       xs.add("kraus");
23       rhyme(xs);
24       System.out.println(xs);
25   }
26
27 }
```

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 gleichem 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] haben Reihungen einen festen Elementtypen.

[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.

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 Objekttypen als auch von einem primitiven Typen sein, also gibt es auch den Typen `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 `lengthvom` Typ `int`, das die feste Länge einer Reihungen angibt.

ObjectArray.java

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

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

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

Wie man beim Starten dieser kleiner 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 nut 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 einen Komplex von vielen einzelnen Feldern gesehen werden. Die Felder haben keine eigenen Namen, sondern werden über den Namen der Reihung zusammen mit ihren 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 dafür, daß Information verloren geht. Hierzu betrachte man folgendes kleine Beispiel.

Kovarianz.java

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

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

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

denselben Elementtypen 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` nicht so schreiben, daß sie für Reihungen mit beliebigen Elementtypen[7] angeendet 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

```
1 public static void arraycopy(Object src,
2                               int srcPos,
3                               Object dest,
4                               int destPos,
5                               int length)
```

zum Kopieren bestimmter Reihungsabschnitte in eine zweite Reihung.

7.5.7 Reihungen von Reihungen

Da Reihungen ganz normale Objekttypen sind, lassen sich für Reihungen ebenso wie es Listen von Listen geben kann, auch Reihungen von Reihungen erzuegen. 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

```
1  class SchachFeld {
2    Schachfigur [][] spielFeld
3      =  {new Schachfigur[8]
4         ,new Schachfigur[8]
5         ,new Schachfigur[8]
6         ,new Schachfigur[8]
7         ,new Schachfigur[8]
8         ,new Schachfigur[8]
9         ,new Schachfigur[8]
10        ,new Schachfigur[8]
11        };
12
13   public void zug(int vonX,int vonY,int nachX,int nachY){
14     spielFeld[nachX][nachY] = spielFeld[vonX][vonY];
```

[7] primitive Typen sind dabei ausgenommen

```
15      spielFeld[vonX][vonY] = null;
16    }
17  }
```

7.5.8 Blubbersortierung

Wir haben bisher Sortierverfahren für unsere eigenen Listen geschrieben. Dabei haben wir festgestellt, daß die Blubbersortierung für unsere unveränderbaren Listen denkbar ungeeignet ist. Die Blubbersortierung basiert darauf, daß in einer Sammlungsstruktur zwei Elemente vertauscht werden. Für unveränderbare Listen folgt daraus, daß jeweils für einen solchen Vertauschungsschritt eine neue Liste konstruiert werden muß, die gegenüber der Ausgangsliste nur zwei Elemente in vertauschter Reihenfolge haben.

Reihungen eignen sich hingegen hervorragend für die Blubbersortierung. In einer Reihung lassen sich zwei Elemente relativ einfach vertauschen. Keine neue Reihung ist hierzu anzulegen, keine Kopie anzulegen. Die Blubbersortierung ist damit wie folgt umsetzbar:

BubbleSort.java

```
1  class BubbleSort {
2
3    static void bubbleSort(Comparable [] obja){
4      boolean toBubble = true;
5      while (toBubble){
6        toBubble = bubble(obja);
7      }
8    }
9
10   static boolean bubble(Comparable [] obja){
11     boolean result = false;
12     for (int i=0;i<obja.length;i=i+1){
13       try {
14         if (obja[i].compareTo(obja[i+1])>0){
15           Comparable o = obja[i];
16           obja[i]=obja[i+1];
17           obja[i+1]=o;
18           result=true;
19         }
20       }catch (ArrayIndexOutOfBoundsException _){}
21     }
22     return result;
23   }
24
25   static public void main(String [] args){
26     String [] stra = {"a","zs","za","bb","aa","aa","y"};
27     System.out.println(ArrayToString.arrayToString(stra));
28     bubbleSort(stra);
29     System.out.println(ArrayToString.arrayToString(stra));
30   }
31
32 }
```

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

Aufgabe 19 Diese Aufgabe ist die letzte Punkteaufgabe. Für sie gibt es insgesammt vier Punkte. Dieses ist der erste Teil der Aufgabe für den es 2 Punkte gibt.

Im 2. Teil werden graphische Komponenten benutzt. Der zweite Teil der Aufgabe befindet sich im Kapitel über graphische Komponenten.

Abgabe der gesammten Aufgabe ist spätestens am 30. Januar 2003.

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/vier.zip`)

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

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

```
1 public static void main(String[] args) {
2   new VierKonsole().spiel(new VierImplementierung());
3 }
```

7.6 Ein- und Ausgabe

[8] 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 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 Schrift, über kyrillische, griechische, arabische, chinesische bis hin zu exotische Schriften wie die keltische Keilschrift. Dabei ist entscheidend in welcher Codierierung die Buchstaben in der Datenquelle vorliegen. Die Codierung wird in der Regel beim Konstruieren

[8] Dieses Kapitel ist noch recht oberflächlich und flüchtig. Für eine weitere etwas gesetztere Darstellung empfehle ich auch das Kapitel 18 aus dem Skript von Herrn Grude zu konsultieren.

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, der 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`

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 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.

7.6.4 Dateiströme

Zum Schreiben und Lesen in Dateien haben die entsprechenden Stromklassen Konstruktoren, die eine Dateistrom 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 Datenströme aufzurufen, die nicht mehr benutzt werden.

Unter diesen Voraussetzungen läßt sich ein Programm zum Copieren einer Datei wie folgt schreiben:

Copy.java

```
1  import java.io.*;
2
3  public class Copy {
4    public static void main(String[] args) throws IOException {
5      FileReader in = new FileReader("eingabeDatei.txt");
6      FileWriter out = new FileWriter("ausgabedatei.txt");
7      int c;
8
9      while ((c = in.read()) != -1){
10       out.write(c);
11     }
12   }
13 }
```

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.

int vs. char

Wir haben im entsprechenden kurzem Kapitel über primitiven Typen, nicht genau den Zusammenhang zwischen den unterschiedlichen Typen kennengelernt. Im Fall von `Reader` und `Writer` sehen wir jetzt den entscheidenen Unterschied zwischen den primitiven Typen `char` und `int`.

Die Methode `read()` der Klasse `Reader` liest genau einen Zeichen ein, hat aber als Rückgabewert nicht den Typ `char`, der Zeichen darstellt, sondern den Typ `int`. Der Typ `char` kann 65536 Zahlen darstellen, die als Zeichen verschiedenster Schriften interpretiert werden. Der Typ `int` kann

$$2^{32}$$

verschiedene Zahlen darstellen.

Die Methode `read()` der Klasse `Reader` hat den Unicodewert eines Zeichen als Rückgabewert. Soll dieser Wert als Zeichen interpretiert wird, dann ist der entsprechende Wert mit einer Typsicherung auf `char` umzuwandeln. Folgendes Programm veranschaulicht den Unterschied. Eine Datei wird Zeichenweise gelesen. Einmal werden die erhaltenen Zahlen als Zahlen auf den Bildschirm ausgegeben, ein weiteres Mal werden diese Zahlen als Buchstaben interpretiert auf den Bildschirm ausgegeben.

Print.java

```
1  import java.io.*;
2
3  public class Print {
4    public static void main(String[] args) throws IOException {
5      FileReader in = new FileReader("eingabedatei.txt");
6      int c;
7
8      while ((c = in.read()) != -1){
9        System.out.println(c);
```

```
10      }
11      in.close();
12
13      in = new FileReader("eingabedatei.txt");
14      while ((c = in.read()) != -1){
15        System.out.print((char)c);
16      }
17
18    }
19  }
```

7.6.5 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 soll, 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 gepufferte Strom hat einen internen Speicher. Bei einer Datenübertragung werden 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 übertagen. Für lesende Ströme, werden 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:
Folgendes Programm ist eine minimale Umsetzung des aus Unix bekannten Programms `more`. Es ermöglicht den Inhalt einer Datei auf dem Bildschirm auszugeben. Es werden immer 20 Zeilen einer Datei ausgegeben, bevor eine Benutzereingabe erwartet wird. Die Leertaste bedeutet, die nächste Seite soll ausgegeben werden, mit 'q' wird das Programm beendet. Das Programm terminiert, wenn die angezeigte Datei keine weiteren Zeichen enthält.

Die Datei wird wird über einen gepufferten `Reader` eingelesen.

More.java

```
1  import java.io.*;
2
3  public class More {
4    /* the number of lines diplayed at once */
5    static final int LINES_PER_PAGE=20;
```

```
6
7
8     public static void main(String [] args)throws Exception{
9       Reader reader =new BufferedReader(new FileReader(args[0]));
10      getPages(reader);
11    }
12
13    /**
14       Diplays one page on the screen and waits for user input.
15
16      param in is the reader, where data is read from.
17    **/
18    static private void getPages(Reader in)throws Exception{
19      int input=' ';
20      InputStream stdin = System.in;
21      boolean endOfFile= false;
22      while (!endOfFile && input!='q'){
23        if (input==' ') printPage(in);
24        input=stdin.read();
25      }
26    }
27
28    /**
29      Reads a page from the reader and prints it to the screen.
30
31      Terminates the program no more data can be read from the reader.
32      param in is the reader, where data is read from.
33    **/
34    static private void printPage(Reader in)throws Exception{
35      StringBuffer result = new StringBuffer();
36      int newLines = 0;
37      int c = 0;
38      while (c!=-1 && newLines < LINES_PER_PAGE){
39        c=in.read();
40        if (c!=-1) result.append((char)c);
41        if (c=='\n') newLines = newLines+1;
42      }
43
44      System.out.print(result.toString());
45      if (c==-1) System.exit(0);
46    }
47  }
```

7.6.6 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 Dateivormat eines Textdokumentes. Am Anfang der Datei ist eine Tabelle, die angibt, auf welcher Position in der Datei 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.

Kapitel 8

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 einen 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.

8.1 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 Bilschirm 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 Stuerfäden umschaltet.

SimpleThread.java

```
1  class SimpleThread extends Thread {
2
3    public void run(){
4      while (true){
5        System.out.print(getName());
6      }
7    }
8
9    public static void main(String [] args){
10     new SimpleThread().start();
11     new SimpleThread().start();
12   }
13 }
```

8.1.1 Schlafenlegen von Prozessen

In der Klasse `Thread` gibt es eine statische Methode `void sleep(long millis)`. Der Aufruf dieser Methode bewirkt, daß der aktuell laufenden Steuerfaden für entsprechend seines Parameters mindestens einen bestimmten Zeitraum nicht weiterarbeitet. Die Einheit für den Parameter sind tausendstel Sekunden.

Beispiel:
Unseren ersten Steuerfaden aus dem letzten Beispiel erweitern wir hier darum, daß er sich jeweils eine bestimmte Zeit zur Ruhe legt, nachdem er seinen Namen ausgedruckt hat.

SleepingThread.java

```
1  class SleepingThread extends Thread {
2    public Thread(int s){this.s=s;}
3    private int s;
4
5    public void run(){
6      while (true){
7        System.out.print(getName());
8        try {Thread.sleep(s);}catch (InterruptedException _){}
9      }
10   }
```

```
11
12    public static void main(String [] args){
13      new SleepingThread(1000).start();
14      new SleepingThread(300).start();
15    }
16  }
```

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-0Thread-1Thread-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-0Thread-1Thread-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 die Steuerfadenobjekte ausgeführt sondern global und gelten für den geraden aktiven Steuerfaden. Das wird dann jeweils der Steuerfaden sein, in dessen `run`-Methode der Aufruf von `sleep` steht.

8.2 Koordination nebenläufiger Steuerfäden

Wenn zwei Steuerfäden sich die Daten Teilen, dann kann es zu Problemen kommen.

8.2.1 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

```
1  class TwoNumbers{
2     int x;
3     int y;
4
5     TwoNumbers(int x, int y) {this.x=x; this.y=y;}
6
7     void swap(){
8       int z=x;
9       x=y;
10      try {Thread.sleep(100);}catch (InterruptedException _){}
11      y=z;
12    }
13
```

```
14    public String toString(){return "("+x+","+y+")";}
15  }
```

Jetzt können wir zwei Steuerfäden schreiben. Der eine ruft immer wieder die Methode `swap` auf ein `TwoNumbers`-Objekt auf:

Swap.java

```
1   class Swap extends Thread{
2
3     public Swap(TwoNubmers twoN){this.twoN=twoN;}
4
5     private TwoNubmers twoN;
6
7     public void run(){
8       while (true){twoN.swap();}
9     }
10  }
```

Der andere druckt immer wieder ein `TwoNumbers`-Objekt aus:

PrintTwoNumbers.java

```
1   class PrintTwoNumbers extends Thread {
2     public Swap(TwoNumbers twoN){this.twoN=twoN;}
3
4     private TwoNumbers twoN;
5
6     public void run(){
7       while (true){
8         System.out.println(twoN);
9         try {Thread.sleep(1000);}catch (InterruptedException _){}
10      }
11    }
12
13    public static void main(String [] _){
14      TwoNumbers twoN = new TwoNumbers(1,2);
15      new Swap(twoN).start();
16      new PrintTwoNumbers(twoN).start();
17    }
18  }
```

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 mitten in der Audführung 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, daß es eigentlich nicht haben soll, so kann man in Java das Mittel der Synchronisation anweden.

8.2.2 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

```
1  class SafeTwoNumbers{
2    int x;
3    int y;
4
5    SafeTwoNumbers(int x, int y) {this.x=x; this.y=y;}
6
7    synchronized void swap(){
8      int z=x;
9      x=y;
10     try {Thread.sleep(400);}catch (InterruptedException _){}
11     y=z;
12   }
13
14   synchronized public String toString(){
15     return "("+x+","+y+")";
16   }
17 }
```

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 geschweiften 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

```
1  class SafeSwap extends Thread{
2
3    public SafeSwap(TwoNumbers twoN){this.twoN=twoN;}
4
5    private TwoNumbers twoN;
6
7    public void run(){
8      while (true){
9        synchronized(twoN){twoN.swap();}
10     }
11   }
12 }
```

Und wir synchronisieren entsprechend auch den Aufruf, der dafür sorgt, daß das Objekt ausgedruckt wird.

SafePrintTwoNumbers.java

```
1  class SafePrintTwoNumbers extends Thread {
2    public SafePrintTwoNumbers(TwoNumbers twoN){this.twoN=twoN;}
3
```

```
4   private TwoNumbers twoN;
5
6   public void run(){
7     while (true){
8       synchronized (twoN){System.out.println(twoN);}
9       try {Thread.sleep(1000);}catch (InterruptedException _){}
10    }
11  }
12
13  public static void main(String [] _){
14    TwoNumbers twoN = new TwoNumbers(1,2);
15    new SafeSwap(twoN).start();
16    new SafePrintTwoNumbers(twoN).start();
17  }
18 }
```

Auch in dieser Version drucken wir immer nur konsistente Zustände des Objekts `twoN`. Es wird kein temporärer Zwischenzustand der Methode `swap` ausgegeben.

8.2.3 Verklemmungen

Mit der Synchronisation von Operationen über verschiedene Objekten handelt man sich leider ein Problem ein, daß unter den Namen Verklemmung (eng. deadlock)[1] 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 den 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 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 findet sich die etwas drastischere deutsche Bezeichnung *tötliche Umklammerung.*

Das 5 Philosophenproblem

Eine zweite Illustration von Verklemmungen ist das sogenannte 5 Philosophenproblem. 5 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 rein, daß sie den 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 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

```
1  class Deadlock extends Thread{
2
3    Object a;
4    Object b;
5
6    Deadlock (Object a,Object b){this.a=a;this.b=b;}
7
8    public void run(){
9      synchronized (a){
10       System.out.println(
11         getName()+"jetzt habe ich das erste Objekt: "+a);
12       try {Thread.sleep(1000);}catch (InterruptedException _){}
13       synchronized (b){
14           System.out.println(getName()+"jetzt habe ich beide");
15       }
16     }
17   }
18
19   public static void main(String [] _){
20     String s1 = "hallo";
```

```
21      String s2 = "welt";
22      new Deadlock(s1,s2).start();
23      new Deadlock(s2,s1).start();
24    }
25  }
```

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 den jeweils das andere Objekt anfordern, müssen sie darauf warten. 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
```

8.2.4 Warten und Benachrichtigen

Über die Synchronisation haben wir sichergestellt, daß bestimmte Operationen bur 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 ausgelsen wurde, damit er nicht überschrieben wird.

Ohne Vorkehrungen dafür, daß abwechselnd gelesen und geschrieben wird, sieht die Klasse wie folgt aus:

GetSet.java

```
1  class GetSet {
2    private int i;
3    GetSet(int i){this.i=i;}
4
5    synchronized int get(){return i;}
6    synchronized void set(int i){this.i=i;}
7  }
```

Unsere beiden Steuerfäden, die auf einem solchen Objekt lesen und schreiben können wir schreiben als:

Set.java

```
class Set extends Thread{
  GetSet o;
  Set(GetSet 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

```
class Get extends Thread{
  GetSet o;
  Get(GetSet 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 [] _){
    GetSet gss = new GetSet(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:

GetSet.java

```
class GetSet {
  private int i;
  private boolean available= true;
  GetSet(int i){this.i=i;}

  synchronized int get(){
    while (!available){
      try{Thread.sleep(1000);}catch (Exception _){}
    }
    available=false;
    return i;
  }

```

```
14    synchronized void set(int i){
15      while (available){
16        try{Thread.sleep(1000);}catch (Exception _){}
17      }
18      this.i=i;
19      available=true;
20    }
21  }
```

Starten wir damit unser Testprogramm, so kommt es zu einer Verklemmung. Um dieses zu umgehen, hat Java die 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:

GetSet.java

```
1   class GetSet {
2     private int i;
3     private boolean available= true;
4     GetSet(int i){this.i=i;}
5
6     synchronized int get(){
7       while (!available){
8         try{wait();}catch (Exception _){}
9       }
10      available=false;
11      notifyAll();
12
13      return i;
14    }
15
16    synchronized void set(int i){
17      while (available){
18        try{wait();}catch (Exception _){}
19      }
20      this.i=i;
21      available=true;
22      notifyAll();
23    }
24  }
```

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 9

Graphische Benutzeroberflächen

Die meisten Programme auf heutigen Rechnersystemen haben eine graphische Benutzeroberfläche (GUI)[1].

Java stellt Klassen zur Verfügung, mit denen graphische Objekte erzeugt und in ihrem Verhalten instrumentalisiert werden können. Leider gibt es historisch gewachsen zwei Pakete in Java mit Klassen zur GUI-Programmierung. Die beiden Pakete überschneiden sich in der Funktionalität.

- `java.awt`: Dieses ist das ältere Paket zur GUI-Programmierung. Es enthält Klassen für viele graphische Objekte (z.B. eine Klasse `Button`) und Unterpakete, zur Programmierung der Funktionalität der einzelnen Komponenten.
- `javax.swing`: Dieses neuere Paket ist noch universeller und platformunabhängiger als das `java.awta`-Paket. Auch hier finden sich Klassen für unterschiedliche GUI-Komponenten. Sie entsprechen den Klassen aus dem Paket `java.awt`. Die Klassen haben oft den gleichen Klassennamen wie in `java.awt` jedoch mit einem `J` vorangestellt. So gib es z.B. eine Klasse `JButton`.

Zu allen Unglück sind die beiden Pakete nicht unabhängig. Man ist zwar angehalten, sofern man sich für eine Implementierung seines Guis mit den Paket `javax.swing` entschieden hat, nur die graphischen Komponenten aus diesem Paket zu benutzen; die Klassen leiten aber von Klassen des Pakets `java.awt` ab. Hinzu kommt, daß die Ereignissklassen, die die Funktionalität graphischer Objekte bestimmen, nur in `java.awt` existieren und nicht nocheinmal für `javax.swing` extra umgesetzt wurden.

Sind Sie verwirrt. Sie werden es hoffentlich nicht mehr sein, nachdem Sie die Beispiele dieses Kapitels durchgespielt haben.

Zur Programmierung eines GUIs bedarf es graphischer Objekte, eine Möglichkeit, graphische Objekte zu gruppieren, und schließlich eine Möglichkeit zu bestimmen, wie die Objekte auf bestimmte Ereignisse reagieren (z.B. was passiert, wenn ein Knopf gedrückt wird). Java kennt für die Aufgaben:

[1] GUI ist die Abkürzung für *graphical user interface*. Entsprechend wäre GRABO eine Abkürzung für das deutsche *graphische Benutzeroberfläche*. Im Skript von Herrn Grude wird dieser Ausdruck für GUI verwendet.

- Komponenten
- Layout Manager
- Event Handler

In den folgenden Abschnitten, werden wir an ausgewählten Beispielen Klassen für diese drei wichtigen Schritte der GUI-Programmierung kennenlernen.

9.1 Gui Komponenten

Javas `swing`-Paket kennt drei Arten von Komponenten.

- Top-Level Komponenten
- Zwischenkomponenten
- Atomare Komponenten

Leider spiegelt sich diese Unterscheidung nicht in der Ableitungshierarchie wieder. Alle Komponenten leiten schließlich von der Klasse `java.awt.Component` ab. Es gibt keine Schnittmengen, die Beschreiben, daß bestimmte Komponenten atomar oder top-level sind.

Komponenten können Unterkomponenten enthalten; ein Fenster kann z.B. verschiedene Knöpfe und Textflächen als Unterkomponenten enthalten.

9.1.1 Top-Level Komponenten

Eine top-level Komponenten ist ein GUI-Objekt, dasß weitere graphische Objekte enthalten kann, selbst aber kein graphisches Objekt hat, in dem es enthalten ist. Somit sind top-level Komponenten in der Regel Fenster, die weitere Komponenten als Fensterinhalt haben. Swing top-level Komponenten sind Fenster und Dialogfenster. Hierfür stellt Swing entsprechende Klassen zur Verfügung: `JFrame, JDialog`

Eine weitere top-level Komponente steht für *Applets* zur Verfügung: `JApplet`.

Graphische Komponenten haben Konstruktoren, mit denen sie erzeugt werden. Für die Klasse `JFrame` existiert ein parameterloser Konstruktor.

Beispiel:
Das minimalste GUI-Programm ist wahrscheinlich folgendes Programm, das ein Fensterobjekt erzeugt und dieses sichtbar macht.

JF.java

```
1 import javax.swing.*;
2
3 class JF {
4   public static void main(String [] _){
5     new JFrame().setVisible(true);
6   }
7 }
```

Dieses Programm erzeugt ein leeres Fenster und gibt das auf dem Bildschirm aus.

Die Klasse `JFrame` hat einen zweiten Konstruktor, der noch ein Stringargument hat. Der übergebene String wird als Fenstertiteltext benutzt.

Beispiel:
Ein Programm, kann auch mehrere Fensterobjekte erzeugen und sichtbar machen.

JF2.java

```
1  import javax.swing.*;
2
3  class JF2 {
4    public static void main(String [] _){
5      new JFrame("erster Rahmen").setVisible(true);
6      new JFrame("zweiter Rahmen").setVisible(true);
7    }
8  }
```

Dieses Programm öffnet zwei leere Fenster mit unterschiedlichen Titeln.

9.1.2 Zwischenkomponenten

Graphische Zwischenkomponenten haben primär die Aufgabe andere Komponenten als Unterkomponenten zu haben und diese in einer bestimmten Weise anzuordnen. Zwischenkomponenten haben oft keine eigene visuelle Ausprägung. Sie sind dann unsichtbare Komponenten, die als Behälter weiterer Komponenten dienen.

Die gebräuchlichste Zwischenkomponenten ist von der Klasse `JPanel`. Weitere Zwischenkomponenten sind`JScrollPane` und `JTabbedPane`. Diese haben auch eine eigene visuelle Ausprägung.

Die Objekte der Klasse `JFrame` enthalten bereits eine Zwischenkomponente. Diese kann mit der Methode `getContentPane` erhalten werden, um dieser Zwischenkomponenten schließlich weitere Unterkomponenten, den Inhalt des Fensters, hinzufügen zu können.

9.1.3 Atomare Komponenten

Die atomaren Komponenten sind schließlich Komponenten, die ein konkretes graphisches Objekt darstellen, das keine weiteren Unterkomponenten enthalten kann. Solche Komponenten sind z.B.
`JButton, JTextField, JTable` und `JComBox`.

Diese Komponenten lassen sich über ihre Konstruktoren instanziieren, um sie dann einer Zwischenkomponenten über deren Methode `add` als Unterkomponente hinzuzufügen. Sind alle gewünschten graphischen Objekte einer Zwischenkomponente jinzugefügt worden, so kann auf der zugehörigen top-level Komponenten die Methode `pack` aufgerufen wurden. Diese berechnet die notwendige Größe und das Layout des Fensters, welches schließlich sichtbar gemacht wird.

Beispiel:
Das folgende Programm erzeugt ein Fenster mit einer Textfläche.

JT.java

```
1  import javax.swing.*;
2
3  class JT {
4    public static void main(String [] _){
5      JFrame frame = new JFrame();
6      JTextArea textArea = new JTextArea();
7      textArea.setText("hallo da draußen");
8      frame.getContentPane().add(textArea);
9      frame.pack();
10     frame.setVisible(true);
11   }
12 }
```

Das Programm erzeugt das folgende Fenster:

In der Regel wird man nicht die Fenster des Programms nacheinander in der Hauptmethode definieren, sondern die gebrauchten Fenster werden als eigene Objekte definiert. Hierzu leitet man eine spezifische Fensterklasse von der Klasse `JFrame` ab und fügt bereits im Konstruktor die entsprechenden Unterkomponenten hinzu.

Beispiel:
Die folgende Klasse definiert ein Fenster, das einen Knopf und eine Textfläche enthält. In der Hauptmethode wird das Objekt instanziiert:

JTB.java

```
1  import javax.swing.*;
2
3  class JTB extends JFrame {
4    public JTB(){
```

```
5       JTextArea textArea = new JTextArea();
6       textArea.setText("hallo da draußen");
7       JPanel pane = new JPanel();
8       pane.add(textArea);
9       pane.add(new JButton("ein knopf"));
10      getContentPane().add(pane);
11      pack();
12      setVisible(true);
13    }
14
15    public static void main(String [] _){
16      new JTB();
17    }
18  }
```

Das Programm erzeugt folgendes Fenster:

9.2 Gruppierung

Bisher haben wir Unterkomponenten weitere Komponenten mit der Methode `add` hinhzugefügt, ohne uns Gedanken über die Plazierung der Komponenten zu machen. Wir haben einfach auf das Standardverhalten zur Plazierung von Komponenten vertraut.

Um das Layout von graphischen Komponenten zu steuern, steht das Konzept der sogenannten *Layout-Manager* zur Verfügung. Ein Layout-Manager ist ein Objekt, das einer Komponente hinzugefügt wird. Der Layout-Manager steuert dann, in welcher Weise die Unterkomponenten gruppiert werden.

`LayoutManager` ist eine Schnittstelle. Es gibt mehrere Implementierungen dieser Schnittstelle. Wir werden in den nächsten Abschnitten drei davon kennenlernen. Es steht einem natürlich frei, eigene Layout-Manager durch Implementierung dieser Schnittstelle zu schreiben. Es wird aber davon abgeraten, weil dieses notorisch schwierig ist und die in Java bereits vorhandenen Layout-Manager bereits sehr mächtig und ausdrucksstark sind.

Zum Hinzufügen eines Layout-Manager gibt es die Methode `setLayout`.

9.2.1 Flow Layout

Der vielleicht einfachste Layout-Manager nennt sich `FlowLayout`. Hier werden die Unterkomponenten einfach der Reihe nach in einer Zeile angeordnet. Erst wenn das Fenster zu schmal hierzu ist, werden weitere Komponenten in eine neue Zeile gruppiert.

Beispiel:
Die folgende Klasse definiert ein Fenster, dessen Layout über ein Objekt der Klasse `FlowLayout` gesteuert wird. Dem Fenster werden fünf Knöpfe hinzugefügt:

FlowLayoutTest.java

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  class FlowLayoutTest extends JFrame {
5
6    public FlowLayoutTest(){
7      Container pane = getContentPane();
8      pane.setLayout(new FlowLayout());
9      pane.add(new JButton("eins"));
10     pane.add(new JButton("zwei"));
11     pane.add(new JButton("drei (ein langer Knopf)"));
12     pane.add(new JButton("vier"));
13     pane.add(new JButton("fuenf"));
14     pack();
15     setVisible(true);
16   }
17
18   public static void main(String [] _){
19     new FlowLayoutTest();
20   }
21 }
```

Das Fenster hat folgende optische Ausprägung.

Verändert man mit der Maus die Fenstergröße, macht es z.B. schmal und hoch, so werden die Knöpfe nicht mehr nebeneinander sonder übereinander angeordnet.

9.2.2 Border Layout

Die Klasse `BorderLayout` definiert einen Layout Manager, der fünf feste Positionen kennt: eine Zentralposition, und jeweils links/rechts und oberhalb/unterhalb der Zentralposition eine Position für Unterkomponenten. Die Methode `add` kann in diesem Layout auch noch mit einem zweitem Argument aufgerufen werden, das eine dieser fünf Positionen angibt. Hierzu bedient man sich der konstanten Felder der Klasse `BorderLayout`.

Beispiel:
In dieser Klasse wird die Klasse `BorderLayout` zur Steuerung des Layoout benutzt. Die fünf Knöpfe werden an jeweils eine der fünf Positionen hinzugefügt:

BorderLayoutTest.java

```
1  import java.awt.*;
2  import javax.swing.*;
```

```
3
4  class BorderLayoutTest extends JFrame {
5
6    public BorderLayoutTest(){
7      Container pane = getContentPane();
8      pane.setLayout(new BorderLayout());
9      pane.add(new JButton("eins"),BorderLayout.NORTH);
10     pane.add(new JButton("zwei"),BorderLayout.SOUTH);
11     pane.add(new JButton("drei (ein langer Knopf)")
12                                  ,BorderLayout.CENTER);
13     pane.add(new JButton("vier"),BorderLayout.WEST);
14     pane.add(new JButton("fuenf"),BorderLayout.EAST);
15     pack();
16     setVisible(true);
17   }
18
19   public static void main(String [] _){
20     new BorderLayoutTest();
21   }
22 }
```

Die Klasse erzeugt das folgende Fenster.

Das Layout ändert sich nicht, wenn man mit der Maus die Größe und das Format des Fensters verändert.

9.2.3 Grid Layout

Die Klasse `GridLayout` ordnet die Unterkomponenten tabellarisch an. Jede Komponente wird dabei gleich groß ausgerichtet. Die Größe richtet sich also nach dem größten Element.

Beispiel:
Folgende Klasse benutzt ein Grid-Layout mit zwei Zeilen zu je drei Spalten.

GridLayoutTest.java

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  class GridLayoutTest extends JFrame {
5
6    public GridLayoutTest(){
7      Container pane = getContentPane();
```

```
8       pane.setLayout(new GridLayout(2,3));
9       pane.add(new JButton("eins"));
10      pane.add(new JButton("zwei"));
11      pane.add(new JButton("drei (ein langer Knopf)"));
12      pane.add(new JButton("vier"));
13      pane.add(new JButton("fünf"));
14      pack();
15      setVisible(true);
16    }
17
18    public static void main(String [] _){
19      new GridLayoutTest();
20    }
21  }
```

Folgendes Fensters wird durch dieses Programm geöffnet.

Auch hier ändert sich das Layout nicht, wenn man mit der Maus die Größe und das Format des Fensters verändert.

9.3 Ereignissbehandlung

Um den graphischen Komponenten eine Funktionalität hinzuzufügen, kennt Java das Konzept der Ereignisbehandlung. Graphische Objekte sollen in der Regel auf bestimmte Ereignisse auf eine definierte Weise reagieren. Solche Ereignisse könne Mausbewegungen, Mausklicks, Ereignisse an einem Fenster, wie das Schließen des Fensters oder etwa Eingaben auf der Tastatur sein. Für die verschiedenen Arten von Ereignissen sind im Paket `java.awt.event` Schnittstellen definiert. In diesen Schnittstellen stehen Methoden, in denen die Reaktion auf bestimmte Ereignisse definiert werden kann. So gibt es z.B. eine Schnittstelle `Mouselistener`, in der Methoden für verschiedene Ereignisse auf den Mausknöpfen bereitstehen.

Soll einer bestimmten graphischen Komponente eine bestimmte Reaktion auf bestimmte Ereignisse zugefügt werden, so ist die entsprechende Schnittstelle mit Methoden für das anvisierte Ereignis ausgekuckt und implementiert werden. Ein Objekt dieser Implementierung kann dann der graphischen Komponente mit einer entsprechenden Methode hinzugefügt werden.

9.3.1 Action Listener

Das allgemeinste Ereignis ist ein `ActioEvent`. Die entsprechende Schnittstelle `ActionListener` enthält nur eine Methode, die auszuführen ist, wenn eine Aktion aufgetreten ist:

Beispiel:
Wir implementieren die Schnittstelle `ActionListener` so, daß in einem internen Zähler vermerkt wird, wie oft ein Ereignis aufgetreten ist. Bei jedem Auftreten des Ereignisses wird die entsprechende Zahl auf einer Textfläche gesetzt:

CountActionListener.java

```
1  import java.awt.event.*;
2  import javax.swing.text.*;
3
4  class CountActionListener implements ActionListener {
5    JTextComponent textArea;
6    int count;
7
8    CountActionListener(JTextComponent textArea){
9      this.textArea=textArea;
10   }
11
12   public void actionPerformed(ActionEvent e) {
13     count=count+1;
14     textArea.setText(""+count);
15   }
16 }
```

In einer zweiten Klasse definieren wir eine Fensterkomponente mit zwei atomaren Komponenten: einen Knopf und eine Textfläche. Dem Knopf fügen wir die oben geschriebene Ereignisbehandlung hinzu.

Count.java

```
1  import java.awt.*;
2  import javax.swing.*;
3  import javax.swing.text.*;
4
5  class Count extends JFrame {
6
7    public Count(){
8
9      JTextComponent textArea = new JTextField(8);
10     JButton button = new JButton("click");
11     JPanel pane = new JPanel();
12     pane.add(button);
13     pane.add(textArea);
14
15     getContentPane().add(pane);
16     button.addMouseListener(new CountActionListerner(textArea));
17     pack();
18     setVisible(true);
19   }
20
21   public static void main(String [] _){
22     new Count();
23   }
```

```
24 }
```

Wir erhalten ein Fenster, in dem die Anzahl der Mausklicks auf dem Knopf im Textfeld angezeigt wird.

9.3.2 Mausereignisse

Eine spezifischere Ereignisbehandlung läßt sich über die Schnittstelle `MouseListener` definieren. Hier befinden sich mehrere Methoden für verschiedene Ereignisse auf den Mausknöpfen. Oft will man nur auf eines dieser Ereignisse reagieren. Um nicht alle Methoden der Schnittstelle mit einem leeren Methodenrumpf zu implementieren, stellt Java mit der Klasse `MouseAdapter` eine Implementierung der Schnittstelle `MouseListener` bereit. Diese Implementierung hat keine Reaktion auf die Ereignisse. Leitet man von dieser Klasse ab und überschreibt beliebige Methoden, so läßt sich am schnellsten eine eigene Implementierung der Schnittstelle erhalten.

Beispiel:
Wir schreiben das Programm des vorherigen Beispiels jetzt mit Hilfe einer Implementierung von `MouseListener`. Hierzu leiten wir unsere Klasse von der Klasse `MouseAdapter` ab und überschreiben die Methode `mouseClicked`.

CountButtonListener.java

```
1  import java.awt.event.*;
2  import javax.swing.text.*;
3
4  class CountButtonListener extends MouseAdapter {
5    JTextComponent textArea;
6    int count;
7
8    CountButtonListener(JTextComponent textArea){
9      this.textArea=textArea;
10   }
11
12   public void mouseClicked(MouseEvent e) {
13     count=count+1;
14     textArea.setText(""+count);
15   }
16 }
```

Jetzt fügen wir ein Objekt dieser Klasse den Knopf unseres GUIs hinzu. Dazu benutzen wir die Methode `addMouseListener`.

```
1  import java.awt.*;
2  import javax.swing.*;
3  import javax.swing.text.*;
4
5  class Count extends JFrame {
6
7    public Count(){
8
```

```
9       JTextComponent textArea = new JTextField(8);
10      JButton button = new JButton("click");
11      JPanel pane = new JPanel();
12      pane.add(button);
13      pane.add(textArea);
14
15      getContentPane().add(pane);
16      button.addMouseListener(new CountButtonListener(textArea));
17      pack();
18    }
19
20    public static void main(String [] _){
21      new Count().setVisible(true);
22    }
23  }
```

9.3.3 Fensterereignisse

Eine weitere wichtige Klasse von Ereignissen sind Fensterereignisse. Hierzu gibt es die Schnittstelle `WindowListener` und auch schon eine leere Implementierung dieser Schnittstelle in der Klasse `WindowAdapter`. Um eine Ereignissbehandlung für Fenster einem Fenster hinzuzufügen, steht die Methode `addWindowListener` zur Verfügung.

Beispiel:
Wir schreiben eine ganz kleine Ereignissbehandlungsklasse für Fensterereignisse, die bewirkt, daß das komplette Programm beendet wird, wenn das Fesnter geschlossen wird.

CountWindowListener.java

```
1  import java.awt.event.*;
2  class CountWindowListener extends WindowAdapter {
3      public void windowClosing(WindowEvent e) {System.exit(0);}
4  }
```

Diese Ereignissbehandlung können wir dem Fenster aus dem letztem Beispiel hinzufügen:

Close.java

```
1  import javax.swing.*;
2  class Close {
3    public static void main(String [] _){
4      JFrame frame = new Count();
5      frame.addWindowListener(new CountWindowListener());
6      frame.setVisible(true);
7    }
8  }
```

9.4 Innere und anonyme Klassen

Bei der Programmierung eine graphischen Oberfläche, werden viele Klasse geschrieben, die nur ein einziges Mal instanziiert werden. Die Ereignissbehandlung für einen Knopf ist oft so spezifisch, daß für die entsprechende Klasse genau ein Objekt erzeugt wird. Java erlaubt in solchen Situationen, für solche Klassen nicht eine eigene Datei zu schreiben, sondern Klassen innerhalb von anderen Klassen zu definieren.

Auf weitere Implikation der inneren Klassen werden wir in diesem Kapitel nicht eingahen, sondern nur zwei Beispiele aus der GUI-Programmierung zeigen.

9.4.1 Innere Klassen

Bei innerern Klassen steht eine weitere Klasse innerhalb einer Klasse. Beim Übersetzen der äußeren Klasse werden zwei `.class` Dateien erzeugt. So läßt sich innerhalb der Definition eines Fensters direkt die Klasse zur Definition seiner Ereignissbehandlung schreiben.

Beispiel:
Das Programm aus dem letzten Beispiel läßt sich in einer Datei schreiben, indem der `WindowListener` als innere Klasse definiert wird.

Count2.java

```
1  import java.awt.*;
2  import javax.swing.*;
3  import javax.swing.text.*;
4  import java.awt.event.*;
5
6  class Count2 extends JFrame {
7
8    //innere Klasse
9    class CountWindowListener extends WindowAdapter {
10     public void windowClosing(WindowEvent e) {System.exit(0);}
11   }
12
13   public Count2(){
14     JTextComponent textArea = new JTextField(8);
15     JButton button = new JButton("click");
16     JPanel pane = new JPanel();
17     pane.add(button);
18     pane.add(textArea);
19
20     getContentPane().add(pane);
21     button.addActionListener(new CountActionListener(textArea));
22     addWindowListener(new CountWindowListener());
23     pack();
24     setVisible(true);
25   }
26
27   public static void main(String [] _){
28     new Count2();
```

```
29    }
30  }
```

9.4.2 Anonyme Klassen

Java erlaubt es, sogar noch einen Schritt weiter zu gehen. Man kann für Klassen, für die nur ein Objekt erzeugt wird, sogar auf einen Namen für diese Klasse verzichten. Hierzu wird mit `new` ein Konstruktoraufruf der gewünschten Oberklasse gemacht, dem in geschweiften Klammern ein Klassenrumpf folgt, in dem Methoden überschrieben werden können.

Der folgende Ausdruck erzeugt ein Objekt einer Unterklasse der Klasse `WindowAdapter`. Diese Unterklasse hat keinen Namen und überschreibt die Methode `windowClosing`.

```
1  new WindowAdapter(){
2     public void windowClosing(WindowEvent e) {System.exit(0);}
3  });
```

Auch für anonyme Klasse werden eigene `.class`-Dateien erzeugt.

Beispiel:
Schreiben wir unser letztes Beispiel mit einer anonymen Klasse zur Fensterereignisbehandlung, so erhalten wir folgende Klasse.

Count3.java

```
1  import java.awt.*;
2  import javax.swing.*;
3  import javax.swing.text.*;
4  import java.awt.event.*;
5
6  class Count3 extends JFrame {
7
8    public Count3(){
9      JTextComponent textArea = new JTextField(8);
10     JButton button = new JButton("click");
11     JPanel pane = new JPanel();
12     pane.add(button);
13     pane.add(textArea);
14
15     getContentPane().add(pane);
16     button.addActionListener(new CountActionListener(textArea));
17     addWindowListener(
18       new WindowAdapter(){
19         public void windowClosing(WindowEvent e) {System.exit(0);}
20       });
21     pack();
22     setVisible(true);
23   }
24
25   public static void main(String [] _){
```

```
26      new Count3();
27    }
28  }
```

Aufgabe 20 Dieses ist der zweite Teil der Aufgabe zum Spiel *Vier Gewinnt*. Es gibt noch einmal 2 Punkte für diese Aufgabe. Abgabe ist spätestens in der Übung am 30.01.2002.

Laden sie sich für diese Aufgabe die Datei
vier2.zip (`http://www.tfh-berlin.de/~panitz/prog1/vier2.zip`)

a) Der Schnittstelle `VierLogik` ist eine weitere Methode zugefügt worden: `newGame()`. Implementieren Sie diese in Ihrer Klasse `VierImplementierung`

b) Schreiben Sie die folgende Hauptmethode und starten Sie diese. Sie sollten jetzt ein GUI bekommen, auf dem Sie Vier-Gewinnt spielen können.

```
1  public static void main(String [] _){
2    new Spiel(new VierImplementierung());
3  }
```

c) Schreiben Sie eine Klasse `SpielWindowListener`, die von der Klasse `WindowAdapter` ableitet. Überschreiben Sie die Methode `windowClosing`, so daß sie die Methode `System.exit` ausführt.

d) Sorgen Sie dafür, daß im Konstruktor der Klasse `Spiel` eine Instanz Ihrer Klasse `SpielWindowListener` als *WindowListener* zugefügt wird.

Kapitel 10

Schlußkapitel

10.1 Überbleibsel

Im Laufe der Vorlesung mußten einige kleinere Spezialkonstrukte, um den Gesamtfluß der Vorlesung nicht zu sehr zu unterbrechen, zurückgestellt werden. Andere wurden auch schlichtweg vergessen. In diesen Abschnitt wird kurz auf einige dieser ausgelassenen kleinen Themen eingegangen.

10.1.1 die switch Anweisung

Einen zusammengesetzte Anweisung haben wir im Laufe der Vorlesung ausgelassen. Es ist eine Anweisung für eine Fallunterscheidung mit mehreren Fällen, die `switch`-Anweisung. Die Idee dieser Anweisung ist, eine Kette von mehreren `if-then`-Anweisungen zu vermeiden. Leider ist die `switch`-Anweisung in seiner Anwendungsbereite recht begrenzt und in Form und Semantik ziemlich veraltet.

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

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

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

Programms feststehen. Der Konstante folgt ein Doppelpunkt, dem dann die Anweisungen für diesen Fall folgen.

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

Beispiel:
Ein kleines Beispiel soll die operationale Semantik dieser Anweisung verdeutlichen.

Switch.java

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

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

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

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

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

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

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

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

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

10.1.2 Das final Attribut

Ein weiteres Attribut für Klassen und Eigenschaften haben wir bisher unerwähnt gelassen, das Attribut `final`. Von einer Klasse die das Attribut `final` hat, darf keine Unterklasse abgelitten werden. Ein prominentes Beispiel für eine Klasse mit Attribut `final` ist die Klasse `String`.

Einem Feld mit dem Attribut `final`, darf nur ein initialer Wert zugewiesen werden. Auf dieses Feld darf keine weitere Zuweisung gemacht werden.

`final` Felder werden benutzt um für Konstanten feste Variablen einzuführen. Eine Konvention ist, solche statischen konstanten Felder Namen bestehend nur aus Großbuchstaben zu benutzen:

```
static String THE_ANSWER = "forty two";
```

10.1.3 String Konstanten

String-Objekte können erzeugt werden, indem ein Text in Anführungszeichen eingeschlossen wird. Es lassen sich auf diese Weise aber nicht alle Zeichen darstellen, insbesondere z.B. nicht das Anführungszeichen selbst, denn damit wird die String Konstante ja beendet. Um solche Zeichen in einer Konstante zu schreiben, wird ihr der schräge Rückstrich \ vorangestellt. Die Sequenz `\"` in einer String Konstante symbolisiert das Zeichen". Entsprechend muß es jetzt eine weitere Sequenz geben, um den Rückstrich zu beschreiben. Dieses wird durch die Sequenz `\\` bewerkstelligt. Weitere wichtige solche Sequnzen sind: `\n` für einen Zeilenumbruch[1] und `\t` steht für einen Tabulatoreinschub.

[1] Dieses gilt für unixartige Betriebssysteme. Traditionell gibt es zwei Zeichen in der ASCII-Tabelle für den Zeilenumbruch. Das eine wird von Unix benutzt, das zweite in MAC-OS und Windows benutzt beide. Die meiste Anwenungsprogramme kommen heute zum Glück mit allen drei Varianten zurecht, so daß sich Dokumente zwischen den drei Bertriebssystemfamilien austauschen lassen.

Darüberhinaus kann man mit Hilfe des Unicode-Werts eines beliebigen Zeichens aus der Unicode-Tabelle über die Sequenz \u gefolgt von den Unicodewert des gewünschten Zeichens in Hexadezimaldarstellung.

Beispiel:
Im folgenden Programm werden ein Paar Strings mit besonderen Stringsequenzen auf dem Bildschirm ausgegeben.

StringConstants.java

```
1  class StringConstants {
2    static public void main(String [] _){
3      System.out.println("a\tb");
4      System.out.println("a\nb");
5      System.out.println("a\rb");
6      System.out.println("a\\b");
7      System.out.println("a\"b\"c");
8      System.out.println("a\u004Ab");
9      System.out.println("a\u004Bb");
10     System.out.println("a\u004Cb");
11   }
12 }
```

Das Programm hat folgende Ausgabe:

```
sep@swe10:~/fh/internal/beispiele> java StringConstants
a       b
a
b
b
a\b
a"b"c
aJb
aKb
aLb
sep@swe10:~/fh/internal/beispiele>
```

10.1.4 Operatoren

Wer die Tabelle der Operatoren ansieht, stellt fest, daß wir bisher viele außer acht gelassen haben. Sie seien hier kurz erwähnt.

Zuweisungsoperatoren

In der Javagrammatik gibt es eine Regel, die beschreibt, welche Zuweisungsoperatoren es gibt.

AssignmentOperator ::=

```
 =
|+=
|-=
|*=
|/=
|&=
||=
|^=
|%=
|<<=
|>>=
|>>>=
```

Wir kannten bisher nur den ersten Fall, das einfache Gleichheitszeichen `=` für die Zuweisung. Java kennt bei der Zuweisung zusätzlich noch Operatoren, die einen klassischen Infixoperator mit der Zuweisung verbinden, so z.B. die Zuweisung `+=`. Diese Zuweisungen sind eine Abkürzung. Der Ausdruck `result+=5;` ist eine Abkürzung für `result=5+result;`.

Zusätzlich gibt es noch die Operatoren `++` und`--`. Diese erhöhen bzw. verringern den Wert eines Feldes um eins.

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*. Der Unterschied liegt in der Art, wie das Ergebnis dieser bool'schen Operation berechnet wird. Die doppelte Version schaut sich erst nur den ersten (den linken) Operanden an und entscheidet anhand dessen Werts, ob er das Ergebnis schon kennt, oder sich auch den zweiten Operanden anschauen muß. Ist der Operator z.B. das logische *und* und der erste Operand wertet zu `false` aus, so brauch der zweite Operand nicht mehr angeschaut zu werden, denn egal, was er für einen Wert hat, das Ergebnis de logischen *und* wird auch `false` sein.

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 sich immer erst die Operanden einer Operation (entsprechend die Parameter eines Methodenaufrufs) alle komplett anszuschauen und auszuwerten, nennt man *strikt*. Die gegenteilige Strategie, nur das notwendigste von den Operanden auszuwerten entsprechend, nicht-strikt.[2]

> **Beispiel:**
> Mit einer Methode, die für einen bool'schen Wert diesen auf dem Bildschirm audruckt, um ihn dann wieder als Ergebnis auszugeben, läßt sich gut beobachten, wie die zwei unterschiedlichen Gruppen der logischen Operatoren funktionieren.

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

BoolOps.java

```
1  class BoolOps {
2    static boolean traceBool(boolean b){
3      System.out.println(b);
4      return b;
5    }
6
7    public static void main(String [] _){
8      boolean b;
9      System.out.println("test 1");
10     b=traceBool(true)||traceBool(false);
11     System.out.println("test 2");
12     b=traceBool(true)|traceBool(false);
13     System.out.println("test 3");
14     b=traceBool(false)&&traceBool(false);
15     System.out.println("test 4");
16     b=traceBool(false)&traceBool(true);
17   }
18 }
```

Die Tests für die strikten logischen Operatoren erzeugen die Ausgabe immer von beiden Operanden auf den Bildschirm. Die nicht-strikte Auswertung wertet hier nur den ersten Operanden aus.

```
sep@swe10:~/fh/internal/beispiele> java BoolOps
test 1
true
test 2
true
false
test 3
false
test 4
false
true
sep@swe10:~/fh/internal/beispiele>
```

10.2 Ausblick

Viele Aspekte von Java, viele Programmieraufgaben und viele Programmiertechniken konnten naturgemäß in dieser Vorlesung noch nicht behandelt werden. Es folgt eine unsystematische und unvollständige Liste von Themen, die in einer Nachfolgevorlesung behandelt werden könnten.

- **innere und anonyme Klassen**: was ist zu beachten. Inweiweit kann auf die äußere Klasse zugegriffen werden.
- **generische Typen**: was kommt in Java 1.5.
- **Reflektion**: Objekte können nach den in ihnen vorhandenen Methoden gefragt werden.

- **Serialisierung**: Wie kann ein Objekt über einen Datenstrom geschickt werden.
- **JNI**: Wie wird auf Programme, die in anderen Sprachen geschrieben wurden, zugegriffen, bzw. wie lassen sich Javaprogramme aus anderen Programmiersprachen ausführen.
- **RMI**: der Zugriff auf Methoden über ein Netzwerk.
- **JDBC**: Datenbankverbindung mit Java
- **ClassLoader**: Wie lädt Java Klassen und inwieweit läßt sich das Steuern.
- **Yacc**: wie läßt sich ein in Java geschriebener Parser für eine Grammatik generieren.
- **Internationalisierung**: Wie werden unterschiedliche Codierungen in Java benutzt.
- **Verträge**: Vor- und Nachbedingungen für Methoden.
- **Bäume**: und Algorithmen auf Bäumen.
- **DOM und SAX**: Programmieren von XML-Dokumenten.

10.3 Persönliches Schlußwort

Diese Version des Skripts ist begleitend zu meiner ersten eigenverantwortlichen Lehrveranstaltung[3] entstanden. Vorlesung und Skript sind *from scratch* ohne Vorbereitung oder aufgrund vorhandener Unterlagen entstanden. Es soll nicht verschwiegen sein, daß die Skripte meiner Kollegen an der TFH Berlin gute Leitfäden bei der Entwicklung meiner Vorlesung waren.

Eine Wiederholung dieser Vorlesung wird sicherlich zu drastischen Revisionen in Aufbau und Inhalt von Skript und Vorlesung führen und über die Korrektur der vielen Flüchtigkeitsfehler hinausgehen.

Leider war im laufenden Betrieb des Semesters nicht die Zeit die Kapitel noch einmal gründlich zu überlesen und sprachlich zu korrigieren.

Ich danke meinen Studenten, deren Wißbegier und Lerneifer auch mich immer wieder motiviert haben, Zeit und Energie in diese Vorlesung zu stecken und Vorlesungs- und Übungsstudenten zu Veranstaltungen gamacht haben, zu denen ich gerne angetreten bin. Ich freue mich entsprechend auf die Nachfolgevorlesung *Programmieren II* im nächsten Semester.

[3] Und bei der Vorstellung, daß ich irgendwann in 30 Jahren die letzte Vorlesung halten werden, wird mir etwas komisch.

Anhang A

Beispielaufgaben

Aufgabe 1 Führen Sie die folgende Klasse von Hand aus und schreiben Sie auf, was auf dem Bildschirm ausgegeben wird:

```
1 class Aufgabe1{
2   public static void main(String [] args){
3     int i = 42;
4     for (int j = i; j>=i%15;j=j-5){
5       System.out.println(j);
6     }
7   }
8 }
```

Aufgabe 2 Die folgenden Javaprogramme enthalten Fehler. Beschreiben Sie die Fehler und korrigieren Sie sie:

a)
```
1 class Aufgabe2a {
2    static int dividiere(int z, int n){
3      if (n!=0) return z/n;
4    }
5 }
```

b)
```
1 class Aufgabe2b {
2    static int dividiere(int z, int n){
3      if (n!=0) return z/n;
4      throw new Exception("division durch 0");
5    }
6 }
```

c)
```
1 class Aufgabe2c {
2   String x;
3   String y;
4
```

```
5     Aufgabe2c(String x, StringBuffer y){
6       this.x = x;
7       this.y = y;
8     }
9   }
```

d)
```
1   class Aufgabe2d{
2     public void printTwiceAsUpperCase(String s){
3       Object doppelS = s+s;
4       System.out.println(doppelS.toUpperCase());
5     }
6   }
```

e)
```
1   class Aufgabe2e_1{
2     private String s = "hallo";
3     public String getS(){return s;}
4   }
```

```
1   class Aufgabe2e_2{
2     public static String getS(Aufgabe2e_1 e1){return e1.s;}
3   }
```

f)
```
1   class Aufgabe2f{
2     public void printString(String s){
3       System.out.println(s);
4     }
5
6     public static void main(String [] args){
7       printString("hallo");
8     }
9   }
```

Aufgabe 3 Gegeben sei die folgende abstrakte Klasse für Listen, gemäß unserer Spezifikation aus der Vorlesung:

```
1   abstract class AbList{
2       abstract public AbList empty();
3       abstract public AbList cons(Object x, AbList xs);
4       abstract public boolean isEmpty();
5       abstract public Object head();
6       abstract public AbList tail();
7   }
```

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:

```
1  class Aufgabe5_1{
2      public String getInfo(){return "5_1";}
3  }
```

```
1  class Aufgabe5_2 extends Aufgabe5_1{
2      public String getInfo(){return "5_2";}
3  }
```

```
1  class Aufgabe5_3 extends Aufgabe5_2{
2      public String getInfo(){return "5_3";}
3  }
```

```
1  class Aufgabe5 {
2    static String getInfo(Aufgabe5_1 o){return o.getInfo();}
3
4    public static void main(String [] args){
5      Aufgabe5_1 a1 = new Aufgabe5_3();
6      Aufgabe5_1 a2 = new Aufgabe5_2();
7      Aufgabe5_1 a3 = new Aufgabe5_1();
8      System.out.println(a3.getInfo());
9      System.out.println(getInfo(a3));
10     System.out.println(getInfo(a1));
11     System.out.println(a2.getInfo());
12   }
13 }
```

Führen Sie die Methode `main` von Hand aus. Was wird auf dem Bildschirm ausgegeben?

Aufgabe 6 Betrachten Sie die folgende Klasse:

```
1  class Aufgabe6{
2    String g1(){
3      new NullPointerException();
4      return "g1";
5    }
6
7    String g2()throws Exception{
8      try {
9        return g1();
10     }catch (NullPointerException _){
11       throw new Exception();
12     }
13   }
14
15   String g3(){
16     String result="";
17     try {
18       result=g2();
19     }catch (NullPointerException _){
20       return "null pointer";
21     }catch (Exception _){
22       return "exception";
23     }
24     return result;
25   }
26   public static void main(String [] args){
27     System.out.println(new Aufgabe6().g3());
28   }
29 }
```

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 einem Texteditor oder einer Javaprogrammierumgebung ihrer Wahl und lassen Sie es laufen. Machen Sie diese Übung sowohl auf einem Windowsbetriebssystem als auch auf einem Unix-artigen Betriebssystem.

Aufgabe 2 Schreiben sie ein Programm, das ein Objekt der Klasse `Minimal` erzeugt und auf dem Bildschirm ausgibt.

Aufgabe 3 Schreiben Sie Klassen, die je eines der folgenden Objektarten repräsentieren können:

- Personen mit Namen und Adresse.
- Bücher, die einen Titel und einen Autor haben.
- Buchausleihe, in der vermerkt ist, daß eine Person ein bestimmtes Buch ausgeliehen hat.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben sie für jede dieser Klassen eine Methode `toString` mit dem Ergibnistyp `String`. Das Ergebnis soll eine gute textuelle Beschreibung des Objektes sein.

c) Fügen Sie der Klasse `Person` ein statisches Feld zu, in dem stets der Name der zuletzt erzeugten Person gespeichert ist.

d) 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 Bildschrim ausgeben. Zeigen Sie in diesen Hauptmethode, daß das statische Feld der Klasse `Person` wie spezifiziert funktioniert.

e) Testen Sie, ob sich statische Eigenschaften auch über die Objekte einer Klasse und nicht nur über den Klassennamen zugreifen lassen.

Aufgabe 4 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 eininge der für die Klasse `String` definierten Methoden.

Aufgabe 5 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 6 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 7 Modellieren und schreiben Sie eine Klasse, die ein Bankkonto darstellt. Auf das Bankkonto sollen Einzahlungen und Auszahlungen vorgenommen werden können. Es gibt eine 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 8 In dieser Aufgabe können Sie zum ersten mal eine Funktionalität schreiben, die durch eine graphisches Benutzeroberfläche bedient wird.

a) Kopieren Sie sich die Klassen

  - `Dialogue` (`http://www.tfh-berlin.de/~panitz/prog1/../Dialogue.java`) und
  - `String2String` (`http://www.tfh-berlin.de/~panitz/prog1/../String2String.java`)

  Die Klasse `Dialogue` definiert eine kleine graphische Benutzerschnittstelle. Zwei Textflächen und ein Schaltknopf. Beim Drücken des Schaltknopfs wird der Text aus der oberen Textfläche ausgelesen, mit dem Text die Methode `string2string` des Objektes der Klasse `String2String` aufgerufen und das Ergebnis in die untere Textfläche geschrieben.

  Übersetzen Sie die Klassen und starten Sie die `main`-Methode.

b) Ändern sie die Methode `string2string` so, daß das Ergebnis alle Buchstaben in entsprechende Großbuchstaben umwandelt.

Aufgabe 9 Schreiben Sie eine Methode, `int readInt(String str)`. Diese Methode soll einen String, der nur aus Ziffern besteht in die von ihm repräsentierte Zahl umwandeln. Benutzen sie hierzu Methoden der `String`-Klasse, die es erlauben, einzelne Buchstaben einer Zeichenkette zu selektieren. Behandeln sie den primitiven Typen `char` dabei, als sei es der Typ `int`.

Aufgabe 10 **Hinweis:** Die Aufgabe wird fortgesetzt werden.

a) Schreiben Sie eine Klasse `From`. Objekte dieser Klasse sollen die Methode `int next()` haben. Ausgehend von einem initialen Anfangswert, soll diese Methode stets eine um eins höhere Zahl bei jedem Aufruf ausgeben.

Beispiel: Für ein Objekt `frm`, das mit `new From(42)` konstruiert wurde, sollen die Aufrufe

```
1 System.out.println(frm.next());
2 System.out.println(frm.next());
3 System.out.println(frm.next());
4 System.out.println(frm.next());
```

die Zahlen `42, 43, 44, 45` ausgegeben.

b) Schreiben Sie eine Klasse `Sieb`. Objekte dieser Klasse sollen ein Objekt der Klasse `From` sowie eine Zahl `n` enthalten. Auch in dieser Klasse schreiben sie eine Methode `int next()`. Das Ergebnis soll die nächste Zahl die das Objekt vom Typ `From` ausgibt sein, die nicht durch `n` teilbar ist.

Hinweis: der Operator `%` berechnet den ganzzahligen Rest einer Division.

Aufgabe 11 Mit dem Prinzip der Vererbung können wir die Klassen `From` und `Sieb` aus der letzten Aufgabe so ändern, daß wir mit ihnen ein Programm schreiben können, das alle Primzahlen erzeugen kann.

a) Ändern sie die Klasse `Sieb` der letzten Aufgabe so ab, daß Sie eine Unterklasse der Klasse `From` erhalten.

b) Sofern sie in der ersten Unteraufgabe eine korrekte Lösung haben, druckt folgendes Programm alle Primzahlen. Testen sie dieses.

PrintPrim.java

```
1  class PrintPrim{
2    /**
3      Prints prime numbers on screen.
4    **/
5    static public void main(String [] args){
6      //initial sieve: starts with all numbers greater 1
7      From sieb = new From(2);
8
9      while (true){
10       //get the first number of new sieve
11       int i = sieb.next();
12
13       //print it
14       System.out.println(i);
15
16       //create a new sieve, which deletes all multiples of i
17       sieb = new Sieb(sieb,i);
18     }//while
```

```
19    }
20  }
```

Aufgabe 12 Lösen Sie die Aufgabe 8 jetzt, indem Sie eine Unterklasse der Klasse `String2String` schreiben.

Aufgabe 13 In dieser Aufgabe sollen Sie die Klasse `Counter` aus einer der vorhergehenden Aufgaben in ein Gui einbinden. Laden Sie sich hierzu die Klasse
`CounterGUI.java` (`http://www.tfh-berlin.de/~panitz/prog1/../CounterGUI.java`)
vom Netz und kompilieren Sie diese mit Ihrer Klasse `Counter`. Passen Sie gegebenenfalls Ihre Klasse `Counter` an.

Aufgabe 14 Für die Lösung dieser Aufgabe gibt es 3 Punkte, die auf die Klausur angerechnet werden. Voraussetzung hierzu ist, daß die Lösung mir spätestens in der Übung am 21.11.2002 gezeigt und erklärt werden kann.

In dieser Aufgabe sollen Sie römische Zahlen in arabische Zahlen umwandeln.

a) Laden Sie sich hierzu die Klasse
`Roman.java` (`http://www.tfh-berlin.de/~panitz/prog1/../Roman.java`)
vom Netz und kompilieren Sie diese mit der Klasse `Dialogue`.
Die Methode `readRoman` ist noch nicht ausprogrammiert. Ergänzen Sie den Rumpf dieser Methode, so daß die Methode entsprechend der Dokumentation
`Roman.html` (`http://www.tfh-berlin.de/~panitz/prog1/../Roman.html`)
funktioniert.

b) Schreiben Sie eine Unterklasse der Klasse `Roman`, in der die Methode `toString` so überschrieben ist, so daß sie eine Repräsentation als römische und nicht als arabische Zahl ausgibt.

c) Fügen Sie zur Klasse `Roman` Methoden `add,sub,mul,div` hinzu, mit den Signaturen: `Roman add(Roman other)`,`Roman sub(Roman other)`,....

Diese Methoden sollen die vier Grundrechenarten auf römische Zahlen realisieren.

Aufgabe 15 Nehmen Sie eine der in diesem Kapitel vorgestellten Umsetzungen von Listen und fügen Sie ihrer Listenklasse folgende Methoden zu. Führen Sie Tests für diese Methoden durch.

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

b) `List concat(List 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 Stelle der Liste zurück.

Aufgabe 16 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 17 Diese Aufgabe soll mir helfen, Listen für Ihre Leistungsbewertung zu erzeugen.

a) Implementieren Sie für Ihre Listenklasse eine Methode `String toHtmlTable()`, die für Listen Html-Code für eine Tabelle erzeugt, z.B:

```
1 <TABLE>
2   <TR>erstes Listenelement</TR>
3   <TR>zweites Listenelement</TR>
4   <TR>drittes Listenelement</TR>
5 </TABLE>
```

b) Schreiben Sie eine Klasse `Student`. Ein Student soll Felder für Namen, Vornamen und Matrikelnummer haben. Implementieren Sie für Studenten eine Methode `String toTableRow()`, die für Studenten eine Zeile eine Html Tabelle erzeugt:

```
1
2 Student s1 = new Student("Müller","Hans",167857);
3 System.out.println(s1.toTableRow());
```

soll folgende Ausgabe ergeben:

```
1 <TD>Müller</TD><TD>Hans</TD><TD>167857</TD>
```

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 Klasse:
`HtmlView` (`http://www.tfh-berlin.de/~panitz/prog1/../HtmlView.java`)
benutzen.

Aufgabe 18 Für die Lösung dieser Aufgabe gibt es 3 Punkte, die auf die Klausur angerechnet werden. Voraussetzung hierzu ist, daß die Lösung mir spätestens in der Übung am 19.12.2002 gezeigt und erklärt werden kann.

In dieser Aufgabe wird ein kleines Programm, das einen Psychoanalytiker simuliert, vervollständigt.

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

b) In der Klasse `Li`, sind die Methoden:
`reverse`, `words`, `unwords`, `stripPunctuation`, `drop`,
`tails`, `isPrefixOf`, `isPrefixIgnoreCaseOf`
noch nicht ausprogrammiert. Implementieren Sie diese Methoden und schreiben Sie Tests für jede Methode.

Wenn Ihre Tests erfolgreich sind, starten Sie die `main`-Methode der Klasse `Eliza`.

c) Ergänzen sie in der Klasse `Eliza` die Liste im Feld `respMsgs` um die auskommentierten Einträge.

d) Erfinden Sie eigene Einträge für die Liste `respMsgs`.

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

Aufgabe 19 Diese Aufgabe ist die letzte Punkteaufgabe. Für sie gibt es insgesammt vier Punkte. Dieses ist der erste Teil der Aufgabe für den es 2 Punkte gibt.

Im 2. Teil werden graphische Komponenten benutzt. Der zweite Teil der Aufgabe befindet sich im Kapitel über graphische Komponenten.

Abgabe der gesammten Aufgabe ist spätestens am 30. Januar 2003.

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/vier.zip`)

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

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

```
1 public static void main(String[] args) {
2   new VierKonsole().spiel(new VierImplementierung());
3 }
```

Aufgabe 20 Dieses ist der zweite Teil der Aufgabe zum Spiel *Vier Gewinnt*. Es gibt noch einmal 2 Punkte für diese Aufgabe. Abgabe ist spätestens in der Übung am 30.01.2002.

Laden sie sich für diese Aufgabe die Datei
vier2.zip (`http://www.tfh-berlin.de/~panitz/prog1/vier2.zip`)

a) Der Schnittstelle `VierLogik` ist eine weitere Methode zugefügt worden: `newGame()`. Implementieren Sie diese in Ihrer Klasse `VierImplementierung`

b) Schreiben Sie die folgende Hauptmethode und starten Sie diese. Sie sollten jetzt ein GUI bekommen, auf dem Sie Vier-Gewinnt spielen können.

```
1 public static void main(String [] _){
2   new Spiel(new VierImplementierung());
3 }
```

c) Schreiben Sie eine Klasse `SpielWindowListener`, die von der Klasse `WindowAdapter` ableitet. Überschreiben Sie die Methode `windowClosing`, so daß sie die Methode `System.exit` ausführt.

d) Sorgen Sie dafür, daß im Konstruktor der Klasse `Spiel` eine Instanz Ihrer Klasse `SpielWindowListener` als *WindowListener* zugefügt wird.

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 auftretenen 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

Abbildung, 7-11
AbstractList, 6-15
ArrayToString, 7-16

BoolOps, 10-5
BorderLayoutTest, 9-6
Bottom, 3-9
BreakTest, 3-13
BubbleSort, 7-19

CastError, 4-11
CastTest, 4-11
Catch1, 6-24
Catch2, 6-24
Catch3, 6-25
Catch4, 6-26
Catch5, 6-28
Clone, 7-4
Close, 9-11
Cons, 5-4
ContTest, 3-13
Copy, 7-21
Count, 9-9
Count2, 9-12
Count3, 9-13
CountActionListener, 9-9
CountButtonListener, 9-10
CountWindowListener, 9-11

Deadlock, 8-8
Dialogue, 4-7, 6-11
DialogueFunction, 6-9, 6-13
DialogueTest, 6-13
Dic, 7-12
Dictionary, 7-11
DoTest, 3-10
DoubleTheString, 4-8

ElseIf, 3-6
Empty, 5-4
EqualVsIdentical, 7-2
ErsteFelder, 2-3

FifthThrow, 6-23
FilterCondition, 5-14
Finally, 6-30
FirstArray, 7-15
FirstIf, 3-5
FirstIf2, 3-6
FirstThrow, 6-18
FlowLayoutTest, 9-6
ForBreak, 3-13
ForTest, 3-11
FourthThrow, 6-22

Get, 8-10
GetSet, 8-9–8-11
GreaterX, 5-15
GridLayoutTest, 9-7

InstanceOfTest, 4-10
Iterate, 7-9

JF, 9-2
JF2, 9-3
JT, 9-4
JTB, 9-4

Kovarianz, 7-17

LessEqualX, 5-14
Li, 5-6
LiIterator, 7-9
LinkedList, 6-16
List, 5-4, 6-16
ListTypeCast, 5-10
ListUsage, 7-7

Minimal, 2-2
Minus, 3-18
More, 7-23
MoreFinally, 6-30
MyNonPublicClass, 6-5
MyPublicClass, 6-5

NegativeNumberException, 6-22
NegativeOrderingCondition, 5-17
NoClone, 7-3
NonThrow, 6-19
NotThrowable, 6-19
NumberTooLargeException, 6-26

ObjectArray, 7-15
Operatorauswertung, 3-19
OrderingCondition, 5-17

PackageTest, 6-8

Pair, 5-20
Person, 2-4, 2-5, 4-1
Print, 7-22
PrintPrim, 4-12, B-3
PrintTwoNumbers, 8-4
PrivateTest, 6-7
ProtectedTest, 6-8
PunktVorStrich, 3-3

Reihenfolge, 3-19
Relation, 5-16
ReturnString, 7-6
ReverseComporator, 7-13
Rhyme, 7-13

SafePrintTwoNumbers, 8-6
SafeSwap, 8-6
SafeTwoNumbers, 8-5
SchachFeld, 7-18
SecondArray, 7-15
SecondThrow, 6-20
Set, 8-9
SimpleThread, 8-2
SleepingThread, 8-2
SortStringLi, 5-11
Square, 3-4
StackTest, 3-15
StackTrace, 6-29
StaticTest, 2-7
StringConstants, 10-4
StringLessEqual, 5-17
StringLi, 5-10
StringOrdering, 5-11
StringVsStringBuffer, 7-5
Student, 4-4
StudentOhneVererbung, 4-1
StuedentError1, 4-5
StuedentError2, 4-5
Summe, 3-7
Summe2, 3-9
Summe3, 3-12
Summe4, 3-12
Swap, 8-4
Switch, 10-2

TerminationTest, 3-20
Test, 6-10
TestArrayList, 6-3
Testbool, 3-2
TestboolOperator, 3-4
TestboolOperator2, 3-5
TestFirstLi, 5-6
TestFirstList, 5-5
TestImport, 6-3
TestImport2, 6-4
TestImport3, 6-4
Testint, 3-2
TestLateBinding, 4-6
TestLi, 5-10
TestPaket, 6-1
TestStudent1, 4-4
TestStudent2, 4-5
ThirdThrow, 6-21
ThrowIndex, 6-20
ToHTMLString, 6-10
TwoNumbers, 8-3

UpperCaseFunction, 6-10
UseNonPublic, 6-6
UsePublic, 6-5
UseThis, 2-8

Vergleich, 3-4
VisibilityOfFeatures, 6-7
VorUndNach, 3-10

WhileTest, 3-9
WrongCatch, 6-27

Abbildungsverzeichnis

Literaturverzeichnis