Eine Java Einführung in die Programmierung

(Entwurf)

WS 05/06

Sven Eric Panitz

FH Wiesbaden

Version 19. Januar 2006

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

Mit diesem Semester habe ich mich entschieden, das Skript wieder als allgemeine Einführung in die Programmierung mit Java zu gestalten und somit ein monolythisches Skript, das für sich stehen kann, zu erhalten. So ergibt sich, daß viele Kapitel Wiederholungen von Stoff aus den C-Vorlesungen und auch der Vorlesung Informatik 1 sind. In der Vorlesung werden diese Kapitel allerdings des Umfangs wegen nicht behandelt werden.

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

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

Inhaltsverzeichnis

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 1-3

1.2.3 Klassifizierung von Programmiersprachen 1-4

1.2.4 Arbeitshypothese 1-6

2 Objektorientierte Programmierung **2-1**

2.1 Grundkonzepte der Objektorientierung 2-1

2.1.1 Objekte und Klassen 2-1

2.1.2 Felder und Methoden 2-5

2.1.3 Überladen von Methoden 2-10

2.1.4 der this-Bezeichner 2-11

2.2 statische Eigenschaften 2-11

2.3 Vererben und Erben 2-12

2.3.1 Hinzufügen neuer Eigenschaften 2-13

2.3.2 Überschreiben bestehender Eigenschaften 2-14

2.3.3 Konstruktion 2-14

2.3.4 Zuweisungskompatibilität 2-15

2.3.5 Späte Bindung (late binding) 2-17

2.3.6 Zugriff auf Methoden der Oberklasse 2-19

2.3.7 Die Klasse Object 2-20

2.3.8 Klassentest 2-21

2.3.9 Typzusicherung (Cast) 2-22

3 Imperative Konzepte **3-1**
3.1 Primitive Typen, die klassenlose Gesellschaft . . . 3-1
3.1.1 Zahlenmengen in der Mathematik . . . 3-2
3.1.2 Zahlenmengen im Rechner . . . 3-2
3.1.3 natürliche Zahlen . . . 3-2
3.1.4 ganze Zahlen . . . 3-3
3.1.5 Kommazahlen . . . 3-6
3.1.6 Der Typ: boolean . . . 3-9
3.1.7 Boxen für primitive Typen . . . 3-10
3.2 Operatoren . . . 3-11
3.3 Zusammengesetzte Befehle . . . 3-13
3.3.1 Bedingungsabfrage mit: if . . . 3-13
3.3.2 Iteration . . . 3-16
3.3.3 die switch Anweisung . . . 3-22
3.3.4 Rekursion und Iteration . . . 3-24
3.4 Ausdrücke und Befehle . . . 3-25
3.4.1 Bedingungen als Befehl oder als Ausdruck . . . 3-25
3.4.2 Auswertungsreihenfolge und Seiteneffekte von Ausdrücken . . . 3-26
3.4.3 Auswertung der bool'schen Operatoren . . . 3-29
3.5 Reihungen . . . 3-32
3.5.1 Deklaration von Reihungen . . . 3-32
3.5.2 Erzeugen von Reihungen . . . 3-32
3.5.3 Zugriff auf Elemente . . . 3-33
3.5.4 Ändern von Elementen . . . 3-34
3.5.5 Das Kovarianzproblem . . . 3-34
3.5.6 Weitere Methoden für Reihungen . . . 3-35
3.5.7 Reihungen von Reihungen . . . 3-35
3.5.8 Iteration durch Reihungen . . . 3-36
3.5.9 Blubbersortierung . . . 3-36
3.6 Aufzählungstypen . . . 3-38

4 Weiterführende Konzepte **4-1**
4.1 Pakete . . . 4-1
4.1.1 Paketdeklaration . . . 4-1
4.1.2 Übersetzen von Paketen . . . 4-2
4.1.3 Starten von Klassen in Paketen . . . 4-2

4.1.4 Das Java Standardpaket 4-3
4.1.5 Benutzung von Klassen in anderen Paketen 4-3
4.1.6 Importieren von Paketen und Klassen 4-3
4.1.7 Statische Imports 4-5
4.2 Sichtbarkeitsattribute 4-6
4.2.1 Sichtbarkeitsattribute für Klassen 4-6
4.2.2 Sichtbarkeitsattribute für Eigenschaften 4-7
4.2.3 Private Felder mit get- und set-Methoden 4-10
4.2.4 Überschriebene Methodensichtbarkeiten 4-10
4.3 Schnittstellen (Interfaces) und abstrakte Klassen 4-11
4.3.1 Schnittstellen 4-11
4.3.2 Abstrakte Klassen 4-18
4.4 Ausnahme- und Fehlerbehandlung 4-18
4.4.1 Ausnahme- und Fehlerklassen 4-19
4.4.2 Werfen von Ausnahmen 4-19
4.4.3 Deklaration von geworfenen Ausnahmen 4-22
4.4.4 Eigene Ausnahmeklassen 4-23
4.4.5 Fangen von Ausnahmen 4-24
4.4.6 Der Aufrufkeller 4-30
4.4.7 Schließlich und finally 4-31
4.4.8 Anwendungslogik per Ausnahmen 4-32

5 Generische Programmierung 5-1
5.1 Generische Typen 5-1
5.1.1 Generische Klassen 5-1
5.1.2 Generische Schnittstellen 5-7
5.1.3 Kovarianz gegen Kontravarianz 5-10
5.1.4 Sammlungsklassen 5-11
5.1.5 Generische Methoden 5-12
5.2 Iteration 5-12
5.2.1 Die neuen Schnittstellen Iterable 5-14

6 Datentypen und Algorithmen 6-1
6.1 Listen . . . 6-1
6.1.1 Formale Spezifikation . . . 6-1
6.1.2 Modellierung . . . 6-5
6.1.3 Codierung . . . 6-7
6.1.4 Methoden für Listen . . . 6-12
6.1.5 Sortierung . . . 6-15
6.1.6 Formale Beweise über Listenalgorithmen . . . 6-27
6.2 Bäume . . . 6-29
6.2.1 Formale Spezifikation . . . 6-30
6.2.2 Modellierung . . . 6-31
6.2.3 Implementierung . . . 6-32
6.3 Algorithmen auf Bäumen . . . 6-33
6.3.1 Knotenanzahl . . . 6-33
6.3.2 toString . . . 6-34
6.3.3 Linearisieren . . . 6-34
6.3.4 Lösung . . . 6-36
6.3.5 Eltern und Geschwister . . . 6-46
6.3.6 Modifizierende Methoden . . . 6-47
6.4 Binärbäume . . . 6-50
6.4.1 Probleme der Klasse BinTree . . . 6-51
6.4.2 Linearisieren binärer Bäume . . . 6-52
6.5 Binäre Suchbäume . . . 6-54
6.5.1 Suchen in Binärbäumen . . . 6-58
6.5.2 Entartete Bäume . . . 6-58

7 Java Standardklassen 7-1
7.1 Die Klasse Objekt . . . 7-1
7.1.1 Die Methoden der Klasse Object . . . 7-2
7.2 String und Stringbuffer . . . 7-6
7.3 Sammlungsklassen . . . 7-7
7.3.1 Listen . . . 7-7
7.3.2 Mengen . . . 7-8
7.3.3 Iteratoren . . . 7-8
7.3.4 Abbildungen . . . 7-10
7.3.5 Weitere Sammlungsmethoden . . . 7-13

7.3.6 Vergleiche . . . 7-13
7.4 Ein- und Ausgabe . . . 7-14
7.4.1 Zeichenströme . . . 7-15
7.4.2 Byteströme . . . 7-15
7.4.3 Ausnahmen . . . 7-15
7.4.4 Schreiben und Lesen . . . 7-15
7.4.5 Datenströme . . . 7-15
7.4.6 Vom InputStream zum Reader, vom OutputStream zum Writer . . . 7-16
7.4.7 Ströme für Objekte . . . 7-20
7.4.8 Stromloses IO . . . 7-21

8 Qualitätssicherung 8-1
8.1 Verträge . . . 8-1
8.2 Unit Tests . . . 8-2
8.3 Logging . . . 8-2
8.4 Einführung in JavaDoc (Autor: Robert Klein) . . . 8-2
8.4.1 Einleitung und Motivation . . . 8-2
8.4.2 Das Format eines JavaDoc-Kommentars . . . 8-3
8.4.3 Schreibstil . . . 8-4
8.4.4 Tags . . . 8-5
8.4.5 Kommandozeilenoptionen . . . 8-6
8.4.6 Epilog . . . 8-6
8.4.7 ein vollständiges Beispiel . . . 8-6
8.5 Qualitätsmaße . . . 8-9

9 Algebraische Datentypen 9-1
9.1 Das Besuchsmuster . . . 9-1
9.1.1 Klassen zur Baumdarstellung eines Programms . . . 9-3
9.2 Putzen der Kochplatte . . . 9-7
9.2.1 Itroduction . . . 9-7
9.2.2 The Problem . . . 9-8
9.2.3 Naïve hand coded instances . . . 9-10
9.2.4 Generic Transformers . . . 9-11
9.2.5 Generic Queries . . . 9-14
9.2.6 Controling the traversal . . . 9-17
9.2.7 Conclusion . . . 9-19

9.2.8 Company classes 9-19
9.3 Generierung von Code mit APT 9-21
9.3.1 ADT 9-21
9.3.2 Annotations for Algebraic Types 9-21
9.3.3 Implementation 9-22
9.3.4 Example: a tiny imperativ programming language 9-30
9.3.5 javacc Parser für Klip 9-33
9.3.6 Conclusion 9-36

10 XML **10-1**
10.1 XML-Format 10-2
10.1.1 Elemente 10-2
10.1.2 Attribute 10-4
10.1.3 Kommentare 10-4
10.1.4 Character Entities 10-5
10.1.5 CDATA-Sections 10-5
10.1.6 Processing Instructions 10-6
10.1.7 Namensräume 10-6
10.2 Codierungen 10-7
10.3 Dokumente als Bäume in Java 10-8
10.3.1 APIs für XML 10-8
10.4 Transformationen und Queries 10-16
10.4.1 XPath: Pfade in Dokumenten 10-16
10.4.2 XSLT: Transformationen in Dokumenten 10-42
10.4.3 XQuery: Anfragen 10-47
10.5 Dokumenttypen 10-48
10.5.1 DTD 10-48
10.5.2 Schema 10-50
10.5.3 Geläufige Dokumenttypen 10-50
10.6 Aufgaben 10-51

11 Steuerfäden **11-1**
11.1 Schreiben von Steuerfäden 11-1
11.1.1 Schlafenlegen von Prozessen 11-2
11.2 Koordination nebenläufiger Steuerfäden 11-3
11.2.1 Benutzung gemeinsamer Objekte 11-3

11.2.2 Synchronisation 11-5
11.2.3 Verklemmungen 11-7
11.2.4 Warten und Benachrichtigen 11-9

12 Swing: Eine Bibliothek für graphische Oberflächen 12-1
12.1 Graphische Benutzeroberflächen 12-2
12.1.1 Graphische Komponenten 12-2
12.1.2 Ereignisbehandlung 12-5
12.1.3 Setzen des Layouts 12-7
12.2 Exkurs: verschachtelte Klassen 12-10
12.2.1 Innere Klassen 12-10
12.2.2 Anonyme Klassen 12-11
12.2.3 Statische verschachtelte Klassen 12-12
12.3 Applets 12-12
12.3.1 Die Klasse JApplet 12-13
12.3.2 Methoden eines Applets 12-14
12.3.3 Sicherheit 12-14
12.3.4 Laden von weiteren Resourcen 12-14
12.4 Selbstdefinierte graphische Komponenten 12-14
12.4.1 Graphics Objekte 12-14
12.4.2 Dimensionen 12-16
12.4.3 Farben 12-17
12.4.4 Fonts und ihre Metrik 12-23
12.4.5 Erzeugen graphischer Dateien 12-25
12.4.6 Graphics2D 12-27
12.5 Bäume graphisch darstellen 12-29
12.5.1 Bäume mit JTree graphisch darstellen 12-29
12.5.2 Bäume zeichnen 12-32
12.6 Weitere Komponente und Ereignisse 12-38
12.6.1 Mausereignisse 12-39
12.6.2 Fensterereignisse 12-40
12.6.3 Weitere Komponenten 12-41
12.7 Swing und Steuerfäden 12-44
12.7.1 Timer in Swing 12-46
12.7.2 Eine kleine Spielbibliothek 12-50
12.7.3 SwingWorker für Steuerfäden in der Ereignisbehandlung 12-59

13 3 dimensionale Graphiken **13-1**

14 Formale Sprachen, Grammatiken, Parser **14-1**

14.1 formale Sprachen 14-1

14.1.1 kontextfreie Grammatik 14-1

14.1.2 Erweiterte Backus-Naur-Form 14-9

14.2 Parser 14-11

14.2.1 Parsstrategien 14-11

14.3 Handgeschriebene Parser 14-14

14.3.1 Basisklassen der Parserbibliothek 14-15

14.3.2 Parser zum Erkennen von Token 14-16

14.3.3 Parser zum Bilden der Sequenz zweier Parser 14-17

14.3.4 Parser zum Bilden der Alternative zweier Parser 14-18

14.3.5 Parser zum Verändern des Ergebnisses 14-19

14.3.6 Beispiel: arithmetische Ausdrücke 14-20

14.3.7 Tokenizer 14-29

14.4 Parsergeneratoren 14-33

14.4.1 javacc 14-34

15 Java und der Rest der Welt **15-1**

15.1 I18N, L12N 15-1

15.2 Socketkommunikation 15-1

15.3 RMI 15-1

15.3.1 RMI Server 15-2

15.3.2 RMI Client 15-3

15.3.3 Serialisierbarkeit 15-4

15.3.4 Verteilte Spielanwendung 15-4

15.4 Java und Datenbanken 15-8

15.4.1 JDBC Verbindungen 15-8

15.5 JNI: Aufruf von C 15-10

15.6 Java unterwegs 15-10

15.6.1 Hello Mobile World 15-11

15.6.2 Pong fürs Handy 15-13

16 Eine Beispielwebapplikation **16-1**
16.1 Einführung 16-1
16.1.1 Die Applikation 16-1
16.2 Java Datenobjekte (JDO) 16-2
16.2.1 Javaklasse Schreiben 16-2
16.2.2 Objektpersistenz definieren 16-3
16.2.3 Klassendatei für Persistenz modifizieren 16-3
16.2.4 Datenbanktabellen Erzeugen 16-5
16.2.5 Objekte persistent in Datenbank schreiben 16-5
16.2.6 Objekte aus Datenbank instanziieren 16-8
16.2.7 Modifizieren persistenter Objekte 16-9
16.2.8 Persistent verschiener Typen 16-11
16.2.9 Komplexere Anfragen 16-18
16.3 Servlets 16-18
16.3.1 Servlet Container 16-18
16.3.2 Struktur einer Webapplikation 16-18
16.3.3 Servletkonfiguration 16-19
16.3.4 Servletklassen 16-21
16.4 Java Server Pages 16-26
16.5 XML 16-26

17 Scala, das Java der Zukunft? **1**

A Javawerkzeuge **A-1**
A.1 Klassenpfad und Java-Archive A-1
A.1.1 Benutzung von jar-Dateien A-1
A.1.2 Der Klassenpfad A-4
A.2 APT A-5
A.3 Applets testen mit: appletviewer A-5
A.4 native2ascii A-5
A.5 Klassendateien analysieren mit javap A-5
A.6 Parsergeneratoren A-5
A.6.1 javacc A-5
A.6.2 antlr A-5

B Jugs: Java interpetiert **B-1**

C Generierung von Klassen für XML DTDs **C-1**
C.1 Introduction . . . C-1
C.1.1 Functional Programming . . . C-1
C.1.2 XML . . . C-3
C.1.3 Java . . . C-5
C.2 Tree Parser . . . C-5
C.2.1 Parser Type . . . C-5
C.2.2 Combinators . . . C-7
C.2.3 Atomic Parsers . . . C-13
C.2.4 Building parsers . . . C-16
C.3 Building a Tree Structure . . . C-19
C.3.1 Java types for DTDs . . . C-19
C.3.2 An algebraic type for DTDs . . . C-19
C.3.3 Generation of tree classes . . . C-22
C.3.4 Generation of parser code . . . C-24
C.3.5 Flattening of a DTD definition . . . C-28
C.3.6 Main generation class . . . C-32
C.3.7 Main generator class . . . C-33
C.3.8 JaxB . . . C-36
C.4 Example: Generating Java Classes for a DTD . . . C-37
C.5 Example . . . C-37
C.5.1 DTD . . . C-37
C.5.2 Converting to LaTeX . . . C-38
C.6 Conclusion . . . C-42
C.7 Javacc input file for DTD parser . . . C-42

D Hilfklasssen zu Swing **D-1**

E Fragen und Antworten als Lernhilfe **E-1**

F Wörterliste **F-1**

G Gesammelte Aufgaben **G-1**
Klassenverzeichnis . . . G-19

Kapitel 1

Einführung

1.1 Ziel der Vorlesung

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

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

1.2 Programmieren

1.2.1 Disziplinen der Programmierung

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

- **Spezifikation:** Bevor eine Programmieraufgabe bewerkstelligt werden kann, muß das zu lösende Problem spezifiziert werden. Dieses kann informell durch eine natürlichsprachliche Beschreibung bis hin zu mathematisch beschriebenen Funktionen geschehen. Gegen die Spezifikation wird programmiert. Sie beschreibt das gewünschte Verhalten des zu erstellenden Programms.
- **Modellieren:** Bevor es an die eigentliche Codierung geht, wird in der Regel die Struktur des Programms modelliert. Auch dieses kann in unterschiedlichen Detaillierungsgraden geschehen. Manchmal reichen Karteikarten als hilfreiches Mittel aus, andernfalls empfiehlt sich eine umfangreiche Modellierung mit Hilfe rechnergestützter Werkzeuge. Für die objektorientierte Programmierung hat sich UML als eine geeignete Modellierungssprache

durchgesetzt. In ihr lassen sich Klassendiagramme, Klassenhierarchien und Abhängigkeiten graphisch darstellen. Mit bestimmten Werkzeugen wie *Together* oder *Rational Rose* läßt sich direkt für eine UML-Modellierung Programmtext generieren.

- **Codieren:** Die eigentliche Codierung ist in der Regel der einzige Schritt, der direkt Code in der gewünschten Programmiersprache von Hand erzeugt. Alle anderen Schritte der Programmierung sind mehr oder weniger unabhängig von der zugrundeliegenden Programmiersprache.

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

- **Testen:** Beim Testen sind generell zu unterscheiden:

  - *Entwicklertests:* Diese werden von den Entwicklern während der Programmierung selbst geschrieben, um einzelne Programmteile (Methoden, Funktionen) separat zu testen. Es gibt eine Schule, die propagiert, Entwicklertests vor dem Code zu schreiben (*test first*). Die Tests dienen in diesem Fall als kleine Spezifikationen.
  - *Qualitätssicherung:* In der Qualitätssicherung werden die fertigen Programme gegen ihre Spezifikation getestet (*black box tests*). Hierzu werden in der Regel automatisierte Testläufe geschrieben. Die Qualitätssicherung ist personell von der Entwicklung getrennt. Es kann in der Praxis durchaus vorkommen, daß die Qualitätsabteilung mehr Mitarbeiter hat als die Entwicklungsabteilung.

- **Optimieren:** Sollten sich bei Tests oder in der Praxis Performanzprobleme zeigen, sei es durch zu hohen Speicherverbrauch als auch durch zu lange Ausführungszeiten, so wird versucht, ein Programm zu optimieren. Hierzu bedient man sich spezieller Werkzeuge (*profiler*), die für einen Programmdurchlauf ein Raum- und Zeitprofil erstellen. In diesem Profil können Programmteile, die besonders häufig durchlaufen werden, oder Objekte, die im großen Maße Speicher belegen, identifiziert werden. Mit diesen Informationen lassen sich gezielt inperformante Programmteile optimieren.

- **Verifizieren:** Eine formale Verifikation eines Programms ist ein mathematischer Beweis der Korrektheit bezüglich der Spezifikation. Das setzt natürlich voraus, daß die Spezifikation auch formal vorliegt. Man unterscheidet:

  - *partielle Korrektheit:* wenn das Programm für eine bestimte Eingabe ein Ergebnis liefert, dann ist dieses bezüglich der Spezifikation korrekt.
  - *totale Korrektheit:* Das Programm ist partiell korrekt und terminiert für jede Eingabe, d.h. liefert immer nach endlich langer Zeit ein Ergebnis.

  Eine formale Verifikation ist notorisch schwierig und allgemein nicht automatisch durchführbar. In der Praxis werden nur in ganz speziellen kritischen Anwendungen formale Verifikationen durchgeführt, z.B. bei Steuerungen gefahrenträchtiger Maschinen, so daß Menschenleben von der Korrektheit eines Programms abhängen können.

- **Wartung/Pflege:** den größten Teil seiner Zeit verbringt ein Programmierer nicht mit der Entwicklung neuer Software, sondern mit der Wartung bestehender Software. Hierzu gehören die Anpassung des Programms an neue Versionen benutzter Bibliotheken oder des Betriebssystems, auf dem das Programm läuft, sowie die Korrektur von Fehlern.

- **Debuggen:** Bei einem Programm ist immer damit zu rechnen, daß es Fehler enthält. Diese Fehler werden im besten Fall von der Qualitätssicherung entdeckt, im schlechteren Fall treten sie beim Kunden auf. Um Fehler im Programmtext zu finden, gibt es Werkzeuge, die ein schrittweises Ausführen des Programms ermöglichen *(debugger)*. Dabei lassen sich die Werte, die in bestimmten Speicherzellen stehen, auslesen und auf diese Weise der Fehler finden.

- **Internationalisieren (I18N)[1]:** Softwarefirmen wollen möglichst viel Geld mit ihrer Software verdienen und streben deshalb an, ihre Programme möglichst weltweit zu vertreiben. Hierzu muß gewährleistet sein, daß das Programm auf weltweit allen Plattformen läuft und mit verschiedenen Schriften und Textcodierungen umgehen kann. Das Programm sollte ebenso wie mit lateinischer Schrift auch mit Dokumenten in anderen Schriften umgehen können. Fremdländische Akzente und deutsche Umlaute sollten bearbeitbar sein. Aber auch unterschiedliche Tastaturbelegungen bis hin zu unterschiedlichen Schreibrichtungen sollten unterstützt werden.

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

- **Lokalisieren (L12N):** Ebenso wie die Internationalisierung beschäftigt sich die Lokalisierung damit, daß ein Programm in anderen Ländern eingesetzt werden kann. Beschäftigt sich die Internationalisierung damit, daß fremde Dokumente bearbeitet werden können, versucht die Lokalisierung, das Programm komplett für die fremde Sprache zu übersetzen. Hierzu gehören Menueinträge in der fremden Sprache, Beschriftungen der Schaltflächen oder auch Fehlermeldungen in fremder Sprache und Schrift. Insbesondere haben verschiedene Schriften unterschiedlichen Platzbedarf; auch das ist beim Erstellen der Programmoberfläche zu berücksichtigen.

- **Portieren:** Oft wird es nötig, ein Programm auf eine andere Plattform zu portieren. Ein unter Windows erstelltes Programm soll z.B. auch auf Unix-Systemen zur Verfügung stehen.

- **Dokumentieren:** Der Programmtext allein reicht in der Regel nicht aus, damit das Programm von Fremden oder dem Programmierer selbst nach geraumer Zeit gut verstanden werden kann. Um ein Programm näher zu erklären, wird im Programmtext Kommentar eingefügt. Kommentare erklären die benutzten Algorithmen, die Bedeutung bestimmter Datenfelder oder die Schnittstellen und Benutzung bestimmter Methoden.

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

1.2.2 Was ist ein Programm

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

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

pragmatische Antwort

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

mathematische Antwort

Eine Funktion, die deterministisch für Eingabewerte einen Ausgabewert berechnet.

sprachwissenschaftliche Antwort

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

operationale Antwort

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

1.2.3 Klassifizierung von Programmiersprachen

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

Hauptklassen

Im folgenden eine hilfreiche Klassifizierung in fünf verschiedene Hauptklassen.

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

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

- **logisch** (Prolog): aus der KI (künstlichen Intelligenz) stammen logische Programmiersprachen. Hier wird ein Programm als logische Formel, für die ein Beweis gesucht wird, verstanden.

Ausführungsmodelle

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

- **kompiliert** (C, Cobol, Fortran): in einem Übersetzungsschritt wird aus dem Quelltext direkt das ausführbare Programm erzeugt, das dann unabhängig von irgendwelchen Hilfen der Programmiersprache ausgeführt werden kann.
- **interpretiert** (Lisp, Scheme): der Programmtext wird nicht in eine ausführbare Datei übersetzt, sondern durch einen Interpreter Stück für Stück anhand des Quelltextes ausgeführt. Hierzu muß stets der Interpreter zur Verfügung stehen, um das Programmm auszuführen. Interpretierte Programme sind langsamer in der Ausführung als übersetzte Programme.
- **abstrakte Maschine über** *byte code* (Java, ML): dieses ist quasi eine Mischform aus den obigen zwei Ausführungsmodellen. Der Quelltext wird übersetzt in Befehle nicht für einen konkreten Computer, sondern für eine abstrakte Maschine. Für diese abstrakte Maschine steht dann ein Interpreter zur Verfügung. Der Vorteil ist, daß durch die zusätzliche Abstraktionsebene der Übersetzer unabhängig von einer konkreten Maschine Code erzeugen kann und das Programm auf auf allen Systemen laufen kann, für die es einen Interpreter der abstrakten Maschine gibt.

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

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

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

```
javac Test.java
```

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

```
java Test
```

Übersetzen eines erste Javaprogramms

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

FirstProgram.java

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

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

Aufgabe 1 Schreiben Sie das obige Programm mit einen Texteditor ihrer Wahl. Speichern Sie es als `FirstProgram.java` ab. Übersetzen Sie es mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `FirstProgram.class`. Führen Sie das Programm mit dem Javainterpreter `java` aus. Führen Sie dieses sowohl einmal auf Linux als auch einmal unter Windows durch.

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

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

Answer.java

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

1.2.4 Arbeitshypothese

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

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

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

Kapitel 2

Objektorientierte Programmierung

2.1 Grundkonzepte der Objektorientierung

2.1.1 Objekte und Klassen

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

Objektorientierte Modellierung

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

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

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

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

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

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

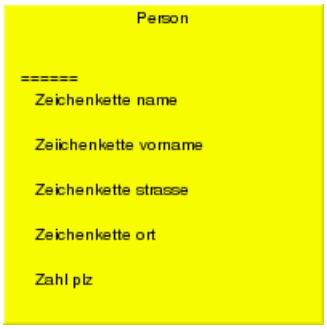


Abbildung 2.1: Modellierung einer Person.

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

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

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

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

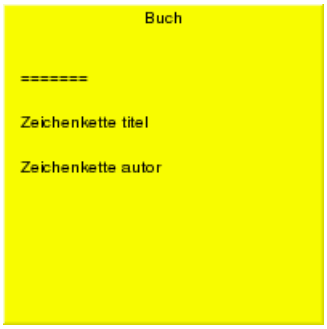


Abbildung 2.2: Modellierung eines Buches.

Abbildung 2.3: Modellierung eines Datums.

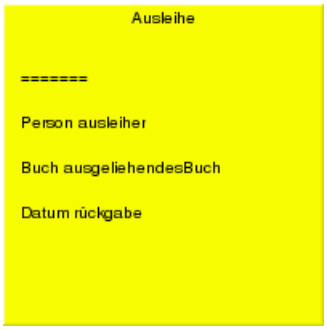


Abbildung 2.4: Modellierung eines Ausleihvorgangs.

Klassen in Java

Wir haben in einem Modellierungsschritt im letzten Abschnitt verschiedene Objektarten identifiziert und ihre Eigenschaften spezifiziert. Dazu haben wir vier Karteikarten geschrieben. Jetzt können wir versuchen, diese Modellierung in Java umzusetzen. In Java beschreibt eine Klasse eine Menge von Objekten gleicher Art. Damit entspricht eine Klasse einer der Karteikarten in unsrer Modellierung. Die Klassendefinition ist eine Beschreibung der möglichen Objekte. In ihr ist definiert, was für Daten zu den Objekten gehören. Zusätzlich können wir in einer Klasse noch schreiben, welche Operationen auf diesen Daten angewendet werden können. Klassendefinitionen sind die eigentlichen Programmtexte, die der Programmierer schreibt.

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

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

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

Minimal.java

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

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

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

Objekte

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

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

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

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

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

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

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

[2] Auch hier werden wir Ausnahmen kennenlernen.

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

2.1.2 Felder und Methoden

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

Felder

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

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

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

ErsteFelder.java

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

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

Feldnamen werden per Konvention immer klein geschrieben.

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

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

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

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

Nach einer Zuweisung repräsentiert das Feld das ihm zugewiesene Objekt.

Methoden

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

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

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

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

Methoden und Parameternamen werden per Konvention immer klein geschrieben.

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

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

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

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

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

```
4
5     String getFullName(){
6       return (vorname+" "+nachname);
7     }
8   }
```

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

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

PersonExample2.java

```
1  class PersonExample2 {
2    String vorname;
3    String nachname;
4
5    void setVorname(String newName){
6      vorname = newName;
7    }
8    void setNachname(String newName){
9      nachname = newName;
10   }
11 }
```

Obige Methoden weisen konkrete Objekte den Feldern des Objektes zu.

Konstruktoren

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

Person1.java

```
1 class Person1 {
2   String vorname;
3   String nachname;
4
5   Person1(String derVorname, String derNachname){
6     vorname  = derVorname;
7     nachname = derNachname;
8   }
9
```

```
10    String getFullName(){
11      return (vorname+" "+nachname);
12    }
13  }
```

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

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

TestePerson1.java

```
1  class TestePerson1 {
2
3    public static void main(String [] _){
4      new Person1("Nicolo","Paganini");
5    }
6  }
```

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

lokale Felder

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

FirstLocalFields.java

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

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

Zugriff auf Eigenschaften eines Objektes

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

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

GetPersonName.java

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

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

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

CallFullNameMethod.java

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

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

Aufgabe 4 Schreiben Sie Klassen, die die Objekte des Bibliotheksystems repräsentieren können:

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

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

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

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

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

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

Aufgabe 5 Suchen Sie auf Ihrer lokalen Javainstallation oder im Netz auf den Seiten von Sun (`http://www.javasoft.com`) nach der Dokumentation der Standardklassen von Java. Suchen Sie die Dokumentation der Klasse `String`. Testen Sie einige der für die Klasse `String` definierten Methoden.

2.1.3 Überladen von Methoden

Variable Parameteranzahl

Als zusätzliches kleines Gimmik ist in Java eingebaut worden, daß Methoden mit einer variablen Parameteranzahl definiert werden können. Dieses wird durch drei Punkte nach dem Parametertyp in der Signatur gekennzeichnet. Damit wird angegeben, daß eine bieliebige Anzahl dieser Parameter bei einem Methodenaufruf geben kann.

Beispiel:
Es läßt sich so eine Methode schreiben, die mit beliebig vielen Stringparametern aufgerufen werden kann.

VarParams.java

```
1  public class VarParams{
2    static public String append(String... args){
3      String result="";
4      for (String a:args)
5        result=result+a;
6      return result;
7    }
8
9    public static void main(String [] _){
10     System.out.println(append("hello"," ","world"));
11   }
12 }
```

Die Methode `append` konkateniert endlich viele `String`-Objekte.

Wie schon für Aufzählungen können wir auch einmal schauen, was für Code der Javakompilierer für solche Methoden erzeugt.

```
sep@linux:~/fh/java1.5/examples> javap -classpath classes/ name.panitz.java15.VarParams
Compiled from "VarParams.java"
public class name.panitz.java15.VarParams extends java.lang.Object{
    public name.panitz.java15.VarParams();
    public static java.lang.String append(java.lang.String[]);
    public static void main(java.lang.String[]);
}

sep@linux:~/fh/java1.5/examples>
```

Wie man sieht wird für die variable Parameteranzahl eine Reihung erzeugt. Der Javakompilierer sorgt bei Aufrufen der Methode dafür, daß die entsprechenden Parameter in eine Reihung verpackt werden. Daher können wir mit dem Parameter wie mit einer Reihung arbeiten.

2.1.4 der this-Bezeichner

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

UseThis.java

```
class UseThis {
  String aField ;
  UseThis(String aField){
    this.aField = aField;
  }
}
```

2.2 statische Eigenschaften

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

StaticTest.java

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

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

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

CallStaticTest.java

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

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

Statische Eigenschaften nennt man auch Klassenfelder bzw. Klassenmethoden.

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

Aufgabe 6 Ergänzen Sie jetzt die Klasse `Person` aus der letzten Aufgabe um ein statisches Feld `letzerVorname` mit einer Zeichenkette, die angeben soll, welchen Vornamen das zuletzt erzeugte Objekt vom Typ `Person` hatte. Hierzu müssen Sie im Konstruktor der Klasse Person dafür sorgen, daß nach der Zuweisung der Objektfelder auch noch das Feld `letzerVorname` verändert wird. Testen Sie in einer Testklasse, daß sich tatsächlich nach jeder Erzeugung einer neuen Person dieses Feld verändert hat.

2.3 Vererben und Erben

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

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

Person2.java

```
class Person2 {

  String name ;
  String address;

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

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

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

StudentOhneVererbung.java

```
class StudentOhneVererbung {

```

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

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

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

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

Student.java

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

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

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

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

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

2.3.1 Hinzufügen neuer Eigenschaften

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

[6] Dieses ist z.B. in C++ möglich.

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

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

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

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

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

2.3.2 Überschreiben bestehender Eigenschaften

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

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

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

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

2.3.3 Konstruktion

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

Student.java

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

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

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

TestStudent.java

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

2.3.4 Zuweisungskompatibilität

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

TestStudent1.java

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

Alle Studenten sind auch Personen.

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

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

Die Kompilierung dieser Klasse führt zu folgender Fehlermeldung:

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

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

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

TestStudent2.java

```
class TestStudent2 {

  static void printPerson(Person2 p){
    System.out.println(p.toString());
  }

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

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

```
class StudentError2{

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

  public static void main(String [] args){
     Person2 p = new Person2("Martin Müller","Hauptstraße");
     printStudent(p);
  }
}
```

Auch hier führt die Kompilierung zu einer entsprechenden Fehlermeldung:

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

2.3.5 Späte Bindung (late binding)

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

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

TestLateBinding.java

```
class TestLateBinding {

  public static void main(String [] args){
    Student s = new Student("Martin Müller","Hauptstraße",756456);
    Person2 p1 = new Person2("Harald Schmidt","Marktplatz");

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

    Person2 p2 = new Student("Martin Müller","Hauptstraße",756456);
    System.out.println(p2.toString());
  }
}
```

Dieses Programm erzeugt folgende Ausgabe:

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

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

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

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

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

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

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

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

Aufgabe 7 In dieser Aufgabe sollen Sie eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben.

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

ButtonLogic.java

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

Dialogue.java

```
1 • import javax.swing.*;
2   import java.awt.event.*;
3   import java.awt.*;
4   class Dialogue extends JFrame{
5
6     final ButtonLogic logic;
7
8     final JButton button;
9     final JTextField inputField  = new JTextField(20) ;
10    final JTextField outputField = new JTextField(20) ;
11    final JPanel p = new JPanel();
12
13    Dialogue(ButtonLogic l){
14      logic = l;
15      button=new JButton(logic.getDescription());
16      button.addActionListener
17       (new ActionListener(){
18          public void actionPerformed(ActionEvent _){
19            outputField.setText
20               (logic.eval(inputField.getText().trim()));
21          }
22        });
23      p.setLayout(new BorderLayout());
24      p.add(inputField,BorderLayout.NORTH);
25      p.add(button,BorderLayout.CENTER);
26      p.add(outputField,BorderLayout.SOUTH);
27      getContentPane().add(p);
28      pack();
29      setVisible(true);
30    }
31  }
```

TestDialogue.java

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

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

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

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

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

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

Methoden als Daten

Mit dem Prinzip der späten Methodenbindung können wir unsere ursprüngliche Arbeitshypothese, daß Daten und Programme zwei unterschiedliche Konzepte sind, etwas aufweichen. Objekte enthalten in ihren Feldern die Daten und mit ihren Methoden Unterprogramme. Wenn in einem Methodenaufruf ein Objekt übergeben wird, übergeben wir somit in dieser Methode nicht nur spezifische Daten, sondern auch spezifische Unterprogramme. Dieses haben wir uns in der letzten Aufgabe zunutze gemacht, um Funktionalität an eine graphische Benutzeroberfläche zu übergeben. Hierzu betrachten wir den Konstruktor und die Benutzung der Klasse `Dialogue`, die in einer der letzten Aufgaben vorgegeben war:

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

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

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

2.3.6 Zugriff auf Methoden der Oberklasse

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

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

Der Ausdruck

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

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

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

2.3.7 Die Klasse Object

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

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

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

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

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

Die Methode equals

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

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

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

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

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

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

2.3.8 Klassentest

Wie wir oben gesehen haben, können wir zu wenige Informationen über den Typen eines Objektes haben. Objekte wissen aber selbst, von welcher Klasse sie einmal erzeugt wurden. Java stellt einen binären Operator zur Verfügung, der erlaubt, abzufragen, ob ein Objekt zu einer Klasse gehört. Dieser Operator heißt `instanceof`. Er hat links ein Objekt und rechts einen Klassennamen. Das Ergebnis ist ein bool'scher Wert, der genau dann wahr ist, wenn das Objekt eine Instanz der Klasse ist.

InstanceOfTest.java

```
class InstanceOfTest {
  public static void main(String [] str){
     Person2 p1 = new Person2("Strindberg","Skandinavien");
     Person2 p2 = new Student("Ibsen","Skandinavien",789565);
     if (p1 instanceof Student)
       System.out.println("p1 ist ein  Student.");
     if (p2 instanceof Student)
       System.out.println("p2 ist einStudent.");
     if (p1 instanceof Person2)
       System.out.println("p1 ist eine Person.");
     if (p2 instanceof Person2)
       System.out.println("p2 ist eine Person.");
  }
}
```

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

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

2.3.9 Typzusicherung (Cast)

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

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

CastTest.java

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

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

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

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

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

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

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

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

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

Kapitel 3

Imperative Konzepte

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

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

3.1 Primitive Typen, die klassenlose Gesellschaft

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

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

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

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

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

3.1.1 Zahlenmengen in der Mathematik

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

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

Es gilt folgende Mengeninklusion zwischen diesen Mengen:

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

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

3.1.2 Zahlenmengen im Rechner

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

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

3.1.3 natürliche Zahlen

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

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

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

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

Gebräuchliche Basen sind:

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

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

> **Beispiel:**
> Die Zahl 10 in Bezug auf unterschiedliche Basen dargestellt:
>
> $$(10)_{10} = (A)_{16} = (12)_8 = (1010)_2$$

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

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

3.1.4 ganze Zahlen

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

Vorzeichen und Betrag

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

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

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

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

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

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

> **Beispiel:**
> Versuchen wir, das gängige Additionsverfahren für 10 und -10 in der Vorzeichendarstellung anzuwenden, so erhalten wir:
>
> $$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$
>
> Das Ergebnis stellt keinenfalls die Zahl 0 dar, sondern die Zahl -20.

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

Einerkomplement

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

> **Beispiel:**
> Bei einer Wortlänge von 8 Bit werden die Zahlen 10 und -10 durch folgende Bitmuster dargestellt: 00001010 und 11110101.
>
> Jetzt können auch negative Zahlen mit dem gängigen Additionsverfahren addiert werden, also kann die Subtraktion durch ein Additionsverfahren durchgeführt werden.
>
> $$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$$
>
> Das errechnete Bitmuster stellt die *negative* Null dar.

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

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

Zweierkomplement

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

> **Beispiel:**
> Bei einer Wortlänge von 8 Bit werden die Zahlen 10 und -10 durch folgende Bitmuster dargestellt: 00001010 und 11110110.
>
> Jetzt können weiterhin auch negative Zahlen mit dem gängigen Additionsverfahren addiert werden, also kann die Subtraktion durch ein Additionsverfahren durchgeführt werden.
>
> $$\begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$
>
> Das errechnete Bitmuster stellt die Null dar.

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

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

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

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

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

Aufgabe 8 Starten Sie folgendes Javaprogramm:

TestInteger.java

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

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

Erklären Sie die Ausgabe.

Frage eines Softwaremenschen an die Hardwarebauer

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

3.1.5 Kommazahlen

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

Festkommazahlen

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

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

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

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

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

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

Fließkommazahlen

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

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

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

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

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

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

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

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

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

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

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

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

DoubleTest.java

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

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

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

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

Rounded.java

```
class Rounded {
  public static void main(String [] args){
    System.out.println(8f);
    System.out.println(88f);
    System.out.println(8888f);
    System.out.println(88888f);
    System.out.println(888888f);
    System.out.println(8888888f);
    System.out.println(88888888f);
    System.out.println(888888888f);
    System.out.println(8888888888f);
    System.out.println(88888888888f);
    System.out.println(888888888888f);
    System.out.println(8888888888888f);

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

```

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

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

3.1.6 Der Typ: boolean

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

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

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

Testbool.java

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

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

3.1.7 Boxen für primitive Typen

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

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

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

ManualBoxing.java

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

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

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

AutomaticBoxing.java

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

```
10      System.out.println((i2+i3)*i4);
11    }
12  }
```

3.2 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

```
class Vergleich{

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

Bool'sche Operatoren

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

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

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

TestboolOperator.java

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

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

TestboolOperator2.java

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

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

3.3 Zusammengesetzte Befehle

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

3.3.1 Bedingungsabfrage mit: if

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

FirstIf.java

```
class FirstIf {

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

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

}
```

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

[6] Konstrukte mit Operatoren nennt man dagegen Ausdrücke.

FirstIf2.java

```
class FirstIf2 {

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

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

}
```

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

ElseIf.java

```
class ElseIf {

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

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

}
```

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

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

Summe.java

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

```
6         return summe(i-1) + i;
7       }
8     }
9   }
```

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

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

Aufgabe 10 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 11 Schreiben Sie mit den bisher vorgestellten Konzepten ein Programm, das unendlich oft das Wort `Hallo` auf den Bildschirm ausgibt. Was beobachten Sie, wenn sie das Programm lange laufen lassen?

Aufgabe 12 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 13 Modellieren und schreiben Sie eine Klasse, die ein Bankkonto darstellt. Auf das Bankkonto sollen Einzahlungen und Auszahlungen vorgenommen werden können. Es gibt einen maximalen Kreditrahmen. Das Konto soll also nicht beliebig viel in die Miese gehen können. Schließlich muß es eine Möglichkeit geben, Zinsen zu berechnen und dem Konto gutzuschreiben.

3.3.2 Iteration

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

Schleifen mit: while

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

- die Bedingung
- und den Schleifenrumpf.

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

Vorgeprüfte Schleifen Die vorgeprüften Schleifen haben folgendes Schema in Java:

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

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

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

WhileTest.java

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

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

Summe2.java

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

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

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

    return erg;
  }
}
```

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

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

Bottom.java

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

   static boolean wahr(){return true;}

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

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

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

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

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

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

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

DoTest.java

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

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

VorUndNach.java

```
class VorUndNach {

  public static void main(String [] args){

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

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

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

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

Schleifen mit: for

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

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

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

Eine *for*-Schleife hat im Kopf

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

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

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

ForTest.java

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

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

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

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

Summe3.java

```
1 class Summe3 {
2   public static int summe(int n){
```

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

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

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

Summe4.java

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

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

Vorzeitiges Beenden von Schleifen

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

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

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

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

BreakTest.java

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

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

ForBreak.java

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

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

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

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

ContTest.java

```
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);
```

```
8         }
9     }
10 }
```

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

Aufgabe 14 Schreiben Sie jetzt die Methode zur Berechnung der Fakultät, indem Sie eine Iteration und nicht eine Rekursion benutzen.

Aufgabe 15 Schreiben Sie eine Methode
`static String darstellungZurBasis(int x,int b)`, die als Parameter eine Zahl x und eine zweite Zahl b erhält. Sie dürfen annehmen, daß $x > 0$ und $1 < b < 11$. Das Ergebnis soll eine Zeichenkette vom Typ `String` sein, in der die Zahl x zur Basis b dargestellt ist. Testen Sie ihre Methode mit unterschiedlichen Basen.

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

Aufgabe 16 Schreiben Sie eine Methode `static int readIntBase10(String str)`. Diese Methode soll einen String, der nur aus Ziffern besteht, in die von ihm repräsentierte Zahl umwandeln. Benutzen sie hierzu die Methode `charAt` der `String`-Klasse, die es erlaubt, einzelne Buchstaben einer Zeichenkette zu selektieren.

3.3.3 die switch Anweisung

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

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

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

Dem Schlüsselwort `switch` folgt ein Ausdruck, nach dessen Wert eine Fallunterscheidung getroffen werden soll. In geshweiften Klammern folgen die verschiedenen Fälle. Ein Fall beginnt

mit dem Schüsselwort `case` gefolgt von einer Konstante. Diese Konstante ist von einem ganzzahligen Typ und darf kein Ausdruck sein, der erst während der Laufzeit berechnet wird. Es muß hier eine Zahl stehen. Die Konstante muß während der Übersetzungszeit des Programms feststehen. Der Konstante folgt ein Doppelpunkt, dem dann die Anweisungen für diesen Fall folgen.

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

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

Switch.java

```
class Switch {

  public static void main (String [] args){
    switch (4*new Integer(args[0]).intValue()){
      case 42 : System.out.println(42);
      case 52 : System.out.println(52);
      case 32 : System.out.println(32);
      case 22 : System.out.println(22);
      case 12 : System.out.println(12);
      default : System.out.println("default");
    }
  }
}
```

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.

```
class Switch2 {

  public static void main (String [] args){
    switch (4*new Integer(args[0]).intValue()){
      case 42 : System.out.println(42);break;
      case 52 : System.out.println(52);break;
      case 32 : System.out.println(32);break;
      case 22 : System.out.println(22);break;
      case 12 : System.out.println(12);break;
      default : System.out.println("default");
    }
  }
}
```

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

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

3.3.4 Rekursion und Iteration

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

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

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

StackTest.java

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

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

```
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
```

3.4 Ausdrücke und Befehle

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

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

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

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

Beispiele für Befehle, die Unterbefehle und Unterausdrücke enthalten, haben wir bereits zu genüge gesehen. An bestimmten Stellen von zusammengesetzten Befehle müssen Ausdrücke eines bestimmten Typs stehen: so muß z.B. die Schleifenbedingung ein Ausdruck des Typs `boolean` sein.

3.4.1 Bedingungen als Befehl oder als Ausdruck

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

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

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

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

Folgender Code:

CondExpr.java

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

druckt erst die Zahl 1 und dann die Zahl 4.

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

3.4.2 Auswertungsreihenfolge und Seiteneffekte von Ausdrücken

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

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

Reihenfolge der Operanden

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

Minus.java

```
class Minus {

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

```
                        +erg+" wurde durchgeführt.");
    return erg;
  }
}
```

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

Operatorauswertung.java

```
class Operatorauswertung {

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

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

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

Auswertung der Methodenargumente

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

Reihenfolge.java

```
public class Reihenfolge{

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

```

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

Reihenfolge.java

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

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

Reihenfolge.java

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

Terminierung

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

TerminationTest.java

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

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

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

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

TerminationTest.java

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

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

TerminationTest.java

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

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

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

3.4.3 Auswertung der bool'schen Operatoren

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

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

In bestimmten Fällen läßt sich das Ergebnis einer bool'schen Operation bereits bestimmen, ohne den zweiten Operanden anzuschauen. Die Frage ist, ob auch Java in diesen Fällen sich zunächst

den linken Operanden anschaut und nur bei Bedarf noch den rechten Operanden betrachtet. Hierzu können wir ein Testprogramm schreiben. Wir bedienen uns dabei wieder einer Methode, die nicht nur einen bool'schen Wert als Ergebnis, sondern auch einen Seiteneffekt in Form einer Ausgabe auf den Bildschirm hat:

LazyBool.java

```
class LazyBool {

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

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

}
```

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

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

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

strikte bool'sche Operatoren

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

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

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

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

StrictBool.java

```
class StrictBool {

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

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

}
```

Und tatsächlich bekommen wir jetzt zusätzliche Ausgaben auf dem Bildschirm:

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

Aufgabe 17 Für die Lösung dieser Aufgabe gibt es 3 Punkte, die auf die Klausur angerechnet werden. Voraussetzung hierzu ist, daß die Lösung mir in der Übung gezeigt und erklärt werden kann.

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

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

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

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

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

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

e) Testen Sie Ihre Klasse `Roman`.

3.5 Reihungen

Java kennt, wie fast alle Programmiersprachen, ein weiteres Konzept von Sammlungen: Reihungen (eng. arrays).[10] Reihungen stellen im Gegensatz zu Listen oder Mengen eine Menge von Daten gleichen Typs mit fester Anzahl dar. Jedes Element einer Reihung hat einen festen Index, über den es direkt angesprochen werden kann.

3.5.1 Deklaration von Reihungen

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

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

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

ObjectArray.java

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

3.5.2 Erzeugen von Reihungen

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

Aufzählung der Reihung

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

FirstArray.java

```
1 class FirstArray {
2
3  static String [] komponisten
4   = {"carcassi","carulli","giuliani"
5     ,"molino","monzino","paganini","sor"};
6
```

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

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

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

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

Uninitialisierte Reihungen

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

SecondArray.java

```
class SecondArray {

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

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

3.5.3 Zugriff auf Elemente

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

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

```
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();
  }
}
```

3.5.4 Ändern von Elementen

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

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

3.5.5 Das Kovarianzproblem

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

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

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

Kovarianz.java

```
class Kovarianz {

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

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

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

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

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

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

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

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

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

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

3.5.6 Weitere Methoden für Reihungen

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

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

```
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.

3.5.7 Reihungen von Reihungen

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

SchachFeld.java

```
1 class SchachFeld {
2   Schachfigur [][] spielFeld
3     =  {new Schachfigur[8]
4        ,new Schachfigur[8]
5        ,new Schachfigur[8]
```

[13] Primitive Typen sind dabei ausgenommen.

```
        ,new Schachfigur[8]
        ,new Schachfigur[8]
        ,new Schachfigur[8]
        ,new Schachfigur[8]
        ,new Schachfigur[8]
        };

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

3.5.8 Iteration durch Reihungen

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

FromTo.java

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

import java.util.Iterator;

public class FromTo implements Iterable<Integer>,Iterator<Integer>{
  private final int to;
  private int from;
  public FromTo(int f,int t){to=t;from=f;}
  public boolean hasNext(){return from<=to;}
  public Integer next(){int result = from;from=from+1;return result;}
  public Iterator<Integer> iterator(){return this;}
  public void remove(){new UnsupportedOperationException();}
}
```

3.5.9 Blubbersortierung

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

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

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

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

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

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

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

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

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

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

BubbleSort.java

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

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

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

}
```

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

Aufgabe 18 (3 Punkte)

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

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

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

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

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

3.6 Aufzählungstypen

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

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

Beispiel:
Ein erster Aufzählungstyp für die Wochentage.

Wochentage.java

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

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

```
sep@linux:fh/> javap name.panitz.enums.Wochentage
Compiled from "Wochentage.java"
public class name.panitz.enums.Wochentage extends java.lang.Enum{
    public static final name.panitz.enums.Wochentage montag;
    public static final name.panitz.enums.Wochentage dienstag;
    public static final name.panitz.enums.Wochentage mittwoch;
    public static final name.panitz.enums.Wochentage donnerstag;
    public static final name.panitz.enums.Wochentage freitag;
    public static final name.panitz.enums.Wochentage sonnabend;
    public static final name.panitz.enums.Wochentage sonntag;
    public static final name.panitz.enums.Wochentage[] values();
    public static name.panitz.enums.Wochentage valueOf(java.lang.String);
    public name.panitz.enums.Wochentage(java.lang.String, int);
    public int compareTo(java.lang.Enum);
    public int compareTo(java.lang.Object);
    static {};
}
```

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

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

Tage.java

```
1  package name.panitz.enums;
2  public enum Tage {
3    montag,dienstag,mittwoch,donnerstag
4   ,freitag,sonnabend,sonntag;
5
6    public boolean isWerktag(){
7      switch (this){
8        case sonntag    :
9        case sonnabend  :return false;
10       default         :return true;
11     }
12   }
13
14   public static void main(String [] _){
15     Tage tag = freitag;
16     System.out.println(tag);
17     System.out.println(tag.ordinal());
18     System.out.println(tag.isWerktag());
19     System.out.println(sonntag.isWerktag());
20   }
21 }
```

Das Programm gibt die erwartete Ausgabe:

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

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

Beispiel:
Wir iterieren in diesem Beispiel einmal über alle Wochentage.

IterTage.java

```
1  package name.panitz.enums;
2  public class IterTage {
3    public static void main(String [] _){
4      for (Tage tag:Tage.values())
5        System.out.println(tag.ordinal()+": "+tag);
6    }
7  }
```

Die erwarttete Ausgabe ist:

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

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

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

Euroschein.java

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

Das Programm hat die folgende Ausgabe:

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

Kapitel 4

Weiterführende Konzepte

4.1 Pakete

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

4.1.1 Paketdeklaration

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

Folgende Klasse definiert sie dem Paket `testPackage` zugehörig:

MyClass.java

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

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

TestPaket.java

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

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

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

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

Der Name einer Klasse ohne die Paketnennung heißt unqualifiziert.

4.1.2 Übersetzen von Paketen

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

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

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

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

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

4.1.3 Starten von Klassen in Paketen

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

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

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

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

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

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

4.1.4 Das Java Standardpaket

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

4.1.5 Benutzung von Klassen in anderen Paketen

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

TestArrayList.java

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

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

4.1.6 Importieren von Paketen und Klassen

Importieren von Klassen

Vollqualifizierte Namen können sehr lang werden. Wenn Klassen, die in einem anderen Paket als die eigene Klasse liegen, unqualifiziert benutzt werden sollen, dann kann dieses zuvor angegeben

[3] `ArrayList` ist eine generische Klasse, ein Konzept, das wir erst in einem späteren Kapitel kennenlernen werden.

werden. Dieses geschieht zu Beginn einer Klasse in einer Importanweisung. Nur die Klassen aus dem Standardpaket `java.lang` brauchen nicht explizit durch eine Importanweisung bekannt gemacht zu werden.

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

TestImport.java

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

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

TestImport2.java

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

[4] Aus historischen Gründen wird in diesem Kapitel als Beispiel bereits mit den generischen Klassen `ArrayList` und `Vector` ein Konzept benutzt, das erst im nächsten Kapitel erklärt wird.

Importieren von Paketen

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

TestImport3.java

```
package name.panitz.utilTest;

import java.util.*;

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

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

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

4.1.7 Statische Imports

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

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

StringUtil.java

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

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

UseStringUtil.java

```
package name.panitz.staticImport;
import static name.panitz.staticImport.StringUtil.*;
public class UseStringUtil {
  static public void main(String [] args) {
    for (String arg:args)
     System.out.println(reverse(arg));
  }
}
```

Die Ausgabe dieses programms:

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

4.2 Sichtbarkeitsattribute

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

4.2.1 Sichtbarkeitsattribute für Klassen

Für Klassen gibt es zwei Möglichkeiten der Sichtbarkeit. Entweder darf von überall aus eine Klasse benutzt werden oder nur von Klassen im gleichen Paket. Syntaktisch wird dieses dadurch ausgedrückt, daß der Klassendefinition entweder das Schlüsselwort `public` vorangestellt ist oder aber kein solches Attribut voransteht:

MyPublicClass.java

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

MyNonPublicClass.java

```
package name.panitz.p1;
class MyNonPublicClass {
}
```

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

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

UsePublic.java

```
package name.panitz.p2;

import name.panitz.p1.*;

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

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

```
package name.panitz.p2;

import name.panitz.p1.*;

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

Java gibt bei der Übersetzung eine entsprechende gut verständliche Fehlermeldung:

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

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

4.2.2 Sichtbarkeitsattribute für Eigenschaften

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

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

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

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

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

VisibilityOfFeatures.java

```
package name.panitz.p1;

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

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

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

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

PrivateTest.java

```
package name.panitz.p1;

public class PrivateTest
{
```

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

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

PackageTest.java

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

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

ProtectedTest.java

```
1  package name.panitz.p2;
2  import name.panitz.p1.VisibilityOfFeatures;
3
4  public class ProtectedTest  {
5
6    public static void main(String [] args){
7      VisibilityOfFeatures v = new VisibilityOfFeatures();
8      //s1 is private and cannot be accessed
9      // System.out.println(v.s1);
10
```

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

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

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

Java wird in seinem Sichtbarkeitskonzept oft kritisiert, und das von zwei Seiten. Einerseits ist es mit den vier Sichtbarkeiten schon relativ unübersichtlich; die verschiedenen Konzepte der Vererbung und der Pakete spielen bei Sichtbarkeiten eine Rolle. Andererseits ist es nicht vollständig genug und kann verschiedene denkbare Sichtbarkeiten nicht ausdrücken.

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

4.2.3 Private Felder mit get- und set-Methoden

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

Beispiel:
Eine kleine Klasse mit Kapselung eines privaten Feldes:

```
package name.panitz.sep.skript;

public class GetSetMethod {
  private int value=0;

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

4.2.4 Überschriebene Methodensichtbarkeiten

Beim Überschreiben einer Methode darf ihr Sichtbarkeitsattribut nicht enger gemacht werden. Man darf also eine öffentliche Methode aus der Oberklasse nicht mit einer privaten, geschützten

oder paketsichtbaren Methode überschreiben. Der Javaübersetzer weist solche Versuche, eine Methode zu überschreiben, zurück:

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

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

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

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

4.3 Schnittstellen (Interfaces) und abstrakte Klassen

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

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

4.3.1 Schnittstellen

Schnittstellendeklaration

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

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

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

DialogueLogic.java

```
package name.panitz.dialoguegui;

public interface DialogueLogic {
```

```
  public String getDescription();
  public String eval(String input);
}
```

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

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

Implementierung von Schnittstellen

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

Eine mögliche Implementierung der obigen Schnittstelle ist:

ToUpperCase.java

```
package name.panitz.dialoguegui;

public class ToUpperCase implements DialogueLogic{
  protected String result;

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

Die Klausel `implements DialogueLogic` verspricht, daß in dieser Klasse für alle Methoden aus der Schnittstelle eine Implementierung existiert. In unserem Beispiel waren zwei Methoden zu implementieren, die Methode `eval` und `getDescription()`.

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

Definieren wir zum Beispiel ein zweite Schnittstelle:

ToHTMLString.java

```
package name.panitz.html;

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

Diese Schnittstelle verlangt, daß implementierende Klassen eine Methode haben, die für das Objekt eine Darstellung als HTML erzeugen können.

Jetzt können wir eine Klasse schreiben, die die beiden Schnittstellen implementiert.

ToUpper.java

```
package name.panitz.dialoguegui;

import name.panitz.html.*;

public class ToUpper extends    ToUpperCase
                     implements ToHTMLString, DialogueLogic {

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

Schnittstellen können auch einander erweitern. Dieses geschieht dadurch, daß Schnittstellen auch eine `extends`-Klausel haben. Wir können also auch eine Schnittstelle definieren, die die beiden obigen Schnittstellen zusammenfaßt:

DialogueLogics.java

```
package name.panitz.dialoguegui;

import name.panitz.html.*;

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

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

UpperConversion.java

```
package name.panitz.dialoguegui;

class UpperConversion extends ToUpper
                      implements DialogueLogics{}
```

Benutzung von Schnittstellen

Schnittstellen sind genauso Typen wie Klassen. Wir kennen jetzt also drei Arten von Typen:

- primitive Typen
- Klassen
- Schnittstellen

Parameter können vom Typ einer Schnittstellen sein, ebenso wie Felder oder Rückgabetypen von Methoden. Die Zuweisungkompatibilität nutzt nicht nur die Unterklassenbeziehung, sondern auch die Implementierungsbeziehung. Ein Objekt der Klasse `C` darf einem Feld des Typs der Schnittstelle `I` zugewiesen werden, wenn `C` die Schnittstelle `I` implementiert.

Im Folgenden eine kleine Gui-Anwendung, die wir im einzelnen noch nicht verstehen müssen. Man beachte, daß der Typ `DialogueLogics` an mehreren Stellen benutzt wird wie ein ganz normaler Klassentyp. Nur einen Konstruktoraufruf mit `new` können wir für diesen Typ nicht machen.

HtmlDialogue.java

```
1  package name.panitz.dialoguegui;
2  import java.awt.event.*;
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.plaf.basic.*;
6  import javax.swing.text.*;
7  import javax.swing.text.html.*;
8
9  public class HtmlDialogue extends JFrame{
10   final DialogueLogics logic;
11   final JButton button;
12   final JTextField inputField = new JTextField(20) ;
13   final JTextPane outputField = new JTextPane();
14   final JPanel p = new JPanel();
15
16   public HtmlDialogue(DialogueLogics l){
17     outputField.setEditorKit(new HTMLEditorKit());
18     logic = l;
19     button=new JButton(logic.getDescription());
20     button.addActionListener
21      (new ActionListener(){
22         public void actionPerformed(ActionEvent _){
23           logic.eval(inputField.getText().trim());
24           outputField.setText(logic.toHTMLString());
25           pack();
26         }
27       });
28
29     p.setLayout(new BorderLayout());
30     p.add(inputField,BorderLayout.NORTH);
31     p.add(button,BorderLayout.CENTER);
32     p.add(outputField,BorderLayout.SOUTH);
33     getContentPane().add(p);
34     pack();
35     setVisible(true);
```

```
36    }
37  }
```

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

HtmlDialogueTest.java

```
1  package name.panitz.dialoguegui;
2  public class HtmlDialogueTest {
3    public static void main(String [] args){
4      new HtmlDialogue(new UpperConversion());
5    }
6  }
```

Die Anwendung in voller Aktion kann in Abbildung 4.1 bewundert werden.

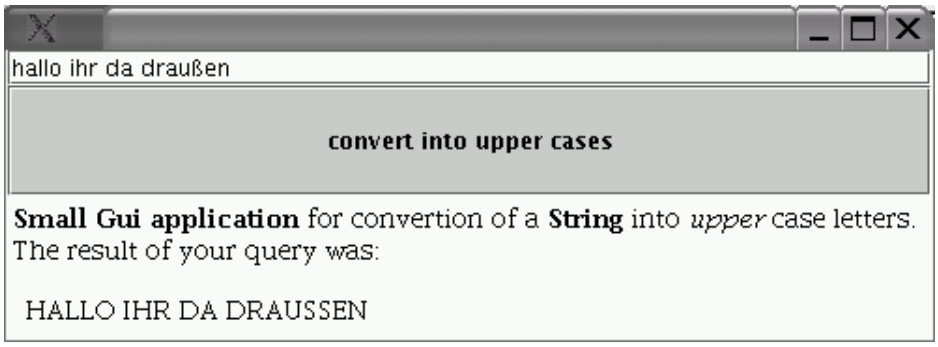


Abbildung 4.1: Ein Gui-Dialog mit Html-Ausgabe.

Semantische Einschränkungen für Schnittstellen

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

- Schnittstellen können nur Schnittstellen, nicht aber Klassen erweitern.
- Jede Methode einer Schnittstelle muß öffentlich sein, braucht also das Attribut `public`. Wenn dieses für eine Methode nicht deklariert ist, so wird Java dieses von selbst hinzufügen. Trotzdem müssen implementierende Klassen diese Methode dann als öffentlich deklarieren. Daher ist es besser, das Attribut `public` auch hinzuschreiben.
- Es gibt keine statischen Methoden in Schnittstellen.
- Jede Methode ist abstrakt, d.h. hat keinen Rumpf. Man kann dieses noch zusätzlich deutlich machen, indem man das Attribut `abstract` für die Methode mit angibt.
- Felder einer Schnittstelle sind immer statisch, brauchen also das Attribut `static` und zusätzlich noch das Attribut `final`.

Iteratorschnittstellen

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

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

Die Funktionalität eines Iteratorobjekts läßt sich über eine Schnittstellendefinition gut beschreiben.[6]

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

IntIterator.java

```
1 package name.panitz.iterator;
2 public interface IntIterator{
3   public int next();
4   public boolean hasNext();
5 }
```

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

PrintIntIteratorElements.java

```
1 package name.panitz.iterator;
2 public class PrintIntIteratorElements{
3   static void print(IntIterator it){
4     while (it.hasNext()) System.out.print(it.next()+" ");
5   }
6 }
```

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

FromToStep.java

```
1  package name.panitz.iterator;
2  public class FromToStep implements IntIterator{
3    private int from;
4    private int to;
5    private int step;
6
7    public FromToStep(int from,int to,int step){
8      this.from=from;this.to=to;this.step=step;}
9
10   public boolean hasNext(){return from<=to;}
```

[6] Und das wird auch tatsächlich in Javas Standardklassen so gemacht.

```
  public int next(){
    int result = from;
    from=from+step;
    return result;
  }
}
```

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

FromTo.java

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

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

TestFromTo.java

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

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

Aufgabe 19 In dieser Aufgabe sollen Sie ein Programm schreiben, das nach und nach die Primzahlen ausgibt.

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

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

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

c) Übersetzen Sie ihren Code mit folgender Klasse:

```
package name.panitz.iterator;
class PrimIterator implements IntIterator{
```

```
3     private IntIterator sieb = new From(2);
4     public boolean hasNext(){return sieb.hasNext();}
5     public int next(){
6       int result = sieb.next();
7       sieb= new Sieb(result,sieb);
8       return result;
9     }
10  }
```

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

4.3.2 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.

Abstrakte Klassen und Schnittstellen

Abstrakte Klassen können auch Schnittstellen implementieren. Entgegen konkreter Klassen brauchen sie aber nicht alle Eigenschaften einer Schnittstelle zu implementieren. Erst eine konkrete Klasse muß alle abstrakten Eigenschaften, die sie auf irgendeinem Weg erbt, implementieren.

4.4 Ausnahme- und Fehlerbehandlung

Es gibt während des Ablaufs eines Programmes Situationen, die als Ausnahmen zum eigentlichen Programmablauf betrachtet werden können. 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.

4.4.1 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`.

4.4.2 Werfen von Ausnahmen

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

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

FirstThrow.java

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

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

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

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

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

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

NonThrow.java

```
1  package name.panitz.exceptions;
2  public class NonThrow {
3
4    public static int fakultät(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(fakultät(5));
12     System.out.println(fakultät(-3));
13     System.out.println(fakultät(4));
14   }
15 }
```

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

```
swe10:~> java name.panitz.exceptions.NonThrow
120

An irrecoverable stack overflow has occurred.
```

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

Folgende Klasse:

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

```
6   }
7 }
```

führt zu einem Übersetzungsfehler:

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

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

ThrowIndex.java

```
1 package name.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 name.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 name.panitz.exceptions.ThrowIndex.main(ThrowIndex.java:5)
swe10:~>
```

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

SecondThrow.java

```
1 package name.panitz.exceptions;
2
3 public class SecondThrow {
4   public static int fakultät(int n){
5     if (n==0) return 1;
6     if (n<0)
7      throw
```

```
8         new RuntimeException
9              ("negative Zahl für Fakultätsberechnung");
10      return n*fakultät(n-1);
11    }
12
13    public static void main(String [] args){
14      System.out.println(fakultät(5));
15      System.out.println(fakultät(-3));
16      System.out.println(fakultät(4));
17    }
18  }
```

Damit erhalten wir folgende Ausgabe:

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

4.4.3 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:

```
1  package name.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 folgendem Fehler:

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

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

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

FourthThrow.java

```
1  package name.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.

4.4.4 Eigene Ausnahmeklassen

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

NegativeNumberException.java

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

So kann unser Beispielprogramm jetzt unsere eigene Ausnahme werfen :

FifthThrow.java

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

4.4.5 Fangen von Ausnahmen

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

Syntax

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

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

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

Catch1.java

```
1  package name.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 alle Ausnahmen, die eventuell während ihrer Auswertung geworfen wurden, ab. Das Programm erzeugt folgende Ausgabe:

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

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

Granularität des Abfangens

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

Catch2.java

```
1   package name.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[7] erzeugt folgende Ausgabe auf dem Bildschirm:

```
swe10:~/fh/beispiele> java name.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 name.panitz.exceptions;
2  public class Catch3 {
3
4    public static int fakultät(int n)
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 {
```

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

```
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 Ausnahmen auch unterschiedliche Ausnahmebehandlungen vorsehen. Dieses geschieht einfach, dadurch, daß mehrere `catch`-Klauseln untereinander stehen.

Hierzu definieren wir uns zunächst eine weitere Ausnahmeklasse:

NumberTooLargeException.java

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

Jetzt können wir unterschiedliche Ausnahmen werfen und wieder fangen:

Catch4.java

```
1  package name.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 name.panitz.exceptions.Catch4
Fakultät von zu großer Zahl
Fakultät von negativer Zahl
24
swe10:~/fh/beispiele>
```

Zusammenspiel mit Rückgabewerten

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

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

Catch5.java

```
package name.panitz.exceptions;
public class Catch5 {

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

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

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

```

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

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

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

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

4.4.7 Schließlich und finally

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

Finally.java

```
class Finally {

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

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

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

Die `finally`-Klausel wird nicht nur ausgeführt, wenn keine Ausnahme geworfen oder eine Ausnahme gefangen wurde, sondern auch, wenn eine Ausnahme geworfen wurde, für die es keine `catch`-Klausel gibt[8].

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

MoreFinally.java

```
class MoreFinally {

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

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

}
```

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

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

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

4.4.8 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 werden, wird als Ausnahmefall behandelt. Statt in einer Schleife über eine Reihung die Indizes abzufragen, kann man auch versuchen, das Ende der Reihung über eine Ausnahme abzufangen:

CatchOutOfBounds.java

```
1  class CatchOutOfBounds{
2    public static void main(String [] args){
3      int[] xs = {1,2,3,4,5,6,5,5,2,4,5,7,4,3,2,2,5,6};
4      int result = 0;
5      try{
6        int i =0;
7        while (true) {result=result+xs[i];i=i+2;}
8      }catch (IndexOutOfBoundsException e){}
9      System.out.println(result);
10   }
11 }
```

Kapitel 5

Generische Programmierung

5.1 Generische Typen

Generische Typen wurden im JSR014 definiert. In der Expertengruppe des JSR014 war der Autor dieses Skripts zeitweilig als Stellvertreter der Software AG Mitglied. Die Software AG hatte mit der Programmiersprache Bolero bereits einen Compiler für generische Typen implementiert[Pan00]. Der Bolero Compiler generiert auch Java Byte Code. Von dem ersten Wunsch nach Generizität bis zur nun vorliegenden Javaversion 1.5 sind viele Jahre vergangen. Andere wichtige JSRs, die in Java 1.5 integriert werden, tragen bereits die Nummern 175 und 201. Hieran kann man schon erkennen, wie lange es gedauert hat, bis generische Typen in Java integriert wurden.

Interessierten Programmierern steht schon seit Mitte der 90er Jahre eine Javaerweiterung mit generischen Typen zur Verfügung. Unter den Namen Pizza [OW97] existiert eine Javaerweiterung, die nicht nur generische Typen, sondern auch algebraische Datentypen mit *pattern matching* und Funktionsobjekten zu Java hinzufügte. Unter den Namen *GJ* für *Generic Java* wurde eine allein auf generische Typen abgespeckte Version von *Pizza* publiziert. *GJ* ist tatsächlich der direkte Prototyp für Javas generische Typen. Die Expertenrunde des JSR014 hat *GJ* als Grundlage für die Spezifikation genommen und an den grundlegenden Prinzipien auch nichts mehr geändert.

5.1.1 Generische Klassen

Die Idee für generische Typen ist, eine Klasse zu schreiben, die für verschiedene Typen als Inhalt zu benutzen ist. Das geht bisher in Java, allerdings mit einem kleinen Nachteil. Versuchen wir einmal, in traditionellem Java eine Klasse zu schreiben, in der wir beliebige Objekte speichern können. Um beliebige Objekte speichern zu können, brauchen wir ein Feld, in dem Objekte jeden Typs gespeichert werden können. Dieses Feld muß daher den Typ `Object` erhalten:

OldBox.java

```
1 class OldBox {
2   Object contents;
3   OldBox(Object contents){this.contents=contents;}
4 }
```

Der Typ `Object` ist ein sehr unschöner Typ; denn mit ihm verlieren wir jegliche statische Typinformation. Wenn wir die Objekte der Klasse `OldBox` benutzen wollen, so verlieren wir sämtliche Typinformation über das in dieser Klasse abgespeicherte Objekt. Wenn wir auf das Feld `contents` zugreifen, so haben wir über das darin gespeicherte Objekte keine spezifische Information mehr. Um das Objekt weiter sinnvoll nutzen zu können, ist eine dynamische Typzusicherung durchzuführen:

UseOldBox.java

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

Wann immer wir mit dem Inhalt des Felds `contents` arbeiten wollen, ist die Typzusicherung während der Laufzeit durchzuführen. Die dynamische Typzusicherung kann zu einem Laufzeitfehler führen. So übersetzt das folgende Programm fehlerfrei, ergibt aber einen Laufzeitfehler:

UseOldBoxError.java

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

```
sep@linux:~/fh/java1.5/examples/src> javac UseOldBoxError.java
sep@linux:~/fh/java1.5/examples/src> java UseOldBoxError
Exception in thread "main" java.lang.ClassCastException
        at UseOldBoxError.main(UseOldBoxError.java:4)
sep@linux:~/fh/java1.5/examples/src>
```

Wie man sieht, verlieren wir Typsicherheit, sobald der Typ `Object` benutzt wird. Bestimmte Typfehler können nicht mehr statisch zur Übersetzungszeit, sondern erst dynamisch zur Laufzeit entdeckt werden.

Der Wunsch ist, Klassen zu schreiben, die genauso allgemein benutzbar sind wie die Klasse `OldBox` oben, aber trotzdem die statische Typsicherheit garantieren, indem sie nicht mit dem allgemeinen Typ `Object` arbeiten. Genau dieses leisten generische Klassen. Hierzu ersetzen wir in der obigen Klasse jedes Auftreten des Typs `Object` durch einen Variablennamen. Diese Variable ist eine Typvariable. Sie steht für einen beliebigen Typen. Dem Klassennamen fügen wir zusätzlich in der Klassendefinition in spitzen Klammern eingeschlossen hinzu, daß diese Klasse eine Typvariable benutzt. Wir erhalten somit aus der obigen Klasse `OldBox` folgende generische Klasse `Box`.

Box.java

```
class Box<elementType> {
  elementType contents;
```

```
  Box(elementType contents){this.contents=contents;}
}
```

Die Typvariable `elementType` ist als allquantifiziert zu verstehen. Für jeden Typ `elementType` können wir die Klasse `Box` benutzen. Man kann sich unsere Klasse `Box` analog zu einer realen Schachtel vorstellen: Beliebige Dinge können in die Schachtel gelegt werden. Betrachten wir dann allein die Schachtel von außen, können wir nicht mehr wissen, was für ein Objekt darin enthalten ist. Wenn wir viele Dinge in Schachteln packen, dann schreiben wir auf die Schachtel jeweils drauf, was in der entsprechenden Schachtel enthalten ist. Ansonsten würden wir schnell die Übersicht verlieren. Und genau das ermöglichen generische Klassen. Sobald wir ein konkretes Objekt der Klasse `Box` erzeugen wollen, müssen wir entscheiden, für welchen Inhalt wir eine Box brauchen. Dieses geschieht, indem in spitzen Klammern dem Klassennamen `Box` ein entsprechender Typ für den Inhalt angehängt wird. Wir erhalten dann z.B. den Typ `Box<String>`, um Strings in der Schachtel zu speichern, oder `Box<Integer>`, um Integerobjekte darin zu speichern:

UseBox.java

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

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

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

Wie man im obigen Beispiel sieht, fallen jetzt die dynamischen Typzusicherungen weg. Die Variablen `b1` und `b2` sind jetzt nicht einfach vom Typ `Box`, sondern vom Typ `Box<String>` respektive `Box<Integer>`.

Da wir mit generischen Typen keine Typzusicherungen mehr vorzunehmen brauchen, bekommen wir auch keine dynamischen Typfehler mehr. Der Laufzeitfehler, wie wir ihn ohne die generische Box hatten, wird jetzt bereits zur Übersetzungszeit entdeckt. Hierzu betrachte man das analoge Programm:

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

Die Übersetzung dieses Programms führt jetzt bereits zu einen statischen Typfehler:

```
sep@linux:~/fh/java1.5/examples/src> javac UseBoxError.java
UseBoxError.java:3: cannot find symbol
symbol  : constructor Box(java.lang.Integer)
location: class Box<java.lang.String>
    Box<String> b = new Box<String>(new Integer(42));
                    ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

Vererbung

Generische Typen sind ein Konzept, das orthogonal zur Objektorientierung ist. Von generischen Klassen lassen sich in gewohnter Weise Unterklassen definieren. Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein. So können wir unsere einfache Schachtelklasse erweitern, so daß wir zwei Objekte speichern können:

GPair.java

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

  bt snd;

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

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

UsePair.java

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

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

Wie man sieht kommen wir wieder ohne Typzusicherung aus. Es gibt keinen dynamischen Typcheck, der im Zweifelsfall zu einer Ausnahme führen könnte.

```
sep@linux:~/fh/java1.5/examples/classes> java UsePair
(hallo,40)
HALLO
42
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können auch eine Unterklasse bilden, indem wir mehrere Typvariablen zusammenfassen. Wenn wir uniforme Paare haben wollen, die zwei Objekte gleichen Typs speichern, können wir hierfür eine spezielle Paarklasse definieren.

UniPair.java

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

Da beide gespeicherten Objekte jeweils vom gleichen Typ sind, konnten wir jetzt eine Methode schreiben, in der diese beiden Objekte ihren Platz tauschen. Wie man sieht, sind Typvariablen ebenso wie unsere bisherigen Typen zu benutzen. Sie können als Typ für lokale Variablen oder Parameter genutzt werden.

UseUniPair.java

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

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

Wie man bei der Benutzung der uniformen Paare sieht, gibt man jetzt natürlich nur noch einen konkreten Typ für die Typvariablen an. Die Klasse `UniPair` hat ja nur eine Typvariable.

```
sep@linux:~/fh/java1.5/examples/classes> java UseUniPair
(welt,hallo)
(hallo,welt)
sep@linux:~/fh/java1.5/examples/classes>
```

Wir können aber auch Unterklassen einer generischen Klasse bilden, die nicht mehr generisch ist. Dann leiten wir für eine ganz spezifische Instanz der Oberklasse ab. So läßt sich z.B. die Klasse `Box` zu einer Klasse erweitern, in der nur noch Stringobjekte verpackt werden können:

StringBox.java

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

Diese Klasse kann nun vollkommen ohne spitze Klammern benutzt werden:

UseStringBox.java

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

Einschränken der Typvariablen

Bisher standen in allen Beispielen die Typvariablen einer generischen Klasse für jeden beliebigen Objekttypen. Hier erlaubt Java uns, Einschränkungen zu machen. Es kann eingeschränkt werden, daß eine Typvariable nicht für alle Typen ersetzt werden darf, sondern nur für bestimmte Typen.

Versuchen wir einmal, eine Klasse zu schreiben, die auch wieder der Klasse `Box` entspricht, zusätzlich aber eine `set`-Methode hat und nur den neuen Wert in das entsprechende Objekt speichert, wenn es größer ist als das bereits gespeicherte Objekt. Hierzu müssen die zu speichernden Objekte in einer Ordnungsrelation vergleichbar sein, was in Java über die Implementierung der Schnittstelle `Comparable` ausgedrückt wird. Im herkömmlichen Java würden wir die Klasse wie folgt schreiben:

CollectMaxOld.java

```
class CollectMaxOld{
  private Comparable value;

  CollectMaxOld(Comparable x){value=x;}

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

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

Die Klasse `CollectMaxOld` ist in der Lage, beliebige Objekte, die die Schnittstelle `Comparable` implementieren, zu speichern. Wir haben wieder dasselbe Problem wie in der Klasse `OldBox`: Greifen wir auf das gespeicherte Objekt mit der Methode `getValue` erneut zu, wissen wir nicht mehr den genauen Typ dieses Objekts und müssen eventuell eine dynamische Typzusicherung durchführen, die zu Laufzeitfehlern führen kann.

Javas generische Typen können dieses Problem beheben. In gleicher Weise, wie wir die Klasse `Box` aus der Klasse `OldBox` erhalten haben, indem wir den allgemeinen Typ `Object` durch eine Typvariable ersetzt haben, ersetzen wir jetzt den Typ `Comparable` durch eine Typvariable, geben aber zusätzlich an, daß diese Variable für alle Typen steht, die die Untertypen der Schnittstelle `Comparable` sind. Dieses wird durch eine zusätzliche `extends`-Klausel für die Typvariable angegeben. Wir erhalten somit eine generische Klasse `CollectMax`:

CollectMax.java

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

```
3
4     CollectMax(elementType x){value=x;}
5
6     void setValue(elementType x){
7       if (value.compareTo(x)<0) value=x;
8     }
9
10    elementType getValue(){return value;}
11  }
```

Für die Benutzung diese Klasse ist jetzt für jede konkrete Instanz der konkrete Typ des gespeicherten Objekts anzugeben. Die Methode `getValue` liefert als Rückgabetyp nicht ein allgemeines Objekt des Typs `Comparable`, sondern exakt ein Objekt des Instanzstyps.

UseCollectMax.java

```
1  class UseCollectMax {
2    public static void main(String [] _){
3      CollectMax<String> cm = new CollectMax<String>("Brecht");
4      cm.setValue("Calderon");
5      cm.setValue("Horvath");
6      cm.setValue("Shakespeare");
7      cm.setValue("Schimmelpfennig");
8      System.out.println(cm.getValue().toUpperCase());
9    }
10 }
```

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

5.1.2 Generische Schnittstellen

Generische Typen erlauben es, den Typ `Object` in Typsignaturen zu eleminieren. Der Typ `Object` ist als schlecht anzusehen, denn er ist gleichbedeutend damit, daß keine Information über einen konkreten Typ während der Übersetzungszeit zur Verfügung steht. Im herkömmlichen Java ist in APIs von Bibliotheken der Typ `Object` allgegenwärtig. Sogar in der Klasse `Object` selbst begegnet er uns in Signaturen. Die Methode `equals` hat einen Parameter vom Typ `Object`, d.h. prinzipiell kann ein Objekt mit Objekten jeden beliebigen Typs verglichen werden. Zumeist will man aber nur gleiche Typen miteinander vergleichen. In diesem Abschnitt werden wir sehen, daß generische Typen es uns erlauben, allgemein eine Gleichheitsmethode zu definieren, in der nur Objekte gleichen Typs miteinander verglichen werden können. Hierzu werden wir eine generische Schnittstelle definieren.

Generische Typen erweitern sich ohne Umstände auf Schnittstellen. Im Vergleich zu generischen Klassen ist nichts Neues zu lernen. Syntax und Benutzung funktionieren auf die gleiche Weise.

Äpfel mit Birnen vergleichen

Um zu realisieren, daß nur noch Objekte gleichen Typs miteinander verglichen werden können, definieren wir eine Gleichheitsschnitstelle. In ihr wird eine Methode spezifiziert, die für die

Gleichheit stehen soll. Die Schnittstelle ist generisch über den Typen, mit dem vergleichen werden soll.

EQ.java

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

Jetzt können wir für jede Klasse nicht nur bestimmen, daß sie die Gleichheit implementieren soll, sondern auch, mit welchen Typen Objekte unserer Klasse verglichen werden sollen. Schreiben wir hierzu eine Klasse `Apfel`. Die Klasse Apfel soll die Gleichheit auf sich selbst implementieren. Wir wollen nur Äpfel mit Äpfeln vergleichen können. Daher definieren wir in der `implements`-Klausel, daß wir `EQ<Apfel>` implementieren wollen. Dann müssen wir auch die Methode `eq` implementieren, und zwar mit dem Typ `Apfel` als Parametertyp:

Apfel.java

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

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

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

Jetzt können wir Äpfel mit Äpfeln vergleichen:

TestEQ.java

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

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

Birne.java

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

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

  public boolean eq(Birne other){
```

```
    return this.typ.equals(other.typ);
  }
}
```

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

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

Wir bekommen die verständliche Fehlermeldung, daß die Gleichheit auf Äpfel nicht für einen Birnenparameter aufgerufen werden kann.

```
./TestEQError.java:6: eq(Apfel) in Apfel cannot be applied to (Birne)
    System.out.println(a.eq(b));
                           ^
1 error
```

Wahrscheinlich ist es jedem erfahrenden Javaprogrammierer schon einmal passiert, daß er zwei Objekte verglichen hat, die er gar nicht vergleichen wollte. Da der statische Typcheck solche Fehler nicht erkennen kann, denn die Methode `equals` läßt jedes Objekt als Parameter zu, sind solche Fehler mitunter schwer zu lokalisieren.

Der statische Typcheck stellt auch sicher, daß eine generische Schnittstelle mit der korrekten Signatur implementiert wird. Der Versuch, eine Birneklasse zu schreiben, die eine Gleichheit mit Äpfeln implementieren soll, dann aber die Methode `eq` mit dem Parametertyp `Birne` zu implementieren, führt ebenfalls zu einer Fehlermeldung:

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

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

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

Wir bekommen folgende Fehlermeldung:

```
sep@linux:~/fh/java1.5/examples/src> javac BirneError.java
BirneError.java:1: BirneError is not abstract and does not override abstract method eq(Apfel) in EQ
```

```
class BirneError implements EQ<Apfel>{
^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

5.1.3 Kovarianz gegen Kontravarianz

Gegeben seien zwei Typen `A` und `B`. Der Typ `A` soll Untertyp des Typs `B` sein, also entweder ist `A` eine Unterklasse der Klasse `B` oder `A` implementiert die Schnittstelle `B` oder die Schnittstelle `A` erweitert die Schnittstelle `B`. Für diese Subtyprelation schreiben wir das Relationssymbol $\sqsubseteq$. Es gelte also $A \sqsubseteq B$. Gilt damit auch für einen generischen Typ `C`: `C<A>`$\sqsubseteq$`C<B>`?

Man mag geneigt sein, zu sagen ja. Probieren wir dieses einmal aus:

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

Der Javaübersetzer weist dieses Programm zurück:

```
sep@linux:~/fh/java1.5/examples/src> javac Kontra.java
Kontra.java:4: incompatible types
found   : Box<java.lang.String>
required: Box<java.lang.Object>
    Box<Object> b  = new Box<String>("hello");
                     ^
1 error
sep@linux:~/fh/java1.5/examples/src>
```

Eine `Box<String>` ist keine `Box<Object>`. Der Grund für diese Entwurfsentscheidung liegt darin, daß bestimmte Laufzeitfehler vermieden werden sollen. Betrachtet man ein Objekt des Typs `Box<String>` über eine Referenz des Typs `Box<Object>`, dann können in dem Feld `contents` beliebige Objekte gespeichert werden. Die Referenz über den Typ `Box<String>` geht aber davon aus, daß in `contents` nur Stringobjekte gespeichert werden.

Man vergegenwärtige sich nochmals, daß Reihungen in Java sich hier anders verhalten. Bei Reihungen ist die entsprechende Zuweisung erlaubt. Eine Reihung von Stringobjekten darf einer Reihung beliebiger Objekte zugewiesen werden. Dann kann es bei der Benutzung der Reihung von Objekten zu einen Laufzeitfehler kommen.

Ko.java

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

Das obige Programm führt zu folgendem Laufzeitfehler:

```
sep@linux:~/fh/java1.5/examples/classes> java Ko
Exception in thread "main" java.lang.ArrayStoreException
        at Ko.main(Ko.java:5)
sep@linux:~/fh/java1.5/examples/classes>
```

Für generische Typen wurde ein solcher Fehler durch die Strenge des statischen Typchecks bereits ausgeschlossen.

5.1.4 Sammlungsklassen

Die Paradeanwendung für generische Typen sind natürlich Sammlungsklassen, also die Klassen für Listen und Mengen, wie sie im Paket `java.util` definiert sind. Mit der Version 1.5 von Java finden sich generische Versionen der bekannten Sammlungsklassen. Jetzt kann man angeben, was für einen Typ die Elemente einer Sammlung genau haben sollen.

ListTest.java

```
1  import java.util.*;
2  import java.util.List;
3  class ListTest{
4    public static void main(String [] _){
5      List<String> xs = new ArrayList<String>();
6      xs.add("Schimmelpfennig");
7      xs.add("Shakespeare");
8      xs.add("Horvath");
9      xs.add("Brecht");
10     String x2 = xs.get(1);
11     System.out.println(xs);
12   }
13 }
```

Aus Kompatibilitätsgründen mit bestehendem Code können generische Klassen auch weiterhin ohne konkrete Angabe des Typparameters benutzt werden. Während der Übersetzung wird in diesen Fällen eine Warnung ausgegeben.

WarnList.java

```
1  import java.util.*;
2  import java.util.List;
3  class WarnTest{
4    public static void main(String [] _){
5      List xs = new ArrayList<String>();
6      xs.add("Schimmelpfennig");
7      xs.add("Shakespeare");
8      xs.add("Horvath");
9      xs.add("Brecht");
10     String x2 = (String)xs.get(1);
11     System.out.println(xs);
12   }
13 }
```

Obiges Programm übersetzt mit folgender Warnung:

```
sep@linux:~/fh/java1.5/examples/src> javac WarnList.java
Note: WarnList.java uses unchecked or unsafe operations.
Note: Recompile with -warnunchecked for details.
sep@linux:~/fh/java1.5/examples/src>
```

5.1.5 Generische Methoden

Bisher haben wir generische Typen für Klassen und Schnittstellen betrachtet. Generische Typen sind aber nicht an einen objektorientierten Kontext gebunden, sondern basieren ganz im Gegenteil auf dem Milner-Typsystem, das funktionale Sprachen, die nicht objektorientiert sind, benutzen. In Java verläßt man den objektorientierten Kontext in statischen Methoden. Statische Methoden sind nicht an ein Objekt gebunden. Auch statische Methoden lassen sich generisch in Java definieren. Hierzu ist vor der Methodensignatur in spitzen Klammern eine Liste der für die statische Methode benutzten Typvariablen anzugeben.

Eine sehr einfache statische generische Methode ist eine *trace*-Methode, die ein beliebiges Objekt erhält, dieses Objekt auf der Konsole ausgibt und als Ergebnis genau das erhaltene Objekt unverändert wieder zurückgibt. Diese Methode `trace` hat für alle Typen den gleichen Code und kann daher entsprechend generisch geschrieben werden:

Trace.java

```
1  class Trace {
2
3    static <elementType>  elementType trace(elementType x){
4      System.out.println(x);
5      return x;
6    }
7
8    public static void main(String [] _){
9      String x = trace ((trace ("hallo")
10                        +trace( " welt")).toUpperCase());
11
12     Integer y = trace (new Integer(40+2));
13   }
14 }
```

In diesem Beispiel ist zu erkennen, daß der Typchecker eine kleine Typinferenz vornimmt. Bei der Anwendung der Methode `trace` ist nicht anzugeben, mit welchen Typ die Typvariable `elementType` zu instanziieren ist. Diese Information inferriert der Typchecker automatisch aus dem Typ des Arguments.

5.2 Iteration

Typischer Weise wird in einem Programm über die Elemente eines Sammlungstyp iteriert oder über alle Elemente einer Reihung. Hierzu kennt Java verschiedene Schleifenkonstrukte. Leider kannte Java bisher kein eigenes Schleifenkonstrukt, das bequem eine Iteration über die Elemente

einer Sammlung ausdrücken konnte. Die Schleifensteuerung mußte bisher immer explizit ausprogrammiert werden. Hierbei können Programmierfehler auftreten, die insbesondere dazu führen können, daß eine Schleife nicht terminiert. Ein Schleifenkonstrukt, das garantiert terminiert, kannte Java bisher nicht.

Beispiel:
In diesen Beispiel finden sich die zwei wahrscheinlich am häufigsten programmierten Schleifentypen. Einmal iterieren wir über alle Elemente einer Reihung und einmal iterieren wir mittels eines Iteratorobjekts über alle Elemente eines Sammlungsobjekts:

OldIteration.java

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

class OldIteration{

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

    for (int i= 0;i<ar.length;i++){
      final String s = ar[i];
      xs.add(s);
    }

    for (Iterator it=xs.iterator();it.hasNext();){
      final String s = (String)it.next();
      System.out.println(s.toUpperCase());
    }
  }
}
```

Die Codemenge zur Schleifensteuerung ist gewaltig und übersteigt hier sogar die eigentliche Anwendungslogik.

Mit Java 1.5 gibt es endlich eine Möglichkeit, zu sagen, mache für alle Elemente im nachfolgenden Sammlungsobjekt etwas. Eine solche Syntax ist jetzt in Java integriert. Sie hat die Form:

`for` (*Type identifier* : *expr*){*body*}

Zu lesen ist dieses Kosntrukt als: für jedes *identifier* des Typs *Type* in *expr* führe *body* aus.[1]

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

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

NewIteration.java

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

class NewIteration{

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

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

Der gesamte Code zur Schleifensteuerung ist entfallen. Zusätzlich ist garantiert, daß für endliche Sammlungsiteratoren auch die Schleife terminiert.

Wie man sieht, ergänzen sich generische Typen und die neue `for`-Schleife.

5.2.1 Die neuen Schnittstellen Iterable

Beispiel:

ReaderIterator.java

```
package sep.util.io;

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


public class ReaderIterator
              implements Iterable<Character>
                         ,Iterator<Character>{
  private Reader reader;
  private int n;
  public ReaderIterator(Reader r){
    reader=new BufferedReader(r);
    try{n=reader.read();
    }catch(IOException _){n=-1;}
  }
  public Character next(){
    Character result = new Character((char)n);
```

```
    try{n=reader.read();
    }catch(IOException _){n=-1;}
    return result;
  }

  public boolean hasNext(){
    return n!=-1;
  }

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

  public Iterator<Character> iterator(){return this;}
}
```

TestReaderIterator.java

```
import sep.util.io.ReaderIterator;
import java.io.FileReader;

class TestReaderIterator {
  public static void main(String [] args) throws Exception{
      Iterable<Character> it
        =new ReaderIterator(new FileReader(args[0]));
      for (Character c:it){
      System.out.print(c);
    }
  }
}
```

Beispiel:
Ein abschließendes kleines Beispiel für generische Sammlungsklassen und die neue `for`-Schleife. Die folgende Klasse stellt Methoden zur Verfügung, um einen String in eine Liste von Wörtern zu spalten und umgekehrt aus einer Liste von Wörtern wieder einen String zu bilden:

TextUtils.java

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

class TextUtils {

  static List<String> words (String s){
    final List<String> result = new ArrayList<String>();

    StringBuffer currentWord = new StringBuffer();

    for (char c:s.toCharArray()){
      if (Character.isWhitespace(c)){
        final String newWord = currentWord.toString().trim();
```

```
14          if(newWord.length()>0){
15            result.add(newWord);
16            currentWord=new StringBuffer();
17          }
18        }else{currentWord.append(c);}
19      }
20      return result;
21    }
22
23    static String unwords(List<String> xs){
24      StringBuffer result=new StringBuffer();
25      for (String x:xs) result.append(" "+x);
26      return result.toString().trim();
27    }
28
29    public static void main(String []_){
30      List<String> xs = words("  the   world  is my Oyster  ");
31
32      for (String x:xs) System.out.println(x);
33
34      System.out.println(unwords(xs));
35    }
36  }
```

Kapitel 6

Datentypen und Algorithmen

Die bisher kennengelernten Javakonstrukte bilden einen soliden Kern, der genügend programmiertechnische Mittel zur Verfügung stellt, um die gängigsten Konzepte der Informatik umzusetzen. In diesem Kapitel werden wir keine neuen Javakonstrukte kennenlernen, sondern mit den bisher bekannten Mitteln die häufigsten in der Informatik gebräuchlichen Datentypen und auf ihnen anzuwendende Algorithmen erkunden.

6.1 Listen

Eine der häufigsten Datenstrukturen in der Programmierung sind Sammlungstypen. In fast jedem nichttrivialen Programm wird es Punkte geben, an denen eine Sammlung mehrerer Daten gleichen Typs anzulegen sind. Eine der einfachsten Strukturen, um Sammlungen anzulegen, sind Listen. Da Sammlungstypen oft gebraucht werden, stellt Java entsprechende Klassen als Standardklassen zur Verfügung. Bevor wir uns aber diesen bereits vorhandenen Klassen zuwenden, wollen wir in diesem Kapitel Listen selbst spezifizieren und programmieren.

6.1.1 Formale Spezifikation

Wir werden Listen als abstrakten Datentyp formal spezifizieren. Ein abstrakter Datentyp (ADT) wird spezifiziert über eine endliche Menge von Methoden. Hierzu wird spezifiziert, auf welche Weise Daten eines ADT konstruiert werden können. Dazu werden entsprechende Konstruktormethoden spezifiziert. Dann wird eine Menge von Funktionen definiert, die wieder Teile aus den konstruierten Daten selektieren können. Schließlich werden noch Testmethoden spezifiziert, die angeben, mit welchem Konstruktor ein Datum erzeugt wurde.

Der Zusammenhang zwischen Konstruktoren und Selektoren sowie zwischen den Konstruktoren und den Testmethoden wird in Form von Gleichungen spezifiziert.

Der Trick, der angewendet wird, um abstrakte Datentypen wie Listen zu spezifizieren, ist die Rekursion. Das Hinzufügen eines weiteren Elements zu einer Liste wird dabei als das Konstruieren einer neuen Liste aus der Ursprungsliste und einem weiteren Element betrachtet. Mit dieser Betrachtungsweise haben Listen eine rekursive Struktur: eine Liste besteht aus dem zuletzt vorne angehängten neuen Element, dem sogenannten Kopf der Liste, und aus der alten Teilliste, an die dieses Element angehängt wurde, dem sogenannten Schwanz der Liste. Wie bei jeder

rekursiven Struktur bedarf es eines Anfangs der Definition. Im Falle von Listen wird dieses durch die Konstruktion einer leeren Liste spezifiziert.[1]

Konstruktoren

Abstrakte Datentypen wie Listen lassen sich durch ihre Konstruktoren spezifizieren. Die Konstruktoren geben an, wie Daten des entsprechenden Typs konstruiert werden können. In dem Fall von Listen bedarf es nach den obigen Überlegungen zweier Konstruktoren:

- einem Konstruktor für neue Listen, die noch leer sind.
- einem Konstruktor, der aus einem Element und einer bereits bestehenden Liste eine neue Liste konstruiert, indem an die Ursprungsliste das Element angehängt wird.

Wir benutzen in der Spezifikation eine mathematische Notation der Typen von Konstruktoren.[2] Dem Namen des Konstruktors folgt dabei mit einem Doppelpunkt abgetrennt der Typ. Der Ergebnistyp wird von den Parametertypen mit einem Pfeil getrennt.

Somit lassen sich die Typen der zwei Konstruktoren für Listen wie folgt spezifizieren:

- Empty: () →**List**
- Cons: (**Object**,**List**) →**List**

Selektoren

Die Selektoren können wieder auf die einzelnen Bestandteile der Konstruktion zurückgreifen. Der Konstruktor **Cons** hat zwei Parameter. Für **Cons**-Listen werden zwei Selektoren spezifiziert, die jeweils einen dieser beiden Parameter wieder aus der Liste selektieren. Die Namen dieser beiden Selektoren sind traditioneller Weise *head* und *tail*.

- head: **(List)→Object**
- tail: **(List)→List**

Der funktionale Zusammenhang von Selektoren und Konstruktoren läßt sich durch folgende Gleichungen spezifizieren:

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

[1] Man vergleiche es mit der Definition der natürlichen Zahlen: die 0 entspricht der leeren Liste, der Schritt von n nach $n + 1$ dem Hinzufügen eines neuen Elements zu einer Liste.

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

Testmethoden

Um für Listen Algorithmen umzusetzen, ist es notwendig, unterscheiden zu können, welche Art der beiden Listen vorliegt: die leere Liste oder eine **Cons**-Liste. Hierzu bedarf es noch einer Testmethode, die mit einem bool'schen Wert als Ergebnis angibt, ob es sich bei der Eingabeliste um die leere Liste handelte oder nicht. Wir wollen diese Testmethode *isEmpty* nennen. Sie hat folgenden Typ:

- isEmpty: **List→boolean**

Das funktionale Verhalten der Testmethode läßt sich durch folgende zwei Gleichungen spezifizieren:

$$\begin{aligned} isEmpty(Empty()) &= true \\ isEmpty(Cons(x, xs)) &= false \end{aligned}$$

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

Listenalgorithmen

Allein diese fünf Funktionen beschreiben den ADT der Listen. Wir können aufgrund dieser Spezifikation Algorithmen für Listen schreiben.

Länge Und ebenso läßt sich durch zwei Gleichungen spezifizieren, was die Länge einer Liste ist:

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

Mit Hilfe dieser Gleichungen läßt sich jetzt schrittweise die Berechnung einer Listenlänge auf Listen durchführen. Hierzu benutzen wir die Gleichungen als Ersetzungsregeln. Wenn ein Unterausdruck in der Form der linken Seite einer Gleichung gefunden wird, so kann diese durch die entsprechende rechte Seite ersetzt werden. Man spricht bei so einem Ersetzungsschritt von einem Reduktionsschritt.

Beispiel:
Wir errechnen in diesem Beispiel die Länge einer Liste, indem wir die obigen Gleichungen zum Reduzieren auf die Liste anwenden:

$$\begin{aligned} & length(\mathbf{Cons}(a, \mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))))) \\ \rightarrow\ & 1 + length(\mathbf{Cons}(b, \mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + length(\mathbf{Cons}(c, \mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + (1 + length(\mathbf{Empty}()))) \\ \rightarrow\ & 1 + (1 + (1 + 0)) \\ \rightarrow\ & 1 + (1 + 1) \\ \rightarrow\ & 1 + 2 \\ \rightarrow\ & 3 \end{aligned}$$

Letztes Listenelement Wir können mit einfachen Gleichungen spezifizieren, was wir unter dem letzten Element einer Liste verstehen.

$$\begin{aligned} last(Cons(x, Empty())) &= x \\ last(Cons(x, xs))) &= last(xs) \end{aligned}$$

Auch die Funktion `last` können wir von Hand auf einer Beispielliste einmal per Reduktion ausprobieren:

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

Listenkonkatenation Die folgenden Gleichungen spezifizieren, wie zwei Listen aneinandergehängt werden:

$$\begin{aligned} concat(Empty(), ys) &= ys \\ concat(Cons(x, xs), ys) &= Cons(x, concat(xs, ys)) \end{aligned}$$

Auch diese Funktion läßt sich beispielhaft mit der Reduktion einmal durchrechnen:

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

Schachtel- und Zeiger-Darstellung

Listen lassen sich auch sehr schön graphisch visualisieren. Hierzu wird jede Liste durch eine Schachtel mit zwei Feldern dargestellt. Von diesen beiden Feldern gehen Pfeile aus. Der erste Pfeil zeigt auf das erste Element der Liste, dem `head`, der zweite Pfeil zeigt auf die Schachtel, die für den Restliste steht dem *tail*. Wenn eine Liste leer ist, so gehen keine Pfeile von der Schachtel aus, die sie repräsentiert.

Die Liste *Cons(a,Cons(b,Cons(c,Empty())))* hat somit die Schachtel- und Zeigerr- Darstellung aus Abbildung 6.1.

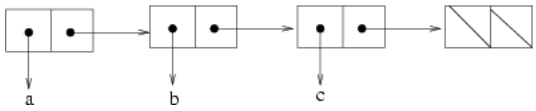


Abbildung 6.1: Schachtel Zeiger Darstellung einer dreielementigen Liste.

In der Schachtel- und Zeiger-Darstellung läßt sich sehr gut verfolgen, wie bestimmte Algorithmen auf Listen dynamisch arbeiten. Wir können die schrittweise Reduktion der Methode `concat` in der Schachtel- und Zeiger-Darstellung gut nachvollziehen:

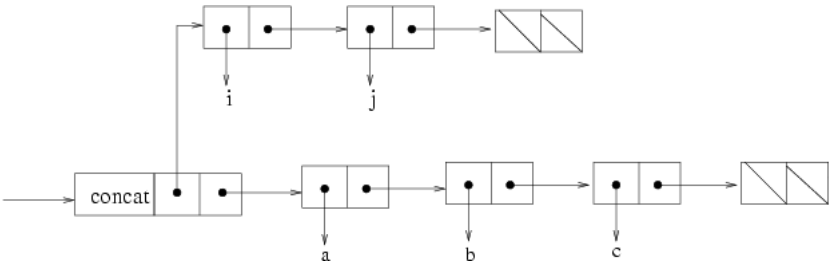


Abbildung 6.2: Schachtel Zeiger Darstellung der Funktionsanwendung von concat auf zwei Listen.

Abbildung 6.2 zeigt die Ausgangssituation. Zwei Listen sind dargestellt. Von einer Schachtel, die wir als die Schachtel der Funktionsanwendung von `concat` markiert haben, gehen zwei Zeiger aus. Der erste auf das erste Argument, der zweite auf das zweite Argument der Funktionsanwendung.

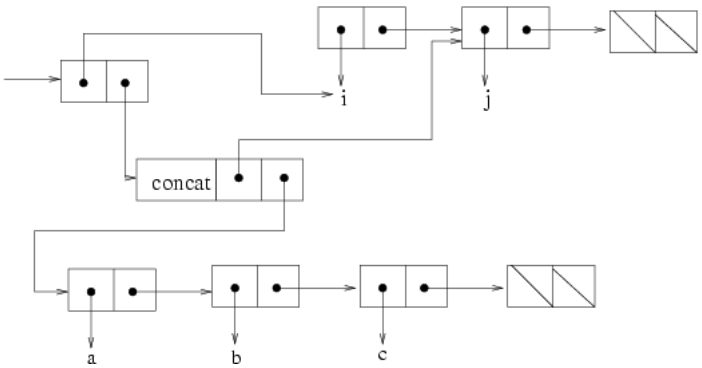


Abbildung 6.3: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

Abbildung 6.3 zeigt die Situation, nachdem die Funktion `concat` einmal reduziert wurde. Ein neuer Listenknoten wurde erzeugt. Dieser zeigt auf das erste Element der ursprünglich ersten Argumentliste. Der zweite zeigt auf den rekursiven Aufruf der Funktion `concat`, diesmal mit der Schwanzliste des ursprünglich ersten Arguments.

Abbildung 6.4 zeigt die Situation nach dem zweiten Reduktionsschritt. Ein weiterer neuer Listenknoten ist entstanden und ein neuer Knoten für den rekursiven Aufruf ist entstanden.

Abbildung 6.5 zeigt die endgültige Situation. Der letzte rekursive Aufruf von `concat` hatte als erstes Argument eine leere Liste. Deshalb wurde kein neuer Listenknoten erzeugt, sondern lediglich der Knoten für die Funktionsanwendung gelöscht. Man beachte, daß die beiden ursprünglichen Listen noch vollständig erhalten sind. Sie wurden nicht gelöscht. Die erste Argumentliste wurde quasi kopiert. Die zweite Argumentliste teilen sich gewisser Maßen die neue Ergebnisliste der Funktionsanwendung und die zweite ursprüngliche Argumentliste.

6.1.2 Modellierung

Java kennt keine direkte Unterstützung für ADTs, die nach obigen Prinzip spezifiziert werden. Es gibt jedoch eine Javaerweiterung namens *Pizza*[OW97], die eine solche Unterstützung einge-

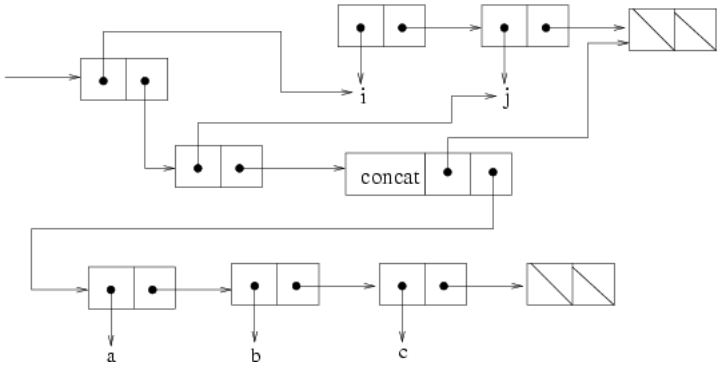


Abbildung 6.4: Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt.

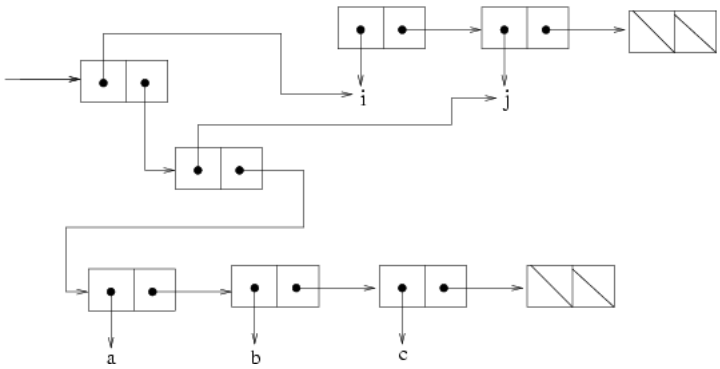


Abbildung 6.5: Schachtel Zeiger Darstellung des Ergebnisses nach der Reduktion.

baut hat. Da wir aber nicht auf *Pizza* zurückgreifen wollen, bleibt uns nichts anderes, als Listen in Java zu implementieren.

Nachdem wir im letzten Abschnitt formal spezifiziert haben, wie Listen konstruiert werden, wollen wir in diesem Abschnitt betrachten, wie diese Spezifikation geeignet mit unseren programmiersprachlichen Mitteln modelliert werden kann, d.h. wie viele und was für Klassen werden benötigt. Wir werden in den folgenden zwei Abschnitten zwei alternative Modellierungen der Listenstruktur angeben.

Modellierung als Klassenhierarchie

Laut Spezifikation gibt es zwei Arten von Listen: leere Listen und Listen mit einem Kopfelement und einer Schwanzliste. Es ist naheliegend, für diese zwei Arten von Listen je eine eigene Klasse bereitzustellen, jeweils eine Klasse für leere Listen und eine für **Cons**-Listen. Da beide Klassen zusammen einen Datentyp Liste bilden sollen, sehen wir eine gemeinsame Oberklasse dieser zwei Klassen vor. Wir erhalten die Klassenhierarchie in Abbildung: 6.6.

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

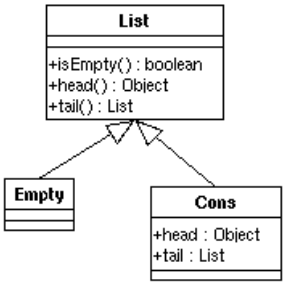


Abbildung 6.6: Modellierung von Listen mit drei Klassen.

Ergebnis zurückgeben.

In einer Klasse

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

6.1.3 Codierung

Die obigen beiden Modellierungen der Listen lassen sich jetzt direkt in Javacode umsetzen.

Implementierung als Klassenhierarchie

Die Oberklasse List Die zusammenfassende Oberklasse der Listenstruktur `List` stellt für die Listenmethoden `head`, `tail` und `isEmpty` jeweils eine prototypische Implementierung zur Verfügung. Die Methode `isEmpty` setzen wir in unserer Umsetzung standardmäßig auf `false`.

List.java

```
1 package name.panitz.data.list;
2 class List<a>{
3  boolean isEmpty(){return false;}
4  a head(){return null;}
5  List<a> tail(){return null;}
```

Die Methoden `tail` und `head` können nur für nichtleere Listen Objekte zurückgeben. Daher haben wir uns für die prototypischen Implementierung in der Klasse `List` dazu entschieden, den Wert `null` zurückzugeben. `null` steht in Java für das

Fehlen eines Objektes. Der Versuch, auf Felder und Methoden von `null` zuzugreifen, führt in Java zu einem Fehler.

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

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

Empty.java

```
package name.panitz.data.list;
class Empty<a> extends List<a> {
  Empty(){}
  boolean isEmpty(){return true;}
}
```

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

Cons.java

```
package name.panitz.data.list;
class Cons<a> extends List<a>{

  a hd;
  List<a> tl ;

  Cons(a x,List<a> xs){
    hd = x;
    tl = xs;
  }

  a head(){
    return hd;
  }

  List<a> tail(){
    return tl;
  }
```

Damit ist die Listenstruktur gemäß unserer formalen Spezifikation vollständig implementiert.

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

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

TestFirstList.java

```
package name.panitz.data.list;
class TestFirstList {
  public static void main(String [] args){

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

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

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

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

Tatsächlich bekommen wir für unsere Tests die erwartete Ausgabe:

```
sep@linux:~/fh/prog1/examples/classes> java name.panitz.data.list.TestFirstList
1. Element: friends
2. Element: romans
3. Element: countrymen
leere Liste nach drittem Element.
null
sep@linux:~/fh/prog1/examples/classes>
```

Implementierung als eine Klasse

In der zweiten Modellierung haben wir auf eine Klassenhierarchie verzichtet. Was wir in der ersten Modellierung durch die verschiedenen Klassen mit verschieden überschriebenen Methoden ausgedrückt haben, muß in der Modellierung in einer Klasse über ein bool'sches Feld vom Typ `boolean` ausgedrückt werden. Die beiden unterschiedlichen Konstruktoren setzen ein bool'sches Feld `empty`, um für das neu konstruierte Objekt zu markieren, mit welchem Konstruktor es konstruiert wurde. Die Konstruktoren setzen entsprechend die Felder, so daß die Selektoren diese nur auszugeben brauchen.

Li.java

```
package name.panitz.data.list;
class Li<a> implements Iterable<a>{
  boolean empty = true;
```

```
  a  hd;
  Li<a> tl;

  Li(){}

  Li(a x,Li<a> xs){
   hd = x;
   tl = xs;
   empty = false;
  }

  boolean isEmpty() {return empty;}
  a  head(){return hd;}
  Li<a>      tail(){return tl;}

  public java.util.Iterator<a> iterator(){
    final Li<a>  itout = this;
    return new java.util.Iterator<a>(){
      Li<a>  it = itout;
      public boolean hasNext(){return !it.isEmpty();}
      public a next(){a result=it.head();it=it.tail();return result;}
      public void remove(){throw new UnsupportedOperationException();}
    };
  }
```

Zum Testen dieser Listenimplementierung sind nur die Konstruktoraufrufe bei der Listenkonstruktion zu ändern. Da wir nur noch eine Klasse haben, gibt es keine zwei Konstruktoren mit unterschiedlichen Namen, sondern beide haben denselben Namen:

TestFirstLi.java

```

package name.panitz.data.list;
class TestFirstLi {
  public static void main(String [] args){

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

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

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

```
21
22      //Ausgabe eines null Wertes
23      System.out.println(xs.tail().tail().tail().head());
24    }
25
26  }
```

Wie man aber sieht, ändert sich an der Benutzung von Listen nichts im Vergleich zu der Modellierung mittels einer Klassenhierarchie. Die Ausgabe ist ein und dieselbe für beide Implementierungen:

```
sep@linux:~/fh/prog1/examples/classes> java TestFirstLi
1. Element: friends
2. Element: romans
3. Element: countrymen
leere Liste nach drittem Element.
null
sep@linux:~/fh/prog1/examples/classes>
```

Konstruktion langer Listen Die Konstruktion der dreielementigen Liste oben war schon recht umständlich. Jede Teilliste mußte explizit durch einen Konstruktoraufruf generiert werden. Die variable Argumentanzahl für Methoden erlaubt es, einen Konstruktor zu definieren, der die einzelnen Elemente der zu erzeugenen Liste als Argumente erhält.

Li.java

```
27    public Li(a... xs){
28      this(0,xs);
29    }
30
31    private Li(int i,a... xs){
32      if (i<xs.length) {
33        empty=false;
34        hd=xs[i];
35        tl = new Li<a>(i+1,xs);
36      }
37    }
```

Jetzt lassen sich bequem auch längere Listen durch einen einzigen Konstruktoraufruf erzeugen:

TestFirstLi2.java

```
1
2   package name.panitz.data.list;
3   class TestFirstLi2 {
4     public static void main(String [] args){
5
6       //Konstruktion einer Testliste
7       Li<String> xs
8        = new Li<String>
9            ("Friends", "Romans", "countrymen", "lend", "me","your", "ears"
10           ,"I","come","to","bury","Caesar","not","to","praise","him");
```

```
11
12      //Zugriffe auf einzelne Elemente
13      System.out.println("1. Element: "+xs.head());
14      System.out.println("2. Element: "+xs.tail().head());
15      System.out.println("3. Element: "+xs.tail().tail().head());
16
17      //Test für die Methode isEmpty()
18      if (!xs.tail().tail().tail().isEmpty()){
19        System.out.println("keine leere Liste nach drittem Element.");
20      }
21
22      //Ausgabe 4. Wertes
23      System.out.println(xs.tail().tail().tail().head());
24    }
25
26  }
```

6.1.4 Methoden für Listen

Mit der formalen Spezifikation und schließlich Implementierung von Listen haben wir eine Abstraktionsebene eingeführt. Listen sind für uns Objekte, für die wir genau die zwei Konstruktormethoden, zwei Selektormethoden und eine Testmethode zur Verfügung haben. Unter Benutzung dieser fünf Eigenschaften der Listenklasse können wir jetzt beliebige Algorithmen auf Listen definieren und umsetzen.

Länge

Eine interessante Frage bezüglich Listen ist die nach ihrer Länge. Wir können die Länge einer Liste berechnen, indem wir durchzählen, aus wievielen **Cons**-Listen eine Liste besteht. Wir haben bereits die Funktion *length* durch folgende zwei Gleichungen spezifiziert.

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

Diese beiden Gleichungen lassen sich direkt in Javacode umsetzen. Die zwei Gleichungen ergeben genau zwei durch eine `if`-Bedingung zu unterscheidende Fälle in der Implementierung. Wir können die Klassen `List` und `Li` um folgende Methode `length` ergänzen:

Li.java

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

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

TestLength.java

```
package name.panitz.data.list;
class TestLength {
  static  Li<String> XS
   =  new Li<String>("friends"
     ,new Li<String>("romans"
     ,new Li<String>("countrymen"
     ,new Li<String>()))));

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

Und wie wir bereits schon durch Reduktion dieser Methode von Hand ausgerechnet haben, errechnet die Methode `length` tatsächlich die Elementanzahl der Liste:

```
sep@linux:~/fh/prog1/examples/classes> java TestLength
3
sep@linux:~/fh/prog1/examples/classes>
```

Alternative Implementierung im Fall der Klassenhierarchie Im Fall der Klassenhierarchie können wir auch die `if`-Bedingung der Methode `length` dadurch ausdrücken, daß die beiden Fälle sich in den unterschiedlichen Klassen für die beiden unterschiedlichen Listenarten befinden. In der Klasse `List` kann folgende prototypische Implementierung eingefügt werden:

List.java

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

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

Cons.java

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

Auch diese Längenimplementierung können wir testen:

TestListLength.java

```
package name.panitz.data.list;
class TestListLength {
  static  List<String> XS
    = new Cons<String>("friends"
     ,new Cons<String>("romans"
     ,new Cons<String>("countrymen"
     ,new Empty<String>()))));

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

```
10      System.out.println(XS.length());
11    }
12  }
```

An diesem Beispiel ist gut zu sehen, wie durch die Aufsplittung in verschiedene Unterklassen beim Schreiben von Methoden `if`-Abfragen verhindert werden können. Die verschiedenen Fälle einer `if`-Abfrage finden sich dann in den unterschiedlichen Klassen realisiert. Der Algorithmus ist in diesem Fall auf verschiedene Klassen aufgeteilt.

Iterative Lösung Die beiden obigen Implementierungen der Methode `length` sind rekursiv. Im Rumpf der Methode wurde sie selbst gerade wieder aufgerufen. Natürlich kann die Methode mit den entsprechenden zusammengesetzten Schleifenbefehlen in Java auch iterativ gelöst werden.

Wir wollen zunächst versuchen, die Methode mit Hilfe einer `while`-Schleife zu realisieren. Der Gedanke ist naheliegend. Solange es sich noch nicht um die leere Liste handelt, wird ein Zähler, der die Elemente zählt, hochgezählt:

Li.java

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

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

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

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

Li.java

```
1  int lengthFor(){
2    int erg=0;
3    for (Li<a> xs=this;!xs.isEmpty();xs=xs.tail()){
4      erg = erg +1;
5    }
6    return erg;
7  }
```

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

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

toString

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

Li.java

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

Aufgabe 20 Nehmen Sie beide der in diesem Kapitel entwickelten Umsetzungen von Listen und fügen Sie ihrer Listenklassen folgende Methoden hinzu. Führen Sie Tests für diese Methoden durch.

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

b) `List<a> concat(List<a> other)` bzw.: `Li<a> concat(Li<a> other)`: erzeugt eine neue Liste, die erst die Elemente der `this`-Liste und dann der `other`-Liste hat, es sollen also zwei Listen aneinander gehängt werden.

c) `a elementAt(int i)`: gibt das Element an einer bestimmten Indexstelle der Liste zurück. Spezifikation:

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

6.1.5 Sortierung

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

StringOrdering.java

```
1 package name.panitz.data.list;
2 class StringOrdering{
```

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

Diese statische Methode werden wir zum Sortieren benutzen.

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

Sortieren durch Einfügen

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

SortStringLi.java

```
package name.panitz.data.list;
class SortStringLi extends Li<String> {

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

  SortStringLi sortTail(){return (SortStringLi)tail();}
```

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

SortStringLi.java

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

Im Falle einer leeren Liste wird die einelementige Liste zurückgegeben:

SortStringLi.java

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

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

SortStringLi.java

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

Anderfalls wird das neue Element mit dem Konstruktor vorne eingefügt:

SortStringLi.java

```
14      } else return new SortStringLi(x,this);
15    }//method insertSorted
```

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

SortStringLi.java

```
16    SortStringLi getSorted(){
17      SortStringLi result = new SortStringLi();
18
19      for (Li<String> xs= this;!xs.isEmpty();xs = xs.tail()){
20        result = result.insertSorted(xs.head());
21      }
22
23      return result;
24    }//method sort
```

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

SortStringLi.java

```
25    public static void main(String [] args){
26      SortStringLi xs
27        = new SortStringLi("zz"
28         ,new SortStringLi("ab"
29         ,new SortStringLi("aaa"
30         ,new SortStringLi("aaa"
31         ,new SortStringLi("aaz"
32         ,new SortStringLi("aya"
33         ,new SortStringLi()))))));
34
35
36      System.out.println("Die unsortierte Liste:");
37      System.out.println(xs);
38
39      Li<String> ys = xs.getSorted();
40      System.out.println("Die sortierte Liste:");
41      System.out.println(ys);
42    }
43  }//class SortStringLi
```

Die Ausgabe unseres Testprogramms zeigt, daß tatsächlich die Liste sortiert wird:

```
sep@swe10:~/fh/prog1/Listen> java SortStringLi
Die unsortierte Liste:
(zz,ab,aaa,aaa,aaz,aya)
Die sortierte Liste:
(aaa,aaa,aaz,ab,aya,zz)
sep@swe10:~/fh/prog1/Listen>
```

Quick Sort

Der Namen *quick sort* hat sich für eine Sortiermethode durchgesetzt, die sich das Prinzip des Teilens des Problems zu eigen macht, bis die durch Teilen erhaltenen Subprobleme trivial zu lösen sind.[4]

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

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

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

Die zweite Gleichung spezifiziert den Algorithmus für nichtleere Listen. Der in der Gleichung benutzte Operator `++` steht für die Konkatenation zweier Listen mit der in der letzten Aufgabe geschriebenen Methode `concat`.

Die Gleichung ist zu lesen als:

> Um eine nichtleere Liste zu sortieren, filtere alle Elemente aus der Schwanzliste, die kleiner sind als der Kopf der Liste. Sortiere diese Teilliste. Mache dasselbe mit der Teilliste aus den Elementen des Schwanzes, die größer als das Kopfelement sind. Hänge schließlich diese beiden sortierten Teillisten aneinander und das Kopfelement dazwischen.

Modellierung Anders als in unserer obigen Sortierung durch Einfügen in eine neue Liste, für die wir eine Unterklasse der Klasse `Li` geschrieben haben, wollen wir die Methode `quicksort` in der Klasse `Li` direkt implementieren, d.h. allgemein für alle Listen zur Verfügung stellen.

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

FilterCondition.java

```
1 package name.panitz.data.list;
2 class FilterCondition<a> {
3   boolean condition(a testMe){
4     return true;
5   }
6 }
```

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

Für bestimmte Bedingungen können für eine solche Klasse Unterklassen definiert werden, die die Methode `condition` entsprechend überschreiben. Für unsere Sortierung brauchen wir zwei Bedingungen: einmal wird ein Objekt getestet, ob es größer ist als ein vorgegebenes Objekt, ein anderes Mal, ob es kleiner ist. Wir erhalten also folgende kleine Klassenhierarchie aus Abbildung 6.7:

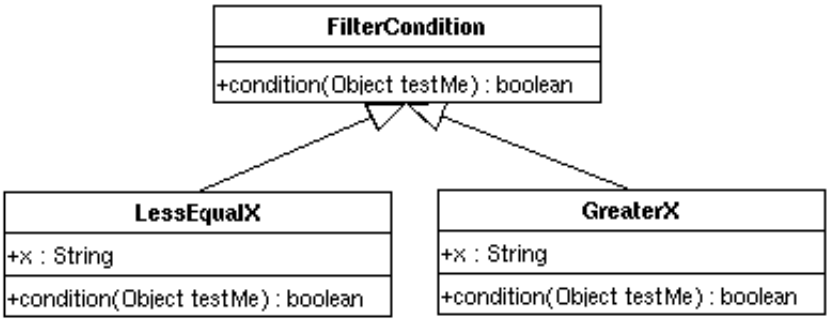


Abbildung 6.7: Modellierung der Filterbedingungen.

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

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

```
Li<a> filter(Condition<a> cond);
```

Codierung Die entscheidende Methode für den Sortieralgorithmus ist `filter`. Mit dieser Methode werden entsprechend einer Filterbedingung bestimmte Elemente aus einer Liste selektiert.

In der Klasse `Li` kann nun die Methode `filter` eingefügt werden:

Li.java

```
  Li<a> filter(FilterCondition<a> cond){
    Li<a> result = new Li<a>();

    //test all elements of this list
    for (Li<a> 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<a>(xs.head(),result);
      }
    }
```

```
    return result;
  }
```

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

Beispiel:
Bevor wir die Methode `quicksort` implementieren, wollen wir ein paar Tests für unsere Methode `filter` schreiben. Hierzu schreiben wir Klassen für die Bedingungen, nach denen wir Elemente aus einer Liste filtern wollen:

Zunächst eine Bedingung, die Stringobjekte mit einer Länge größer als 10 selektiert:

LongString.java

```
package name.panitz.data.list;
class LongString extends FilterCondition<String>{
  boolean condition(String testMe){
    return testMe.length()>10;
  }
}
```

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

StringStartsWithA.java

```
package name.panitz.data.list;
class StringStartsWithA extends FilterCondition<String>{
  boolean condition(String testMe){
    return testMe.charAt(0)=='A';
  }
}
```

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

ContainsNoA.java

```
package name.panitz.data.list;
class ContainsNoA extends FilterCondition<String>{
  boolean condition(String testMe){
    for (int i= 0;i<testMe.length();i=i+1){
      final char c = testMe.charAt(i);

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

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

TestFilter.java

```
package name.panitz.data.list;
class TestFilter {

```

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

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

In der Ausgabe können wir uns vom korrekten Lauf der Methode `filter` überzeugen:

```
sep@linux:~/fh/prog1/examples/classes> java TestFilter
(Shakespeare,Brecht,Achternbusch,Calderon,Moliere,Sorokin,Schimmelpfennig,Kane,Wilde)
(Wilde,Schimmelpfennig,Sorokin,Moliere,Brecht)
(Achternbusch)
(Schimmelpfennig,Achternbusch,Shakespeare)
sep@linux:~/fh/prog1/examples/classes>
```

Interessant zu beobachten mag sein, daß unsere Methode `filter` die Reihenfolge der Elemente der Liste umdreht.

Zurück zu unserer eigentlichen Aufgabe, dem *quick sort*-Verfahren. Hier wollen wir die Eingabeliste einmal nach allen Elementen, die kleiner als das erste Element sind, filtern; und einmal nach allen Elementen, die größer als dieses sind. Hierzu brauchen wir zwei Filterbedingungen. Diese hängen beide von einem bestimmten Element, nämlich dem ersten Element der Liste, ab. Die beiden Klassen für die Bedingung lassen sich relativ einfach aus der Modellierung ableiten:

LessEqualX.java

```
package name.panitz.data.list;
class LessEqualX<a> extends FilterCondition<a>{
  Comparable<a> x;

  LessEqualX(Comparable<a> x){
    this.x=x;
  }

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

Entsprechend für die größer-Relation:

GreaterX.java

```
package name.panitz.data.list;
class GreaterX<a> extends FilterCondition<a> {
  Comparable<a> x;

  GreaterX(Comparable<a> x){
    this.x=x;
  }

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

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

Li.java

```
  static public <a extends Comparable<a>> Li<a> quicksort(Li<a> xs){
    Li<a> result = new Li<a>();
    if (!xs.isEmpty()){
      result
       = //filter the smaller elements out of the tail
         //and sort these
         quicksort(xs.tail().filter(new LessEqualX<a>(xs.head())))
          //concatenate it with the sorted
          //sublist of greater elements
          .append(new Li<a>(xs.head()
                 ,quicksort(xs.tail().filter(new GreaterX<a>(xs.head())))
            ));
    }
    return result;
  }
```

Obige Umsetzung des *quick sort*-Algorithmus ist allgemeiner als der zuvor entwickelte Algorithmus zur Sortierung durch Einfügen. Die entscheidende Methode `filter` ist parameterisiert über die Bedingung, nach der gefiltert werden soll. Damit läßt sich schnell eine *quick sort*-Methode schreiben, deren Filter nicht auf der größer-Relation von `String` Objekten basiert. Hierzu sind nur entsprechende Unterklassen der Klasse `FilterCondition` zu schreiben und in der Sortiermethode zu benutzen. Wieder einmal haben wir unsere strenge Trennung aus der anfänglichen Arbeitshypothese durchbrochen: Die Objekte der Klasse `FilterCondition` stellen nicht primär Daten dar, sondern eine Methode, die wir als Argument einer anderen Methode (der Methode `filter`) übergeben.

Sortieren für beliebige Relationen Es ist naheliegend, die Parameterisierung über die eigentliche Ordnungsrelation der Sortiermethode mitzugeben, also eine Sortiermethode zu schreiben, die einen Parameter hat, der angibt, nach welchem Kriterium zu sortieren ist:

```
Li<a> sortBy(Relation rel)
```

Hierzu brauchen wir eine Klasse `Relation`, die eine Methode hat, in der entschieden wird, ob zwei Objekte in einer Relation stehen:

Relation.java

```
package name.panitz.data.list;
class Relation<a,b>{
  boolean lessEqual(a x,b y){
    return true;
  }
}
```

Je nachdem, was wir sortieren wollen, können wir eine Subklasse der Klasse `Relation` definieren, die uns sagt, wann zwei Objekte in der kleiner-Relation stehen. Für Objekte des Typs `String` bietet folgende Klasse eine adäquate Umsetzung:

StringLessEqual.java

```
package name.panitz.data.list;
class StringLessEqual extends Relation<String,String> {
  boolean lessEqual(String x,String y){
    return x.compareTo(y)<=0;
  }
}
```

Um den *quick sort*-Algorithmus anzuwenden, benötigen wir nun noch eine Möglichkeit, aus einer Relation die beiden Bedingungen für die Methode `filter` generieren. Wir schreiben zwei neue Subklassen der Klasse `FilterCondition`, die für eine Relation jeweils kleinere bzw. größere Objekte als ein vorgegebenes Objekt filtern.

OrderingCondition.java

```
package name.panitz.data.list;
class OrderingCondition<a> extends FilterCondition<a>{
  a x;
  Relation<a,a> rel;

  OrderingCondition(a x,Relation<a,a> rel){
    this.x=x;
    this.rel = rel;
  }

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

Entsprechend für die negierte Relation:

NegativeOrderingCondition.java

```
package name.panitz.data.list;
class NegativeOrderingCondition<a>  extends FilterCondition<a>{
  a x;
  Relation<a,a> rel;

  NegativeOrderingCondition(a x,Relation<a,a> rel){
    this.x=x;
    this.rel = rel;
  }

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

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

Li.java

```
  Li<a> sortBy(Relation<a,a>  rel){
    Li<a> result = new Li<a>();
    if (!isEmpty()){
      FilterCondition<a> le
       = new OrderingCondition<a>(head(),rel);
      FilterCondition <a> gr
       = new NegativeOrderingCondition<a>(head(),rel);
      result = tail()
                .filter(le)
                .sortBy(rel)
                .append(new Li<a>(head()
                                ,tail()
                                .filter(gr)
                                .sortBy(rel)));
    }
    return result;
  }
}
```

Beim Aufruf der Methode `sortBy` ist ein Objekt mitzugeben, das die Relation angibt, nach der sortiert werden soll.

Beispiel:
Im folgenden Beispiel werden Strings einmal nach ihrer lexikographischen Ordnung, einmal nach ihrer Länge sortiert:

StringLengthLessEqual.java

```
package name.panitz.data.list;
class StringLengthLessEqual extends Relation<String,String> {
  boolean lessEqual(String x,String y){
```

```
    return x.length()<= y.length();
  }
}
```

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

TestSortBy.java

```
package name.panitz.data.list;
class TestSortBy{
  public static void main(String [] args){
    Li<String> xs = TestFilter.XS;

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

Und tatsächlich können wir jetzt die Sortiermethode benutzen, um nach unterschiedlichen Kriterien zu sortieren:

```
sep@linux:~/fh/prog1/examples/classes> java TestSortBy
Die unsortierte Liste:
(Shakespeare,Brecht,Achternbusch,Calderon,Moliere,Sorokin,Schimmelpfennig,Kane,Wilde)
Die alphabetisch sortierte Liste:
(Achternbusch,Brecht,Calderon,Kane,Moliere,Schimmelpfennig,Shakespeare,Sorokin,Wilde)
Die nach der Länge sortierte Liste:
(Kane,Wilde,Brecht,Moliere,Sorokin,Calderon,Shakespeare,Achternbusch,Schimmelpfennig)
sep@linux:~/fh/prog1/examples/classes>
```

Aufgabe 21 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 22 Diese Aufgabe soll mir helfen, Listen für Ihre Leistungsbewertung zu erzeugen.

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

```
<table>
  <tr>erstes Listenelement</tr>
  <tr>zweites Listenelement</tr>
  <tr>drittes Listenelement</tr>
</table>
```

b) Nehmen Sie die Klasse `Student`, die Felder für Namen, Vornamen und Matrikelnummer hat. Implementieren Sie für diese Klasse eine Methode `String toTableRow()`, die für Studenten eine Zeile einer Html-Tabelle erzeugt:

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

soll folgende Ausgabe ergeben:

```
<td>Müller</td><td>Hans</td><td>167857</td>
```

Ändern Sie die Methode `toString` so, daß sie dasselbe Ergebnis wie die neue Methode `toTableRow` hat.

c) Legen Sie eine Liste von Studenten an, sortieren Sie diese mit Hilfe der Methode `sortBy`nach Nachnamen und Vornamen und erzeugen Sie eine Html-Seite, die die sortierte Liste anzeigt.

Sie können zum Testen die folgende Klasse benutzen:

HtmlView.java

```
package name.panitz.data.list;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.basic.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

public class HtmlView extends JPanel {

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

  JFrame frame;
  JTextPane ausgabe = new JTextPane();

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

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

```
31
32    void run(){
33      frame = new JFrame("HtmlView");
34      frame.getContentPane().add(this);
35      frame.pack();
36      frame.setVisible(true);
37    }
38  }
```

6.1.6 Formale Beweise über Listenalgorithmen

Im ersten Kapitel haben wir unter den Disziplinen der Programmierung auch die formale Verifikation aufgezählt. Die formale Verifikation erlaubt es, Eigenschaften von Programmen allgemein mathematisch zu beweisen. Anders als durch Testfälle, die nur Aussagen über ein Programm für ausgewählte Fälle machen können, können über die Verifikation allgemeine Aussagen bewiesen werden, die sich auf alle möglichen Argumente für ein Programm beziehen können. Formale Verifikation ist seit Jahrzehnten ein weites Feld der Forschung, die zumeist im Gebiet der KI (künstlichen Intelligenz) angesiedelt ist.

Voraussetzung für eine formale Verifikation ist, daß sowohl die Datentypen als auch die programmierten Algorithmen in einer formalen Weise spezifiziert und notiert wurden. Für unsere Listentypen haben wir das bisher getan. Unsere Listen sind daher bestens geeignet, um formale Beweise zu führen.

Vollständige Induktion

Das aus der Mathematik bekannte Verfahren der vollständigen Induktion über die natürlichen Zahlen ist ein gängiges Verfahren zur Verifikation von Algorithmen. Ein Induktionsbeweis geht dabei in zwei Schritten. Zunächst wird im sogenannten Induktionsanfang die Aussage für das kleinste Datum geführt, bei natürlichen Zahlen also für die Zahl 0. Im zweiten Schritt, dem sogenannten Induktionsschritt, wird angenommen, daß die Aussage bereits für alle Werte kleiner eines bestimmten Wertes n bewiesen wurde. Dann wird versucht, unter dieser Annahme die Aussage auch für n zu beweisen. Sind beide Beweisschritte gelungen, so ist die Aussage für alle endlichen Werte bewiesen.

Induktion über Listen

Das Beweisverfahren der Induktion läßt sich auf rekursiv definierte Datentypen, so wie unsere Listen, anwenden. Der Basisfall, also entsprechend der 0 bei den natürlichen Zahlen, sind die Daten, die durch einen nicht rekursiven Konstruktor erzeugt werden. Für Listen entsprechend sind dieses die Listen, die mit dem Konstruktor für leere Listen erzeugt wurden. Im Induktionsschritt wird angenommen, die zu beweisende Aussage sei bereits für Daten, die mit weniger Konstruktoren erzeugt wurden, bewiesen. Unter dieser Annahme wird versucht zu zeigen, daß die Aussage auch für mit einem weiteren Konstruktor erzeugte Daten gilt. Für Listen bedeutet das, man versucht, die Annahme für die Liste der Form *Cons(x,xs)* zu beweisen unter der Annahme, daß die Aussage für die Liste *xs* bereits bewiesen wurde.

Beispiel:
Als Beispiel wollen wir eine Eigenschaft über Listen im Zusammenhang mit den Funktionen *concat* und *length* beweisen. Wir wollen beweisen, daß die Länge der Konkatenation zweier Listen gleich der Summe der Längen der beiden Listen ist, also daß für alle Listen *xs* und *ys* gilt:

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

Dabei seien die beiden Funktionen wieder spezifiziert als:

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

und

$$\begin{aligned} concat(Empty(), ys) &= ys \\ concat(Cons(x, xs), ys) &= Cons(x, concat(xs, ys)) \end{aligned}$$

Wir werden die Aussage beweisen mit einer Induktion über das erste Argument der Funktion *concat*, also dem *xs*:

- **Induktionsanfang:** Wir versuchen, die Aussage zu beweisen mit *xs=Empty()*. Wir erhalten die folgende Aussage:

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

  Wir können jetzt auf beiden Seiten der Gleichung durch Reduktion mit den Gleichungen aus der Spezifikation die Gleichung vereinfachen:

  $$\begin{aligned} length(concat(Empty(), ys)) &= length(Empty()) + length(ys) \\ length(ys) &= length(Empty()) + length(ys) \\ length(ys) &= 0 + length(ys) \\ length(ys) &= length(ys) \end{aligned}$$

  Wir haben die Aussage auf eine Tautologie reduziert. Für *xs* als leere Liste ist damit unsere Aussage bereits bewiesen.
- **Induktionsschritt:** Jetzt wollen wir die Aussage für *xs=Cons(x',xs')* beweisen, wobei wir als Induktionsvoraussetzung annehmen, daß sie für *xs'* bereits wahr ist, daß also gilt:

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

  Hierzu stellen wir die zu beweisende Gleichug auf und reduzieren sie auf beiden Seiten:

  $$\begin{aligned} length(concat(Cons(x', xs'), ys)) &= length(Cons(x', xs')) + length(ys) \\ length(Cons(x', concat(xs', ys))) &= length(Cons(x', xs')) + length(ys) \\ 1 + length(concat(xs', ys)) &= 1 + length(xs') + length(ys) \\ length(concat(xs', ys)) &= length(xs') + length(ys) \end{aligned}$$

  Die letzte Gleichung ist gerade die Induktionsvoraussetzung, von der wir angenommen haben, daß diese bereits wahr ist. Wir haben unsere Aussage für alle endlichen Listen bewiesen.

6.2 Bäume

Bäume sind ein gängiges Konzept um hierarchische Strukturen zu modellieren. Sie sind bekannt aus jeder Art von Baumdiagramme, wie Stammbäumen oder Firmenhierarchien. In der Informatik sind Bäume allgegenwärtig. Fast alle komplexen Daten stellen auf die eine oder andere Art einen Baum dar. Beispiele für Bäume sind mannigfach:

- **Dateisystem**: Ein gängiges Dateisystem, wie es aus Unix, MacOS und Windows bekannt ist, stellt eine Baumstruktur dar. Es gibt einen ausgezeichneten Wurzelordner, von dem aus zu jeder Datei einen Pfad existiert.
- **Klassenhierarchie**: Die Klassen in Java stellen mit ihrer Ableitungsrelation eine Baumstruktur dar. Die Klasse `Object` ist die Wurzel dieses Baumes, von der aus alle anderen Klassen über einen Pfad entlang der Ableitungsrelation erreicht werden können.
- **XML**: Die logische Struktur eines XML-Dokuments ist ein Baum. Die Kinder eines Elements sind jeweils die Elemente, die durch das Element eigeschlossen sind.
- **Parserergebnisse**: Ein Parser, der gemäß einer Grammatik prüft, ob ein bestimmter Satz zu einer Sprache gehört, erzeugt im Erfolgsfall eine Baumstruktur. Im nächsten Kapitel werden wir dieses im Detail kennenlernen.
- **Listen**: Auch Listen sind Bäume, allerdings eine besondere Art, in denen jeder Knoten nur maximal ein Kind hat.
- **Berechnungsbäume**: Zur statischen Analyse von Programmen, stellt man Bäume auf, in denen die Alternativen eines bedingten Ausdrucks Verzweigungen im Baum darstellen.
- **Tableaukalkül**: Der Tableaukalkül ist ein Verfahren zum Beweis logischer Formeln. Die dabei verwendeten Tableaux sind Bäume.
- **Spielbäume**: Alle möglichen Spielverläufe eines Spiels können als Baum dargestellt werden. Die Kanten entsprechen dabei einem Spielzug.
- **Prozesse**: Auch die Prozesse eines Betriebssystems stellen eine Baumstruktur dar. Die Kinder eines Prozesses sind genau die Prozesse, die von ihm gestartet wurden.

Wie man sieht, lohnt es sich, sich intensiv mit Bäumen vertraut zu machen, und man kann davon ausgehen, was immer in der Zukunft neues in der Informatik entwickelt werden wird, Bäume werden darin in irgendeiner Weise eine Rolle spielen.

Ein Baum besteht aus einer Menge von Knoten die durch gerichtete Kanten verbunden sind. Die Kanten sind eine Relation auf den Knoten des Baumes. Die Kanten verbinden jeweils einen Elternknoten mit einem Kinderknoten. Ein Baum hat einen eindeutigen Wurzelknoten. Dieses ist der einzige Knoten, der keinen Elternknoten hat, d.h. es gibt keine Kante, die zu diesen Knoten führt. Knoten, die keinen Kinderknoten haben, d.h. von denen keine Kante ausgeht, heißen Blätter.

Die Kinder eines Knotens sind geordnet, d.h. sie stehen in einer definierten Reihenfolge.

Eine Folge von Kanten, in der der Endknoten einer Vorgängerkante der Ausgangsknoten der nächsten Kanten ist, heißt Pfad.

In einem Baum darf es keine Zyklus geben, das heißt, es darf keinen Pfad geben, auf dem ein Knoten zweimal liegt.

Knoten können in Bäumen markiert sein, z.B. einen Namen haben. Mitunter können auch Kanten eine Markierung tragen.

Bäume, in denen Jeder Knoten maximal zwei Kinderknoten hat, nennt man Binärbäume. Bäume, in denen jeder Knoten maximal einen Kinderknoten hat, heißen Listen.

6.2.1 Formale Spezifikation

Allgemeine Bäume

Wir wollen in diesem Abschnitt Bäume als einen abstrakten Datentypen spezifizieren.

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

In dem Fall von Bäumen bedarf es nach den obigen Überlegungen nur eines Konstruktors, den für Knoten. Er konstruiert aus der Liste der Kinderbäume einen neuen Baumknoten. Als zusätzliches Argument bekommt er ein Objekt, das eine Markierung für den Baumknoten darstellt. Den Typ dieses Markierungsobjektes können wir erst einmal offen lassen. Hier darf ein beliebiger Typ gewählt werden.

Wir benutzen in der Spezifikation eine mathematische Notation der Typen von Konstruktoren.[5] Dem Namen des Konstruktors folgt dabei mit einem Doppelpunkt abgetrennt der Typ. Der Ergebnistyp wird von den Parametertypen mit einem Pfeil getrennt.

Somit läßt sich der Typ des Konstruktors für Bäume wie folgt spezifizieren:

- Node: (α,**List<Tree<**α**>>**) $\rightarrow$**Tree<**α**>**

Selektoren Die Selektoren können wieder auf die einzelnen Bestandteile der Konstruktion zurückgreifen. Der Konstruktor **Node** hat zwei Parameter. Für Bäume werden zwei Selektoren spezifiziert, die jeweils einen dieser beiden Parameter wieder aus dem Baum selektieren.

- theChildren: **Tree<**α**>**$\rightarrow$**List<Tree<**α**>>**
- mark: **Tree<**α**>**$\rightarrow\alpha$

Der funktionale Zusammenhang von den Selektoren und Konstruktoren läßt sich durch folgende Gleichungen spezifizieren:

mark(Node(x,xs)) = x
theChildren(Node(x,xs)) = xs

Testmethoden Da wir nur einen Konstruktor vorgesehen haben, brauchen wir keine Testmethode, die unterscheidet, mit welchem Konstruktor ein Baum konstruiert wurde.

Die Unterscheidung, ob es sich um ein Blatt oder um einen Knoten mit Kindern handelt, läßt sich über die Abfrage, ob die Liste der Kinderknoten leer ist, erfahren. Wir können eine entsprechende Funktion spezifizieren:

isLeaf(Node(x,xs)) = isEmpty(xs)

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

6.2.2 Modellierung

Wir haben wieder verschiedene Möglichkeiten eine Baumstruktur mit Klassen zu modellieren.

mit einer Klasse

Die einfachste Modellierung ist mit einer Klasse. Diese Klasse stellt einen Knoten dar. Sie enthält ein Feld für die Knotenmarkierung und ein Feld für die Kinder des Knotens. Nach unserer Spezifikation sind die Kinder eine Liste weiterer Knoten. Ein entsprechendes UML Diagramm befindet sich in Abbildung 6.8

Abbildung 6.8: Modellierung von Bäumen mit einer Klasse.

Mit einer Klassenhierarchie

Ebenso, wie wir es bei Listen vorgeschlagen haben, können wir auch wieder eine Modellierung mit mehreren Klassen vornehmen. Hierzu beschreibt eine Schnittstelle die allgemeine Funktionalität von Bäumen. Zwei Unterklassen, jeweils eine für innere Knoten und eine für Blätter implementieren diese Schnittstelle. Ein entsprechendes UML Diagramm befindet sich in Abbildung 6.9

Welche dieser beiden Modellierungen jeweils vorzuziehen ist, hängt sehr davon ab, für was die Bäume in der Gesammtanwendung benutzt werden sollen. Eine grobe Faustregel für abstrakte Datentypen ist:

- Wenn zu erwarten ist, daß im Laufe der Softwareentwicklung noch weitere Konstruktoren zum abstrakten Datentyp hinzudefiniert werden, dann ist eine Modellierung als Hierarchie vorteilhaft. Der neue Konstruktor führt zu einer neuen Klasse, in der lokale Version der für die implementierte Schnittstelle benötigten Methoden implementiert werden.
- Wenn zu erwarten ist, daß spezialisierte Versionen des abstrakten Datentyps benötigt werden, so ist eine Modellierung als eine Klasse vorteilhaft, weil dann für dieser einen Klassen eine Unterklasse implementiert werden kann.

Wir werden im Folgenden für unsere Bäume die Modellierung mit einer Klasse bevorzugen und anschließend spezialisierte Baumtypen von dieser einen Klasse ableiten.

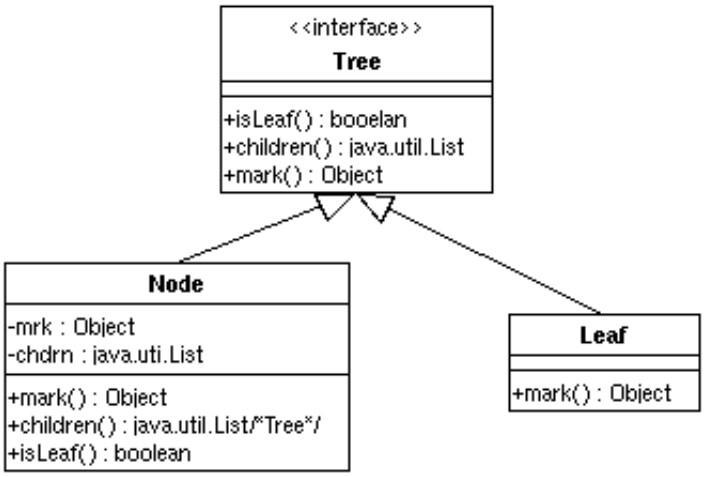


Abbildung 6.9: Modellierung von Bäumen mittels einer Klassenhierarchie.

6.2.3 Implementierung

Für unsere Umsetzung von Bäumen benutzen wir die Modellierung in einer Klasse. Wir schreiben die Klasse `Tree` entsprechend unserer Spezifikation. Zusätzlich sehen wir zwei weitere Konstruktoren vor, die jeweils einen Standardwert für die beiden Felder setzen:

- `Tree(List children)`: als Knotenmarkierung wird `null` genommen.
- `Tree(a mark)`: für die Kinder wird eine leere Liste erzeugt.

Wir erhalten folgende einfache Klasse:

Tree.java

```
1  package name.panitz.data.tree;
2
3  import java.util.List;
4  import java.util.Iterator;
5  import java.util.Enumeration;
6  import java.util.ArrayList;
7  import javax.swing.tree.TreeNode;
8
9  public class Tree<a> implements TreeNode{
10
11   private a mrk;
12   private List<Tree<a>> chldrn;
```

Die beiden Felder werden durch den Konstruktor gesetzt:

```
1   public Tree(a mark,List<Tree<a>> children){
2     this.mrk=mark;
```

```
    this.chldrn=children;
  }
```

Zwei weitere Konstruktoren setzen beziehen sich auf diesen Konstruktor:

Tree.java

```
  public Tree(List<Tree<a>> children){
    this(null,children);
  }

  public Tree(a mark){
    this(mark,new ArrayList<Tree<a>>());
  }


  public List<Tree<a>> theChildren(){return chldrn;}
  public a mark(){return mrk;}

  public boolean isLeaf(){return theChildren().isEmpty();}

```

Aufgabe 23 Erzeugen Sie ein Baumobjekt, das einen Stammbaum Ihrer Familie darstellt. Die Knoten und Blätter sind mit Namen markiert. Die Wurzel ist mit Ihren Namen markiert. Kinderknoten sind mit den leiblichen Eltern der Person eines Knotens markiert.

6.3 Algorithmen auf Bäumen

Die Selektormethoden für Bäume und Listen können wir jetzt gemeinsam nutzen, um einfache Algorithmen auf Bäumen umzusetzen.

6.3.1 Knotenanzahl

Entsprechend der Methode `length`, wie wir sie für Listen geschrieben haben, lassen sich die Knoten eines Baumes zählen.

Tree.java

```
public int count(){
  int result = 1; // Startwert 1 für diesen Knoten

  //für jedes Kind
  for (Tree<a> child: theChildren())
    //addiere die Knotenanzahl des Kindes zum Ergebnis
    result=result+child.count();

  return result;
}
```

Diese Methode ist rekursiv geschrieben. Der terminierende Fall versteckt sich dieses mal in der `for-Schleife`. Wenn die Liste der Kinder leer ist, so wird die `for`-Schleife nicht durchlaufen. Nur im Rumpf der Schleife steht ein rekursiver Aufruf. Daher gibt es keine Anweisung der Form: `if (theChildren().isEmpty()) return 1`

6.3.2 toString

Unter Benutzung der Methode `toString` aus der Listenimplementierung, läßt sich relativ einfach eine `toString`-Methode für unsere Baumimplementierung umsetzen.

```
public String toString(){
  //Nimm erst die Markierung an diesem Knoten
  String result = mark().toString();

  //Gibt es keine Kinder, dann bist du fertig
  if (isLeaf()) return result;

  //ansonsten nimm implizit durch Operator + das
  //toString() der Kinder
  return result+theChildren();
}
```

Auch wenn wir es nicht sehen, ist diese Methode rekursiv. Der Ausdruck `result+theChildren()` führt dazu, daß auf einer Liste von Bäumen die Methode `toString()` ausgeführt wird. Dieses führt wiederum zur Ausführung der `toString`-Methode auf jedes Element und dieses sind wiederum Bäume. Der Aufruf von `toString` der Klasse `Tree` führt über den Umweg der Methode `toString` der Schnittstelle `List` abermals zur Ausführung der `ToString`-Methode aus `Tree`. Man spricht dabei auch von *verschränkter* Rekursion.

6.3.3 Linearisieren

Die Methode `flatten` erzeugt eine Liste, die die Markierung jedes Baumknotens genau einmal enthält. Aus einem Baum wird also eine Liste erzeugt. Die logische Struktur des Baumes wird flachgeklopft.

Tree.java

```
public List<a> flatten(){
  //leere Liste für das Ergebnis
  List<a> result = new ArrayList<a>();

  //füge die Markierung dieses Knotens zur
  //Ergebnisliste hinzu
  result.add(mark());

  //für jedes Kind
  for (Tree<a> child : theChildren())
    //erzeuge dessen Knotenliste und füge alle deren
    //Elemente zum Ergebnis hinzu
```

```
24      result.addAll(child.flatten());
25
26    return result;
27  }
```

Als Ergebnisliste wird ein Objekt der Klasse `ArrayList` angelegt. Diesem Objekt wird die Markierung des aktuellen Knotens zugefügt und dann die Linearisierung jedes der Kinder. Die Methode `addAll` aus der Schnittstelle `List` sorgt dafür, daß alle Elemente der Parameterliste hinzugefügt werden.

Aufgabe 24 Testen Sie die in diesem Abschnitt entwickelten Methoden auf Bäumen mit dem in der letzten Aufgabe erzeugten Baumobjekt.

Aufgabe 25 Schreiben Sie die folgenden weiteren Methoden für die Klasse `Tree`. Schreiben Sie Tests für diese Methoden.

a) `List leaves()`: erzeugt eine Liste der Markierungen an den Blättern des Baumes.

b) `String show()`: erzeugt eine textuelle Darstellung des Baumes. Jeder Knoten soll eine eigene Zeile haben. Kinderknoten sollen gegenüber einem Elternknoten eingerückt sein.

Beispiel:

```
william
  charles
    elizabeth
    phillip
  diana
    spencer
```

Tipp: Benutzen Sie eine Hilfsmethode
`String showAux(String prefix)`,
die den String `prefix` vor den erzeugten Zeichenketten anhängt.

c) `boolean contains(a o)`: ist wahr, wenn eine Knotenmarkierung gleich dem Objekt `o` ist.

d) `int maxDepth()`: gibt die Länge des längsten Pfades von der Wurzel zu einem Blatt an.

e) `List<a> getPath(a o)`: gibt die Markierungen auf dem Pfad von der Wurzel zu dem Knoten, der mit dem Objekt `o` markiert ist, zurück. Ergebnis ist eine leere Liste, wenn ein Objekt gleich `o` nicht existiert.

f) `java.util.Iterator<a> iterator()`: erzeugt ein Iterator über alle Knotenmarkierungen des Baumes.
Tipp: Benutzen Sie bei der Umsetzung die Methode `flatten`.

g) `public boolean equals(Object other)`: soll genau dann wahr sein, wenn `other` ein Baum ist, der die gleiche Struktur und die gleichen Markierungen hat.

h) `<b>boolean sameStructure(Tree<a> other)`: soll genau dann wahr sein, wenn der Baum `other` die gleiche Struktur hat. Die Markierungen der Knoten können hingegen unterschiedlich sein.

6.3.4 Lösung

leaves

`List leaves()`: erzeugt eine Liste der Markierungen an den Blättern des Baumes.

Wir können die Methode `leaves` aus der Methode `flatten` ableiten. Beide Methoden nehmen Elemente eines Baumes und fügen diese in eine Ergebnisliste ein.

Tree.java

```
28 public List<a> leaves(){
29   //leere Liste für das Ergebnis
30   List<a> result = new ArrayList<a>();
31
32   //füge die Markierung dieses Knotens zur Ergebnisliste
33   //nur hinzu, wenn es ein Blatt ist
34   if (isLeaf()) result.add(mark());
35
36   //für jedes Kind
37   for (Tree<a> child:theChildren())
38     //erzeuge dessen Knotenliste und füge alle deren Blätter
39     //zum Ergebnis hinzu
40     result.addAll(child.leaves());
41
42   return result;
43 }
```

Alles, was zu ändern ist, ist eine Bedingung aufzustellen, wann ein Knoten der Ergebnisliste hinzugefügt werden soll. Bei `flatten` war das immer der Fall, bei `leaves` nur, wenn es sich bei dem Baumknoten um ein Blatt handelt.

Akkumulative Lösung Wir haben uns bisher noch keine Gedanken um die Performanz unserer Methoden gemacht. Wenn wir die die Methoden `flatten` und `leaves` einmal genau betrachten, so stellen wir fest, daß jeder Aufruf der Methode eine neue Ergebnisliste erzeugt. Das bedeutet, daß die Methoden `flatten` und `leaves` durch die rekursiven Aufrufe, soviel Listen erzeugt, wie Knoten im Baum sind. Mit dem Aufruf der Methode `addAll` auf Listen, werden die Elemente einer Liste in eine neue Liste eingetragen. Wir sind aber nur an einer einzigen Liste interessiert, der endgültigen Ergebnisliste.

Es wäre also schön, wenn wir eine Methode `flatten` hätten, die nicht eine neue Liste erzeugt, sondern eine bestehende Liste benutzt, um dort weitere Elemente einzufügen. Wir können hierzu der Methode einen Listenparameter geben, in dem die Ergebnisse zu sammeln sind.

Tree.java

```
44 public List<a> flatten(List<a> result){
45   result.add(mark());
46
47   for (Tree<a> child:theChildren())
48     child.flatten(result);
49
50   return result;
51 }
```

Die eigentliche Methode `flatten` kann jetzt einmal das Objekt für die Ergebnisliste erzeugen, und dann diese jeweils weitereichen, um das Erzeugen neuer Listen und das umkopieren der Elemente von einer Liste in eine andere zu verhindern.

Tree.java

```
52 public List<a> flattenAkk(){
53   return flatten(new ArrayList<a>());
54 }
```

Und entsprechend können wir die Methode `leaves` auch umschreiben, so daß sie nicht für jeden rekursiven Aufruf ein neues Listenobjekt erzeugt:

Tree.java

```
55 public List<a> leaves(List<a> result){
56   if (isLeaf()) result.add(mark());
57
58   for (Tree<a> child:theChildren())
59     child.leaves(result);
60
61   return result;
62 }
63
64 public List<a> leavesAkk(){
65   return leaves(new ArrayList<a>());
66 }
```

Dieses Prinzip, ein Ergebnis in einem Parameter nach und nach zusammenzusammeln nennt man *Akkumulation*. Der Parameter, in dem das Ergebnis jeweils hinzugefügt wird, ist der Akkumulator. Mit Hilfe des Befehls `time` auf der Unix-Shell, läßt sich sogar ungefähr messen, daß die akkumulierende Methode schneller abgearbeitet wird.

Programmierung höherer Ordnung Wir haben im letztem Abschnitt bereits beobachten können, daß die Methoden `flatten` und `leaves` fast identischen Code haben. Beide selektieren aus einem Baum bestimmte Baummarkierungen heraus. Das einzige was sie unterscheidet, ist die Bedingung, die steuert, ob eine Knotenmarkierung zu selektieren ist oder nicht. Ansonsten haben wir Code verdoppelt.

Unser Bestreben ist stets eine Codeverdoppelung zu vermeiden, denn verdoppelter Code ist auch doppelt zu pflegen und zu warten. Das haben wir sogar schon festgestellt, als wir auf die akkumulative Lösung der beiden Methoden gekommen sind. Da haben wir beide Methoden geändert, um für sie die akkumulative Lösung einzuführen.

Wir können eine Methode schreiben, die den gemeinsamen Code der beiden Methoden enthält. Das einzige, was die beiden Methoden unterscheidet ist eine Bedingung. Diese Bedingung ist eine Prüfung auf die Baumknoten. Die gemeinsame Methode muß diese Bedingung als Parameter übergeben bekommen. Eine Prüfung ist eine Methode. Wir können in Java keine Methoden direkt als Parameter übergeben, sondern nur Objekte. Wir definieren daher eine Schnittstelle, die beschreibt, daß ein Objekt eine Prüfmethoden enthält.

TreeCondition.java

```
1 package name.panitz.data.tree;
2
```

```
3  public interface TreeCondition<a>{
4    boolean takeThis(Tree<a> o);
5  }
```

Mit dieser Schnittstelle sind wir jetzt in der Lage eine Bedingung auf Baumknoten auszudrücken. Wir können damit eine allgemeine Filtermethode schreiben, die Knoten mit einer bestimmten Bedingung aus einem Baum selektiert. Die Methode `filter` ist die Verallgemeinerung die Methoden `leaves` und `flatten`. Wir verallgemeinern die akkumulative Lösung.

Tree.java

```
6  public List<a> filter(List<a> result,TreeCondition<a> c){
7    //wenn die Bedingung für den Knoten wahr ist, füge
8    //ihm dem Ergebnis hinzu
9    if (c.takeThis(this)) result.add(mark());
10
11   //für alle Kinder
12   for (Tree<a> child:theChildren())
13     //rufe den Filter mit gleicher Bedingung auf
14     child.filter(result,c);
15
16   return result;
17 }
```

Die akkumulative Methode kann noch wie gewohnt gekapselt werden:

Tree.java

```
18 public List<a> filter(TreeCondition<a> c){
19   return filter(new ArrayList<a>(),c);
20 }
```

Jetzt haben wir eine Methode geschrieben, die einmal durch einen Baum läuft und Knoten aufgrund einer Bedingung in einer Ergebnisliste sammelt. Wir können jetzt die Methoden `leaves` und `flatten` über diese Methode ausdrücken:`leaves` ist ein Filter mit der Bedingung, daß die Baumknoten ein Blatt ist. Wir erzeugen das Bedingungsobjekt als anonyme innere Klasse:

Tree.java

```
21 public List leavesFilter(){
22   return filter(
23      new TreeCondition<a>(){
24         public boolean takeThis(Tree<a> t){
25            return t.isLeaf();
26         }
27      });
28 }
```

`flatten` ist ein Filter mit der Bedingung die immer wahr ist. Also erzeugen wir eine Baumbedingung, die konstant `true` als Ergebnis liefert:

Tree.java

```
29 public List<a> flattenFilter(){
30   return filter(new TreeCondition<a>(){
31                   public boolean takeThis(Tree<a> _){ return true;}
32                 });
33 }
```

Entsprechend können wir jetzt beliebige Bedingungen benutzen, um Knoten aus dem Baum zu selektieren; z.B. alle Knoten mit genau zwei Kindern:

Tree.java

```
34 public List<a> nodesWithTwoChildren(){
35   return filter(new TreeCondition<a>(){
36                   public boolean takeThis(Tree<a> t){
37                      return t.theChildren().size()==2;
38                   }
39                 });
40 }
```

Oder Knoten, die mit einem bestimmten Objekt markiert sind:

Tree.java

```
41 public List<a> nodesMarkedWith(final a o){
42   return filter(new TreeCondition<a>(){
43                   public boolean takeThis(Tree<a> t){
44                      return t.mark().equals(o);
45                   }
46                 });
47 }
```

Letztere Methode kann sogar genutzt werden, um zu entscheiden, ob ein Objekt in einem Baum enthalten ist.

Tree.java

```
48 public boolean containsFilter(a o){
49   return !nodesMarkedWith(o).isEmpty();
50 }
```

Das in dieser Lösung vorgestellte Prinzip, daß quasi eine Methode als Parameter an eine andere Methode übergeben wird, heißt *Programmierung höherer Ordnung*. Java unterstützt dieses Konzept nur rudimentär, weil zum Übergeben einer Methode als Parameter erst ein Objekt erzeugt werden muß. In Sprachen wie C kann dieses Programmierprinzip über Funktionszeiger realisiert werden. In Sprachen der `Lisp`-Tradition ist die Übergabe von Methoden als Parameter ein fundamentaler Bestandteil. Mit Hilfe der Programmierung höherer Ordnung hat man ein sehr mächtiges Ausdrucksmittel.

show

`String show()`: erzeugt eine textuelle Darstellung des Baumes. Jeder Knoten soll eine eigene Zeile haben. Kinderknoten sollen gegenüber einem Elternknoten eingerückt sein.

Wir schreiben als erstes eine Methode, die einen Baum mit einer als Parameter übergebenen Einrückung zeigt. Diese Einrückung wird für die Kinder um zwei Zeichen verlängert:

Tree.java

```
51  public String show(String indent){
52    //zeige diesen Knoten mit Einrückung
53    String result = indent+mark();
54
55    //erhöhe die Einrückung
56    indent=indent+"  ";
57
58    //für jedes Kind
59    for (Tree<a> child:theChildren())
60     //erzeuge neue Zeile und zeige Kind mit neuer Einrückung
61      result=result+"\n"+child.show(indent);
62
63    return result;
64  }
```

Die Wurzel eines Baumes zeigen wir ohne Einrückung. Wir kapseln die obige Methode entsprechend:

Tree.java

```
65  public String show(){
66      return show("");
67  }
```

Akkumulative Lösung Auch für die Methode `show` können wir eine akkumulative Lösung schreiben. Der Grund, weshalb dieses sinnvoll ist, ist in der Technik von der Stringumsetzung in Java versteckt. Der Übeltäter ist hierbei der Operator `+` auf `String`. Dieser Operator hängt nicht einen `String` an einen anderen `String` an, sondern kopiert beide in einen neuen `String`. Jede Anwendung des Operators `+` kopiert also `String`-Objekte; und wir wenden den Operator oft an, um unser Ergebnis zu berechnen.

Java stellt eine Klasse zur Verfügung mit der `String`-Objekte akkumuliert werden können, und Kopiervorgänge weitgehendst vermieden werden: die Klasse `StringBuffer`.

Objekte der Klasse `StringBuffer` können mit der Methode `append` weitere `String`-Objekte angehängt werden. Sie wird benutzt, wenn ein `String` erst nach und nach zusammengebaut wird.

Ähnlich wie bei der Akkumulation eines Listenergebnisses können wir mit `StringBuffer` eine Zeichenkette akkumulieren. Hierzu geben wir in einem Parameter einen `StringBuffer` als Akkumulator mit. Wie im Listenfall mit der Methode `add` benutzen wir die Methode `append` zum Aufsammeln der Ergebnisteile:

Tree.java

```
68  public StringBuffer show(String indent,StringBuffer result){
69    //zeige diesen Knoten mit Einrückung
70    result.append(indent);
71    result.append(mark().toString());
72
73    //erhöhe die Einrückung
74    indent=indent+"  ";
75
76    //für jedes Kind
77    for (Tree<a> child:theChildren()){
78     //erzeuge neue Zeile und zeige Kind mit neuer Einrückung
79      result.append("\n");
80      child.show(indent,result);
81    }
82    return result;
83  }
```

Die kapselnde Methode, erzeugt einen neuen `StringBuffer`, läßt diesen akkumulieren und gibt schließlich den in ihm enthaltenen `String` als Ergebnis aus.

Tree.java

```
84  public String showAkk(){
85      return show("",new StringBuffer()).toString();
86  }
```

contains

`boolean contains(a o)`: ist wahr, wenn eine Knotenmarkierung gleich dem Objekt `o` ist.

Wir haben im Zuge der Methode `filter` schon eine Lösung für die Methode `contains` quasi umsonst bekommen. Diese Lösung hat alle Knoten die das fragliche Objekt enthalten in einer Liste gesammelt und gekuckt, ob die Liste leer ist oder nicht. Dieses Prinzip wird manchmal auch unter dem Motto: *representing failure by a list of successes* vorgestellt. Die Ergebnisliste zeigt die Treffer an. `contains` falliert, wenn diese Liste leer ist.

Der Nachteil an dieser Lösung war, daß der gesammte Baum durchlaufen wurde, auch wenn z.B. die Wurzel bereits der fragliche Knoten war. Wir entwickeln nun eine Lösung, die einen Baum nur bis zum ersten Auftreten des fraglichen Objektes durchläuft. Nur wenn das Objekt nicht im Baum enthalten ist, wird der komplette Baum durchlaufen.

Tree.java

```
87  public boolean contains(a o){
88    //bin ich schon der gesuchte Knoten? Wenn ja ende mit erfolg
89    if (mark().equals(o)) return true;
90
91    //für jedes Kind
92   for (Tree<a> child:theChildren()){
93      //wenn es den Knoten enthält, beende mit erfolg
94      if (child.contains(o)) return true;
```

```
95      }
96      //keiner der Knoten enthielt das Objekt
97      return false;
98    }
```

maxDepth

`int maxDepth()`: gibt die Länge des längsten Pfades von der Wurzel zu einem Blatt an.

Die maximale Tiefe läßt sich relativ einfach finden. Jedes Kind wird nach seiner maximalen Tiefe befragt und aus diesen die maximale Zahl genommen. Schließlich wird diese um eins erhöht, für die Wurzel selbst.

Tree.java

```
99    public int maxDepth(){
100     //anfangstiefe
101     int result = 0;
102
103     //für jedes Kind
104    for (Tree<a> child:theChildren()){
105       //berechne seine maximale Tiefe
106       int n = child.maxDepth();
107
108       //ist sie höher als die bisher gefundene, dann nimm diese
109       if (n>result) result=n;
110     }
111
112     //erhöhe Ergebnis um eins für die Wurzel
113     return result+1;
114   }
```

getPath

`List getPath(Object o)`: gibt die Markierungen auf dem Pfad von der Wurzel zu dem Knoten, der mit dem Objekt `o` markiert ist, zurück. Ergebnis ist eine leere Liste, wenn ein Objekt gleich `o` nicht existiert.

naive Lösung Fast alle Studenten haben eine Lösung entwickelt, in der die Aufgabe mit Hilfe der Mehode `contains` realisiert wird. Jeder Knoten, der auf dem Pfad zu dem fraglichen Objekt liegt, enthält diesen Knoten. Es wird eine Ergebnisliste aufgebaut, die genau die Knoten enthält, deren Bäume den fraglichen Knoten enthalten:

Tree.java

```
115   public List<a> getPathBad(a o){
116     List<a> result = new ArrayList<a>();
117
118   //wenn Du irgendwo das Objekt enthältst, liegst Du auf den Pfad
119     if (contains(o)) { result.add(mark());}
120
```

```
121     //für jedes Kind
122    for (Tree<a> child:theChildren()){
123       //füge den Pfad zum fraglichen Objekt hinzu
124       result.addAll(child.getPathBad(o));
125     }
126     return result;
127   }
```

Lösung höherer Ordnung mit Hilfe von filter Die obige Lösung ist wieder von der Art: selektiere alle Knoten mit einer bestimmten Eigenschaft aus den Baum. Die Eigenschaft ist keine lokal den Knoten betreffende Eigenschaft, sondern die Tatsache, daß der Teilbaum das Objekt enthält. Wir können also auch für diese Aufgabe die Methode `filter` anwenden.

Tree.java

```
128 public List<a> getPathFilter(final a o){
129    return filter(
130       new TreeCondition<a>(){
131         public boolean takeThis(Tree<a> e){
132           return e.contains(o);
133         }
134       }
135    );
136 }
```

Spätestens jetzt merkt man, was für eine ausdrucksstarke Methode die Methode `filter` durch ihren Parameter höherer Ordnung ist.

performantere Lösung Beide obigen Lösungen waren schlecht, wenn wir ihr Laufzeitverhalten betrachten. Stellen wir uns der Einfachheit halber einen zur Liste entartetetn Baum mit n Knoten vor. Das einzige Blatt ist der Knoten, zu dem der Pfad gesucht wird. Für jeden Knoten des Baumes fragen wir, ob er das fragliche Objekt in seinem Teilbaum enthält, und jedesmal durchlaufen wir dabei den Baum bis zu seinem Blatt, weil wir die Methode `contains` aufrufen. Wir rufen also n-Mal die Methode `contains` auf, die jedesmal den Baum bis zum Blatt durchläuft. Wir durchlaufen also $n + (n-1) + (n-2) + \ldots + (n-n)$
Baumknoten, was nach der Summenformel $\frac{n*(n+1)}{2}$ ist.

Damit haben wir einen quadratischen Aufwand in Abhängigkeit von der Tiefe des gesuchten Pfades. Quadratische Aufwände sind unbedingt zu vermeiden, weil mit zunehmender Datengröße die Laufzeit eines Algorithmus enorm schnell zunimmt.

Wir können mit einem kleinen Trick, auf die Benutzung der Methode `contains` verzichten. Hierzu erinnern wir uns an das Prinzip, einen Fehlschlag durch eine leere Liste zu charakterisieren. Wir fragen einfach jedes Kind nach seinem Pfad zu dem bewußten Objekt. Wenn diese Pfad leer ist, so enthält das Kind diesen Knoten nicht. Sobald ein Kind uns einen nichtleeren Pfad gibt, benutzen wir die überladene Methode `add` der Listenschnittstelle, die uns erlaubt, ein Element an eine beliebige Stelle einer Liste einzufügen.

Tree.java

```
137   public List<a> getPath(a o,List<a> result){
138     //bist Du selbst der Knoten,
```

```
139     //dann ende mit der einelementigen Liste
140     if (mark().equals(o)) { result.add(mark()); return result;}
141
142    for (Tree<a> child:theChildren()){
143       //berechne Pfad vom Kind zum Objekt
144       final List<a> path = child.getPath(o,result);
145
146       //Kind hat einen solchen Pfad
147       if (!path.isEmpty()){
148         //hänge Dich vorne dran
149         result.add(0,mark());
150         //und fertig
151         return result;
152       }
153     }
154     return result;
155   }
```

Da wir oben die akkumulative Methode geschrieben haben, benötigen wir noch eine kapselnde Methode, in der die Ergebnisliste erzeugt wird:

```
1   public List<a> getPath(a o){
2       return getPath(o,new ArrayList<a>());
3   }
```

iterator

`java.util.Iterator iterator()`: erzeugt ein Iterator über alle Knotenmarkierungen des Baumes.

Wir brauchen einen Iterator, der alle Markierungen des Baumes durchläuft. Wir können es und hier sehr einfach machen. Anstatt eine eigene Iteratorklasse zu implementieren, die über die Knoten eines Baumes läuft, können wir einfach unsere schon bestehende Methode `flatten` benutzen, die die Elemente eines Baumes in eine Liste steckt, um dann einfach den Iterator dieser Liste zu benutzen:

Tree.java

```
4   public Iterator<a> iterator(){
5       return flatten().iterator();
6   }
```

sameStructure

`boolean sameStructure(Tree other)`: soll genau dann wahr sein, wenn der Baum `other` die gleiche Struktur hat.

Zur Lösung durchlaufen wir die Struktur beider Bäume in gleicher Weise, so lange bis, wir einen Unterschied in ihrer Struktur gefunden haben. Falls wir, ohne auf eine Widerlegung der Strukturgleichheit zu stoßen, beide Bäume komplett durchlaufen haben, dann haben beide Bäume die gleiche Struktur.

Tree.java

```
  public <b> boolean sameStructure(Tree<b> other){
    //haben beide Bäume gleich viel Kinder.
    if(this.theChildren().size()!=other.theChildren().size())
        //wenn nein, dann sind sie nicht strukturgleich
        return false;

    //Iteriere über die Kinder beider Bäume
    Iterator<Tree<a>> it1=this.theChildren().iterator();
    Iterator<Tree<b>> it2=other.theChildren().iterator();

    //solange es noch Kinder gibt
    while (it1.hasNext()){
      //hohle das nächste Kind beider Bäume
      final Tree<a> thisChild = it1.next();
      final Tree<b> otherChild =it2.next();

      //sind diese nicht strukturgleich, dann breche ab
      if (!thisChild.sameStructure(otherChild)) return false;
    }

    //die Strukturgleichheit wurde nie verletzt
    return true;
  }
```

Die einzige Schwierigkeit, die viele Studenten bei dieser Aufgabe hatten, war gleichzeitig durch zwei Kinderlisten zu iterieren. Hierzu bedarf es zweier Iteratoren, die aber in nur einer Schleife gleichzeitig immer um eins weitergeschaltet werden.

equals

`public boolean equals(Object other)`: soll genau dann wahr sein, wenn `other` ein Baum ist, der die gleiche Struktur und die gleichen Markierungen hat.

Wir können mit der obigen Lösung sehr schnell zu einer Lösung kommen. Entweder wir kopieren die Methode der Strukturgleichheit und fügen noch einen zusätzlichen Test ein, ob die Knotenmarkierungen gleich sind:

```
if (!this.mark().equals(other.mark())) return false;
```

Oder aber, wir prüfen erst auf Strukturgleichheit und vergleichen dann die Listen der Knoten beider Bäume auf Gleichheit.

Tree.java

```
  public <b> boolean equals(Tree<b> other){
      if (!this.sameStructure(other)) return false;
      return this.flatten().equals(other.flatten());
  }
```

Diese Lösung ist zwar schön kurz, aber durchläuft den Baum zweimal: einmal, um die Struktur zu testen und ein zweites Mal um die Markierungen auf Gleichheit zu testen.

In der Aufgabe war die Gleichheitsmethode mit dem Parameter des Typs `Object` verlangt. Hierzu wird erst der Parameter untersucht, ob er überhaupt ein Baum ist, und dann erst mit den eigentlichen Vergleich begonnen:

Tree.java

```
6   public boolean equals(Object other){
7       if (! (other instanceof Tree)) return false;
8       Tree<?> o = (Tree<?>) other;
9       return this.equals(o);
10  }
```

Dies ist eine typische Implementierung für die Gleichheit. Es soll in den meisten Fällen nur Gleiches mit Gleichen verglichen werden.

6.3.5 Eltern und Geschwister

Wir können in unserer Implementierung von einem Knoten nicht mehr seinen Eltern- und seine Geschwisterknoten angeben. Wir können nur einen Baum von der Wurzel an zu den Blätter durchlaufen, aber nicht umgekehrt, von einem beliebiegen Knoten zurück zur Wurzel laufen. Dieses liegt daran, daß wir keinerlei Verbindung von einem Kind zu seinem Elternknoten haben. Wenn wir eine solche haben wollen, müssen wir hierfür ein Feld vorsehen. Wir ergänzen die Klasse `Tree` somit um ein weiteres Feld.

Tree.java

```
11
12  public Tree<a> parent=null;
13
```

Für einen neuen Baumknoten, der neu erzeugt wird, ist das Feld auf den Elternknoten zunächst auf `null` gesetzt. Erst wenn ein Baumknoten als Kind in einen Baum eingehängt wird, ist sein Feld für den Elternknoten auf den entsprechenden Baum zu setzen:

Tree.java

```
14  public Tree(a mark,List<Tree<a>> children){
15    //setze die entsprechenden Felder des Knotens
16    this.mrk=mark;
17    this.chldrn=children;
18
19    //es gibt noch keinen Elternknoten
20    this.parent=null;
21
22    //setze den neuen Knoten als Elternknoten der Kinder
23    respectParents();
24  }
25
26 public void respectParents(){
27   //für jedes Kind
```

```
28     for (Tree<a> child:theChildren()){
29      // ist dieser Knoten jetzt der Elternknoten
30      child.parent=this;
31     }
32   }
```

Jetzt haben wir allerdings eine neue Situation. Bisher konnten wir Baumknoten mehrfach in einen Baum einhängen oder einen Knoten in mehrere Bäume einhängen. Das geht jetzt nicht mehr. Zwei Bäume können sich nicht mehr einen Knoten teilen, denn der Knoten kann ja nur einen Elternknoten speichern. Wir modifizieren jedesmal das Feld `parent`, wenn wir einen Knoten als Kinderknoten einen neuen Baumknoten übergeben.

Aufgabe 26 Schreiben Sie eine Methode `List<Tree<a>> siblings()`, die die Geschwister eines Knotens als Liste ausgibt. In dieser Liste der Geschwister soll der Knoten selbst nicht auftauchen.

6.3.6 Modifizierende Methoden

Bisher haben wir eine rein funktionale Umsetzung von Bäumen erarbeitet. Einmal erzeugte Bäume sind unveränderbar. Die einzige Ausnahme war hierbei das Feld `parent`, das geändert wurde, sobald ein Knoten in einen Baum eingehängt wurde.

Die Felder der Klasse `Tree` sind privat. Es gibt keine Methoden, die sie verändern. Ebenso war auch unsere Listenimplementierung gehalten. Die Java Standardsammlumgsklassen haben im Gegensatz dazu Methoden, die ein Listenobjekt verändern.

Wir können für unsere Klasse `Tree` auch solche modifizierenden Methoden implementieren. Solche Methoden fügen typischer Weise einen neuen Knoten in einen Baum ein oder löschen Knoten. Da es modifizierende Methoden sind, empfiehlt es sich, sie als `void`-Methoden zu schreiben.

addChild

Wir schreiben eine Methode, die an einen Knoten als letztes Kind einen neuen Knoten einfügt:

Tree.java

```
33 public void addChild(Tree<a> newChild){
34   List<Tree<a>> chs = theChildren();
35   newChild.parent=this;
36   chs.add(newChild);
37 }
```

Es wird direkt auf dem Feld mit der Liste der Kinder operiert und die modifizierende methode `add` aus der Schnitstelle `List` benutzt.

addLeaf

Eine weitere Methode fügt dem Baum ein neues Blatt ein:

Tree.java

```
38 public void addLeaf(a o){addChild(new Tree<a>(o));}
```

deleteChild

Folgende Methode löscht das n-te Kind aus einem Knoten:

Tree.java

```
39 public void deleteChild(int n){
40    theChildren().remove(n);
41 }
```

In dieser Implementierung wird die modifizierte Liste durch die Methode `theChildren()` ermittelt, anstatt auf das Feld `chldrn` direkt zuzugreifen.

setMark

Das Feld `mrk` hat das Attribut `private`. Wollen wir seinen Wert aus einem anderen Kontext ändern, so brauchen wir eine Methode, die dieses macht. Wir können eine typische `Set`-Methode schreiben:

Tree.java

```
42 public void setMark(a mark){mrk=mark;}
```

Beispiel: Baumerzeugung mit modifizierenden Methoden

SkriptTree.java

```
1  package name.panitz.data.tree.example;
2
3  import  name.panitz.data.tree.Tree;
4  public class SkriptTree{
5    public static Tree<String> getSkript(){
6      Tree<String> skript = new Tree<String>("Programmieren 2");
7
8      Tree<String> kap1=new Tree<String>("Frühe und späte Bindung");
9      Tree<String> kap2
10      = new Tree<String>
11         ("Graphische Benutzeroberflächen und Graphiken");
12     Tree<String> kap3
13      = new Tree<String>("Abstrakte Datentypen");
14
15     skript.addChild(kap1);
16     skript.addChild(kap2);
17     skript.addChild(kap3);
18
```

```
19      kap1.addLeaf("Wiederholung: Das Prinzip der späten Bindung");
20      kap1.addLeaf("Keine späte Bindung für Felder");
21      kap1.addLeaf("Keine Späte Bindung für überladene Methoden");
22
23      Tree<String> sec2_1
24       = new Tree<String>("Graphische Benutzeroberflächen");
25      Tree<String> sec2_2
26       = new Tree<String>("Exkurs: verschachtelte Klassen");
27      Tree<String> sec2_3
28       =new Tree<String>("Selbstdefinierte graphische Komponenten");
29      Tree<String> sec2_4
30       = new Tree<String>("Weitere Komponente und Ereignisse");
31      Tree<String> sec2_5
32       = new Tree<String>("Swing und Steuerfäden");
33
34      kap2.addChild(sec2_1);kap2.addChild(sec2_2);
35      kap2.addChild(sec2_3);kap2.addChild(sec2_4);
36      kap2.addChild(sec2_5);
37
38      Tree<String> sec3_1
39       = new Tree<String>("Bäume");
40      Tree<String> sec3_2
41       = new Tree<String>("Algorithmen auf Bäumen");
42      Tree<String> sec3_3
43       = new Tree<String>("Binärbäume");
44      Tree<String> sec3_4
45       = new Tree<String>("Binäre Suchbäume");
46      Tree<String> sec3_5
47       = new Tree<String>("XML-Dokumente als Bäume");
48      Tree<String> sec3_6
49       = new Tree<String>("Keller");
50
51      kap3.addChild(sec3_1);kap3.addChild(sec3_2);
52      kap3.addChild(sec3_3);kap3.addChild(sec3_4);
53      kap3.addChild(sec3_5);kap3.addChild(sec3_6);
54
55      sec2_1.addLeaf("Graphische Komponenten");
56      sec2_1.addLeaf("Ereignisbehandlung");
57      sec2_1.addLeaf("Setzen des Layouts");
58
59      sec2_2.addLeaf("Innere Klassen");
60      sec2_2.addLeaf("Anonyme Klassen");
61      sec2_2.addLeaf("Statisch verschachtelte Klassen");
62
63      sec2_3.addLeaf("Graphics Objekte");
64      sec2_3.addLeaf("Dimensionen");
65      sec2_3.addLeaf("Farben");
66      sec2_3.addLeaf("Fonts und ihre Metrik");
67      sec2_3.addLeaf("Erzeugen graphischer Dateien");
68      sec2_3.addLeaf("Graphics2D");
```

```
69
70      sec2_4.addLeaf("Mausereignisse");
71      sec2_4.addLeaf("Fenstereignisse");
72      sec2_4.addLeaf("Weitere Komponenten");
73
74      sec2_5.addLeaf("Timer in Swing");
75      sec2_5.addLeaf
76       ("SwingWorker für Steuerfäden in der Ereignisbehandlung");
77      return skript;
78    }
79  }
```

Aufgabe 27 Schreiben Sie eine modifizierende Methode
`void deleteChildNode(int n)`,
die den n-ten Kindknoten löscht, und stattdessen die Kinder des n-ten Kindknotens als neue Kinder mit einhängt.
Beispiel:

```
          a                 wird durch                          a
         / \                deleteChildNode(1)                 /|\
        /   \               zu:                               / | \
       b     c                                               /  |  \
      / \   / \                                             /   |   \
     d   e  f  g                                           b    f    g
           /|\                                            /|   /|\
          h i j                                          d e  h i j
```

6.4 Binärbäume

Listen sind Bäume, in denen jeder Knoten maximal ein Kind hat. Binärbäume sind Bäume, in denen jeder Knoten maximal zwei Kinder hat. Wir können eine Unterklasse von `Tree` schreiben, in der ein Konstruktor sicherstellt, daß ein Knoten maximal zwei Kinder hat.

Wir wollen zusätzlich zulassen, daß es ein rechtes aber kein linkes Kind gibt. Dafür stellen wir zwei Felder zur Verfügung, die die entsprechenden Kinder enthalten. Existiert ein Kind nicht, so sei das entsprechende Feld mit `null` belegt.

Damit wir die geerbten Methoden fehlerfrei benutzen können, müssen wir sicherstellen, daß die Methode `children` auch für diese Unterklasse gemäß der Spezifikation funktioniert.

BinTree.java

```
1  package name.panitz.data.tree;
2
3  import java.util.List;
4  import java.util.ArrayList;
5
6  public class BinTree<a> extends Tree<a>{
7
8    //private Felder für rechtes und linkes Kind
9    private BinTree<a> rght;
10   private BinTree<a> lft;
```

Wir sehen zwei naheliegende Konstuktoren vor:

BinTree.java

```
11  public BinTree(BinTree<a> left,a mark,BinTree<a> right){
12    super(mark);lft=left; rght=right;}
13
14  public BinTree(a mark){this(null,mark,null);}
```

Zwei Selektormethoden greifen auf das linken bzw. rechte Kind zu:

BinTree.java

```
15  public BinTree<a> left(){return lft;}
16  public BinTree<a> right(){return rght;}
```

Wir müssen die Methode `theChildren` überschreiben. Die Kinder befinden sich nicht mehr in einer Liste gespeichert sondern in den zwei speziellen hierfür angelegten Feldern:

BinTree.java

```
17  public List<Tree<a>> theChildren(){
18    List<Tree<a>> result = new ArrayList<Tree<a>>();
19    if (left()!=null)  result.add(left());
20    if (right()!=null) result.add(right());
21    return result;
22  }
```

6.4.1 Probleme der Klasse BinTree

Als Unterklasse der Klasse `Tree` erben wir sämtliche Methoden der Klasse `Tree`. Das ist zunächst einmal positiv, weil wir automatisch Methoden wie `count` oder `flatten` erben und nicht neu zu implementieren brauchen.

Leider spielen uns die modifizierenden Methoden der Klasse `Tree` dabei einen Streich. Kurz gesagt: sie funktionieren nicht für die Klasse `BinTree`. Sie modifizieren die Liste der Kinder, wie wir sie von der Methode `theChildren()` erhalten. Diese Liste ist in der Klasse `BinTree` transient, d.h. sie wird nicht im Objekt direkt gespeichert, sondern jeweils neu aus den beiden Feldern `rght` und `lft` erzeugt. Eine Modifikation an dieser Liste hat also keinen bleibenden (persistenten) Effekt.

Andererseits ist dieses eine gute Eigenschaft, denn damit wird verhindet, daß eine modifizierende Methode die Binäreigenschaft zerstört. Die Methode `addChild` könnte sonst einen binären Baum an einem Knoten ein drittes Kind einfügen.

Es empfiehlt sich trotzdem, dieses Problem nicht stillschweigend hinzunehmen, und eine entsprechende Ausnahme zu werfen. Daher fügen wir der Klasse `BinTree` folgende Variante der Methode `addLeaf` hinzu:

BinTree.java

```
23 public void addLeaf(a o){
24     throw new UnsupportedOperationException();
25 }
```

Für die Methode `deleteChild` können wir eine Variante anbieten, die auf binären Bäumen funktioniert.

BinTree.java

```
26 public void deleteChild(int n){
27   if (n==0) lft=null;
28   else if (n==1) rght=null;
29   else throw new IndexOutOfBoundsException ();
30 }
```

Modifizierende Methoden, die ungefährlich für die Binärstrukrur der Bäume sind, setzen die beiden Kinder eines Baumes neu:

BinTree.java

```
31   void setLeft(BinTree<a> l){ lft=l;}
32   void setRight(BinTree<a> r){ rght=r;}
```

Aufgabe 28 Überschreiben Sie die Methoden `addLeaf` und `deleteChildNode` in der Klasse `BinTree`, so daß sie nur eine Ausnahme werfen, wenn die Durchführung der Modifikation dazu führen würde, daß das Objekt, auf dem die Methode angewendet wird, anschließend kein Binärbaum mehr wäre.

6.4.2 Linearisieren binärer Bäume

In der Klasse `Tree` haben wir eine Methode definiert, die die Knoten eines Baumes in einer linearisierten Form in eine Liste speichert. Wir haben uns bisher keine Gedanken gemacht, in welcher Reihenfolge die Baumknoten in der Ergebnisliste stehen. Hierbei sind mehrere fundamentale Ordnungen der Knoten des Baumes vorstellbar:

Preordnung

Unsere bisherige Implementierung der Methode `flatten` benutzt die Reihenfolge, in der ein Knoten in der Ergebnisliste immer vor seinen Kindern steht und Geschwisterknoten stets erst nach den Knoten folgen. Diese Ordnung nennt man Preordnung. In der Preordnung kommt die Wurzel eines Baumes als erstes Element der Ergebnisliste.

Postordnung

Wie der Name vermuten läßt, ist die Postordnung gerade der umgekehrte Weg zur Preordnung. Jeder Knoten steht in der Liste nach allen Kinderknoten. Die Wurzel wird damit das letzte Element der Ergebnisliste.

Aufgabe 29 Schreiben Sie analog zur Methode `flatten` in der Klasse `Tree` eine Methode `List<a> postorder()`, die die Knoten eines Baumes in Postordnung linearisiert.

Inordnung

Pre- und Postordnung kann man nicht nur auf Binärbäumen definieren, sondern allgemein für alle Bäume. Die dritte Variante hingegen, die Inordnung, kann nur für Binärbäume sinnvoll definiert werden. Hier steht jeder Knoten nach allen Knoten seines linken Kindes und vor allen Knoten seines rechten Kindes.

BinTree.java

```
33 public List<a> inorder(){
34   //Ergebnisliste
35   List<a> result = new ArrayList<a>();
36
37   //gibt es ein linkes Kind, füge dessen
38   //Linearisierung hinzu
39   if (left()!=null) result.addAll(left().inorder());
40
41   //dann den Knoten selbst
42   result.add(mark());
43
44   //und gegebenenfalls dann das recte Kind
45   if (right()!=null) result.addAll(right().inorder());
46
47   return result;
48 }
```

Pre-, Post- und Inordnung entsprechen der Pre-, Post- und Infixschreibweise von Operatorausdrücken. Betrachtet man einen Operatorausdruck als Baum mit dem Operator als Wurzel und den beiden Operanden als Kindern, dann ergibt sich die Analogie sofort.

Militärordnung

Eine Ordnung, die der Baumstruktur sehr entgegen steht, ist die sogenannte Militärordnung. Diese Ordnung geht ebenenweise vor. Geschwisterknoten stehen in der Liste direkt nebeneinander. Abbildung 6.10 veranschaulicht diese Ordnung:

Zunächst wird die Wurzel genommen, dann der Reihe nach alle Kinder der Wurzel, dann die Kinder der Kinder usw. Es werden also alle Knoten einer Generation hintereinander geschrieben. Diese Ordnung entspricht nicht der rekursiven Definition einer Baumstruktur. Daher ist ihre Implementierung auch nicht sehr elegant. Hierfür sind ein paar Hilfslisten zu verwalten. Eine Liste für die aktuelle Generation und eine in der die Knoten der nächsten Generation gesammelt werden.

BinTree.java

```
49 public List military(){
50   //Ergebnisliste enthält erst diesen Knoten
51   List<a> result=new ArrayList<a>();
52   result.add(mark());
53
54   //Wir speichern uns die Knoten einer
55   //aktuellen Generation
56   List<Tree<a>> currentGeneration = theChildren();
```

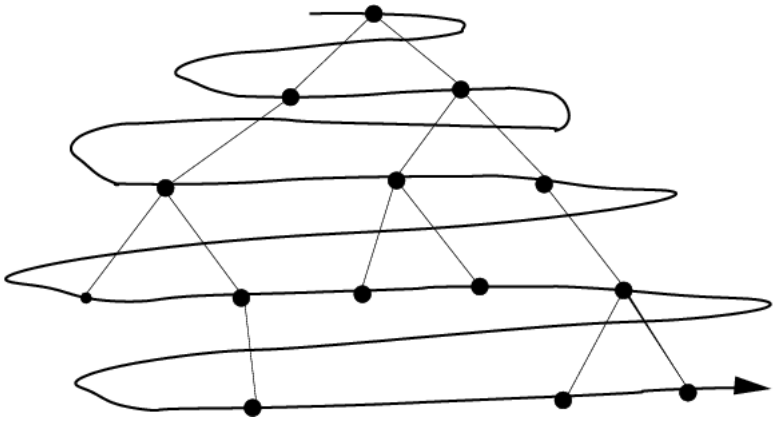


Abbildung 6.10: Schematische Darstellung der Militärordnung.

```
57
58    //Solange es in der aktuellen Generation Knoten gibt
59    while (!currentGeneration.isEmpty()){
60      //erzeuge die nächste Generation
61      List<Tree<a>> nextGeneration=new ArrayList<Tree<a>>();
62
63      for (Tree<a> nextChild:currentGeneration){
64        nextGeneration.addAll(nextChild.theChildren());
65
66        //Füge das Elemente der aktuellen Generation
67        //zum Ergebnis
68        result.add(nextChild.mark());
69      }
70      //schalte eine Generation tiefer
71      currentGeneration=nextGeneration;
72    }
73    return result;
74  }}
```

Aufgabe 30 Rechnen Sie auf dem Papier ein Beispiel für die Arbeitsweise der Methode `military()`.

6.5 Binäre Suchbäume

Binärbäume können dazu genutzt werden, um Objekte effizient gemäß einer Ordnung zu speichern und effizient nach ihnen unter Benutzung dieser Ordnung wieder zu suchen. Die Knotenmarkierungen sollen hierzu nicht mehr beliebige Objekte sein, sondern Objekte, die die Schnittstelle `Comparable` implementieren. Solche Objekte können mit beliebigen Objekten verglichen werden, so daß sie eine Ordnungsrelation haben.

Binäre Bäume sollen diese Eigenschaft so ausnutzen, daß für einen Knoten gilt:

- jede Knotenmarkierung eines linken Teilbaums ist kleiner oder gleich als die Knotenmarkierung.
- jede Knotenmarkierung eines rechten Teilbaums ist größer als die Knotenmarkierung.

Man erkennt, daß die Konstruktion solcher Bäume nicht mehr lokal geschehen kann, indem der linke und der rechte Teilbaum in einem neuen Knoten zusammengefasst werden. Bei der Baumkonstruktion ist ein neuer Knoten entsprechend der Ordnung an der richtige Stelle als Blatt einzuhängen.

Wir schreiben eine Unterklasse `SearchTree` der Klasse `BinTree`. Diese hat nur einen Konstruktor, um einen Baum mit nur einem Wurzelknoten zu konstruieren. Weitere Knoten können mit der Methode `insert` in den Baum eingefügt werden. Die Methode `insert` stellt sicher, daß der Baum die größer Relation der Elemente berücksichtigt. Die Elemente des Baumes müssen die Schnittstelle `Comparable` implementieren.

SearchTree.java

```
1  package name.panitz.data.tree;
2
3  public class SearchTree<a extends Comparable<a>>
4                                  extends BinTree<a>{
```

Wir sehen nur einen Konstruktor vor, der ein Blatt erzeugt

SearchTree.java

```
5    SearchTree(a o){super(o);}
```

Suchbäume wachsen durch eine modifizierende Einfügeoperation:

SearchTree.java

```
6    public void insert(a o){
7      if (o.compareTo(mark())<=0){
8        //neuer Knoten ist kneiner, also in den
9        //linken Teilbaum
10       if (left()==null){
11         setLeft(new SearchTree<a>(o));
12       }else
13         ((SearchTree<a>)left()).insert(o);
14     }else{
15       //wenn der neue Knoten also größer, dann dasselbe
16       //im rechten Teilbaum
17       if (right()==null){
18          setRight(new SearchTree<a>(o));
19       }else
20         ((SearchTree<a>)right()).insert(o);
21     }
22   }
23
```

Wir können für interne Zwecke einen Konstruktor vorsehen, der einen Suchbaum direkt aus den Kindern konstruiert. Hierbei wollen wir uns aber nicht darauf verlassen, daß dem Konstruktor solche Teilbäume und eine solche Knotenmarkierung übergeben werden, daß der resultierende Baum ein korrekter Suchbaum ist.

SearchTree.java

```
24 private SearchTree
25            (SearchTree<a> l,a o,SearchTree<a> r){
26   super(l,o,r);
27   try {
28    if (!(   o.compareTo(l.mark())>=0
29          && o.compareTo(r.mark())<=0))
30      throw new IllegalArgumentException
31      ("ordering violation in search tree construction");
32   }catch(NullPointerException _){
33   }
34 }
```

Die `if`-Bedingung mit der Prüfung auf eine Eigenschaft der Parameter ist ein typischer Fall für eine Zusicherung, wie sie seit Javas Version 1.4 ein fester Bestandteil von Java ist.

Die Linearisierung eines binären Suchbaums in Inordnung ergibt eine sortierte Liste der Baumknoten, wie man sich mit folgenden Test vergegenwärtigen kann:

TestSearchTree.java

```
1  package name.panitz.data.tree;
2  import javax.swing.*;
3
4  public class TestSearchTree {
5    public static void main(String [] args){
6      SearchTree<String> t = new SearchTree<String>("otto");
7      t.insert("sven");
8      t.insert("eric");
9      t.insert("lars");
10     t.insert("uwe");
11     t.insert("theo");
12     t.insert("otto");
13     t.insert("kai");
14     t.insert("henrik");
15     t.insert("august");
16     t.insert("berthold");
17     t.insert("arthur");
18     t.insert("arno");
19     t.insert("william");
20     t.insert("tibor");
21     t.insert("hassan");
22     t.insert("erwin");
23     t.insert("anna");
24     System.out.println(t.inorder());
25
26 /*    JFrame frame = new JFrame("Baum Fenster");
27     frame.getContentPane().add(new DisplayTree(t));
28     frame.pack();
29     frame.setVisible(true);*/
30   }
```

```
31 }
```

Das Programm führt zu folgender Ausgabe auf der Kommandozeile.

```
sep@swe10:~/fh/prog2/beispiele/Tree> java TestSearchTree
[anna, arno, arthur, august, berthold, eric, erwin, hassan, henrik,
kai, lars, otto, otto, sven, theo, tibor, uwe, william]
sep@swe10:~/fh/prog2/beispiele/Tree>
```

Aufgabe 31 Zeichnen Sie schrittweise die Baumstruktur, die im Programm `TestSearchTree` aufgebaut wird.

Beweis der Sortiereigenschaft

Wir haben oben behauptet und experiementell an einem Beispiel ausprobiert, daß die Inordnung eines binären Suchbaums eine sortierte Liste als Ergebnis hat. Wir wollen jetzt versuchen diese Aussage zu beweisen.

Zum Beweis einer Aussage über eine rekursiv definierten Datenstruktur (wie Listen und Bäume) bedient man sich der vollständigen Induktion.[6] Hierbei geht die Induktion meist über die Größe der Datenobjekte. Als Induktionsanfang dient das kleinste denkbare Datenobjekt, bei Listen also leere Listen, bei Bäumen Blätter. Der Induktionsschluß setzt dabei voraus, daß die Aussage für alle kleineren Objekte bereits bewiesen ist, also für alle Teilobjekte (Teilbäume und Teillisten) bereits gilt.

Beweisen wir entsprechend die Aussage, daß die Liste der Elemente eines binären Suchbaums in Inordnung sortiert ist:

- **Anfang:** Der Induktionsanfang behauptet, die Inordung eines Blattes ist sortiert. Für ein Blatt k gilt: $\mathit{left}(k) = \mathit{null}$ und $\mathit{right}(k) = \mathit{null}$. Es folgt, daß die Inordung nur ein Element, nämlich $\mathit{mark}(k)$, enthält. Einelementige Listen sind immer sortiert.

- **Schritt:** Wir wollen die Aussage beweisen für alle Bäume k mit: $\mathit{count}(k) = n$.

  Annahme: die Aussage ist wahr für alle Bäume k' mit $\mathit{count}(k') < n$.

  Die Definition der Inordnung ist:
  $\mathit{inorder}(k) = (\mathit{inorder}(\mathit{left}(k)), \mathit{mark}(k), \mathit{inorder}(\mathit{right}(k)))$.

  Für endliche Bäume gilt:
  $\mathit{count}(\mathit{left}(k)) < n$ und $\mathit{count}(\mathit{right}(k)) < n$
  Nach der Induktionsvoraussetzung gilt damit, daß
  $\mathit{inorder}(\mathit{left}(k))$ und $\mathit{inorder}(\mathit{right}(k))$
  sortierte Listen sind.

  Über die Definition der binären Suchbäume gilt:
  für alle $x \in \mathit{inorder}(\mathit{left}(k))$ gilt $x \leq \mathit{mark}(k)$
  und für alle $x \in \mathit{inorder}(\mathit{right}(k))$ gilt $x > \mathit{mark}(k)$

  Damit ist insbesondere die Liste $\mathit{inorder}(k)$ sortiert.

[6] Die im philosophischen Sinne keine Induktion sondern eine Deduktion ist.

6.5.1 Suchen in Binärbäumen

Die Sortiereigenschaft der Suchbäume erlaubt es jetzt, die Methode `contains` so zu schreiben, daß sie maximal einen Pfad durchlaufen muß. Es muß nicht mehr der ganze Baum betrachtet werden.

SearchTree.java

```
32 public boolean contains(a o){
33   //bist du es schon selbst
34   if (mark().equals(o)) return true;
35
36   //links oder rechts suchen:
37   final int compRes = mark().compareTo(o);
38
39   if (compRes>0){
40     //keine Knoten mehr. Dann ist er nicht enthalten
41     if (left()==null) return false;
42     //Sonst such weiter unten im Baum
43     else return ((SearchTree<a>)left()).contains(o);
44   }
45
46   if (compRes<0){
47     if (right()==null) return false;
48     else return ((SearchTree<a>)right()).contains(o);
49   }
50   return false;
51 }
```

Wie man sieht, ist die Methode in ihrer Struktur analog zur Methode `insert`.

6.5.2 Entartete Bäume

Bäume, die keine eigentliche Baumstruktur mehr haben, sondern nur noch Listen sind, werden manchmal auch als entartete Bäume bezeichnet. Insbesondere bei binären Suchbäumen verlieren wir die schönen Eigenschaften, nämlich daß wir maximal in der maximalen Pfadlänge im Baum zu suchen brauchen.

Aufgabe 32 Erzeugen Sie ein Objekt des typs `SearchTree` und fügen Sie nacheinander die folgenden Elemente ein:
`"anna","berta","carla","dieter","erwin","florian","gustav"`
Lassen Sie anschließend die maximale Tiefe des Baumes ausgeben.

Balanzieren von Bäumen

Wie wir in der letzten Aufgabe sehen konnnten, liegt es an der Reihenfolge, in der die Elemente in einem binären Suchbaum eingefügt werden, wie gut ein Baum ausbalanziert ist. Wir sprechen von einem ausbalanzierten Baum, wenn sein linker und rechter Teilbaum die gleiche Tiefe haben.

Man kann einen Baum, der nicht ausbalanziert ist, so verändern, daß er weiterhin ein korrekter binärer Suchbaum in Bezug auf die Ordnung ist, aber die Tiefen der linken und rechten Kinder ausgewogen sind, d.h. sich nicht um mehr als eins unterscheiden.

Beispiel:
Betrachten Sie den Baum aus Abbildung 6.11

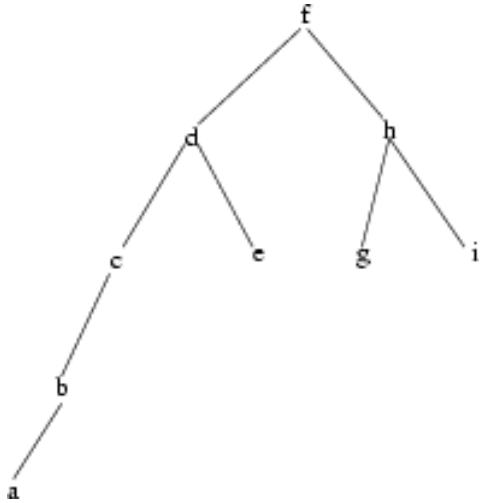


Abbildung 6.11: Nicht balanzierter Baum.

Der linke Teilbaum hat eine Tiefe von 4, der rechte eine Tiefe von 2. Wir können den Baum so verändern, daß beide Kinder eine Tiefe von 3 haben. Wir bekommen den Baum aus Abbildung 6.12

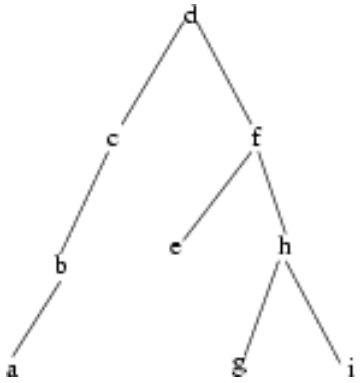


Abbildung 6.12: Baum nach Rotierungsschritt.

Das linke Kind dieses Baumes ist auch nicht ausbalanziert, wir können den Baum noch einmal verändern und erhalten den Baum in Abbildung 6.13:

Spezifikation Zum Ausbalanzieren wird ein Baum gedreht. Einer seiner Kinderknoten wird die neue Wurzel und die alte Wurzel dessen Kind. Es gibt zwei Richtungen, in die gedreht werden

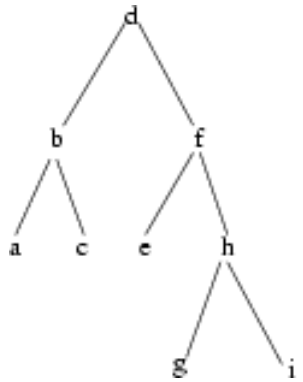


Abbildung 6.13: Baum nach weiteren Rotierungsschritt.

kann: mit und gegen den Urhzeigersinn. Schematisch lassen sich diese beiden Operationen mit folgenden Gleichungen spezifizieren:

- rotateRight(Node(Node(l2,x2,r2),x1,r1)) = Node(l2,x2,Node(r2,x1,r1))
- rotateLeft(Node(l1,x1,Node(l2,x2,r2))) = Node(Node(l1,x1,l2),x2,r2)

In graphischer Darstellung wird die erste Gleichung zu:

```
      x1                                          x2
     / \        wird mit rotateRight zu:         / \
    /   \                                      /   \
   x2   r1                                    l2   x1
  / \                                             / \
 l2 r2                                           r2 r1
```

Die Operationen erhalten die Eigenschaft eines Suchbaumes: linke Kinder sind kleiner, rechte Kinder größer als die Wurzel. Die Linearisierung in der Inordnung bleibt gleich. Die maximale Tiefe der Kinder verändert sich um 1. Die Anzahl der Knoten im Baum ändert sich nicht.

Implementierung Wir können generell zwei Umsetzungen des oben spezifizierten Algorithmus wählen: eine die einen neues Baumobjekt erzeugt und keinde der bestehenden Bäume modifiziert und eine, die die Felder der bestehenden Baumknoten so umändert, daß das Objekt, auf dem die Methode aufgerufen wird, modifiziert wird.

Funktionale Umsetzung Für die funktionale Umsetzung können wir eine statische Methode schreiben. Sie bekommt einen Baum und soll einen neuen Baum, der durch Rotation aus den Eingabebaum entstanden ist, als Ergebnis liefern.

Die Umsetzung leitet sich direkt aus der Spezifikation über eine Gleichung ab. Wir benutzen dieselben Bezeichner wie in der Spezifikation.

SearchTree.java

```
52 public static <a extends Comparable<a>> SearchTree<a>
53                                rotateRight(SearchTree<a> t){
54   try{
```

```
55      //speichere die einzelnen Teile in die Bezeichner,
56      //die wir in der Spezifikation benutzt haben
57      final SearchTree<a> l2 = (SearchTree<a>)t.left().left();
58      final SearchTree<a> r2 = (SearchTree<a>)t.left().right();
59      final SearchTree<a> r1 = (SearchTree<a>)t.right();
60      final a x1 = t.mark();
61      final a x2 = t.left().mark();
62
63      return new SearchTree<a>
64                   (l2,x2,new SearchTree<a>(r2,x1,r1));
65    }catch (NullPointerException _){
66        return t;
67    }
68  }
```

Die Ausnahme tritt auf, wenn das linke Kind nicht existiert, also hier der Wert `null` steht. Dann kann nicht nach rechts rotiert werden (da links nichts zum Rotieren steht). In dem Fall, das nicht rotiert werden kann, wird der Eingabebaum unverändert ausgegeben.

Modifizierende Umsetzung Wir können die modifizierende Methode zum Rotieren eines Baumes in der Klasse `BinTree` implementieren. Wir benutzen wieder die Bezeichner aus unserer Spezifikation. Anstatt aber wie eben neue Knoten zu erzeugen, setzen die entsprechenden Referenzen um.

SearchTree.java

```
69  void rotateRight(){
70    try{
71      BinTree<a> l1 = left();
72      BinTree<a> l2 = l1.left();
73      BinTree<a> r2 = l1.right();
74      BinTree<a> r1 = right();
75      a  x1 = mark();
76      a  x2 = l1.mark();
77
78      setMark(x2);
79      setLeft(l2);
80      setRight(l1);
81      l1.setMark(x1);
82      l1.setLeft(r2);
83      l1.setRight(r1);
84    }catch (NullPointerException _){
85    }
86  }}
```

Die Ausnahme wird schon in der zweiten Zeile geworfen, bevor Referenzen verändert wurden. Das Objekt bleibt unverändert. Ansonsten werden die Knoten entsprechend der Spezifikation umgehängt.

Aufgabe 33

a) Schreiben sie entsprechend die Methode:
static public SearchTree rotateLeft(SearchTree t)

b) Schreiben Sie in der Klasse `BinTree` die entsprechende modifizierende Methode `rotateLeft()`

c) Testen Sie die beiden Rotierungsmethoden. Testen Sie, ob die Tiefe der Kinder sich verändert und ob die Inordnung gleich bleibt. Testen Sie insbesondere auch einen Fall, in dem die `NullPointerException` abgefangen wird.

Kapitel 7

Java Standardklassen

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

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

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

7.1 Die Klasse Objekt

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

7.1.1 Die Methoden der Klasse Object

equals

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

EqualVsIdentical.java

```
class EqualVsIdentical {

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

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

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

}
```

Die Ausgabe dieses Tests ist:

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

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

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

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

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

hashCode

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

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

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

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

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

clone

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

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

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

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

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

NoClone.java

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

Dieses Programm wirft eine Ausnahme:

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

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

Clone.java

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

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

Dieses Programm führt zu einem Programmdurchlauf ohne Ausnahme:

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

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

finalize

`protected void finalize() throws Throwable` ist eine Methode, die ein ähnliches Anwendungsgebiet wie die `finally`-Klausel in einer Ausnahmebehandlung hat. Wenn Java ein Objekt in seinem Speicher findet, von dem Java zeigen kann, daß es von keinem laufenden Programmteil mehr genutzt wird, dann löscht Java dieses Objekt aus dem Speicher. Bevor das Objekt

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

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

Finalize.java
```
1  class Finalize {
2     int i;
3     Finalize(int i){this.i=i;}
4     protected void finalize(){
5       System.out.print(i+",");
6     }
```

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

Finalize.java
```
7     public static void main(String [] _){
8       for (int i=0;i<100000;i=i+1){
9         new Finalize(i);
10        System.out.print("*");
11      }
12    }
13 }
```

Aufgabe 34 Übersetzen und starten Sie die Klasse `Finalize` und beobachten Sie, wie und wann Java Objekte aus dem Speicher löscht.

toString

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

weitere Methoden

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

7.2 String und Stringbuffer

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

Der Unterschied ist leicht durch das folgende Programm zu verdeutlichen:

StringVsStringBuffer.java

```
class StringVsStringBuffer {

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

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

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

ReturnString.java

```
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){
```

```
22      System.out.println(toRoman(1999));
23      System.out.println(toRoman(494));
24    }
25
26  }
```

7.3 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.

7.3.1 Listen

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

- `add` zum Hinzufügen von Elementen.
- `clear` zum Löschen aller Elemente.
- `remove` zum Löschen einzelner Elemente.

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

Listenklassen

Listen sind mit Hilfe einer abstrakten Klasse implementiert. Die eigentlichen konkreten Klassen, die Listen implementieren, sind: `ArrayList`, `LinkedList` und `Vector`. Dabei ist `ArrayList` die gebräuchlichste Implementierung. `Vector` ist eine ältere Implementierung, die als Nachteil hat, daß sie stets eine Synchronisation für nebenläufige Steuerfäden vornimmt, die in der Regel nicht unbedingt benötigt wird.

Folgende Klasse zeigt, wie eine Liste erzeugt wird und ihr nach und nach Elemente hinzugefügt werden:

ListUsage.java

```
1  import java.util.List;
2  import java.util.ArrayList;
3
4  class ListUsage{
```

```
5
6   public static void main(String [] args){
7     List<String> xs = new ArrayList<String>();
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.

7.3.2 Mengen

Im Gegensatz zu Listen können Mengenobjekte keine Elemente doppelt enthalten. Hierbei werden zwei Objekte als gleich betrachtet, wenn der Aufruf der Methode `equals` für sie den Wert `true` ergibt. Daraus folgt insbesondere, daß es nur sinnvoll ist, Elemente zu einer Menge zuzufügen, für die auch eine sinnvolle Überschreibung der Methode `equals` existiert.

Die Klasse `HashSet` ist eine Implementierung der Schnittstelle `Set`. Sie basiert darauf, daß die Objekte, die in der Menge gespeichert werden sollen, eine möglichst gute Implementierung der Methode `hashCode` haben. Möglichst gut bedeutet in diesem Fall, daß in möglichst vielen Fällen gilt:

> Aus `!x.equals(y)` folgt `x.hashCode != y.hashCode()`

7.3.3 Iteratoren

Java stellt eine Schnittstelle `Iterator` bereit. Ihr einziger Zweck ist, zu beschreiben, daß durch eine Sammlung von Elementen durchiteriert werden kann.

Ein Iterator kann nacheinander gebeten werden, sein nächstes Element auszugeben, und ein Iterator kann gefragt werden, ob es noch weitere Elemente gibt.

Hierzu gibt es die entsprechenden Methoden:

- `Object next()`, die den Iterator um ein Element weiterschaltet und das aktuelle Element als Ergebnis hat.
- `boolean hasNext()`, die `true` als Ergebnis hat, wenn die Methode `next` ein weiteres Element liefert.

Iteratoren sind zum einmaligen Gebrauch gedacht. Sie werden benutzt, um einmal durch die Elemente einer Sammlung durchzuiterieren.

Iterator für Li

Am einfachsten können wir das Prinzip eines Iteratorobjekts verstehen, wenn wir eine entsprechende Iteratorklasse für unsere eigenen Listen schreiben:

LiIterator.java

```
package name.panitz.data.list;

public class LiIterator<a> implements java.util.Iterator<a>{

  public LiIterator(Li<a> xs){this.xs=xs;}

  private Li<a> xs;

  public boolean hasNext() {return !xs.isEmpty();}

  public a next(){
    a result = xs.head();
    xs = xs.tail();
    return result;
  }

  public void remove(){
   throw new UnsupportedOperationException ();
  }
}
```

Der Klasse `Li` können wir eine Methode hinzufügen, die den entsprechenden Iterator für die Liste zurückgibt:

```
  public java.util.Iterator<a> iterator(){
    return new LiIterator<a>(this);
  }
```

Benutzung von Iteratoren

Iteratoren sind dafür gedacht, über die Elemente von Sammlungsklassen zu iterieren. Hierzu bedient man sich normaler Weise der `for`-Schleife. Sie wird hierzu mit einem Iterator initialisiert und hat als Bedingung, daß der Iterator noch ein weiteres Element liefert. Die Weiterschaltung wird in der Regel leer gelassen, weil Iteratoren mit dem Aufruf der Methode `next` beides machen: das nächste Element ausgeben und den Iterator weiterschalten.

Wir können als kleines Beispiel eine kleine Methode schreiben, die die Elemente einer Sammlung als `String` aneinanderhängt:

Iterate.java

```
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;

class Iterate {
  static public String unwords(Collection<String> cs){
    String result = "";

    for (String n:cs){
      result = result+" "+cs;
    }
    return result;
  }

  static public void main(String [] _){
    Collection<String> cs = new ArrayList<String>();
    cs.add("lebt");
    cs.add("wohl");
    cs.add("gott");
    cs.add("weiß");
    cs.add("wann");
    cs.add("wir");
    cs.add("uns");
    cs.add("wiedersehen");

    System.out.println(unwords(cs));
  }
}
```

7.3.4 Abbildungen

Abbildungen assoziieren Elemente einer Art mit Elementen einer anderen Art. Sie sind vergleichbar mit Wörterbüchern. In einem deutsch/englischen Wörterbuch werden die deutschen Wörter auf die englischen Wörter mit der entsprechenden Bedeutung abgebildet.

Wir benutzen eine Abbildung, indem wir für einen Schlüssel (dem deutschen Wort) den entsprechenden Werteeintrag suchen (das englische Wort).

Abbildungen als Liste von Paaren

Da eine Abbildung jeweils ein Element mit einem anderem assoziiert, ist eine Umsetzung als Liste von Paaren naheliegend.[3] Ein Paar besteht dabei aus einem Schlüssel und dem assoziierten Wert.

Paar.java

```

package name.panitz.data.list;
```

[3]Das Programm `Eliza` aus einer der vorangegangenen Aufgaben hat genau so Abbildungen umgesetzt.

```
public class Paar<a,b>{
  public a fst;
  public b snd;
  public Paar(a f,b s){fst=f;snd=s;}
}
```

So ist eine einfache Umsetzung von Abbildungen durch Erweitern unserer Listenklassen zu bekommen:

Abbildung.java

```
package name.panitz.data.list;
class Abbildung<a,b> extends Li<Paar<a,b>>{
  public Abbildung(a schlüssel,b wert,Abbildung<a,b> xs){
    super(new Paar<a,b>(schlüssel,wert),xs);
  }

  public Abbildung(){}

  b lookup(a schluessel){
    b result = null;

    for (Paar<a,b> p:this) {
      if (p.fst.equals(schluessel)){
        return p.snd;
      }
    }
    return result;
  }
}
```

Die Methode `lookup` iteriert über die Liste, bis sie ein Listenelement gefunden hat, dessen Schlüssel dem gesuchten Schlüssel gleich kommt.

Dictionary.java

```
package name.panitz.data.list;
class Dictionary {
  static final Abbildung<String,String> dic =
     new Abbildung<String,String>("Menge","set"
    ,new Abbildung<String,String>("Abbildung","map"
    ,new Abbildung<String,String>("Liste","list"
    ,new Abbildung<String,String>("Iterator","iterator"
    ,new Abbildung<String,String>("Schnittstelle","interface"
    ,new Abbildung<String,String>("Klasse","class"
    ,new Abbildung<String,String>()))))));

  static String english(String deutsch){
    return dic.lookup(deutsch);
  }

  public static void main(String [] _){
```

```
    System.out.println(english("Schnittstelle"));
    System.out.println(english("Abbildung"));
  }
}
```

Standardklasse für Abbildungen

Java stellt eine Schnittstelle zur Verwirklichung von Abbildungen zur Verfügung, die Schnittstelle `java.util.Map`.

Die wichtigsten zwei Methoden dieser Schnittstelle sind:

- `void put(a key, b value)`: ein neues Schlüssel-/Wertpaar wird der Abbildung hinzugefügt.
- `Object get(a 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

```
import java.util.*;

class Dic{
  static Map<String,String> map = new HashMap<String,String>();

  static {
    map.put("Menge","set");
    map.put("Abbildung","map");
    map.put("Liste","list");
    map.put("Iterator","iterator");
    map.put("Schnittstelle","interface");
    map.put("Klasse","class");
  }

  static String english(String deutsch){
   return map.get(deutsch);
  }

  public static void main(String [] _){
    System.out.println(english("Schnittstelle"));
    System.out.println(english("Abbildung"));
  }

}
```

7.3.5 Weitere Sammlungsmethoden

In der Klasse `java.util.Collections` befinden sich statische Methoden zur Manipulation von Listen, wie z.B. zum Rotieren, Rückwärtsdrehen oder Sortieren.

7.3.6 Vergleiche

Ähnlich, wie wir für unsere Sortiermethode `sortBy` eine Klasse `Relation` als Parameter übergeben haben, in der es eine Methode gab, die prüfte, ob zwei Elemente in der kleiner-Relation stehen, gibt es in Java die Schnittstelle `java.util.Comparator`. In dieser Klasse gibt es die Methode `int compare(Object o1, Object o2)`, deren Ergebnis eine negative Zahl, 0 oder eine positive Zahl ist, je nachdem, ob das erste Argument kleiner, gleich oder größer als das zweite Argument ist.

Wir können einen Comparator für Stringobjekte schreiben, der zwei Strings vergleicht, indem er sie erst umdreht und erst dann die Standardvergleiche für Strings anwendet:

ReverseComparator.java

```
1  import java.util.Comparator;
2  class ReverseComparator implements Comparator<String> {
3
4    public int compare(String o1, String 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 sich reimende Wörter hintereinander befinden:

Rhyme.java

```
1  import java.util.List;
2  import java.util.ArrayList;
3  import java.util.Collections;
4
5  class Rhyme {
6    public static List<String> rhyme(List<String> xs){
7      Collections.sort(xs,new ReverseComparator());
8      return xs;
9    }
10
11   public static void main(String [] args){
```

```
12      List<String> xs = new ArrayList<String>();
13      xs.add("maus");
14      xs.add("rasieren");
15      xs.add("bauhaus");
16      xs.add("verlieren");
17      xs.add("regen");
18      xs.add("laus");
19      xs.add("segen");
20      xs.add("kapitulieren");
21      xs.add("monieren");
22      xs.add("existieren");
23      xs.add("kraus");
24      rhyme(xs);
25      System.out.println(xs);
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`.

7.4 Ein- und Ausgabe

Im Paket `java.io` befinden sich eine Reihe von Klassen, die Ein- und Ausgabe von Daten auf externen Datenquellen erlauben. In den häufigsten Fällen handelt es sich hierbei um Dateien.

Ein-/Ausgabeoperationen werden in Java auf sogenannten Datenströmen ausgeführt. Datenströme verbinden das Programm mit einer externen Datenquelle bzw. Senke. Auf diesen Strömen stehen Methoden zum Senden (Schreiben) von Daten an die Senke bzw. Empfangen (Lesen) von Daten aus der Quelle. Typischer Weise wird ein Datenstrom angelegt, geöffnet, dann werden darauf Daten gelesen und geschrieben und schließlich der Strom wieder geschlossen.

Ein Datenstrom charakterisiert sich dadurch, daß er streng sequentiell arbeitet. Daten werden also auf Strömen immer von vorne nach hinten gelesen und geschrieben.

Java unterscheidet zwei fundamentale unterschiedliche Arten von Datenströmen:

- **Zeichenenströme**: Die Grundeinheit der Datenkommunikation sind Buchstaben und andere Schriftzeichen. Hierbei kann es sich um beliebige Zeichen aus dem Unicode handeln, also Buchstaben so gut wie jeder bekannten Schrift, von lateinischer über kyrillische, griechische, arabische, chinesische bis hin zu exotischen Schriften wie der keltischen Keilschrift. Dabei ist entscheidend, in welcher Codierurung die Buchstaben in der Datenquelle vorliegen. Die Codierung wird in der Regel beim Konstruieren eines Datenstroms festgelegt. Geschieht dieses nicht, so wird die Standardcodierung des Systems, auf dem das Programm läuft, benutzt.

- **Byteströme (Oktettströme)**: Hierbei ist die Grundeinheit immer ein Byte, das als Zahl verstanden wird.

7.4.1 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`.

7.4.2 Byteströme

Entsprechend gibt es auch zwei Hauptklassen zum Lesen bzw. Schreiben von Byteströmen:

- `InputStream`
- `OutputStream`

Byteströme sind dafür gedacht, binäre Dateien zu lesen und zu schreiben, wie z.B. Bilddateien.

Auch hier stehen verschiedene Unterklassen für verschiedene Arten von Datenquellen zur Verfügung. Für Dateien sind dieses entsprechend die Klassen `FileInputStream` und `FileOutputStream`.

7.4.3 Ausnahmen

Ein-/Ausgabeoperationen sind fehleranfällig. Externe Datenquellen können fehlerhaft sein, nicht existieren oder sich unerwartet verhalten. Daher kann es bei fast allen Methoden auf Datenströmen zu Ausnahmen kommen, die alle von der Oberklasse `IOException` ableiten.

7.4.4 Schreiben und Lesen

Sowohl für Zeichenströme als auch für Byteströme stehen Methoden `int read()` bzw. `int write(int c)` zur Verfügung. Die Methode `read` liefert entweder einen positiven Wert oder den Wert `-1`, der anzeigt, daß keine weitere Daten im Datenstrom vorhanden sind.

7.4.5 Datenströme

Zum Schreiben und Lesen in Dateien haben die entsprechenden Stromklassen Konstruktoren, die einen Datenstrom für einen Dateinamen erzeugen. Mit der Konstruktion wird der Strom geöffnet. Java schließt den Strom automatisch wieder, wenn das Objekt nicht mehr benutzt wird; da dieser Zeitpunkt abhängig davon ist, wann zufällig die Speicherverwaltung ein Objekt aus dem Speicher löscht, ist es zu empfehlen, immer explizit die Methode `close` für solche Datenströme aufzurufen, die nicht mehr benutzt werden.

Textdateien lesen

Zum Lesen einer Textdatei läßt sich bequem die Klasse `FileReader` benutzen. Im folgenden Programm wird eine über die Kommandozeile spezifizierte Datei gelesen und jedes darin enthaltene Zeichen zusammen mit seiner Unicodenummer auf dem Bildschirm ausgegeben:

ShowFile.java

```
import java.io.*;
class ShowFile{
  public static void main(String [] args) throws Exception{
    Reader reader = new FileReader(args[0]);
    while (true){
      final int next = reader.read();
      if (next==-1) break;
      System.out.print((char)next);
      System.out.print(next);
    }
  }
}
```

Textdateien schreiben

Entsprechend gibt es auch eine Klasse, die zum Schreiben einer Textdatei bequem benutzt werden kann: `FileWriter`. So läßt sich ein Programm zum Kopieren einer Datei wie folgt schreiben:

Copy.java

```
import java.io.*;

public class Copy {
  public static void main(String[] args) throws IOException {
    FileReader in = new FileReader(args[0]);
    FileWriter out = new FileWriter(args[1]);
    int c;

    while ((c = in.read()) != -1){
      out.write(c);
    }
  }
}
```

7.4.6 Vom InputStream zum Reader, vom OutputStream zum Writer

`Reader` und `Writer` sind praktische Klassen zur Verarbeitung von Zeichenströmen. Primär sind aber auch Textdateien lediglich eine Folge von Bytes.

InputStreamReader und OutputStreamWriter

Mit den Klassen `InputStreamReader` und `OutputStreamWriter` lassen sich Objekte vom Typ `IntputStream` bzw. `OutputStream` zu `Reader`- bzw. `Writer`-Objekten machen.

Beispiel:
Statt die vorgefertigte Klasse `FileWriter` zum Schreiben einer Textdatei zu benutzen, erzeugt die folgende Version zum Kopieren von Dateien einen über einen `FileOutputStream` erzeugten `Writer` bzw. einen über einen `FileInputStream` erzeugten `Reader`:

Copy2.java

```
import java.io.*;

class Copy2 {
  static public void main(String [] args)
     throws Exception {
    Reader reader = new FileReader(args[0]);
    Writer writer
     = new OutputStreamWriter(new FileOutputStream(args[1]));

    int c;
    while ((c = reader.read()) != -1){
      writer.write(c);
    }
    writer.close();
  }
}
```

Zeichenkodierungen

Java-Strings sind Zeichenketten, die nicht auf eine Kultur mit einer bestimmten Schrift beschränkt , sondern in der Lage sind, alle im Unicode erfassten Zeichen darzustellen; seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines Strings kann potentiell eines dieser mehreren zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige, kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell Zahlen, die acht Bit im Speicher belegen, ein sogenanntes Byte. Ein Byte ist in der Lage, 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments) darzustellen. Für ein chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Codierung* eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem ß zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit zwei Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet.

Man sieht, daß es Codierungen geben muß, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüberhinaus auch eine Reihe Misch-

formen; manche Zeichen werden durch ein Byte, andere durch zwei oder sogar durch zwei Byte dargestellt.

Die Klasse `OutputStreamWriter` sieht einen Konstruktor vor, dem man zusätzlich zum `OutputStream`, in den geschrieben werden soll, als zweites Element auch die Codierung angeben kann, in der die Buchstaben abgespeichert werden sollen. Wenn diese Codierung nicht explizit angegeben wird, so benutzt Java die standardmäßig auf dem Betriebssystem benutzte Codierung.

Beispiel:
In dieser Version der Kopierung einer Textdatei wird für den `Writer` ein Objekt der Klasse `OutputStreamWriter` benutzt, in der als Zeichenkodierung utf-16 benutzt wird.

EncodedCopy.java

```
import java.nio.charset.Charset;
import java.io.*;

class EncodedCopy {
  static public void main(String [] args)
     throws Exception {
    Reader reader = new FileReader(args[0]);
    Writer writer = new OutputStreamWriter
                         (new FileOutputStream(args[1])
                         ,Charset.forName("UTF-16"));

    int c;
    while ((c = reader.read()) != -1){
      writer.write(c);
    }
    writer.close();
  }
}
```

Betrachtet man die Größe der geschriebenen Datei, so wird man feststellen, daß sie mehr als doppelt so groß ist wie die Ursprungsdatei.

```
sep@linux:~/fh/prog1/> java EncodedCopy EncodedCopy.java EncodedCopyUTF16.java
sep@linux:~/fh/prog1/> ls -l EncodedCopy.java
-rw-r--r--    1 sep      users         443 2004-01-07 19:12 EncodedCopy.java
sep@linux:~/fh/prog1/> ls -l EncodedCopyUTF16.java
-rw-r--r--    1 sep      users         888 2004-01-07 19:13 EncodedCopyUTF16.java
sep@linux:~/fh/prog1/>
```

Aufgabe 35 Schreiben Sie ein Programm `FileConvert` zum Konvertieren von Textdateien in eine andere Kodierung. Dem Programm sollen über die Kommandozeilenparameter der Name der Eingabedatei, der Name der Ausgabedatei und der Name der benutzten Codierung übergeben werden. Ein möglicher Aufruf wäre also:

```
linux:~/>java FileConvert test.txt konvertiertTest.txt utf-8
```

Lassen Sie eine deutsche Textdatei mit Umlauten in eine Datei mit der Codierung utf-8 konvertieren. Betrachten Sie die Ergebnisdatei. Was stellen Sie fest?

Gepufferte Ströme

Die bisher betrachteten Ströme arbeiten immer exakt zeichenweise, bzw. byteweise. Damit wird bei jedem `read` und bei jedem `write` direkt von der Quelle bzw. an die Senke ein Zeichen übertragen. Für Dateien heißt das, es wird über das Betriebssystem auf die Datei auf der Festplatte zugegriffen. Handelt es sich bei Quelle/Senke um eine teure und aufwändige Netzwerkverbindung, so ist für jedes einzelne Zeichen über diese Netzwerkverbindung zu kommunizieren. Da in der Regel nicht nur einzelne Zeichen über einen Strom übertragen werden sollen, ist es effizienter, wenn technisch gleich eine Menge von Zeichen übertragen wird. Um dieses zu bewerkstelligen, bietet java an, Ströme in gepufferte Ströme umzuwandeln.

Ein gepufferter Strom hat einen internen Speicher. Bei einer Datenübertragung wird für schreibende Ströme erst eine Anzahl von Zeichen in diesem Zwischenspeicher abgelegt, bis dieser seine Kapazität erreicht hat, um dann alle Zeichen aus dem Zwischenspeicher *en bloc* zu übertragen. Für lesende Ströme wird entsprechend für ein `read` gleich eine ganze Anzahl von Zeichen von der Datenquelle geholt und im Zwischenspeicher abgelegt. Weitere `read`-Operationen holen dann die Zeichen nicht mehr direkt aus der Datenquelle, sondern aus dem Zwischenspeicher, bis dieser komplett ausgelesen wurde und von der Datenquelle wieder zu füllen ist.

Die entsprechenden Klassen, die Ströme in gepufferte Ströme verpacken, heißen:
`BufferedInputStream, BufferedOutputStream` und
entsprechend `BufferedReader, BufferedWriter`.

Beispiel:
Jetzt ergänzen wir zur Effizienzsteigerung noch das Kopierprogramm, so daß der benutzte `Writer` gepuffert ist:

BufferedCopy.java

```
1  import java.io.*;
2  import java.nio.charset.Charset;
3
4  class BufferedCopy{
5    static public void main(String [] args)
6       throws Exception {
7
8      Reader reader = new FileReader(args[0]);
9      Writer writer = new BufferedWriter
10                        (new OutputStreamWriter
11                            (new FileOutputStream(args[1])
12                            ,Charset.forName("UTF-16")));
13
14     int c;
15     while ((c = reader.read()) != -1){
16       writer.write(c);
17     }
18     writer.close();
19   }
20 }
```

7.4.7 Ströme für Objekte

Bisher haben wir uns darauf beschränkt, Zeichenketten über Ströme zu lesen und zu schreiben. Java bietet darüberhinaus die Möglichkeit an, beliebige Objekte über Ströme zu schreiben und zu lesen. Hierzu können mit den Klassen `ObjectOutputStream` und `ObjectInputStream` beliebige `OutputStream`- bzw. `InputStream`-Objekte zu Strömen für Objekte gemacht werden.

In diesen Klassen stehen Methoden zum Lesen und Schreiben von Objekten zur Verfügung. Allerdings können über diese Ströme nur Objekte von Klassen geschickt werden, die die Schnittstelle `java.io.Serializable` implementieren. Die meisten Standardklassen implementieren diese Schnittstelle. `Serializable` enthält keine Methoden, es reicht also zum Implementieren aus, die Klausel `implements Serializable` für eine Klasse zu benutzen, damit Objekte der Klasse über Objektströme geschickt werden können.

Objektströme haben zusätzlich Methoden zum Lesen und Schreiben von primitiven Typen.

Beispiel:
Folgendes Testprogramm schreibt eine Zahl und ein Listenobjekt in eine Datei, um diese anschließend wieder aus der Datei auszulesen.

WriteReadObject.java

```
1  import java.io.*;
2  import java.util.List;
3  import java.util.ArrayList;
4
5  public class WriteReadObject {
6    public static void main(String [] args) throws Exception{
7      FileOutputStream fos = new FileOutputStream("t.tmp");
8      ObjectOutputStream oos = new ObjectOutputStream(fos);
9
10     List<String> xs = new ArrayList<String>();
11     xs.add("the");
12     xs.add("world");
13     xs.add("is");
14     xs.add("my");
15     xs.add("oyster");
16
17     oos.writeInt(12345);
18     oos.writeObject(xs);
19     oos.close();
20
21     FileInputStream fis = new FileInputStream("t.tmp");
22     ObjectInputStream ois = new ObjectInputStream(fis);
23
24     int i = ois.readInt();
25     List ys = (List) ois.readObject();
26
27     ois.close();
28
29     System.out.println(i);
30     System.out.println(ys);
```

```
31    }
32 }
```

Aufgabe 36 Erweitern Sie die Klasse `Li`, so daß Sie Ihre Listenobjekte in Dateien schreiben und wieder aus Dateien lesen können. Testen Sie ihre Implementierung.

7.4.8 Stromloses IO

Die Ein- und Ausgabe mit Stromklasse zwingt immer, eine Datenquelle vom Anfang an durchzulesen, bis man zu den Daten der Quelle kommt, die einen interessieren. Oft interessiert man sich aber nur für bestimmte Daten irgendwo in der Mitte einer Datei. Ein typisches Beispiel wäre ein Dateiformat eines Textdokumentes. Am Anfang der Datei ist eine Tabelle, die angibt, auf welcher Position in der Datei sich die einzelnen Seiten befinden. Ist man nur an einer bestimmten Seite des Dokuments interessiert, so kann man ihren Index in der Datei der Tabelle am Anfang entnehmen und dann direkt diese Seite lesen, ohne alle vorherigen Seiten betrachten zu müssen. Für eine solche Arbeitsweise stellt Java die Klasse `java.io.RandomAccessFile` zur Verfügung.

Aufgabe 37 Studieren Sie die Dokumentation von `java.io.RandomAccessFile` und schreiben Sie einige Testbeispiele zur Benutzung dieser Klasse.

Kapitel 8

Qualitätssicherung

8.1 Verträge

AssertFac.java

```
public class AssertFac{
  static public int fac(int x){
    assert x>=0;
    if (x==0) return 1;
    return x*fac(x-1);
  }

  static public void main(String [] args){
    for (String arg:args)
      System.out.println(fac(new Integer(arg)));
  }
}
```

```
sep@pc305-3:~/fh/java> java -cp student/classes/ -ea AssertFac 1 2 3 4 5 -1 17
1
2
6
24
120
Exception in thread "main" java.lang.AssertionError
        at AssertFac.fac(AssertFac.java:4)
        at AssertFac.main(AssertFac.java:11)
sep@pc305-3:~/fh/java>
```

Seiteneffektfrei, apt für pre und postcondition?

8.2 Unit Tests

8.3 Logging

JustLogMe.java

```
package name.panitz.log.example;
import java.util.logging.Logger;
import java.util.logging.Level;
public class JustLogMe{
  private static Logger logger = Logger.getLogger("name.panitz.log.example");

  static public void main(String [] args){
    logger.info("eine log info");
    logger.log(Level.WARNING, "ein kleines problem" );
    logger.log(Level.SEVERE, "ein großes problem");
    logger.fine("eine super feine info");
  }
}
```

8.4 Einführung in JavaDoc (Autor: Robert Klein)

8.4.1 Einleitung und Motivation

> There is nothing in the programming field more despicable than an undocumented program.
>
> *(Edward Yourdon)*

Tagebuch eines Informatikers

25. Juli 2007 Ah! Erster Tag auf der Arbeit. Die 2 Monate Urlaub nach dem Diplom waren super. Jetzt kann's richtig losgehn! Neues Projekt bei G. Was geowissenschaftliches. Alles voll mit Objektorientierung, Design Pattern, alles soll dokumentiert werden. Klingt saugut.

13. August 2007 Fast zwei Wochen nur geplant! Heute endlich der erste Code. SUper, erklärt sich alles selbst. Braucht man fast nix zu kommentiern! Ich hab' das nie gemocht. Schon damals beim K. in programmieren 2, der wollte das immer ganz genau. Alles Unsinn.

Zwei Jahre vergehen...

28. September 2009 mm... noch ein paar Monate, spätestens Januar, dann ist das Projekt endlich fertig! Hätte aber doch von Anfang an mehr dokumentieren sollen. Klassennamen wie IGPTNStorage sind doch nicht alzu selbsterklärend. GP für GeoPlan, die Firma, das war klar. I für Interface, mm... hab's dann doch rausgefunden, VTN war "'Tree node"'.

7. März 2010 Sch...!! Dachte der M., mein Kollege, wäre mit seiner Bibliothek fast fertig, als er gekündigt hat. Pustekuchen! Und dokumentiert hat der genausowenig wie ich. Grrrr!

> *1. April 2011* April, April! Hahahaha. Gestern die Kündigung bekommen. Nachdem der Rest des Teams schon vor Monaten weg ist. Alles GanzPlöd bei der GeoPlan GDV GmbH. Nächsten Monat wird hier alles dichtgemacht. Am Ende hat keiner mehr seinen Code verstanden. Nix war richtig dokumentiert. Fast vier Jahre für die Füße!

Leicht übertrieben könte es fast jedem Studenten nach dem Abschluß so ergehen. Vielleicht wird man auch nur beauftragt, ein Programm auf den aktuellen Stand einer Norm zu bringen, ein Programm, dass pro Quelldatei im Durchschnitt zwei Zeilen Kommentar hat. So etwas mag sehr lehrreich sein. Soll das Projekt in knapper Zeit fertig werden, dann kann es sich schnell als unmöglich erweisen.

Java hat drei Arten von Kommentaren; die einen sind die aus C bekannten mit /* und */ eingeschlossenen, die anderen die einzeiligen Kommentare aus C++ mit vorangestelltem //. Diese dritte sind die, die mit /** beginnen und mit */ enden. Diese JavaDoc-Kommentare sind mehrzeilig, wie Kommentare in C.

Oft hat die Firma einen "'Style Guide,"' der vorschreibt, wie man die ersten beiden Arten verwendet, etwa daß Kommentare nur mit // anfangen dürfen, und daß Quellcodeblöcke mit /* und */ auskommentiert werden können. Moderne Editoren und Entwicklungsumgebungen erlauben es auch, ganze Blöcke mit // auszukommentieren. Was man verwendet hängt vom Arbeitgeber oder den eigenen Vorlieben ab.

Das Thema dieses Kapitels beschränkt sich ab hier auf die dritte Art, die JavaDoc-Kommentare.

Diese Einleitung basiert auf der Referenz "'How to Write Doc Comments for the JavaDoc Tool"'[1] und der JavaDoc-Programmbeschreibung.[2]

8.4.2 Das Format eines JavaDoc-Kommentars

JavaDoc-Kommentare werden in HTML geschrieben. Sie stehen vor den Klassen, Attributen oder Methoden, die sie beschreiben. Ein JavaDoc-Kommentar besteht aus zwei Teilen, einer Beschreibung gefolgt von einem oder mehreren Tags.

```
1  /**
2   * Gibt den Wert "Wurzel aus -1" als Datentyp Komplex üzurck.
3   *
4   * @return Wurzel aus -1
5   */
6  public static Komplex i() {
7      return new Komplex(0, -1);
8  }
9
10 /**
11  * Addiert zwei komplexe Zahlen und gibt das Ergebnis üzurck
12  *
13  * @param a der erste zu addierende Operand
14  * @param b der zweite zu addierende Operand
```

[1] http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
[2] http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html

```
15  *
16  * @return das Additionsergebnis
17  */
18 public static Komplex add(Komplex a, Komplex b) {
19     return new Komplex(a.re + b.re, a.im + b.im);
20 }
21
22 /**
23  * Addiert eine Komplexe Zahl zum Komplexen Objekt.
24  *
25  * @param a die zu addierende komplexe Zahl
26  */
27 public void add(Komplex a) {
28     this.re += a.re;
29     this.im += a.im;
30 }
```

Wie im Beispiel beginnt ein Kommentar mit */**. Die führenden Sternchen in jeder Zeile sind in neueren Versionen von JavaDoc nicht mehr nötig. In der letzten Zeile wird der Kommentar mit */ abgeschlossen. Im Anhang A zeigt ein Screenshot, wie dieses Beispiel später im Browser aussieht.

8.4.3 Schreibstil

Der erste Satz eines JavaDoc-Kommentars kommt sowohl in den Index als auch in die Tabelle der Methodenzusammenfassungen. Er sollte eine kurze Zusammenfassung der Methode sein.

Hat eine Beschreibung mehrere Absätze, dann werden sie durch <*p*> getrennt.

Idealerweise sollte eine Beschreibung so gut sein, daß man ohne andere Hilfen den beschriebenen Code neu implementieren kann.

In einigen Fällen kann man es sich sparen, JavaDoc-Kommentare abzuschreiben. Wenn eine Methode einer Klasse eine Methode der Superklasse überschreibt, dann erzeugt JavaDoc einen Verweis überschreibt mit einem Link auf die überschriebene Methode. Das gleiche gilt, wenn eine Methode in einem Interface eine Methode eines Superinterfaces überschreibt. In einem dritten Fall, wenn eine Methode die Methode eines Interfaces implementiert, erzeugt JavaDoc einen Verweis "'specified by"' mit einem Link auf die implementierte Methode.

Die Beschreibung endet mit der ersten Zeile die ein @ enthält. Zwischen der Beschreibung und den @-Tags sollte eine Leerzeile sein. Jeder JavaDoc-Kommentar kann nur einen Beschreibungsblock enthalten. Nach den Tags kann die Beschreibung nicht fortgesetzt werden.

In <*code*>...</*code*> Tags eingeschlossen werden Java-Schlüsselwörter, Namen von Variablen, Klassen, Methoden, Interfaces, Feldern und Argumenten sowie Beispielcode.

Verwende keine Klammern für die allgemeine Form von Methoden und Konstruktoren. Betrachten wir einmal die beiden *add*-Methoden im obigen Beispiel. Es gibt die zwei Formen *add(Komplex, Komplex)* und *add(Komplex)*.

> Die *add(Komplex, Komplex)*-Methode ist statisch; sie addiert zwei übergebene komplexe Zahlen und gibt das Ergebnis zurück.

Wenn man jedoch beide Formen der Methode meint, dann läßt man die Klammern ganz weg. Würde man sie dennoch verwenden, so könnte der Leser annehmen, es gäbe noch eine dritte *add*-Methode, die keine Parameter hat.

Statt Sätzen kann man der Kürze wegen einfach Satzglieder verwenden. Insbesondere der Eingangssatz der Beschreibung und die @param-Tags sind dafür geeignet. Die Bechreibung sollte mit einem Verbausdruck anfangen: "'Addiert zwei komplexe Zahlen...'" statt "'Diese Methode addiert zwei komplexe Zahlen"'. Die Beschreibung von Klassen, Interfaces und Feldern kann das Subjekt weglassen: "'Eine Schaltflächenbeschriftung"' statt "'Dieses Feld ist eine Schaltflächenbeschriftung"'.

Vermeide Abkürzungen!

8.4.4 Tags

Die Tags jeden Kommentares müssen in einer bestimmten Reihenfolge aufgeführt werden. Sie sind in dieser Reihenfolge beschrieben.

@author Das *@author*-Tag ist optional. Sie können ein, kein oder mehrere *@author*-Tags verwenden. Werden mehrere *@author*-Tags verwendet, so sollten sie chronologische geordnet sein, der Erzeuger der Klasse zuerst.

> *@author* Zaphod Beeblebrox

@version Statt einer eigenen Versionsnummer ist es auch möglich, die Versionsnummern von Sourcecodeverwaltungssystemen wie SCCS zu verwenden.

> *@version* 1.02

@param Dem *@param*-Tag folgt zuerst der Name (nicht der Datentyp) des Parameters, gefolgt von einer Beschreibung des Parameters. Es hat sich eingebürgert, als erstes Hauptwort in der Beschreibung den Datentyp des Parameters zu verwenden, wobei davor noch ein Artikel ("'ein,"' "'der,"' &c) stehen kann. Bei dem primitiven Datentyp *int* wird der Datentyp üblicherweise weggelassen.

Parameternamen fangen mit Kleinbuchstaben an um sie von Klassennamen zu unterscheiden. Ist die Beschreibung im *@param*-Tag ein Satzglied, dann fängt sie mit einem Kleinbuchstaben an, mit einem Großbuchstaben, wenn es ein vollständiger Satz ist.

Der Parametername nach dem *@param*-Tag wird nicht in *<code>...</code>*-Tags eingeschlossen.

> *@param* Farbe die Farbe, die der Kreis haben soll.

@return Für Methoden die nichts zurückgeben und Konstruktoren sollte kein *@return*-Tag angegeben werden. Bei allen anderen Methoden empfiehlt es sich etwas zu schreiben, auch wenn die Beschreibung schon alles enthält. Dadurch läßt sich diese Information leichter finden, wenn man nur danach sucht. Wenn möglich sollte man hier auch Rückgabewerte für Spezialfälle aufführen.

> *@return* den gewürfelten Wert.

@throws* und *@exception Die beiden Tags sind synonym. Üblicherweise wird nur noch *@throws* verwendet. Für jede checked Exception (die in der *throws*-Klausel deklariert wird) sollte es ein *@throws*-Tag geben. Genauso sollte es für jede unchecked Exception ein *@throws*-Tag geben, wenn man davon ausgehen kann, daß der Aufrufer der Methode diese eventuell abfangen möchte.

> *@throws IOException* wenn eine Ein- oder Ausgabe-Exception vorkommt.

@see Hiermit wird ein Verweis auf ein anderes Element der Dokumentation hinzugefügt. Es gibt drei Arten von Verweisen. Die erste ist einfach ein String, der von Hochkommata begrenzt wird. Die zweite ist eine URL, wie sie in HTML verwendet wird: *Java Spezifikation*. Die dritte verweist auf andere JavaDoc-Elemente. Dabei kann man jedes gültige Programmelement in der Form *Paket.Klasse#Member* angeben. Zusätzlich kann noch eine Beschriftung angegeben werden. Gibt es keine Beschriftung, so wird der Verweis in der Form *Paket.Klasse.Member* angezeigt werden.

> *@see* semester2.Graphobjekt#flaeche Fläche des Graphobjektes

8.4.5 Kommandozeilenoptionen

-public Die Dokumentation wird nur für öffentliche Klassen und Member erstellt.

-protected Die Dokumentation wird für geschützte und öffentliche Klassen und Member erstellt. Dies ist die Voreinstellung

-package Die Dokumentation wird für Klassen und Member der Sichtbarkeiten Paket, geschützt und öffentlich erstellt.

-private Die Dokumentation wird für alle Klassen und Member erzeugt.

8.4.6 Epilog

Diese Einführung zeigt nur einige einfache Möglichkeiten von JavaDoc. Eine vollständige Dokumentation findet sich unter den beiden Internetlinks, die in der Einleitung genannt wurden. Dort gibt es auch weitere Ratschläge, wie man eine gute Dokumentation aufbauen kann.

Die sorgfältige Erstellung von Dokumentation wird oft als zeitraubend betrachtet. Langfristig ist sie jedoch fast immer hilfreich; in einer Menge von mehreren hundert Klassen und Dutzenden von Paketen ist es sonst unmöglich etwas zu finden. Gleichzeitig ist man während des Schreibens gezwungen noch einmal über die Klasse und Methode nachzudenken, was manchen Fehler aufdeckt, bevor er an den Kunden ausgeliefert wird.

8.4.7 ein vollständiges Beispiel

```
1 package de.xyzzy.java.tools;
2
3 /*
4  * Pair.java
5  *
```

```
 6  * (c) 2005 Robert Klein
 7  *
 8  * Redistribution and use in source and binary forms, with
 9  * with or without modification, are permitted under the
10  * BSD License.
11  */
12
13 /**
14  * implementiert eine Klasse, deren Instanzen in einem
15  * TreeSet nach der "nat&uuml;rlichen" Ordnung der
16  * Paarteile geordnet sind, wobei das erste Paarelement
17  * das h&ouml;herwertige ist.
18  *
19  * @author Robert Klein
20  * @version 1.0
21  *
22  * @param <A> der Typ des ersten Paarelements.
23  * @param <B> der Typ des zweiten Paarelements.
24  */
25 public class Pair<A extends Comparable<A>,
26                   B extends Comparable<B>>
27                   implements Comparable<Pair<A, B>> {
28     public A e1;
29     public B e2;
30
31     /**
32      * erzeugt eine Instanz mit den &uuml;bergebenen
33      * Parametern als Inhalt.
34      *
35      * @param e1 Erstes im Paar zu speicherndes Element
36      * @param e2 Zweites im Paar zu speicherndes Element
37      */
38     public Pair(A e1, B e2) {
39         this.e1 = e1;
40         this.e2 = e2;
41     }
42
43     /**
44      * vergleicht die aktuelle Instanz mit dem
45      * &uuml;bergebenen Paar.
46      *
47      * @param other Paar, mit dem verglichen werden soll.
48      * @return ein Wert <0, 0, oder >0, wenn die Instanz
49      *         kleiner, gleich bzw. ößgrer als das
50      * ü        bergebene Paar ist.
51      * @throws NullPointerException
52      *             wenn das Paar mit dem verglichen werden
53      *             soll nicht initialisiert ist.
54      * @see java.lang.Comparable#compareTo
55      */
```

```
56      public int compareTo(Pair<A, B> other) {
57          if (this == other) return 0;
58          if (null == other) throw new NullPointerException();
59          if (0 != e1.compareTo(other.e1))
60              return e1.compareTo(other.e1);
61          return e2.compareTo(other.e2);
62      }
63
64      /**
65       * zeigt an, ob ein anderes Objekt gleich diesem ist.
66       * Diese Methode implementiert die
67       * &Auml;quivalenzrelation:
68       * <ul>
69       * <li>Sie ist reflexiv: F&uuml;r jedes Paar
70       * <code>x&lt;A, B&gt;</code> erh&auml;t
71       * <code>x.equals(x)</code> den Wert <code>true</code>.
72       * <li>Sie ist symmetrisch: F&uuml;r alle Paare
73       * <code>x&lt;A, B&gt;</code>, <code>y&lt;A, B&gt;</code>
74       * gilt, da&szlig; <code>x.equals(y)</code> genau dann
75       * <code>true</code> zur&uuml;ckgibt, wenn
76       * <code>y.equals(x)</code>  <code>true</code>
77       * zur&uuml;ckgibt.
78       * <li>Sie ist transitiv: F&uuml;r drei Paare
79       * <code>x&lt;A, B&gt;</code>, <code>y&lt;A, B&gt;</code>
80       * und <code>z&lt;A, B&gt;</code>, bei denen
81       * <code>x.equals(y)</code> und <code>y.equals(z)</code>
82       * beide <code>true</code> zur&uuml;ckgeben, gibt
83       * <code>x.equals(z)</code> auch <code>true</code>
84       * zur&uuml;ck.
85       * <li>Sie ist konsistent: F&uuml;r zwei Paare
86       * <code>x&lt;A, B&gt;</code>
87       * und <code>y&lt;A, B&gt;</code> gibt
88       * <code>x.equals(y)</code> immer das gleiche Ergebnis
89       * zur&uuml;ck, solange sich in <code>x</code> und
90       * <code>y</code> keine Daten &auml;ndern, die f&uuml;r
91       * die <code>equals()</code>-Methode verwendet werden.
92       * </ul>
93       *
94       * @param other
95       *            das Objekt, mit dem verglichen werden soll.
96       * @return <code>true</code>, falls dieses Objekt gleich
97       *         dem Objekt im Argment ist, <code>false</code>
98       *         sonst.
99       * @see Object#equals(java.lang.Object)
100      */
101     public boolean equals(Pair<A, B> other) {
102         if (this == other) return true;
103         if (null == other) return false;
104         return (e1.equals(other.e1) && e2.equals(other.e2));
105     }
```

```
106
107     /**
108      * gibt einen String zur&uuml;ck, der das Objekt
109      * repr&auml;sentiert. Hier gibt er die beiden Elemente
110      * in  einem Klammerpaar, durch ein komma voneinander
111      * getrennt zur&uuml;ck.
112      *
113      * @return einen String, der das Objekt
114      *         repr&auml;sentiert.
115      *
116      * @see Object#toString()
117      */
118     public String toString() {
119         return "(" + e1.toString() + ",␣"
120                + e2.toString() + ")";
121     }
122 }
```

8.5 Qualitätsmaße

Kapitel 9

Algebraische Datentypen

Wir haben in diesem Skript bereits Datenstrukturen auf algebraische Weise spezifiziert und sie dann mit Hilfe von objektorientierter Programmierung realisiert: Listen und Bäume.

In der Programmierung gibt es oft die Situation, daß die Art der Daten eines Problemfeldes relativ fest ist und im Laufe des Projektes sich nicht in seiner Struktur groß ändert. Dann sind die Klassen, die die Datenstruktur realisieren in ihrer Struktur sehr stabil über den Entwicklungsprozess hinweg. Hingegen will man mit diesen Daten die unterschiedlichsten Berechnungen anstellen.

Ein typisches Beispiel für diese Situation findet sich im Compilerbau: In der Regel schreibt man einen Compiler für eine Programmiersprache, die fertig definiert ist und sich selten ändert. Der Parser erzeugt für ein Programm eine strukturierte Darstellung des Programms als Baum. Da die Sprachdefinition einer Programmiersprache relativ stabil ist, ist der Typ des Baumes auch sehr stabil und ändert sich selten. Hingegen wird im Laufe der Entwicklung und Weiterentwicklung eines Compilers auf diesem Baum immer wieder neue Operationen ausgeführt. Es werden neue Optimierungen eingebaut, für verschiedene Maschinen Code generiert, neue Checks für Warnungen eingebaut und und und. Wir haben also die Situation, daß die Datenstruktur sich selten ändert, aber permanent neue Algorithmen für diese Struktur geschrieben werden. Nun trägt das objektorientierte Programmiermodell diesem wenig Rechnung. Algorithmen finden sich hier verteilt auf die unterschiedlichen Klassen eines Datentyps in den verschieden überschriebenen Versionen der Methode, die zusammen einen Algorithmus realisieren. Will man einen neuen Algorithmus schreiben, so muß man in allen Klassen der die die Datenstruktur realisiert, die Methode überschreiben. Nicht nur der rein praktische Aspekt, daß man in vielen Dateien programmieren muß, um eine bestimmte Funktionalität zu programmieren ist lästig, sondern es fehlt auch ein gebündelter Überblick über den gesamten Algorithmus. Es gibt aber einen Trick, wie man algorithmuszentriert im Gegensatz zu datenzentriert in einer objektorientierten Sprache programmieren kann.

9.1 Das Besuchsmuster

Das Programmiermuster, das einen eine algorithmszentrierte Programmierung erlaubt, nennt sich da Besuchsmuster. Es spielt in virtuoser Weise mit den folgenden zwei Konzepten:

- **Überladung:** eine Methode existiert mehrfach mit gleichem Namen in einer Klasse. Die verschiedenen Versionen dieser Methode unterscheiden sich in ihren Parametern, indem sie unterschiedliche Parametertypen oder eine unterschiedliche Anzahl von Parametern haben.
- **Überschreibung:** eine Methode ist in verschiedenen Unterklassen einer Klasse überschrieben worden. Hier hat jede der überschreibenen Methoden dieselben Parameter. Man spricht im objektorientierten Umfeld auch von *Polymorphie*.

So ähnlich die beiden Konzepte von außen auch erscheinen, so haben sie einen fundamentalen Unterschied in der Art, wie entschieden wird, welche der unterschiedlichen Versionen einer Methode mit einem bestimmten Namen ausgeführt wird:

- **Überladung:** Der Compiler entscheidet statisch bei der Codegenerierung. Hierzu schaut er sich bei einem Methodenaufruf an, von welchem statischen Typ die Parameter des Aufrufs sind und wählt anhand dessen die Methode aus. Hierbei wird für alle Argumente der deklarierte Typ genommen, es werden also nicht die Argumentojekte gefragt, ob sie eventuell von einer Unterklasse ihres deklarierten Typs sind:

  Beispiel:
  Im folgenden kleinen Programm ist eine Funktion `f` überladen. In der Hauptmethode wird die Version für den Parametertyp `Object` ausgeführt, da das lokale Feld `s` als von Typ `Object` deklariert wurde. Es ist dabei unerheblich, daß das Objekt `s` aber eigentlich ein Stringobjekt ist.

  Overloaded.java
  ```
  1 class Overloaded{
  2   void f(String s){System.out.println("Stringversion");}
  3   void f(Object s){System.out.println("Objectversion");}
  4   public static void main(String [] _){
  5     Object s = "das ist ein String";
  6     new Overloaded().f(s);
  7   }
  8 }
  ```

- **Überschreibung**: Hier wird erst während der Ausführung des Programms entschieden, welche Version der Methode aufgerufen wird. Dieses ist das Prinzip dert späten Bindung. Während der Ausführungszeit wird das Objekt gebeten seine Version der überschriebenen Methode auszuführen.

  Beispiel:
  In diesem Beispiel benutzen wir die Eigenschaft, daß die Methode `toString` aus der Klasse `Object` in der Klasse `String` überschrieben wurde. Obwohl das lokale Feld `s` als von Typ `Object` deklariert ist, wird doch die Methode `toString` der Klasse `String` ausgeführt.

  Overridden.java
  ```
  1 class Overridden{
  2   public static void main(String [] _){
  3     Object s = "das ist ein String";
  4     System.out.println(s.toString());
  5   }
  6 }
  ```

9.1.1 Klassen zur Baumdarstellung eines Programms

Wir werden das Besuchsmuster anhand eines Beispiels aus dem Compilerbau erklären. Betrachten wir dazu das folgende kleine Programm einer winzigen kleinen ausgedachten Programmiersprache:

```
result=1;
x=5;
loop (x){
  result=result*x;
  x=x-1
}
```

Node.java

```
interface Node{
    void welcome(Visitor x);
}
```

Num.java

```
class Num implements Node {
  int value;
  Num(int n){value=n;}
  public void welcome(Visitor x){x.visit(this);}
}
```

Var.java

```
class Var implements Node {
  String name;
  Var(String n){name=n;}
  public void welcome(Visitor x){x.visit(this);}
}
```

Seq.java

```
class Seq implements Node {
  Node leftChild;
  Node rightChild;
  Seq(Node lC,Node rC){leftChild=lC;rightChild=rC;}
  public void welcome(Visitor x){x.visit(this);}
}
```

Assign.java

```
class Assign implements Node {
  Var leftChild;
  Node rightChild;
  Assign(Var lC,Node rC){leftChild=lC;rightChild=rC;}
  public void welcome(Visitor x){x.visit(this);}
}
```

Sub.java

```
class Sub implements Node {
  Node leftChild;
  Node rightChild;
  Sub(Node lC,Node rC){leftChild=lC;rightChild=rC;}
  public void welcome(Visitor x){x.visit(this);}
}
```

Mult.java

```
class Mult implements Node {
  Node leftChild;
  Node rightChild;
  Mult(Node lC,Node rC){leftChild=lC;rightChild=rC;}
  public void welcome(Visitor x){x.visit(this);}
}
```

Loop.java

```
class Loop implements Node {
  Node leftChild;
  Node rightChild;
  Loop(Node lC,Node rC){leftChild=lC;rightChild=rC;}
  public void welcome(Visitor x){x.visit(this);}
}
```

GetSimpleProg.java

```
class GetSimpleProg{
  public static Node getSimpleProg(){
    Var result=new Var("result");
    Var x=new Var("x");
    Node n1=new Num(1);
    Node n5=new Num(5);
    Node as1 = new Assign(result,n1);
    Node as2 = new Assign(x,n5);
    Node mult = new Mult(result,x);
    Node sub = new Sub(x,n1);
    Node as3 = new Assign(result,mult);
    Node as4 = new Assign(x,sub);
    Node seq3 = new Seq(as3,as4);
    Node loop = new Loop(x,seq3);
    Node seq2 = new Seq(as2,loop);
    Node seq1 = new Seq(as1,seq2);
    return seq1;
  }
}
```

Visitor.java

```
interface Visitor {
  void visit(Var x);
```

```
  void visit(Num x);
  void visit(Assign x);
  void visit(Loop x);
  void visit(Mult x);
  void visit(Sub x);
  void visit(Seq x);
}
```

Besucher zum Ausgeben

.java

```
class PrettyPrinter implements Visitor{
  StringBuffer result= new StringBuffer();
  int indent=0;
  void newline(){
    result.append("\n");
    for (int i=indent;i>0;i--)result.append(' ');
  }
  void addIndent(){i=i+2;}
  void removeIndent(){i=i-2;}

  public void visit(Var x){result.append(x.name);}
  public void visit(Num x){result.append(""+x.value);}
  public void visit(Assign x){
    x.leftChild.welcome(this);
    result.append(" = ");
    x.rightChild.welcome(this);
  }
  public void visit(Loop x){
    result.append("loop (");
    x.leftChild.welcome(this);
    resultappend("){");
    addIndent();
    newline();
    x.rightChild.welcome(this);
    removeIndent();
    newline();
    result.append("}");
  }
  public void visit(Mult x){
    x.leftChild.welcome(this);
    result.append(" * ");
    x.rightChild.welcome(this);
  }
  public void visit(Sub x){
    x.leftChild.welcome(this);
    result.append(" - ");
    x.rightChild.welcome(this);
  }
```

```
  public void visit(Seq x){
    x.leftChild.welcome(this);
    result.append(";")
    newline();
    x.rightChild.welcome(this);
  }
}
```

Besucher zum Ausführen eines Programms

Calculate.java

```
import java.util.*;
class Calculate implements Visitor{
  Map<String,Integer> env=new HashMap<String,Integer>();
  private int result;
  public void visit(Var x){result=env.get(x.name);}
  public void visit(Num x){result=x.value;}
  public void visit(Assign x){
    x.rightChild.welcome(this);
    env.put(x.leftChild.name,result);
  }
  public void visit(Loop x){
    x.leftChild.welcome(this);
    while (result!=0){
      x.rightChild.welcome(this);
      x.leftChild.welcome(this);
    }
  }
  public void visit(Mult x){
    x.leftChild.welcome(this);
    int op1 = result;
    x.rightChild.welcome(this);
    int op2 = result;
    result= op1*op2;
  }
  public void visit(Sub x){
    x.leftChild.welcome(this);
    int op1 = result;
    x.rightChild.welcome(this);
    int op2 = result;
    result= op1-op2;
  }
  public void visit(Seq x){
    x.leftChild.welcome(this);
    x.rightChild.welcome(this);
  }
}
```

9.2 Putzen der Kochplatte

Dieses Kapitel ist ein als Papier konzipiert und liegt daher in englischer Sprache vor. Es geht weiterhin um hierarchische Daten. Es werden weitere Programmiermuster vorgestellt, die es erlauben noch bequemer durche eine Datenstruktur durchzulaufen, um mehrere unterschiedliche Aufgaben zu bewerkstelligen:

- Über die komplette Datenstruktur zu gehen um etwas zu berechnen.
- Bestimmte Knoten der Datenstruktur zu verändern.
- Bestimmte Anfrage an die Datenstruktur zu stellen, um Information aus Ihr zu bekommen.

Und nun der englischsprachige Teil:

9.2.1 Itroduction

Genericity is the key to software reuse. Generic code solves general problems and can be specialized to specific problems. A general problem is the handling of hierarchical data. Almost all interesting data is hierarchical data. XML[T. 04], the universal data exchange format pays tribute to this fact. XML documents represent a hierarchical structure. In object oriented languages tree nodes are represented by different classes. Child nodes are represented as fields of these classes. Number, type and name of fields representing child nodes are arbitrary.

Quite often the classes representing tree nodes are generated by some tool, as e.g. by the parser generator tool javacc or for some XML DTD.

A common programming task for tree like structures is to traverse through the tree. Generally two different task may be archieved during traversal:

- certain tree nodes may be modified,
- or certain information may be collected from the tree.

In object oriented languages there are mainly two ways how tree traversal may be put into realization:

- in a pure object oriented manner: classes representing tree nodes contain specialized methods for different tasks.
- in a functional way by means of the visitor pattern[GHJV95]: tree classes implement an interface `Visitable`. Algorithms can be expressed by terms of visitor classes. A visitor class overloads a function for every kind of tree node.

Both ways are not generic. Both require that the tree traversal code is implemented for every node type. Even worse, in both cases the tree traversal code mixes with the actual application logic. Thus the tree traversal code gets duplicated for every algorithm working on the tree. The methods in the tree node classes or the overloaded methods `visits` in the visitor need to code, how to access the child nodes.

Furthermore, both techniques are not very flexible as in respect to program evolution. If in the object oriented scenario a new class is added, then this class needs to implement all methods,

which for some reason traverse over the tree. A visitor class, on the other hand is only applicable to a certain type of tree node. It is specialized for some type. If a new node class is added to the tree type, then every visitor needs to implement a new overloaded method variant for the new class.

A generic way for tree manipulation will clearly seperate the tree traversal code and the actual application logic. Furthermore it will be very flexible in terms of programm evolution: new classes arriving on the scene will not force any changes in the existing code.

A simple pattern for tree traversal has been proposed for the programming language Haskell[PJ03]. The *scrap your boilerplate code* pattern[LP03][LP04] factors out the tree traversal code and makes it reusable for different algorithms. We will apply this pattern to the programming language Java.

The paper is structured as follows: in section 9.2.2 the problem to be solved is presented by way of a concrete example taken from [LP03]. Section 9.2.3 gives some naïv non generic solutions to the problem. Section 9.2.4 implements a generic tree transformer and section 9.2.5 implements a generic query method for trees. Both implementations are refined in section 9.2.6. The Java code within this paper is complete and can be downloaded from the web (`http://panitz.name/paper/boilerplate/index.html`) .

9.2.2 The Problem

We use the same example as presented in [LP03]. It is a simple company structure, which can be expressed by the following XML document type definition.

company.dtd

```
<!DOCTYPE Company SYSTEM "company.dtd" [
  <!ELEMENT Company (Dept)*>
  <!ELEMENT Dept (Name,Employee,(PU|DU)*)>
  <!ELEMENT PU (Employee)>
  <!ELEMENT DU (Dept)>
  <!ELEMENT Employee (Person,Salary)>
  <!ELEMENT Person (Name,Address)>
  <!ELEMENT Salary (#PCDATA)>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Address (#PCDATA)>
]>
```

This definition gives directely rise to a Java implementation. For every element type defined in the DTD we can provide a Java class. The child elements are represented by way of fields in these classes. For the alternative operator `|` of several child elements as e.g. expressed in `(PU|DU)` a common superinterface for the classes `PU` and `DU` can be defined. For the repetitive occurrence of child elements as e.g. `(Dept)*` the standard Java interface `List` is used.

Java classes for the DTD above are given in appendix 9.2.8. Usually some generator tool can be used to generate classes from a DTD or from a XML schema definition.

Throughout this paper we will work with an example instance of `Company`. It is the same example instance as used in [LP03].[1]

MyComp.java

```
package name.panitz.boilerplate;
import java.util.*;
```

[1]Not quite it is extended with a subdepartment, otherwise one of the algorithms in this (as well as in the original paper) does not have any effect.

```
class MyComp{
  static Company getCompany(){
   Company company= new Company();
   Employee blair= new Employee(new Person("Blair","London"),new Salary(100000.0));
   Employee nn=new Employee(new Person("NN","Mars"),new Salary(90000.0));
   Employee marlow=new Employee(new Person("Marlow","Cambridge"),new Salary(2000.0));
   Employee joost = new Employee(new Person("Joost","Amsterdam"),new Salary(1000.0));
   Employee ralf = new Employee(new Person("Ralf","Amsterdam"),new Salary(8000.0));
   List<SubUnit> ralfSu = new ArrayList<SubUnit>();
   ralfSu.add(new PU(joost));ralfSu.add(new PU(marlow));
   company.add(new Dept("Research",ralf,ralfSu));
   company.add(new Dept("Strategy",blair,new ArrayList<SubUnit>()));
   ralfSu.add(new DU(new Dept("Research2",nn,new ArrayList<SubUnit>())));
   return company;
  }
}
```

Tasks to be solved for a structure like the company above are:

- increase the salary of each employee.
- liquidate a certain sub department.
- sum up all salary numbers for an overall salary bill of the company.
- represent the company by a `String`.
- compare two companies for equality.
- represent the company in an XML document.

All these task need to travers the company. This will make it necessary to rewrite the traversal code for each of these tasks. We want traverse tree structures in a generic way, i.e. write the traversal code once for all.

The domain specific language XSLT[Wor99], the for transforming XML documents gives us some hints on how this can be achieved in Java. In XSLT rules for nodes with a certain condition can be expressed. Elements nodes, which do not match any of these conditions simply traverse to the child nodes. XSLT allows to express default templates of how to traverse for elements, which do not match any specific rule. The first of the above tasks can easily be expressed in XSLT by:

IncreaseSalary.xsl

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="Salary">
  <xsl:variable name="arg1"><xsl:apply-templates select="text()"/></xsl:variable>
  <Salary><xsl:value-of select="$arg1 * 1.17" /></Salary>
</xsl:template>
<xsl:template match="*">
  <xsl:copy select="*" ><xsl:apply-templates/></xsl:copy >
</xsl:template>
</xsl:stylesheet>
```

Note that this XSLT program will work on any XML document. It is not necessary that the document is structured according to some schema or DTD. It will automatically traverse the complete document and increase the numbers found within every `Salary` node. XSLT relies on the fact that XML documents are trees. The `<xsl:apply-templates/>` apllies to the direct children of the current node. We would like to get an comparatively simple solution for this task in Java.

9.2.3 Naïve hand coded instances

In the previous section it turned out that the notion of child nodes is crucial for any tree traversal. In order to result in some generic tree traversal code, we will need some uniform method of how to retrieve the direct child nodes of some node. For this we can instrumentalize Java's standard interface `Iterable`. We will use `Iterable` such that it will not iterate over all descendents of a tree node, but only its direct children. In fact this is what we did for the classes representing the `Company` DTD as can be found in appendix 9.2.8. Since the type of the child nodes is arbitrary we implemented the most general instantiation of the interface: `Iterable<Object>`.

Show

With iterator over the direct child nodes, the boilerplate code allready implodes to a minimal amount. As an example we give an implementation of a generic `show` method. Thanks to Java's new *for-each* loop the tree traversal boilerplate code is just one line.

Show.java

```
package name.panitz.boilerplate;
class Show{
  static String show(Object o){
    StringBuffer result = new StringBuffer();
    if (o instanceof Iterable) {
      result.append("new "+o.getClass().getSimpleName()+"(");
      boolean first = true;
      for (Object x:((Iterable)o)){//the tree traversal code
        if (first) first=false;else result.append(",");
        result.append(show(x));
      }
      result.append(")");
    }else result.append(o);
    return result.toString();
  }
}
```

This method calculates a `String` representation for any object. If the object is iterable, we will traverse its child nodes.

TestShow.java

```
package name.panitz.boilerplate;
class TestShow{
  public static void main(String [] args){
    System.out.println(Show.show(MyComp.getCompany()));}}
```

Eq

Eq.java

```
package name.panitz.boilerplate;
import java.util.Iterator;
class Eq{
  static boolean eq(Object x,Object y){
    if (x instanceof Iterable && y instanceof Iterable){
      Iterator ity = ((Iterable)y).iterator();
      for (Object z:(Iterable)x){if (!ity.hasNext())    return false;
                                 if (!eq(z,ity.next())) return false;}
    }else return x.equals(y);
    return true;
```

```
  }
}
```

TestEq.java

```
package name.panitz.boilerplate;
class TestEq{
  public static void main(String [] args){
    System.out.println(Eq.eq(MyComp.getCompany(),MyComp.getCompany()));}}
```

9.2.4 Generic Transformers

In this section, we will generalize the traversal code. The actual application logic will be encapsulated in a method `transform`. We can define an interface for this method:

Transformer.java

```
package name.panitz.boilerplate;
interface Transformer{void transform(Object o);}
```

The generic tree traversal code is divided onto two methods. The first method is a generic `map` function. It iterates over the direct child nodes of the argument node and applies the transform method to these:

Transform.java

```
package name.panitz.boilerplate;
public abstract class Transform  implements Transformer{
  public  void gmap(Object x){
    if (x instanceof Iterable) for (Object a:((Iterable)x)) transform(a);
  }
```

For the complete traversal of the tree, the method `everywhere` is provided. it needs an inner `Transform` object, which will be cached:

Transform.java

```
  Transform everywhereT = null;
  public void everywhere(Object x){
```

The inner `everywhere` transformer will recursively call the method `everywhere` of this transformer:

Transform.java

```
    final Transform dies = this;
    if (everywhereT==null)
      everywhereT
       = new Transform(){public void transform(Object o){dies.everywhere(o);}};
```

Now we can use the generic `map` function to recurse everywhere into the child nodes. Afterwards we can apply the actual transform method to the root node:

Transform.java

```
    everywhereT.gmap(x);
    transform(x);
  }
}
```

The technique of providing a general `map` function for the direct children of an node is not very novel, e.g. [CF92] a tree like data definition induces the compiler to generate a `map` function.

Example: increase of all salary

Now we can implement a very simple solution to the first task of increasing the salary in every salary node of the tree. We can achieve this by extending the abstract class `Transform` where we implement the method `transform` according to our needs.

Increase.java

```
package name.panitz.boilerplate;
import java.util.*;
public class Increase extends Transform{
 Double k; Increase(Double k){this.k=k;}
 public void transform(Object o){
  if (o instanceof Salary){Salary s = (Salary)o;s.amount=s.amount*(1+k);}
 }
}
```

This looks even simpler than the XSLT solution. We do not need to bother about the tree traversal code. Applying this transformer to out company is equally as easy and straightforward.[2]

TestIncrease.java

```
package name.panitz.boilerplate;
class TestIncrease{
  public static void main(String [] args){
    Company company = MyComp.getCompany();
    for (int i=0;i<100000;i++)
      new Increase(.000017).everywhere(company);
    System.out.println(Show.show(company));}}
```

Example: flatten company structure

The second of the task stated in the beginning of this paper, was to liquidate certain subdepartments and integrate its employees into the next upper department. As we see, this is allready a rather complex application task. Fortunatly we no longer are concerned with the tree traversal code and can concentrate on the actual task at hand:

Flatten.java

```
package name.panitz.boilerplate;
import java.util.*;
class Flatten extends Transform{
  String deptName;
  Flatten(String deptName){this.deptName=deptName;}
  public void transform(Object o){
   if (o instanceof Dept){
     List<SubUnit> newSus=new ArrayList<SubUnit>();
     Dept d = (Dept)o;
     for (SubUnit su:d.subUnits){
       if (su instanceof DU){
         Dept du= ((DU)su).dept;
         if (du.name.equals(deptName)){
           newSus.add(new PU(du.manager));newSus.addAll(du.subUnits);
         }else newSus.add(su);
       }else newSus.add(su);
     }
     d.subUnits=newSus;
   }
  }
}
```

[2] We apply this transformer 100000 times. This is done for some rough performance measure later in the paper.

This new transformer can be applied to our company in the same way as the `Increase` transformer.

TestFlatten.java

```
package name.panitz.boilerplate;
class TestFlatten{
  public static void main(String [] args){
    Company company = MyComp.getCompany();
    new Flatten("Research2").everywhere(company);
    System.out.println(Show.show(company));}}
```

Note, that our transformers work on any object. We do not need to know the exact tree structure or all classes involved in a company structure. Our algorithms still work, even if new classes representing new types of subdepartments are added to the company structure.

Using Reflection

In the solution above, we did not get completely rid of boilerplate code. We needed to specify the concrete class, for which the method transform is specified. This amounts in the `instanceof` construct. It is well known, how to get rid of such conditions by way of Java's reflection mechanism, e.g. in [PJ98] generic tree traversal is achieved by way of using reflection. Following [PJ98] we can implement a generic default instance of the method `transform`.

TransformReflect.java

```
package name.panitz.boilerplate;
class TransformReflect  extends Transform{
  public void transform(Object o){
    try{getClass().getMethod("transform",o.getClass()).invoke(this,o);
    }catch(IllegalAccessException  _){}
     catch(IllegalArgumentException _){}
     catch(java.lang.reflect.InvocationTargetException _){}
     catch(NoSuchMethodException _){}
  }
}
```

This method checks the type of the argument and looks up via reflection, if a specialized version of `transform` for this class is available. Thus the transformer to increase the salary can simply be written as:

IncreaseReflect.java

```
package name.panitz.boilerplate;
import java.util.*;
class IncreaseReflect extends TransformReflect{
 Double k; IncreaseReflect(Double k){this.k=k;}
 public void transform(Salary s){s.amount=s.amount*(1+k);}
}
```

This implementation is completely free of any boilerplate code. However, reflection is slow, very slow. As a rough measure compare execution times of the non-reflective with the reflective solution:

TestIncreaseReflect.java

```
package name.panitz.boilerplate;
class TestIncreaseReflect{
  public static void main(String [] args){
    Company company = MyComp.getCompany();
    for (int i=0;i<100000;i++)
      new IncreaseReflect(.000017).everywhere(company);
    System.out.println(Show.show(company));}}
```

Execution of this version compared to the original version gives the following result:

```
sep@pc216-5:~/boilerplate> time java -cp classes/ name.panitz.boilerplate.TestIncrease
new Company(new Dept(Research,new Employee(new Person(Ralf,Amsterdam),new Salary
(43790.9463569525)),new ArrayList(new PU(new Employee(new Person(Joost,Amsterdam)
,new Salary(5473.868294619063))),new PU(new Employee(new Person(Marlow,Cambridge)
,new Salary(10947.736589238126))),new DU(new Dept(Research2,new Employee(new Pers
on(NN,Mars),new Salary(492648.1465157075)),new ArrayList())))),new Dept(Strategy,
new Employee(new Person(Blair,London),new Salary(547386.8294619016)),new ArrayLis
t()))

real    0m0.912s
user    0m0.849s
sys     0m0.013s
sep@pc216-5:~/boilerplate> time java -cp classes/ name.panitz.boilerplate.TestIncreaseReflect
new Company(new Dept(Research,new Employee(new Person(Ralf,Amsterdam),new Salary
(43790.9463569525)),new ArrayList(new PU(new Employee(new Person(Joost,Amsterdam)
,new Salary(5473.868294619063))),new PU(new Employee(new Person(Marlow,Cambridge)
,new Salary(10947.736589238126))),new DU(new Dept(Research2,new Employee(new Pers
on(NN,Mars),new Salary(492648.1465157075)),new ArrayList())))),new Dept(Strategy,
new Employee(new Person(Blair,London),new Salary(547386.8294619016)),new ArrayLis
t()))

real    0m53.016s
user    0m52.010s
sys     0m0.238s
```

9.2.5 Generic Queries

The transformers in the last section destructively updated the tree. In this section we will apply the same technique to retrieve some data from the tree. We want to state queries over the tree data and leave the tree object unchanged. What we have in mind is a simple function taking a tree and giving some query result. Since functions are not first class citizens in Java, we express a funtion by way of an interface:

Function1.java

```
package name.panitz.boilerplate;
interface Function1<argType,resultType>{resultType eval(argType x);}
```

We provide a second interface for binary functions.[3]

Function2.java

```
package name.panitz.boilerplate;
interface Function2<a,b,c>{c eval(a x,b y);}
```

We can now apply the same technique for queries as for transformers. We define the generic query class `Query`. It is gerneric over the result of the query. With Java's generic types this can be elegantly expressed:

Query.java

```
package name.panitz.boilerplate;
import java.util.*;
public abstract class Query<b> implements Function1<Object,b>,Function2<b,b,b>{
```

Traversal is again done in two steps. First we define a generic `map` method. This method applies the query to the child nodes and results in the list of results:

[3] We resisted the temptation of expressing these via currying as `Function1<a,Function1<b,c`■.

Query.java

```
  public List<b> gmap(Object x){
    List<b> result= new ArrayList<b>();
    if (x instanceof Iterable) for (Object a:((Iterable)x)) result.add(eval(a));
    return result;
  }
```

Analogously to the method `everywhere` in transformer case we define a method `everything`. It applies the same trick. It defines an inner query which traverses over the child nodes. Since we get a list of results for the child nodes, we need a function to add these up to an overall result. That is the reason why `Query` implements `Function2<b,b,b>`. This function is used to sum up the partial result to an overall result.

Query.java

```
  public b everything(Object x){
    b result = eval(x);
    final Query<b> dies = this;
    Query<b> everythingQ = null;
    if (everythingQ==null)
      everythingQ = new Query<b>(){
          public b eval(Object o){return dies.everything(o);}
          public b eval(b x,b y) {return dies.eval(x,y);}
      };
    for (b y:everythingQ.gmap(x)) result=eval(result,y);
    return result;
  }
}
```

A concrete query can now be implemented by extension of class `Query`. Two methods need to be implemented: the method `eval`, which represents the actual query method and the binary method `eval`, which defines how to combine partial results. In the example of a salary bill this might look like:

SalaryBill.java

```
package name.panitz.boilerplate;
class SalaryBill extends Query<Double>{
  public Double eval(Object x){
    if (x instanceof Salary) return ((Salary)x).amount;
    return 0.0;
  }
```

Note that the method needs an neutral default result.

The second method to be implemented is used to combine partial results. In our example the simple addition function will do.

SalaryBill.java

```
   public Double eval(Double x,Double y){return x+y;}
}
```

Thus we implemented the task of summing up all salaries, without reimplementation of the tree traversal code.

TestSalaryBill.java

```
package name.panitz.boilerplate;
class TestSalaryBill{
  public static void main(String [] args){
    System.out.println(new SalaryBill().everything(MyComp.getCompany()));
  }
}
```

Show revisited

Show2.java

```
package name.panitz.boilerplate;
class Show2 extends Query<String>{
  public String eval(Object x){
    return x instanceof Iterable?("new "+x.getClass().getSimpleName()):x.toString();
  }
  public String eval(String x,String y){return x+"("+y+")";}
  public static void main(String [] args){
    System.out.println(new Show2().everything(MyComp.getCompany()));}}
```

Back to XML

We started this paper by giving an example as XML document type definition. Eventually we turn back to XML. The class `Query` enables us to generically write a query object, which builds an XML tree for some tree structure. We need some document builder, which we instantiate once via a factory method:

ToXML.java

```
package name.panitz.boilerplate;
import javax.xml.parsers.*;
import org.w3c.dom.*;
class ToXML extends Query<Node>{
  static Document doc;
  static {
   try{doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
   }catch (Exception e) {throw new RuntimeException(e);}
  }
```

The two eval methods are very simple. We create elements with the class name of the object. Non iterable types, especially `String` nodes result in XML text nodes:

ToXML.java

```
  public Node eval(Object o){
    if (o instanceof Iterable)
      return doc.createElement(o.getClass().getSimpleName());
    return doc.createTextNode(o.toString());
  }
```

The second method `eval` for combining partial results, inserts the second argument as child to the first argument.

ToXML.java

```
  public Node eval(Node x,Node y){x.appendChild(y);return x;}
}
```

This simple query object generically transforms any hierarchical structure into an XML document. For a simple test we generate an XML document for the company.

TestToXML.java

```
package name.panitz.boilerplate;
import org.w3c.dom.Node;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;

class TestToXML{
```

```
  public static void main(String [] args){
    System.out.println(serialize(new ToXML().everything(MyComp.getCompany())));}
```

We use the following simple method for transforming an XML node to a `String`:

TestToXML.java

```
  static public String serialize(Node doc){
    try{
      StringWriter writer = new StringWriter();
      javax.xml.transform.Transformer t
        = TransformerFactory.newInstance().newTransformer();
      t.setOutputProperty(OutputKeys.INDENT,"yes");
      t.transform(new DOMSource(doc),new StreamResult(writer));
      return writer.getBuffer().toString();
    }catch (TransformerException _){return "";}
  }
}
```

9.2.6 Controling the traversal

A close look at the result of the last example reveals that it does not build an XML document according to the DTD given in section 9.2.2. This is not a defect in our implementation of `ToXML` but due to the fact that the Java classes for the company do not exactly match this DTD. No classes for person or department names are given. These are represented directly as a `String`.

We need specialized cases for certain nodes. Furthermore we need some way to express a *cut*, i.e. some means to express, that for this node the automatic tree traversal shall stop. Therefore we write a refinement of the class `Query`. A third abstract method is added, the method `condition`. It returns true for some arbitrary node, if the tree traversal is supposed to descent into the child nodes:

QuerySome.java

```
package name.panitz.boilerplate;
import java.util.*;
abstract class QuerySome<a>
        implements Function1<Object,a>,Function2<a,a,a>{
  public abstract boolean condition(Object o);
```

Now the method `gmap` only descents into the child nodes, if the condition for the current node is true:

QuerySome.java

```
  public List<a> gmap(Object x){
    List<a> result= new ArrayList<a>();
    if (condition(x) && x instanceof Iterable)
      for (Object a:((Iterable)x)) result.add(eval(a));
    return result;
  }
```

The rest of this class is implemented exactly like in the class `Query`:

QuerySome.java

```
  public a everything(Object x){
    a result = eval(x);
    final QuerySome<a> dies = this;
    QuerySome<a> everythingQ = null;
    if (everythingQ==null)
      everythingQ = new QuerySome<a>(){
       public a eval(Object o){return dies.everything(o);}
       public a eval(a x,a y) {return dies.eval(x,y);}
       public boolean condition(Object o){return dies.condition(o);}
      };
    for (a y:everythingQ.gmap(x)) result=eval(result,y);
    return result;
  }
}
```

The class `Query` could thus be implemented as the specialisation of `QuerySome` with a constant `true` condition.

Query2.java

```
package name.panitz.boilerplate;
import java.util.*;
abstract class Query2<a> extends QuerySome<a>{
  public boolean condition(Object o){return true;}
}
```

No we can write a specialized version for creation of an XML documents, which meets the special needs of the company structure. Special cases for two node classes are given. Further traversal is stopped for these nodes:

ToXML2.java

```
package name.panitz.boilerplate;
import org.w3c.dom.*;
import java.util.*;
import static name.panitz.boilerplate.ToXML.*;
class ToXML2 extends QuerySome<Node>{
  public Node eval(Object o){
    if (o instanceof Person){
      Element p = doc.createElement("Person");
      Element n = doc.createElement("Name");
      Element a = doc.createElement("Address");
      n.appendChild(doc.createTextNode(((Person)o).name));
      a.appendChild(doc.createTextNode(((Person)o).address));
      p.appendChild(n);
      p.appendChild(a);
      return p;
    }
    if (o instanceof Dept){
      Dept d =(Dept)o;
      Element result = doc.createElement("Dept");
      Element n = doc.createElement("Name");
      n.appendChild(doc.createTextNode(d.name));
      result.appendChild(n);
      result.appendChild(everything(d.manager));
      for (SubUnit su:d.subUnits) result.appendChild(everything(su));
```

```
      return result;
    }
    if (o instanceof Iterable)
      return doc.createElement(o.getClass().getSimpleName());
    return doc.createTextNode(o.toString());
  }
  public boolean condition(Object o){
    return ! (o instanceof Person||o instanceof Dept);}
  public Node eval(Node x,Node y){x.appendChild(y);return x;}
}
```

This will generate an XML document according to `company.dtd`.

TestToXML2.java

```
package name.panitz.boilerplate;
import static  name.panitz.boilerplate.TestToXML.*;
class TestToXML2 extends ToXML2{
  public static void main(String [] args){
    System.out.println(serialize(new ToXML2().everything(MyComp.getCompany())));}}
```

9.2.7 Conclusion

The pattern presented in this paper is very simple, general and powerfull. The techniques presented in the original paper for the programming language Haskell[PJ03] could directly be adapted to an object oriented framework. Instead of higher order functions subtyping and late-binding are used. The functions `gmap`, `everywhere` and `everhting` are parameterized by overridden methods of the transform and query objects.

Things are even simpler than in the Haskell framework, since most necessary means are available: a common object type and a type instance operator.

The pattern can be implemented without using the Java's reflection mechanism. Thus it avoids performance problems concerned with reflection. Reflection can simplify the usage of the pattern, but is not recommendable for reasons of performance.

Future Work

We did not yet explore if with some minimal and optimized use of reflection, construction of trees can be implemented with adequate performance. This would enable to write generic code for the inverse functions to the transformers `ToXML` or `Show`.

In [Vis01] a system based on visitor combination is presented. It might be fruitful to apply these techniques to the transformer presented in this paper.

9.2.8 Company classes

Company.java

```
package name.panitz.boilerplate;
import java.util.*;
class Company extends ArrayList<Dept>{}
```

Dept.java

```
package name.panitz.boilerplate;
import java.util.*;
class Dept implements Iterable<Object>{
  String name;
  Employee manager;
```

```
  List<SubUnit> subUnits;
  Dept(String name,Employee manager,List<SubUnit> subUnits){
    this.name=name;this.manager=manager;this.subUnits=subUnits;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();
    result.add(name);result.add(manager);result.add(subUnits);
    return result.iterator();
  }
}
```

SubUnit.java

```
package name.panitz.boilerplate;
import java.util.*;
class SubUnit implements Iterable<Object>{
  public Iterator<Object> iterator(){return new ArrayList<Object>().iterator();}
}
```

PU.java

```
package name.panitz.boilerplate;
import java.util.*;
class PU extends SubUnit{
  Employee employee;
  PU(Employee employee){this.employee=employee;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();result.add(employee);
    return result.iterator();
  }
}
```

DU.java

```
package name.panitz.boilerplate;
import java.util.*;
class DU extends SubUnit{
  Dept dept;
  DU(Dept dept){this.dept=dept;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();result.add(dept);
    return result.iterator();
  }
}
```

Employee.java

```
package name.panitz.boilerplate;
import java.util.*;
class Employee implements Iterable<Object>{
  Person person;Salary salary;
  Employee(Person person,Salary salary){this.person=person;this.salary=salary;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();
    result.add(person);result.add(salary);
    return result.iterator();
  }
}
```

Person.java

```
package name.panitz.boilerplate;
import java.util.*;
class Person implements Iterable<Object>{
  String name;String address;
  Person(String name,String address){this.name=name;this.address=address;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();
    result.add(name);result.add(address);
    return result.iterator();
  }
}
```

Salary.java

```
package name.panitz.boilerplate;
import java.util.*;
class Salary implements Iterable<Object>{
  Double amount;
  Salary(Double amount){this.amount=amount;}
  public Iterator<Object> iterator(){
    List<Object> result = new ArrayList<Object>();result.add(amount);
    return result.iterator();
  }
}
```

9.3 Generierung von Code mit APT

9.3.1 ADT

Algebraic data types are some powerfull means of programming. They constitute a high level and abstract way of declarative programming. Functional programming languages like Haskell[PJ03], Clean[Pv97] or ML[MTH90][Ler97] provide build-in support for algebraic types. An algebraic type is defined by a set of constructors. In functional languages algebraic types can be directly declared; e.g. the following Haskell code defines a type of binary trees:

Tree.hs

```
data Tree a =
   Branch (Tree a) a (Tree a)
  |Empty
```

Algorithms on algebraic types are usually written as equations that match one of the constructor alternatives of the type. Thus the function that calculates the size of a binary tree can be written with the following two equations:

Tree.hs

```
size Empty          = 0
size (Branch l a r) = size l + 1 + size r
```

Common object oriented languages do not directly support algebraic types. Programming patterns need to be applied. The usual technique used in examples like the one above is the visitor pattern [GHJV95]. The algebraic type is expressed by a base class. The constructors are different subclasses of this. A visitor overloads a method `visit` for each of the subclasses representing a constructor. This overloaded method implements the pattern equations as seen in the Haskell example above.

The visitor pattern amounts in a lot of boilerplate code. The elegant four lines of Haskell in the example above result in at least four classes of Java. Therefor extensions to Java have been proposed, which directly support algebraic types, as e.g. *Pizza*[OW97], which has been around for quite a while. However, these extensions are rarely used in practice. One reason for this certainly is that programmers don't want to leave the official pure Java thread.

Recently algebraic data types have been integrated in the hybrid functional object oriented language *Scala*[Oa04].

With Java 5 an annotation framework has been introduced to Java. Code can be annotated. These may not just be commentary annotations but an integral part of the program. Annotations may be evaluated by some processor. They open a door to language extensions without leaving the Java thread. Typical annotations are concerned with persistency code, transaction code or serialization code.

We will use annotions for generation of boilerplate code for the visitor pattern.

9.3.2 Annotations for Algebraic Types

We propose two annotations for algebraic types in Java. The annotation `@Data` for classes, which indicates that the class represents an algebraic type, and the annotation `@Constr`, which indicates that a method is a constructor case for the algebraic type.

Before we have a look at the actual implementation of the annotations we start with an example. With these two annotations binary trees can now be defined in Java as follows:

T.java

```
package example.tree;
import name.panitz.adt.*;
@Data class T<a> {
  @Constr void Branch(T<a> left,a element,T<a> right){};
  @Constr void Empty(){};
}
```

This looks quite similar to the implementation in Haskell and compares well to the corresponding definition in *Scala*. An algebraic type `T` is defined. This type is generic. It has a type variable for the node contents. Note that the class above is a normal Java class. It can be compiled with the standard Java compiler `javac`. However for generation of the boilerplate code we will need to use Java's annotation processing tool `apt`.

Algorithms on algebraic types

In the Haskell case algorithms over algebraic types could easily declared by pattern equations. We will generate a bunch of classes for the definition above. These provide an general visitor class `TVisitor`, a common base class `TAdt` and specialized classes for the constructors `Branch` and `Empty`. With these an algorithm over trees can be implemented as a specialized visitor. The `visit` function gets overloaded for the two constructor cases:

TSize.java

```
package example.tree;
public class TSize<a> extends TVisitor<a,Integer>{
  public Integer visit(Branch<a> x){
    return size(x.left) + 1 + size(x.right);}
  public Integer visit(Empty<a> _){return 0;}
```

A general function `size` was assumed, that calls the methods `welcome`. For some technical reason, we need a cast to the generated base class `TAdt`.

TSize.java

```
  public int size(T<a> t){return ((TAdt<a>)t).welcome(this);}
}
```

In fact, we are not far away from the Haskell implementation. In the next section we will reveal, how the processing of the two annotations `@Data` and `@Constr` has been implemented.

9.3.3 Implementation

In this section we give the complete implementation of the annotations for algebraic types in Java. There is no hidden code. In order to meet the size restriction of a conference paper, brevity is preferred over clarity in the code.

First of all, annotations are interfaces that need to be defined in Java.

Data.java

```
package name.panitz.adt;
public @interface Data{}
```

Constr.java

```
package name.panitz.adt;
public @interface Constr{}
```

Now we need a generator. Generators can be loaded by the `apt` tool through a factory class. The standard way to implement this is by way of a subclass of `AnnotationProcessorFactory`.

AdtGenFact.java

```
package name.panitz.adt;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.util.*;

public class AdtGenFact implements AnnotationProcessorFactory{
  static final String DATA_ANNOT   = "name.panitz.adt.Data";
  static final String CONSTR_ANNOT = "name.panitz.adt.Constr";

  public AnnotationProcessor getProcessorFor
    (Set<AnnotationTypeDeclaration> types
    ,AnnotationProcessorEnvironment env){
      return new AdtGen(types,env);
  }
  public Collection<String> supportedAnnotationTypes(){
    ArrayList<String> result= new ArrayList<String>();
    result.add(CONSTR_ANNOT);result.add(DATA_ANNOT);
    return result;
  }
  public Collection<String> supportedOptions(){
    return new ArrayList<String>();
  }
}
```

We decided for the most simple implementation, which simply returns a instance of the actual generator class.

AdtGen.java

```
package name.panitz.adt;
import com.sun.mirror.apt.*;
import com.sun.mirror.util.*;
import com.sun.mirror.declaration.*;
import java.util.*;
public class AdtGen extends SimpleDeclarationVisitor
                    implements AnnotationProcessor{
  Set<AnnotationTypeDeclaration> types;
  AnnotationProcessorEnvironment env;
  public AdtGen(Set<AnnotationTypeDeclaration> types
                ,AnnotationProcessorEnvironment env){
    this.types=types;
    this.env=env;
    constrDeclaration = env.getTypeDeclaration(AdtGenFact.CONSTR_ANNOT);
    dataDeclaration = env.getTypeDeclaration(AdtGenFact.DATA_ANNOT);
  }
  public void process(){
   for (TypeDeclaration td :env.getSpecifiedTypeDeclarations()){
      td.accept(this);
   }
  }
```

The actual generator is implemented as an visitor over the declaration types in package `com.sun.mirror.declaration`.

AdtGen.java

```
  final private TypeDeclaration constrDeclaration;
  final private TypeDeclaration dataDeclaration;

  boolean hasAnnot(TypeDeclaration a,Declaration m){
```

```
    for (AnnotationMirror am : m.getAnnotationMirrors())
      if(am.getAnnotationType().getDeclaration().equals(a))
        return true;
    return false;
  }

  public void visitClassDeclaration(ClassDeclaration d){
   if (hasAnnot(dataDeclaration,d)){
    ADT adt = new ADT(env,d);
    for (MethodDeclaration m:d.getMethods()){
      if (hasAnnot(constrDeclaration,m)){
        adt.addConstr(m.getSimpleName(),m.getParameters());
      }
    }
    adt.generateClasses();
  }}}
```

Eventually we come to the actual generation of the boilerplate code.

representation of algebraic types

The main class `ADT` represents an algebraic type:

ADT.java

```
package name.panitz.adt;

import java.util.*;
import java.io.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.apt.*;

public class ADT {
```

It needs the following information:

- a name for the algebraic type.
- the package for the type.
- a list of constructors.
- furthermore we need the `AnnotationProcessorEnvironment`, which will give us a filer, for generating new source files.

ADT.java

```
  String name;
  String thePackage;
  public List<Constructor> constructors;
  TypeDeclaration cd;
  final Filer filer;
```

These fields get initialized in a simple constructor:

ADT.java

```
  public ADT
    (AnnotationProcessorEnvironment env,TypeDeclaration cd){
      thePackage = cd.getPackage().getQualifiedName();
      name=cd.getSimpleName();
```

```
      constructors=new ArrayList<Constructor>();
      this.cd=cd;
      filer=env.getFiler();
  }
```

We provide some auxiliary methods for retrieving the name of the type and the textual representation of the type parameters:

ADT.java

```
  public String getFullName(){return cd.toString();}
  public String getName(){return name;}
  String commaSepPs(){
    final String qN=getFullName();
    final int index=qN.indexOf('<');
    return index>=0?qN.substring(index+1,qN.length()-1):"";
  }
  public String getParamList(){
    return commaSepPs().length()==0?"":("<"+commaSepPs()+">");
  }
  String getPackageDef(){
    return  thePackage.length()==0?""
           :"package "+thePackage+";\n\n";
  }
```

The following method is used to add a further constructor to the algebraic type. The class `Constructor` will be defined as inner class of `ADT` below.

ADT.java

```
  void addConstr(String n,Collection<ParameterDeclaration> ps){
   constructors.add(new Constructor(n,ps));
}
```

Wie need to generate:

- a common base class for the type,
- a visitor interface,
- and for each constuctor a subclass of the base class.

ADT.java

```
  public void generateClasses(){
    try{generateClass();
        generateVisitorClass();
        for (Constructor c:constructors)c.generateClass(this);
    }catch (IOException _){}}
```

Generating the base class The base class extends the class, which has been annotated as `@Data` class. It has the same name with suffix `Adt`. We use the filer to create the source for the base class:

ADT.java

```
  public void generateClass() throws IOException{
    final String fullName = getFullName();
    Writer out
     = filer.createSourceFile(thePackage+"."+name+"Adt");
```

The generated class is abstract.

ADT.java
```
    out.write( getPackageDef());
    out.write("public abstract class ");
    out.write(getName()+"Adt"+getParamList());
    out.write(" extends "+fullName+"\n");
    out.write(" implements Iterable<Object>{\n");
```

It has the abstract method `welcome`. This is the method that welcomes a visitor.[4] It is an generic method. Its type variable, which we hard coded as `b_`, represents the result type of the visitor.

ADT.java
```
    out.write("  abstract public <b_> b_ welcome("
            +name+"Visitor<" + commaSepPs()
                             +(commaSepPs().length()==0?"":",")
                             +"b_> visitor);\n");
    out.write("}");
    out.close();
  }
```

Generating the visitor interface The next class that gets generated is the general visitor class. We decided for an abstract class, which has an abstract method `visit` for each constructor, and additionally a concrete general method `visit` overloaded for the base class. This ensures that cases, which were forgotten to be implemented in a concrete visitor class will cause a runtime exception.

The visitor class is generic. It has the type variables of the base class and additionally a type variable for the result type of the methods `visit`.

ADT.java
```
  public void generateVisitorClass(){
    try{
      final String csName = name+"Visitor";
      final String fullName
       = csName+"<"+commaSepPs()
                  +(commaSepPs().length()==0?"":",")+"result>";
      Writer out=filer.createSourceFile(thePackage+"."+csName);
      out.write( getPackageDef()+ "\n");
      out.write("public abstract class ");
      out.write(fullName+"{\n");
      for (Constructor c:constructors)
        out.write("  "+c.mkVisitMethod(this)+"\n");

      out.write("  public result visit("+getFullName()+" xs){");
      out.write("\n    throw new RuntimeException(");
      out.write("\"unmatched pattern: \"+xs.getClass());\n");
      out.write("  }\n}");
      out.close();
    }catch (Exception _){}
  }
```

Generating constructor classes

Eventually we generate classes for each of the constructors in the algebraic type. Constructors are represented by an inner class of class `ADT`:

[4] This usually is called `accept`.

ADT.java
```
private class Constructor {
```

Constructors have a name and list of parameters:

ADT.java
```
  String name;
  Collection<ParameterDeclaration> params;
```

These get initialized during construction:

ADT.java
```
  public Constructor
             (String n,Collection<ParameterDeclaration> ps){
    name=n;params=ps;}
```

We generate a class with the name of the constructor. It extends the base class and has the same type variables as the base class:

ADT.java
```
  public void generateClass(ADT theType){
    try{
      Writer out
        = filer.createSourceFile(theType.thePackage+"."+name);
      out.write( theType.getPackageDef());
      out.write("public class "+name);
      out.write(theType.getParamList()+" extends ");
      out.write(theType.getName()+"Adt"+theType.getParamList());
      out.write("{\n");
```

In the body of the class we generate:

- fields for the arguments of the constructor.
- the method `welcome`
- the method `equals`
- the method `toString`
- the method `iterator`

This is done in seperate methods:

ADT.java
```
      mkFields(out);
      mkConstructor(out);
      mkWelcomeMethod(theType, out);
      mkToStringMethod(out);
      mkEqualsMethod(out);
      mkIteratorMethod(out);
      out.write("}\n");out.close();
    }catch (Exception _){}
  }
```

Fields For every argument of the constructor we generate a field in the class.

ADT.java
```
  private void mkFields(Writer out)throws IOException{
    for (ParameterDeclaration p:params){
     out.write("  public "+p.getType().toString()+" ");
     out.write(p.getSimpleName());
     out.write(";\n");
    }
  }
```

Constructor A single constructor is generated, which initializes the fields.

ADT.java
```
  private void mkConstructor(Writer out)throws IOException{
    out.write("\n  public "+name+"(");
    boolean first= true;
    for (ParameterDeclaration p:params){
      if (!first){out.write(",");}
      out.write(p.getType().toString()+" ");
      out.write(p.getSimpleName());
      first=false;
    }
    out.write("){\n");
    for (ParameterDeclaration p:params){
      out.write("    this."+p.getSimpleName()+" = ");
      out.write(p.getSimpleName()+";\n");
    }
    out.write("  }\n\n");
  }
```

Get-methods For each private field a public get-method is generated.

ADT.java
```
  private void mkGetterMethods(Writer out)throws IOException{
    for (ParameterDeclaration p:params){
      out.write("  public ");
      out.write(p.getType().toString());
      out.write(" get");
      out.write(
       Character.toUpperCase(p.getSimpleName().charAt(0)));
      out.write(p.getSimpleName().substring(1));
      out.write("(){return "+p.getSimpleName() +";}\n");
    }
  }
```

Set-methods For each private field a public set-method is generated.

ADT.java
```
  private void mkSetterMethods(Writer out)throws IOException{
    for (ParameterDeclaration p:params){
      out.write("  public void set");
      out.write(
       Character.toUpperCase(p.getSimpleName().charAt(0)));
      out.write(p.getSimpleName().substring(1));
      out.write("(");
      out.write(p.getType().toString());
      out.write(" ");
      out.write(p.getSimpleName());
      out.write("){this."+p.getSimpleName());
```

```
      out.write("= "+p.getSimpleName()+";}\n");
    }
  }
```

Welcome method We generate the standard method `welcome`, which calls the method `visit` of the passed visitor with this object:

ADT.java

```
  private void mkWelcomeMethod(ADT theType,Writer out)
        throws IOException{
      out.write("  public <_b> _b welcome("
                +theType.name+"Visitor<"+theType.commaSepPs()
                      +(theType.commaSepPs().length()==0?"":",")
                                +"_b> visitor){"
                +"\n    return visitor.visit(this);\n  }\n");
  }
```

toString For the sake of comfort, we also generate the method `toString` in a natural way.

ADT.java

```
  private void mkToStringMethod(Writer out) throws IOException{
    out.write("  public String toString(){\n");
    out.write("    return \""+name+"(\"");
    boolean first=true;
    for (ParameterDeclaration p:params){
     if (first){first=false;}
     else out.write("+\",\"");
     out.write("+"+p.getSimpleName());
    }
    out.write("+\")\";\n  }\n");
  }
```

equals In the same way a standard method `equals` is generated:

ADT.java

```
  private void mkEqualsMethod(Writer out) throws IOException{
    out.write("  public boolean equals(Object other){\n");
    out.write("    if (!(other instanceof "+name+")) ");
    out.write("return false;\n");
    out.write("    final "+name+" o= ("+name+") other;\n");
    out.write("    return true  ");
    for (ParameterDeclaration p:params){
      out.write("&& "+p.getSimpleName()
                      +".equals(o."+p.getSimpleName()+")");
    }
    out.write(";\n  }\n");
  }
```

iterator In the same way a method `iterator` is generated. This enables us to apply the *scrap your boilerplate code*[LP03][LP04] pattern to the generated algebraic type.

ADT.java

```
  private void mkIteratorMethod(Writer out) throws IOException{
    out.write("  public java.util.Iterator<Object>iterator(){");
    out.write("\n    java.util.List<Object> res\n");
    out.write("         =new java.util.ArrayList<Object>();\n");
    for (ParameterDeclaration p:params)
```

```
      out.write("    res.add("+p.getSimpleName()+");\n");
    out.write("    return res.iterator();\n  }\n");
  }
```

the method visit For each constructor an abstract method `visit` had been generated in the general abstract visitor class. This was done by calls to the following method.

ADT.java

```
  public String mkVisitMethod(ADT theType){
    return "public abstract result visit("
            +name+theType.getParamList()+" _);";
  }
}}
```

9.3.4 Example: a tiny imperativ programming language

A typical example application for the visitor pattern is compiler construction. The abstract syntax tree of a programming language can easily be defined as an algebraic type. The following `@Data` class defines an abstract syntax tree of statements in a tiny imperative programming language.

Klip.java

```
package name.panitz.adt.examples;
import name.panitz.adt.*;
import java.util.List;

abstract @Data class Klip implements Iterable<Object> {
  @Constr void Num(Integer i){};
  @Constr void Add(Klip e1,Klip e2){};
  @Constr void Mul(Klip e1,Klip e2){};
  @Constr void Sub(Klip e1,Klip e2){};
  @Constr void Div(Klip e1,Klip e2){};
  @Constr void Var(String name){};
  @Constr void Assign(String var,Klip e){};
  @Constr void While(Klip cond,Klip body){};
  @Constr void Block(List<Klip> stats){};
}
```

Show

The first visitor we present for this type, represents it as a `String`. The corresponding visitor can be written as:

ShowKlip.java

```
package name.panitz.adt.examples;
import name.panitz.*;
import java.util.*;

public class ShowKlip extends KlipVisitor<String> {
  public String s(Klip a){return ((KlipAdt)a).welcome(this);}

  public String visit(Num x){return x.i.toString();}
  public String visit(Add x){return"("+s(x.e1)+"+"+s(x.e2)+")";}
  public String visit(Sub x){return"("+s(x.e1)+"-"+s(x.e2)+")";}
  public String visit(Div x){return"("+s(x.e1)+"/"+s(x.e2)+")";}
  public String visit(Mul x){return"("+s(x.e1)+"*"+s(x.e2)+")";}
  public String visit(Var v){return v.name;}
```

```
  public String visit(Assign x){return x.var+" := "+s(x.e);}
  public String visit(Block b){
    StringBuffer result=new StringBuffer();
    for (Klip x:b.stats)  result.append(s(x)+";\n");
    return result.toString();}
  public String visit(While w){
    StringBuffer result=new StringBuffer("while (");
    result.append(s(w.cond)+"){\n");
    result.append(s(w.body)+"\n}");
    return result.toString();}
}
```

This will e.g. print a program for calculating the factorial as follows:

fak.klip

```
x := 5;
y := 1;
while (x){
y := (y*x);
x := (x-1);

};
y;
```

Evaluation visitor

A further visitor defines an interpreter for the tiny imperative language. This visitor has an internal map for binding of the variables used in the program:

EvalKlip.java

```
package name.panitz.adt.examples;
import name.panitz.*;
import java.util.*;

public class EvalKlip extends KlipVisitor<Integer> {
 Map<String,Integer> env = new HashMap<String,Integer>();
 public Integer v(Klip x){return ((KlipAdt)x).welcome(this);}

 public Integer visit(Num x){return x.i;}
 public Integer visit(Add x){return v(x.e1)+v(x.e2);}
 public Integer visit(Sub x){return v(x.e1)-v(x.e2);}
 public Integer visit(Div x){return v(x.e1)/v(x.e2);}
 public Integer visit(Mul x){return v(x.e1)*v(x.e2);}
 public Integer visit(Var v){return env.get(v.name);}
 public Integer visit(Assign ass){
   final Integer i = v(ass.e);env.put(ass.var,i);
   return i;}
 public Integer visit(Block b){
    Integer result = 0;
    for (Klip x:b.stats) result=v(x);
    return result;}
 public Integer visit(While w){
    Integer result = 0;
    while (v(w.cond)!=0) result = v(w.body);
    return result;}
}
```

Generic queries

Since we implemented the interface `Iterable<Object>` for all node classes, we can apply the *scrap your boilerplate* pattern.[Pan05]. Arbitrary tree traversal code can be implemented through subclasses of `Query` or `Transform`.

Query The following simple example collects all variable names within an abstract syntax tree.

GetVars.java

```
package name.panitz.adt.examples;
import name.panitz.boilerplate.Query;
import java.util.*;
class GetVars extends Query<Set<String>>{
  public Set<String> eval(Object x){
    Set<String> result = new HashSet<String>();
    if (x instanceof Var) result.add(((Var)x).name);
    if (x instanceof Assign) result.add(((Assign)x).var);
    return result;
  }
  public Set<String> eval(Set<String> x,Set<String> y){
    x.addAll(y);
    return x;
  }
}
```

Transform The following transformer adds a prefix to every variable name occuring in the abstract syntax tree.

AddPrefix.java

```
package name.panitz.adt.examples;
import name.panitz.boilerplate.Transform;
import java.util.*;
public class AddPrefix extends Transform{
  final String prefix;
  public AddPrefix(String prefix){this.prefix=prefix;}
  public void transform(Object x){
    if (x instanceof Var) ((Var)x).name=prefix+((Var)x).name;
    else if (x instanceof Assign)
       ((Assign)x).var=prefix+((Assign)x).var;;
  }
}
```

A `javacc` parser definition for the tiny imperative language can be found at the corresponding website to this paper (`http://panitz.name/paper/adt`) .

The different algorithms on the abstract syntax tree in the previous subsections can be invoked as follows

TestKlip.java

```
package name.panitz.adt.examples;
import java.io.FileReader;
public class TestKlip{
  public static void main(String [] args)throws Exception{
   final Klip klip = new KlipParser(new FileReader(args[0]))
                     .statementList();

```

```
    System.out.println(new ShowKlip().s(klip));
    System.out.println(new EvalKlip().v(klip));
    System.out.println(new GetVars().everything(klip));
    new AddPrefix("$").everywhere(klip);
    System.out.println(new ShowKlip().s(klip));
  }
}
```

9.3.5 javacc Parser für Klip

Pair.java

```
package name.panitz.adt;
public class Pair<a,b>{
  a e1;
  b e2;
  Pair(a e1,b e2){this.e1=e1;this.e2=e2;}
}
```

Schließlich, um Klipprogramme ausführen zu können, benötigen wir einen Parser, der die textuelle Darstellung eines Klipprogramms in die Baumstruktur umwandelt. Wir schreiben einen solchen Parser mit Hilfe des Parsergenerators `javacc`.

Der Parser soll zunächst eine Hauptmethode enthalten, die ein Klipprogramm parst und die beiden Besucher auf ihn anwendet:

KlipParser.jj

```
options {
   STATIC=false;
}

PARSER_BEGIN(KlipParser)
package name.panitz.adt.examples;

import  name.panitz.adt.Pair;
import java.util.*;
import java.io.FileReader;

public class KlipParser  {
  public static void main(String [] args)throws Exception{
    final Klip klip = new KlipParser(new FileReader(args[0]))
                      .statementList();

    System.out.println(new ShowKlip().s(klip));
    System.out.println(new EvalKlip().v(klip));
    System.out.println(new GetVars().everything(klip));
    new AddPrefix("$").everywhere(klip);
    System.out.println(new ShowKlip().s(klip));
  }
}
PARSER_END(KlipParser)
```

Scanner

In einer `javacc`-Grammatik wird zunächst die Menge der Terminalsymbole spezifiziert.

KlipParser.jj

```
TOKEN :
{<WHILE: "while">
```

```
|<#ALPHA:        ["a"-"z","A"-"Z","_",".']        >
|<NUM:                ["0"-"9"]                >
|<#ALPHANUM:        <ALPHA> | <NUM>                >
|<NAME: <ALPHA> ( <ALPHANUM> )*>
|<ASS: ":=">
|<LPAR: "(">
|<RPAR: ")">
|<LBRACKET: "{">
|<RBRACKET: "}">
|<SEMICOLON: ";">
|<STAR: "*">
|<PLUS: "+">
|<SUB: "-">
|<DIV: "/">
}
```

Zusätzlich läßt sich spezifizieren, welche Zeichen als Leerzeichen anzusehen sind:

KlipParser.jj

```
SKIP :
{ "\u0020"
| "\t"
| "\n"
| "\r"
}
```

Parser

Es folgen die Regeln der Klip-Grammatik. Ein Klip Programm ist zunächst eine Sequenz von Befehlen:

KlipParser.jj

```
Klip statementList() :
{  List<Klip> stats = new ArrayList<Klip>();
   Klip stat;}
{
  (stat=statement() {stats.add(stat);} <SEMICOLON>)*
  {return new Block(stats);}
}
```

Ein Befehl kann zunächst ein arithmetischer Ausdruck in Punktrechnung sein.

KlipParser.jj

```
Klip statement():
{Klip e2;Klip result;boolean sub=false;}
{
  result=multExpr()
  [ (<PLUS>|<SUB>{sub=true;}) e2=statement()
   {result = sub?new Sub(result,e2):new Add(result,e2);}]
  {return result;}
}
```

Die Operanden der Punktrechnung sind arithmetische Ausdruck in Strichrechnung. Auf diese Weise realisiert der Parser einen Klip-Baum, in dem Punktrechnung stärker bindet als Strichrechnung.

KlipParser.jj

```
Klip multExpr():
{Klip e2;Klip result;boolean div= false;}
{
```

```
  result=atomicExpr()
  [ (<STAR>|<DIV>{div=true;})
    e2=multExpr()
    {result = div?new Div(result,e2):new Mul(result,e2);}]
  {return result;}
}
```

Die Operanden der Punktrechnung sind entweder Literale, Variablen, Zuweisungen, Schleifen oder geklammerte Ausdrücke.

KlipParser.jj

```
Klip atomicExpr():
{Klip result;}
{
   (result=integerLiteral()
   |result=varOrAssign()
   |result=whileStat()
   |result=parenthesesExpr()
   )
  {return result;}
}
```

Ein Literal ist eine Sequenz von Ziffern.

KlipParser.jj

```
Klip integerLiteral():
{ int result = 0;
  Token n;
  boolean m=false;}
{  [<SUB> {m = true;}]
   (n=<NUM>
    {result=result*10+n.toString().charAt(0)-48;})+
   {return new Num(new Integer(m?-result:result));}
}
```

Geklammerte Ausdrücke klammern beliebige Befehle.

KlipParser.jj

```
Klip parenthesesExpr():
{Klip result;}
{ <LPAR> result = statement() <RPAR>
{return result;}}
```

Variablen können einzeln oder auf der linken Seite einer Zuweisung auftreten.

KlipParser.jj

```
Klip varOrAssign():
{ Token n;Klip result;Klip stat;}
{ n=<NAME>{result=new Var(n.toString());}
  [<ASS> stat=statement()
    {result = new Assign(n.toString(),stat);}
  ]
  {return result;}
}
```

Und schließlich noch die Regel für die `while`-Schleife.

KlipParser.jj

```
Klip whileStat():{
  Klip cond; Klip body;}
{ <WHILE> <LPAR>cond=statement()<RPAR>
```

```
  <LBRACKET> body=statementList()<RBRACKET>
  {return new While(cond,body);}
}
```

Klip-Beispiele

Unser Klip-Interpreter ist fertig. Wir können Klip-Programme ausführen lassen.

Zunächst mal zwei Programme, die die Arithmetik demonstrieren:

arith1.klip

```
2*7+14*2;
```

arith2.klip

```
2*(7+9)*2;
```

```
sep@linux:~> java name.panitz.adt.examples.KlipParser arith1.klip
((2 * 7) + (14 * 2));

42
sep@linux:~> java name.panitz.adt.examples.KlipParser arith2.klip
(2 * ((7 + 9) * 2));

64
sep@linux:~>
```

Auch unser erstes Fakultätsprogramm in Klip läßt sich ausführen:

```
sep@linux:~> java name.panitz.adt.examples.KlipParser fak.klip
x := 5;
y := 1;
while (x){
y := (y * x);
x := (x - 1);

};
y;

{y=1, x=5}
{y=5, x=4}
{y=20, x=3}
{y=60, x=2}
{y=120, x=1}
120
sep@linux:~>
```

Wie man sieht bekommen wir auch eine Traceausgabe über die Umgebung während der Auswertung.

9.3.6 Conclusion

We have presented an example of how Java annotations can be used to generate boilerplate code for a usefull programming pattern. The implementation of the code generator was fairly easy and could be printed out completely within a conference paper. The annotations extend

Java with some new high level programming construct: a declarative way of defining algebraic data types. While we achieved this, we did not even exploit the full power of Java annotations.

A drawback to our solutions is, that it is not possible to add features to existing classes other than by way of subtyping. Instead of creating a new base class `KlipAdt` we would have liked to change class `Klip`. This amounts in casts from the defining class to the generated base class. It does not seem to be possible, to change existing classes.

Generation of classes with the `apt` api is (still) somewhat clumsy. A writer is used. There are no classes, which help to construct new source code.

It turns out that annotations offer a new way of programming: instead of switching to newer powerfull languages, which need an seperate compiler, the language can be extended with new features. We expect to see quite a number of such extensions to come up in the future. A lot of common patterns seem to be candidates for generation of boilerplate code driven by annotations.

However, this mechanism seems to be limited. It might be interesting future work, to investigate how far we can go. Is it possible to express some real pattern matching expression in terms of Java annotations?

Kapitel 10

XML

XML ist eine Sprache, die es erlaubt Dokumente mit einer logischen Struktur zu beschreiben. Die Grundidee dahinter ist, die logische Struktur eines Dokuments von seiner Visualisierung zu trennen. Ein Dokument mit einer bestimmten logischen Struktur kann für verschiedene Medien unterschiedlich visualisiert werden, z.B. als HTML-Dokument für die Darstellung in einem Webbrowser, als pdf- oder postscript-Datei für den Druck des Dokuments und das für unterschiedliche Druckformate. Eventuell sollen nicht alle Teile eines Dokuments visualisiert werden. XML ist zunächst eine Sprache, die logisch strukturierte Dokumente zu schreiben, erlaubt.

Dokumente bestehen hierbei aus den eigentlichen Dokumenttext und zusätzlich aus Markierungen dieses Textes. Die Markierungen sind in spitzen Klammern eingeschlossen.

Beispiel: Der eigentliche Text des Dokuments sei:

```
The Beatles White Album
```

Die einzelnen Bestandteile dieses Textes können markiert werden:

```
<cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
</cd>
```

Die XML-Sprache wird durch ein Industriekonsortium definiert, dem W3C (`http://www.w3c.org`) . Dieses ist ein Zusammenschluß vieler Firmen, die ein gemeinsames Interesse eines allgemeinen Standards für eine Markierungssprache haben. Die eigentlichen Standards des W3C heißen nicht Standard, sondern Empfehlung *(recommendation)*, weil es sich bei dem W3C nicht um eine staatliche oder überstaatliche Standartisiertungsbehörde handelt. Die aktuelle Empfehlung für XML liegt seit anfang des Jahres als Empfehlung in der Version 1.1 vor [T. 04].

XML enstand Ende der 90er Jahre und ist abgeleitet von einer umfangreicheren Dokumentenbeschreibungssprache: SGML. Der SGML-Standard ist wesentlich komplizierter und krankt daran, daß es extrem schwer ist, Software für die Verarbeitung von SGML-Dokumenten zu

entwickeln. Daher fasste SGML nur Fuß in Bereichen, wo gut strukturierte, leicht wartbare Dokumente von fundamentaler Bedeutung waren, so daß die Investition in teure Werkzeuge zur Erzeugung und Pflege von SGML-Dokumenten sich rentierte. Dies waren z.B. Dokumentationen im Luftfahrtbereich.[1]

Die Idee bei der Entwicklung von XML war: eine Sprache mit den Vorteilen von SGML zu Entwickeln, die klein, übersichtlich und leicht zu handhaben ist.

10.1 XML-Format

Die grundlegendste Empfehlung des W3C legt fest, wann ein Dokument ein gültiges XML-Dokument ist, die Syntax eines XML-Dokuments. Die nächsten Abschnitte stellen die wichtigsten Bestandteile eines XML-Dokuments vor.

Jedes Dokument beginnt mit einer Anfangszeile, in dem das Dokument angibt, daß es ein XML-Dokument nach einer bestimmten Version der XML Empfehlung ist:

```
<?xml version="1.0"?>
```

Dieses ist die erste Zeile eines XML-Dokuments. Vor dieser Zeile darf kein Leerzeichen stehen. Die derzeitig aktuellste und einzige Version der XML-Empfehlung ist die Version 1.0. Ein Entwurf für die Version 1.1 liegt vor. Nach Aussage eines Mitglieds des W3C ist es sehr unwahrscheinlich, daß es jemals eine Version 2.0 von XML geben wird. Zuviele weitere Techniken und Empfehlungen basieren auf XML, so daß die Definition von dem, was ein XML-Dokument ist kaum mehr in größeren Rahmen zu ändern ist.

10.1.1 Elemente

Der Hauptbestandteil eines XML-Dokuments sind die Elemente. Dieses sind mit der Spitzenklammernotation um Teile des Dokuments gemachte Markierungen. Ein Element hat einen *Tagnamen*, der ein beliebiges Wort ohne Leerzeichen sein kann. Für einen Tagnamen *name* beginnt ein Element mit `<name>` und endet mit `</name>`. Zwischen dieser Start- und Endemarkierung eines Elements kann Text oder auch weitere Elemente stehen.

Es wird für XML-Dokument verlangt, daß es genau ein einziges oberstes Element hat.

Beispiel: Somit ist ein einfaches XML-Dokument ein solches Dokument, in dem der gesammte Text mit einem einzigen Element markiert ist:

```
<?xml version="1.0"?>
<myText>Dieses ist der Text des Dokuments. Er ist
mit genau einem Element markiert.
</myText>
```

[1] Man sagt, ein Pilot brauche den Copiloten, damit dieser die Handbücher für das Flugzeug trägt.

Im einführenden Beispiel haben wir schon ein XML-Dokument gesehen, das mehrere Elemente hat. Dort umschließt das Element `<cd>` zwei weitere Elemente, die Elemente `<artist>` und `<title>`. Die Teile, die ein Element umschließt, werden der Inhalt des Elements genannt.

Ein Element kann auch keinen, sprich den leeren Inhalt haben. Dann folgt der öffnenden Markierung direkt die schließende Markierung.

Beispiel: Folgendes Dokument enthält ein Element ohne Inhalt:

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page></page>
  <page>dritte Seite</page>
</skript>
```

Leere Elemente

Für ein Element mit Tagnamen *name*, das keinen Inhalt hat, gibt es die abkürzenden Schreibweise: `<name/>`

Beispiel: Das vorherige Dokument läßt sich somit auch wie folgt schreiben:

```
<?xml version="1.0"?>
<skript>
  <page>erste Seite</page>
  <page/>
  <page>dritte Seite</page>
</skript>
```

Gemischter Inhalt

Die bisherigen Beispiele haben nur Elemente gehabt, deren Inhalt entweder Elemente oder Text waren, aber nicht beides. Es ist aber auch möglich Elemente mit Text und Elementen als Inhalt zu schreiben. Man spricht dann vom gemischten Inhalt *(mixed content)*.

Beispiel: Ein Dokument, in dem das oberste Element einen gemischten Inhalt hat:

```
<?xml version="1.0"?>
<myText>Der <landsmann>Italiener</landsmann>
<eigename>Ferdinand Carulli</eigename> war  als Gitarrist
ebenso wie der <landsmann>Spanier</landsmann>
<eigename>Fernando Sor</eigename> in <ort>Paris</ort>
ansäßig.</myText>
```

XML-Dokumente als Bäume

Die wohl wichtigste Beschränkung für XML-Dokumente ist, daß sie eine hierarchische Struktur darstellen müssen. Zwei Elemente dürfen sich nicht überlappen. Ein Element darf erst wieder geschlossen werden, wenn alle nach ihm geöffneten Elemente wieder geschlossen wurden.

Beispiel: Das folgende ist kein gültiges XML-Dokument. Das Element `<bf>` wird geschlossen bevor das später geöffnete Element `<em>` geschlossen worde.

```
<?xml version="1.0"?>
<illegalDocument>
  <bf>fette Schrift <em>kursiv und fett</bf>
  nur noch kursiv</em>.
</illegalDocument>
```

Das Dokument wäre wie folgt als gültiges XML zu schreiben:

```
<?xml version="1.0"?>
<validDocument>
  <bf>fette Schrift <em>kursiv und fett</em></bf>
  <em>nur noch kursiv</em>.
</validDocument>
```

Dieses Dokument hat eine hierarchische Struktur.

Die hierarchische Struktur von XML-Dokumenten läßt sich sehr schön veranschaulichen, wenn man die Darstellung von XML-Dokumenten in Microsofts Internet Explorer betrachtet.

10.1.2 Attribute

Die Elemente eines XML-Dokuments können als zusätzliche Information auch noch Attribute haben. Attribute haben einen Namen und einen Wert. Syntaktisch ist ein Attribut dargestellt durch den Attributnamen gefolgt von einem Gleichheitszeichen gefolgt von dem in Anführungszeichen eingeschlossenen Attributwert. Attribute stehen im Starttag eines Elements.

Attribute werden nicht als Bestandteils des eigentlichen Textes eines Dokuments betrachtet.

Beispiel: Dokument mit einem Attribut für ein Element.

```
<?xml version="1.0"?>
<text>Mehr Information zu XML findet man auf den Seiten
des <link address="www.w3c.org">W3C</link>.</text>
```

10.1.3 Kommentare

XML stellt auch eine Möglichkeit zur Verfügung, bestimmte Texte als Kommentar einem Dokument zuzufügen. Diese Kommentare werden mit `<!--` begonnen und mit `-->` beendet. Kommentartexte sind nicht Bestandteil des eigentlichen Dokumenttextes.

Beispiel: Im folgenden Dokument ist ein Kommentar eingefügt:

```
<?xml version="1.0"?>
<drehbuch filmtitel="Ben Hur">
<akt>
  <szene>Ben Hur am Vorabend des Wagenrennens.
    <!--Diese Szene muß noch ausgearbeitet werden.-->
  </szene>
</akt>
</drehbuch>
```

10.1.4 Character Entities

Sobald in einem XML-Dokument eine der spitze Klammern `<` oder `>` auftaucht, wird dieses als Teil eines Elementtags interpretiert. Sollen diese Zeichen hingegen als Text und nicht als Teil der Markierung benutzt werden, sind also Bestandteil des Dokumenttextes, so muß man einen Fluchtmechanismus für diese Zeichen benutzen. Diese Fluchtmechanismen nennt man *character entities*. Eine Character Entity beginnt in XML mit dem Zeichen `&` und endet mit einem Semikolon `;`. Dazwischen steht der Name des Buchstabens. XML kennt die folgenden Character Entities:

Entity	Zeichen	Beschreibung
`<`	`<`	(less than)
`>`	`>`	(greater than)
`&`	`&`	(ampersant)
`"`	`"`	(quotation mark)
`'`	`'`	(apostroph)

Somit lassen sich in XML auch Dokumente schreiben, die diese Zeichen als Text beinhalten.

Beispiel: Folgendes Dokument benutzt Character Entities um mathematische Formeln zu schreiben:

```
<?xml version="1.0"?>
<gleichungen>
  <gleichung>x+1&gt;x</gleichung>
  <gleichung>x*x&lt;x*x*x für x&gt;1</gleichung>
</gleichungen>
```

10.1.5 CDATA-Sections

Manchmal gibt es große Textabschnitte in denen Zeichen vorkommen, die eigentlich durch character entities zu umschreiben wären, weil sie in XML eine reservierte Bedeutung haben.

XML bietet die Möglichkeit solche kompletten Abschnitte als eine sogenannte *CData Section* zu schreiben. Eine *CData section* beginnt mit der Zeichenfolge `<![CDATA[` und endet mit der Zeichenfolge: `]]>`. Dazwischen können beliebige Zeichenstehen, die eins zu eins als Text des Dokumentes interpretiert werden.

Beispiel: Die im vorherigen Beispiel mit Character Entities beschriebenen Formeln lassen sich innerhalb einer CDATA-Section wie folgt schreiben.

```
<?xml version="1.0"?>
<formeln><![CDATA[
  x+1>x
  x*x<x*x*x für x > 1
]]></formeln>
```

10.1.6 Processing Instructions

In einem XML-Dokument können Anweisung stehen, die angeben, was mit einem Dokument von einem externen Programm zu tun ist. Solche Anweisungen können z.B. angeben, mit welchen Mitteln das Dokument visualisiert werden soll. Wir werden hierzu im nächsten Kapitel ein Beispiel sehen. Syntaktisch beginnt eine *processing instruction* mit `<?` und endet mit `?>`. Dazwischen stehen wie in der Attributschreibweise Werte für den Typ der Anweisung und eine Referenz auf eine externe Quelle.

Beispiel: Ausschnitt aus dem XML-Dokument diesen Skripts, in dem auf ein Stylesheet verwiesen wird, daß das Skript in eine HTML-Darstellung umwandelt:

```
<?xml version="1.0"?>
<?xml-stylesheet
   type="text/xsl"
   href="../transformskript.xsl"?>

<skript>
<titelseite>
<titel>Grundlagen der Datenverarbeitung<white/>II</titel>
<semester>WS 02/03</semester>
</titelseite>
</skript>
```

10.1.7 Namensräume

Die *Tagnamen* sind zunächst einmal Schall und Rauch. Erst eine externes Programm wird diesen Namen eine gewisse Bedeutung zukommen lassen, indem es auf die *Tagnamen* in einer bestimmten Weise reagiert.

Da jeder Autor eines XML-Dokuments zunächst vollkommen frei in der Wahl seiner *Tagnamen ist*, wird es vorkommen, daß zwei Autoren denselben *Tagnamen* für die Markierung gewählt haben, aber semantisch mit diesem Element etwas anderes ausdrücken wollen. Spätestens dann,

wenn verschiedene Dokumente verknüpft werden, wäre es wichtig, daß *Tagnamen* einmalig mit einer Eindeutigen Bedeutung benutzt wurden. Hierzu gibt es in XML das Konzept der Namensräume.

Tagnamen können aus zwei Teilen bestehen, die durch einen Doppelpunkt getrennt werden:

- dem Präfix, der vor dem Doppelpunkt steht.
- dem lokalen Namen, der nach dem Doppelpunkt folgt.

Hiermit allein ist das eigentliche Problem gleicher *Tagnamen* noch nicht gelöst, weil ja zwei Autoren den gleichen Präfix und gleichen lokalen Namen für ihre Elemente gewählt haben können. Der Präfix wird aber an einem weiteren Text gebunden, der eindeutig ist. Dieses ist der eigentliche Namensraum. Damit garantiert ist, daß dieser Namensraum tatsächlich eindeutig ist, wählt man als Autor seine Webadresse, denn diese ist weltweit eindeutig.

Um mit Namensräume zu arbeiten ist also zunächst ein Präfix an eine Webadresse zu binden; dies geschieht durch ein Attribut der Art:
`xmlns:myPrefix="http://www.myAdress.org/myNamespace"`.

Beispiel: Ein Beispiel für ein XML-Dokument, daß den Präfix `sep` an einem bestimmten Namensraum gebunden hat:

```
<?xml version="1.0"?>
<sep:skript
   xmlns:sep="http://www.tfh-berlin.de/~panitz/dv2">
  <sep:titel>Grundlagen der DV 2</sep:titel>
  <sep:autor>Sven Eric Panitz</sep:autor>
</sep:skript>

```

Die Webadresse eines Namensraumes hat keine eigentliche Bedeutung im Sinne des Internets. Das Dokument geht nicht zu dieser Adresse und holt sich etwa Informationen von dort. Es ist lediglich dazu da, einen eindeutigen Namen zu haben. Streng genommen brauch es diese Adresse noch nicht einmal wirklich zu geben.

10.2 Codierungen

XML ist ein Dokumentenformat, das nicht auf eine Kultur mit einer bestimmten Schrift beschränkt ist, sondern in der Lage ist, alle im Unicode erfassten Zeichen darzustellen, seien es Zeichen der lateinischen, kyrillischen, arabischen, chinesischen oder sonst einer Schrift bis hin zur keltischen Keilschrift. Jedes Zeichen eines XML-Dokuments kann potentiell eines dieser mehrerern zigtausend Zeichen einer der vielen Schriften sein. In der Regel benutzt ein XML-Dokument insbesondere im amerikanischen und europäischen Bereich nur wenige kaum 100 unterschiedliche Zeichen. Auch ein arabisches Dokument wird mit weniger als 100 verschiedenen Zeichen auskommen.

Wenn ein Dokument im Computer auf der Festplatte gespeichert wird, so werden auf der Festplatte keine Zeichen einer Schrift, sondern Zahlen abgespeichert. Diese Zahlen sind traditionell

Zahlen die 8 Bit im Speicher belegen, ein sogenannter Byte (auch Oktett). Ein Byte ist in der Lage 256 unterschiedliche Zahlen darzustellen. Damit würde ein Byte ausreichen, alle Buchstaben eines normalen westlichen Dokuments in lateinischer Schrift (oder eines arabischen Dokuments darzustellen). Für ein Chinesisches Dokument reicht es nicht aus, die Zeichen durch ein Byte allein auszudrücken, denn es gibt mehr als 10000 verschiedene chinesische Zeichen. Es ist notwendig, zwei Byte im Speicher zu benutzen, um die vielen chinesischen Zeichen als Zahlen darzustellen.

Die *Codierung* eines Dokuments gibt nun an, wie die Zahlen, die der Computer auf der Festplatte gespeichert hat, als Zeichen interpretiert werden sollen. Eine Codierung für arabische Texte wird den Zahlen von 0 bis 255 bestimmte arabische Buchstaben zuordnen, eine Codierung für deutsche Dokumente wird den Zahlen 0 bis 255 lateinische Buchstaben inklusive deutscher Umlaute und dem `ß` zuordnen. Für ein chinesisches Dokument wird eine *Codierung* benötigt, die den 65536 mit 2 Byte darstellbaren Zahlen jeweils chinesische Zeichen zuordnet.

Man sieht, daß es Codierungen geben muß, die für ein Zeichen ein Byte im Speicher belegen, und solche, die zwei Byte im Speicher belegen. Es gibt darüberhinaus auch eine Reihe Mischformen, manche Zeichen werden durch ein Byte andere durch 2 oder sogar durch 3 Byte dargestellt.

Im Kopf eines XML-Dokuments kann angegeben werden, in welcher Codierung das Dokument abgespeichert ist.

Beispiel: Dieses Skript ist in einer Codierung gespeichert, die für westeuropäische Dokumente gut geeignet ist, da es für die verschiedenen Sonderzeichen der westeuropäischen Schriften einen Zahlenwert im 8-Bit-Bereich zugeordnet hat. Die Codierung mit dem Namen: `iso-8859-1`. Diese wird im Kopf des Dokuments angegeben:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<skript><kapitel>blablabla</kapitel></skript>
```

Wird keine Codierung im Kopf eines Dokuments angegeben, so wird als Standardcodierung die sogenannte `utf-8` Codierung benutzt. In ihr belegen lateinische Zeichen einen Byte und Zeichen anderer Schriften (oder auch das Euro Symbol) zwei bis drei Bytes.

Eine Codierung, in der alle Zeichen mindestens mit zwei Bytes dargestellt werden ist: `utf-16`, die Standardabbildung von Zeichen, wie sie im *Unicode* definiert ist.

10.3 Dokumente als Bäume in Java

Wie wir festgestellt haben, sind XML-Dokumente mit ihrer hierarchischen Struktur Bäume. In den Vorgängerkapitel haben wir uns schon auf unterschiedliche Weise mit Bäumen beschäftigt. Von den dabei gemachten Erfahrungen können wir jetzt profitieren.

10.3.1 APIs für XML

Wir wollen jetzt einen Blick auf die gängigen Schnittstellen für die XML-Programmierung werfen.

DOM

Die allgemeine Schnittstellenbeschreibung für XML als Baumstruktur ist das *distributed object modell* kurz *dom*[Arn04], für das das W3C eine Empfehlung herausgibt. Die Ursprünge von *dom* liegen nicht in der XML-Programmierung sondern in der Verarbeitung von HTML-Strukturen im Webbrowser über Javascript. Mit dem Auftreten von XML entwickelte sich der Wunsch, eine allgemeine, platform- und sprachunabhänge Schnittstellenbeschreibung für XML- und HTML-Baumstrukturen zu bekommen, die in verschiedenen Sprachen umgesetzt werden kann.

Da es sich um ein implementierungsunabhängiges API handelt, finden wir *dom* in der Javabibliothek nur als Schnittstellen.

Wichtige Schnittstellen Die zentrale Schnittstelle in *dom* ist `Node`. Sie hat als Unterschnittstellen alle Knotentypen, die es in XML gibt. Folgende Graphik gibt über diese Knotentypen einen Überblick.

```
interface org.w3c.dom.Node
 |
 |--interface org.w3c.dom.Attr
 |--interface org.w3c.dom.CharacterData
 |   |
 |   |--interface org.w3c.dom.Comment
 |   |--interface org.w3c.dom.Text
 |       |
 |       |--interface org.w3c.dom.CDATASection
 |
 |--interface org.w3c.dom.Document
 |--interface org.w3c.dom.DocumentFragment
 |--interface org.w3c.dom.DocumentType
 |--interface org.w3c.dom.Element
 |--interface org.w3c.dom.Entity
 |--interface org.w3c.dom.EntityReference
 |--interface org.w3c.dom.Notation
 |--interface org.w3c.dom.ProcessingInstruction
```

Eine der entscheidenen Methoden der Schnittstelle `Node` selektiert die Liste der Kinder eines Knotens:
`public NodeList getChildNodes()`

Knoten, die keine Kinder haben können (Textknoten, Attribute etc.) geben bei dieser Methode die leere Liste zurück. Attribute zählen auch wie in unserer Modellierung nicht zu den Kindern eines Knotens. Um an die Attribute zu gelangen, gibt es eine eigene Methode:
`NamedNodeMap getAttributes()`

Wie man sieht, benutzt Javas *dom* Umsetzung keine von Javas Listenklassen zur Umsetzung einer Knotenliste, sondern nur genau die in *dom* spezifizierte Schnittstelle `NodeList`. Eine `NodeList` hat genau zwei Methoden:

```
int getLength()
Node item(int index)
```

Dieses ist insofern schade, da somit nicht die neue *for*-Schleife aus Java 1.5 für die Knotenliste des *dom* benutzt werden kann.

Einen Parser für XML Wir benötigen einen Parser, der uns die Baumstruktur eines XML-Dokuments erzeugt. In der Javabibliothek ist ein solcher Parser integriert, allerdings nur über seine Schnittstellenbeschreibung. Im Paket `javax.xml.parsers` gibt es nur Schnittstellen. Um einen konkreten Parser zu erlangen, bedient man sich einer Fabrikmethode: In der Schnittstelle `DocumentBuilderFactory` gibt es eine statische Methode `newInstance` und über das `DocumentBuilderFactory`-Objekt, läßt sich mit der Methode `newDocumentBuilder` ein Parser erzeugen.

Beispiel:
Wir können so eine statischen Methode zum Parsen eines XML-Dokuments schreiben:

ParseXML.java

```
package name.panitz.domtest;

import org.w3c.dom.Document;
import javax.xml.parsers.*;
import java.io.File;

public class ParseXML {
  public static Document parseXml(String xmlFileName){
    try{
     return
      DocumentBuilderFactory
       .newInstance()
       .newDocumentBuilder()
       .parse(new File(xmlFileName));
    }catch(Exception _){return null;}
  }

  public static void main(String [] args){
   System.out.println(parseXml(args[0]));
  }
}
```

Wir können jetzt z.B. den Quelltext dieses Skripts parsen.

```
sep@linux:~/fh/prog4/examples> java -classpath classes/ name.panitz.domtest.ParseXML ../skript.xml
[#document: null]
```

Wie man sieht ist die Methode `toString` in der implementierenden Klasse der Schnittstelle `Document`, die unser Parser benutzt nicht sehr aufschlußreich.

Beispielalgorithmen auf DOM Wir können versuchen für das *dom*-Objekten ähnliche Methoden zu schreiben, wie für die Bäume im vorangegangenen Kapitel.

Zunächst zählen wir wieder alle Knoten im Dokument:

CountNodes.java

```
package name.panitz.domtest;

import org.w3c.dom.Node;

public class CountNodes{
  static int count(Node node){
    int result = 1;
    for (Node n:new NoteListIterator(node.getChildNodes()))
      result=result+count(n);
    return result;
  }
  public static void main(String [] args){
    System.out.println(count(ParseXML.parseXml(args[0])));
  }
}
```

Hierbei haben wir in der **for**-Schleife für Objekte die die Schnittstelle `NodeList` implementieren einen Wrapper benutzt, der diese Objekte zu einem Iteratorobjekt verpackt.

NoteListIterator.java

```
package name.panitz.domtest;

import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import java.util.Iterator;

public class NoteListIterator implements Iterator<Node>
                                       , Iterable<Node>{
  NodeList nodes;
  int current=-1;
  public NoteListIterator(NodeList n){nodes=n;}

  public Node next(){
    current=current+1; return nodes.item(current);}

  public boolean hasNext(){return current+1<nodes.getLength();}

  public void remove(){
    throw new UnsupportedOperationException();
  }

  public Iterator<Node> iterator(){return this;}
}
```

Wir können uns zum Beispiel die Anzahl der Knoten in diesem Skript ausgeben lassen:

```
sep@linux:~/fh/prog4> java -classpath classes/ name.panitz.domtest.CountNodes skript.xml
1316
```

Als einen weiteren Algorithmus können wir wieder die maximale Pfadlänge berechnen lassen:

DomDepth.java

```
package name.panitz.domtest;

import org.w3c.dom.Node;

public class DomDepth{
  static int depth(Node node){
    int result = 0;
    for (Node n:new NoteListIterator(node.getChildNodes())){
      final int currentDepth = depth(n);
      if (result<currentDepth) result=currentDepth;
    }
    return result+1;
  }
  public static void main(String [] args){
    System.out.println(depth(ParseXML.parseXml(args[0])));
  }
}
```

Auch dieses läßt sich wunderbar mit dem Quelltext dieses Skriptes testen.

```
sep@linux:~/fh/prog4> java -classpath classes/ name.panitz.domtest.DomDepth skript.xml
13
```

Manuelles Manipulieren von Dom Das DOM Api ermöglicht nicht nur in einem XML-Baum beliebig zu navigieren, sondern auch diesen Baum zu manipulieren. Es lassen sich neue Knoten einhängen und bestehende Knoten löschen. Hierzu stehen in der Schnittstelle `Node` entsprechende Methoden zur Verfügung:

```
Node appendChild(Node newChild)      throws DOMException;
Node insertBefore(Node newChild,Node refChild)
                                      throws DOMException;
Node replaceChild(Node newChild,Node oldChild)
                                      throws DOMException;
Node removeChild (Node oldChild)      throws DOMException;
```

Speziellere Methoden zum Manipulieren der verschieden Baumknoten finden sich in den Unterschnittstellen von `Node`.

Zum Erzeugen eines neuen Knotens ist es notwendig den Dokumentknoten des zu manipulierenden Knotens zu kennen. Der Dokumentknoten eines Knotens läßt sich über die Methode `Document getOwnerDocument()` erfragen. Hier gibt es dann Methoden zur Erzeugung neuer Knoten:

```
Attr createAttribute(String name);
Attr createAttributeNS(String namespaceURI, String qualifiedName);
CDATASection createCDATASection(String data);
```

```
Comment createComment(String data);
DocumentFragment createDocumentFragment();
Element createElement(String tagName);
Element createElementNS
                 (String namespaceURI, String qualifiedName);
EntityReference createEntityReference(String name);
ProcessingInstruction createProcessingInstruction
                                 (String target, String data);
Text createTextNode(String data);
```

SAX

Oft brauchen wir nie das komplette XML-Dokument als Baum im Speicher. Eine Großzahl der Anwendungen auf XML-Dokumenten geht einmal das Dokument durch, um irgendwelche Informationen darin zu finden, oder ein Ergebnis zu erzeugen. Hierzu reicht es aus, immer nur einen kleinen Teil des Dokuments zu betrachten. Und tatsächlich hätte diese Vorgehensweise, bai allen bisher geschriebenen Programmen gereicht. Wir sind nie im Baum hin und her gegangen. Wir sind nie von einem Knoten zu seinem Elternknoten oder seinen vor ihm liegenden Geschwistern gegangen.

Ausgehend von dieser Beobachtuung hat eine Gruppe von Programmierern ein API zur Bearbeitungen von XML-Dokumenten vorgeschlagen, das nie das gesammte Dokument im Speicher zu halten braucht. Dieses API heißt *SAX*, für *simple api for xml processing*. *SAX* ist keine Empfehlung des W3C. Es ist außerhalb des W3C entstanden.

Die Idee von *SAX* ist ungefähr die, daß uns jemand das Dokument vorliest, einmal von Anfang bis Ende. Wir können dann auf das gehörte reagieren. Hierzu ist für einen Parse mit einem SAX-Parser stets mit anzugeben, wie auf das Vorgelesene reagiert werden soll. Dieses ist ein Objekt der Klasse `DefaultHandler`. In einem solchen *handler* sind Methoden auszuprogrammieren, in denen spezifiziert ist, was gemacht werden soll, wenn ein Elementstarttag, Elementendtag, Textknoten etc. vorgelsen wird. Man spricht bei einem SAX-Parser von einem ereignisbasierten Parser. Wir reagieren auf bestimmte Ereignisse des Parses, nämlich dem Starten/Enden von Elementen und so weiter.

Instanziieren eines SAX-Parsers Auch ein SAX-Parser liegt in Java nur als Schnittstelle vor und kann nur über eine statische Fabrikmethode instanziiert werden.

SaxParse.java

```
package name.panitz.saxtest;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;
import org.xml.sax.*;
import java.io.File;

public class SaxParse{
  public static void parse(File file,DefaultHandler handler)
                                 throws Exception{
    SAXParserFactory.newInstance()
                    .newSAXParser()
                    .parse(file,handler);
```

```
  }
}
```

Zählen von Knoten Als erstes Beispiel wollen wir unser altbekanntes Zählen der Knoten programmieren. Hierzu ist ein eigener `DefaultHandler` zu schreiben, der, sobald beim Vorlesen ihm der Beginn eines Elements gemeldet wird, darauf reagiert, indem er seinen Zähler um eins weiterzählt. Wir überschreiben demnach genau eine Methode aus dem `DefaultHandler`, nämlich die Methode `startElement`:

SaxCountNodes.java

```
package name.panitz.saxtest;
import org.xml.sax.helpers.DefaultHandler;
import java.util.*;
import org.xml.sax.*;

public class SaxCountNodes extends DefaultHandler{
  public int result = 0;
  public void startElement
   (String uri, String localName
   , String qName, Attributes attributes)
                          throws SAXException {
    result=result+1;
  }

  public static void main(String [] args) throws Exception{
    SaxCountNodes counter = new SaxCountNodes();
    SaxParse.parse(new java.io.File(args[0]),counter);
    System.out.println(counter.result);
  }
}
```

Selektion von Code-Knoten In einem nächsten Beispiel für einen Handler, schreiben wir einen Handler, der bestimmte Knoten selektiert und in einer Ergebnisliste sammelt.

Wir wollen die `code` Knoten aus diesem Skript selektieren.

Hierzu können wir als algebraischen Datentypen einfach eine Klasse vorsehen, die ein Codefragment aus dem Skript darstellt. Dieses hat einen Programmnamen, ein Paketnamen und schließlich den darin enthaltenen Code.

CodeFragment.adt

```
package name.panitz.saxtest;
data class CodeFragment {
  CodeFragment(String progName,String packageName,String code);
}
```

Der entsprechende Handler sammelt die benötigte Information auf.

SelectNodes.java

```
package name.panitz.saxtest;
import org.xml.sax.helpers.DefaultHandler;
```

```
import java.util.*;
import org.xml.sax.*;

public class SelectNodes extends DefaultHandler{
  final List<String> names;
  final List<CodeFragment> result = new ArrayList<CodeFragment>();

  public SelectNodes(List<String> ls){names=ls;}
  public SelectNodes(String ls){
    names=new ArrayList<String>(); names.add(ls);}

  private StringBuffer currentCode = new StringBuffer();
  private String currentProgName = "";
  private String currentPackageName = "";

  public void startElement
   (String uri, String localName, String qName, Attributes attributes)
                throws SAXException {
     if (names.contains(qName)){
       currentCode = new StringBuffer();
       currentProgName = attributes.getValue("class");
       currentPackageName = attributes.getValue("package");
     }
  }

  public void endElement
    (String uri, String localName, String qName)
                throws SAXException {
     if (names.contains(qName)){
       result.add(
         new CodeFragment
            (currentProgName
            ,currentPackageName
            ,currentCode.toString()));
     }
  }

  public void characters(char[] ch,int start,int length)
                throws SAXException {
    currentCode.append(ch,start,length);
  }
}
```

Den obigen Handler können wir jetzt z.B. benutzen, um aus dem Quelltext dieses Skriptes bestimmte Beispielklassen zu extrahieren und in eine Javadatei zu speichern.

GetCode.java

```
package name.panitz.saxtest;

import javax.xml.parsers.*;
```

```
import java.io.*;

public class GetCode {
  public static void main(String [] args) throws Exception{
    final  SelectNodes selectCodeHandler = new SelectNodes("code");

    SaxParse.parse(new File(args[0]),selectCodeHandler);
    final Writer out = new FileWriter(args[1]+".java");

    for (CodeFragment cf:selectCodeHandler.result){
      if (args[1].equals(cf.getProgName()))
        out.write(cf.getCode());
    }
    out.flush();
    out.close();
  }
}
```

Wie man insgesamt sehen kann, ist das Vorgehen zum Schreiben eines *handlers* in SAX ähnlich zum Schreiben eines Besuchers unseres algebraischen Typens für XML-Dokumente.

10.4 Transformationen und Queries

Dieses Kapitel beschäftigt sich mit Sprachen, die darauf zugeschnitten sind, XML-Dokumente zu verarbeiten.

10.4.1 XPath: Pfade in Dokumenten

Wir haben XML-Dokumente als Bäume betrachtet. Eine der fundamentalen Konzepten in einem Baum, sind die Pfade innerhalb eines Baumes. So kann zum Beispiel jeder Knoten eines Baumes eindeutig mit einem Pfad beschrieben werden. Zum Beschreiben von Pfaden innerhalb eines XML existiert eine Sprache, die als Empfehlung des W3C vorliegt, XPath[CD99]. Eine sehr schöne formale Beschreibung einer denotationalen Semantik für XPath findet sich in [Wad00].

XPath lehnt sich syntaktisch an eine Beschreibungssprache für Pfade in Bäumen an, die sehr verbreitet ist, nämlich Pfadbeschreibungen im Dateibaum. Um eine Datei in einem Dateisystem zu adressieren geben wir einen Pfad von der Wurzel des Dateibaums bis zu dieser Datei an. Die Datei dieses Skriptes findet sich z.B. in folgenden Pfad auf meinem Rechner:
`/home/sep/fh/prog4/skript.xml`

Ein Schrägstrich bedeutet in dieser Syntax, es wird nach diesem Strich sich auf die Kinder des vor dem Strich spezifizierten Knotens bezogen. Ist vor dem Strich nichts spezifiziert, so beginnen wir mit der Wurzel des Baums. Nach einem Schrägstrich kann man die gewünschten Kinderknoten selektieren. Bei der Pfadangabe eines Dateibaums, spezifizieren wir die Knoten über die Datei- und Ordnernamen.

XPath greift diese Idee auf. Hier bezeichnen die Namen die Tagnamen der Elemente. Im vorliegenden Dokument läßt sich so der folgende Pfad beschreiben:
`/skript/kapitel/section/subsection/code`.

Damit sind alle Elementknoten dieses Dokuments gemeint, deren Tagname `code`, die ein Elternknoten `subsection` haben, deren Elternlnoten den Tagnamen `section` haben, deren Eltern `kapitel`, deren Eltern das Wurzelelement `skript` ist.

Wie man sieht, bezeichnen XPath-Ausdrücke in der Regel Mengen von Dokumentknoten.

XPath Pfade lassen sich in zwei Weisen benutzen:

- Zum Selektieren bestimmter Knoten in einem Dokument. Ausgehend von einem Wurzelknoten werden anhand des Ausdrucks aus dem Baum bestimmte Knoten selektiert.
- Zum Prüfen, ob ein Knoten die in dem Pfad beschriebene Eigenschaft hat. Ausgehend von einem Knoten wird anhand des Ausdrucks getestet, ob er diese beschriebene Pfadeigenschaft hat.

Beide Arten, XPath-Ausdrücke zu verwenden sind in der Praxis üblich. Beispiele hierfür werden wir in den nächsten Abschnitten sehen.

Um XPath möglichst gut kennenzulernen, wird im folgenden ein kleiner Prozessor implementiert, der für einen XPath-Ausdruck die Menge der von ihm beschriebenen Knoten ausgehend von einem Knoten selektiert.

Achsen

Eines der wichtigsten Konzepten von XPath sind die sogenannten Achsen. Sie beschreiben bestimmte Arten sich in einem Baum zu bewegen. XPath kennt 13 Achsen. Die am häufigste verwendete Achse ist, die Kinderachse. Sie beschreibt die Menge aller Kinder eines Knotens. Entsprechend gibt es die Elternachse, die den Elternknoten beschreibt. Eine sehr simple Achse beschreibt den Knoten selbst. Achsen für Vorfahren und Nachkommen sind der transitive Abschluß der Eltern- bzw. Kinderachse. Zusätzlich gibt es noch Geschwisterachsen, eine für die Geschwister, die vor dem aktuellen Knoten stehen und einmal für die Geschwister, die nach dem aktuellen Knoten stehen. Eigene Achsen stehen für Attribute und Namensraumdeklarationen zur Verfügung.

Mit den seit Java 1.5 zur Verfügung stehenden Aufzählungstypen, läßt sich in Java einfach ein Typ der alle 13 Achsen beschreibt implementieren.

AxisType.java

```
1  package name.panitz.xml.xpath;
2  public enum AxisType
3    {self
4    ,child
5    ,descendant
6    ,descendant_or_self
7    ,parent
8    ,ancestor
9    ,ancestor_or_self
10   ,following_sibling
11   ,following
12   ,preceding_sibling
13   ,preceding
14   ,namespace
```

```
  ,attribute
}
```

Für jede dieser Achsen geben wir eine Implementierung auf DOM. Eine Achsenimplementierung entspricht dabei einer Funktion, die für einen Knoten eine Liste von Knoten zurückgibt.

Achsenhilfsklasse Wir schreiben eine Klasse mit statischen Methoden zur Berechnung der Achsen. Da in DOM nicht die Listen aus `java.util` benutzt werden sondern die Schnittstelle `org.w3c.dom.Nodelist`, schreiben wir zunächst eine kleine Methode, die eine `NodeList` in eine `java.util.List<Node>` umwandelt.

Axes.java

```
package name.panitz.xml.xpath;
import name.panitz.crempel.util.FromTo;
import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;
import name.panitz.domtest.*;

public class Axes{
  public static List<Node> nodelistToList (NodeList nl){
    List<Node> result = new ArrayList<Node>();
    for (int i:new FromTo(0,nl.getLength()-1)){
      result.add(nl.item(i));
    }
    return result;
  }
```

self Die einfachste Achse liefert genau den Knoten selbst. Die entsprechende Methode liefert die einelementige Liste des Eingabeknotens:

Axes.java

```
  public static List<Node> self(Node n){
    List<Node> result = new ArrayList<Node>();
    result.add(n);
    return result;
  }
```

child Die gebräuchlichste Achse beschreibt die Menge aller Kinderknoten. Wir lassen uns die Kinder des DOM Knotens geben und konvertieren die `NodeList` zu einer Javaliste:

Axes.java

```
  public static List<Node> child(Node n){
    return nodelistToList(n.getChildNodes());
  }
```

descendant Die Nachkommen sind Kinder und deren Nachkommen. Für fügen jedes Kind zur Ergebnisliste hinzu, sowie alle deren Nachkommen:

Axes.java

```
24  public static List<Node> descendant(Node n){
25    List<Node> result = new ArrayList<Node>();
26    for (Node child:child(n)){
27      result.add(child);
28      result.addAll(descendant(child));
29    }
30    return result;
31  }
```

descendant-or-self In der Nachkommenachse taucht der Knoten selbst nicht auf. Diese Achse fügt den Knoten selbst zusätzlich ans Ergebnis an:

Axes.java

```
32  public static List<Node> descendant_or_self(Node n){
33    List<Node> result = self(n);
34    result.addAll(descendant(n));
35    return result;
36  }
```

parent Die Kinderachse lief in den Baum Richtung Blätter eine Ebene nach unten, die Elternachse läuft diese eine Ebene Richtung Wurzel nach oben:

Axes.java

```
37  public static List<Node> parent(Node n){
38     List<Node> result = new ArrayList<Node>();
39     result.add(n.getParentNode());
40     return result;
41  }
```

ancestor Die Vorfahrenachse ist der transitive Abschluß über die Elternachse, sie bezeichnet also die Eltern und deren Vorfahren:

Axes.java

```
42  public static List<Node> ancestor(Node n){
43    List<Node> result = new ArrayList<Node>();
44    for (Node parent:parent(n)){
45      if (parent!=null){
46        result.addAll(ancestor(parent));
47        result.add(parent);
48      }
49    }
50    return result;
51  }
```

ancestor-or-self Die Vorfahrenachse enthält nicht den Knoten selbst. Entsprechend wie auf der Nachkommenachse gibt es auch hier eine Version, die den Knoten selbst enthält:

Axes.java
```
52  public static List<Node> ancestor_or_self(Node n){
53    List<Node> result = ancestor(n);
54    result.addAll(self(n));
55    return result;
56  }
```

following-sibling Diese Achse beschreibt die nächsten Geschwister. Wir erhalten diese, indem wir den Elternknoten holen, durch dessen Kinder iterieren und nachdem wir an unseren aktuellen Knoten gelangt sind, anfangen diese Kinderknoten ins Ergebnis aufzunehmen.

Axes.java
```
57  public static List<Node> following_sibling(Node n){
58    List<Node> result = new ArrayList<Node>();
59    int nodeType = n.getNodeType();
60    boolean take = false;
61    if (nodeType != Node.ATTRIBUTE_NODE){
62      for (Node sibling:child(n.getParentNode())){
63        if (take) result.add(sibling);
64        if (sibling==n) take=true;
65      }
66    }
67    return result;
68  }
```

preceding-sibling Entsprechend gibt es die Achse, die die vor dem aktuellen Knoten stehenden Geschwister extrahieren. Wir gehen ähnlich vor wie oben, sammeln jetzt jedoch die Kinder nur bis zum aktuellen Knoten auf.

Axes.java
```
69  public static List<Node> preceding_sibling(Node n){
70    List<Node> result = new ArrayList<Node>();
71    int nodeType = n.getNodeType();
72    if (nodeType != Node.ATTRIBUTE_NODE){
73      for (Node sibling:child(n.getParentNode())){
74        if (sibling==n) break;
75        result.add(sibling);
76      }
77    }
78    return result;
79  }
```

following Die Nachfolgerachse bezieht sich auf die Dokumentordnung in serialisierter Form. Sie bezeichnet alle Knoten, deren Tag im gedruckten XML Dokument nach dem Ende des aktuellen Knotens beginnen. Dieses sind die nachfolgenden Geschwister und deren Nachkommen:

Axes.java

```
80  public static List<Node> following(Node n){
81    List<Node> result = new ArrayList<Node>();
82    for (Node follow_sib:following_sibling(n)){
83      result.add(follow_sib);
84      result.addAll(descendant(follow_sib));
85    }
86    return result;
87  }
```

preceding Analog hierzu funktioniert die Vorgängerachse. Auch sie bezieht sich auf die Dokumentordnung:

Axes.java

```
88  public static List<Node> preceding(Node n){
89    List<Node> result = new ArrayList<Node>();
90    for (Node preced_sib:preceding_sibling(n)){
91      result.add(preced_sib);
92      result.addAll(descendant(preced_sib));
93    }
94    return result;
95  }
```

attribute Attribute werden in einer gesonderten Achse beschrieben. Sie tauchen in keiner der vorherigen Achsen auf. Ausgenommen sind in dieser Achse die Attribute die eine Namensraumdefinition beschreiben:

Axes.java

```
96  public static List<Node> attribute(Node n){
97    List<Node> result = new ArrayList<Node>();
98    if (n.getNodeType()==Node.ELEMENT_NODE){
99      NamedNodeMap nnm = n.getAttributes();
100     for (int i:new FromTo(0,nnm.getLength()-1)){
101       Node current = nnm.item(i);
102       if (!current.getNodeName().startsWith("xmlns"))
103         result.add(current);
104     }
105   }
106   return result;
107 }
```

namespace Ebenso gesondert werden in der letzten Achse die Namensraumdefinitionen behandelt. XML-technisch sind diese Attribute, deren Attributname mit `xmlns` beginnt:

Axes.java

```
108 public static List<Node> namespace(Node n){
109   List<Node> result = new ArrayList<Node>();
110   if (n.getNodeType()==Node.ELEMENT_NODE){
111     NamedNodeMap nnm = n.getAttributes();
```

```
      for (int i:new FromTo(0,nnm.getLength()-1)){
        Node current = nnm.item(i);
        if (current.getNodeName().startsWith("xmlns"))
          result.add(current);
      }
    }
    return result;
  }
```

Achsenberechnung Für Aufzählungstypen ab Java 1.5 läßt sich eine schöne `switch`-Anweisung schreiben. So können wir eine allgemeine Methode zur Achsenberechnung schreiben, die für jede Achse die entsprechende Methode aufruft.[2]

Axes.java

```
  public static List<Node> getAxis(Node n,AxisType axis){
     switch (axis){
      case ancestor          : return ancestor(n);
      case ancestor_or_self  : return ancestor_or_self(n);
      case attribute         : return attribute(n);
      case child             : return child(n);
      case descendant        : return descendant(n);
      case descendant_or_self: return descendant_or_self(n);
      case following         : return following(n);
      case following_sibling : return following_sibling(n);
      case namespace         : return namespace(n);
      case parent            : return parent(n);
      case preceding         : return preceding(n);
      case preceding_sibling : return preceding_sibling(n);
      case self              : return self(n);
      default                :throw new UnsupportedOperationException();
    }
  }
}
```

Knotentest

Die Kernausdrücke in XPath sind von der Form:

$$axisType::nodeTest$$

axisType beschreibt dabei eine der 13 Achsen. *nodeTest* ermöglicht es, aus den durch die Achse beschriebenen Knoten bestimmte Knoten zu selektieren. Syntaktisch gibt es 8 Arten des Knotentests:

- `*`: beschreibt Elementknoten mit beliebigen Namen.

[2] Die Version Beta1 von Java 1.5 hat einen Bug, so daß der javac Commpiler bei der folgenden Klasse während der Codegenerierung abstürzt. Daher ist der Code vorerst auskommentiert.

- *pref*:`*`: beschreibt Elementknoten mit dem Prefix *pref* und beliebigen weiteren Namen.
- *pref*:*name*: beschreibt Elementknoten mit dem Prefix *pref* und den weiteren Namen *name*. Der Teil vor den Namen kann dabei auch fehlen.
- `comment()`: beschreibt Kommentarknoten.
- `text()`: beschreibt Textknoten.
- `processing-instruction()`: beschreibt Processing-Instruction-Knoten.
- `processing-instruction(`*target*`)`: beschreibt Processing-Instruction-Knoten mit einem bestimmten Ziel.
- `node()`: beschreibt beliebige Knoten.

Algebraischer Typ für Knotentests Wir können die acht verschiedene Knotentests durch einen algebraischen Typ ausdrücken:

NodeTest.java

```
package name.panitz.xml.xpath;
import java.util.List;
import name.panitz.adt.*;

@Data abstract class NodeTest{
  @Constr void StarTest(){};
  @Constr void PrefixStar(String prefix){};
  @Constr void QName(String prefix,String name){};
  @Constr void IsComment(){};
  @Constr void IsText(){};
  @Constr void IsProcessingInstruction(){};
  @Constr void IsNamedProcessingInstruction(String name){};
  @Constr void IsNode(){};

  abstract public <b_> b_ welcome(NodeTestVisitor<b_> visitor);
}
```

Textuelle Darstellung für Knotentests Für diesen algebraischen Typ läßt sich ein einfacher Besucher zur textuellen Darstellung des Typs schreiben. Er erzeugt für die einzelnen Knotentests die Syntax wie sie in XPath-Ausdrücken vorkommt.

ShowNodeTest.java

```
package name.panitz.xml.xpath;

public class ShowNodeTest extends NodeTestVisitor<String> {
  public String visit(StarTest _){return "*";}
  public String visit(PrefixStar e){return e.prefix+":*";}
  public String visit(QName e){String result="";
    if (e.prefix.length()>0) result=e.prefix+":";
    result=result+e.name;
    return result;
```

```
  }
  public String visit(IsComment _){return "comment()";}
  public String visit(IsText _){return "text()";}
  public String visit(IsProcessingInstruction _){
    return "processing-instruction()";}
  public String visit(IsNamedProcessingInstruction e){
    return "processing-instruction("+e.name+")";
  }
  public String visit(IsNode _){return "node()";}
}
```

Auswerten von Knotentests Ein Knotentest ergibt für einen konkreten Knoten entweder `true` oder `false`. Hierfür können wir einen Besucher schreiben, der für einen Knotentest und einen konkreten Knoten diesen bool'schen Wert berechnet.

DoNodeTest.java

```
package name.panitz.xml.xpath;

import java.util.List;
import java.util.ArrayList;
import static org.w3c.dom.Node.*;
import org.w3c.dom.*;

public class DoNodeTest extends NodeTestVisitor<Boolean> {
  Node current;
  public DoNodeTest(Node n){current=n;}
```

Der Test auf einen Stern ist wahr für jedes Element oder Attribut:

DoNodeTest.java

```
  public Boolean visit(StarTest _){
    return    current.getNodeType()==ELEMENT_NODE
            || current.getNodeType()==ATTRIBUTE_NODE;
  }
```

Der Test auf ein bestimmtes Prefix ist wahr für jeden Knoten, der diesen Prefix hat:

DoNodeTest.java

```
  public Boolean visit(PrefixStar e){
    String currentPrefix = current.getPrefix();
    if (currentPrefix==null) currentPrefix="";
    return   new StarTest().welcome(this)
         && currentPrefix.equals(e.prefix);
  }
```

Jeder Test nach einen qualifizierten Namen ist wahr, wenn der Knoten diesen Namen hat:

DoNodeTest.java

```
21  public Boolean visit(QName e){
22    return   new PrefixStar(e.prefix).welcome(this)
23         &&  current.getNodeName().equals(e.name);
24  }
```

Der Test nach Kommentaren ist für Kommantarknoten wahr:

DoNodeTest.java

```
25  public Boolean visit(IsComment _){
26    return current.getNodeType()==COMMENT_NODE;}
```

Der Test nach Text ist für Textknoten wahr:

DoNodeTest.java

```
27  public Boolean visit(IsText _){
28    return current.getNodeType()==TEXT_NODE;}
```

Der Test nach Processing-Instruction ist für PI-Knoten wahr:

DoNodeTest.java

```
29  public Boolean visit(IsProcessingInstruction _){
30    return current.getNodeType()==PROCESSING_INSTRUCTION_NODE;}
```

Der Test nach Processing-Instruction mit bestimmten Namen ist für PI-Knoten mit diesen Namen wahr:

DoNodeTest.java

```
31  public Boolean visit(IsNamedProcessingInstruction e){
32    return current.getNodeType()==PROCESSING_INSTRUCTION_NODE
33        && e.name.equals(current.getNodeName());
34  }
```

Der Test nach Knoten ist immer wahr:

DoNodeTest.java

```
35  public Boolean visit(IsNode _){return true;}
```

Einen großen Teil eines XPath-Prozessors haben wir damit schon implmentiert. Wir können uns die Knoten einer Achse geben lassen und wir können diese Knoten auf einen Knotentest hin prüfen. Zusammen läßt sich damit bereits eine Methode zu Auswertung eines Kernausdrucks mit Achse und Knotentest schreiben. Hierzu iterieren wir über die Liste der durch die Achse spezifizierten Knoten und fügen diese bei positiven Knotentest dem Ergebnis zu:

DoNodeTest.java

```
36  static public List<Node> evalAxisExpr
37        (AxisType axis,NodeTest test1,Node context){
38    NodeTestAdt test = (NodeTestAdt)test1;
39    List<Node> result = new ArrayList<Node>();
40    for (Node node:Axes.getAxis(context,axis)){
```

```
41        if (test.welcome(new DoNodeTest(node)))
42          result.add(node);
43      }
44      return result;
45    }
46  }
```

Pfadangaben

In obiger Einführung haben wir bereits gesehen, daß XPath den aus dem Dateissystem bekannten Schrägstrich für Pfadangaben benutzt. Betrachten wir XPath-Ausdrücke als Terme, so stellt der Schrägstrich einen Operator dar. Diesen Operator gibt es sowohl einstellig wie auch zweistellig.

Die Grundsyntax in XPath sind eine durch Schrägstrich getrennte Folge von Kernausdrücke mit Achsentyp und Knotentest.

> **Beispiel:**
> Der Ausdruck
>
> `child::skript/child::*/descendant::node()/self::code/attribute::class`
>
> beschreibt die Attribute mit Attributnamen `class`, die an einem Elementknoten mit Tagnamen `code` hängen, die Nachkommen eines beliebigen Elementknotens sind, die Kind eines Elementknotens mit Tagnamen `skript` sind, die Kind des aktuellen Knotens, auf dem der Ausdruck angewendet werden soll sind.
>
> Vorwärts gelesen ist dieser Ausdruck eine Selektionsanweisung:
> Nehme alle `skript`-Kinder des aktuellen Knotens. Nehme von diesen beliebige Kinder. Nehme von diesen alle `code`-Nachkommen. Und nehme von diesen jeweils alle `class`-Attribute.

Der einstellige Schrägstrichoperator bezieht sich auf die Dokumentwurzel des aktuellen Knotens.

Abkürzende Pfadangaben

XPath kennt einen zweiten Pfadoperator `//`. Auch er existiert jeweils einmal einstellig und einmal zweistellig. Der doppelte Schrägstrich ist eine abkürzende Schreibweise, die übersetzt werden kann in Pfade mit einfachen Schrägstrichoperator.

//*expr*	Betrachte beliebige Knoten unterhalt des Dokumentknotens, die durch *expr* charakterisiert werden.
$e_1//e_2$	Betrachte beliebiege Knoten unterhalb der durch e_1 charkterisierten Knoten.und prüfe diese auf e_2.

Übersetzung in Kernsyntax Der Doppelschrägstrich ist eine abkürzende Schreibweise für: `/descendant-or-self::node()/`

Weitere abkürzende Schreibweisen

In XPath gibt es weitere abkürzende Schreibweisen, die auf die Kernsyntax abgebildet werden können.

der einfache Punkt Für die Selbstachse kann als abkürzende Schreibweise ein einfacher Punkt `.` gewählt werden, wie er aus den Pfadangaben im Dateisystem bekannt ist.

Der Punkt `.` ist die abkürzende Schreibweise für: `self::node()`.

der doppelte Punkt Für die Elternachse kann als abkürzende Schreibweise ein doppelter Punkt `..` gewählt werden, wie er aus den Pfadangaben im Dateisystem bekannt ist.

Der Punkt `..` ist die abkürzende Schreibweise für: `parent::node()`.

implizite Kinderachse Ein Kernausdruck, der Form `child::`*nodeTest* kann abgekürzt werden durch *nodeTest*. Die Kinderachse ist also der Standardfall.

Attributselektion Auch für die Attributachse gibt es eine abkürzende Schreibweise: ein Kernausdruck, der Form `attribute::`*pre*`:`*name* kann abgekürzt werden durch `@`*pre*`:`*name*.

> **Beispiel:**
> Insgesamt läßt sich der obige Ausdruck abkürzen zu: `skript/*//code/@class`

Vereinigung

In XPath gibt es ein Konstrukt, das die Vereinigung zweier durch einen XPath-Ausdruck beschriebener Listen beschreibt. Syntaktisch wird dieses durch einen senkrechten Strich `|` ausgedrückt.

> **Beispiel:**
> Der Audruck `/skript/kapitel | /skript/anhang/kapitel` beschreibt die `kapitel`-Elemente die unter top-level Knoten `skript` hängen oder die unter einen Knoten `anhang`, der unter dem `skript`-Element hängt.

Funktionen und Operatoren

XPath kommt mit einer großen Zahl eingebauter Funktionen und Operatoren. Diese sind in einer eigenen Empfehlung des W3C spezifiziert[**?**]. Sie können auf Mengen von Knoten aber auch auf Zahlen, Strings und bool'schen Werten definiert sein. So gibt es z.B. die Funktion `count`, die für eine Knotenliste die Anzahl der darin enthaltenen Knoten angibt.

> **Beispiel:**
> Der Ausdruck `count(//code)` gibt die Anzahl der in einem Dokument enthaltenen `code`-Elemente zurück.

Literale

Für die Rechnung mit Funktionen und Operatorn gibt es Literalen für Strings und Zahlen in XPath.

> **Beispiel:**
> Der Ausdruck `count(//kapitel)=5` wertet zu `true` aus, wenn des Dokument genau fünf `kapitel`-Elemente hat.

Qualifizierung

Bisher haben wir gesehen, wie entlang der Achsen Mengen von Teildokumenten von XML-Dokementen selektiert werden können. XPath sieht zusätzlich vor, durch ein Prädikat über diese Menge zu filtern. Ein solches Prädikat gibt dabei an, ob ein solcher Knoten in der Liste gewünscht ist oder nicht. Man nennt Ausdrücke mit einem Prädikat auch qulifizierte Ausdrücke.

Syntaktisch stehen die Prädikate eines XPath-Ausdrucks in eckigen Klammern hinter den Ausdruck. Das Prädikat ist selbst wieder ein XPath-Ausdruck. Die Auswertung eines qualifizierten XPath-Ausdrucks funktioniert nach folgender Regel:

> Berechne die Ergebnisliste des Ausdrucks. Für jedes Element dieser Liste als aktuellen Kontextknoten berechne das Ergebnis des Prädikats. Interpretiere dieses Ergebnis als bool'schen Wert und verwerfe entweder den Kontextknoten oder nimm ihn in das Ergebnis aus.

Je nachdem was das Prädikat für ein Ergebnis hat, wird es als wahr oder falsch interpretiert. Diese Interpretation ist relativ pragmatisch. XPath war ursprünglich so konzipiert, daß es vollkommen ungetypt ist und es während der Ausführung zu keinerlei Typcheck kommt, geschweige denn ein statisches Typsyystem existiert. Daher wird bei der Anwendung von Funktionen und Operatoren als auch bei der Auswertung eines Prädikats versucht, jeden Wert als jeden beliebige Typ zu interpretieren.

bool'sches Ergebnis Prädikate die direkt durch ein Funktionsergebnis oder eine Operatoranwendung einen bool'schen Wert als Ergebnis haben, können direkt als Prädikat interpretiert werden.

> **Beispiel:**
> Der XPath-Ausdruck `//code[@lang="hs"]` selektiert alle Code-Knoten des Dokuments, die ein Attribut `lang` mit dem Wert `hs` haben.

leere oder nichtleere Ergebnisliste Wenn der XPath-Ausdruck eines Prädikats zu einer Liste auswertet, dann wird eine leere Liste als der bool'sche Wert `false` ansonsten als `true` interpretiert.

> **Beispiel:**
> Der XPath-Ausdruck `//example[.//code]` selektiert alle `example`-Knoten des Dokuments, die mindestens einen `code`-Nachkommen haben. Also alle Beispiele, in denen ein Programm vorkommt.

eine Zahl als Ergebnis Über Funktionen kann ein XPath-Ausdruck auch eine Zahl als Ergebnis haben. In dem Fall, daß ein XPath-Ausdruck zu einer Zahl auswertet, wird ein Knoten selektiert, wenn er an der Stelle dieser Zahl in der Liste des qualifizierten Ausdrucks ist.

> **Beispiel:**
> Der Ausdruck `/skript/kapitel[3]` selektiert jeweils das dritte `kapitel`-Element innerhalb eines `skript`-Elements.

Klammerung

Schließlich kennt XPath noch die Klammerung von Teilausdrücken.

> **Beispiel:**
> Der Ausdruck `./skript/(./kapitel | ./anhang)` bezeichnet alle `kapitel`- oder `anhang`-Elemente ausgehend vom `skript`-Kind des Kontextknotens. Hingegen `./skript/kapitel | ./anhang` bezeichnet die `kapitel`-Elemente unter dem `skript`-Kinder des Kontextknotens oder `anhang`-Kinder des Kontextknotens.

Algebraischer Typ für XPath-Ausdrücke

In diesem Abschnitt wollen wir einmal versuchen einen eigenen zumindest rudimentären XPath Prozessor zu schreiben. Ein XPath Prozessor selektiert ausgehend von einem aktuellen Knoten anhand eines gegebenen XPath Ausdrucks eine Liste von Teildokumenten. Zunächst brauchen wir einen Typ, der XPath-Ausdrücke beschreiben kann. Wir können dieses in einem algebraischen Tyen einfach zusammenfassen. Für jedes XPath Konstrukt gibt es einen eigenen Konstruktor:

XPath.java

```
1  package name.panitz.xml.xpath;
2  import java.util.List;
3  import name.panitz.adt.*;
4
5  @Data abstract class XPath {
6    @Constr void Axis(AxisType type,NodeTest nodeTest){};
7    @Constr void RootSlash(XPath expr){};
8    @Constr void RootSlashSlash(XPath expr){};
9    @Constr void Slash(XPath e1,XPath e2){};
10   @Constr void SlashSlash(XPath e1,XPath e2){};
11   @Constr void TagName(String name){};
12   @Constr void AttributeName(String name){};
13   @Constr void Dot(){};
14   @Constr void DotDot(){};
15   @Constr void NodeSelect(){};
16   @Constr void TextSelect(){};
17   @Constr void PISelect(){};
18   @Constr void CommentSelect(){};
19   @Constr void Star(){};
20   @Constr void AtStar(){};
21   @Constr void Union(XPath e1,XPath e2){};
```

```
  @Constr void Function(String name,List<XPath> arguments){};
  @Constr void BinOperator(String name, XPath e1,XPath e2){};
  @Constr void UnaryOperator(String name, XPath expr){};
  @Constr void QualifiedExpr(XPath expr,XPath qualifier){};
  @Constr void NumLiteral(Integer value){};
  @Constr void StringLiteral(String value){};

  abstract public <b_> b_ welcome(XPathVisitor<b_> visitor);
}
```

Die Klammerung wird in Objekten diesen algebraischen Typs durch die Baumstruktur dargestellt.

Textuelle Darystellung von XPath Ausdrücken Zunächst folgt ein Besucher, der uns ein XPath-Objekt wieder in der XPath-Syntax darstellt:

ShowXPath.java

```
package name.panitz.xml.xpath;

public class ShowXPath extends XPathVisitor<StringBuffer> {
  StringBuffer result = new StringBuffer();
  public StringBuffer visit(Axis e){
    result.append(e.type);
    result.append("::");
    result.append(e.nodeTest.welcome(new ShowNodeTest()));
    return result;
  }
  public StringBuffer visit(RootSlash e){
    result.append("/"); e.expr.welcome(this);
    return result;}
  public StringBuffer visit(RootSlashSlash e){
    result.append("//"); e.expr.welcome(this);
    return result;}
  public StringBuffer visit(Slash e){
    e.e1.welcome(this);result.append("/");e.e2.welcome(this);
    return result;}
  public StringBuffer visit(SlashSlash e){
    e.e1.welcome(this);result.append("//");e.e2.welcome(this);
    return result;}
  public StringBuffer visit(TagName e){result.append(e.name);
    return result;}
  public StringBuffer visit(AttributeName e){
    result.append("@");result.append(e.name);
    return result;}
  public StringBuffer visit(Dot e){result.append(".");
    return result;}
  public StringBuffer visit(DotDot e){result.append("..");
    return result;}
  public StringBuffer visit(NodeSelect e){result.append("node()");
```

```
33      return result;}
34    public StringBuffer visit(TextSelect e){result.append("text()");
35      return result;}
36    public StringBuffer visit(CommentSelect e){result.append("comment()");
37      return result;}
38    public StringBuffer visit(PISelect e){
39      result.append("processing-instruction()");
40      return result;}
41    public StringBuffer visit(Star e){result.append("*");
42      return result;}
43    public StringBuffer visit(AtStar e){result.append("@*");
44      return result;}
45    public StringBuffer visit(Union e){
46      e.e1.welcome(this);result.append("|");e.e2.welcome(this);
47      return result;}
48    public StringBuffer visit(Function e){
49      result.append(e.name);result.append("(");
50      boolean first = true;
51      for (XPath arg:e.arguments){
52        if (!first) result.append(",");else first=false;
53        arg.welcome(this);
54      }
55      result.append(")");
56      return result;}
57    public StringBuffer visit(UnaryOperator e){
58      result.append(e.name+"\u0020");e.expr.welcome(this);
59      return result;}
60    public StringBuffer visit(BinOperator e){
61      e.e1.welcome(this);result.append("\u0020"+e.name+"\u0020");
62      e.e2.welcome(this);return result;}
63    public StringBuffer visit(QualifiedExpr e){
64      e.expr.welcome(this);result.append("[");
65      e.qualifier.welcome(this);result.append("]");
66      return result;}
67    public StringBuffer visit(NumLiteral e){result.append(""+e.value);
68      return result;}
69    public StringBuffer visit(StringLiteral e){
70     result.append("\"");result.append(e.value) ;result.append("\"");
71     return result;}
72  }
```

Entfernen von abkürzenden Schreibweisen

Wir wollen XPath-Ausdrücke auf Dokumentknoten anwenden. Wenn wir zunächst alle abkürzende Schreibweisen in einem XPath-Ausdruck durch ihren Kernausdruck ersetzen, so brauchen wir bei der Anwendung eines XPath-Ausdrucks nicht mehr um abgekürzte Ausdrücke kümmern. Daher schreiben wir zunächst einen Besucher, der in einem XPath-Ausdruck alle Abkürzungen löscht:

RemoveAbbreviation.java

```
1  package name.panitz.xml.xpath;
2
3  import java.util.List;
4  import java.util.ArrayList;
5  import org.w3c.dom.*;
6  import name.panitz.domtest.*;
7
8  public class RemoveAbbreviation extends XPathVisitor<XPath> {
```

Entfernung von: //e Die komplexeste abkürzende Schreibweise ist der Doppelschrägstrich. Er wird ersetzt durch den Ausdruck: `/descendant-or-self::node()/`. Für den einstelligen //-Operator ergibt das den Javaausdruck:

```
new RootSlash(new Slash(new Axis(AxisType.descendant_or_self,new IsNode()),expr))
```

Wir erhalten folgende Implementierung:

RemoveAbbreviation.java

```
9  public XPath visit(RootSlashSlash e){
10   /* //expr -> /descendant-or-self::node()/expr  */
11   return
12     new RootSlash(
13      new Slash(new Axis(AxisType.descendant_or_self,new IsNode())
14               ,e.expr.welcome(this)));}
```

Entfernung von: e1//e2 In gleicher Weise ist der doppelte Schrägstrich als zweistelliger Operator durch den Kernausdruck `/descendant-or-self::node()/` zu ersetzen.

RemoveAbbreviation.java

```
15 public XPath visit(SlashSlash e){
16   final XPath e1 = e.e1;
17   final XPath e2 = e.e2;
18   /* e1//e2 -> e1/descendant-or-self::node()/e2 */
19   return
20    new Slash
21      (e1.welcome(this)
22      ,new Slash(new Axis(AxisType.descendant_or_self,new IsNode())
23                ,e2.welcome(this)));}
```

Entfernung von: . Der einfache Punkt wird durch einen Kernausdruck auf der Selbstachse ersetzt.

RemoveAbbreviation.java

```
24 public XPath visit(Dot e){
25   return new Axis(AxisType.self,new IsNode());}
```

Entfernung von: .. Der doppelte Punkt wird durch einen Kernausdruck auf der Elternachse ersetzt.

RemoveAbbreviation.java

```
public XPath visit(DotDot e){
  return new Axis(AxisType.parent,new IsNode());}
```

Entfernung von: qname Für einen einfachen Tagnamen wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java

```
public XPath visit(TagName e){
  return new Axis(AxisType.child,new QName("",e.name));}
```

Entfernung von: @attr Für einen einfachen Attributnamen wird die implizit vorhandene Attributachse eingefügt.

RemoveAbbreviation.java

```
public XPath visit(AttributeName e){
  return new Axis(AxisType.attribute,new QName("",e.name));}
```

Entfernung von: node() Für einen einfache Knotentest wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java

```
public XPath visit(NodeSelect e){
  return new Axis(AxisType.child,new IsNode());}
```

Entfernung von: text() Für einen einfache Texttest wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java

```
public XPath visit(TextSelect e){
  return new Axis(AxisType.child,new IsText());}
```

Entfernung von: processing-instruction() Für einen einfache Processing-Instruction-Test wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java

```
public XPath visit(PISelect e){
  return new Axis(AxisType.child,new IsProcessingInstruction());}
```

Entfernung von: comment() Für einen einfache Kommentartest wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java

```
public XPath visit(CommentSelect e){
  return new Axis(AxisType.child,new IsComment());}
```

Entfernung von: * Für einen einfache Sterntest wird die implizit vorhandene Kinderachse eingefügt.

RemoveAbbreviation.java

```
public XPath visit(Star e){
  return new Axis(AxisType.child,new StarTest());}
```

Entfernung von: @* Für einen einfache Sterntest auf Attributen wird die implizit vorhandene Attributachse eingefügt.

RemoveAbbreviation.java

```
public XPath visit(AtStar e){
  return new Axis(AxisType.attribute,new StarTest());}
```

Nichtabgekürzte Ausdrücke Für die Ausdrücke, die keine abkürzende Schreibweise darstellen, wird der Besucher in die Unterausdrücke geschickt, um in diesen abkürzende Schreibweisen zu ersetzen.

RemoveAbbreviation.java

```
public XPath visit(Slash e){
  return new Slash(e.e1.welcome(this),e.e2.welcome(this));}
public XPath visit(Axis e){return e;}
public XPath visit(RootSlash e){
  return new RootSlash(e.expr.welcome(this));}
public XPath visit(Union e){
  return new Union(e.e1.welcome(this),e.e2.welcome(this));}
public XPath visit(Function e){
  final List<XPath> args = e.arguments;
  List<XPath> newArgs = new ArrayList<XPath>();
  for (XPath arg:args) newArgs.add(arg.welcome(this));
  return new Function(e.name,newArgs);}
public XPath visit(BinOperator e){
  return new BinOperator(e.name
                        ,e.e1.welcome(this)
                        ,e.e2.welcome(this));}
public XPath visit(UnaryOperator e){
  return new UnaryOperator(e.name,e.expr.welcome(this));}
public XPath visit(QualifiedExpr e){
  return new QualifiedExpr(e.expr.welcome(this)
                          ,e.qualifier.welcome(this));}
public XPath visit(NumLiteral e){return e;}
public XPath visit(StringLiteral e){return e;}
}
```

Auswerten von XPath Ausdrücken

Wir haben nun alles beisammen, um einen XPath-Prozessor zu definieren.

Auswertungsergebnis Die Auswertung eines XPath-Ausdrucks kann eine Liste von Knoten, eine Zahl, ein String oder ein bool'sches Ergebnis haben. Hierfür sehen wir einen algebraischen Typ vor.

XPathResult.java

```
package name.panitz.xml.xpath;
import org.w3c.dom.Node;
import java.util.List;
import name.panitz.adt.*;

@Data abstract class XPathResult {
  @Constr void BooleanResult(Boolean value){};
  @Constr void NumResult(Integer value){};
  @Constr void StringResult(String value){};
  @Constr void Nodes(List<Node> value){};

  abstract public <b_> b_ welcome(XPathResultVisitor<b_> visitor);
}
```

Auswertung Der eigentliche Prozessor wird als Besucher über einen XPath-Ausdruck geschrieben. Dieser Besucher braucht drei Informationen:

- den Kontextknoten, auf den der Ausdruck angewendet werden soll.
- die Größe der Liste, in der der Knotextknoten ist.
- die Position des Kontextknotens in der Liste, in der er sich befindet.

EvalXPath.java

```
package name.panitz.xml.xpath;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;
import static org.w3c.dom.Node.*;
import name.panitz.domtest.*;

import static name.panitz.xml.xpath.Axes.*;

public class EvalXPath extends XPathVisitor<XPathResult> {
  Node current;
  int contextPosition=1;
  int contextSize=1;

  public EvalXPath(Node n){current=n;}
  public EvalXPath(Node n,int pos,int size){
    this(n);contextPosition=pos;contextSize=size;}
```

Einen Kernausdruck mit einer Achse können wir bereits auswerten. Die entsprechende Methode haben wir bereits in der Klasse `DoNodeTest` implementiert.

EvalXPath.java

```
19  public XPathResult visit(Axis e){
20    return new Nodes(
21      DoNodeTest.evalAxisExpr(e.type,e.nodeTest,current));
22  }
```

Für den einstelligen Schrägstrich ist der Dokumentknoten des Knotextsknotens zu beschaffen, um für diesen als neuen Kontextknoten den XPath-Ausdruck rechts von dem Schrägstrich anzuwenden.

EvalXPath.java

```
23  public XPathResult visit(RootSlash e){
24    Node doc ;
25    if (current.getNodeType()==DOCUMENT_NODE) doc=current;
26    else{
27      doc = current.getOwnerDocument();
28    }
29    if (doc!=null) return e.expr.welcome(new EvalXPath(doc));
30
31    return new Nodes(new ArrayList<Node>());
32  }
```

Für den zweistelligen Schrägstrich ist zunächst der linke Kinderausdruck zu besuchen. Falls dieses eine Knotenliste darstellt, ist für jedes Element dieser Liste als Kontextknoten der zweite Operand zu besuchen.

EvalXPath.java

```
33  public XPathResult visit(Slash e){
34    XPathResult res1 = e.e1.welcome(this);
35
36    if (res1 instanceof Nodes){
37      final List<Node> resultNodes = new ArrayList<Node>();
38      final List<Node> e1s = ((Nodes)res1).value;
39      final int size = e1s.size();
40      int pos = 1;
41      for (Node e1:e1s){
42        XPathResult e2s = e.e2.welcome(new EvalXPath(e1,pos,size));
43        if (e2s instanceof Nodes)
44           resultNodes.addAll(((Nodes)e2s).value);
45        else return e2s;
46        pos=pos+1;
47      }
48      return new Nodes(resultNodes);
49    }
50    return res1;
51  }
```

Für die Vereinigung zweier XPath-Ausdrücke, sind diese beide Auszuwerten und in Falle einer Knotenliste als Ergebnis beide Liste zu vereinigen.

EvalXPath.java

```
52  public XPathResult visit(Union e){
53    XPathResult r1 = e.e1.welcome(this);
54    XPathResult r2 = e.e1.welcome(this);
55    if (r1 instanceof Nodes && r2 instanceof Nodes){
56      List<Node> resultNodes = ((Nodes)r1).value;
57      resultNodes.addAll(((Nodes)r2).value);
58      return new Nodes(resultNodes);
59    }
60    return r1;
61  }
```

Ein numerisches Literal ist gerade nur der Wert dieses Literals das Ergebnis.

EvalXPath.java

```
62  public XPathResult  visit(NumLiteral e){
63    return new NumResult(e.value);}
```

Ein Stringliteral ist gerade nur der Wert dieses Literals das Ergebnis.

EvalXPath.java

```
64  public XPathResult visit(StringLiteral e){
65    return new StringResult(e.value);}
```

Für einen qualifizierten Ausdruck ist erst der XPath-Ausdruck auszuwerten, und dann für jedes Element der Knotenliste das Prädikat. Es ist zu testen, ob dieses Prädikat als wahr oder falsch zu interpretieren ist. Hierzu wird ein Besucher auf `XPathResult` geschrieben.

EvalXPath.java

```
66  public XPathResult visit(QualifiedExpr e){
67    XPathResult result1 = e.expr.welcome(this);
68    if (result1 instanceof Nodes){
69      final List<Node> resultNodes = new ArrayList<Node>();
70      final List<Node> rs = ((Nodes)result1).value;
71      final int size = rs.size();
72      int pos = 1;
73      for (Node r : rs){
74        XPathResult qs
75           = e.qualifier.welcome(new EvalXPath(r,pos,size));
76        if (qs.welcome(new TestQualifier(r,pos,size)))
77          resultNodes.add(r);
78        pos=pos+1;
79      }
80      return new Nodes(resultNodes);
81    }
82    return result1;
83  }
```

Die folgenden Fälle brauchen wir in der Auswertung nicht betrachten, weil wir sie der Elemination von abkürzenden Schreibweisen aus dem XPath-Baum entfernt haben

EvalXPath.java

```
  public XPathResult visit(RootSlashSlash e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(SlashSlash e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(TagName e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(AttributeName e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(Dot e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(DotDot e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(NodeSelect e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(TextSelect e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(PISelect e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(CommentSelect e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(Star e){
   throw new UnsupportedOperationException();}
  public XPathResult visit(AtStar e){
   throw new UnsupportedOperationException();}
```

Operatoren und Funktionen sind in unserem Prozessor noch nicht implementiert.

EvalXPath.java

```
  public XPathResult visit(Function e){
   throw new UnsupportedOperationException();
  }
  public XPathResult visit(BinOperator e){
   throw new UnsupportedOperationException();
  }
  public XPathResult visit(UnaryOperator e){
   throw new UnsupportedOperationException();
  }
```

Es folgen ein paar erste Beispielaufrufe:

EvalXPath.java

```
  public static XPathResult eval(XPathAdt xpath,Node n){
    System.out.println(xpath.welcome(new NormalizeXPath())
                            .welcome(new ShowXPath()));

    System.out.println(xpath.welcome(new NormalizeXPath())
                            .welcome(new RemoveAbbreviation())
                            .welcome(new ShowXPath()));

```

```
    return xpath.welcome(new NormalizeXPath())
                .welcome(new RemoveAbbreviation())
                .welcome(new EvalXPath(n));
  }

  public static void main(String [] args){
    Node doc = ParseXML.parseXml(args[0]);

    XPathResult result
     =  eval(new QualifiedExpr
           (new RootSlashSlash
               (new Slash(new TagName("code"),new AttributeName("class")))
           ,new StringLiteral("XPath"))
            ,doc);
    System.out.println(result);

    result
     =  eval
         (new QualifiedExpr(new RootSlash
            (new Slash(new TagName("skript")
            ,new Slash(new TagName("kapitel")
            ,new Slash(new TagName("section"),new DotDot()))))
          ,new NumLiteral(1))
         ,doc);

    System.out.println(result);

    result
     =  eval
         (new RootSlash
            (new Slash(new TagName("skript")
            ,new Slash(new TagName("kapitel")
            ,new QualifiedExpr(new TagName("section"),new NumLiteral(2)))))
         ,doc);

    System.out.println(result);
  }
}
```

Prädikatauswertung Es bleibt das Ergebnis eines Prädikats als bool'schen Wert zu interpretieren. Hierzu schreiben wir einen Besucher auf `XPathResult`.

TestQualifier.java

```
package name.panitz.xml.xpath;

import org.w3c.dom.Node;

public class TestQualifier extends XPathResultVisitor<Boolean> {
  final int pos;
```

```
  final int size;
  final Node context;

  public TestQualifier(Node c,int p,int s){context=c;pos=p;size=s;}

  public Boolean visit(BooleanResult e){return e.value;}
  public Boolean visit(NumResult e){return e.value==pos;}
  public Boolean visit(StringResult e) {
   return e.value.equals(context.getNodeValue());}
  public Boolean visit(Nodes e) {return !e.value.isEmpty();}
}
```

NormalizeXPath.java

```
package name.panitz.xml.xpath;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;
import name.panitz.domtest.*;

public class NormalizeXPath extends XPathVisitor<XPath> {

public XPath visit(Axis e){return e;}

public XPath visit(RootSlash e){
//  final XPath n1 = e.expr;
  return new RootSlash(e.expr.welcome(this));}


public XPath visit(RootSlashSlash e){
  return new RootSlashSlash(e.expr.welcome(this));}

/**  mache Slash(Slash(e1,e2),e3)
   zu Slash(e1,Slash(e2,e3))
und
  mache Slash(SlashSlash(e1,e2),e3)
   zu SlashSlash(e1,Slash(e2,e3))
*/

public XPath visit(Slash e){
  final XPath n1 = e.e1;
  final XPath e3 = e.e2;
  if (n1 instanceof Slash){
    final XPath e1 = ((Slash)n1).e1;
    final XPath e2 = ((Slash)n1).e2;
    return new Slash(e1,new Slash(e2,e3)).welcome(this);
  }
  if (n1 instanceof SlashSlash){
    final XPath e1 = ((SlashSlash)n1).e1;
```

```
    final XPath e2 = ((SlashSlash)n1).e2;
    return new SlashSlash(e1,new Slash(e2,e3)).welcome(this);
  }
  return new Slash(n1.welcome(this),e3.welcome(this));
}

public XPath visit(SlashSlash e){
  final XPath n1 = e.e1;
  final XPath e3 = e.e2;
  if (n1 instanceof Slash){
    final XPath e1 = ((Slash)n1).e1;
    final XPath e2 = ((Slash)n1).e2;
    return new Slash(e1,new SlashSlash(e2,e3)).welcome(this);
  }
  if (n1 instanceof SlashSlash){
    final XPath e1 = ((SlashSlash)n1).e1;
    final XPath e2 = ((SlashSlash)n1).e2;
    return new SlashSlash(e1,new SlashSlash(e2,e3)).welcome(this);
  }
  return new Slash(n1.welcome(this),e3.welcome(this));
}

public XPath visit(Union e){
  return new Union(e.e1.welcome(this),e.e2.welcome(this));}
public XPath visit(TagName e){return e;}
public XPath visit(AttributeName e){return e;}
public XPath visit(Dot e){return e;}
public XPath visit(DotDot e){return e;}
public XPath visit(NodeSelect e){return e;}
public XPath visit(TextSelect e){return e;}
public XPath visit(PISelect e){return e;}
public XPath visit(CommentSelect e){return e;}
public XPath visit(Star e){return e;}
public XPath visit(AtStar e){return e;}

public XPath visit(Function e){return e;}
public XPath visit(BinOperator e){return e;}
public XPath visit(UnaryOperator e){return e;}
public XPath visit(QualifiedExpr e){
  return new QualifiedExpr(e.expr.welcome(this)
                           ,e.qualifier.welcome(this));}
public XPath visit(NumLiteral e){return e;}
public XPath visit(StringLiteral e){return e;}
}
```

10.4.2 XSLT: Transformationen in Dokumenten

XML-Dokumente enthalten keinerlei Information darüber, wie sie visualisiert werden sollen. Hierzu kann man getrennt von seinem XML-Dokument ein sogenanntes *Stylesheet* schreiben. XSL ist eine Sprache zum Schreiben von Stylesheets für XML-Dokumente. XSL ist in gewisser Weise eine Programmiersprache, deren Programme eine ganz bestimmte Aufgabe haben: XML Dokumente in andere XML-Dokumente zu transformieren. Die häufigste Anwendung von XSL dürfte sein, XML-Dokumente in HTML-Dokumente umzuwandeln.

Beispiel:
Wir werden die wichtigsten XSLT-Konstrukte mit folgendem kleinem XML Dokument ausprobieren:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="cdTable.xsl"?>
<cds>
 <cd>
  <artist>The Beatles</artist>
  <title>White Album</title>
  <label>Apple</label>
 </cd>
 <cd>
  <artist>The Beatles</artist>
  <title>Rubber Soul</title>
  <label>Parlophone</label>
 </cd>
 <cd>
  <artist>Duran Duran</artist>
  <title>Rio</title>
  <label>Tritec</label>
 </cd>
 <cd>
  <artist>Depeche Mode</artist>
  <title>Construction Time Again</title>
  <label>Mute</label>
 </cd>
 <cd>
  <artist>Yazoo</artist>
  <title>Upstairs at Eric's</title>
  <label>Mute</label>
 </cd>
 <cd>
  <artist>Marc Almond</artist>
  <title>Absinthe</title>
  <label>Some Bizarre</label>
 </cd>
 <cd>
  <artist>ABC</artist>
  <title>Beauty Stab</title>
  <label>Mercury</label>
```

```
 </cd>
</cds>

```

Gesamtstruktur

XSLT-Skripte sind syntaktisch auch wieder XML-Dokumente. Ein XSLT-Skript hat feste Tagnamen, die eine Bedeutung für den XSLT-Prozessor haben. Diese Tagnamen haben einen festen definierten Namensraum. Das äußerste Element eines XSLT-Skripts hat den Tagnamen `stylesheet`. Damit hat ein XSLT-Skript einen Rahmen der folgenden Form:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>

```

Einbinden in ein XML-Dokument Wir können mit einer *Processing-Instruction* am Anfang eines XML-Dokumentes definieren, mit welchem XSLT Stylesheet es zu bearbeiten ist. Hierzu wird als Referenz im Attribut `href` die XSLT-Datei angegeben.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="cdTable.xsl"?>
<cds>
 <cd>........
```

Templates (Formulare)

Das wichtigste Element in einem XSLT-Skript ist das Element `xsl:template`. In ihm wird definiert, wie ein bestimmtes Element transformiert werden soll. Es hat schematisch folgende Form:
`<xsl:template match="`*Elementname*`">`
zu erzeugender Code
`</xsl:template >`

Beispiel:
Folgendes XSLT-Skript transformiert die XML-Datei mit den CDs in eine HTML-Tabelle, in der die CDs tabellarisch aufgelistet sind.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- Startregel für das ganze Dokument. -->
```

```
7  <xsl:template match="/">
8    <html><head><title>CD Tabelle</title></head>
9      <body>
10       <xsl:apply-templates/>
11     </body>
12   </html>
13 </xsl:template>
14
15
16 <!-- Regel für die CD-Liste. -->
17 <xsl:template match="cds">
18   <table border="1">
19     <tr><td><b>Interpret</b></td><td><b>Titel</b></td></tr>
20     <xsl:apply-templates/>
21   </table>
22 </xsl:template>
23
24 <xsl:template match="cd">
25   <tr><xsl:apply-templates/></tr>
26 </xsl:template>
27
28 <xsl:template match="artist">
29   <td><xsl:apply-templates/></td>
30 </xsl:template>
31
32 <xsl:template match="title">
33   <td><xsl:apply-templates/></td>
34 </xsl:template>
35
36 <!-- Regel für alle übrigen Elemente.
37      Mit diesen soll nichts gemacht werden -->
38 <xsl:template match="*">
39 </xsl:template>
40
41 </xsl:stylesheet>
```

Öffnen wir nun die XML Datei, die unsere CD-Liste enthält im Webbrowser, so wendet er die Regeln des referenzierten XSLT-Skriptes an und zeigt die so generierte Webseite wie in Abbildung 10.1 zu sehen an.

Läßt man sich vom Browser hingegen den Quelltest der Seite anzeigen, so wird kein HTML-Code angezeigt sondern der XML-Code.

Auswahl von Teildokumenten

Das Element `xsl:apply-templates` veranlasst den XSLT-Prozessor die Elemente des XML Ausgangsdokuments weiter mit dem XSL Stylesheet zu bearbeiten. Wenn dieses Element kein Attribut hat, wie in unseren bisherigen Beispielen, dann werden alle Kinder berücksichtigt. Mit dem Attribut `select` lassen sich bestimmte Kinder selektieren, die in der Folge nur noch betrachtet werden sollen.

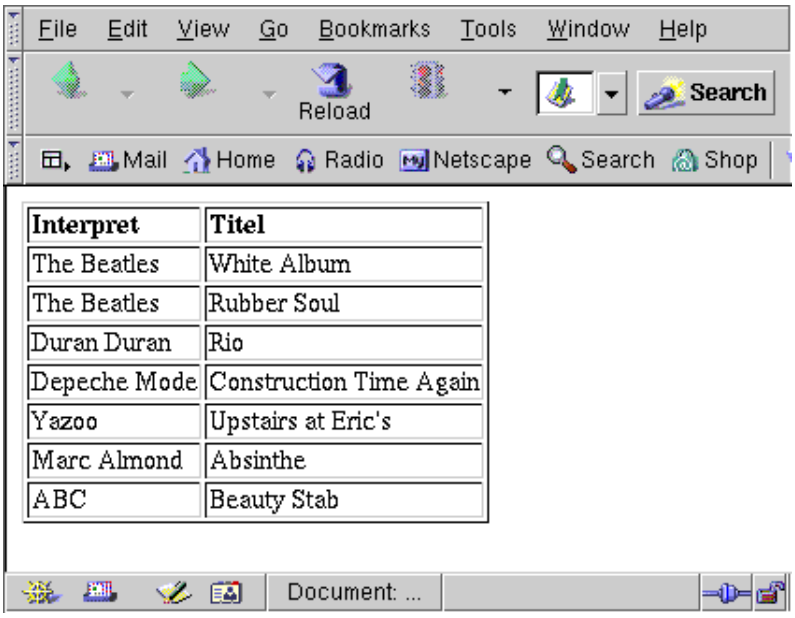


Abbildung 10.1: Anzeige der per XSLT-Skript generierten HTML-Seite.

Beispiel:
Im folgenden XSL Element selektieren wir für `cd`-Elemente nur die `title` und `artist` Kinder und ignorieren die `label` Kinder.

```

<!-- für das Element "cd" -->
<xsl:template match="cd">
 <!--erzeuge ein Element "tr" -->
 <tr>
   <!-- wende das stylesheet weiter an auf die Kinderelemente
   "title" -->
   <xsl:apply-templates select="title"/>
   <!-- wende das stylesheet weiter an auf die Kinderelemente
   "artist" -->
   <xsl:apply-templates select="artist"/>
 </tr>
</xsl:template>
```

Sortieren von Dokumentteilen

XSLT kennt ein Konstrukt, um zu beschreiben, daß bestimmte Dokumente sortiert werden sollen nach bestimmten Kriterien. Hierzu gibt es das XSLT-Element `xsl:sort`. In einem Attribut `select` wird

angegeben, nach welchen Elementteil sortiert werden soll.

Beispiel:

Zum Sortieren der CD-Liste kann mit `xsl:sort` das Unterelement `artist` als Sortierschlüssel bestimmt werden.

```
<xsl:template match="cds">
  <table border="1">
    <tr><td><b>Interpret</b></td><td><b>Titel</b></td></tr>
    <xsl:apply-templates select="cd">
      <xsl:sort select="artist"/>
    </xsl:apply-templates>
  </table>
</xsl:template>
```

Weitere Konstrukte

XSLT kennt noch viele weitere Konstrukte, so z.B. für bedingte Ausdrücke wie in herkömmlichen Programmiersprachen durch einen **if**-Befehl und explizite Schleifenkonstrukte, Möglichkeiten das Ausgangsdokument, mehrfach verschiedentlich zu durchlaufen, und bei einem Element nicht nur auf Grund seines Elementnamens zu reagieren, sondern auch seine Kontext im Dokument zu berücksichtigen. Der interessierte Leser sei auf eines der vielen Tutorials im Netz oder auf die Empfehlung des W3C verwiesen.

XSLT in Java

XSLT.java

```
package name.panitz.xml.xslt;

import org.w3c.dom.Node;

import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.TransformerException;

import java.io.StringWriter;
import java.io.File ;

public class XSLT {
  static public String transform(File xslt,File doc){
    return transform(new StreamSource(doc),new StreamSource(doc));
  }

  static public String transform(Source xslt,Source doc){
    try{
      StringWriter writer = new StringWriter();
      Transformer t =
```

```
        TransformerFactory
         .newInstance()
         .newTransformer(xslt);
      t.transform(doc,new StreamResult(writer));
      return writer.getBuffer().toString();
    }catch (TransformerException _){
      return "";
    }
  }

  public static void main(String [] args)throws Exception {
    System.out.println(transform(new File(args[0]),new File(args[1])));
  }

}
```

10.4.3 XQuery: Anfragen

fac.xquery

```
declare function fac($n) {if ($n=0) then 1 else $n*fac($n - 1)};
fac(5)
```

htmlfac.xquery

```
declare function fac($n) {if ($n=0) then 1 else $n*fac($n - 1)};

<html><body>fac(5)={fac(5)}</body></html>
```

countKapitel.xquery

```
count(doc("/home/sep/fh/prog4/skript.xml")//kapitel)
```

countCode.xquery

```
let $name := distinct-values
                (doc("/home/sep/fh/prog4/skript.xml")//code/@class)
return count($name)
```

exampleProgs1.xquery

```
<ul>{
for $name in distinct-values(doc("/home/sep/fh/prog4/skript.xml")//code/@class
return <li>
         {string($name)}
       </li>
}</ul>
```

exampleProgs2.xquery

```
<ul>{
for $name in distinct-values(doc("/home/sep/fh/prog4/skript.xml")//code/@class
order by $name
```

```
return <li>
         {string($name)}
       </li>
}</ul>
```

exampleProgs.xquery

```
<html>
<title>Klassen aus dem Skript</title>
<body>
<ul>{
let $codeFrags := doc("/home/sep/fh/prog4/skript.xml")//code[@class]
let
 $codes :=
   for $name in distinct-values($codeFrags/@class)
   let $codeFrag := $codeFrags[@class=$name][1]
   return
     <code>
        <className>{string($codeFrag/@class)}</classname>
        <lang>{if ($codeFrag/@lang)
               then string($codeFrag/@lang)
               else "java"}</lang>
        <package>{if ($codeFrag/@package)
                  then string($codeFrag/@package)
                  else "."}</package>
     </code>
for $code in $codes
order by $code/className
return
  let $name:= ($code/className/text(),".",$code/lang/text())
  let $fullname:= ($code/package/text(),"/",$name)
  return
    <li><a>{attribute href {$fullname}}{$name}</a></li>
}</ul></body></html>
```

10.5 Dokumenttypen

Wir haben bereits verschiedene Typen von XML-Dokumenten kennengelernt: XHTML-, XSLT- und SVG-Dokumente. Solche Typen von XML-Dokumenten sind dadurch gekennzeichnet, daß sie gültige XML-Dokumente sind, in denen nur bestimmte vordefinierte Tagnamen vorkommen und das auch nur in einer bestimmten Reihenfolge. Solche Typen von XML-Dokumenten können mit einer Typbeschreibungssprache definiert werden. Für XML gibt es zwei solcher Typbeschreibungssprachen: DTD und Schema.

10.5.1 DTD

DTD *(document type description)* ermöglicht es zu formulieren, welche Tags in einem Dokument vorkommen sollen. DTD ist keine eigens für XML erfundene Sprache, sondern aus SGML geerbt.

DTD-Dokumente sind keine XML-Dokumente, sondern haben eine eigene Syntax. Wir stellen im einzelnen diese Syntax vor:

- `<!DOCTYPE` *root-element* `[` *doctype-declaration*`... ]>`
  Legt den Namen des top-level Elements fest und enthält die gesammte Definition des erlaubten Inhalts.
- `<!ELEMENT` *element-name content-model*`>`
  assoziiert einen *content model* mit allen Elementen, die diesen Namen haben.

  content models können wie folgt gebildet werden.:

  - `EMPTY`: Das Element hat keinen Inhalt.
  - `ANY`: Das Element hat einen beliebigen Inhalt
  - `#PCDATA`: Zeichenkette, also der eigentliche Text.
  - durch einen regulären Ausdruck, der aus den folgenden Komponenten gebildet werden kann.
    - Auswahl von Alternativen (oder): `(...|...|...)`
    - Sequenz von Ausdrücken: `(...,...,...)`
    - Option: `...?`
    - Ein bis mehrfach Wiederholung: `...*`
    - Null bis mehrfache Wiederholung: `...+`
- `<!ATTLIST`*element-name attr-name attr-type attr-default* `...>`
  Liste, die definiert, was für Attribute ein Element hat:

  Attribute werden definiert durch:

  - `CDATA`: beliebiger Text ist erlaubt
  - `(`*value*`|...)`: Aufzählung erlaubter Werte

  Attribut *default* sind:

  - `#REQUIRED`: Das Attribut muß immer vorhanden sein.
  - `#IMPLIED`: Das Attribut ist optional
  - `"value"`: Standardwert, wenn das Attribut fehlt.
  - `#FIXED "value"`: Attribut kennt nur diesen Wert.

Beispiel: Ein Beispiel für eine DTD, die ein Format für eine Rezeptsammlung definiert.

```
<!DOCTYPE collection SYSTEM "collection.dtd" [
<!ELEMENT collection (description,recipe*)>

<!ELEMENT description ANY>

<!ELEMENT recipe (title,ingredient*,preparation
                  ,comment?,nutrition)>

```

```
<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>

<!ELEMENT preparation (step)*>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition fat CDATA #REQUIRED
                    calories CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>]>
```

Aufgabe 38 Schreiben Sie ein XML Dokument, daß nach den Regeln der obigen DTD gebildet wird.

10.5.2 Schema

Daß DTDs keine XML-Dokumente sind, hat die Erfinder von XML im Nachhinein recht geärgert. Außerdem war für die vorgesehen Zwecke im Bereich Datenverwaltung mit XML, die Ausdrucksstärke von DTD zum Beschreiben der Dokumenttypen nicht mehr ausdruckstark genug. Es läßt sich z.B. nicht formulieren, daß ein Attribut nur Zahlen als Wert haben darf. Aus diesen Gründen wurde eine Arbeitsgruppe ins Leben gerufen, die einen neuen Standard definieren sollte, der DTDs langfristig ersetzen kann. Die neue Sprache sollte in XML Syntax notiert werden und die bei DTDs vermissten Ausdrucksmittel beinhalten. Diese neue Typbneschreibungssprache heißt Schema und ist mitlerweile im W3C verabredet worden. Leider ist das endgültige Dokument über Schema recht komplex und oft schwer verständlich geworden. Wir wollen in dieser Vorlesung nicht näher auf Schema eingehen.

10.5.3 Geläufige Dokumenttypen

XHTML

SVG

Wir haben bisher XML-Dokumente geschrieben, um einen Dokumenttext zu strukturieren. Die Strukturierungsmöglichkeiten von XML machen es zu einem geeigneten Format, um beliebige Strukturen von Daten zu beschreiben. Eine gängige Form von Daten sind Graphiken. SVG *(scalable vector graphics)* sind XML-Dokumente, die graphische Elemente beschreiben. Zusätzlich kann in SVG ausgedrückt werden, ob und in welcher Weise Graphiken animiert sind.

SVG-Dokumente sind XML-Dokumente mit bestimmten festgelegten Tagnamen. Auch SVG ist dabei ein Standard, der vom W3C definiert wird.

Beispiel:
Folgendes kleine SVG-Dokument definiert eine Graphik, die einen Kreis, ein Viereck und einen Text enthält.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="200">
 <ellipse cx="100" cy="100" rx="48" ry="90" fill="limegreen" />
 <text x="20" y="115">SVG Textobjekt</text>
 <rect x="50" y="50" width="50" height="60" fill="red"/>
</svg>
```

Für ausführliche Informationen über die in SVG zur Verfügung stehenden Elemente sei der interessierte Student auf eines der vielen Tutorials im Netz oder direkt auf die Seiten des W3C verwiesen.

10.6 Aufgaben

Aufgabe 39 Die folgenden Dokumente sind kein wohlgeformetes XML. Begründen Sie, wo der Fehler liegt, und wie dieser Fehler behoben werden kann.

a)
```
<a>to be </a><b> or not to be</b>
```

Lösung

Kein top-level Element, das das ganze Dokument umschließt.

b)
```
<person geburtsjahr=1767>Ferdinand Carulli</person>
```

Lösung

Attributwerte müssen in Anführungszeichen stehen.

c)
```
<line>lebt wohl<br/>
<b>Gott weiß, wann wir uns wiedersehen</line>
```

Lösung

`<b>` wird nicht geschlossen.

d)
```
<kockrezept><!--habe ich aus dem Netz
  <name>Saltimbocca</name>
  <zubereitung>Zutaten aufeinanderlegen
               und braten.</zubereitung>
</kockrezept>
```

Lösung

Kommentar wird nicht geschlossen.

e)
```
<cd>
  <artist>dead&alive</artist>
  <title>you spin me round</title></cd>
```

Lösung

Das Zeichen & muß als character entity `&` geschrieben werden.

Aufgabe 40 Gegeben sind das folgende XML-Dokument:

TheaterAutoren.xml
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<autoren>
<autor>
  <person>
    <nachname>Shakespeare</nachname>
    <vorname>William</vorname>
  </person>
  <werke>
    <opus>Hamlet</opus>
    <opus>Macbeth</opus>
    <opus>King Lear</opus>
  </werke>
</autor>
<autor>
  <person>
    <nachname>Kane</nachname>
    <vorname>Sarah</vorname>
  </person>
  <werke>
    <opus>Gesäubert</opus>
    <opus>Psychose 4.48</opus>
    <opus>Gier</opus>
  </werke>
</autor>
</autoren>
```

a) Schreiben Sie eine DTD, das die Struktur dieses Dokuments beschreibt.

Lösung

AutorenType.dtd
```
<!DOCTYPE autoren SYSTEM "AutorenType.dtd" [
<!ELEMENT autoren (autor+)>
<!ELEMENT autor (person,werke)>
```

```
<!ELEMENT werke (opus*)>
<!ELEMENT opus (#PCDATA)>
<!ELEMENT person (nachname,vorname)>
<!ELEMENT nachname (#PCDATA)>
<!ELEMENT vorname (#PCDATA)>]>
```

b) Entwerfen Sie Java Schnittstellen, die Objekte ihrer DTD beschreiben.

Lösung

Autoren.java

```
package name.panitz.xml.exercise;
import java.util.List;
public interface Autoren {
  List<Autor> getAutorList();
}
```

Autor.java

```
package name.panitz.xml.exercise;
public interface Autor {
  Person getPerson();
  Werke  getWerke();
}
```

Person.java

```
package name.panitz.xml.exercise;
public interface Person {
  Nachname getNachname();
  Vorname  getVorname();
}
```

Werke.java

```
package name.panitz.xml.exercise;
import java.util.List;
public interface Werke {
  List<Opus> getOpusList();
}
```

HasJustTextChild.java

```
package name.panitz.xml.exercise;
public interface HasJustTextChild {
  String getText();
}
```

Opus.java

```
package name.panitz.xml.exercise;
public interface Opus extends HasJustTextChild {}
```

Nachname.java

```
package name.panitz.xml.exercise;
public interface Nachname extends HasJustTextChild {}
```

Vorname.java

```
package name.panitz.xml.exercise;
public interface Vorname extends HasJustTextChild {}
```

c) Schreiben Sie ein XSLT-Skript, das obige Dokument in eine Html Liste der Werke ohne Autorangabe transformiert.

Lösung

WerkeListe.xsl

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
     <html><head><title>Werke</title></head>
       <body><xsl:apply-templates /></body>
     </html>
  </xsl:template>

  <xsl:template match="autoren">
     <ul><xsl:apply-templates select="autor/werke/opus"/></ul>
  </xsl:template>

  <xsl:template match="opus">
     <li><xsl:apply-templates/></li>
  </xsl:template>
</xsl:stylesheet>
```

d) Schreiben Sie eine Javamethode
`List<String> getWerkListe(Autoren autoren);`,
die auf den Objekten Ihrer Schnittstelle für `Autoren` eine Liste von Werknamen erzeugt.

Lösung

GetWerke.java

```
package name.panitz.xml.exercise;
import java.util.List;
import java.util.ArrayList;

public class GetWerke{
  public List<String> getWerkListe(Autoren autoren){
    List<String> result = new ArrayList<String>();
    for (Autor a:autoren.getAutorList()){
      for (Opus opus: a.getWerke().getOpusList())
        result.add(opus.getText());
    }
    return result;
  }
}
```

Aufgabe 41 Gegeben sei folgendes XML-Dokument:

Foo.xml
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<x1><x2><x5>5</x5><x6>6</x6></x2><x3><x7>7</x7></x3>
<x4><x8/><x9><x10><x11></x11></x10></x9></x4></x1>
```

a) Zeichnen Sie dieses Dokument als Baum.

b) Welche Dokumente selektieren, die folgenden XPath-Ausdrücke ausgehend von der Wurzel dieses Dokuments

- `//x5/x6`
- `/descendant-or-self::x5/..`
- //x5/ancestor::*
- /x1/x2/following-sibling::*
- /descendant-or-self::text()/parent::node()

Aufgabe 42 Schreiben Sie eine Methode
`List<Node> getLeaves(Node n);`,
die für einen DOM Knoten, die Liste aller seiner Blätter zurückgibt.

Lösung

GetLeaves.java
```
package name.panitz.xml.exercise;
import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;

public class GetLeaves{
  public List<Node> getLeaves(Node n){
    List<Node> result = new ArrayList<Node>();
    NodeList ns = n.getChildNodes();
    if (ns.getLength()==0){
      result.add(n); return result;
    }
    for (int i=0;i<ns.getLength();i++){
      result.addAll(getLeaves(ns.item(i)));
    }
    return result;
  }
}
```

Kapitel 11

Steuerfäden

Bisher sind wir nur in der Lage, Programme zu beschreiben, deren Abarbeitung streng sequentiell nacheinander im Programmtext festgelegt ist. Insbesondere wenn man graphische Komponenten programmiert, möchte man Progammteile haben, die nebeneinander her arbeiten. In zwei Fenstern können graphisch animierte Objekte stehen, und wir wollen nicht, daß erst das eine und dann, wenn dieses beendet ist, das andere Fenster seine Animation spielt.

Trotzdem sollen die beiden Fenster nicht zu unterschiedlichen Programmen gehören, sondern Bestandteil eines Programms sein, und auch in der Lage sein, mit denselben Objekten zu arbeiten.

Um nebenläufige Programmteile zu programmieren, stellt Java das Konzept der Steuerfäden zur Verfügung. Steuerfäden sind zu unterscheiden von Prozessen im Betriebssystem. Die Prozesse eines Betriebssystems haben eine eigene Speicherumgebung, während die Steuerfäden in Java, alle in derselben virtuellen Maschine laufen und sich einen Speichbereich für ihre Daten teilen und sich darin auch Daten teilen können.

11.1 Schreiben von Steuerfäden

Es gibt zwei Arten, wie in Java Steuerfäden umgesetzt werden können:

- Durch Ableiten von der Klasse `Thread` und überschreiben der Methode `run`.
- Durch Implementierung der Schnittstelle `Runnable`, die Methode `run` enthält.

In der Regel wird man die Klasse `Thread` überschreiben und nur die Schnittstelle `Runnable` selbst implementieren, wenn das Verbot der mehrfachen Erbung, verhindert, daß von der Klasse `Thread` abgelitten werden kann.

Java vergibt bei der Erzeugung eines Steuerfadens jedem Steuerfaden einen Namen, der mit der Methode `getName()` abgefragt werden kann. Dieser Name kann aber auch im Konstruktor gesetzt werden.

Für ein Objekt der Klasse `Thread` kann nach seiner Erzeugung die Methode `start()` aufgerufen werden, die dafür sorgt, daß der Code der Methode `run()` ausgeführt wird. Wurden mehrere

Steuerfäden gestartet, so wird Java mehr oder weniger zufällig dafür sorgen, daß beide Steuerfäden Gelegenheit bekommen, ihre `run`-Methode auszuführen. Es gibt keinerlei Garantie dafür, wie fair und in welcher Weise Java dafür sorgt, daß alle Steuerfäden ihre Methode `run` ausführen können. Insbesondere können Programme, die Steuerfäden benutzen, auf unterschiedlichen Betriebssystem ganz unterschiedliche Verhalten aufweisen.

Beispiel:
Folgendes Programm definiert einen simplen Steuerfaden, der unendlich oft seinen Namen auf den Bildschirm ausgibt. In der `main`-Methode werden zwei Instanzen dieses Steuerfadens erzeugt und gestartet. Anhand der Ausgabe des Programms kann verfolgt werden, wie Java zwischen den zwei Steuerfäden umschaltet:

SimpleThread.java

```
class SimpleThread extends Thread {

  public void run(){
    while (true){
      System.out.print(getName());
    }
  }

  public static void main(String [] args){
    new SimpleThread().start();
    new SimpleThread().start();
  }
}
```

11.1.1 Schlafenlegen von Prozessen

In der Klasse `Thread` gibt es eine statische Methode `void sleep(long millis)`. Der Aufruf dieser Methode bewirkt, daß der aktuell laufende Steuerfaden entsprechend seines Parameters für einen bestimmten Zeitraum nicht weiterarbeitet. Die Einheit für den Parameter sind tausendstel Sekunden.

Beispiel:
Unseren ersten Steuerfaden aus dem letzten Beispiel erweitern wir jetzt darum, daß er sich jeweils eine bestimmte Zeit zur Ruhe legt, nachdem er seinen Namen ausgedruckt hat:

SleepingThread.java

```
class SleepingThread extends Thread {
  public SleepingThread(int s){this.s=s;}
  private int s;

  public void run(){
    while (true){
      System.out.print(getName());
      try {Thread.sleep(s);}catch (InterruptedException _){}
    }
  }

```

```
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 den Steuerfadenobjekten ausgeführt, sondern global und gilt für den gerade aktiven Steuerfaden. Das wird dann jeweils der Steuerfaden sein, in dessen `run`-Methode der Aufruf von `sleep` steht.

11.2 Koordination nebenläufiger Steuerfäden

Wenn zwei Steuerfäden sich Daten teilen, dann kann es zu Problemen kommen.

11.2.1 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

```
class Swap extends Thread{

  public Swap(TwoNumbers twoN){this.twoN=twoN;}

  private TwoNumbers twoN;

  public void run(){
    while (true){twoN.swap();}
  }
}
```

Der andere druckt immer wieder ein `TwoNumbers`-Objekt aus:

PrintTwoNumbers.java

```
class PrintTwoNumbers extends Thread {
  public PrintTwoNumbers(TwoNumbers twoN){this.twoN=twoN;}

  private TwoNumbers twoN;

  public void run(){
    while (true){
      System.out.println(twoN);
      try {Thread.sleep(1000);}catch (InterruptedException _){}
    }
  }

  public static void main(String [] _){
    TwoNumbers twoN = new TwoNumbers(1,2);
    new Swap(twoN).start();
    new PrintTwoNumbers(twoN).start();
  }
}
```

Wenn wir dieses Programm starten, dann sehen wir, daß wir so gut wie nie die beiden Zahlen ausdrucken, sondern immer ein Paar zwei gleicher Zahlen:

```
sep@swe10:~/fh/internal/beispiele> java PrintTwoNumbers
(1,2)
(1,1)
(2,2)
(1,1)
(2,2)
(1,1)
(1,1)
(2,2)
(1,1)
(2,2)
(1,1)
```

```
(1,1)
(2,2)

sep@swe10:~/fh/internal/beispiele>
```

Hier wird das Objekt der Klasse `TwoNumbers` immer nur ausgedruckt, wenn es inmitten der Ausführung der Methode `swap` ist. In diesem Zwischenzustand sind gerade im Zuge des Vertauschungsprozesses beide Zahlen gleich. Will man verhindern, daß andere Steuerfäden einen Zwischenstand einer Operation sehen, also das Objekt in einem Zustand, den es eigentlich nicht haben soll, so kann man in Java das Mittel der Synchronisation anwenden.

11.2.2 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 geschweifte Klammern gesetzt. Zusätzlich ist noch anzugeben, über welches Objekt synchronisiert wird. Insgesamt hat eine `synchronized`-Anweisung die folgende syntaktische Form:

`synchronized` (*obj*){*stats*}

Statt also die Synchronisation an den Methoden vorzunehmen, können wir auch erst bei ihrem Aufruf verlangen, daß dieser synchronisiert ist.

Hierzu synchronisieren wir über das Objekt `twoN` den Aufruf der Methode `swap`:

SafeSwap.java

```
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.

11.2.3 Verklemmungen

Mit der Synchronisation von Operationen über verschiedene Objekte handelt man sich leider ein Problem ein, das unter dem Namen Verklemmung (eng. deadlock)[1] bekannt ist. Wenn es zu einer Verklemmung in einem Programm kommt, friert das Programm ein. Es rechnet nichts mehr, sondern wartet intern darauf, daß andere Programmteile mit der Berechnung eines kritischen Abschnitts fertig werden.

Damit es zu einer Verklemmung kommt, braucht es mindestens zwei Steuerfäden und zwei Objekte, über die diese Steuerfäden synchronisieren.

Verklemmung bei der Buchausleihe

Es gibt eine schöne Analogie für Verklemmungen aus dem Alltag. Hierzu denke man als Steuerfäden zwei Studenten (*Hans* und *Lisa*) und als Objekte zwei Bücher in der Bibliothek (*Programmieren* und *Softwaretechnik*). Hans leiht sich das Buch Programmieren und Lisa das Buch Softwaretechnik aus. Beide haben sich damit exklusiv je ein Objekt gesichert (es gibt nicht mehrere Exemplare des gleichen Buches in der Bibliothek.). Nun steht im Buch Programmieren eine Referenz auf das Buch Softwaretechnik und umgekehrt. Um das Buch Programmieren zu Ende durchzustudieren, braucht Hans jetzt auch das Buch Softwaretechnik, und für Lisa gilt das entsprechende umgekehrt. Beide wollen sich jetzt das andere Buch in der Bibliothek ausleihen, stellen aber fest, daß es verliehen ist. Jetzt warten sie darauf, daß es zurückgegeben wird. Beide geben ihr Buch nicht zurück, weil sie ja mit beiden Büchern arbeiten wollen. Somit warten beide aufeinander. Dieses Phänomen nennt man Verklemmung.

Das 5-Philosophenproblem

Eine zweite Illustration von Verklemmungen ist das sogenannte 5-Philosophen-Problem. Fünf Philosophen sitzen um einen runden Tisch. Zwischen ihnen liegen Gabeln. Ein Philosoph kann entweder denken oder essen. Zum Essen braucht er zwei Gabeln, denken kann er nur, wenn er keine Gabel in der Hand hat. Nun greifen sich die Philosophen entweder Gabeln und essen oder

[1] Im Skript von Herrn Grude[Gru03] findet sich die etwas drastischere deutsche Bezeichnung *tödliche Umklammerung*.

lassen die Gabeln liegen und denken. Problematisch wird es, wenn sich jeder Philosoph erst die Gabel zu seiner Rechten greift, um dann festzustellen, daß auf der linken Seite keine Gabel liegt. Wenn jetzt alle darauf warten, daß links wieder eine Gabel hingelegt wird, und die Gabel in ihrer rechten Hand nicht zurückgeben, so kommt es zu einer Verklemmung. Keiner ißt und keiner denkt.

Verklemmung im Straßenverkehr

Auch im Straßenverkehr kann man mitunter Verklemmungen beobachten. Zwei in entgegengesetzte Richtung fahrende Autos fahren auf eine Kreuzung zu und wollen dort beide links abbiegen. Sie fahren soweit in die Kreuzung ein, daß sie dem entgegenkommenden Fahrzeug jeweils die Fläche auf der Fahrbahn, die es zum Linksabbiegen braucht, blockieren. Beide warten jetzt darauf, daß der andere weiterfährt. Der Verkehr steht.

Verklemmung in Java

Mit Steuerfäden und Synchronisation haben wir in Java die Mechanismen zur Verfügung, die zu einer klassischen Verklemmung führen können. Wir schreiben eine Steuerfadenklasse, in der es zwei Objekte gibt. In der Methode `run` wird erst über das eine Objekt synchronisiert und dann über das andere. Wir erzeugen in der Hauptmethode zwei Objekte dieser Steuerfadenklasse jeweils mit vertauschten Objekten:

Deadlock.java

```
class Deadlock extends Thread{

  Object a;
  Object b;

  Deadlock (Object a,Object b){this.a=a;this.b=b;}

  public void run(){
    synchronized (a){
      System.out.println(
        getName()+"jetzt habe ich das erste Objekt: "+a);
      try {Thread.sleep(1000);}catch (InterruptedException _){}
      synchronized (b){
          System.out.println(getName()+"jetzt habe ich beide");
      }
    }
  }

  public static void main(String [] _){
    String s1 = "hallo";
    String s2 = "welt";
    new Deadlock(s1,s2).start();
    new Deadlock(s2,s1).start();
  }
}
```

Dieses Programm hat zur Folge, daß sich der erste Stuerfaden zunächst das Objekt `s1` reserviert und der zweite Steuerfaden das Objekt `s2` reserviert. Wenn sie jetzt jeweils das andere Objekt anfordern, müssen sie darauf warten, bis der andere Steuerfaden dieses wieder frei gibt. Das Programm ist verklemmt, nichts geht mehr:

```
sep@swe10:~/fh/internal/beispiele> java Deadlock
Thread-0jetzt habe ich das erste Objekt: hallo
Thread-1jetzt habe ich das erste Objekt: welt
```

11.2.4 Warten und Benachrichtigen

Über die Synchronisation haben wir sichergestellt, daß bestimmte Operationen nur ausgeführt werden, wenn man bestimmte Objekte exklusiv hat. Falls ein anderer Steuerfaden dieses Objekt gesperrt hat, so muß der Steuerfaden warten, bis das Objekt für ihn zur Verfügung steht.

Darüberhinaus kann es erwünscht sein, daß ein Steuerfaden warten soll, bis ein anderer Steuerfaden bestimmte Operationen durchgeführt hat. Hierzu gibt es in Java die Methoden `wait` und `notifyAll`.

Stellen wir uns vor, wir haben ein Objekt, in dem eine Zahl gespeichert wird. Es gibt eine Methode, diese Zahl zu setzen, und eine Methode, um diese auszulesen. Zwei Steuerfäden sollen über dieses Objekt kommunizieren. Der eine schreibt immer Zahlen hinein, der andere liest diese wieder aus. Nun soll der lesende Steuerfaden immer erst so lange warten, bis der schreibende eine neue Zahl gespeichert hat, und der speichernde Steuerfaden soll immer so lange warten, bis der vorherige Wert ausgelesen wurde, damit er nicht überschrieben wird.

Ohne Vorkehrungen dafür, daß abwechselnd gelesen und geschrieben wird, sieht die Klasse wie folgt aus:

GetSet1.java

```
1 class GetSet1 {
2   private int i;
3   GetSet1(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

```
1 class Set extends Thread{
2   GetSet1 o;
3   Set(GetSet1 o){this.o=o;}
4   public void run(){
5     for (int i=1;i<=1000;i=i+1){
6       o.set(i);
7     }
8   }
9 }
```

Der lesende Steuerfaden mit entsprechender Hauptmethode:

Get.java

```
class Get extends Thread{
  GetSet1 o;
  Get(GetSet1 o){this.o=o;}
  public void run(){
    for (int i=1;i<=1000;i=i+1){
      System.out.print(o.get()+" ");
    }
  }

  public static void main(String [] _){
    GetSet1 gss = new GetSet1(0);
    new Get(gss).start();
    new Set(gss).start();
  }
}
```

Starten wir dieses Programm, so werden nicht die Zahlen 0 bis 1000 auf dem Bildschirm ausgeben. Es ist nicht garantiert, daß erst nach einem Schreiben ein Lesen stattfindet. Wir können dieses durch einen internen Merker signalisieren, der angibt, ob eine neue Zahl verfügbar ist. Bevor wir eine neue Zahl lesen, können wir uns immer wieder schlafenlegen und dann schauen, ob jetzt die neue Zahl verfügbar ist:

GetSet2.java

```
class GetSet2 {
  private int i;
  private boolean available= true;
  GetSet2(int i){this.i=i;}

  synchronized int get(){
    while (!available){
      try{Thread.sleep(1000);}catch (Exception _){}
    }
    available=false;
    return i;
  }

  synchronized void set(int i){
    while (available){
      try{Thread.sleep(1000);}catch (Exception _){}
    }
    this.i=i;
    available=true;
  }
}
```

Starten wir damit unser Testprogramm, so kommt es zu einer Verklemmung. Um dieses zu umgehen, hat Java das Prinzip von `wait` und `notifyAll`. Statt `sleep` rufen wir die Methode `wait`

auf und erlauben damit anderen Steuerfäden, obwohl wir in einer synchronisierten Methode sind, auf demselben Objekt aktiv zu werden. Die `wait`-Anweisung wird aufgehoben, wenn irgendwo eine `notifyAll`-Anweisung aufgerufen wird. Dann konkurrieren wieder alle Steuerfäden um die Objekte, über die sie synchronisieren.

Unser Programm sieht schließlich wie folgt aus:

GetSet3.java

```
class GetSet3 {
  private int i;
  private boolean available= true;
  GetSet3(int i){this.i=i;}

  synchronized int get(){
    while (!available){
      try{wait();}catch (Exception _){}
    }
    available=false;
    notifyAll();

    return i;
  }

  synchronized void set(int i){
    while (available){
      try{wait();}catch (Exception _){}
    }
    this.i=i;
    available=true;
    notifyAll();
  }
}
```

Damit ist gewährleistet, daß tatsächlich nur abwechselnd gelesen und geschrieben wird. Wie man sieht, kann es relativ kompliziert werden, über synchronisierte Methoden und der `wait`-Anweisung zu programmieren, und Programme mit mehreren Steuerfäden, die untereinander kommunizieren, sind schwer auf Korrektheit zu prüfen.

Kapitel 12

Swing: Eine Bibliothek für graphische Oberflächen

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.awt`-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 gibt es z.B. eine Klasse `JButton`.

Ein fundamentaler Unterschied dieser beiden Pakete besteht in ihrer Behandlung von Steuerfäden. Während das `awt`-Paket sicher in Bezug auf Steuerfäden ist, können im `swing`-Paket Steuerfäden nicht ohne zusätzlihe Absicherung benutzt werden.

Zu allem Unglück sind die beiden Pakete nicht ganz 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 noch einmal für `javax.swing` extra umgesetzt wurden.

Sind Sie verwirrt. Sie werden es hoffentlich nicht mehr sein, nachdem Sie die Beispiele dieses Kapitels durchgespielt haben.

[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 sehr schöne Ausdruck für GUI verwendet.

12.1 Graphische Benutzeroberflächen

Zur Programmierung eines GUIs bedarf es graphischer Objekte, einer 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 diese Aufgaben jeweils eigene Klassen:

- Komponenten
- Ereignisbehandlung
- Layoutsteuerung

In den folgenden Abschnitten, werden wir an ausgewählten Beispielen Klassen für diese drei wichtigen Schritte der GUI-Programmierung kennenlernen.

12.1.1 Graphische Komponenten

Javas `swing`-Paket kennt drei Arten von Komponenten.

- Top-Level Komponenten
- Zwischenkomponenten
- Atomare Komponenten

Leider spiegelt sich diese Unterscheidung nicht in der Ableitungshierarchie wider. 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.

Top-Level Komponenten

Eine top-level Komponenten ist ein GUI-Objekt, das 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.

Window1.java

```
1 package name.panitz.simpleGui;
2 import javax.swing.*;
3
4 public class Window1{
5   public static void main(String [] _){
6     new JFrame("erstes Fenster").setVisible(true);
7   }
8 }
```

Dieses Programm erzeugt ein leeres Fenster und gibt das auf dem Bildschirm aus. Der Fenstertitel kann als Argument dem Konstruktor der Klasse `JFrame` als `String` übergeben werden. Die optische Ausprägung des Fensters kann vom Betriebssystem abhängen. In Abbildung 12.1 ist dieses Fenster beispielhaft in einer Suse Linux Umgebung mit KDE zu bewundern.

Abbildung 12.1: Ein erstes Fenster.

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.

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 hinzugefügt 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

```
package name.panitz.simpleGui;
import javax.swing.*;

class JT {
  public static void main(String [] _){
    JFrame frame = new JFrame();
    JTextArea textArea = new JTextArea();
    textArea.setText("hallo da draußen");
    frame.getContentPane().add(textArea);
    frame.pack();
    frame.setVisible(true);
  }
}
```

Das Programm erzeugt das folgende Fenster:

Abbildung 12.2: Fenster mit Textfläche

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

```
package name.panitz.simpleGui;
import javax.swing.*;

class JTB extends JFrame {
  JPanel pane = new JPanel();
  JTextArea textArea = new JTextArea();
  JButton button = new JButton("ein Knopf");

  public JTB(){
    textArea.setText("hallo da draußen");
    pane.add(textArea);
    pane.add(button);
    getContentPane().add(pane);
    pack();
```

```
15      setVisible(true);
16    }
17
18    public static void main(String [] _){
19      new JTB();
20    }
21  }
```

Das Programm erzeugt folgendes Fenster:

Abbildung 12.3: Fenster mit Textfläche und Knopf

12.1.2 Ereignisbehandlung

Wir haben es bisher schon geschafft ein Fenster zu definieren, in dem mehrere graphische Komponenten dargestellt werden. Für ein funktionierendes GUI ist es nun notwendig, für einzelne Komponenten eine Funktionalität zu definieren. Hierzu kennt Swing das Konzept der Ereignisbehandlung. Graphischen Komponenten kann ein Ereignisbehandlungsobjekt angehänt werden. Objekte zur Ereignisbehandlung müssen dabei bestimmte Schnittstellen implementieren, die sogenannten `Listener`.

Ereignisarten gibt es eine ganze Reihe auf den unterschiedlichsten graphischen Komponenten: ein Knöpf kann gedrückt werden, ein Fenster minimiert, mit der Maus auf eine Komponente gezeigt werden, ein Menupunkt gewählt werden etc. Für die unterschiedlichen Ereignisarten gibt es unterschiedliche Schnittstellen, die beschreiben, wie auf solche Ereignisse reagiert werden soll. Die Schnittstellen für Ereignisse befinden sich im Paket: `java.awt.event.*`.

Beispiel:
Als erstes Beispiel für eine Ereignisbehandlung betrachten wir die vielleicht elementarste Art eines Ereignisses: ein Knopf wird gedrückt. Hierzu gibt es die elementare Ereignisbehandlungsschnittstelle: `java.awt.event.ActionListener`. Diese Schnittstelle enthält nur eine Methode: `actionPerformed`

Wir implementieren diese Schnittstelle. Unsere Klasse zur Ereignisbehandlung bekommt dabei ein Textfeld übergeben. Intern enthalte die Klasse einen Zähler. Wann immer das Ereignis eintritt, soll der Zähler um eins erhöht werden und das neue Ergebnis des Zählers im Textfeld angezeigt werden.

CounterListener.java

```
1  package name.panitz.simpleGui;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class CounterListener implements ActionListener{
6    private final JTextArea textArea;
```

```
  private int counter = 0;
  public CounterListener(JTextArea ta){textArea=ta;}

  public void actionPerformed(ActionEvent _){
    counter = counter+1;
    textArea.setText(counter+"");
  }
}
```

Jetzt können wir eine Unterklasse unserer letzten kleinen Gui-Klasse `JTB` schreiben, in der diese Ereignisbehandlung dem Knopf hinzugefügt wird. Hierzu gibt es für Knöpfe die Methode `addActionListener`.

Counter.java

```
package name.panitz.simpleGui;
import javax.swing.*;

class Counter extends JTB {
  public Counter(){
    super();
    button.addActionListener(new CounterListener(textArea));
  }

  public static void main(String [] _){new Counter();}
}
```

Und tatsächlich, wenn wir dieses Programm starten, so bewirkt, daß ein Drücken des Knopfes mit der Maus bewirkt, daß im Textfeld angezeigt wird, wie oft bisher der Knopf gedrückt wurde.

Aufgabe 43 Die Methode `actionPerformed` der Schnittstelle `ActionListener` bekommt ein Objekt der Klasse `ActionEvent` übergeben. In dieser gibt es eine Methode `getWhen`, die den Zeitpunkt, zu dem das Ereignis aufgetreten ist als eine Zahl kodiert zurückgibt. Mit dieser Zahl können Sie die Klasse `java.util.Date` instanziieren, um ein Objekt zu bekommen, das Datum und Uhrzeit enthält.

Schreiben Sie jetzt eine Unterklasse von `JTB`, so daß beim Drücken des Knopfes, jetzt die aktuelle Uhrzeit im Textfeld erscheint.

Aufgabe 44 Schreiben Sie eine kleine Guianwendung mit einer Textfläche und drei Knöpfen, die einen erweiterten Zähler darstellt:

- einen Knopf zum Erhöhen eines Zählers.
- einen Knopf zum Verringern des Zählers um 1.
- einen Knopf zum Zurücksetzen des Zählers auf 0.

12.1.3 Setzen des Layouts

Bisher haben wir Unterkomponenten weitere Komponenten mit der Methode `add` hinzugefügt, ohne uns Gedanken über die Plazierung der Komponenten zu machen. Wir haben einfach auf das Standardverhalten zur Plazierung von Komponenten vertraut. Ob die Komponenten schließlich nebeneinander, übereinander oder irgendwie anders Gruppiert im Fenster erschienen, haben wir nicht spezifiziert.

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`.

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.

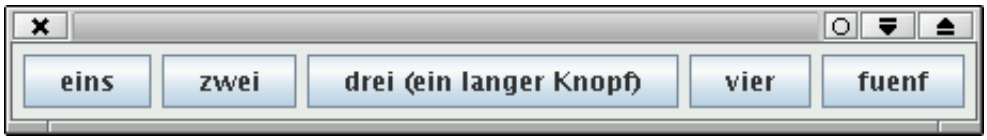
Beispiel:
Die folgende Klasse definiert ein Fenster, dessen Layout über ein Objekt der Klasse `FlowLayout` gesteuert wird. Dem Fenster werden fünf Knöpfe hinzugefügt:

FlowLayoutTest.java

```
package name.panitz.gui.layoutTest;

import java.awt.*;
import javax.swing.*;

class FlowLayoutTest extends JFrame {

  public FlowLayoutTest(){
    Container pane = getContentPane();
    pane.setLayout(new FlowLayout());
    pane.add(new JButton("eins"));
    pane.add(new JButton("zwei"));
    pane.add(new JButton("drei (ein langer Knopf)"));
    pane.add(new JButton("vier"));
    pane.add(new JButton("fuenf"));
    pack();
    setVisible(true);
  }

```

```
20    public static void main(String [] _){new FlowLayoutTest();}
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.

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   package name.panitz.gui.layoutTest;
2
3   import java.awt.*;
4   import javax.swing.*;
5
6   class BorderLayoutTest extends JFrame {
7
8     public BorderLayoutTest(){
9       Container pane = getContentPane();
10      pane.setLayout(new BorderLayout());
11      pane.add(new JButton("eins"),BorderLayout.NORTH);
12      pane.add(new JButton("zwei"),BorderLayout.SOUTH);
13      pane.add(new JButton("drei (ein langer Knopf)")
14                                  ,BorderLayout.CENTER);
15      pane.add(new JButton("vier"),BorderLayout.WEST);
16      pane.add(new JButton("fuenf"),BorderLayout.EAST);
17      pack();
18      setVisible(true);
19    }
20
21    public static void main(String [] _){new BorderLayoutTest();}
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.

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

```
package name.panitz.gui.layoutTest;

import java.awt.*;
import javax.swing.*;

class GridLayoutTest extends JFrame {
  public GridLayoutTest(){
    Container pane = getContentPane();
    pane.setLayout(new GridLayout(2,3));
    pane.add(new JButton("eins"));
    pane.add(new JButton("zwei"));
    pane.add(new JButton("drei (ein langer Knopf)"));
    pane.add(new JButton("vier"));
    pane.add(new JButton("fünf"));
    pack();
    setVisible(true);
  }

  public static void main(String [] _){new GridLayoutTest();}
}
```

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.

12.2 Exkurs: verschachtelte Klassen

Bisher haben wir gelernt, daß jede Java Quelltextdatei genau eine Klasse enthält und Klassen Eigenschaften in Form von Feldern und Methoden enthalten. In den ersten GUI-Beispielen und Aufgaben haben wir schon gesehen, daß es insbesondere für die Ereignisbehandlung oft zu sehr kleinen Klassen kommt, die lediglich für eine ganz spezielle Funktionalität innerhalb einer GUI-Klasse benötigt werden. Um in solchen Situationen den Code etwas struktuierter zu schreiben, so daß ein Ereignisbehandler direkt im Quelltext bei dem GUI-Objekt zu finden ist, gibt es in Java die Möglichkeit, innerhalb einer Klasse ein interne Klasse zu schreiben. Hierbei spricht man von verschachtelten Klassen. Verschachtelte Klassen können statisch sein, oder als nicht-statische Eigenschaft an konkrete Objekte gebunden sein. In diesem Fall heißen sie: *innere Klassen*. Tatsächlich können verschachtelte überall im Code auftauchen: sie können auf obere Ebene parallel zu Feldern und Methoden in einer Klasse definiert werden, aber auch an beliebiger Stelle innerhalb eines Methodenrumpfes.

12.2.1 Innere Klassen

Eine innere Klasse wird gechrieben wie jede andere Klasse auch, nur daß sie eben im Rumpf einer äußeren Klasse auftauchen kann. Die innere Klasse hat das Privilig auf die Eigenschaften der äußeren Klasse zuzugreifen, sogar auf die als privat markierten Eigenschaften. Das Attribut privat soll lediglich verhindern, daß eine Eigenschaft von außerhalb der Klasse benutzt wird. Innere Klasse befinden sich aber innerhalb der Klasse.

Beispiel:
Unser erstes GUI mit einer Funktionalität läßt sich jetzt mit Hilfe einer inneren Klasse in einer Datei schreiben. Das Feld `counter`, das wir in der vorherigen Implementierung als privates Feld der Klass `CounterListener` definiert hatten, haben wir hier als Feld der GUI-Klasse modelliert. Trotzdem kann die Klasse `CounterListener` weiterhin darauf zugreifen. Ebenso brauch die Textfläche nicht der Klasse `CounterListener` im Konstruktor übergeben werden. Als innere Klasse kann in `CounterListener` auf dieses Feld der äußeren Klasse zugegriffen werden.

InnerCounter.java

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3  import java.awt.event.*;
4
5  class InnerCounter extends JTB {
6    private int counter = 0;
7
8    class CounterListener implements ActionListener{
9      public void actionPerformed(ActionEvent _){
10       counter = counter+1;
```

```
11        textArea.setText(counter+"");
12      }
13    }
14
15    public InnerCounter(){
16      button.addActionListener(new CounterListener());
17    }
18
19    public static void main(String [] _){new InnerCounter();}
20  }
```

Tatsächlich ist die Implementierung kürzer und etwas übersichtlicher geworden.

Beim Übersetzen einer Klasse mit inneren Klassen, erzeugt der Javaübersetzer für jede innere Klasse eine eigene Klassendatei:

```
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui> ll *.class
-rw-r--r--    1 sep      users        1082 2004-03-29 11:36 InnerCounter$CounterListener.class
-rw-r--r--    1 sep      users         892 2004-03-29 11:36 InnerCounter.class
```

Der Javaübersetzer schreibt intern den Code um in eine Menge von Klassen ohne innere Klassendefinition und erzeugt für diese den entsprechenden Code. Für die innere Klasse generiert der Javaübersetzer einen Namen, der sich aus äußeren und inneren Klassenamen durch ein Dollarzeichen getrennt zusammensetzt.

Aufgabe 45 Schreiben Sie Ihr Programm eines Zählers mit drei Knöpfen jetzt so, daß sie innere Klassen benutzen.

12.2.2 Anonyme Klassen

Im letzten Abschnitt hatten wir bereits das Beispiel einer inneren Klasse, für die wir genau einmal ein Objekt erzeugen. In diesem Fall wäre es eigentlich unnötig für eine solche Klasse einen Namen zu erfinden, wenn man an genau dieser einen Stelle, an der das Objekt erzeugt wird, die entsprechende Klasse spezifizieren könnte. Genau hierzu dienen anonyme Klassen in Java. Sie ermöglichen, Klassen ohne Namen zu instanziieren. Hierzu ist nach dem Schlüsselwort `new` anzugeben, von welcher Oberklasse namenlose Klasse ableiten soll, oder welche Schnittstelle mit der namenlosen Klasse implementiert werden soll. Dann folgt nach dem leeren Klammerpaar für dem Konstruktoraufruf in geschweiften Klammern der Rumpf der namenlosen Klasse.

Beispiel:
Wir schreiben ein drittes Mal die Klasse `Counter`. Diesmal wird statt der nur einmal instanziierten inneren Klasse eine anonyme Implementierung der Schnittstelle `ActionListener` Instanziiert.

AnonymousCounter.java

```
1  package name.panitz.simpleGui;
2  import javax.swing.*;
3  import java.awt.event.*;
4
```

```
5  class AnonymousCounter extends JTB {
6    private int counter = 0;
7
8    public AnonymousCounter(){
9      button.addActionListener(
10       new ActionListener(){
11         public void actionPerformed(ActionEvent _){
12           counter = counter+1;
13           textArea.setText(counter+"");
14         }
15       });
16   }
17
18   public static void main(String [] _){new AnonymousCounter();}
19 }
```

Auch für anonyme Klassen generiert der Javaübersetzer eigene Klassendateien. Mangels eines Names, numeriert der Javaübersetzer hierbei die inneren Klassen einfach durch.

```
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui> ll *.class
-rw-r--r--    1 sep      users        1106 2004-03-29 11:59 AnonymousCounter$1.class
-rw-r--r--    1 sep      users         887 2004-03-29 11:59 AnonymousCounter.class
sep@linux:~/fh/prog2/examples/classes/name/panitz/simpleGui>
```

12.2.3 Statische verschachtelte Klassen

Selten benutzt gibt es auch statische verschachtelte Klassen. Diese können entsprechend auch nur auf statische Eigenschaften der äußeren Klasse zugreifen.

12.3 Applets

Ältere Einführung in Java begannen gerne mit der Programmierung von sogenannten *applets*. *Applet* ist der englische Diminutiv für Applikation; es handelt sich also bei einem *Applet* um eine kleine oft auch niedliche Applikation. Applets werden im Webbrowser ausgeführt. Auf einer Webseite ist hierzu ein *Tag* `<applet>` einzufügen. In diesem ist eine Javaklasse anzugeben, die der Browser an dieser Stelle der Webseite ausführen soll.

Applets waren ursprünglich tatsächlich die Killerapplikation von Java, denn damit verfügte Java über eine Technik, die weder eine andere Programmiersprache angeboten hat, noch auf andere Weise umgesetzt werden konnte. Mitte der 90er Jahre waren Webseite noch recht nüchterne Angelegenheiten. Es gab vor allen Dingen kaum bewegte Bilder oder Lauftexte. Javas virtuelle Maschine konnte in den Webbrowsern *mosaic* und später *netscape* integriert werden, so daß sich Javaapplikationen auf Webseiten laufen ließen, die bewegte Bilder und ähnliches realisierten. Eine der häufigsten kleinen Demonstrationsprogramme war das Javamännchen, das den Betrachter zuwinkte. Dieses animierte Männchen war als Applet realisiert.[2] Aufgrund der

[2] Wohingehend, wenn man heute das winkende Männchen sieht, es sich meistens um eine animierte gif-Datei handelt. Animiert hinjtereinander zu spielende Bilder kannte auch damals bereits das gif-Format, wurde allerdings nur von wenigen Werkzeugen unterstützt.

Applets brach ein regelrechter Hype um Java aus. Es gab sogar die Vorstellung, daß dereinst komplette Office-Pakete als Applets realisiert werden und Arbeitsplatzrechner kaum noch eine eigene Festplatte brauchen, sondern alle Programme als Applets übers Netz holen.

Wer heutzutage mit einem nicht javafähigen Browser durchs Netz surft, wird feststellen, daß es so gut wie keine Webseite gibt, auf der sich ein Applet befindet. In dieser Hinsicht sind Applets als Technologie mehr oder weniger tot. Java hingegen hatte noch Jahre mit dem Vorurteil zu kämpfen, es sei lediglich eine Sprache um kleine Programme für den Browser zu schreiben und für große Programme nicht ernsthaft einzusetzen. Wie man sieht, hat sich mitlerweile Java als Programmiersprache etabliert, jedoch der ursprüngliche Türöffner, die Applets spielen keinerlei Rolle mehr.

Trotzdem soll in diesem Abschnitt einmal gezeigt werden, wie sich eine Applikation als Applet schreiben läßt.

12.3.1 Die Klasse JApplet

Um eine im Browser auszführbare Klasse zu schreiben, muß diese von der Klasse `java.applet.Applet` ableiten. Im Swingpaket gibt es die Klasse `javax.swing.JApplet`, die eine Unterklasse von `Applet` ist. Um also eine Klasse, die der Browser ausführen kann zu schreiben, sollte man von einer dieser Klassen ableiten. Da es sich bei `Applet` um eine Unterklasse von `Panel` handelt, können wir mit der bereits gewohnten Methode `add` weitere graphische Komponenten zu einem Applet hinzufügen.

Als Beispiel schreiben wir hierzu wieder die nun schon hinglänglich bekannte Klasse `Counter`. Dieses mal allerdings nicht direkt als `JFrame`, sondern als ein `JPanel`, das dann je nach Bedarf einem `JFrame`, einem `JApplet` oder einer anderen Komponente hinzugefügt werden kann.

CounterPanel.java

```
1
2  import javax.swing.*;
3  import java.awt.event.*;
4  public class  CounterPanel extends JPanel{
5    int i=0;
6    JButton b=new JButton("just Push");
7    JLabel l=new JLabel("  ");
8    public CounterPanel(){
9      b.addActionListener(new ActionListener(){
10       public void actionPerformed(ActionEvent e){
11         i=i+1;
12         l.setText(i+"");
13         repaint();
14       }
15     });
16     add(b);
17     add(l);
18   }
19 }
```

Diese Komponente soll jetzt einer `JApplet`-Klasse zugefügt werden. Hierzu kann die `add`-Methode von `JApplet` benutzt werden:

CountApplet.java

```
import javax.swing.*;
public class CountApplet extends JApplet{
  public CountApplet(){add(new CounterPanel());}
}
```

Es bleibt eine Webseite zu schreiben, in der diese Appletklasse gestartet wird. In dieser HTML-Seite ist der Tag `applet` einzufügen. Im Attribut `code` dieses Tags ist die Javaklasse anzugeben, welche das Applet enthält. Weitere Attribute definieren die Größe des Bereichs, den der Browser für das Applet vorsehen soll. Zusätzlich läßt sich noch definieren, in welchem Verzeichnis auf dem Webserver die Klassen zu finden sind. Folgende HTML-Seite startet schließlich obiges Applet.

ViewApplet.html

```
<html><body>
<h1>Ein erstes Applet</h1>
Dieses Applet zählt einen internen Zähler hoch, wenn der Knopf gedrückt wird.
<applet code="CountApplet.class"
        codebase="../classes/"
        width="600" height="60">
    <param name="maxwidth" value="120">
    <param name="nimgs" value="17">
    <param name="offset" value="-57">
Your browser is completely ignoring the &lt;APPLET&gt; tag!
</applet>
</body></html>
```

12.3.2 Methoden eines Applets

12.3.3 Sicherheit

12.3.4 Laden von weiteren Resourcen

12.4 Selbstdefinierte graphische Komponenten

Bisher graphische Komponenten unter Verwendung der fertigen GUI-Komponente aus der Swing-Bibliothek zusammengesetzt. Oft will man graphische Komponenten schreiben, für die es keine fertige GUI-Komponente in der Swing-Bibliothek gibt. Wir müssen wir eine entsprechende Komponente selbst schreiben.

12.4.1 Graphics Objekte

Um eine eigene GUI-Komponente zu schreiben, schreibt man eine Klasse, die von der GUI-Klasse ableitet. Dieses haben wir bereits in den letzten Beispielen getan, indem wir von der Klasse `JFrame` abgelitten haben. Die dort geschriebenen Unterklassen der Klasse `JFrame` zeichneten sich dadurch aus, daß sie eine Menge von graphischen Objekten (Knöpfe, Textfelder. . .)

in einer Komponente zusammengefasst haben. In diesem Abschnitt werden wir eine neue Komponente definieren, die keine der bestehenden fertigen Komponenten benutzt, sondern selbst alles zeichnet, was zu ihrer Darstellung notwendig ist.

Hierzu betrachten wir eine der entscheidenen Methoden der Klasse `JComponent`, die Methode `paintComponent`. In dieser Methode wird festgelegt, was zu zeichnen ist, wenn die graphische Komponente darzustellen ist. Die Methode `paintComponent` hat folgende Signatur:

```
public void paintComponent(java.awt.Graphics g)
```

Java ruft diese Methode immer auf, wenn die graphische Komponente aus irgendeinen Grund zu zeichnen ist. Dabei bekommt die Methode das Objekt übergeben, auf dem gezeichnet wird. Dieses Objekt ist vom Typ `java.awt.Graphics`. Es stellt ein zweidimensionales Koordinatensystem dar, in dem zweidimensionale Graphiken gezeichnet werden können. Der Nullpunkt dieses Koordinatensystems ist oben links und nicht unten links, wie wir es vielleicht aus der Mathematik erwartet hätten.

In der Klasse `Graphics` sind eine Reihe von Methoden definiert, die es erlauben graphische Objekte zu zeichnen. Es gibt Methoden zum Zeichnen von Geraden, Vierecken, Ovalen, beliebigen Polygonzügen, Texten etc.

Wollen wir eine eigene graphische Komponente definieren, so können wir die Methode `paintComponent` überschreiben und auf dem übergebenen Objekt des Typs `Graphics` entsprechende Methoden zum Zeichnen aufrufen. Um eine eigene graphische Komponente zu definieren, wird empfohlen die Klasse `JPanel` zu erweitern und in ihr die Methode `paintComponent` zu überschreiben.

Beispiel:
Folgende Klasse definiert eine neue graphische Komponente, die zwei Linien, einen Text, ein Rechteck, ein Oval und ein gefülltes Kreissegment enthält.

SimpleGraphics.java

```
package name.panitz.gui.graphicsTest;

import javax.swing.JPanel;
import javax.swing.JFrame;
import java.awt.Graphics;

class SimpleGraphics extends JPanel{
  public void paintComponent(Graphics g){
    g.drawLine(0,0,100,200);
    g.drawLine(0,50,100,50);
    g.drawString("hallo",10,20);
    g.drawRect(10, 10, 60,130);
    g.drawOval( 50,  100, 30, 80);
    g.fillArc(-20, 150, 80, 80, 0, 50);
  }
}
```

Diese Komponente können wir wie jede andere Komponente auch einem Fenster hinzufügen, so daß sie auf dem Bildschirm angezeigt werden kann.

UseSimpleGraphics.java

```
package name.panitz.gui.graphicsTest;

import javax.swing.JFrame;

class UseSimpleGraphics {
  public static void main(String [] args){
    JFrame frame = new JFrame();
    frame.getContentPane().add(new SimpleGraphics());

    frame.pack();
    frame.setVisible(true);
  }
}
```

Wird dieses Programm gestartet, so öfnet Java das Fenster aus Abbildung 12.4 auf dem Bildschirm.

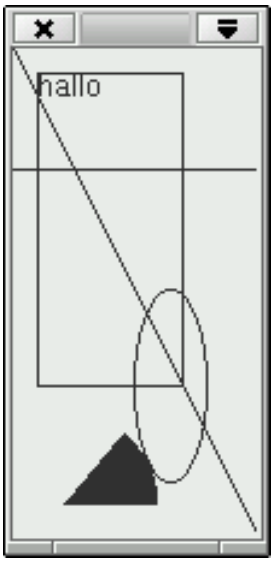


Abbildung 12.4: Einfache graphische Komponente.

Erst wenn wir das Fenster mit der Maus größer ziehen, können wir das ganze Bild sehen.

12.4.2 Dimensionen

Ärgerlich in unserem letzten Beispiel war, daß Java zunächst ein zu kleines Fenster für unsere Komponente geöffnet hat, und wir dieses Fenster mit Maus erst größer ziehen mußten. Die Klasse `JComponent` enthält Methoden, in denen die Objekte angeben können, welches ihre bevorzugte Größe bei ihrere Darstellung ist. Wenn wir diese Methode überschreiben, so daß sie eine Dimension zurückgibt, in der das ganze zu zeichnende Bild passt, so wird von Java

auch ein entsprechend großes Fenster geöffnet. Wir fügen der Klasse `SimpleGraphics` folgende zusätzliche Methode hinzu.

```
  public java.awt.Dimension getPreferredSize() {
    return new java.awt.Dimension(100,200);
  }
```

Jetzt öffnet Java ein Fenster, in dem das ganze Bild dargestellt werden kann.

Aufgabe 46 Schreiben Sie eine GUI-Komponente `StrichKreis`, die mit geraden Linien einen Kreis entsprechend untenstehender Abbildung malt. Der Radius des Kreises soll dabei dem Konstruktor der Klasse `StrichKreis` als `int`-Wert übergeben werden.

Testen Sie Ihre Klasse mit folgender Hauptmethode:

```
public static void main(String [] args) {
  StrichKreis k = new StrichKreis(120);
  JFrame f = new JFrame();
  JPanel p = new JPanel();
  p.add(new StrichKreis(120));
  p.add(new StrichKreis(10));
  p.add(new StrichKreis(60));
  p.add(new StrichKreis(50));
  p.add(new StrichKreis(70));
  f.getContentPane().add(p);

  f.pack();
  f.setVisible(true);
}
```

Hinweis: In der Klasse `java.lang.Math` finden Sie Methoden trigeometrischer Funktionen. Insbesondere `toRadians` zum Umrechnen von Gardwinkel in Bogenmaß sowie `sin` und `cos`.

12.4.3 Farben

In der graphische Komponente aus dem letzten Abschnitt haben wir uns noch keine Gedanken über die Farbe der gezeichneten Grafik gemacht. Die Klasse `Graphics` stellt eine Methode bereit, die es erlaubt die Farbe des Stiftes für die Grafik auf einen bestimmten Wert zu setzen. Alle anschließenden Zeichenbefehle auf das Objekt des Typs `Graphics` werden dann mit dieser Farbe vorgenommen, solange bis die Methode zum Setzen der Farbe ein weiteres Mal aufgerufen wird. Die Methode zum Setzen der Farbe auf einem Objekt der Klasse `Graphics` hat folgende Signatur:

```
public abstract void setColor(java.awt.Color c)
```

Zusätzlich gibt es für Komponenten noch eine Methode, die die Hintergrundfarbe der Komponente beschreibt.

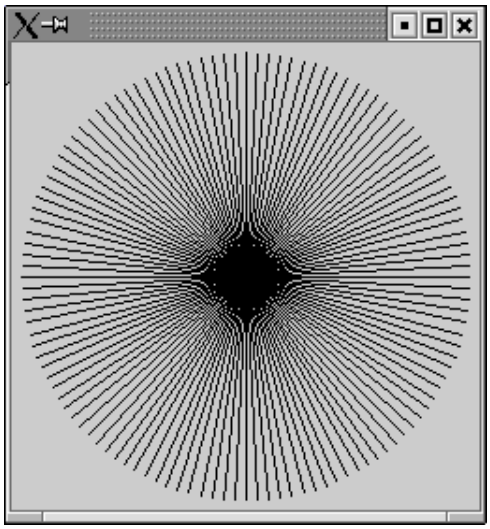


Abbildung 12.5: Kreis mit Durchmesserlinien.

Farben werden mit der Klasse `java.awt.Color` beschrieben. Hier gibt es mehrere Konstruktoren, um eine Farbe zu spezifizieren, z.B. einen Konstruktor, der Intensität der drei Grundfarben in einen Wert zwischen 0 und 255 festlegt.

Für bestimmte gängige Farben stellt die Klasse `Color` Konstanten zur Verfügung, die wir benutzen können, wie z.B.: `Color.RED` oder `Color.GREEN`.

Beispiel:
Wir können eine kleine Klasse schreiben, in der wir ein wenig mit Farben spielen. Wir setzen die Hintergrundfarbe des Objekts auf einen dunklen Grauwert, zeichnen darin ein dunkelgrünes Rechteck, ein rotes Oval und ein gelbes Oval.

ColorTest.java

```
1  package name.panitz.gui.graphicsTest;
2
3  import javax.swing.JPanel;
4  import javax.swing.JFrame;
5
6  public class ColorTest extends JPanel{
7    public ColorTest(){
8      //Hintergrundfarbe auf einen dunklen Grauton setzen
9      setBackground(new java.awt.Color(100,100,100));
10   }
11
12   public void paintComponent(java.awt.Graphics g){
13     //Farbe auf einen dunklen Grünton setzen
14     g.setColor(new java.awt.Color(22,178,100));
15     //Rechteck zeichnen
16     g.fillRect(25,50,50,100);
17     //Farbe auf rot setzen
18     g.setColor(java.awt.Color.RED);
19     g.fillOval(25,50,50,100);
```

```
20      //Farbe auf gelb setzen
21      g.setColor(java.awt.Color.YELLOW);
22      g.fillOval(37,75,25,50);
23    }
24
25    public java.awt.Dimension getPreferredSize() {
26      return new java.awt.Dimension(100,200);
27    }
28
29    public static void main(String [] _){
30      javax.swing.JFrame f = new JFrame("Farben");
31      f.getContentPane().add(new ColorTest());
32      f.pack();f.setVisible(true);
33    }
34  }
```

Das Programm ergibt das Bild aus Abbildung 12.6.

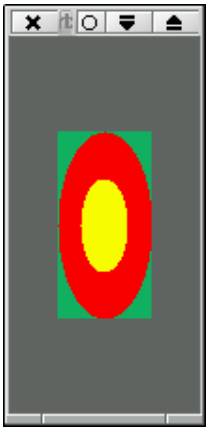


Abbildung 12.6: Erste farbige Graphik.

Aufgabe 47 Erweitern Sie die Strichkreisklasse aus der letzten Aufgabe, so daß jeder Strich in einer anderen Farbe gezeichnet wird.

Fraktale

Um noch ein wenig mit Farben zu spielen, zeichnen wir in diesem Abschnitt die berühmten Apfelmännchen. Apfelmännchen werden definiert über eine Funktion auf komplexen Zahlen. Die aus der Mathematik bekannten komplexen Zahlen sind Zahlen mit zwei reellen Zahlen als Bestandteil, den sogenannten Imaginärteil und den sogenannten Realteil. Wir schreiben zunächst eine rudimentäre Klasse zur Darstellung von komplexen Zahlen:

Complex.java

```
1  package name.panitz.crempel.tool.apfel;
2
```

```
3 public class Complex{
```

Diese Klasse braucht zwei Felder um Real- und Imaginärteil zu speichern:

Complex.java

```
4   public double re;
5   public double im;
```

Ein naheliegender Konstruktor für komplexe Zahlen füllt diese beiden Felder.

Complex.java

```
6  public Complex(double re,double im){
7   this.re=re;this.im=im;
8 }
```

Im Mathematikbuch schauen wir nach, wie Addition und Multiplikation für komplexe Zahlen definiert sind, und schreiben entsprechende Methoden:

Complex.java

```
9   public Complex add(Complex other){
10    return new Complex(re+other.re,im+other.im);
11  }
12
13  public Complex mult(Complex other){
14    return new Complex
15      (re*other.re-im*other.im,re*other.im+im*other.re);
16  }
```

Zusätzlich finden wir in Mathematik noch die Definition der Norm einer komplexen Zahl und setzen auch diese Definition in eine Methode um. Zum Quadrat des Realteils wird das Quadrat des Imaginärteils addiert.

Complex.java

```
17  public double norm(){return re*re+im*im;}
18 }
```

Soweit komplexe Zahlen, wie wir sie für Apfelmännchen brauchen.

Grundlage zum Zeichnen von Apfelmännchen ist folgende Iterationsgleichung auf komplexen Zahlen: $z_{n+1} = z_n^2 + c$. Wobei z_0 die komplexe Zahl $0 + 0i$ mit dem Real- und Imaginärteil 0 ist.

Zum Zeichnen der Apfelmännchen wird ein Koordinatensystem so interpretiert, daß die Achsen jeweils Real- und Imaginärteil von komplexen Zahlen darstellen. Jeder Punkt in diesem Koordinatensystem steht jetzt für die Konstante c in obiger Gleichung. Nun wir geprüft ob und für welches n die Norm von z_n größer eines bestimmten Schwellwertes ist. Je nach der Größe von $n$wird der Punkt im Koordinatensystem mit einer anderen Farbe eingefärbt.

Mit diesem Wissen können wir nun versuchen die Apfelmännchen zu zeichnen. Wir müssen nur geeignete Werte für die einzelnen Parameter finden. Wir schreiben eine eigene Klasse für das graphische Objekt, in dem ein Apfelmännchen gezeichnet wird. Wir deklarieren die Imports der benötigten Klassen:

Apfelmaennchen.java

```
package name.panitz.crempel.tool.apfel;

import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Apfelmaennchen extends JPanel {
```

Als erstes deklarieren wir Konstanten für die Größe des Apfelmännchens.

Apfelmaennchen.java

```
  final int width = 480;
  final int height = 430;
```

Eine weitere wichtige Konstante ist der Faktor, der angibt, welcher reellen Zahl ein Pixel entspricht:

Apfelmaennchen.java

```
  double zelle=0.00625;
```

Eine weitere Konstanten legt die Farbe fest, mit der die Punkte, die nicht über einen bestimmten Schwellwert konvergieren, eingefärbt werden sollen:

Apfelmaennchen.java

```
  final Color colAppleman = new Color(0,129,190);
```

Weitere Konstanten legen fest welche komplexe Zahl der Nullpunkt unseres `Graphics`-Objekts darstellt.

Apfelmaennchen.java

```
  double startX = -2;
  double startY = -1.35;
```

Weitere Konstanten sind der Schwellwert und die maximale Rekursionstiefe n, für die wir jeweils z_n berechnen:

Apfelmaennchen.java

```
  final int recDepth = 50;
  final int schwellwert = 4;
```

Die wichtigste Methode berechnet die Werte für die Gleichung $z_{n+1} = z_n^2 + c$. Der Eingabeparameter ist die komplexe Zahl c. Das Ergebnis dieser Methode ist das n, für das z_n größer als der Schwellwert ist:

Apfelmaennchen.java

```
18  //C-Werte checken nach zn+1 = zn*zn + c,
19  public int checkC(Complex c) {
20    Complex zn = new Complex(0,0);
21
22    for (int n=0;n<recDepth;n=n+1) {
23      final Complex znp1 = zn.mult(zn).add(c);
24      if (znp1.norm() > schwellwert) return n;
25      zn=znp1;
26    }
27    return recDepth;
28  }
```

Jetzt gehen wir zum Zeichnen jedes Pixel unseres `Graphics`-Objekts durch, berechnen welche komplexe Zahl an dieser Stelle steht und benutzen dann die Methode *checkC*, um zu berechnen ob und nach wieviel Iterationen die Norm von z_n größer als der Schwellwert wird. Abhängig von dieser Zahl, färben wir den Punkt mit einer Farbe ein.

Apfelmaennchen.java

```
29  public void paint(Graphics g) {
30    for (int y=0;y<height;y=y+1) {
31      for (int x=0;x<width;x=x+1) {
32
33        final Complex current
34          =new Complex(startX+x*zelle,startY+y*zelle);
35
36        final int iterationenC = checkC(current);
37
38        paintColorPoint(x,y,iterationenC,g);
39      }
40    }
41  }
```

Zur Auswahl der Farbe benutzen wir folgende kleine Methode, die Abhängig von ihrem Parameter `it` an der Stelle `(x,y)` einen Punkt in einer bestimmten Farbe zeichnet.

Apfelmaennchen.java

```
42  private void paintColorPoint
43                 (int x,int y,int it,Graphics g){
44    final Color col
45      = it==recDepth
46        ?colAppleman
47        :new Color(255-5*it%1,255-it%5*30,255-it%5* 50);
48    g.setColor(col);
49    g.drawLine(x,y,x,y);
50  }
```

Schließlich können wir noch die Größe festlegen und das Ganze in einer Hauptmethode starten:

Apfelmaennchen.java

```
51    public Dimension getPreferredSize(){
52      return new Dimension(width,height);
53    }
54
55    public static void main(String [] args){
56      JFrame f = new JFrame();
57      f.getContentPane().add(new Apfelmaennchen());
58      f.pack();
59      f.setVisible(true);
60    }
61  }
```

Das Programm ergibt das Bild aus Abbildung 12.7.

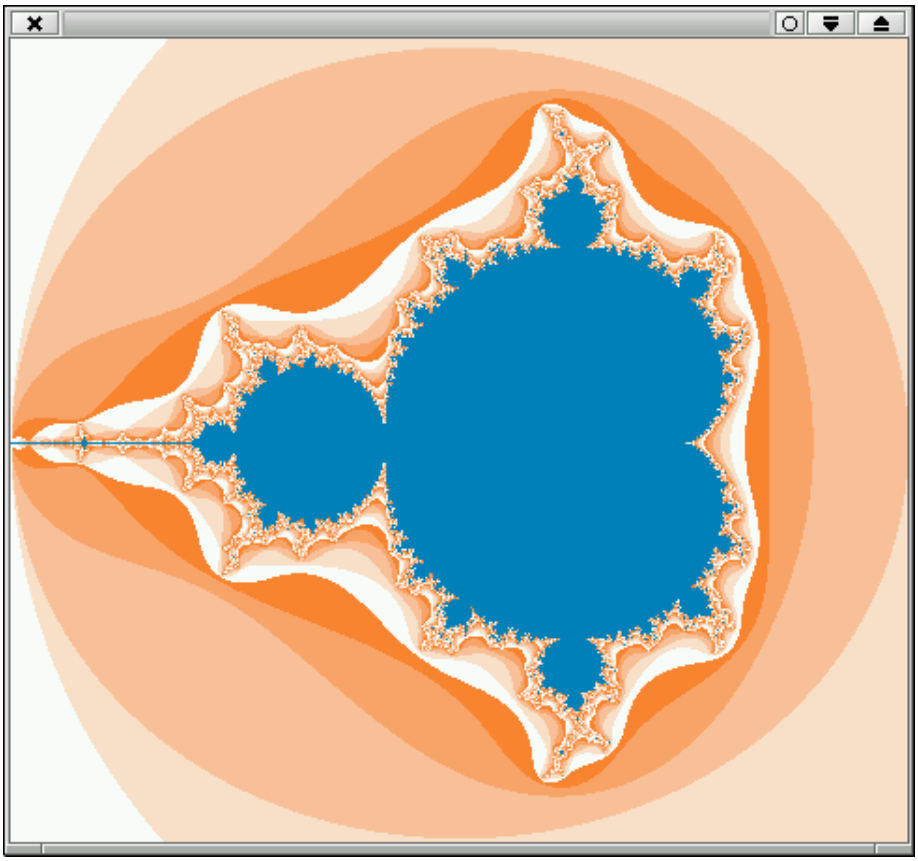


Abbildung 12.7: Apfelmännchen.

12.4.4 Fonts und ihre Metrik

Ähnlich wie für Farben hat ein Objekt des Typs `Graphics` auch einen aktuellen Font. Dieser ist vom Typ `java.awt.Font`. Ebenso wie es die Methoden `getColor` und `setColor` in der Klasse

`Graphics` gibt, gibt es dort auch Methoden `getFont` und `setFont`.

Eine wichtige Information zu einem Font ist beim Zeichnen, wieviel Platz ein bestimmter Text für in diesem Font benötigt. Hierzu existiert eine Klasse `java.awt.FontMetrics`. Entsprechend gibt es Methoden, die für ein `Graphics`-Objekt ein Objekt mit den aktuellen `FontMetrics`-Objekt zurückgibt.

Beispiel:
Folgende Kleine Klasse vergrößert nach und nach den Font, und malt in verschiedenen Größen einen Text in ein Fenster.

FontTest.java

```
1  package name.panitz.gui.graphicsTest;
2
3  import java.awt.Font;
4  import java.awt.FontMetrics;
5  import java.awt.Graphics;
6  import java.awt.Dimension;
7  import javax.swing.JPanel;
8  import javax.swing.JFrame;
9
10
11 class FontTest extends JPanel {
12
13   public Dimension getPreferredSize(){
14     return new Dimension(250,650);
15   }
16
17   public void paintComponent(Graphics g){
18     //die y-Koordinate für einen Text
19     int where = 0;
20
21     //25 mal
22     for (int i=1; i<=25;i=i+1){
23       //nimm aktuellen Font und seine Maße
24       final Font f = g.getFont();
25       final FontMetrics fm = g.getFontMetrics(f);
26
27       //gehe die Höhe des Fonts weiter in Richtung x
28       where = where + (f.getSize()) ;
29
30       //male "hallo"
31       g.drawString("hallo",0,where);
32
33       //berechne wo der gerade gemalte String endet
34       //und male dort "welt"
35       g.drawString
36         ("welt!",fm.stringWidth("hallo "),where);
37
38       //verändere die Fontgröße. Erhöhe sie um 1.
39       g.setFont(getFont().deriveFont(f.getSize2D()+1));
40     }
```

```
41    }
42
43    public static void main(String [] _){
44      JFrame f = new JFrame("Fonts");
45      f.getContentPane().add(new FontTest());
46      f.pack();
47      f.setVisible(true);
48    }
49  }
```

Das Programm ergibt das Bild aus Abbildung 12.8.

12.4.5 Erzeugen graphischer Dateien

Bisher haben wir alles auf dem Bildschirm in irgendeinen Fenster gezeichnet. Das Schöne ist, daß wir die gleichen graphischen Objekte jetzt benutzen können, um Graphiken in eine Datei zu schreiben.

Hierzu stellt Java die Klasse `java.awt.image.RenderedImage` zur Verfügung. Für Objekte dieser Klasse gibt es ein Objekt des Typs `Graphics`. Man kann also ein `RenderedImage`-Objekt erzeugen und mit beliebige `paint`-Methoden auf dessen `Graphics`-Objekt anwenden.

In der Klasse `javax.imageio.ImageIO` gibt es schließlich Methoden, die es erlauben ein `RenderedImage`in eine Datei zu schreiben.

Beispiel:
Wir schreiben eine kleine Klasse mit Methoden, um das Bild einer graphischen Komponente in eine Datei zu speichern.

Zunächst brauchen wir eine ganze Reihe von Klassen, die wir importieren:

ComponentToFile.java

```
1  package name.panitz.gui.graphicsFile;
2
3  import name.panitz.crempel.tool.apfel.Apfelmaennchen;
4  import java.awt.image.BufferedImage;
5  import java.awt.image.RenderedImage;
6  import java.awt.Dimension;
7  import java.awt.Graphics;
8
9  import javax.imageio.ImageIO;
10 import javax.swing.JComponent;
11
12 import java.io.File;
13 import java.io.IOException;
```

Wir schreiben eine Hauptmethode, die als Kommandozeilenparameter den Namen der zu schreibenen Bilddatei übergeben bekommt:

ComponentToFile.java

```
14 public class ComponentToFile {
15   public static void main(String [] args){
```

Abbildung 12.8: Verschiedene Fontgrößen.

Zu Testzwecken schreiben wir je eine `png`- und eine `jpg`-Datei, in die wir die Komponente zeichnen wollen:

ComponentToFile.java

```
16    try {
17      final JComponent component = new Apfelmaennchen();
18      final RenderedImage image = createComponentImage(component);
19      ImageIO.write(image,"png", new File(args[0]+".png"));
```

```
20      ImageIO.write(image,"jpg", new File(args[0]+".jpg"));
21     }catch (IOException e) {System.out.println(e);}
22   }
```

Die folgende Methode erzeugt das eigentliche Bild der Komponente:

ComponentToFile.java

```
23   static public RenderedImage createComponentImage
24                                  (JComponent component) {
```

Wir erzeugen ein Bildobjekt in der für die Komponente benötigten Größe.

ComponentToFile.java

```
25      Dimension d = component.getPreferredSize();
26      BufferedImage bufferedImage
27       = new BufferedImage((int)d.getWidth(),(int)d.getHeight()
28                          ,BufferedImage.TYPE_INT_RGB);
```

Nun besorgen wir dessen `Graphics`-Umgebung:

ComponentToFile.java

```
29     Graphics g = bufferedImage.createGraphics();
```

Darin läßt sich unsere Komponente zeichnen:

ComponentToFile.java

```
30    component.paint(g);
```

Die Methode kann das fertig gezeichnete Bild als Ergebnis zurückgeben:

ComponentToFile.java

```
31     return bufferedImage;
32   }
33 }
```

12.4.6 Graphics2D

Seit Java 1.2 existiert eine Unterklasse der Klasse `Graphics` mit wesentlich mächtigeren Mehoden zum Zeichnen von 2-Dimenionalen Graphiken, die Klasse `java.awt.Graphics2D`. Leider konnten nicht die Signaturen der entsprechenden Methoden `paint` in den Komponentenklassen geändert werden, so daß dem übergebene `Graphics`-Objekt zunächst zugesichert werden muß, daß es sich um ein `Graphics2D`-Objekt handelt:

```
1 public void paint(Graphics g) {
2     Graphics2D g2 = (Graphics2D) g;
3     ...
4 }
```

`Graphics2D` enthält Methoden zum Darstellen verschiedener Formen, wahlweise umrandet oder gefüllt. Auf geometrische Formen können Transformationen angewendet werden, einzelne Formen können kombiniert werden zu komplexeren Figuren, Farbverläufe können definiert werden. Auch Text stellt sich hierbei als geometrische Form dar.

Beispiel:
Folgende Klasse benutzt willkürlich ein paar Eigenschaften der `Graphics2D`-Klasse, um einen Eindruck über die Arbeitsweise dieser Klasse zu geben. Eine systematische Behandlung aller graphischen Methoden ist nicht Gegenstand dieser Vorlesung.

G2DTest.java

```
package name.panitz.gui.g2d;
import javax.swing.JPanel;
import javax.swing.JFrame;

import java.awt.font.*;
import java.awt.*;
import java.awt.geom.*;

public class G2DTest extends JPanel{
  int w = 550;
  int h = 650;

  public Dimension getPreferredSize(){
    return new Dimension(w,h);}

  public void paintComponent(Graphics g){
    Graphics2D g2 = (Graphics2D) g;
    TextLayout textTl
     = new TextLayout
             ("Kommunismus", new Font("Helvetica", 1, 96)
             , new FontRenderContext(null, false, false));

    AffineTransform textAt = new AffineTransform();
    textAt.translate(0,(float)textTl.getBounds().getHeight());
    Shape shape = textTl.getOutline(textAt);

    AffineTransform at = new AffineTransform();
    at.rotate(Math.toRadians(45));
    at.shear(1.5, 0.0);

    g2.transform(at);

    float dash[] = {10.0f};
    g2.setStroke(
      new BasicStroke
           (3.0f, BasicStroke.CAP_BUTT
           ,BasicStroke.JOIN_MITER, 10.0f, dash, 0.0f)
    );

    g2.setPaint(
      new GradientPaint(0,0,Color.black,w,h
                        ,new Color(255,100,100),false)
    );

    g2.fill(shape);
```

```
46      g2.setColor(Color.darkGray);
47      g2.draw(shape);
48    }
49
50    public static void main(String [] _){
51      JFrame f = new JFrame("G2");
52      f.getContentPane().add(new G2DTest());
53      f.pack();
54      f.setVisible(true);
55    }
56  }
```

Das Programm öffnet das Fenster aus Abbildung 12.9 auf dem Bildschirm.

12.5 Bäume graphisch darstellen

12.5.1 Bäume mit JTree graphisch darstellen

Wir haben uns in dieser Vorlesung ausgiebig mit Bäumen beschäftigt, sie aber bisher nicht graphisch sondern lediglich textuell dargestellt. In diesem Kapitel sollen zwei Möglichkeiten zur graphischen Darstellung von Bäumen in Java untersucht werden. Zunächst benutzen wir eine bereits in der `swing`-Bibliothek angebotene Komponente zur Baumdarstellung, anschließend definieren wir eine eigene neue Komponente zur Baumdarstellung.

Im Paket `javax.swing` gibt es eine Klasse, die es erlaubt beliebige Baumstrukturen graphisch darzustellen, die Klasse `JTree`. Diese Klasse übernimmt die graphische Darstellung eines Baumes. Der darzustellende Baum ist dieser Klasse zu übergeben. Hierzu hat sie einen Konstruktor, der einen Baum erhält: `JTree(TreeNode root)`.

Um entsprechend die Klasse `JTree` benutzen zu können, so daß sie uns einen Baum darstellt, müssen wir einen Baum, der die Schnittstelle `javax.swing.tree.TreeNode` implementiert, zur Verfügung stellen. Diese Schnittstelle hat 7 relativ naheliegende Methoden, die sich in ähnlicher Form fast alle in unserer Klasse `Tree` auch schon befinden:

```
1  package javax.swing.tree;
2  interface TreeNode{
3    Enumeration children();
4    int getChildCount();
5    TreeNode getParent();
6    boolean isLeaf();
7    TreeNode getChildAt(int childIndex);
8
9    boolean getAllowsChildren();
10   int getIndex(TreeNode node);
11 }
```

Wir können unsere Klasse `Tree` entsprechend modifizieren, damit sie diese Schnittstelle implementiert. Hierzu fügen wir Implementierungen der in der Schnittstelle verlangten Methoden der

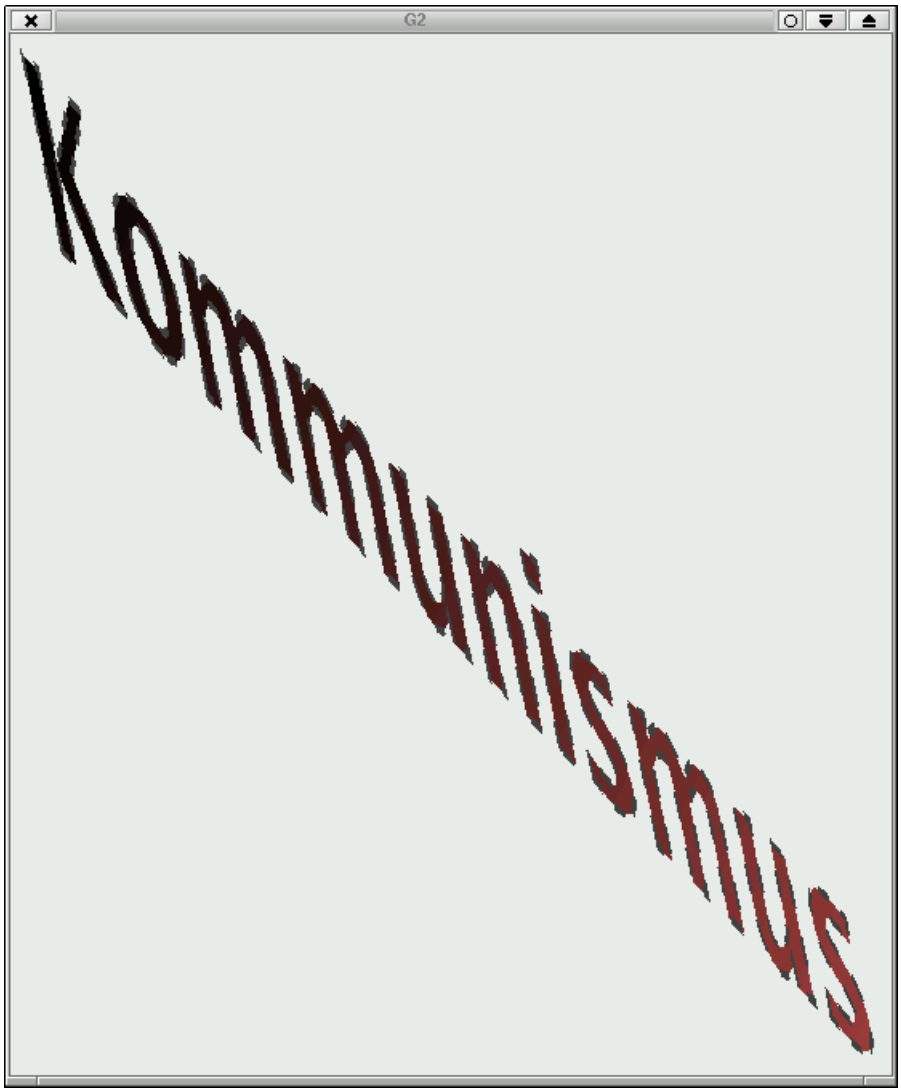


Abbildung 12.9: Transformationen auf Text in Graphics2D.

Klasse `Tree` hinzu und sorgen so dafür, daß unsere Bäume alle Eigenschaften haben, so daß die Klasse `JTree` sie graphisch darstellen können.

```
1 public class Tree<a>implements TreeNode{
```

Entscheidend ist natürlich die Methode `children`, die inhaltlich unserer Methode `theChildren` entspricht. Leider benutzt die Schnittstelle als Rückgabetyp eine weitere Klasse aus dem Paket `java.util`. Dieses ist die Schnittstelle `Enumeration`. Logisch ist sie fast funktionsgleich mit der Schnittstelle `Iterator`. Die Existenz dieser zwei sehr ähnlichen Schnittstellen hat historische Gründe. Mit einer anonymen inneren Klassen können wir recht einfach ein Rückgabeobjekt der Schnittstelle erzeugen.

Tree.java

```
  public Enumeration<TreeNode> children() {
    final Iterator<Tree<a>> it = theChildren().iterator();

    return new Enumeration<TreeNode>(){
      public boolean hasMoreElements(){return it.hasNext();}
      public Tree<a> nextElement() {return it.next();}
    };
  }
```

Die Methoden `getChildCount`, `getParent`, `isLeaf` `getChildAt` existieren bereits oder sind einfach aus der bestehenden Klasse ableitbar.

Tree.java

```
  public TreeNode getChildAt(int childIndex) {
    return theChildren().get(childIndex);
  }

  public int getChildCount(){return theChildren().size();}

  public TreeNode getParent(){return parent;}
```

Die übrigen zwei Methoden aus der Schnittstelle `TreeNode` sind für unsere Zwecke unerheblich. Wir implementieren sie so, daß sie eine Ausnahme werfen.

Tree.java

```
  public boolean getAllowsChildren(){
    throw new UnsupportedOperationException();
  }

  public int getIndex(TreeNode node) {
    throw new UnsupportedOperationException();
  }
```

Schließlich ändern wir die Methode `toString` so ab, daß sie nicht mehr eine `String`-Repräsentation des ganzen Baumes ausgibt, sondern nur des aktuellen Knotens:

Tree.java

```
  public String toString(){return mark().toString();}}
```

Somit haben wir unsere Klasse `Tree` soweit massiert, daß sie von der Klasse `JTree` als Baummodell genutzt werden kann.

Beispiel:
Wir können jetzt ein beliebiges Objekt der Klasse `Tree` der Klasse `JTree` übergeben, um es graphisch darzustellen. Der Code hierzu ist denkbar einfach.

JTreeTest.java

```
package name.panitz.data.tree.example;

```

```
3  import javax.swing.JFrame;
4  import javax.swing.JTree;
5
6  public class JTreeTest {
7    public static void main(String [] args){
8      JFrame frame = new JFrame("Baumtest Fenster");
9      frame.getContentPane()
10           .add(new JTree(SkriptTree.getSkript()));
11
12     frame.pack();
13     frame.setVisible(true);
14   }
15 }
```

Diese Methode erzeugt die graphische Baumdarstellung aus Abbildung 12.10.

Dieser Baum reagiert dabei sogar auf Mausaktionen, in dem durch Klicken auf Baumknoten deren Kinder ein- und ausgeblendet werden können.

Die Klasse `JTree` verfügt noch über eine große Anzahl weiterer Funktionalitäten, die wir hier nicht im Detail betrachten wollen.

12.5.2 Bäume zeichnen

Im letzem Unterkapitel haben wir eine fertige Swingkomponente benutzt um Bäume darzustellen. In diesem Abschnitt werden wir eine eigene Komponente schreiben, die Bäume graphisch darstellt. Dabei soll ein Baum in der Weise gezeichnet werden, wir wir es handschriftlich gerne machen: oben zentriert liegt die Wurzel und Linien führen zu den weiter unten hingeschriebenen Kindern.

Die Hauptschwierigkeit hierbei wird sein, die einzelnen Positionen der Baumknoten zu berechnen.

DisplayTree.java

```
1  package name.panitz.data.tree;
2
3  import java.util.Enumeration;
4
5  import javax.swing.JComponent;
6  import javax.swing.JPanel;
7  import javax.swing.tree.TreeNode;
8
9  import java.awt.Font;
10 import java.awt.FontMetrics;
11 import java.awt.Color;
12 import java.awt.Graphics;
13 import java.awt.Dimension;
14
15 public class DisplayTree extends JPanel{
```

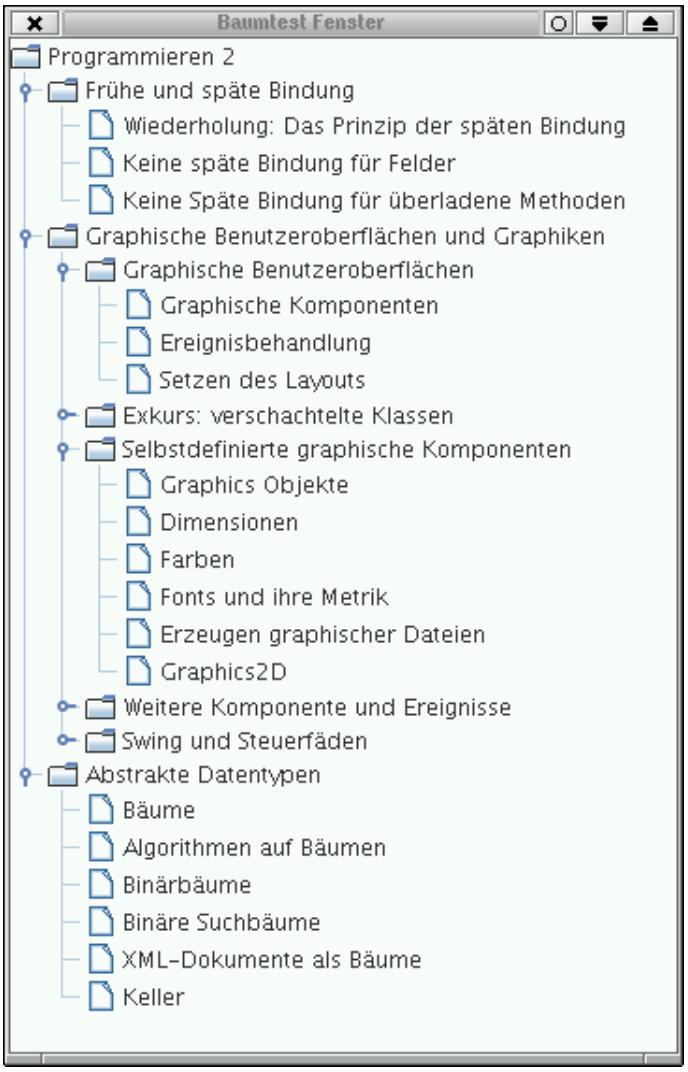


Abbildung 12.10: Baumdarstellung mit JTree.

Ähnlich wie in der Klasse `JTree` enthält das Objekt, das einen Baum graphisch darstellt, ein Baummodell:

DisplayTree.java

```
16  private TreeNode treeModell;
```

In zwei Konstanten sei spezifiziert, wieviel Zwischenraum horizontal zwischen Geschwistern und vertikal zwischen Eltern und Kindern liegen soll.

DisplayTree.java

```
17  static final public int VERTICAL_SPACE = 50;
18  static final public int HORIZONTAL_SPACE = 20;
```

In einem Feld werden wir uns die Maße des benutzten Fonts vermerken. Dieses ist wichtig, um die Größe eines Baumknotens zu errechnen.

DisplayTree.java

```
19    private FontMetrics fontMetrics = null;
```

Die Felder für Höhe und Breite werden die Maße der Baumzeichnung wiederspiegeln.

DisplayTree.java

```
20  private int height=0;
21  private int width=0;
```

Diese Maße sind zu berechnen, abhängig vom Font der benutzt wird. Eine bool'scher Wert wird uns jeweils angeben, ob die Maße bereits berechnet wurden.

DisplayTree.java

```
22  private boolean dimensionCalculated = false;
```

Es folgen zwei naheliegende Konstruktoren.

DisplayTree.java

```
23  public DisplayTree(TreeNode tree){treeModell=tree;}
24
25  public DisplayTree(TreeNode t,FontMetrics fm){
26    treeModell=t;fontMetrics=fm;
27  }
```

Die schwierigste Methode berechnet, wie groß der zu zeichnende Baum wird:

DisplayTree.java

```
28  private void calculateDimension(){
```

Zunächst stellen wir sicher, daß die Fontmaße bekannt sind:

DisplayTree.java

```
29
30    if (fontMetrics==null){
31      final Font font = getFont();
32      fontMetrics = getFontMetrics(font);
33    }
```

Die Breite der Knotenmarkierung wird berechnet. Dieses ist die minimale Breite des Gesamtbaums. Zunächst berechnen wir die Breite des Wurzelknotens:

DisplayTree.java

```
34    final int x
35      = fontMetrics.stringWidth(treeModell.toString());
```

Für jedes Kind berechnen wir nun die Breite und Höhe. Die maximale Höhe der Kinder bestimmt die Höhe des Gesamtbaums, die Summe aller Breiten der Kinder, bestimmt die Breite des Gesamtbaums.

DisplayTree.java

```
    final Dimension childrenDim = childrenSize();
    final int childrenX = (int)childrenDim.getWidth();
    final int childrenY = (int)childrenDim.getHeight();
```

Mit der Höhe und Breite der Kinder, läßt sich die Höhe und Breite des Gesamtbaums berechnen:

DisplayTree.java

```
    width=x>childrenX?x:childrenX;

    height
      =  childrenY == 0
        ?fontMetrics.getHeight()
        :VERTICAL_SPACE+childrenY;

    dimensionCalculated = true;
  }
```

Zum Berechnen der Größe aller Kinder haben wir oben eine Methode angenommen, die wir hier implementieren.

DisplayTree.java

```
  Dimension childrenSize(){
    int x = 0;
    int y = 0;

    final Enumeration<TreeNode> it=treeModell.children();
    while (it.hasMoreElements()){
      final DisplayTree t
       = new DisplayTree(it.nextElement(),fontMetrics);

      y =  y > t.getHeight()?y:t.getHeight();

      x=x+t.getWidth();

      if (it.hasMoreElements())  x=x+HORIZONTAL_SPACE;
    }
    return new Dimension(x,y);
  }
```

Mehrere Methoden geben die entsprechende Größe aus, nachdem sie gegebenenfalls vorher ihre Berechnung angestoßen haben:

DisplayTree.java

```
  public int getHeight(){
    if (!dimensionCalculated) calculateDimension();
    return height;
  }

```

```
70    public int getWidth(){
71      if (!dimensionCalculated) calculateDimension();
72      return width;
73    }
74
75    public Dimension getSize(){
76      if (!dimensionCalculated) calculateDimension();
77      return new Dimension(width,height);
78    }
79
80    public Dimension getMinimumSize(){
81      return getSize();
82    }
83
84    public Dimension getPreferredSize(){
85      return getSize();
86    }
```

Schließlich die eigentliche Methode zum Zeichnen des Baumes. Hierbei benutzen wir eine Hilfsmethode, die den Baum an eine bestimmte Position auf der Leinwand plaziert:

DisplayTree.java

```
87    public void paintComponent(Graphics g){paintAt(g,0,0);}
88
89    public void paintAt(Graphics g,int x,int y){
90      fontMetrics = g.getFontMetrics();
91      final String marks =treeModell.toString();
```

Zunächst zeichne die Baummarkierung für die Wurzel des Knotens. Zentriere diese in bezug auf die Breite des zu zeichnenden Baumes:

DisplayTree.java

```
92      g.drawString
93       (marks
94       ,x+(getWidth()/2-fontMetrics.stringWidth(marks)/2)
95       ,y+10);
```

Jetzt sind die Kinder zu zeichenen. Hierzu ist zu berechnen, wo ein Kind zu positionieren ist. Zusätzlich ist die Kante von Elternknoten zu Kind zu zeichnen. Hierzu merken wir uns den Startpunkt der Kanten beim Elternknoten. Auch dieser ist auf der Breite zentriert:

DisplayTree.java

```
96      final int startLineX = x+getWidth()/2;
97      final int startLineY = y+10;
```

Für jedes Kind ist jetzt die x-Position zu berechnen, das Kind zu zeichnen, und die Kante zum Kind zu zeichenen:

DisplayTree.java

```
    final Enumeration it=treeModell.children();

    final int childrenWidth = (int)childrenSize().getWidth();

    //wieviel nach rechts zu rücken ist
    int newX
      = getWidth()>childrenWidth?(getWidth()-childrenWidth)/2:0;

    //die y-Koordinate der Kinder
    final int nextY = y+VERTICAL_SPACE;

    while (it.hasMoreElements()){
      DisplayTree t
       = new DisplayTree((TreeNode)it.nextElement());

      //x-Positionen für das nächste Kind
      final int nextX = x+newX;

      //zeichne das Kind
      t.paintAt(g,nextX,nextY);

      //zeichne Kante
      g.drawLine(startLineX,startLineY
                ,nextX+t.getWidth()/2,nextY);

      newX = newX+t.getWidth()+HORIZONTAL_SPACE;
    }
  }
}
```

Damit ist die Klasse zur graphischen Darstellung von Bäumen komplett spezifiziert. Folgendes kleine Programm benutzt unsere Klasse `DisplayTree` in der gleichen Weise, wie wir auch die Klasse `JTree` benutzt haben. Das optische Ergebnis ist in Abbildung 12.11 zu bewundern.

DisplayTreeTest.java

```
package name.panitz.data.tree.example;
import name.panitz.data.tree.DisplayTree;
import javax.swing.JFrame;
import javax.swing.JScrollPane;

class DisplayTreeTest {
  public static void main(String [] args){
    JFrame frame = new JFrame("Baum Fenster");
    frame.getContentPane()
         .add(new JScrollPane(
                new DisplayTree(SkriptTree.getSkript())));

    frame.pack();
    frame.setVisible(true);
```

```
  }
}
```

Abbildung 12.11: Baum graphisch dargestellt mit DisplayTree.

Wir schreiben eine kleine Klasse mit Methoden, um das Bild eines Baumes in eine Datei zu speichern.

Zunächst brauchen wir eine ganze Reihe von Klassen, die wir importieren:

TreeToFile.java

```
package name.panitz.data.tree;
import name.panitz.gui.graphicsFile.ComponentToFile;
import static name.panitz.gui.graphicsFile.ComponentToFile.*;

import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.swing.JComponent;
import java.awt.image.RenderedImage;

public class TreeToFile{
  public static void treeToFile(Tree tree,String fileName){
    try {
     final JComponent component = new DisplayTree(tree);
     final RenderedImage image = createComponentImage(component);
     ImageIO.write(image,"png", new File(fileName+".png"));
     ImageIO.write(image,"jpg", new File(fileName+".jpg"));
    }catch (IOException e) {System.out.println(e);}
  }
}
```

12.6 Weitere Komponente und Ereignisse

Im Prinzip haben wir bisher alles kennengelernt um graphische Benutzerflächen und Anwendungen zu beschreiben. Die Schwierigkeit bei graphischen Anwendungen liegt hauptsächlich in der

schieren Menge der unterschiedlichen Komponenten und ihrem Zusammenspiel. Die Anzahl der vordefinierten Komponenten der Swing-Bibliothek ist immens. Diese Komponenten können auf eine Vielzahl von unterschiedlichen Ereignisarten reagieren. Bisher haben wir nur die Schnittstelle `ActionListener` kennengelernt, die lediglich ganz allgemein ein Ereignis behandelt.

In diesem Abschnitt schauen wir uns ein paar wenige weitere Komponenten und Ereignistypen an.

12.6.1 Mausereignisse

Ein in modernen graphischen Oberflächen häufigst benutztes Eingabemedium ist die Maus. Zwei verschiedene Ereignisarten sind für die Maus relevant:

- Mausereignisse, die sich auf das Drücken, Freilassen oder Klicken auf einen der Mausknöpfe bezieht. Hierfür gibt es eine Schnittstelle zur Behandlung solcher Ereignisse: `MouseListener`.
- Mausereignisse, die sich auf das Bewegen der Maus beziehen. Die Behandlung solcher Ereignisse kann über eine Implementierung der Schnittstelle `MouseMotionListener` spezifiziert werden.

Entsprechend gibt es für graphische Komponenten Methoden, um solche Mausereignisbehandler der Komponente hinzuzufügen: `addMouseListener` und `addMouseMotionListener`.

Um die Arbeit mit Ereignisbehandlern zu vereinfachen, gibt es für die entsprechnden Schnittstellen im Paket `java.awt.event` prototypische Implementierungen, in denen die Methoden der Schnittstelle so implementiert sind, daß ohne Aktion auf die entsprechenden Ereignisse reagiert wird. Diese prototypischen Implementierung sind Klassen, deren Namen mit *Adapter* enden. So gibt es zur Schnittstelle `MouseListener` die implementierende Klasse `MouseAdapter`. Will man eine bestimmte Mausbehandlung programmieren, reicht es aus, diesen Adapter zu erweitern und nur die Methoden zu überschreiben, für die bestimmte Aktionen vorgesehen sind. Es erübrigt sich dann für alle sechs Methoden der Schnittstelle `MouseListener` Implementierungen vorzusehen.

Beispiel:
Wir erweitern die Klasse `Apfelmaennchen` um eine Mausbehandlung. Der mit gedrückter Maus markierte Bereich soll vergrößert in dem Fenster dargestellt werden.

ApfelWithMouse.java

```
1  package name.panitz.crempel.tool.apfel;
2
3  import java.awt.Graphics;
4  import java.awt.event.*;
5  import javax.swing.JFrame;
6
7  public class ApfelWithMouse extends Apfelmaennchen{
8    public ApfelWithMouse(){
```

Im Konstruktor fügen wir der Komponente eine Mausbehandlung hinzu. Der Mausbehandler merkt sich die Koordinaten, an denen die Maus gedrückt wird und berechnet beim Loslassen des Mausknopfes den neuen darzustellenden Zahlenbereich:

ApfelWithMouse.java

```
9     addMouseListener(new MouseAdapter(){
10      int mouseStartX=0;
11      int mouseStartY=0;
12
13      public void mousePressed(MouseEvent e) {
14        mouseStartX=e.getX();
15        mouseStartY=e.getY();
16      }
17
18      public void mouseReleased(MouseEvent e) {
19        int endX = e.getX();
20        int endY = e.getY();
21        startX = startX+(mouseStartX*zelle);
22        startY = startY+(mouseStartY*zelle);
23        zelle  = zelle*(endX-mouseStartX)/width;
24        repaint();
25      }
26    });
27  }
```

Auch für diese Klasse sehen wir eine kleine Startmethode vor:

ApfelWithMouse.java

```
28  public static void main(String [] _){
29    JFrame f = new JFrame();
30    f.getContentPane().add(new ApfelWithMouse());
31    f.pack();
32    f.setVisible(true);
33  }
34 }
```

12.6.2 Fensterereignisse

Auch für Fenster in einer graphischen Benutzeroberfläche existieren eine Reihe von Ereignissen. Das Fenster kann minimiert oder maximiert werden, es kann das aktive Fenster oder im Hintergrund sein und es kann schließlich auch geschlossen werden. Um die Reaktion auf solche Ereignisse zu spezifizieren existiert die Schnittstelle `WindowListener` mit entsprechender prototypischer Adapterklasse `WindowAdapter`. Die Objekte der Fensterereignisbehandlung können mit der Methode `addWindowListener` Fensterkomponenten hinzugefügt werden.

Beispiel:
In den bisher vorgestellten Programmen wird Java nicht beendet, wenn das einzige Fenster der Anwendung geschlossen wurde. Man kann an der Konsole sehen, daß der Javainterpreter weiterhin aktiv ist. Das liegt daran, daß wir bisher noch nicht spezifiziert haben, wie die Fensterkomponenten auf das Ereignis des Schließens des Fensters reagieren sollen. Dieses kann mit einem Objekt, das `WindowListener` implementiert in der Methode `windowClosing` spezifiziert werden. Wir schreiben hier eine Version des Apfelmännchenprogramms, in dem das Schließen des Fensters den Abbruch des gesammten Programms bewirkt.

ClosingApfelFrame.java

```
package name.panitz.crempel.tool.apfel;

import javax.swing.JFrame;
import java.awt.event.*;

public class ClosingApfelFrame {
  public static void main(String [] args){
    JFrame f = new JFrame();
    f.getContentPane().add(new ApfelWithMouse());
    f.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    f.pack();
    f.setVisible(true);
  }
}
```

12.6.3 Weitere Komponenten

Um einen kleinen Überblick der vorhandenen Swing Komponenten zu bekommen, können wir ein kleines Programm schreiben, daß möglichst viele Komponenten einmal instanziiert. Hierzu schreiben wir eine kleine Testklasse:

ComponentOverview.java

```
package name.panitz.gui.example;

import javax.swing.*;
import java.awt.*;
import java.util.*;

public class ComponentOverview  {
  public ComponentOverview(){
```

Darin definieren wir eine Reihung von einfachen Swing-Komponenten:

ComponentOverview.java

```
   JComponent [] cs1 =
    {new JButton("knopf")
    ,new JCheckBox("check mich")
    ,new JRadioButton("drück mich")
    ,new JMenuItem("ins Menue mit mir")
    ,new JComboBox(combos)
    ,new JList(combos)
    ,new JSlider(0,350,79)
    ,new JSpinner(new SpinnerNumberModel(18,0.0,42.0,2.0))
    ,new JTextField(12)
```

```
19      ,new JFormattedTextField("hallo")
20      ,new JLabel("einfach nur ein Label")
21      ,new JProgressBar(0,42)
22      };
```

Sowie eine zweite Reihung von komplexeren Swing-Komponenten:

ComponentOverview.java

```
23    JComponent [] cs2 =
24     {new JColorChooser(Color.RED)
25     ,new JFileChooser()
26     ,new JTable(13,5)
27     ,new JTree()
28     };
```

Diese beiden Reihungen zeigen sollen mit einer Hilfsmethode in einem Fenster angezeigt werden:

ComponentOverview.java

```
29    displayComponents(cs1,3);
30    displayComponents(cs2,2);
31   }
```

Für die Listen- und Auswahlkomponenten oben haben wir eine Reihung von Strings benutzt:

ComponentOverview.java

```
32   String [] combos = {"friends","romans","contrymen"};
```

Bleibt die Methode zu schreiben, die die Reihungen von Komponenten anzeigen kann. Als zweites Argument bekommt diese Methode übergeben, in wieviel Spalten die Komponenten angezeigt werden sollen.

ComponentOverview.java

```
33   public void displayComponents(JComponent [] cs,int col){
```

Ein Fenster wird definiert, für das eine Gridlayout-Zwischenkomponente mit genügend Zeilen erzeugt wird:

ComponentOverview.java

```
34     JFrame f = new JFrame();
35     JPanel panel = new JPanel();
36     panel.setLayout(
37      new GridLayout(cs.length/col+(cs.length%col==0?0:1),col));
```

Für jede Komponente wird ein `Panel` mit Rahmen und den Klassennamen der Komponente als Titel erzeugt und der Zwischenkomponente hinzugefügt:

ComponentOverview.java

```
    for (JComponent c:cs){
      JPanel p = new JPanel();
      p.add(c);
      p.setBorder(BorderFactory
                  .createTitledBorder(c.getClass().getName()));
      panel.add(p);
    }
```

Schließlich wird noch das Hauptfester zusammengepackt:

ComponentOverview.java

```
    f.getContentPane().add(panel);
    f.pack();
    f.setVisible(true);
  }
```

Und um alles zu starten, noch eine kleine Hauptmethode:

ComponentOverview.java

```
  public static void main(String [] _){
    new  ComponentOverview();
  }
}
```

Wir erhalten einmal die Übersicht von Komponenten wie in Abbildung 12.12 und einmal wie in Abbildung 12.13 dargestellt.

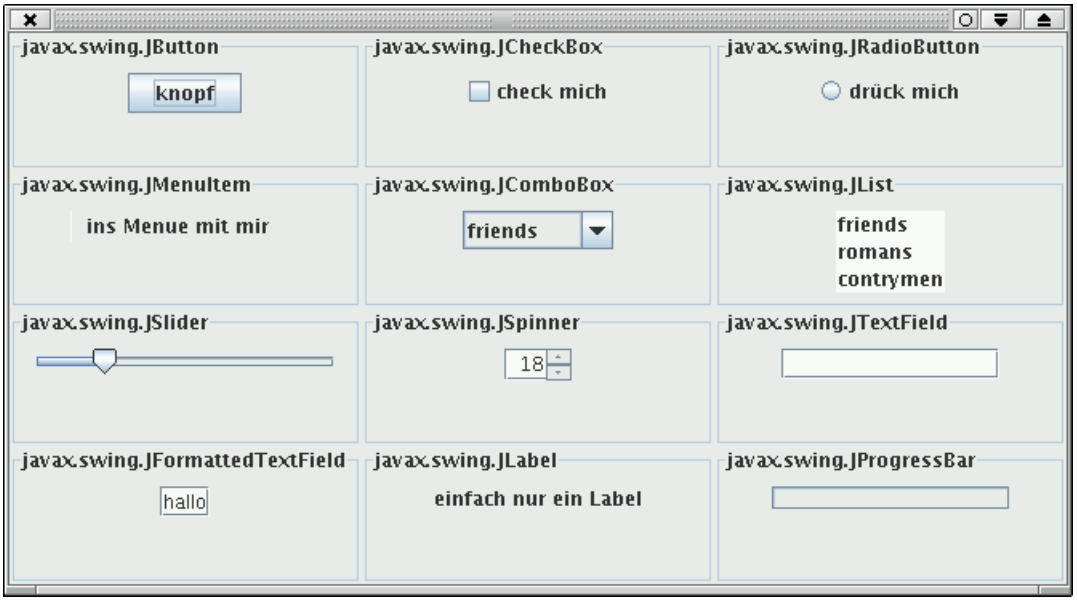


Abbildung 12.12: Überblick über einfache Komponenten.

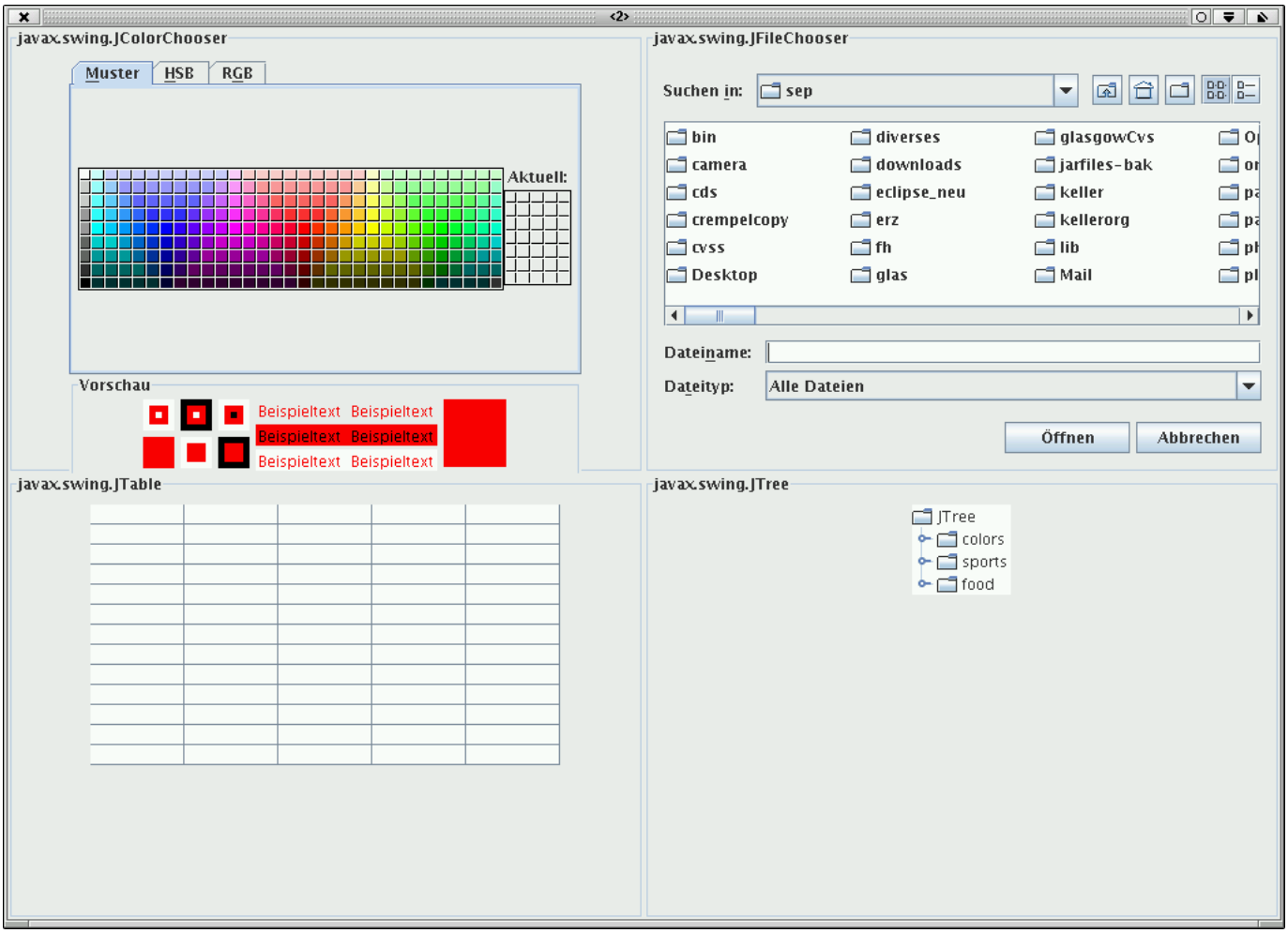


Abbildung 12.13: Überblick über komplexere Komponenten.

Womit wir uns noch nicht im einzelnen beschäftigt haben, ist das Datenmodell, das hinter den einzelnen komplexeren Komponenten steht. Im nächsten Hauptkapitel werden insbesondere Bäume als Datenstruktur, die auch in einer graphischen Komponente darstellbar ist, untersucht werden.

12.7 Swing und Steuerfäden

In der GUI-Programmierung kommt es oft vor, daß Dinge nebenläufig verlaufen sollen. Hierzu kennen wir das Prinzip von Steuerfäden. Und tatsächlich scheinen viele Dinge in unseren GUIs bisher schon nebenläufig zu funktionieren. Die verschiedenen Objekte zur Ereignisbehandlung fragen nebenläufig ab, ob ein Ereignis aufgetreten ist, auf das sie reagieren sollen oder nicht. Solange man nur diese Nebenläufigkeit nutzt, die in Swing quasi für die Ereignisbehandlung fest eingebaut ist, so bekommt man keine problematischen Fälle. Im Java Handbuch von Sun[CWH00] gibt es daher eine erste große Empfehlung im bezug auf Steuerfäden:

> The first rule of using threads is this: avoid them when you can.

Oft braucht man aber Nebenläufigkeit. Wobei man zwei große Anwendungsfälle identifizieren kann, in denen in einem GUI Nebenläufigkeit wünschenswert ist.

- Bestimmte Operationen können sehr lange dauern und damit das GUI blockieren. Das Laden oder Bearbeiten von Multimediadateien kann unter Umständen sehr lange dauern. Während dieser Zeit, werden Fensterinhalte nicht aktualisiert.
- Für Animationen soll nach bestimmten Zeitintervallen die Graphik eines GUIs verändert und neu gezeichnet werden. Hierzu benötigt man so etwas wie ein `sleep` der Steuerfäden.

Beispiel:
Wir können uns zunächst davon überzeugen, daß die einzelnen Komponenten in einer Swinganwendung nicht in einzelnen Steuerfäden laufen. Hierzu schreiben wir ein Programm mit zwei Knopfkomponenten, von denen der eine eine Ereignisbehandlung hat, die nicht terminiert.

GuiHangs.java

```
1  package name.panitz.swing.threadTest;
2
3  import javax.swing.JFrame;
4  import javax.swing.JButton;
5  import java.awt.event.*;
6
7  public class GuiHangs {
8    public static void main(String [] _){
9      JFrame f1 = new JFrame("f1");
10     JFrame f2 = new JFrame("f2");
11     JButton b1 = new JButton("b1");
12     JButton b2 = new JButton("b2");
13
14     b1.addActionListener(new ActionListener(){
15       public void actionPerformed(ActionEvent _){
16         System.out.println("b1 action");
17       }
18     });
19
20     b2.addActionListener(new ActionListener(){
21       public void actionPerformed(ActionEvent _){
22         System.out.println("b2 action");
23         while (true){}
24       }
25     });
26
27     f1.getContentPane().add(b1);
28     f2.getContentPane().add(b2);
29     f1.pack();
30     f2.pack();
31     f1.setVisible(true);
32     f2.setVisible(true);
```

```
33    }
34 }
```

Starten wir dieses Programm, so werden zunächst zwei Fenster mit Knöpfen geöffnet. Sobald der Knopf `b2` das erste Mal gedrückt wird, hängt die komplette Anwendung. Es können keine Aktionen mehr durchgeführt werden und die Fenster werden auch nicht mehr neu gezeichnet.

Leider ist Swing so konzipiert, daß man nicht auf naive Weise eigene Objekte vom Typ `Thread` erzeugen und nebenläufig starten kann. Man spricht davon, daß Swing nicht *thread safe* ist. Insbesondere in den Zeichenmethoden der Komponenten, wie `paintComponent` darf man nicht eigene Steuerfäden benutzen. Das GUI könnte sich sonst mit den Steuerfäden zur Ereignisbehandlung in den Zeichenmethoden zur Darstellung der Komponenten auf unkontrollierte Weise verwurschteln.

Um einen sicheren Umgang mit Steuerfäden zu gewährleisten, stellt Swing ein paar Hilfsklassen zur Verfügung, die die wichtigen Anwendungsfälle für Steuerfäden in GUIs abdecken. Diese Hilfsklasse sorgen dafür, daß weitere Steuerfäden immer mit den Steuerfäden zur Ereignisbehandlung konform sind.

12.7.1 Timer in Swing

Um zeitlich immer wiederkehrende Ereignisse in GUIs zu prgrammieren gibt in Swing eine Hilfsklasse `Timer`. Objekte dieser Klasse können so instanziiert werden, daß sie in bestimmten Zeitabständen Ereignisse auslösen. Der `Timer` ist also so etwas wie ein Ereignisgenerator. Zusätzlich gibt man einem `Timer`-Objekt auch einen `ActionListener` mit, der spezifiziert, wie auf diese in zeitintervallen auftretenen Ereignisse reagiert werden soll.

Beispiel:
Folgende Klasse implementiert eine simple Uhr. In einem `JLabel` wird die aktuelle Zeit angegeben. Die Komponente wird einem `Timer` übergeben, der jede Sekunde eine neues Ereigniserzeugt. Diese Ereignisse sorgen dafür, daß die Zeit im Label aktualisiert wird.

Uhr.java

```
1 package name.panitz.swing.threads;
2
3 import javax.swing.*;
4 import java.util.Date;
5 import java.awt.event.*;
```

Die Klasse Uhr ist nicht nur ein `JPanel`, in dem ein `JLabel` benutzt wird, Datum und Uhrzeit anzuzeigen, sondern implementiert gleichfalls auch einen `ActionListener`.

Uhr.java

```
6 public class Uhr extends JPanel implements ActionListener{
```

Zunächst sehen wir das Datumsfeld für diese Komponente vor:

Uhr.java

```
7   JLabel l = new JLabel(new Date()+"");
```

Im Konstruktor erzeugen wir ein Objekt vom Typ `Timer`. Dieses Objekt soll alle Sekunde (alle 1000 Millisekunden) ein Ereignis erzeugen. Dem `Timer` wird das gerade im Konstruktor erzeute Objekt vom Typ `Uhr` übergeben, das, da es ja einen `ActionListener` implementiert, auf diese Ereignisse reagieren soll.

Uhr.java

```
8   public Uhr (){
9     new Timer(1000,this).start();
10    add(l);
11  }
```

Um die Schnittstelle `ActionListener` korrekt zu implementieren, muß die Methode `actionPerformed` implementiert werden. In dieser setzen wir jeweils Datum und Uhrzeit mit dem aktuellen Wert neu ins Label.

Uhr.java

```
12  public void actionPerformed(ActionEvent _){
13    l.setText(""+new Date());
14  }
```

Und natürlich sehen wir zum Testen eine kleine Hauptmethode vor, die die Uhr in einem Fensterrahmen anzeigt.

Uhr.java

```
15  public static void main(String [] _){
16    JFrame f = new JFrame();
17    f.getContentPane().add(new Uhr());
18    f.pack();
19    f.setVisible(true);
20  }
21 }
```

Das so erzeugte Fenster mit einer laufenden Uhr findet sich in Abbildung 12.14 dargestellt.

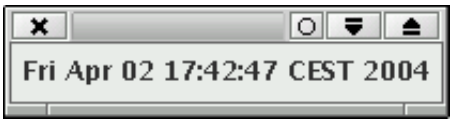


Abbildung 12.14: Eine einfache Digitaluhr.

Aufgabe 48 Implementieren Sie eine Analoguhr. Das Ziffernblatt dieser Uhr können Sie dabei mit den Methoden auf einem `Graphics`-Objekt zeichnen, oder Sie können versuchen eine Bilddatei zu laden und das Ziffernblatt als Bild bereitstellen. Als Anregung für ein Ziffernblatt können Sie sich das Bild des Ziffernblatts der handgefertigten Sekundenuhr (`http://www.panitz.name/prog2/examples/images/wiebking.jpg`) des Uhrmachers G.Wiebking herunterladen.

Animationen mit Timern

Mit dem Prinzip des Timers können wir jetzt auf einfache Weise Animationen realisieren. In einer Animation bewegt sich etwas. Dieses drücken wir durch eine entsprechende Schnittstelle aus:

Animation.java

```
1  package name.panitz.animation;
2
3  public interface Animation {
4    public void move();
5  }
```

Wir wollen einen besonderen `JPanel` realisieren, in dem sich etwas bewegen kann. Damit soll ein solcher `JPanel` auch eine Animation sein. Es bietet sich an, eine abstrakte Klasse zu schreiben, in der die Methode `move` noch nicht implementiert ist:

AnimatedJPanel.java

```
1  package name.panitz.animation;
2
3  import javax.swing.JPanel;
4  import javax.swing.Timer;
5  import java.awt.event.*;
6
7  public abstract class AnimatedJPanel
8    extends JPanel implements Animation {
```

Um zeitgesteuert das Szenario der Animation zu verändern, brauchen wir einen `Timer`.

AnimatedJPanel.java

```
9    Timer t;
```

Im Konstruktor wird dieser initialisiert. Als Ereignisbehandlung wird ein Ereignisbehandlungsobjekt erzeugt, das die Methode `move` aufruft, also dafür sorgt, daß die Szenerie sich weiterbewegt und das dafür sorgt, daß die Szenerie neu gezeichnet wird. Wir starten diesen Timer gleich.

AnimatedJPanel.java

```
10   public AnimatedJPanel(){
11     super(true);
12     t = new Timer(29,new ActionListener(){
13       public void actionPerformed(ActionEvent _){
14         move();
15         repaint();
16       }
17     });
18     t.start();
19   }
20 }
```

Jetzt können wir durch implementieren der Methode `move` und Überschreiben der Methode `paintComponent` beliebige Animationen erzeugen. Als erstes schreiben wir eine Klasse in der ein Kreis sich auf und ab bewegt:

BouncingBall.java

```
1  package name.panitz.animation;
2
3  import java.awt.Graphics;
4  import java.awt.Dimension;
5  import java.awt.Color;
6  import javax.swing.JFrame;
7
8  public class BouncingBall extends AnimatedJPanel {
```

Die Größe des Kreises und des Spielfeldes setzen wir in Konstanten fest:

BouncingBall.java

```
9    final int width = 100;
10   final int height = 200;
11   final int ballSize = 20;
```

Der Ball soll sich entlang der y-Achse bewegen, und zwar pro Bild um 4 Pixel:

BouncingBall.java

```
12   int yDir = 4;
```

Anfangs soll der Ball auf der Hälfte der x-Achse liegen und ganz oben im Bild liegen:

BouncingBall.java

```
13   int ballX = width/2-ballSize/2;
14   int ballY = 0;
```

Wir bewegen den Ball. Wenn er oben oder unten am Spielfeldrand anstößt, so ändert er seine Richtung:

BouncingBall.java

```
15   public void move(){
16     if (ballY>height-ballSize || ballY<0) yDir=-yDir;
17     ballY=ballY+yDir;
18   }
```

Zum Zeichnen, wird ein roter Hintergrund gezeichnet und der Kreis an seiner aktuellen Position.

BouncingBall.java

```
19   public void paintComponent(Graphics g){
20     g.setColor(Color.RED);
21     g.fillRect(0,0,width,height);
22     g.setColor(Color.YELLOW);
23     g.fillOval(ballX,ballY,ballSize,ballSize);
24   }
```

Unsere Größe wird verwendet als bevorzugte Größe der Komponente:

BouncingBall.java

```
25    public Dimension getPreferredSize(){
26      return new Dimension(width,height);
27    }
```

Und schließlich folgt eine kleine Hauptmethode zum Starten der Animation.

BouncingBall.java

```
28    public static void main(String [] _){
29      JFrame f = new JFrame("");
30      f.getContentPane().add(new BouncingBall());
31      f.pack();
32      f.setVisible(true);
33    }
34  }
```

Aufgabe 49 Schreiben Sie eine Animation, in der zwei Kreise sich bewegen. Die Kreise sollen an den Rändern der Spielfläche abprallen und sie sollen ihre Richtung ändern, wenn sie sich berühren.

Aufgabe 50 Schreiben Sie eine Animation, deren Verhalten über Mausklicks beeinflussen können.

12.7.2 Eine kleine Spielbibliothek

In diesem Kapitel wollen wir den `Timer` aus Swing nutzen, um eine kleine Bibliothek zum schreiben von spielen zu entwickeln. Ein Spiel soll dabei auf einer zweidimensionalen Fläche stattfinden, auf der sich Objekte bewegen. Die Bewegung bestimmter Objekte kann dabei über die Tastatur oder die Maus beeinflußt werden. Die Objekte können kollidieren und ihre Richtungen ändern. Wir benötigen also ein fundamentales Konzept, zweidimensionale Objekte die sich bewegen können. Dieses sei durch eine Schnittstelle ausgedrückt:

AMovable.java

```
1   package name.panitz.games;
2   //import java.awt.Graphics;
3
4   public interface AMovable{
5     public void move();
6     public boolean touches(int x, int y);
7     public void reverseXDirection();
8     public void reverseYDirection();
9     public void reverseDirection();
10
11    public boolean leftOf(int x);
12    public boolean rightOf(int x);
13    public boolean above(int     y);
```

```
  public boolean underneath(int y);

  public int getX();
  public void setX(int x);
  public int getY();
  public void setY(int y);
  public int getMoveX();
  public void setMoveX(int x);
  public int getMoveY();
  public void setMoveY(int y);
  public int getHeight();
}
```

Movable.java

```
package name.panitz.games;
import java.awt.Graphics;

public interface Movable extends AMovable{
  public void paintMeTo(Graphics g);
}
```

ASimpleMovable.java

```
package name.panitz.games;

public class ASimpleMovable implements AMovable{
  protected int x=2;
  protected int y=2;
  protected int moveX=1;
  protected int moveY=1;
  protected int width=10;
  protected int height=10;

  public void move(){
    setX(getX()+getMoveX());
    setY(getY()+getMoveY());
  }

  public void reverseYDirection(){
    moveY=-1*moveY;
  }

  public void reverseXDirection(){
    moveX=-1*moveX;
  }

  public void reverseDirection(){
      reverseXDirection();
      reverseYDirection();
  }
```

```

  public boolean touches(int x, int y){
    return  x>=getX()&&x<=getX()+width
         && y>=getY()&&y<=getY()+height;
  }

  public boolean leftOf(int x)    {return x>=getX();}
  public boolean rightOf(int x)   {return x<=getX()+width;}
  public boolean above(int     y) {return getY()<=y;}
  public boolean underneath(int y){return getY()+height>=y;}

  public int getX(){return x;}
  public void setX(int x){this.x=x;}
  public int getY(){return y;}
  public void setY(int y){this.y=y;}
  public int getMoveX(){return moveX;}
  public void setMoveX(int x){moveX=x;}
  public int getMoveY(){return moveY;}
  public void setMoveY(int y){moveY=y;}
  public int getHeight(){return height;}
}
```

SimpleMovable.java

```
package name.panitz.games;
import java.awt.Graphics;
import java.awt.Color;

public class SimpleMovable extends ASimpleMovable implements Movable{
  public void paintMeTo(Graphics g){
    g.setColor(Color.YELLOW);
    g.fillRect(x,y,width,height);
  }
}
```

SimpleAnimation.java

```
package name.panitz.games;

import name.panitz.animation.AnimatedJPanel;
import java.awt.Graphics;
import java.awt.Dimension;
import java.awt.Color;
import javax.swing.JFrame;

import java.util.List;
import java.util.ArrayList;
import java.awt.event.*;

public class SimpleAnimation extends  AnimatedJPanel{
  final int width = 400;
```

```
  final int height = 300;
  final List<Movable> mvs = new ArrayList<Movable>();

  public SimpleAnimation(){
    addMouseListener(new MouseAdapter(){
      public void mouseClicked(MouseEvent me){
        mvs.add(new SimpleMovable());
      }
    });
  }

  public void move(){
    for (Movable mv:mvs) mv.move();
  }

  public void paintComponent(Graphics g){
    g.setColor(Color.RED);
    g.fillRect(0,0,width,height);
    for (Movable mv:mvs) mv.paintMeTo(g);
  }

  public Dimension getPreferredSize(){
    return new Dimension(width,height);
  }

  public static void main(String [] _){
    JFrame f = new JFrame("");
    f.getContentPane().add(new SimpleAnimation());
    f.pack();
    f.setVisible(true);
  }
}
```

Das Telespiel Pong

Ball.java

```
package name.panitz.games;
import java.awt.Graphics;
import java.awt.Color;

public class Ball extends SimpleMovable{
    public Ball(){}
    public Ball(int x,int y){this.x=x;this.y=y;moveX=3;}

  public void paintMeTo(Graphics g){
    g.setColor(Color.YELLOW);
    g.fillOval(x,y,width,height);
  }
}
```

Paddle.java

```
package name.panitz.games;
import java.awt.Graphics;
import java.awt.Color;

public class Paddle extends SimpleMovable{
  public Paddle(int xPos){
    x=xPos;
    width=10;
    height=30;
    moveX=0;
    moveY=2;
  }
}
```

Pong.java

```
package name.panitz.games;

import name.panitz.animation.AnimatedJPanel;
import java.awt.Graphics;
import java.awt.event.*;
import java.awt.Dimension;
import java.awt.Color;
import javax.swing.JFrame;

import java.util.List;
import java.util.ArrayList;

public class Pong extends AnimatedJPanel{
  enum Player{left,right};
  Player nextPlayer=Player.left;

  final char leftPlayerKey = 'a';
  final char rightPlayerKey = 'l';
  final char newBallKey = ' ';
  final List<Movable> mvs = new ArrayList<Movable>();
  final Movable leftPaddle = getPaddle(10);
  final Movable rightPaddle = getPaddle(getWidth()-20);
  final Movable[] paddles = {leftPaddle,rightPaddle};

  Movable ball = null;

  int leftPoints=0;
  int rightPoints=0;

  Movable getPaddle(int x){return new Paddle(x);}
  Movable getBall(int x,int y){return new Ball(x,y);}

  public Pong(){
```

```
34      mvs.add(leftPaddle);
35      mvs.add(rightPaddle);
36
37      setFocusable(true);
38      addKeyListener(new KeyAdapter(){
39        public void keyPressed(KeyEvent e){
40          if (e.getKeyChar() == leftPlayerKey){
41            leftPaddle.setMoveY(Math.abs(leftPaddle.getMoveY()));
42          }
43          if (e.getKeyChar() == rightPlayerKey){
44            rightPaddle.setMoveY(Math.abs(rightPaddle.getMoveY()));
45          }
46        }
47        public void keyReleased(KeyEvent e){
48          if (e.getKeyChar() == leftPlayerKey){
49            leftPaddle.setMoveY(-Math.abs(leftPaddle.getMoveY()));
50          }
51          if (e.getKeyChar() == rightPlayerKey){
52            rightPaddle.setMoveY(-Math.abs(rightPaddle.getMoveY()));
53          }
54        }
55        public void keyTyped(KeyEvent e){
56          if (ball==null && e.getKeyChar() == newBallKey){
57            switch (nextPlayer){
58              case right:
59                      ball=getBall(getWidth()-32,rightPaddle.getY()+10);
60                ball.setMoveX(-ball.getMoveX());
61                break;
62              case left:
63                ball=getBall(22,leftPaddle.getY()+10);
64                break;
65              }
66              mvs.add(ball);
67          }
68        }
69      });
70    }
71
72    public void move(){
73      rightPaddle.setX(getWidth()-20);
74      if (ball!=null){
75        if (ball.above(0) || ball.underneath(getHeight())){
76          ball.reverseYDirection();
77          ball.move();
78        }
79
80        if (ball.leftOf(0) ){
81          mvs.remove(ball);
82          ball=null;
83          rightPoints=rightPoints+1;
```

```
        nextPlayer=Player.right;
      }

      else if (ball.rightOf(getWidth()) ){
        mvs.remove(ball);
        ball=null;
        leftPoints=leftPoints+1;
        nextPlayer=Player.left;
      }

      else if (
         (    ball.rightOf(getWidth()-20)
          && ball.underneath(rightPaddle.getY())
          && ball.above(rightPaddle.getY()+rightPaddle.getHeight())
         )
         ||
         (    ball.leftOf(20)
          && ball.underneath(leftPaddle.getY())
          && ball.above(leftPaddle.getY()+leftPaddle.getHeight())
         )
        )
        ball.reverseXDirection();
    }

    for (Movable mv:mvs) mv.move();
    for (Movable p:paddles){
      if (p.getY()<0) p.setY(0);
      if (p.getY()+p.getHeight()>getHeight())
        p.setY(getHeight()-p.getHeight());
    }
  }

  public void paintComponent(Graphics g){
    g.setColor(Color.GREEN);
    g.fillRect(0,0,getWidth(),getHeight());
    for (Movable mv:mvs) mv.paintMeTo(g);
    g.setColor(Color.BLACK);
    g.drawString(""+leftPoints,40,40);
    g.drawString(""+rightPoints,getWidth()-80,40);
  }

  public Dimension getPreferredSize(){
    return new Dimension(400,300);
  }

  public static void main(String [] _){
    JFrame f = new JFrame("");
    f.getContentPane().add(new Pong());
    f.pack();
    f.setVisible(true);
```

```
  }
}
```

als Applet

PongApplet.java

```
import javax.swing.*;
import name.panitz.games.Pong;
public class PongApplet extends JApplet{
  public PongApplet(){add(new Pong());}
}
```

ViewPong.html

```
<html><body>
<h1>Das Spiel Pong</h1>
<b>Tastaturbelegung:</b><p />

'a': linker Schläger<br/>
'l': rechter Schläger<br/>
' ': neuer Ball<br/>

<applet code="PongApplet.class"
        codebase="../classes/"
        width="400" height="300">
Der Browser ignoriert das Applets. Hier wäre jetzt ein Spiel
gewesen.
</applet>
</body></html>
```

Sound und Bilder

SmileBall.java

```
package name.panitz.games;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.ImageObserver;

public class SmileBall extends Ball{
  static final Toolkit toolkit = Toolkit.getDefaultToolkit();
  static Image smile;
  ImageObserver obs;

  public SmileBall(ImageObserver obs){
    this.obs=obs;
    try {
      if (smile==null)
        smile = toolkit.getImage(
                 getClass().getClassLoader()
```

```
                    .getResource("../../images/smile.gif"));
    }catch(Exception e){System.out.println(e);}
    width=40;height=30;
    moveX=8;
    moveY=6;
  }

  public SmileBall(ImageObserver obs,int x,int y){
    this(obs);
    this.x=x;this.y=y;
  }

  public void paintMeTo(Graphics g){
    g.drawImage(smile, x, y, obs);
  }

  public void loadImages(java.applet.Applet a){
    try {
      smile = a.getImage(a.getCodeBase(),"../../images/smile.gif");
    }catch(Exception _){}
  }
}
```

ImagePaddle.java

```
package name.panitz.games;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.ImageObserver;

public class ImagePaddle extends Paddle{
  static Image paddle;
  ImageObserver obs;

  public ImagePaddle(  ImageObserver obs,int x){
    super(x);
    this.obs=obs;
    try {
      if (paddle==null)
        paddle = SmileBall.toolkit.getImage(
                 getClass().getClassLoader()
                  .getResource("../../images/schlaeger.gif"));
    }catch(Exception e){System.out.println(e);}
    width=10;height=99;
    moveY=4;
  }

  public void paintMeTo(Graphics g){
    g.drawImage(paddle, x, y, obs);
  }
```

```
27
28    public void loadImages(java.applet.Applet a){
29      try {
30        paddle = a.getImage(a.getCodeBase(),"../../images/smile.gif");
31      }catch(Exception _){}
32    }
33  }
```

ImagePong.java

```
1  package name.panitz.games;
2  import java.awt.Dimension;
3  import javax.swing.JFrame;
4  public class ImagePong extends Pong{
5    Movable getPaddle(int x){return new ImagePaddle(this,x);}
6    Movable getBall(int x,int y){return new SmileBall(this,x,y);}
7    public Dimension getPreferredSize(){
8      return new Dimension(800,600);
9    }
10
11   public static void main(String [] _){
12     JFrame f = new JFrame("");
13     f.getContentPane().add(new ImagePong());
14     f.pack();
15     f.setVisible(true);
16   }
17 }
18
```

12.7.3 SwingWorker für Steuerfäden in der Ereignisbehandlung

Sun bietet eine Beispielimplementierung einer Hilfsklasse an, die es erlaubt, Code in einen eigenen Steuerfaden zu verpacken. Die Klasse `SwingWorker`. `SwingWorker` ist nicht Bestandteil des offiziellen Swing APIs, sondern lediglich im Tutorial angegeben. Im Anhang des Skripts ist die Klasse `SwingWorker` abgedruckt.

Die Klasse `SwingWorker` ist dazu gedacht, Code innerhalb einer Ereignisbehandlungsmethode, dessen Ausführung lange dauern kann, in einen Steuerfaden zu kapseln. Statt den Code direkt in der Ereignisbehandlungsmethode zu schreiben:

```
1  public void actionPerformed(ActionEvent e) {
2      ...
3      //...Code, der sehr lange dauert ...
4      ...
5  }
```

Ist er in die Methode `construct` einer `SwingWorker`-Instanz zu kapseln:

```
public void actionPerformed(ActionEvent e) {
    ...
    final SwingWorker worker = new SwingWorker() {
        public Object construct() {
            //...Code, der sehr lange dauert, ist jetzt hier...
            return someValue;
        }
    };
    worker.start();
    ...
}


```

Beispiel:
Jetzt können wir unser Eingangsbeispiel mit der `SwingWorker`-Klasse so umschreiben, daß es nicht mehr zum Hängen des gesamten Programms kommt.

GuiHangsNoLonger.java

```
package name.panitz.swing.threadTest;

import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.*;

public class GuiHangsNoLonger {
  public static void main(String [] _){
    JFrame f1 = new JFrame("f1");
    JFrame f2 = new JFrame("f2");
    JButton b1 = new JButton("b1");
    JButton b2 = new JButton("b2");

    b1.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent _){
        System.out.println("b1 action");
      }
    });

    b2.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent _){
        final SwingWorker worker = new SwingWorker() {
          public Object construct() {
            System.out.println("b2 action");
            int x = 1; while (true){if (x==0) break;}
            return null;
          }
        };
        worker.start();
      }
```

```
    });

    f1.getContentPane().add(b1);
    f2.getContentPane().add(b2);
    f1.pack();
    f2.pack();
    f1.setVisible(true);
    f2.setVisible(true);
  }
}
```

Nach Drücken des Knopfes `b2` bleibt das Gui weiterhin voll funktionsfähig. Allerdings stellt man fest, indem man sich z.B. mit `top` die laufenden Prozesse in einem Unixsystem anschaut, daß jedes Drücken des Knopfes einen neuen Unterprozess von Java erzeugt.

SwingWorkerbeispiel

Im folgenden ist ein weiteres Beispiel angegeben, in dem die Klasse `SwingWorker` benutzt wird. Diesmal soll eine Bilddatei geladen werden. Hierzu schreiben wir eine Klasse, die von `JLabel` ableitet. Das Label soll eine vorgegebene feste Größe haben. Dem Konstruktor wird eine Bilddatei übergeben.

ImageIcon.java

```
package name.panitz.gui;

import java.io.File;
import java.net.MalformedURLException;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ImageIcon extends JLabel {
  private int height = 60;
  private int width = 80;
  final Toolkit toolkit = Toolkit.getDefaultToolkit();
  private Image image;

  public ImageIcon(String fileName){this(new File(fileName));}
  public ImageIcon(final File file){
    setImage(file);
  }
```

Zum Setzen des Bildes, wird die Bilddatei gelesen, um es dann auf die vorgegebene Größe zu skalieren:

ImageIcon.java

```
  public void setImage(File jpgFile){
    try{
```

```
      image=toolkit.getImage(jpgFile.toURL());
      final MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(image, 0,width,height);
      try {tracker.waitForAll();}
      catch (Exception e) {e.printStackTrace();}
      fitImage();
      setIcon(new javax.swing.ImageIcon(image));
    }catch (MalformedURLException e){}
  }
```

Zum Skalieren ist zunächst die Originalgröße zu lesen, die skalierte Größe zu berechnen und schließlich das Bild auf die skalierte Größe zu setzen:

ImageIcon.java

```
  private void fitImage(){
    final int w = image.getWidth(this);
    final int h = image.getHeight(this);

    final int w1 = width;        final int h1 = h*width/w;
    final int w2 = w*height/h;   final int h2 = height;
    final int w3 = w1<w2?w1:w2;  final int h3 = h1<h2?h1:h2;

    image=image.getScaledInstance
                  (w3<w?w3:w,h3<h?h3:h,Image.SCALE_FAST);
  }
}
```

Damit haben wir eine Komponente, die Bilder beliebiger Größe aus Dateien in einer vorgegebenen Größe anzeigen kann. Diese Komponente können wir benutzen, um eine kleine Anwendung zu starten. Die Anwendung soll zwei Fenster Öffnen: im erstem Fenster sollen alle über die Kommandozeile mitgegebenen Biler als Label angezeigt werden, im zweitem Fenster soll eine Uhr laufen:

UseImageIcon.java

```
package name.panitz.gui;

import name.panitz.swing.threads.Uhr;

import java.io.File;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class UseImageIcon {
  public static void main(String [] args){
    JFrame f= new JFrame();
    final JPanel p = new JPanel();
    f.getContentPane().add(p);
    final int c = 10;
```

```
    final int n = args.length;

    p.setLayout(new GridLayout(n/c+((n%c)==0?0:1),c));
    for (final String arg:args)p.add(new ImageIcon(arg));

    f.pack();
    f.setVisible(true);

    JFrame f2 = new JFrame();
    f2.getContentPane().add(new Uhr());
    f2.pack();
    f2.setVisible(true);
  }
}
```

Wenn wir dieses Programm mit allen unseren Urlaubsbildern starten, dann stellen wir fest, daß es sehr lange dauert, bis sich die beiden Fenster öffnen. Erst wenn alle Bilder geladen und skaliert sind, werden die beiden Fenster geöffnet. Auch dieses Programm läßt sich mit Hilfe der Klasse `SwingWorker` verbessern. Das Erzeugen der Objekte vom Typ `ImageIcon` wird hierzu in einem `ActionListener` durchgeführt. Es wird ein `Timer` benutzt, der genau ei Ereignis erzeugt, so daß der Ereignisbehandler genau einmal auf diesen Timer reagieren kann.

BetterUseOfImageIcon.java

```
package name.panitz.gui;

import name.panitz.swing.threads.Uhr;
import name.panitz.swing.threadTest.SwingWorker;

import java.io.File;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BetterUseOfImageIcon{
  public static void main(String [] args){
    final JFrame f= new JFrame();
    final JPanel p = new JPanel();
    f.getContentPane().add(p);
    final int c = 10;
    final int n = args.length;

    p.setLayout(new GridLayout(n/c+((n%c)==0?0:1),c));
    for (final String arg:args){
      ActionListener listen = new ActionListener(){
        public void actionPerformed(ActionEvent e) {
         final SwingWorker worker = new SwingWorker() {
           public Object construct() {
              p.add(new ImageIcon(arg)); f.pack();
              f.repaint();
```

```
28                return null;
29            }
30          };
31          worker.start();
32        }
33      };
34
35      Timer t = new Timer(10,listen);
36      t.setRepeats(false);
37
38      t.start();
39    }
40    f.pack();
41    f.setVisible(true);
42
43    JFrame f2 = new JFrame();
44    f2.getContentPane().add(new Uhr());
45    f2.pack();
46    f2.setVisible(true);
47  }
48 }
```

Und tatsächlich: starten wir dieses Programm jetzt mit vielen Bilddateien, so öffnen sich gleich beide Bilder und die Uhr läuft durch. Nach und nach erscheinen die Bilder in ihrem Fenster, jedes solbald es erfolgreich geladen und skaliert wurde.

Kapitel 13

3 dimensionale Graphiken

Kapitel 14

Formale Sprachen, Grammatiken, Parser

Sprachen sind ein fundamentales Konzept nicht nur der Informatik. In der Informatik begegnen uns als auffälligste Form der Sprache*Programmiersprachen*. Es gibt gewisse Regeln, nach denen die Sätze einen Sprache aus einer Menge von Wörtern geformt werden können. Die Regeln nennen wir im allgemeinen eine *Grammatik*. Ein Satz einer Programmiersprache nennen wir Programm. Die Wörter sind, Schlüsselwörter, Bezeichner, Konstanten und Sonderzeichen.

Ausführlich beschäftigt sich die Vorlesung *Compilerbau*[GKS01] mit formalen Sprachen. Wir werden in diesem Kapitel die wichtigsten Grundkenntnisse hierzu betrachten, wie sie zum Handwerkszeug eines jeden Informatikers gehören.

14.1 formale Sprachen

Eine der bahnbrechenden Erfindungen des 20. Jahrhunderts geht auf den Sprachwissenschaftler Noam Chomsky[Cho56][1] zurück. Er präsentierte als erster ein formales Regelsystem, mit dem die Grammatik einer Sprache beschrieben werden kann. Dieses Regelsystem ist in seiner Idee verblüffend einfach. Es bietet Regeln an, mit denen mechanisch die Sätze einer Sprache generiert werden können.

Systematisch wurden Chomsky Ideen zum erstenmal für die Beschreibung der Syntax der Programmiersprache Algol angewendet[NB60].

14.1.1 kontextfreie Grammatik

Eine kontextfreie Grammatik besteht aus

- einer Menge $\mathcal{T}$ von Wörteren, den *Terminalsymbole*.
- einer Menge $\mathcal{N}$ von Nichtterminalsymbolen.

[1] Chomsky gilt als der am häufigsten zitierte Wissenschaftler des 20. Jahrhunderts. Heutzutage tritt Chomsky weniger durch seine wissenschaftlichen Arbeiten als vielmehr durch seinen Einsatz für Menschenrechte und bedrohte Völker in Erscheinung.

- ein ausgezeichnetes Startsymbol $S \in \mathcal{N}$.
- einer endlichen Menge $\mathcal{R}$ von Regeln der Form:
  nt ::= $t_1 \ldots t_n$, wobei $nt \in \mathcal{N}$, $t_i \in \mathcal{N} \cup \mathcal{T}$.

Mit den Regeln einer kontextfreien Grammatik werden Sätze gebildet, indem ausgehend vom Startsymbol Regel angewendet werden. Bei einer Regelanwendung wird ein Nichtterminalzeichen t durch die Rechte Seite einer Regel, die t auf der linken Seite hat, ersetzt.

Beispiel:
Wir geben eine Grammatik an, die einfache Sätze über unser Sonnensystem auf Englisch bilden kann:

- $\mathcal{T}$= {mars,mercury,deimos,phoebus,orbits,is,a,moon,planet}
- $\mathcal{N}$= {*start,noun-phrase,verb-phrase,noun,verb,article*}
- S =*start*
- *start* ::= *noun-phrase verb-phrase*
  noun-phrase ::= *noun*
  noun-phrase ::= *article noun*
  verb-phrase ::= *verb noun-phrase*
  noun ::= planet
  noun ::= moon
  noun ::= mars
  noun ::= deimos
  noun ::= phoebus
  verb ::= orbits
  verb ::= is
  article ::= a

Wir können mit dieser Grammatik Sätze in der folgenden Art bilden:

- *start*
  →*noun-phrase verb-phrase*
  →*article noun verb-phrase*
  →*article noun verb noun-phrase*
  → a *noun verb noun-phrase*
  → a moon *verb noun-phrase*
  → a moon orbits *noun-phrase*
  → a moon orbits *noun*
  → a moon orbits mars

- *start*
  →*noun-phrase verb-phrase*
  →*noun verb-phrase*
  → mercury *verb-phrase*
  → mercury *verb noun-phrase*
  → mercury is *noun-phrase*
  → mercury is *article noun*
  → mercury is a *noun*
  → mercury is a planet

- Mit dieser einfachen Grammatik lassen sich auch Sätze bilden, die weder korrektes Englisch sind, noch eine vernünftige inhaltliche Aussage machen:
  start
  →*noun-phrase verb-phrase*
  →*noun verb-phrase*
  → planet *verb-phrase*
  → planet *verb noun-phrase*
  → planet orbits *noun-phrase*
  → planet orbits *article noun*
  → planet orbits a *noun*
  → planet orbits a phoebus

Eine Grammatik beschreibt die Syntax einer Sprache im Gegensatz zur Semantik, der Bedeutung, einer Sprache.

Rekursive Grammatiken

Die Grammatik aus dem letzen Beispiel kann nur endlich viele Sätze generieren. Will man mit einer Grammatik unendlich viele Sätze beschreiben, so wie eine Programmiersprache unendlich viele Programme hat, so kann man sich dem Trick der Rekursion bedienen. Eine Grammatik kann rekursive Regeln enthalten; das sind Regeln, in denen auf der rechten Seite das Nichtterminalsymbol der linken Seite wieder auftaucht.

Beispiel:
Die folgende Grammatik erlaubt es arithmetische Ausdrücke zu generieren:

- $\mathcal{T} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$
- $\mathcal{N} = \{$*start*,*expr*,*op*,*integer*,*digit*,$\}$
- $S =$*start*
- *start* ::= *expr*
  expr ::= *integer*
  expr ::= *integer op expr*
  integer ::= *digit*
  integer ::= *digit integer*
  op ::= +
  op ::= -
  op ::= *
  op ::= /
  digit ::= 0
  digit ::= 1
  digit ::= 2
  digit ::= 3
  digit ::= 4
  digit ::= 5
  digit ::= 6
  digit ::= 7
  digit ::= 8
  digit ::= 9

Diese Grammatik hat zwei rekursive Regeln: eine für das Nichtterminal *expr* und eines für das Nichtterminal *integer*.

Folgende Abeleitung generiert einen arithmetischen Ausdrucke mit dieser Grammatik:

- $start$
  $\rightarrow expr$
  $\rightarrow integer\ op\ expr$
  $\rightarrow integer\ op\ integer\ op\ expr$
  $\rightarrow integer\ op\ integer\ op\ integer\ op\ expr$
  $\rightarrow integer\ op\ integer\ op\ integer\ op\ integer$
  $\rightarrow integer + integer\ op\ integer\ op\ integer$
  $\rightarrow integer + integer * integer\ op\ integer$
  $\rightarrow integer + integer * integer - integer$
  $\rightarrow digit\ integer + integer * integer - integer$
  $\rightarrow 1\ integer + integer * integer - integer$
  $\rightarrow 1\ digit\ integer + integer * integer - integer$
  $\rightarrow 12\ integer + integer * integer - integer$
  $\rightarrow 12\ digit + integer * integer - integer$
  $\rightarrow 129 + integer * integer - integer$
  $\rightarrow 129 + digit * integer - integer$
  $\rightarrow 129 + 4 * integer - integer$
  $\rightarrow 129 + 4 * digit\ integer - integer$
  $\rightarrow 129 + 4 * 5 integer - integer$
  $\rightarrow 129 + 4 * 5 digit - integer$
  $\rightarrow 129 + 4 * 53 - integer$
  $\rightarrow 129 + 4 * 53 - digit\ integer$
  $\rightarrow 129 + 4 * 53 - 8\ integer$
  $\rightarrow 129 + 4 * 53 - 8\ digit$
  $\rightarrow 129 + 4 * 53 - 87$

Aufgabe 51 Erweitern Sie die obige Grammatik so, daß sie mit ihr auch geklammerte arithmetische Ausdrücke ableiten können. Hierfür gibt es zwei neue Terminalsymbolde: `(` und `)`.

Schreiben Sie eine Ableitung für den Ausdruck: `1+(2*20)+1`

Grenzen kontextfreier Grammatiken

Kontextfreie Grammatiken sind ein einfaches und dennoch mächtiges Beschreibungsmittel für Sprachen. Dennoch gibt es viele Sprachen, die nicht durch eine kontextfreie Grammatik beschrieben werden können.

syntaktische Grenzen Es gibt syntaktisch recht einfache Sprachen, die sich nicht durch eine kontextfreie Grammatik beschreiben lassen. Eine sehr einfache solche Sprache besteht aus drei Wörtern: $\mathcal{T} = \{a,b,c\}$. Die Sätze dieser Sprache sollen so gebildet sein, daß für eine Zahl n eine Folge von n mal dem Zeichen a, n mal das Zeichen b und schließlich n mal das Zeichen c folgt, also

$$\{a^n b^n c^n | n \in I\!N\}.$$

Die Sätze dieser Sprache lassen sich aufzählen:

abc
aabbcc
aaabbbccc
aaaabbbbcccc
aaaaabbbbbccccc
aaaaaabbbbbbcccccc
...

Es gibt formale Beweise, daß derartige Sprachen sich nicht mit kontextfreie Grammatiken bilden lassen. Versuchen Sie einmal das Unmögliche: eine Grammatik aufzustellen, die diese Sprache erzeugt.

Eine weitere einfache Sprache, die nicht durch eine kontextfreie auszudrücken ist, hat zwei Terminalsymbole und verlangt, daß in jedem Satz die beiden Symbole gleich oft vorkommen, die Reihenfolge jedoch beliebig sein kann.

semantische Grenzen Über die Syntax hinaus, haben Sprachen noch weitere Einschränkungen, die sich nicht in der Grammatik ausdrücken lassen. Die meisten syntaktisch korrekten Javaprogramme werden trotzdem vom Javaübersetzer als inkorrekt zurückgewiesen. Diese Programme verstoßen gegen semantische Beschränkungen, wie z.B. gegen die Zuweisungskompatibilität. Das Programm:

```
1 class SemanticalIncorrect{int i = "1";}
```

ist syntaktisch nach den Regeln der Javagrammatik korrekt gebildet, verletzt aber die Beschränkung, daß einem Feld vom Typ `int` kein Objekt des Typs `String` zugewiesen werden darf.

Aus diesen Grund besteht ein Übersetzer aus zwei großen Teilen. Der syntaktischen Analyse, die prüft, ob der Satz mit den Regeln der Grammatik erzeugt werden kann und der semantischen Analyse, die anschließend zusätzliche semantische Bedingungen prüft.

Das leere Wort

Manchmal will man in einer Grammatik ausdrücken, daß in Nichtterminalsymbol auch zu einem leeren Folge von Symbolen reduzieren soll. Hierzu könnte man die Regel

$$t ::=$$

mit leerer rechter Seite schreiben. Es ist eine Konvention ein spezielles Zeichen für das leere Wort zu benutzen. Hierzu bedient man sich des griechischen Buchtstabens ϵ. Obige Regel würde man also schreiben als:

$$t ::= \epsilon$$

Lexikalische Struktur

Bisher haben wir uns keine Gedanken gemacht, woher die Wörter unserer Sprache kommen. Wir haben bisher immer eine gegebene Menge angenommen. Die Wörter einer Sprache bestimmen ihre lexikalische Struktur. In unseren obigen Beispielen haben wir sehr unterschiedliche Arten von Wörtern: einmal Wörter der englischen Sprache und einmal Ziffernsymbole und arithmetische Operatorsymbole. Im Kontext von Programmiersprachen spricht man von Token.

Bevor wir testen können, ob ein Satz mit einer Grammatik erzeugt werden kann, sind die einzelnen Wörter in diesem Satz zu identifizieren. Dieses geschieht in einer lexikalischen Analyse. Man spricht auch vom *Lexer* und *Tokenizer*. Um zu beschreiben, wie die einzelnen lexikalischen Einheiten einer Sprache aussehen, bedient man sich eines weiteren Formalismus, den regulären Ausdrücken.

Andere Grammatiken

Die in diesem Kapitel vorgestellten Grammatiken heißen *kontextfrei*, weil eine Regel für ein Nichtterminalzeichen angewendet wird, ohne dabei zu betrachten, was vor oder nach dem Zeichen für ein weiteres Zeichen steht, der Kontext also nicht betrachtet wird. Läßt man auch Regeln zu, die auf der linken Seite nicht ein Nichtterminalzeichen stehen haben, so kann man mächtigere Sprachen beschreiben, als mit einer kontextfreien Grammatik.

Beispiel:
Wir können mit der folgenden nicht-kontextfreien Grammatik die Sprache beschreiben, in der jeder Satz gleich oft die beiden Terminalsymbole, eber in beliebiger Reihenfolge enthält.

- $\mathcal{T}$= {a,b}
- $\mathcal{N}$= {*start*,*A*,*B*}
- S =*start*
- *start* ::= *ABstart*
  start ::= ϵ
  AB ::= *BA*
  BA ::= *AB*
  A ::= a
  B ::= b

  start
  →*ABstart*
  →*ABABstart*
  →*ABABABstart*
  →*ABABABABstart*
  →*ABABABABABstart*
  →*ABABABABAB*
  →*AABBABABAB*
  →*AABBBAABAB*
  →*AABBBABAAB*
  →*AABBBABABA*
  →*AABBBABBAA*

$\rightarrow AABBBBABAA$
$\rightarrow AABBBBBAAA$
$\rightarrow$ a$ABBBBBAAA$
$\rightarrow$ aa$BBBBBAAA$
$\rightarrow$ aab$BBBBAAA$
$\rightarrow$ aabb$BBBAAA$
$\rightarrow$ aabbb$BBAAA$
$\rightarrow$ aabbbb$BAAA$
$\rightarrow$ aabbbbbAAA
$\rightarrow$ aabbbbbaAA
$\rightarrow$ aabbbbbaa$A$
$\rightarrow$ aabbbbbaaa

Beispiel:
Auch die nicht durch eine kontextfreie Grammatik darstellbare Sprache:

$\{a^n b^n c^n | n \in I\!N\}$.

läßt sich mit einer solchen Grammatik generieren:

S::=ab$T$c
T::=AbTc$|\epsilon$
bA::=Ab
aA::=aa

Eine Ableitung mit dieser Grammatik sieht wie folgt aus:

S
$\rightarrow$ ab$T$c
$\rightarrow$ ab$A$bTcc
$\rightarrow$ ab$A$bAbTccc
$\rightarrow$ ab$A$bAbAbTcccc
$\rightarrow$ ab$A$bAbAbAbTccccc
$\rightarrow$ ab$A$bAbAbAbccccc
$\rightarrow$ abAAbb$A$bAbccccc
$\rightarrow$ abAAbAbb$A$bccccc
$\rightarrow$ abAAAbbb$A$bccccc
$\rightarrow$ abAAAbb$A$bbccccc
$\rightarrow$ abAAAbAbbbccccc
$\rightarrow$ ab$AAAA$bbbbccccc
$\rightarrow$ aAbAAAbbbbccccc
$\rightarrow$ aAAbAAbbbbccccc
$\rightarrow$ aAAAbAbbbbccccc
$\rightarrow$ a$AAAA$bbbbbccccc
$\rightarrow$ aaAAAbbbbbccccc
$\rightarrow$ aaaAAbbbbbccccc
$\rightarrow$ aaaa$A$bbbbbccccc
$\rightarrow$ aaaaabbbbbccccc

Als Preis dafür, daß man sich nicht auf kontextfreie Grammatiken beschränkt, kann nicht immer leicht und eindeutig erkannt werden, ob ein bestimmter vorgegebener Satz mit dieser Grammatik erzeugt werden kann.

Grammatiken und Bäume

Im letzten Kapitel haben wir uns ausführlich mit Bäumen beschäftigt. Eine Grammatik stellt in naheliegender Weise nicht nur ein Beschreibung einer Sprache dar, sondern jede Generierung eines Satzes dieser Sprache entspricht einem Baum, dem Ableitungsbaum. Die Knoten des Baumes sind mit Terminal- und Nichtterminalzeichen markiert, wobei Blätter mit Terminalzeichen markiert sind. Die Kinder eines Knotens sind die Knoten, die mit der rechten Seite einer Regelanwendung markiert sind.

Liest man die Blätter eines solchen Baumes von links nach rechts, so ergibt sich der generierte Satz.

Beispiel:
Die Ableitungen der Sätze unserer ersten Grammatik haben folgende Baumdarstellung:

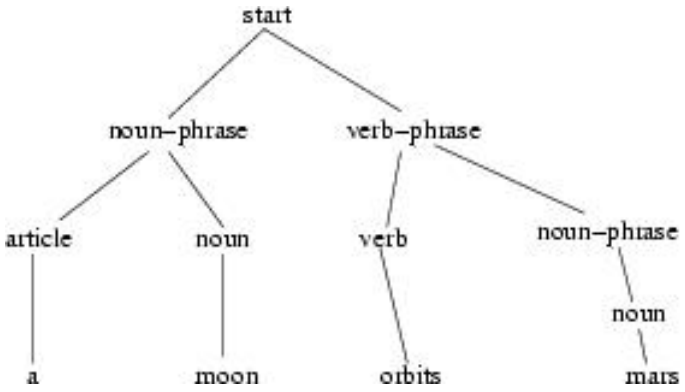


Abbildung 14.1: Ableitungsbaum für a moon orbits mars.

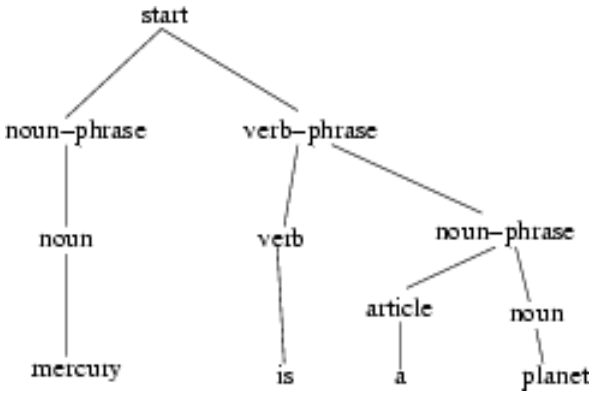


Abbildung 14.2: Ableitungsbaum für mercury is a planet.

Gegenüber unserer bisherigen Darstellung der Ableitung eines Wortes mit den Regeln einer Grammatik, ist die Reihenfolge, in der die Regeln angewendet werden in der Baumdarstellung nicht mehr ersichtlich. Daher spricht man häufiger auch vom Syntaxbaum des Satzes.

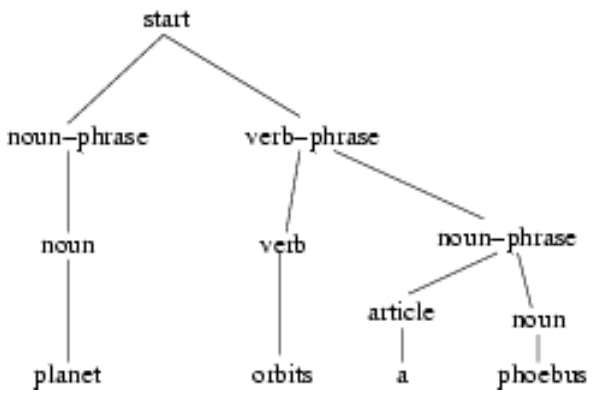


Abbildung 14.3: Ableitungsbaum für planet orbits a phoebus.

Aufgabe 52 Betrachten Sie die einfache Grammatik für arithmetische Ausdrücke aus dem letzen Abschnitt. Zeichnen Sie einen Syntaxbaum für den Audruck `1+1+2*20`.

14.1.2 Erweiterte Backus-Naur-Form

Die Ausdrucksmöglichkeit einer kontextfreien Grammatik ist auf wenige Konstrukte beschränkt. Das macht das Konzept einfach. Im Kontext von Programmiersprachen gibt es häufig sprachliche Konstrukte, wie die mehrfache Wiederholung oder eine Liste von bestimmten Teilen, die zwar mit einer kontextfreien Grammatik darstellbar ist, für die aber spezielles zusätzliche Ausdrucksmittel in der Grammatik eingeführt werden. In den nächsten Abschnitten werden wir diese Erweiterungen kennenlernen, allerdings werden wir in unseren Algorithmen für Sprachen und Grammatiken diese Erweiterungen nicht berücksichtigen.

Alternativen

Wenn es für ein Nichtterminalzeichen mehrere Regeln gibt, so werden diese Regeln zu einer Regel umgestaltet. Die unterschiedlichen rechten Seiten werden dann durch einen vertikalen Strich | gestrennt.

Durch diese Darstellung verliert man leider den direkten Zusammenhang zwischen den Regeln und einen Ableitungsbaum.

Beispiel:
Die Regeln unserer ersten Grammatik können damit wie folgt geschrieben werden:

start ::= *noun-phrase verb-phrase*
noun-phrase ::= *noun*|*article noun*
verb-phrase ::= *verb noun-phrase*
noun ::= planet|moon|mars|deimos|phoebus
verb ::= orbits|is
article ::= a

Gruppierung

Bestimmte Teile der rechten Seite einer Grammatik können durch Klammern gruppiert werden. In diesen Klammern können wieder durch einen vertikalen Strich getrennte Alternativen stehen.

> **Beispiel:**
> Die einfache Grammatik für arithmetische Ausdrücke läßt sich damit ohne das Nichtterminalzeichen *op* schreiben:
>
> *start* ::= *expr*
> *expr* ::= *integer*|*integer* (+|−|∗|/) *expr*
> *integer* ::= *digit*|*digit integer*
> *digit* ::= 0|1|2|3|4|5|6|7|8|9

Wiederholungen

Ein typische Konstrukt in Programmiersprachen ist, daß bestimmte Kontrukte wiederholt werdn können. So stehen z.B. in Java im Rumpf einer Methode mehrere Anweisungen. Solche Sprachkonstrukte lassen sich mit einer kontextfreien Grammatik ausdrücken.

> **Beispiel:**
> Eine Zahl besteht aus einer Folge von n Ziffern ($n > 0$). Dieses läßt sich durch folgende Regel ausdrücken:
>
> *Zahl* ::= *Ziffer Zahl* | *Ziffer*

1 bis n-fach Im obigen Beispiel handelt es sich um eine 1 bis n-fache Wiederholung des Zeichens *Ziffer*. Hierzu gibt es eine abkürzende Schreibweise. Dem zu wiederholenden Teil wird das Zeichen + nachgestellt.

> **Beispiel:**
> Obige Regel für das Nichtterminal *Zahl* läßt sich mit dieser abkürzenden Schreibweise schreiben als:
>
> *Zahl* ::= *Ziffer*+

0 bis n-fach Soll ein Teil in einer Wiederholung auch keinmal vorkommen, so wird statt des Zeichens + das Zeichen ∗ genommen.

> **Beispiel:**
> Folgende Regel drückt aus, daß ein Ausdruck eine durch Operatoren getrennte Liste von Zahlen ist.
>
> *expr* ::= *Zahl* | (*Op Zahl*)*

Option Ein weiterer Spezialfall der Wiederholung ist die, in der der entsprechende Teil keinmal oder einmal vorkommen darf, d.h. der Teil ist optional. Optionale Teile werden in der erweiterten Form in eckige Klammern gesetzt.

14.2 Parser

Grammatiken geben an, wie Sätze einer Sprache gebildet werden können. In der Informatik interessiert der umgekehrte Fall. Ein Satz ist vorgegeben und es soll geprüft werden, ob dieser Satz mit einer bestimmten Grammatik erzeugt werden kann. Ein Javaübersetzer prüft z.B. ob ein vorgegebenes Programm syntaktisch zur durch die Javagrammatik beschriebenen Sprache gehört. Hierzu ist ein Prüfalgorithmus anzugeben, der testet, ob die Grammatik einen bestimmten Satz generieren kann. Ein solches Programm wird *Parser* genannt. Ein Parser[2] zerteilt einen Satz in seine syntaktischen Bestandteile.

14.2.1 Parsstrategien

Man kann unterschiedliche Algorithmen benutzen, um zu testen, daß ein zu der Sprache einer Grammatik gehört. Eine Strategie geht primär von der Grammatik aus, die andere betrachten eher den zu prüfenden Satz.

Rekursiv absteigend

Eine einfache Idee zum Schreiben eines Parsers ist es, einfach nacheinander auszuprobieren, ob die Regelanwendung nach und nach einen Satz bilden kann. Hierzu startet man mit dem Startsymbol und wendet nacheinander die Regeln an. Dabei wendet man immer eine Regel auf das linkeste Nichtterminalzeichen an, so lange, bis der Satzanfang abgelitten wurde, oder aber ein falscher Satzanfang abgelitten wurde. Im letzteren Fall hat man offensichtlich bei einer Regel die falsche Alternative gewählt. Man ist in eine Sackgasse geraten. Nun geht man in seiner Ableitung zurück bis zu der letzten Regelanwendung, an der eine andere Alternative hätte gewählt werden können. Man spricht dann von *backtracking*; auf deutsch kann man treffend von Zurückverfolgen sprechen.

Auf diese Weise gelangt man entweder zu einer Ableitung des Satzes, oder stellt irgendwann fest, daß man alle Alternativen ausprobiert hat und immer in eine Sackgasse geraten ist. Dann ist der Satz nicht in der Sprache.

Diese Strategie ist eine Suche. Es wird systematisch aus allen möglichen Ableitungen in der Grammatik nach der Ableitung gesucht, die den gewünschten Satz findet.

Beispiel:
Wir suchen mit dieser Strategie die Ableitung des Satzes *a moon orbits mars* in dem Sonnensystembeispiel. Sackgassen sind durch einen Punkt • markiert.

(0)		*start*	
(1) aus 0	→	*noun-phrase verb-phrase*	
(2a) aus 1	→	*noun verb-phrase*	
(2a3a) aus 2a	→	planet *verb-phrase*	•
(2a3b) aus 2a	→	moon *verb-phrase*	•
(2a3c) aus 2a	→	mars *verb-phrase*	•
(2a3d) aus 2a	→	deimos *verb-phrase*	•
(2a3e) aus 2a	→	phobus *verb-phrase*	•

[2]Lateinisch Pars bedeutet Teil.

(2b) aus 1	→	*article noun verb-phrase*	
(3) aus 2b	→	a *noun verb-phrase*	
(4a) aus 3	→	a planet *verb-phrase*	•
(4b) aus 3	→	a moon *verb-phrase*	
(5) aus 4b	→	a moon *verb noun-phrase*	
(6) aus 5	→	a moon orbits *noun-phrase*	
(7) aus 6	→	a moon orbits *noun*	
(8a) aus 7	→	a moon orbits planet	•
(8b) aus 7	→	a moon orbits moon	•
(8c) aus 7	→	a moon orbits mars	

Alternativ könnte man diese Strategie auch symetrisch nicht von links sondern von der rechten Seite an ausführen, also immer eine Regel für das rechtestes Nichtterminalsymbol anwenden.

Linksrekursion Die Strategie des rekursiven Abstiegs auf der linken Seite funktioniert für eine bestimmte Art von Regeln nicht. Dieses sind Regeln, die in einer Alternative als erstes Symbol wieder das Nichtterminalsymbol stehen haben, das auch auf der linken Seite steht. Solche Regeln heißen linksrekursiv. Unsere Strategie terminiert in diesen Fall nicht.

Beispiel:
Gegeben sei eine Grammatik mit einer linksrekursiven Regel:

expr::=*expr*+*zahl*|*zahl*

Der Versuch den Satz

zahl+zahl

mit einem links rekursiv absteigenden Parser abzuleiten, führt zu einem nicht terminierenden rekursiven Abstieg. Wir gelangen nie in eine Sackgasse:

	expr
→	*expr*+*zahl*
→	*expr*+*zahl*+*zahl*
→	*expr*+*zahl*+*zahl*+*zahl*
→	*expr*+*zahl*+*zahl*+*zahl*+*zahl*
	...

Es läßt sich also nicht für alle Grammatiken mit dem Verfahren des rekursiven Abstiegs entscheiden, ob ein Satz mit der Grammatik erzeugt werden kann; aber es ist möglich, eine Grammatik mit linksrekursiven Regeln so umzuschreiben, daß sie die gleiche Sprache generiert, jedoch nicht mehr linksrekursiv ist. Hierzu gibt es ein einfaches Schema:

Eine Regel nach dem Schema:

$A ::= A\ rest|alt2$

ist zu ersetzen durch die zwei Regeln:

$A ::= alt2\ R$
$R ::= rest\ R|\epsilon$

wobei R ein neues Nichtterminalsymbol ist.

Linkseindeutigkeit Die rekursiv absteigende Strategie hat einen weiteren Nachteil. Wenn sie streng schematisch angewendet wird, führt sie dazu, daß bestimmte Prüfungen mehrfach durchgeführt werden. Diese mehrfache Ausführung kann sich bei wiederholter Regelanwendung multipizieren und zu einem sehr ineffizienten Parser führen. Grund dafür sind Regeln, die zwei Alternativen mit gleichem Anfang haben:

$S ::= AB|A$

Beide Alternativen für das Nichtterminalzeichen S starten mit dem Zeichen A. Für unseren Parser bedeutet das soweit: versuche nach der ersten Alternative zu parsen. Hierzu parse erst nach dem Symbol A. Das kann eine sehr komplexe Berechnung sein. Wenn sie gelingt, dann versuche anschließend weiter nach dem Symbol B zu parsen. Wenn das fehlschlägt, dann verwerfe die Regelalternative und versuche nach der zweiten Regelalternative zu parsen. Jetzt ist wieder nach dem Symbol A zu parsen, was wir bereits gemacht haben.

Regeln der obigen Art wirken sich auf unsere Parsstrategie ungünstig aus. Wir können aber ein Schema angegeben, wie man solche Regeln aus der Grammatik eleminiert, ohne die erzeugte Sprache zu ändern:

Regeln der Form

$S ::= AB|A$

sind zu ersetzen durch

$S ::= AT$
$T ::= B|\epsilon$

wobei T ein neues Nichtterminalzeichen ist.

Vorausschau Im letzten Abschnitt haben wir gesehen, wie wir die Grammatik umschreiben können, so daß nach dem Zurücksetzen in unserem Algorithmus es nicht vorkommt, bereits ausgeführte Regeln ein weiteres Mal zu durchlaufen. Schöner noch wäre es, wenn wir auf das Zurücksetzen ganz verzichten könnten. Dieses läßt sich allgemein nicht erreichen, aber es gibt Grammatiken, in denen man durch Betrachtung des nächsten zu parsenden Zeichens erkennen kann, welche der Regelalternativen als einzige Alternative in betracht kommt. Hierzu kann man für eine Grammatik für jedes Nichtterminalzeichen in jeder Regelalternative berechnen, welche Terminalzeichen als linkestes Zeichen in einem mit dieser Regel abgelittenen Satz auftreten kann. Wenn die Regelalternativen disjunkte solche Menge des ersten Zeichens haben, so ist eindeutig bei Betrachtung des ersten Zeichens eines zu parsenden Satzes erkennbar, ob und mit welcher Alternative dieser Satz nur parsbar sein kann.

Die gängigsten Parser benutzen diese Entscheidung, nach dem erstem Zeichen. Diese Parser sind darauf angewiesen, daß die Menge der ersten Zeichen der verschiedenen Regelalternativen disjunkt sind.

Der Vorteil an diesem Verfahren ist, daß die Token nach und nach von links nach rechts stückweise konsumiert werden. Sie können durch einen Datenstro, relisiert werden. Wurden sie einmal konsumiert, so werden sie nicht mehr zum Parsen benötigt, weil es kein Zurücksetzen gibt.

Schieben und Reduzieren

Der rekursiv absteigende Parser geht vom Startsymbol aus und versucht durch Regelanwendung den in Frage stehenden Satz abzuleiten. Eine andere Strategie ist die des Schiebens-und-Reduzierens (shift-reduce). Diese geht vom im Frage stehenden Satz aus und versucht die Regeln rückwärts anzuwenden, bis das Startsymbol erreicht wurde. Hierzu wird der Satz von links nach rechts (oder symmetrisch von rechts nach links) betrachtet und versucht, rechte Seiten von Regeln zu finden und durch ihre linke Seite zu ersetzen. Dazu benötigt man einen Marker, der angibt, bis zu welchen Teil man den Satz betrachtet. Wenn links des Markers keine linke Seite einer Regel steht, so wird der Marker ein Zeichen weiter nach rechts verschoben.

Beispiel:
Wir leiten im folgenden unserer allseits bekannten Beispielsatz aus dem Sonnensystem durch Schieben und Reduzieren ab. Als Marker benutzen wir einen Punkt.

		. a moon orbits mars
(shift)	→	a . moon orbits mars
(reduce)	→	*article* . moon orbits mars
(shift)	→	*article* moon . orbits mars
(reduce)	→	*article noun* . orbits mars
(reduce)	→	*noun-phrase* . orbits mars
(shift)	→	*noun-phrase* orbits . mars
(reduce)	→	*noun-phrase verb* . mars
(shift)	→	*noun-phrase verb* mars .
(reduce)	→	*noun-phrase verb noun* .
(reduce)	→	*noun-phrase verb noun-phrase* .
(reduce)	→	*noun-phrase verb-phrase* .
(reduce)	→	*start*.

Wir werden im Laufe dieser Vorlesung diese Parserstrategie nicht weiter verfolgen.

14.3 Handgeschriebene Parser

Genug der Theorie. Wir kennen jetzt Grammatiken und zwei Strategien, um für einen Satz zu entscheiden, ob er mit einer Grammatik generiert werden kann. In diesem Abschnitt wollen wir unsere theoretischen Erkenntnisse in ein Programm praktisch umsetzen. Wir werden hierzu einen rekursiv absteigenden Parser entwickeln.

14.3.1 Basisklassen der Parserbibliothek

Wir werden eine Bibliothek entwerfen, mit der Parser einer beliebigen Grammatik konstruiert werden können. Parser we lang="java"rden dabei Objekte sein, insbesondere mit einer Methode zum Parsen. Entwerfen wir also zunächst eine Schnittstelle, die einen Parser beschreibt. Die Schnittstelle für Parser soll genau eine Methode haben, die Methode `parse`. Sie bekommt ein Argument, die Liste der Token, die zu parsen sind. Ein Parser erzeugt ein Ergebnis. Das Ergebnis kann von beliebigen Typen sein: so gibt es Parser, die einen Baum als Ergebnis produzieren, oder Parser, die nur als einen bool'schen Wert angeben, ob der Parsevorgang erfolgreich war. Um diesem Rechnung zu tragen, lassen wir in der Schnittstelle `Parser` den Ergebnistyp zunächst variabel. Wir erhalten folgende generische Schnittstelle:

Parser.java

```
1 package name.panitz.parser;
2 import java.util.List;
3 public interface Parser<a> {
4   ParseResult<a> parse(List<Token> ts);
5 }
```

Für einen beliebigen aber festen Typen `a` können wir Parser implementieren: so ist der Typ `Parser<String>` ein Parser, der ein Stringobjekt als Ergebnis liefert.

Wir haben in der obigen Definition der Parserschnittstelle einen Typ `Token` vorgesehen. Auch für diesen sehen wir eine Schnittstelle vor:

Token.java

```
1 package name.panitz.parser;
2 public interface Token {}
```

Die Schnittstelle ist leer. Sie hat keine Eigenschaften. Der Einfachheit haben wir als einen Strom von Token eine Liste von Token verwenden.

Schließlich haben wir einen Typ `ParseResult<a>` in der Parserschnittstelle vorgesehen. Das Ergebnis des Parsevorgangs eines Parser des Typs `Parser<a>` ist vom Typ `a`. Es reicht aber nicht aus, allein das Parseergebnis zurückzugeben. Wir müssen zusätzlich angeben, welche Token durch den Parsvorgang konsumiert wurden. Daher haben wir einen Typ `ParseResult<a>` vorgesehen, der aus zwei Teilen besteht:

- dem eigentlichen Ergebnis vom Typ `a`.
- der Liste der durch den Parsvorgang nicht konsumierten restlichen Token.

So läßt sich folgende Klasse für Parsergebnisse definieren:

ParseResult.java

```
1 package name.panitz.parser;
2 import java.util.List;
3 public class ParseResult<a> {
4   final private a result;
5   final private List<Token> remainingToken;
6   public ParseResult(a r,List<Token> toks){
7     result=r;remainingToken=toks;
```

```
8   }
9   public a getResult(){return result;}
10  public List<Token> getRemainingToken(){return remainingToken;}
11  public boolean failed(){return result==null;}
12 }
```

Wir haben für die zwei Teile eines Parsergebnisse jeweils eine `get`-Methode definiert. Die Methode `failed` gibt an ob es sich um einen erfolgreichen Parsvorgang gehandelt hat oder nicht.

Der Epsilon Parser

Wir können schon einen allerersten Parser schreiben, den Epsilonparser, der kein Token der Eingabe konsumiert. Er endet immer erfolgreich. Wir lassen ihm daher immer den Wert `true` als Ergebnis zurückgeben. Kein Token der Eingabeliste wird konsumiert.

Epsilon.java

```
1 package name.panitz.parser;
2 import java.util.List;
3
4 public class Epsilon implements Parser<Boolean>{
5   public ParseResult<Boolean> parse(List<Token> ts){
6      return new ParseResult<Boolean>(true,ts);
7   }
8 }
```

14.3.2 Parser zum Erkennen von Token

Der allgemeine Rahmen für Parser ist nun abgesteckt. Jetzt können wir schrittweise für komplexe Grammatiken Parser bauen. Der elementarste Parser ist der, der prüft, ob ein bestimmtes Token als nächstes in der Eingabeliste folgt. Hierzu ist eine Klasse, die `Parser` implementiert, zu definieren, in der die Methode `parse` entsprechend implementiert ist. In der Methode `parse` ist zunächst zu prüfen, ob die Tokenliste nicht leer ist. Dann ist zu prüfen, ob das nächste Token ein bestimmtes Token, nach dem der Parser sucht, gleicht. Im Erfolgsfall wird ein erfolgreiches Ergebnis mit dem gefundenen Token erzeugt.

ParseToken.java

```
1 package name.panitz.parser;
2 import java.util.List;
3
4 public class ParseToken<a extends Token> implements Parser<a>{
5   final private a t;
6   public ParseToken(a _t){t=_t;}
7   public ParseResult<a> parse(List<Token> ts){
8     if (ts.isEmpty()) return new ParseResult<a>(null,ts);
9     if (ts.get(0).equals(t))
10      return new ParseResult<a>((a)ts.get(0),ts.subList(1,ts.size()));
11    return new ParseResult<a>(null,ts);
12  }
13 }
```

14.3.3 Parser zum Bilden der Sequenz zweier Parser

In einfachen Chomsky-Grammatiken gibt es nur zwei Arten wie für ein Nichtterminalsymbol Regeln gebildet werden:

- durch die Sequenz mehrer Symbole auf der rechten Seite einer Regel.
- durch mehrere Regelalternativen.

Für diese beiden Bildungsprinzipien schreiben wir Parserobjekte, die aus jeweils zwei Parser einen neuen Parser bilden.

In diesen Abschnitt machen wir das zunächst für die Sequenz zweier Parser. Haben wir in der Grammatik eine Regel der Form:

$$A ::= B \;\; C$$

So können wir sie beim Schreiben eines Parser lesen als: Um den Parser für das Nichtterminalsymbol `A` zu schreiben, schreibe die Parser für die Symbole B und C und konstruiere mit diesen beiden den Parser für das Symbol A durch den Sequenzparser aus den beiden Teilparsern.

Das Ergebnis einer Sequenz von zwei Parsern muß die beiden Parsergebnisse irgendwie sinnvoll kombinieren. Hierzu sehen wir eine Klasse vor, die Paare von zwei Objekten repräsentieren kann:

Pair.java

```
1 package name.panitz.parser;
2
3 public class Pair<a,b>{
4   final private a e1;
5   final private b e2;
6   public Pair(a x1,b x2){e1=x1;e2=x2;}
7   public a getE1(){return e1;}
8   public b getE2(){return e2;}
9 }
```

Die Sequenz zweier Parser kann jetzt als Parsergebnis ein Paar aus den beiden Teilergebnissen haben. Wir implementieren den Parser, der die Sequenz zweier Parser darstellt. Die zwei Teilparser werden dem Konstruktor übergeben:

Seq.java

```
1 package name.panitz.parser;
2
3 import java.util.List;
4
5 public class Seq<a,b> implements Parser<Pair<a,b>>{
6   final private Parser<a> p1;
7   final private Parser<b> p2;
8   public Seq(Parser< a> _p1,Parser< b> _p2){p1=_p1;p2=_p2;}
```

Die Methode `parse` wendet zunächst den ersten Parser auf die Tokenliste an:

Seq.java
```
9   public ParseResult<Pair<a,b>> parse(List<Token> ts){
10    ParseResult<a> r1 = p1.parse(ts);
```

Sollte dieses schon mißglücken, so ist das Gesamtergebnis auch ein mißglückter Parsversuch:

Seq.java
```
11    if (r1.failed()) return new ParseResult<Pair<a,b>>(null,ts);
```

Im erfolgreichen Fall sind die übrigen noch nicht konsumierten Token des ersten Parses zu nehmen und der zweite Parser auf diese anzuwenden:

Seq.java
```
12    ParseResult<b> r2 = p2.parse(r1.getRemainingToken());
```

Sollte dieses schon mißglücken, so ist das Gesamtergebnis auch ein mißglückter Parsversuch:

Seq.java
```
13    if (r2.failed()) return new ParseResult<Pair<a,b>>(null,ts);
```

Sind schließlich beide Parser erfolgreich gewesen ist ein erfolgreiches Ergebnis als Paar der beiden Teilergebnisse zu bilden.

Seq.java
```
14    return new ParseResult<Pair<a,b>>
15                  (new Pair<a,b>(r1.getResult(),r2.getResult())
16                  ,r2.getRemainingToken());
17  }
18 }
```

14.3.4 Parser zum Bilden der Alternative zweier Parser

Das zweite Bildungsprinzip, um für eine Grammatik einen Parser zu bauen, ist die alternative Anwendung zweier bestehender Parser. In einer einfachen Chomskygrammatik stellt sich dieses durch zwei Regelalternativen dar.

Haben wir in der Grammatik zwei Regeln der Form:

$$A ::= B$$
$$A ::= C$$

So können wir diese lesen als: um das Nichtterminal `A` zu parsen, versuche den Parser für B, sollte dieses mißglücken versuche mit dem Parser für C zu Parsen.

Entsprechend der Klasse `Seq` für die Sequenz von zwei Parsern definieren wir die Klasse `Alt` für die Alternative aus zwei Parsern:

Alt.java

```
package name.panitz.parser;

import java.util.List;
import java.util.ArrayList;

public class Alt<a> implements Parser<a>{
  final private Parser<a> p1;
  final private Parser<a> p2;
  public Alt(Parser<a> _p1,Parser<a> _p2){p1=_p1;p2=_p2;}
```

Zum Parsen der Alternative zweier Parser wird erst der erste der beiden ausprobiert:

Alt.java

```
  public ParseResult<a> parse(List<Token> ts){
    ParseResult<a> r1 = p1.parse(ts);
```

Sollte dieses mißglücken, so ist der zweite Parser auf derselben Tokenliste anzuwenden, andernfalls ist das erfolgreiche Ergebnis des ersten Parses zurückzugeben.

Alt.java

```
    if (r1.failed()) return p2.parse(ts);
    return r1;
  }
}
```

14.3.5 Parser zum Verändern des Ergebnisses

Unsere kleine Parserbibliothek kann bisher drei Dinge:

- elementare Parser definieren, die bestimmte Token erkennen
- aus zwei Parsern die Sequenz bilden, die als Ergebnis das Paar der zwei Tailparses hat.
- aus zwei Parsern einen neuen Parser bilden, der einen der beiden Parser anzuwendet.

Oft will man etwas mit dem Ergebnis anstellen, nachdem ein Parser es erfolgreich erkannt hat. Man möchte also auf das Ergebnis eine Funktion anwenden. Hierzu definieren wir uns eine Schnittstelle, die zunächst einmal beschreibt, wie Funktionsobjekte definiert sind.

Function.java

```
package name.panitz.parser;
public interface Function<a,b>{
    b apply(a x);
}
```

Unsere letzte Klasse zum Bilden von Parsern, beschreibt einen Parser, der zunächst einen vorgegebenen Parser auf die Tokenliste anwendet und anschließend auf das erhaltene Ergebnis eine gegebene Funktion anwendet.

Map.java

```
1  package name.panitz.parser;
2  import java.util.List;
3  import java.util.ArrayList;
4
5  public class Map<a,b> implements Parser<b>{
6    final private Parser<a> p1;
7    final private Function<a,b> f;
8    public Map(Parser<a> _p1,Function<a,b> _f){p1=_p1;f=_f;}
```

Zum Parsen wird zunächst der gegebene Parser auf die Tokenliste angewendet:

Map.java

```
9    public ParseResult<b> parse(List<Token> ts){
10     ParseResult<a> r1 = p1.parse(ts);
```

Im Erfolgsfall wird ein neues Ergebnis gebaut, indem auf die Ergebiskomponente die mitgegebene Funktion angewendet wird.

Map.java

```
11     if (r1.failed()) return new ParseResult<b>(null,ts);
12     return new ParseResult<b>
13       (f.apply(r1.getResult()),r1.getRemainingToken());
14   }
15 }
```

Unsere Parserbibliothek ist vollständig. In den nächsten Abschnitten sehen wir Beispiele, wie sie zum Schreiben von Parsern benutzt werden kann.

14.3.6 Beispiel: arithmetische Ausdrücke

Als Beispiel betrachten wir eine Grammatik, die einfache arithmetische Ausdrücke generiert.

Grammatik:

start ::= *addExpr*
addExpr ::= *multExpraddOp addExpr*|*multExpr*
multExpr ::= zahl *multOp multExpr*| zahl
multOp ::= `*`|`/`
addOp ::= `+`|`-`

Token

In unserer obigen Grammatik gibt es 5 verschiedene Token. Die vier Operatoren und Zahlen. Für Zahlen können wir eine einfache Tokenklasse definieren, die diese beschreibt.

Zahl.java

```
package name.panitz.parser;
public class Zahl implements Token{
  final public int i;
  public Zahl(int _i){i=_i;}
  public boolean equals(Object o){return o instanceof Zahl;}
}
```

Die vier Operatoren können wir als Aufzählung realisieren:

OpToken.java

```
package name.panitz.parser;
public enum OpToken implements Token{
 add,sub,mult,div;
```

Jeder Operator entspricht einer zweistelligen Funktion. Auch dieses können wir mit Hilfe einer Schnittstelle ausdrücken:

BinFunction.java

```
package name.panitz.parser;

public interface BinFunction<a,b,c>{
  c apply(a x,b y);
}
```

Die Operatoren sollen spezielle zweistellige Funktionen auf ganzen Zahlen sein:

IntOp.java

```
package name.panitz.parser;

public interface IntOp
         extends BinFunction<Integer,Integer,Integer>{}
```

Nun können wir in der Aufzählungsklasse der Operatortoken noch eine Methode vorsehen, die zu dem entsprechenden Operator die entsprechende Funktion als Ergebnis zurückgibt:

OpToken.java

```

 public IntOp getFunction(){
   switch (this){
     case div : return new IntOp(){
          public Integer apply(Integer x,Integer y){return x/y;}};
     case mult : return new IntOp(){
          public Integer apply(Integer x,Integer y){return x*y;}};
     case add : return new IntOp(){
          public Integer apply(Integer x,Integer y){return x+y;}};
     default : return new  IntOp(){
          public Integer apply(Integer x,Integer y){return x-y;}};
   }
  }
}
```

AuswertungsParser

Wir schreiben einen ersten Parser, der für einen arithmetischen Ausdruck seinen ausgewerteten Wert als Ergenis hat. Für jedes Symbol der Klasse schreiben wir einen Parser, der eine ganze Zahl als Ergebnis hat. Wir fangen mit den Terminalsymbolen an. Zunächst der Parser für Zahlen. Hier handelt ich sich um den Tokenparser für das Token `Zahl`, auf den die Funktion, die die eigentliche Zahl extrahiert angewendet wird.

ZahlEval.java

```
package name.panitz.parser;
import java.util.List;

public class ZahlEval implements Parser<Integer>{
  public ParseResult<Integer> parse(List<Token> ts){
    return new Map<Zahl,Integer>
        (new ParseToken<Zahl>(new Zahl(0))
        ,new Function<Zahl,Integer>(){
           public Integer apply(Zahl z){return z.i;}}
        ).parse(ts);
  }
}
```

Als nächstes kümmern wir uns um die Operatoren. In einem Funktionsobjekt kapseln wir die Funktion, die für ein Operatortoken die entsprechende zweistellige Funktion extrahiert:

GetIntOp.java

```
package name.panitz.parser;
import java.util.List;

public class GetIntOp implements Function<OpToken,IntOp>{
  public IntOp apply(OpToken t){
    return t.getFunction();
  }
}
```

Der Parser für Punktrechnungsoperatoren besteht aus der Alternative der beiden Tokenparser mit anschließender Extraktion der binären Funktion:

MultOpEval.java

```
package name.panitz.parser;
import java.util.List;

public class MultOpEval implements Parser<IntOp>{
  public ParseResult<IntOp> parse(List<Token> ts){
    return new Map<OpToken,IntOp>
      (new Alt<OpToken>
         (new ParseToken<OpToken>(OpToken.mult)
         ,new ParseToken<OpToken>(OpToken.div))
      ,new GetIntOp()).parse(ts);
  }
}
```

Analog funktioniert der Parser für die Strichrechnung:

AddOpEval.java

```
package name.panitz.parser;
import java.util.List;

public class AddOpEval implements Parser<IntOp>{
  public ParseResult<IntOp> parse(List<Token> ts){
    return new Map<OpToken,IntOp>
      (new Alt<OpToken>
         (new ParseToken<OpToken>(OpToken.add)
         ,new ParseToken<OpToken>(OpToken.sub))
      ,new GetIntOp()).parse(ts);
  }
}
```

Als nächstes folgt der Parser für das Nichtterminalsymbol *multExpr*.

multExpr ::= zahl *multOp multExpr*| zahl

Dieses ist entweder eine Sequenz aus `Zahl`*multOp multExpr* oder nur eine `Zahl`. Die Sequenz hat als Ergebnistyp doppelt verschachtelte Paare des Typs: `Pair<Integer,Pair<IntOp,Integer>>`. Aus diesen Typ ist eine ganze Zahl zu bilden. Hierfür schreiben wir die Klasse `DoIntOp`:

DoIntOp.java

```
package name.panitz.parser;

public class DoIntOp
   implements Function<Pair<Integer,Pair<IntOp,Integer>>,Integer>{
  public Integer apply(Pair<Integer,Pair<IntOp,Integer>> x){
    return x.getE2().getE1().apply(x.getE1(),x.getE2().getE2());
  }
}
```

Jetzt können wir entsprechend der Grammatik einen Parser für das Nichtterminalsymbol *multExpr* schreiben:

MultExprEval.java

```
package name.panitz.parser;
import java.util.List;

public class MultExprEval implements Parser<Integer>{
  public ParseResult<Integer> parse(List<Token> ts){
    return
      new Alt<Integer>
       (new Map<Pair<Integer,Pair<IntOp,Integer>>,Integer>
         (new Seq<Integer,Pair<IntOp,Integer>>
           (new ZahlEval(),new Seq<IntOp,Integer>
                             (new MultOpEval(),new MultExprEval()))
         ,new DoIntOp()
```

```
13          )
14        ,new ZahlEval()
15        ).parse(ts);
16    }
17  }
```

Analog läßt sich der Parser für das Nichtterminalsymbol *addExpr* schreiben:

$$addExpr ::= multExpraddOp\ addExpr|multExpr$$

AddExprEval.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class AddExprEval implements Parser<Integer>{
5    public ParseResult<Integer> parse(List<Token> ts){
6      return
7        new Alt<Integer>
8         (new Map<Pair<Integer,Pair<IntOp,Integer>>,Integer>
9           (new Seq<Integer,Pair<IntOp,Integer>>
10            (new MultExprEval(),new Seq<IntOp,Integer>
11                              (new AddOpEval(),new AddExprEval()))
12          ,new DoIntOp()
13          )
14        ,new MultExprEval()
15        ).parse(ts);
16    }
17 }
```

Damit sind wir fertig. *addExpr* ist das Startsymbol der Grammatik.

EvalArith.java

```
1  package name.panitz.parser;
2  import java.util.List;
3
4  public class EvalArith implements Parser<Integer>{
5    Parser<Integer> startExpr = new AddExprEval();
6
7    public ParseResult<Integer> parse(List<Token> ts){
8      return startExpr.parse(ts);
9    }
10 }
```

Im folgenden eine kleine Testklasse für unseren ersten handgeschriebenen Parser:

TestEvalArith.java

```
1  package name.panitz.parser;
2  import java.util.List;
3  import java.util.ArrayList;
```

```
4
5  public class TestEvalArith {
6
7    public static void main(String[] _){
8      List<Token> ts = new ArrayList<Token>();
9      ts.add(new Zahl(17));
10     ts.add(OpToken.mult);
11     ts.add(new Zahl(2));
12     ts.add(OpToken.add);
13     ts.add(new Zahl(8));
14
15     ParseResult<Integer> res = new EvalArith().parse(ts);
16     System.out.println(res.getResult());
17     System.out.println(res.getRemainingToken());
18   }
19 }
```

Aufbau eines Parsbaumes

In diesen Abschnitt schreiben wir einen zweiten Parser für die Grammatik der arithmetischen Ausdrücke. Dieses mal liefert der Parser einen Baum als Ergebnis.

Für jedes Symbol der Sprache schreiben wir wiesder genau eine Klasse. Die Klasse unterscheiden sich von den Parserklassen des vorherigen Parser nur in den Funktionsobjekten, die das Ergebnis berechnen.

ZahlTree.java

```
1  package name.panitz.parser;
2  import name.panitz.data.tree.*;
3  import java.util.List;
4
5  public class ZahlTree implements Parser<Tree<String>>{
6    public ParseResult<Tree<String>> parse(List<Token> ts){
7      return new Map<Zahl,Tree<String>>
8          (new ParseToken<Zahl>(new Zahl(0))
9          ,new Function<Zahl,Tree<String>>(){
10            public Tree<String> apply(Zahl z){
11              return new Tree<String>(""+z.i);}}
12         ).parse(ts);
13   }
14 }
```

OpTree.java

```
1  package name.panitz.parser;
2  import name.panitz.data.tree.*;
3
4  public interface OpTree
5      extends BinFunction<Tree<String>,Tree<String>,Tree<String>>{}
```

GetOpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;
import java.util.ArrayList;

public class GetOpTree implements Function<OpToken,OpTree>{
  public OpTree apply(final OpToken t){
    return
      new OpTree(){
        public Tree<String> apply(Tree<String>op1,Tree<String> op2){
          List<Tree<String>> cs = new ArrayList<Tree<String>>();
          cs.add(op1);
          cs.add(op2);
          return new Tree<String>(t.toString(),cs);
        }
      };
  }
}
```

MultOpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;

public class MultOpTree implements Parser<OpTree>{
  public ParseResult<OpTree> parse(List<Token> ts){
    return new Map<OpToken,OpTree>
      (new Alt<OpToken>
         (new ParseToken<OpToken>(OpToken.mult)
         ,new ParseToken<OpToken>(OpToken.div))
      ,new GetOpTree()).parse(ts);
  }
}
```

AddOpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;

public class AddOpTree implements Parser<OpTree>{
  public ParseResult<OpTree> parse(List<Token> ts){
    return new Map<OpToken,OpTree>
      (new Alt<OpToken>
         (new ParseToken<OpToken>(OpToken.add)
         ,new ParseToken<OpToken>(OpToken.sub))
```

```
      ,new GetOpTree()).parse(ts);
  }
}
```

DoOpTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

public class DoOpTree
 implements  Function<Pair<Tree<String>,Pair<OpTree,Tree<String>>>
                     ,Tree<String>> {
  public Tree<String> apply
                   (Pair<Tree<String>,Pair<OpTree,Tree<String>>> x){
    return x.getE2().getE1().apply(x.getE1(),x.getE2().getE2());
  }
}
```

MultExprTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;

public class MultExprTree implements Parser<Tree<String>>{
  public ParseResult<Tree<String>> parse(List<Token> ts){
    return
      new Alt<Tree<String>>
       (new Map<Pair<Tree<String>,Pair<OpTree,Tree<String>>>,Tree<String>>
         (new Seq<Tree<String>,Pair<OpTree,Tree<String>>>
           (new ZahlTree(),new Seq<OpTree,Tree<String>>
                            (new MultOpTree(),new MultExprTree()))
         ,new DoOpTree()
         )
       ,new ZahlTree()
       ).parse(ts);
  }
}
```

AddExprTree.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;

import java.util.List;

public class AddExprTree implements Parser<Tree<String>>{
  public ParseResult<Tree<String>> parse(List<Token> ts){
    return
      new Alt<Tree<String>>
```

```
      (new Map<Pair<Tree<String>,Pair<OpTree,Tree<String>>>,Tree<String>>
        (new Seq<Tree<String>,Pair<OpTree,Tree<String>>>
          (new MultExprTree(),new Seq<OpTree,Tree<String>>
                            (new AddOpTree(),new AddExprTree()))
        ,new DoOpTree()
        )
      ,new MultExprTree()
      ).parse(ts);
  }
}
```

TreeArith.java

```
package name.panitz.parser;
import java.util.List;
import name.panitz.data.tree.*;

public class TreeArith implements Parser<Tree<String>>{
  Parser<Tree<String>> startExpr = new AddExprTree();

  public ParseResult<Tree<String>> parse(List<Token> ts){
    return startExpr.parse(ts);
  }
}
```

TestTreeArith.java

```
package name.panitz.parser;
import name.panitz.data.tree.*;
import java.util.List;
import java.util.ArrayList;
import javax.swing.*;

public class TestTreeArith {

  public static void main(String[] _){
    List<Token> ts = new ArrayList<Token>();
    ts.add(new Zahl(17));
    ts.add(OpToken.mult);
    ts.add(new Zahl(2));
    ts.add(OpToken.add);
    ts.add(new Zahl(8));
    ts.add(OpToken.mult);
    ts.add(new Zahl(2));
    ts.add(OpToken.div);
    ts.add(new Zahl(1));
    ts.add(OpToken.add);
    ts.add(new Zahl(1));
    ts.add(OpToken.sub);
    ts.add(new Zahl(1));
    ParseResult<Tree<String>> res = new TreeArith().parse(ts);
```

```
    JFrame f = new JFrame();
    f.getContentPane().add(new DisplayTree(res.getResult()));
    f.pack();
    f.setVisible(true);
    System.out.println(res.getRemainingToken());
  }
}
```

14.3.7 Tokenizer

Wir wollen wenigstens für die Grammatik der arithmetischen Ausdrücke einen Tokenizer schreiben, so daß wir eine Anwendung schreiben können, die eine Datei liest, deren Inhalt als arithmetischen Ausdruck parst und in einer Bilddatei den Ableitungsbaum darstellt.

Wir sehen eine Klasse vor, die Tokenizerobjekte für die Token in arithmethischen Ausdrücken darstellt:

ArithTokenizer.java

```
package name.panitz.parser;
import java.util.List;
import java.util.ArrayList;
import java.io.Reader;
import java.io.StringReader;
import java.io.IOException;

class ArithTokenizer {
```

Ein Tokenizer soll auf einen Eingabestrom des Typs `Reader` operieren. Ein Feld soll das aktuelle aus diesem Strom untersuchte Zeichen enthalten, und eine Liste die bisher gefundenen Token:

ArithTokenizer.java

```
  Reader reader;
  int next=0;
  List<Token> result = new ArrayList<Token>();
```

Wir sehen einen Konstruktor vor, der den Strom, aus dem gelesen wird übergeben bekommt, und das erste Zeichen aus diesem Strom einliest:

ArithTokenizer.java

```
  ArithTokenizer(Reader reader){
    this.reader=reader;
    try {
      next= reader.read();
    }catch (IOException _){}
  }
```

Die entscheidene Methode `tokenize` liest nach und nach die Zeichen aus dem Eingabestrom und erzeugt ja nach gelesenen Zeichen ein Token für die Ergebnistokenliste:

ArithTokenizer.java

```
18  List<Token> tokenize() throws Exception{
19    try {
20      while (next>=0){
21        char c = (char)next;
22        switch (c){
```

Für Leerzeichen wird kein Token erzeugt:

ArithTokenizer.java

```
23          case '\u0020'  : break;
24          case '\n' : break;
25          case '\t' : break;
```

Token, die aus einen Zeichen bestehen, erzeugen sobald das Zeichen erkannt wurde ein Tokenobjekt für die Ausgabeliste:

ArithTokenizer.java

```
26          case '+'  : result.add(OpToken.add) ; break;
27          case '-'  : result.add(OpToken.sub) ; break;
28          case '*'  : result.add(OpToken.mult); break;
29          case '/'  : result.add(OpToken.div) ; break;
```

Wenn keines der obigen Zeichen gefunden wurde, kann es sich nur noch um den Anfang einer Zahl, oder um eine fehlerhafte Eingabe handeln. Wir unterscheiden, ob das Zeichen eine Ziffer darstellt, und versuchen eine Zahl zu lesen, oder werfen eine Ausnahme:

ArithTokenizer.java

```
30          default   : if (Character.isDigit(c)) {
31                        result.add(getInt());
32                        continue;
33                      }else throw new Exception
34                            ("unexpected Token found: '"+c+"'");
35        }
```

Schließlich können wir den Strom um ein neues Zeichen bitten:

ArithTokenizer.java

```
36
37        next = reader.read();
38      }
39    }catch (IOException _){}
40    return result;
41  }
42
```

Zum Lesen von Zahlen holen wir solange Zeichen, wie noch Ziffern im Eingabestrom zu finden sind und addieren diese auf.

ArithTokenizer.java

```
43
44    Token getInt() throws IOException {
45      int res=0;
46      while (next>=0 && Character.isDigit((char)next)){
47        res=res*10+next-48;
48          next = reader.read();
49      }
50
51      return new Zahl(res);
52    }
53
```

Schließlich schreiben wir uns zwei statische Methoden zur einfachen Benutzung des Tokenizers.

ArithTokenizer.java

```
54
55    static List<Token> tokenize(String inp) throws Exception{
56      return tokenize(new StringReader(inp));
57    }
58
59    static List<Token> tokenize(Reader inp) throws Exception{
60        return new ArithTokenizer(inp).tokenize();
61    }
62  }
```

So lassen sich einfache Test in Jugs schnell durchführen:

```
sep@swe10:~/fh/prog2/examples> java Jugs
     -- --  --  ----   ---      ----------------------------------------
     || ||  || ||  || ||__      Jugs: the interactive Java interpreter
     || ||__|| ||__||  __||     Copyright (c) 2003 Sven Eric Panitz
     ||         ___||           World Wide Web:
||   ||                                 http://www.tfh-berlin.de/~panitz
\\__//  Version: February 2003  ----------------------------------------


>ArithTokenizer.tokenize("888+0009   *+-/   9")
[888, add, 9, mult, add, sub, div, 9]
>ArithTokenizer.tokenize("888+0009   *uiu**+- 09")
java.lang.Exception: unexpected Token found: 'u'
>
```

Wir können die unterschiedlichsten Klassen aus diesem Skript für eine kleine Anwendung zusammensetzen. Wir lesen eine Datei mit der Endung `.arith`, in der ein arithmetischer Ausdruck steht. Die Datei wird gelesen, die Tokenliste erzeugt und diese mit dem Parser geparst. Der erzeugte Ableitungsbaum wird in eine`png`-Bilddatei geschrieben. Wir sehen ein paar praktische statische Methoden zum Testen in Jugs und eine Hauptmethode vor:

ArithParsTreeBuilder.java

```
1  package name.panitz.parser;
2  import java.io.*;
```

```
3  import javax.imageio.ImageIO;
4  import javax.swing.*;
5  import name.panitz.data.tree.Tree;
6  import name.panitz.data.tree.DisplayTree;
7  import static name.panitz.data.tree.TreeToFile.treeToFile;
8
9  class ArithParsTreeBuilder{
10
11   public static void main(String [] args) throws Exception{
12     final String fileName = args[0];
13     writeParsTreeFile(new FileReader(fileName+".arith"),fileName);
14   }
15
16   static void writeParsTreeFile(Reader reader,String fileName)
17                                                  throws Exception{
18     treeToFile(parsArith(reader),fileName);
19   }
20
21   public static Tree<String> parsArith(Reader reader)
22                                                  throws Exception{
23     final ParseResult<Tree<String>> res
24      = new TreeArith().parse(ArithTokenizer.tokenize(reader));
25     return res.getResult();
26   }
27
28   public static DisplayTree parseAndShow(String inp)
29                                        throws Exception{
30     return new DisplayTree(parsArith(new StringReader(inp)));
31   }
32
33   public static void parseAndShowInFrame(String inp)
34                                        throws Exception{
35     JComponent c = ArithParsTreeBuilder.parseAndShow(inp);
36     JFrame f = new JFrame();
37     f.getContentPane().add(c);
38     f.pack();
39     f.setVisible(true);
40   }
41 }
```

Aufgabe 53 (2 Punkte) In dieser Aufgabe sollen Sie den Parser für arithmetische Ausdrücke um geklammerte Ausdrücke erweitern. Hierzu sei die Grammatik für die Regel *multExpr* wie folgt geändert:

$$\mathit{multExpr} ::= \mathit{atomExpr}\ \mathit{multOp}\ \mathit{multExpr} | \mathit{atomExpr}$$
$$\mathit{atomExpr} ::= \texttt{(}\mathit{addExpr}\texttt{)} | \mathit{zahl}$$

Hierzu sei die entsprechende zusätzliche Tokenaufzählung für die zwei Klammersymbole gegeben:

Parentheses.java

```
package name.panitz.parser;
public enum Parentheses implements Token{lpar,rpar;}
```

Zusätzlich gegeben sei die die folgende Funktion:

DoParentheses.java

```
package name.panitz.parser;

public class DoParentheses
  implements
    Function<Pair<Parentheses,Pair<Integer,Parentheses>>,Integer>{
  public Integer
            apply(Pair<Parentheses,Pair<Integer,Parentheses>> x){
    return x.getE2().getE1();
  }
}
```

a) Schreiben Sie eine Klasse `AtomExprEval`, die `Parser<Integer>` implementiert und der Regel für *atomExpr* entspricht.

b) Ändern Sie die Klasse *MultExprEval*, so daß sie der geänderten Grammatikregel entspricht.

c) Erweitern Sie die Klasse `ArithTokenizer`, so daß er auch die beiden Klammersymbole erkennt.

14.4 Parsergeneratoren

Im letzem Abschnitt haben wir gesehen, daß die Aufgabe einen Parser für eine kontextfreie Grammatik zu schreiben eine mechanische Arbeit ist. Nach bestimmten Regeln läßt sich eine Grammatik direkt in einen Parser umsetzen. In unserem in Java geschriebenen Parser führte jede Regel zu genau einer Methode.

Aus der Tatsache, daß einen Parser für eine Grammatik zu schreiben eine mechanische Tätigkeit ist, folgt sehr schnell, daß man diese gerne automatisiert durchführen möchte und in der Tat schon sehr bald gab es Werkzeuge, die diese Aufgabe übernehmen konnten, sogenante Parsergeneratoren. Heutzutage schreibt kaum ein Programmierer in einer Sprache wie Java einen Parser von Hand, sondern läßt den Parser aus der Definition der Grammatik generieren.

Der wohl am weitesten verbreitete Parsergenerator ist Yacc[Joh75]. Wie man seinem Namen, der für *yet another compiler compiler* steht, entnehmen kann, war dieses bei weiten nicht der erste Parsergenerator. Yacc erzeugt traditionell C-Code; es gibt aber mitlerweile auch Versionen, die Java-Code generieren. Ein weitgehendst funktionsgleiches Programm zu Yacc heißt *Bison*. Die lexikalische Analyse steht im im Zusammenspiel mit Yacc und Bison jeweils auch ein Generatorprogramm zur Verfügung, daß einen Tokenizer generiert. Dieses Programm heißt `lex`.

Weitere modernere und auf Java zugeschnittene Parsergeneratoren sind: `yavacc`, `antlr` und `gentle`[Gru01].

14.4.1 javacc

Wir wollen uns die Mühe machen, einen Parser mit einem Parsergenerator zu beschreiben. Hierzu nehmen wir den Parsergenerator `javacc`. Dieser Parsergenerator kommt mit zwei Progammen:

- `jjtree`: Dieses Programm generiert Klassen, mit denen der Ableitungsbaum eines Parses dargestellt werden kann. Es nimmt als Eingabedatei eine Beschreibung der Grammatik. Diese Datei hat die Endung `.jjt`. Sie generiert Klassen für die Baumknoten. Zusätzlich generiert es eine Datei mit der Endung `.jj`. die im nächsten Schritt Eingabe für den eigentlichen Parsergenerator ist.
- `javacc`: Dieses Programm generiert eine Klasse für den eigentlichen Parser. Seine Eingabe ist eine Beschreibung der Grammatik in einer Datei mit Endung `.jj`. Sie generiert eine Klasse für den Parser und eine für die Token, die in der Grammatik spezifiziert werden. Zusätzlich werden Klassen zur Fehlerbehandlung generiert.

Eine wichtige Frage bei der Benutzung eines Parsergenerators ist, nach welcher Strategie der generierte Parser vorgeht. Wenn es ein rekurisv absteigender Parser ist, so dürfen wir keine Linksrekursion in der Grammatik haben. Wenn der generierte Parser kein *backtracking* hat, dann muß er mit Betrachtung des aktuellen Tokens wissen, welche Regelalternative zu wählen ist. Die Alternativen brauchen entsprechend disjunkte Startmengen. Bei generierten Parsern mit der Strategie des Schieben und Reduzierens, kann es sein, daß es zu sogenannten *shift-reduce* Konflikten kommt, wenn nämlich bereits reduziert werden kann, aber auch nach einem *shift*-Schritt reduziert werden kann.

Desweiteren ist wichtig, was für Konstrukte die Eingabegrammtik zuläßt. Konfortable moderne Parsergeneratoren lassen heute die erweiterte Backus-Naur-Form zur Beschreibung der Grammatik zu.

`javacc` generiert einen rekursiv absteigenden Parser ohne `backtracking`. Es darf also keine linksrekursive Regel in der Grammatik sein.

`javacc` integriert einen Tokenizer, d.h. mit der Grammatik werden auch die Token definiert.

Beispiel:
Als erstes Beispiel wollen wir für unsere Sprache astronomischer Aussagen einen Parser generieren lassen. Hierzu schreiben wir eine Eingabegrammatikdatei mit den Namen `sms.jjt`.[3].

Die Datei `sms.jjt` beginnt mit einer Definition der Klasse, die den Parser enthalten soll. In unserem Beispiel soll die generierte Klasse `SMS` heißen:

sms.jj

```
1 PARSER_BEGIN(SMS)
2 public class SMS {
3 }
4 PARSER_END(SMS)
```

Anschließend lassen sich die Token der Sprache definieren. In unserem Beispiel gibt es 8 Wörter. Wir können für `javacc` definieren, daß Groß-und-Kleinschreibung für unsere Wörter irrelevant ist.

[3]SMS stehe hier für *Sonne Mond und Sterne*.

sms.jj

```
5  TOKEN [IGNORE_CASE] :
6  { <MOON: "moon">
7  | <MARS: "mars">
8  | <MERCURY: "mercury">
9  | <PHOEBUS: "phoebus">
10 | <DEIMOS: "deimos">
11 | <ORBITS: "orbits">
12 | <IS: "is">
13 | <A: "a">
14 }
```

Zusätzlich gibt es in `javacc` die Möglichkeit anzugeben, was für Zwischenraum zwischen den Token stehen darf. In unserem Beispiel wollen wir Leerzeichen, Tabularturzeichen und Zeilenendezeichen zwischen den Wörtern als Trennung zulassen:

sms.jj

```
15 SKIP :
16 { " "
17 | "\t"
18 | "\n"
19 | "\r"
20 }
```

Und schließlich und endlich folgen die Regeln für die Grammatik:

sms.jjt

```
21 void start() : {}
22 {
23   nounPhrase() verbPhrase()
24 }
25
26 void nounPhrase() : {}
27 {
28   noun()|article() noun()
29 }
30
31 void noun() : {}
32 { <MOON>|<MARS>|<MERCURY>|<PHOEBUS>|<DEIMOS>
33 }
34
35 void article() : {}
36 {<A>}
37
38 void verbPhrase() : {}
39 {
40 verb() nounPhrase()
41 }
42
43 void verb() : {}
44 {
```

```
45  <IS>|<ORBITS>
46 }
```

Aus dieser Datei `sms.jj` lassen sich jetzt mit dem Programm `jjtree` Klassen für die Darstellung des Ableitungsbaums generieren:

```
sep@swe10:~/fh/prog2/beispiele/javacc/sms> ls
sms.jjt
sep@swe10:~/fh/prog2/beispiele/javacc/sms> jjtree sms.jjt
Java Compiler Compiler Version 2.1 (Tree Builder)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "jjtree" with no arguments for help)
Reading from file sms.jjt . . .
File "Node.java" does not exist.  Will create one.
File "SimpleNode.java" does not exist.  Will create one.
Annotated grammar generated successfully in sms.jj
sep@swe10:~/fh/prog2/beispiele/javacc/sms> ls
JJTSMSState.java  SMSTreeConstants.java  sms.jj
Node.java         SimpleNode.java        sms.jjt
sep@swe10:~/fh/prog2/beispiele/javacc/sms>
```

Wie zu sehen ist, werden Klassen für Baumknoten generiert (`SimpleNode.java`) und eine Eingabedatei für den eigentlichen Parsergenerator (`sms.jj`). Jetzt können wir diesen Parser generieren lassen:

```
sep@swe10:~/fh/prog2/beispiele/javacc/sms> javacc sms.jj
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file sms.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
sep@swe10:~/fh/prog2/beispiele/javacc/sms> ls
JJTSMSState.java     SMSConstants.java      SimpleNode.java     sms.jjt
Node.java            SMSTokenManager.java   Token.java
ParseException.java  SMSTreeConstants.java  TokenMgrError.java
SMS.java             SimpleCharStream.java  sms.jj
sep@swe10:~/fh/prog2/beispiele/javacc/sms>
```

Die generierten Javaklassen lassen sich übersetzen:

```
sep@swe10:~/fh/prog2/beispiele/javacc/sms> javac *.java
sep@swe10:~/fh/prog2/beispiele/javacc/sms> ls
JJTSMSState.class     SMSConstants.class      SimpleNode.class
JJTSMSState.java      SMSConstants.java       SimpleNode.java
Node.class            SMSTokenManager.class   Token.class
Node.java             SMSTokenManager.java    Token.java
ParseException.class  SMSTreeConstants.class  TokenMgrError.class
ParseException.java   SMSTreeConstants.java   TokenMgrError.java
SMS.class             SimpleCharStream.class  sms.jj
SMS.java              SimpleCharStream.java   sms.jjt
sep@swe10:~/fh/prog2/beispiele/javacc/sms>
```

Bevor wir den Parser benutzen können, schauen wir uns die generierter Parserdatei `SMS.java` in Auszügen einmal an:

```
/* Generated By:JJTree&JavaCC: Do not edit this line. SMS.java */
public class SMS/*bgen(jjtree)*/
implements SMSTreeConstants, SMSConstants {/*bgen(jjtree)*/

  protected static JJTSMSState jjtree = new JJTSMSState();

  static final public void start() throws ParseException {
                /* bgen(jjtree) start */
  SimpleNode jjtn000 = new SimpleNode(JJTSTART);
  boolean jjtc000 = true;
  jjtree.openNodeScope(jjtn000);
    try {
      nounPhrase();
      verbPhrase();
    } catch (Throwable jjte000) {

...



  static final public void nounPhrase() throws ParseException {
                    /* bgen(jjtree) nounPhrase */
  SimpleNode jjtn000 = new SimpleNode(JJTNOUNPHRASE);
  boolean jjtc000 = true;
  jjtree.openNodeScope(jjtn000);
    try {
      switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
      case MOON:
      case MARS:
      case MERCURY:
      case PHOEBUS:
      kcase DEIMOS:


  public SMS(java.io.InputStream stream) {
    if (jj_initialized_once) {


....

```

Als erstes ist festzustellen, daß der generierte Parser kein gutes objektorientiertes Design hat. Es gibt für jede Regel der Grammatik eine Methode, allerdings sind alle Methoden statisch.

Es gibt einen Konstruktor dem ein Objekt des Typs `java.io.InputStream`, in dem die zu parsende Eingabe übergeben wird.

Alle Methoden können im Fehlerfall eine `ParseException` werfen, die angibt, wieso eine Eingabe nicht geparst werden konnte.

Es gibt ein Feld `jjtree`des Typs `JJTSMSState`, in welchem das Ergebnis des Parses gespeichert ist. Diese Klasse enthält eine Methode `Node rootNode()`, mit der die Wurzel des Ableitungsbaums erhalten werden kann.

Damit können wir eine Hauptmethode schreiben, in der wir den generierten Parser aufrufen.

```
1 public class MainSMS {
2   public static void main(String args[]) throws ParseException {
3     SMS parser = new SMS(System.in);
4     parser.start();
5     ((SimpleNode)parser.jjtree.rootNode()).dump("");
6   }
7 }
```

Als Eingabestrom definieren wir die Tastatureingabe. Wir benutzen die Methode `dump`, um den Ableitungsbaum auf dem Bilschirm auszugeben. Wir können nun das Hauptprgramm starten und mit der Tastatur Sätze eingeben:

```
sep@swe10:~/fh/prog2/beispiele/javacc/sms> javac *.java
sep@swe10:~/fh/prog2/beispiele/javacc/sms> java MainSMS
a moon orbits mars
start
 nounPhrase
  article
  noun
 verbPhrase
  verb
  nounPhrase
   noun
sep@swe10:~/fh/prog2/beispiele/javacc/sms> java MainSMS
mars is a red planet
Exception in thread "main" TokenMgrError:
Lexical error at line 1, column 11.  Encountered: "r" (114), after : ""
        at SMSTokenManager.getNextToken(SMSTokenManager.java:447)
        at SMS.jj_ntk(SMS.java:298)
        at SMS.noun(SMS.java:84)
        at SMS.nounPhrase(SMS.java:50)
        at SMS.verbPhrase(SMS.java:133)
        at SMS.start(SMS.java:12)
        at MainSMS.main(MainSMS.java:4)
sep@swe10:~/fh/prog2/beispiele/javacc/sms>
```

Aufgabe 54 Fügen Sie der Datei `sms.jjt` folgende Zeilen am Anfang ein:

```
1 options {
2    MULTI=true;
3    STATIC=false;
4 }
```

Generieren Sie mit `jjtree` und `javac` den Parser neu. Was hat sich geändert?

Kapitel 15

Java und der Rest der Welt

15.1 I18N, L12N

15.2 Socketkommunikation

15.3 RMI

Java bietet die Möglichkeit, programme verteilt auf verschiedenen Rechnern laufen zu lassen. Es läuft dabei auf mehreren Rechnern eine Instanz der virtuellen Maschine. Es ist dann möglich, von der einen Maschine Methoden auf Objekten, die auf der anderen Maschinen existieren, aufzurufen. Dieses Prinzip wird RMI bezeichnet. Es steht für *remote method invocation*. Hierzu müssen die Parameter der aufgerufenen Methode über ein Netzwerk an die entfernte viruelle maschine geschickt werden und das Ergebnis der Methode ebenso wieder über das Netzwerk an die aufrufende Methode zurückgeschickt werden. RMI realisiert ein Client-Server Modell.

Schnittstellen sind in der Modellierung einer RMI Anwendung von entscheidener Bedeutung. Über eine Schnittstelle definiert der Server, welche Funktionalität er anbietet. Eine Schnittstelle, die definiert, was ein RMI Servee für Methoden zur Verfügung stellt, leitet von der Schnittstelle `java.rmi.Remote` ab.

Beginnen wir mit einer ersten einfachen Anwendung. Unser Server soll eine Methode `getDate` anbieten, die das aktuelle Datum mit Uhrzeit vom Server zurückgibt. Dieses läßt sich einfach definieren:

GetDate.java

```
package name.panitz.rmi.date;

import java.util.Date;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface GetDate extends Remote {
    Date getDate() throws RemoteException;
}
```

Jede Methode, die ein RMI Server über das Netzwerk als entfernte Methode anbietet, muß in seiner `throws`-Klausel die Ausnahmeklasse `RemoteException` auflisten.

Eine Schnittstelle, die `Remote` erweitert läßt sich wie jede normale Schnittstelle in Java erweitern. Es muß auf keinster Weise berücksichtigt werden, daß die Methoden über ein Netzwerk von entfernten virtuellen Maschinen aufgerufen werden. Für unser erstes Beispiel genügt folgende naheliegende Implementierung.

GetDateImpl.java

```
1 package name.panitz.rmi.date;
2 import java.util.Date;
3
4 public class GetDateImpl implements GetDate {
5   public Date getDate() {return new Date();}
6 }
```

15.3.1 RMI Server

Mit der Schnittstelle `GetDate` haben wir spezifiziert, welche Funktionalität übers Netz angeboten werden soll. Die Implementierung setzt diese Funktionalität um. Jetzt brauchen wir noch ein Programm, das diese Funktionalität auch im Netz bekannt macht und anbietet; entsprechende Methodenaufrufe aus dem Netz entgegennimmt, ausführt und das Ergebnis an die aufrufende Maschine zurücksendet. Hierzu ist eine Serveranwendung zu schreiben. Diese benötigt eine Instanz der implementierenden Klasse, und muß diese für die Benutzung unter einem Namen für das Netzwerk registrieren. Hierzu gibt es in Paketen `java.rmi.registry` bzw. `java.rmi.server` drei entscheidene Klassen:

- `UnicastRemoteObject`: die statische Methode `exportObject` dieser Klasse wandelt ein Objekt in ein Instanz um, die geeignet ist, über das Netz aufgerufen zu werden. Dahinter verbirgt sich tatsächlich unter der Hand, der ganze Code, der für die Vorbereitung der Kommunikation notwendig ist.
- `LocateRegistry`: hier kann mit der statischen Methode `getRegistry` eine Instanz der auf dem Server laufenden Registrierumgebung für RMI Objekte erhalten werden.
- `Registry`: Objekte dieser Klasse haben die Methode `bind`, in der `Remote` Objekte in der RMI Umgebung bekannt gemacht werden können. Hierbei bindet man einen beliebigen Namen an das `Remote` Objekt. Unter diesen Namen können Client-Applikationen auf das `Remote`-Objekt zugreifen.

Für unser kleines erstes Beispiel einer Servers, der die Zeit angeben kann, ergibt das folgenden Code. Zunächst wird eine Instanz der implementierenden Klasse erzeugt. Diese wird als ein Remote Objekt exportiert, dann wird die RMI-Registry geholt, auf der dann das Objekt bekannt gemacht wird.

DateServer.java

```
1 package name.panitz.rmi.date;
2
3 import java.rmi.registry.Registry;
4 import java.rmi.registry.LocateRegistry;
5 import java.rmi.RemoteException;
```

```
6  import java.rmi.server.UnicastRemoteObject;
7
8  public class DateServer {
9    public static void main(String args[]) {
10     try {
11       GetDateImpl obj = new GetDateImpl();
12       GetDate stub = (GetDate) UnicastRemoteObject.exportObject(obj, 0);
13       Registry registry = LocateRegistry.getRegistry();
14       registry.bind("date", stub);
15       System.err.println("Server ready");
16     } catch (Exception e) {
17       System.err.println("Server exception: " + e.toString());
18       e.printStackTrace();
19     }
20   }
21 }
```

Die drei bis hierhin geschriebenen Javadateien sind ganz normal mit dem Javacompiler zu übersetzen. Bevor jedoch die Hauptmethode von `DateServer` gestartet werden kann, ist die RMI-Registrierung zu starten. Hierzu liefert das JDK das Programm `rmiregistry` mit. Dieses ist auf dem Servermaschine zunächst einmal zu starten. Dann ist das Programm `DateServer` zu starten. Diesem ist aber als Umgebungsparameter mitzugeben, wo der Server seine Javaklassen findet. Hier reicht der Klassenpfad nicht aus. Es ist das Attribut `java.rmi.server.codebase` zu setzen. Insgesammt erhalten wir folgende kleine Session auf der Kommandozeile:

```
sep@pc305-3:~/fh/java/student/classes> rmiregistry &
[1] 5492
sep@pc305-3:~/fh/java/student/classes> java -Djava.rmi.server.codebase=fil
e:///home/sep/fh/java/student/classes/ name.panitz.rmi.date.DateServer &
[2] 5502
sep@pc305-3:~/fh/java/student/classes> Server ready

sep@pc305-3:~/fh/java/student/classes>
```

15.3.2 RMI Client

Unser Server ist jetzt bereit, Methodenaufrufe zu empfangen. Diese werden von einem Clientprogramm aus unternommen. Hierzu ist ähnlich wie im Serverprogramm zu verfahren. Für die RMI-Registry des Servers ist eine repräsentierende Instanz zu erzeugen, in der ist unter dem registriertem Namen nach dem angebotenen entfernten Objekt zu fragen. Schließlich kann dieses wie ein ganz normales Javaobjekt benutzt werden. Es ist dabei nur zu beachten, daß eventuell eine Ausnahme auftritt.

Für unser Beispiel schreiben wir eine kleine GUI-Anwendung. Der meiste Code beschäftigt sich dabei mit dem GUI. Die eigentlichen Aufrufe an das auf einer anderen Maschine laufende Objekt, bestehen aus nur drei Zeilen.

DateClient.java

```
1  package name.panitz.rmi.date;
2  import javax.swing.*;
3  import java.awt.event.*;
```

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class DateClient extends JPanel {
  JButton getButton = new JButton("get new time from server");
  JLabel l    = new JLabel();
  DateClient(String host){
    add(getButton);
    add(l);
    try {
      Registry registry = LocateRegistry.getRegistry(host);
      final GetDate stub = (GetDate) registry.lookup("date");
      l.setText(""+stub.getDate());

      getButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
          try {
            l.setText(""+stub.getDate());
          }catch (Exception e1){}
        }
      });
    }catch (Exception e) {}
  }

  public static void main(String[] args) {
    JFrame f = new JFrame();
    f.add(new DateClient(args[0]));
    f.pack();
    f.setVisible(true);
  }
}
```

15.3.3 Serialisierbarkeit

Um entfernte Methode aufrufen zu können, müssen die Argumente des Methodenaufrufs über das Netz evrschickt werden und ebenso das Ergebnis zurückgeschickt werden. Hierzu müssen sich die entsprechenden Daten in eine für das verschicken geeignete Art umwandeln lassen, sie müssen serialisierbar sein. Java kennt das Konzept der Serialisierbarkeit und drückt es über die Schnittstelle `Serializable` aus. Die einige Einschränkung für RMI besteht also in der Serialisierbarkeit der in der methodensignatur beteiligten Typen.

15.3.4 Verteilte Spielanwendung

Da im letzten Abschnitt sich das Schreiben verteilter Anwendungen in Java als so einfach erwiesen hat, wollen wir in diesem Abschnitt eine weitere kleine verteilte Anwenung schreiben. Dieses Mal sollen die entfernten Methoden auch Parameter bekommen. Damit können Clients dem Server Daten schicken, die anderen Clients dann auch vom Server erfragen können. Somit

können die Clients sich gegenseitig Nachrichten zukommen lassen, die grundlage zum Beispiel eines jedem Chatprogramms.

Wir wollen aber keine Chatprogramm entwickeln, sondern eine kleine Anwendung, in der die Clients einen Punkt auf einer zweidimensionalen Fläche plazieren und bewegen können. Die einzelnen Clients sehen dabei nicht nur ihre eigenen Punkte sondern auch die Punkte der anderen Clients.

Die entsprechende `Remote`-Schnittstelle kommt mit drei Methoden aus:

- eine zum Erzeugen eines neuen Balls, wobei jeder Ball durch einen Namen identifiziert wird.
- eine zum Bewegen eines Balles mit einem bestimmten Namen.
- und eine zum Erfragen der Position eines jeden Balls, der existiert.

Wir bekommen die folgende Schnittstelle:

BallGame.java

```
1  package name.panitz.rmi.game;
2
3  import java.rmi.*;
4  import java.util.Map;
5
6  public interface BallGame extends Remote {
7
8    void createBall(String name) throws RemoteException;
9    void move(String name,int x,int y) throws RemoteException;
10   Map<String,java.awt.Dimension> getBalls()throws RemoteException;
11 }
```

Für die x-, y-position der Bälle mißbrauchen wir die Klasse `Dimension`, die eine 2-dimensionale Position speichern kann, auch wenn die entsprechenden Felder `width` und `hieght` statt x und y heißen.

Auch die Implementierung dieser Schnittstelle läßt sich relativ trivial bewerkstelligen. Ein privates Feld speichert alle Namens- Dimensionspaare in einer Abbildung. Die Methoden `createBall` und `move` manipulieren diese Abbildung.

BallGameImpl.java

```
1  package name.panitz.rmi.game;
2  import java.util.*;
3
4  import java.rmi.registry.Registry;
5  import java.rmi.registry.LocateRegistry;
6  import java.rmi.RemoteException;
7  import java.rmi.server.UnicastRemoteObject;
8  import java.awt.Dimension;
9
10 public class BallGameImpl implements BallGame {
11   private Map<String,java.awt.Dimension> bs
12     = new HashMap<String,java.awt.Dimension>();
```

```
13
14    public void createBall(String name){
15      bs.put(name,new Dimension(10,10));}
16
17    public void move(String name,int x,int y){
18      Dimension d=bs.get(name);
19      if (d!=null) d.setSize(d.width+x,d.height+y);
20    }
21
22    public Map<String,java.awt.Dimension> getBalls(){return bs;}
23  }
```

Der Server für diese Anwendung sieht exakt aus, wie der Server für die Datumsanwendung zuvor.

BallGameServer.java

```
1  package name.panitz.rmi.game;
2
3  import java.rmi.registry.Registry;
4  import java.rmi.registry.LocateRegistry;
5  import java.rmi.RemoteException;
6  import java.rmi.server.UnicastRemoteObject;
7  public class BallGameServer {
8    public static void main(String args[]) {
9      try {
10       BallGameImpl obj = new BallGameImpl();
11       BallGame stub = (BallGame) UnicastRemoteObject.exportObject(obj, 0);
12       Registry registry = LocateRegistry.getRegistry();
13       registry.bind("ballgame", stub);
14       System.err.println("Server ready");
15     } catch (Exception e) {
16       System.err.println("Server exception: " + e.toString());
17       e.printStackTrace();
18     }
19   }
20 }
```

Es bleibt das spannenste: der Client. Auch in diesem Beispiel wieder eine GUI-Anwendung.

BallGameClient.java

```
1  package name.panitz.rmi.game;
2  import javax.swing.*;
3  import java.awt.event.*;
4  import java.awt.Graphics;
5  import java.awt.Dimension;
6  import java.awt.Color;
7
8  import java.rmi.registry.LocateRegistry;
9  import java.rmi.registry.Registry;
10 import static java.awt.event.KeyEvent.*;
```

```
11
12 import name.panitz.animation.AnimatedJPanel;
13
14 public class BallGameClient extends AnimatedJPanel {
15   final String name;
16   BallGame game=null;
17   final int width = 400;
18   final int height = 300;
19
20   BallGameClient(String host,String n){
21     this.name=n;
22     setFocusable(true);
23     try {
24       Registry registry = LocateRegistry.getRegistry(host);
25       game = (BallGame) registry.lookup("ballgame");
26     } catch (Exception e) {}
27
28     addMouseListener(new MouseAdapter(){
29       public void mouseClicked(MouseEvent e){
30         try{game.createBall(name);}catch(Exception e1){}
31       }
32     });
33
34     addKeyListener(new KeyAdapter(){
35       public void keyReleased(KeyEvent e){
36         try{
37           switch (e.getKeyCode()){
38             case VK_DOWN: game.move(name,0,2);break;
39             case VK_LEFT:game.move(name,-2,0);break;
40             case VK_RIGHT:game.move(name,2,0);break;
41             case VK_UP:game.move(name,0,-2);break;
42           }
43         }catch(Exception e1){System.out.println(e);}
44       }
45     });
46   }
47
48   public void paintComponent(Graphics g){
49     g.setColor(Color.WHITE);
50     g.fillRect(0,0,width,height);
51     try{
52       for (Dimension d:game.getBalls().values()){
53         g.setColor(Color.RED);
54         g.fillOval(d.height,d.width,10,10);
55       }
56     }catch (Exception e){}
57   }
58
59   public Dimension getPreferredSize(){
60     return new Dimension(width,height);}
```

```
61
62    public static void main(String[] args) {
63      JFrame f = new JFrame();
64      f.add(new BallGameClient(args[0],args[1]));
65      f.pack();
66      f.setVisible(true);
67    }
68    public void move(){}
69  }
```

15.4 Java und Datenbanken

Jede einigermaßen interessante Anwendung besonders im Geschäftsbereich verarbeitet an einem bestimmten Punkt persistente Daten, d.h. Daten, die über die Laufzeit der Anwendung hinaus existieren. Für kleine Datenmengen reicht es in der Regel aus, diese in einer Datei persistent zu speichern, aber schon mittelgroße Datenmengen werden in einer Datenbanksystem gehalten. Die gängiste Form von Datenbanksystemen sind relationale Datenbanken, die als Datenbankanfragesprache die deklarative Programmiersprache SQL benutzen.

Im Umgang mit Datenbanken gibt es prinzipiell zwei Varianten für Java:

- auf eine bestehende relationale Datenbank soll zugegriffen werden. In diesem wahrscheinlich häufigeren Fall definiert die Datenbank die Art der Daten und die Javaanwendung orientiert sich an dieser.
- eine vollkommen neue Anwendung soll objektorientiert entworfen werden. Hier stehen die Objekte im Mittelpunkt. Objekte sollen dann auch in einer Datenbank persistent gehalten werden. Nun sind Objekte auf Tabellen einer relationalen Datenbank abzubilden. Man spricht von OR-Mapping (object relational mapping)

15.4.1 JDBC Verbindungen

Java bietet ein API an, um SQL-Queries an eine Datenbank zu verschicken und somit Daten aus dieser zu extrahieren oder die Daten zu manipulieren. Im einfachsten Fall werden die SQL-Ausdrücke in Java als String erzeugt und diese Stringwerte an eine Datenbank verschickt. Unter diesen Gesichtspunkt reicht es aus, in der Datenbankvorlesung gut aufzupassen, um mit Java auf Daten einer Datenbank zuzugreifen.

Wie immer, wenn es um den Zugriff auf Datenquellen in der externen Welt geht, sind die Schwierigkeiten keine programmiersprachlichen, sondern liegen in der Beherrschung der Spezifika der darunterliegenden Datenquellen in unserem Fall der Datenbank.

MusicDB.java

```
1  package name.panitz.sheetMusic.db;
2  import java.sql.*;
3  public class MusicDB{
4    final public Connection con;
5    public MusicDB(String host,String database,String userid,String password)
6                                                    throws Exception {
```

```
    con = DriverManager
          .getConnection("jdbc:mysql://"+host+"/"+database,userid,password);
  }
```

SQL expressions

Table creation

MusicDB.java

```
  public void createTable(String tableName,String tableSpec)
        throws Exception {
    final Statement s = con.createStatement();
    s.execute ("create table "+tableName+" "+tableSpec);
    s.close ( );
  }
```

Insertion of rows

MusicDB.java

```
  public void insertRow(String tableName,String values )
                                                    throws Exception {
    final Statement s = con.createStatement();
    s.execute("insert into "+tableName+" values "+values);
    s.close ( );
  }
```

Queries

MusicDB.java

```
  public ResultSet query(String sqlExpr)
                                          throws Exception {
    return con.createStatement().executeQuery(sqlExpr);
  }
```

MusicDB.java

```
}
```

Transaktionen

Beispiellauf

Example.java

```
package name.panitz.sheetMusic.db;
public class Example{
  public static void main(String [] args) throws Exception{
    MusicDB db = new MusicDB("localhost","test","sep","stammel");
    try{
    db.createTable
```

```
33      ("composer"
34      ,"(last_name varchar(60),first_name varchar(60),biography text)" );
35     db.createTable
36      ("opus"
37      ,"(composer varchar(60),title varchar(256),opsNr varchar(20),url varchar(
38
39     db.insertRow
40      ("composer"
41      ,"('Legnani','Luigi','')");
42     db.insertRow
43      ("composer"
44      ,"('Giuliani','Mauro','')");
45     db.insertRow
46      ("composer"
47      ,"('Fossa','Francois de','')");
48
49     db.insertRow
50      ("opus"
51      ,"('Legnani'"+
52       ",'36 capricci per tutti i tuoni maggiori e minori per la chitarra'"+
53       ",'20'"+
54       ",'http://www.kb.dk/elib/noder/rischel/RIBS0501.pdf')");
55     db.insertRow
56      ("opus"
57      ,"('Legnani'"+
58       ",'Coro e rondo, Pensa alla patria nell opera L italiana in Algeri del R
59       ",'8a'"+
60       ",'http://www.kb.dk/elib/noder/rischel/RIBS0480.pdf')");
61     db.insertRow
62      ("opus"
63      ,"('Legnani'"+
64       ",'Introduction et thême'"+
65       ",'237'"+
66       ",'http://www.kb.dk/elib/noder/rischel/RIBS0490.pdf')");
67     }catch(Exception e){
68     }
69   }
70 }
```

15.5 JNI: Aufruf von C

15.6 Java unterwegs

Heutzutage tragen die meisten Menschen einen Computer in der Tasche, der in Speicherkapazität und Rechenleistung bereits einen Arbeitsplatzrechner von vor wenigen Jahren übertrifft: das Mobiltelephon auch Handy genannt. Auf den meisten heute handelsüblichen Mobiltelephonen ist auch Javas virtuelle Maschine vorhanden. Hier kann Java sein Ausführungsmodell wieder

perfekt nutzen. Ursprünglich war es entwickelt worden für Set-Top-Boxen im Fernsehbereich, erwies sich für kurze Zeit als Killerapplikation im Webbrowser und kann jetzt auf Mobiltelephonen reüssieren. Was in allen diesen drei Anwendungen identisch ist: es gibt Clients mit verschiedenste Hardware. Ziel ist es auf diesen Clients fremde Software laufen zu lassen, und dieses in einer sicheren Umgebung, die nur auf kontrollierte Art und Weise auf die Resourcen der Hardware zugreifen kann.

Nun untescheidet sich ein Mobiltelephon in vielen Punkten von einem Arbeitsplatzrechner:

- als Bildschirm existiert nur eine kleinste Fläche
- als Tastatur stehen in der Regel nur die Tasten eines Telephons zur Verfügung.
- der Prozessor ist einfacher gestaltet und unterstützt oft keine Fließkommaoperationen.
- die Kommunikation läuft auf eingeschränkte definierte Wege.
- Arbeitsspeicher ist wesentlich kleiner.
- statt einer großen Festplatte gibt es einebaute oder austauschbare Speicherchips.
- eingehende Anrufe sollten in jedem Fall die Oberhand gegenüber anderen Applikationen behalten.

Zieht man diese nicht marginale Unterschiede in Betracht, verwundert es nicht, daß auf einem Mobiltelephon nicht komplette Javaanwendungen, wie wir sie bisher entwickelt haben, laufen können. Die virtuelle wie sie auf Mobiltelephonen installiert ist, stellt nicht die komplette Javastandardbibliothek zur Verfügung. Sie hat ein stark eingeschränktes API. Die Gründe weshalb Standardklassen aus Java auf einem Mobiltelephon nicht zur Verfügung stehen, können unterschiedlichster Natur sein; z.B. lediglich um die Masse der Klassen zu reduzieren oder weil sie sich auf Hardware beziehen, wie sie so nicht im Mobiltelephon existiert.

Um all diesen Überlegungen Rechnung zu tragen, gibt es eine spezielle Java-API Version für Hardware mit eingeschränkten Resourcen: die Java ME. Hierbei handelt es sich um eine Version von Java, die im Programmiersprachlichen Sinne ein volles Java ist. Als Javaübersetzer wird der Standardübersetzer genommen, den wir bisher auch benutzt haben. Lediglich die verfügbaren Bibliotheken sind stark reduziert. Zusätzlich gibt es dafür Bibliotheken, die sich speziell auf die Bedürfnisse von Mobiltelephonen bezieht.

Um bequem Javaanwendungen für Mobiltelephone zu entwicken bieten Java eine kleine Entwicklungsumgebung an: das J2ME Wireless Toolkit. Diese hilft einem dabei Javaanwendungen für Telephone zu übersetzen, testen und als fertige Anwendung zu verpacken. [1]

15.6.1 Hello Mobile World

Genug der Vorrede, Zeit die erste Javaanwendung fürs Handy zu schreiben. So wie Anwendungen, die im Browser laufen sollen, von der Klasse `Applet` ableiten müssen, müssen Anwendungen, die im Handy laufen sollen von einer bestimmten Klasse ableiten: `javax.microedition.midlet.MIDlet`. Wir schreiben also ein sogenanntes Midlet:

[1] Wie man am Namen `J2ME` sieht, hat Java 5 noch nicht in die Welt der Kleingeräte einzug gehalten.

HelloMobileWorld.java

```
1 package name.panitz.mobile;
2
3 import javax.microedition.midlet.MIDlet;
4 import javax.microedition.lcdui.*;
5
6 public class HelloMobileWorld extends MIDlet{
```

Ähnlich den den Applets gibt es in Midlets Methoden, die das Verhalten beim Starten, Pausieren oder endgültigen Zerstören eines Midlets bestimmen. Für unsere erste kleine Anwenung reicht es aus, das Verhalten beim Starten zu definieren. Wir wollen auf dem Display einen Text plazieren. Hierzu erlaubt es ein Midlet sein Display in Form eines Objekts der Klasse `Display` zu erfragen. Diesem Display kann dann ein Objekt der Klasse `Form` hinzugefügt werden, dem wiederum verschiedenste Text- und Bildelemente zugefügt erden können. Man kann vielleicht in Analogie zu GUI-Anwendungen in Swing das `Display` mit einem `JFrame` und das `Form` mit einem `JPanel` vergleichen.

HelloMobileWorld.java

```
7   public void startApp() {
8     final Display display = Display.getDisplay(this);
9     final Form mainForm = new Form("TinyMIDlet");
```

Dem `Form`-Objekt läßt sich nun der gewünschte Text hinzufügen:

HelloMobileWorld.java

```
10    mainForm.append("Welcome to the world of MIDlets!");
```

Auch im Handy gibt es Ereignisse. Das Ereignisbehandlungskonzept ist dabei auf gleiche Weise realisiert, wie wir es bereits aus *AWT* und *Swing* kennen. Ein Ereignisbehandler für `Form`-Objekte sind Kommandoereignisse. Mit einer anonymen Klasse fügen wir dem `Form`-Objekt eine Behandlung des Kommandos `exit` hinzu, die bewirkt, daß das Midlet schließlich zerstört wird:

HelloMobileWorld.java

```
11    mainForm.setCommandListener(new CommandListener(){
12      public void commandAction(Command c, Displayable s) {
13        if (c.getCommandType() == Command.EXIT)
14          notifyDestroyed();
15       }
16    });
```

Das somit erstellte `Form`-Objekt kann nun dem Display zugefügt werden:

HelloMobileWorld.java

```
17    display.setCurrent(mainForm);
```

Schließlich ist noch das Kommando als `Exit` als Auswahlkommando dem Midlet hinzuzufügen.

HelloMobileWorld.java

```
    mainForm.addCommand(new Command("Exit", Command.EXIT, 0));
  }
  public void destroyApp(boolean b) {}
  public void pauseApp() {}
}
```

Übersetzen der drahtlosen Anwendung

über die Kommandozeile

```
javac \
  -bootclasspath \
 /home/sep/software/WTK2.2/lib/cldcapi10.jar\
:/home/sep/software/WTK2.2/lib/midpapi20.jar \
-source 1.3 -target 1.3 -d classes HelloMobileWorld.java
```

mit WTK Abbildung **??** zeigt die Simuation des Handies zu Testzwecken.

15.6.2 Pong fürs Handy

Auch wenn die Vorstellung etwas seltsam ist, daß zwei Spieler auf einem Handy gegeneinander spielen, wollen wir als Beispiel wieder das Telespiel `Pong` programmieren. Im entsprechenden Kapitel über GUI-Programmierung wurde bereits darauf geachtet, daß das Programm so in verschiedene Klassen gesplittet wird, daß möglichst viel Code benutzt werden kann, um die Ponganwendung fürs Telephon zu schreiben. So benutzen wir die Schnittstelle `AMovable`, um sie jetzt mit einer `paint`-Methode zu erweitern, die sich auf ein `Graphics`-Objekt des Pakets `javax.microedition.lcdui` bezieht.

MMovable.java

```

package name.panitz.mobile.pong;
import javax.microedition.lcdui.Graphics;

public interface MMovable extends name.panitz.games.AMovable{
  public void paintMeTo(Graphics g);
}
```

Ebenso benutzen wir die Klasse `ASimpleMovable` und erweitern diese um eine Implementierung der entsprechenden Zeichenmethode.

MSimpleMovable.java

```
package name.panitz.mobile.pong;
import javax.microedition.lcdui.Graphics;

public class MSimpleMovable extends name.panitz.games.ASimpleMovable
                            implements MMovable{
```

Abbildung 15.1: Handysimulation mit WTK2.2.

```
  public void paintMeTo(Graphics g){
    g.setColor(0x00FFFF);
    g.fillRect(x,y,width,height);
  }
}
```

MBall.java

```

package name.panitz.mobile.pong;
import javax.microedition.lcdui.Graphics;
```

```

public class MBall extends MSimpleMovable{
  public MBall(){}
  public MBall(int x,int y){this.x=x;this.y=y;moveX=3;}

  public void paintMeTo(Graphics g ){
    g.setColor(0x00FFFF);
    g.fillArc(x,y,width,height,0,360);
  }
}
```

MPaddle.java

```

package name.panitz.mobile.pong;

public class MPaddle extends MSimpleMovable{
  public MPaddle(int xPos){
    x=xPos;
    width=10;
    height=30;
    moveX=0;
    moveY=2;
  }
}
```

PongCanvas.java

```

package name.panitz.mobile.pong;

import javax.microedition.lcdui.*;

public class PongCanvas extends Canvas implements Runnable {
  Display display;
  MBall ball=null;
  MPaddle leftPaddle = new MPaddle(10);
  MPaddle rightPaddle= new MPaddle(getWidth()-20);
  final MMovable[] paddles = {leftPaddle,rightPaddle};
  int leftPoints=0;
  int rightPoints=0;
  static final int LEFT_PL=0;
  static final int RIGHT_PL=1;
  int nextPlayer=LEFT_PL;

  boolean paused = false;

  public PongCanvas(Display d) {display = d;}

  final int delay = 20;
  public void run() {
```

```
    while (!paused){
      move();
      repaint();
      try {
        Thread.sleep(delay);
      } catch (InterruptedException e) {}
    }
  }

  String msg = null;
  protected void paint(Graphics g) {
    int x = g.getClipX();
    int y = g.getClipY();
    int w = g.getClipWidth();
    int h = g.getClipHeight();

    // Draw the frame
    g.setColor(0xffffff);
    g.fillRect(x, y, w, h);

    paddles[0].paintMeTo(g);
    paddles[1].paintMeTo(g);
    if (null!=ball) ball.paintMeTo(g);

    g.setColor(0x000000);
    g.drawString(""+leftPoints,30,30,g.TOP|g.LEFT);
    g.drawString(""+rightPoints,getWidth()-30,30,g.TOP|g.RIGHT);


    g.setColor(0);
    g.drawRect(0, 0, getWidth()-1, getHeight()-1);
  }

  public void move(){
    rightPaddle.setX(getWidth()-20);
    if (ball!=null){
      if (ball.above(0) || ball.underneath(getHeight())){
        ball.reverseYDirection();
        ball.move();
      }

      if (ball.leftOf(0) ){
        ball=null;
        rightPoints=rightPoints+1;
        nextPlayer=RIGHT_PL;
      }

      else if (ball.rightOf(getWidth()) ){
        ball=null;
        leftPoints=leftPoints+1;
```

```
        nextPlayer=LEFT_PL;
      }

      else if (
         (    ball.rightOf(getWidth()-20)
          && ball.underneath(rightPaddle.getY())
          && ball.above(rightPaddle.getY()+rightPaddle.getHeight())
         )
         ||
         (    ball.leftOf(20)
          && ball.underneath(leftPaddle.getY())
          && ball.above(leftPaddle.getY()+leftPaddle.getHeight())
         )
        )
        ball.reverseXDirection();
    }

    paddles[0].move();
    paddles[1].move();
    if (ball!=null) ball.move();
    for (int i=0;i<2;i++){
      if (paddles[i].getY()<0) paddles[i].setY(0);
      if (paddles[i].getY()+paddles[i].getHeight()>getHeight())
        paddles[i].setY(getHeight()-paddles[i].getHeight());
    }
  }

  public void keyReleased(int keyCode) {
    int action = getGameAction(keyCode);
    switch (keyCode) {
      case KEY_NUM1:
        leftPaddle.setMoveY(-Math.abs(leftPaddle.getMoveY()));
        break;

      case KEY_NUM3:
        rightPaddle.setMoveY(-Math.abs(rightPaddle.getMoveY()));
        break;
    }
  }

  public void keyPressed(int keyCode) {
    int action = getGameAction(keyCode);
    switch (keyCode) {
      case KEY_NUM1:
        leftPaddle.setMoveY(Math.abs(leftPaddle.getMoveY()));
        break;
      case KEY_NUM3:
        rightPaddle.setMoveY(Math.abs(rightPaddle.getMoveY()));
        break;
      case KEY_NUM0:
```

```
        if (ball==null){
          switch (nextPlayer){
            case RIGHT_PL:
                    ball=new MBall(getWidth()-32,rightPaddle.getY()+10);
              ball.setMoveX(-ball.getMoveX());
              break;
            case LEFT_PL:
              ball=new MBall(22,leftPaddle.getY()+10);
              break;
            }
        }
        break;
    }
  }

  void destroy() {}

  boolean isPaused() {return paused;}
  void pause() {paused = true;}

  void start() {
    paused = false;
    display.setCurrent(this);
    new Thread(this).start();
  }
}
```

MobilePong.java

```

package name.panitz.mobile.pong;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class MobilePong extends MIDlet implements CommandListener {
  final Display display = Display.getDisplay(this);
  final PongCanvas canvas = new PongCanvas(display);

  private final Command exitCommand
    = new Command("Exit", Command.EXIT, 99);
  private final Command toggleCommand
    = new Command("Stop/Go", Command.SCREEN, 1);
  private final Command helpCommand
    = new Command("Help", Command.HELP, 2);
  private final Command aboutCommand
    = new Command("About", Command.HELP, 30);

  private final Form helpScreen = new Form("Pong Help");;
  private final String helpText
```

```
    = "0 = new ball\n1 = left paddle\n3 = right paddle";

  public  MobilePong() {
    canvas.addCommand(exitCommand);
    canvas.addCommand(toggleCommand);
    canvas.addCommand(helpCommand);
    canvas.addCommand(aboutCommand);
    canvas.setCommandListener(this);

    helpScreen.append("0 = new ball\n");
    helpScreen.append("1 = move left paddle up/down\n");
    helpScreen.append("3 = move right paddle up/down\n");
  }

  public void startApp() throws MIDletStateChangeException {
    canvas.start();
  }

  public void pauseApp() {canvas.pause();}

  public void destroyApp(boolean unconditional)
      throws MIDletStateChangeException {
    canvas.destroy();
  }

  public void commandAction(Command c, Displayable s) {
    final boolean wasPaused = canvas.isPaused();
    canvas.pause();
    if (c == toggleCommand) {if (wasPaused)canvas.start();}
    else if (c == helpCommand) showHelp();
    else if (c == exitCommand) {
      try {
        destroyApp(false);
        notifyDestroyed();
      } catch (MIDletStateChangeException ex) {}
    } else if (c == aboutCommand) About.showAbout(display);
  }

  void showHelp() {
    helpScreen.addCommand(toggleCommand);
    helpScreen.setCommandListener(this);
    display.setCurrent(helpScreen);
  }
}
```

About.java

```

package name.panitz.mobile.pong;
import javax.microedition.lcdui.*;
```

```

public class About {
  private static final String copyright
    = "Copyright (c) 2005 Sven Eric Panitz. All rights reserved.";

  public static void showAbout(Display display) {
    Alert alert = new Alert("About MIDP");
    alert.setTimeout(Alert.FOREVER);

    if (display.numColors() > 2) {
      String icon = (display.isColor())
                    ?"/icons/JavaPowered-8.png"
                    : "/icons/JavaPowered-2.png";
      try {
        Image image = Image.createImage(icon);
        alert.setImage(image);
      } catch (java.io.IOException x) {}
    }
    alert.setString(copyright);
    display.setCurrent(alert);
  }
}
```

Kapitel 16

Eine Beispielwebapplikation

16.1 Einführung

In diesem Java Abschlußkapitel zum Java Programierskript soll eine komplette Beispielapplikation entstehen. Hierbei werden wir viele der Techniken, die im Laufe des Skriptes eingeführt wurden, angewendet. Insbesondere werden typischer Weise XML-Dokumente unseren Weg kreuzen. Zusätzlich lernen wir noch zwei weitere Techniken kennen, die typisch für eine Webanwendung sind:

- Java Datenobjekte (JDO)
- Servlets

Wie im gesammten Skript fokussieren wir bei dieser Anwendung nicht auf die Aspekte der Softwaretechnik, werden also weder eine saubere Spezifikation, Modellierung noch Dokumentation oder Testumgebung erstellen, sondern allein auf die nakte Programmierung im Mittelpunkt stellen. Auch wird es keine gesonderten Gedanken zu I18N oder L12N geben. Allerdings wird die entstehende Applikation durch Verwendung der benutzten Techniken mehr oder weniger automatisch in weiten Teilen internationalisiert sein.

Insbesondere eine große Grundannahme erlaubt uns in dieser Beispielanwenung große Freiheiten: die Applikation entsteht auf der grünen Wiese vollkommen neu. Es soll keine bereits bestehenden Programmteile, Datensätze oder externen Komponenten geben. Diese Situation treffen wir in der Praxis leider selten an. Besonders im Zusammenhang mit Webapplikationen bestehen in der Praxis bereits Anwendungen und Daten, die in einem Projekt für den Webzugriff ertüchtigt werden sollen.

16.1.1 Die Applikation

Die entstehende Applikation soll Weblinks über Musiknoten verwalten. Auf vielen Webseiten finden sich mitlerweile Musiknoten zum Herunterladen. Eine gezielte Suche nach Noten im Netz ist in der Regel mühsam. Unsere Webapplikation soll ein Sammlung von Links zu Noten im Internet bereitstellen. Die Anwendung ist dabei im Wikistil gehalten, d.h. jedem soll es erlaubt sein, ohne vorherige Anmeldung neue Links in die Datenbank einzutragen oder bestehende Einträge zu verändern.

16.2 Java Datenobjekte (JDO)

Grundstütze der meisten Webapplikationen sind in der Regel persistente Daten, die in einer Datenbank gehalten werden. Dieses können z.B. Katalog und Kundendaten eines Internetshops, oder Angebots- und Bieterdaten eines Auktionshauses, Fahrplaninformation einer Fahrplanauskunft oder Informationen über ein Straßennetz für einen Routenplaner sein. Egal an was für einen Internetdienst man denkt, so wird man feststellen, daß er den Zugriff und die Manipulation persistenter Daten enthält.

In einem klassischen Programmierumfeld wäre eine zentrale Aufgabe beim Erstellen der Webapplikation demnach, die zugrunde liegende Datenbank zu modellieren, d.h. Datenbanktabellen möglichst in Normalform und mit geeigneten Schlüsselattributen zu definieren. In Java existiert eine Technik, die es uns erlaubt, möglichst wenig mit datenbanktechnischen Details konfrontiert zu sein, sondern auch in Punkten der Datenhaltung, objektorientiert zu denken. Diese Technik heißt *java data objects (jdo)* und ist im Java Specification Request 12 (JSR 12) (`http://java.sun.com/products/jdo/`) spezifiziert worden. Die Spezifikation gibt Schnittstellen vor, die eine Javaklasse erfüllen muß, damit Objekte dieser Klasse persistent gespeichert werden können. Die JDO-Spezifikation liefert keine Implementierung der Peristenz, sondern nur eine Spezifikation von Schnittstellen für persistente Objekte. Um JDO benutzen zu können, muß man also den kompletten Code, der sich mit Persistenz beschäftigt selbst schreiben. Dieser Code ist in der Regel technisch, langweilig und schematisch. Im Englischen spricht man von sogenannten *boilerplate code*. Solch Code, möchte man möglichst nicht selbst schreiben, sondern automatisch generieren lassen, was aufgrunde der schematischen Natur des Codes leicht möglich ist. Auch für JDO gibt es Implementierungen, die den Persistenzcode generieren und alle notwendigen Schnittstellen der JDP-Spezifikation implementieren. Einer dieser JDO-Implementierungen ist JPOX (`www.jpox.org`) . JPOX ist unter der *Open Source Apache 2 license* verfügbar. Wir werden in unserer Webapplikation JPOX verwenden. Die nächsten Abschnitte können als kleines Kochbuch zur Benutzung von JPOX angesehen werden.

16.2.1 Javaklasse Schreiben

Das Schöne bei der Benutzung von JDO mit JPOX ist, daß wir uns zunächst einmal überhaupt keine Gedanken über Persistenz machen müssen, sondern in herkömmlicher Weise eine Javaklasse schreiben können. So läßt sich eine naheliegende Klasse zur Darstellung von Weblinks schreiben. Zwei Stringfelder seien hierzu vorgesehen, eines, das die URL des Links enthält und eines, das eine kleine Beschreibung zu dem Link enthält:

WebLink.java

```
1 package name.panitz.sheetMusic;
2 import java.util.*;
3 public class WebLink {
4   public  String uri;
5   public  String description;
6   public WebLink(String des,String u){uri=u;description=des;}
7   public String toString(){
8     return "<a href=\""+uri+"\">"+description+"</a>";
9   }
```

Wir werden in der Applikation oft mit Listen von `WebLink`-Objekten arbeiten und sehen zur Bequemlichkeit eine Methode zum Erstellen einelementiger Listen vor.

WebLink.java

```
  public static List<WebLink> singleLink(String u,String des){
    return singleton(new WebLink(u,des));
  }

  public static <a> List<a> singleton(a x){
    final List<a> result = new ArrayList<a>();
    result.add(x);
    return result;
  }
```

Die Klasse `WebLink` können wir in gewohnter Weise übersetzen und benutzen.

Objekte der Klasse `WebLink` sollen persistent gespeichert werden. Damit das möglich ist, müßte sie die Schnittstelle `javax.jdo.spi.PersistenceCapable` implementieren. Das haben wir aber nicht gemacht. Mit JPOX läßt sich im Nachhinein, die Klassendatei von `WebLink` manipulieren, so daß sie im Nachhiniein doch die Schnittstelle implementiert.

16.2.2 Objektpersistenz definieren

Damit JPOX den entsprechenden Persistenzcode einfügen kann, ist in einer Datei zu spezifizieren, Objekte welcher Klassen persistent zu speichern sind. Dieses kann in Form einer XML-Datei spezifiziert werden. Die Spezifikation des Dokumenttyps für die XML-Datei findet sich in der JDO-Spezifikation. Die DTDs aus Javaspezifikationen finden sich auf den Webseiten von Sun (`http://java.sun.com/dtd/`) .

In unserem Fall sind wir zunächst sehr bescheiden und spezifizieren, daß die Klasse `WebLink` doch bitte die Möglichkeit der Persistenz erhalten soll.

package.jdo

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
 "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
 "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
  <package name="name.panitz.sheetMusic">
    <class name="WebLink" />

```

Die spezifizierende `jdo`-Datei[1] muß zur Laufzeit des Programms im Klassenpfad im Paket für die Klassen liegen, für die es Metainformation enthält und es scheint auch notwendig zu sein, daß sie den Namen `package.jdo` trägt.

16.2.3 Klassendatei für Persistenz modifizieren

Jetzt können wir den Persistenzcode in die Klasse `WebLink` einmassieren lassen. Hierzu gibt es in JPOX einen sogenannten *enhancer*. Er wird mit der obigen Spezifikationsdatei `package.jdo` aufgerufen.

[1] Natürlich muß die schließenden Tags `<package></jdo>` noch angefügt werden, wir werden allerding im Laufe des Kapitels noch weitere Elemente hinzufügen in diese Datei hinzufügen und erst dann alle Tags schließen.

```
java  org.jpox.enhancer.JPOXEnhancer classes/name/panitz/sheetMusic/package.jdo
```

Die in der JDO-Datei aufgeführten Klassen, für die Persistenzcode erzeugt werden soll, müssen dazu auf dem Klassenpfad zu finden sein. Nachdem der *enhancer* gelaufen ist, befindet sich jede Menge neuer Code in der Klassendatei von `WebLink`. Wir können den Effekt des *enhancers* mit Hilfe des Programms `javap` einmal betrachten.

```
sep@pc305-3:~/fh/webapp/student> javap -classpath classes/ name/panitz/sheetMusic/WebLink
Compiled from "WebLink.java"
public class name.panitz.sheetMusic.WebLink extends java.lang.Object implements javax.jdo.spi.PersistenceCapabl
    public java.lang.String uri;
    public java.lang.String description;
    protected transient javax.jdo.spi.StateManager jdoStateManager;
    protected transient byte jdoFlags;
    static {};
    public java.lang.String toString();
    static java.util.List singleLink(java.lang.String, java.lang.String);
    static java.util.List singleton(java.lang.Object);
    public void jdoCopyKeyFieldsFromObjectId(javax.jdo.spi.PersistenceCapable$ObjectIdFieldConsumer, java.lang.
    protected void jdoCopyKeyFieldsFromObjectId(java.lang.Object);
    public final void jdoCopyKeyFieldsToObjectId(java.lang.Object);
    public final void jdoCopyKeyFieldsToObjectId(javax.jdo.spi.PersistenceCapable$ObjectIdFieldSupplier, java.l
    public final java.lang.Object jdoGetObjectId();
    public final java.lang.Object jdoGetVersion();
    protected final void jdoPreSerialize();
    public final javax.jdo.PersistenceManager jdoGetPersistenceManager();
    public final java.lang.Object jdoGetTransactionalObjectId();
    public final boolean jdoIsDeleted();
    public final boolean jdoIsDirty();
    public final boolean jdoIsNew();
    public final boolean jdoIsPersistent();
    public final boolean jdoIsTransactional();
    public final boolean jdoIsDetached();
    public final void jdoMakeDirty(java.lang.String);
    public final java.lang.Object jdoNewObjectIdInstance();
    public final java.lang.Object jdoNewObjectIdInstance(java.lang.Object);
    public final void jdoProvideFields(int[]);
    public final void jdoReplaceFields(int[]);
    public final void jdoReplaceFlags();
    public final synchronized void jdoReplaceStateManager(javax.jdo.spi.StateManager);
    public javax.jdo.spi.PersistenceCapable jdoNewInstance(javax.jdo.spi.StateManager);
    public javax.jdo.spi.PersistenceCapable jdoNewInstance(javax.jdo.spi.StateManager, java.lang.Object);
    public void jdoReplaceField(int);
    public void jdoProvideField(int);
    protected final void jdoCopyField(name.panitz.sheetMusic.WebLink, int);
    public void jdoCopyFields(java.lang.Object, int[]);
    protected static int __jdoGetInheritedFieldCount();
    protected static int jdoGetManagedFieldCount();
    public static java.lang.Class ___jdo$loadClass(java.lang.String);
    public name.panitz.sheetMusic.WebLink();
    public static void jdoSetdescription(name.panitz.sheetMusic.WebLink, java.lang.String);
    public static java.lang.String jdoGetdescription(name.panitz.sheetMusic.WebLink);
    public static void jdoSeturi(name.panitz.sheetMusic.WebLink, java.lang.String);
    public static java.lang.String jdoGeturi(name.panitz.sheetMusic.WebLink);
    public name.panitz.sheetMusic.WebLink(java.lang.String, java.lang.String);
}

sep@pc305-3:~/fh/webapp/student>
```

Wie man sieht, wurde uns viel Arbeit abgenommen und jede Menge Code nachträglich in die Klassendatei eingefügt, den wir wirklich nicht hätten selber schreiben wollen.

16.2.4 Datenbanktabellen Erzeugen

Unsere Klassendatei enthält jetzt allen notwendigen Code, um Objekte persistent zu machen. JPOX bildet dabei Objekte auf eine relationale Datenbank ab. Das müsste nicht zwangsläufig so sein. Die JDO-Spezifikation läßt offen, in welcher Form, die Persistenz realisiert wird. Es könnten die Objekte auch lediglich in Dateien gespeichert, oder in einer XML-Datenbank abgelegt werden. Damit die `WebLink`-Objekte in eine relationale Datenbank gespeichert werden können, sind in einer Datenbank zunächst einmal Tabellen anzulegen, in der die Objekte gespeichert werden. Auch dieses läßt sich mit mit JPOX bewerkstelligen. Hierzu enthält das JPOX-Paket eine Klasse `SchemaTool`, die das entsprechende Datenbankschema erzeugt. Zum Laufen dieser Klasse sind eine Vielzahl von Optionen zu setzen:

- `javax.jdo.option.ConnectionDriverName`: der name der Klasse für den Datenbanktreiber, in unserem Fall der Treiber für *mysql*
- `javax.jdo.option.ConnectionURL`: die Url der tatsächlichen Datenbank. In unserem Fall die Datenbank *playground* auf *localhost*.
- `javax.jdo.option.ConnectionUserName`: der Benutzername für den Datenbankbenutzer.
- `javax.jdo.option.ConnectionPassword`: das Passwort dieses Benutzers. (Sicherheitsaspekte lassen wir vorerst außer acht.)

Alles zusammen erhalten wir folgenden Aufruf zur Schemagenerierung:

```
java -Djavax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver\
     -Djavax.jdo.option.ConnectionURL=jdbc:mysql://localhost/playground\
     -Djavax.jdo.option.ConnectionUserName=sep\
     -Djavax.jdo.option.ConnectionPassword=stammel\
  org.jpox.SchemaTool classes/name/panitz/sheetMusic/package.jdo -create
```

Läuft die Generierung erfolgreich, ist in der Datenbank eine Tabelle erzeugt worden. Wir können uns davon z.B. mit dem Programm `mysqlcc` überzeugen. Abbildung 16.1 zeigt die Datenbank nach Erzeugung des Schemas.

Wie man sieht, ist nicht nur eine Tabelle für die Klasse `WebLink` entstanden, sondern auch eine weitere Tabelle, in der JPOX Informationen darüber speichert, welche Javaklasse wie auf Tabellen abgelegt wurde. Man sieht auch, daß die Tabelle für `WebLink` Objekte nur nur Spalten für die zwei Stringfelder der Klasse hat, sondern auch eine Spalte für einen Schlüssel.

16.2.5 Objekte persistent in Datenbank schreiben

Alle Vorbereitungen sind getroffen. Jetzt können wir mit dem JDO-Api arbeiten. Dreh- und Angelpunkt der Arbeit mit persistenten Objekten ist die Schnittstelle `javax.jdo.PersistenceManager`. Objekte dieser Schnittstelle könne Transaktionen öffnen und in diesen Transaktionen Datenbankanfragen starten, oder aber Objekte in eine Datenbank schreiben. Ein Objekt, das die Schnittstelle `PersistenceManager` für eine bestimmte Form der Persistenz, in unserem Fall der gerade erzeugten Datenbank, realisiert, läßt sich über die Schnittstelle `PersistenceManagerFactory` erzeugen. Ein Objekt dieser Factoryklasse kann über eine statische Methode der Klasse `javax.jdo.JDOHelper` erhalten werden. Dabei ist ein

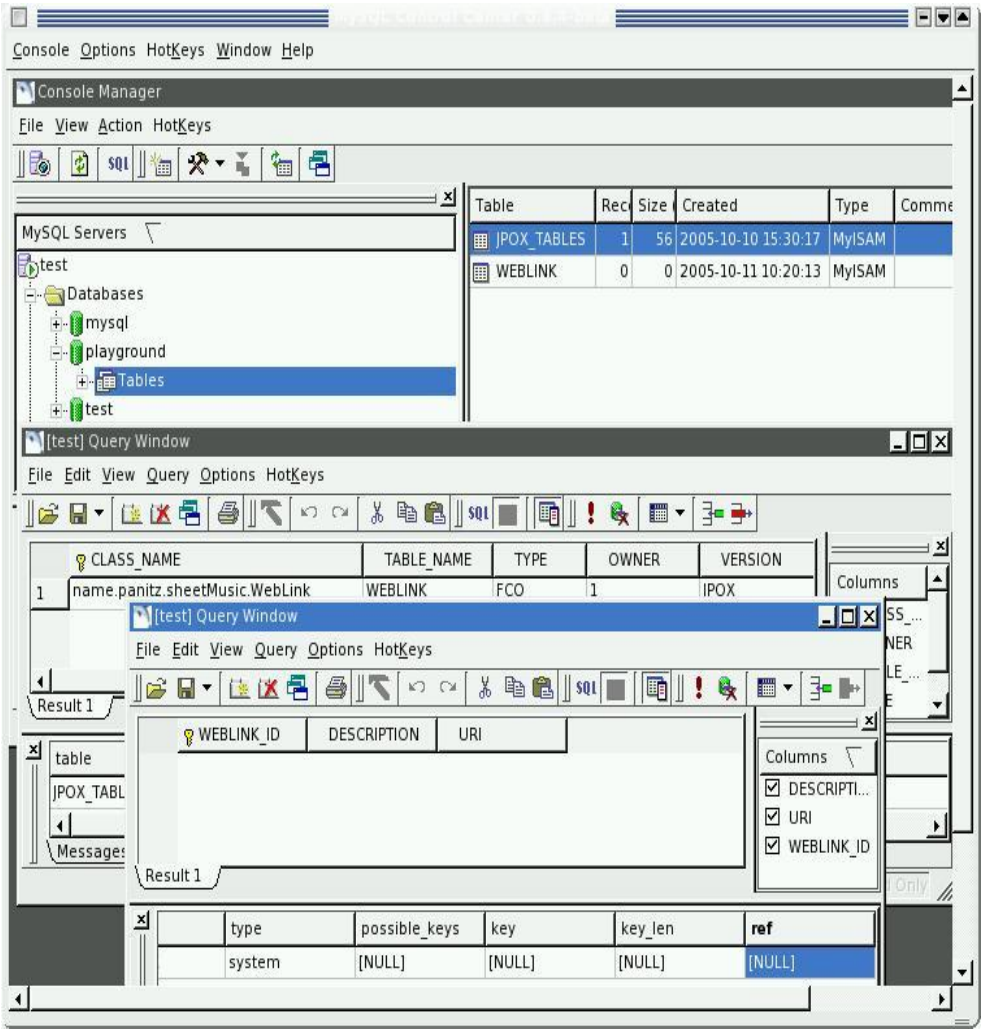


Abbildung 16.1: Datenbank nach Erzeugen des Schemas von JPOX.

einem `Property`-Objekt anzugeben, welche Art von Persistenz gewählt wird, in unserem Fall: die JPOX-Implementierung mit der Datenbank *mysql* für die Datenbank *playground* auf *localhost* mit Benutzer *sep* und Passwort *stammel*. All diese Eigenschaften könnten natürlich global von außen gesetzt sein. Zur Vereinfachung sei hier eine kleine Klasse angegeben, in der mit fest verdrahteten Eigenschaften für unsere Testzwecke diese gesetzt ein `PersistenceManager` erzeugt wird.

PMManager.java

```
1 package name.panitz.sheetMusic;
2 import javax.jdo.*;
3 public class PMManager{
4   static private PersistenceManagerFactory pmf=null;
5   static final private String DB_URL
6                           ="jdbc:mysql://localhost/playground";
7   static final private String USER_NAME="sep";
```

```
8   static final private String PASSWD="stammel";
9
10  public static PersistenceManager getPM(){
11    if (pmf==null){
12       java.util.Properties properties = new java.util.Properties();
13       properties.setProperty
14        ("javax.jdo.PersistenceManagerFactoryClass"
15        ,"org.jpox.PersistenceManagerFactoryImpl");
16       properties.setProperty
17        ("javax.jdo.option.ConnectionDriverName"
18        ,"com.mysql.jdbc.Driver");
19       properties.setProperty
20        ("javax.jdo.option.ConnectionURL",DB_URL);
21       properties.setProperty
22        ("javax.jdo.option.ConnectionUserName",USER_NAME);
23       properties.setProperty
24        ("javax.jdo.option.ConnectionPassword",PASSWD);
25       pmf = JDOHelper.getPersistenceManagerFactory(properties);
26    }
27    return pmf.getPersistenceManager();
28  }
29 }
```

Mit einem `PersistenceManager`-Objekt haben wir nun ein mächtiges Werkzeug, um Objekte in die Datenbank zu schreiben. Dieses geht dabei erstaunlich einfach. Der `PersistenceManager` kann ein Transaktionsobjekt erzeugen. Eine Transaktion läßt sich starten und die in ihr gemachten Aktionen endgültig begehen. Der `PersistenceManager` hat eine Methode `makePersistent`, der Objekte zum Schreiben in die Datenbank übergeben werden können. Somit lassen sich sehr einfach Objekte persistent machen.

FirstPersistentObject.java

```
1  package name.panitz.sheetMusic;
2  import javax.jdo.*;
3  public class FirstPersistentObject{
4    public static void main(String [] xs){
5      final WebLink[] ws
6       = {new WebLink
7          ("Giuliani op 15"
8          ,"http://www.kb.dk/elib/noder/rischel/RiBS0272.pdf")
9        ,new WebLink
10         ("von Call op 107"
11         ,"http://www.panitz.name/music/call107/Call107Score.pdf")
12       ,new WebLink
13         ("Gragnani Ouverture"
14         ,"http://www.eythorsson.com/music/1106.pdf")};
15     final PersistenceManager pm = PMManager.getPM();
16     final Transaction tx=pm.currentTransaction();
17     try{
18       tx.begin();
19       for (WebLink w:ws) pm.makePersistent(w);
```

```
20        tx.commit();
21      }finally{
22        if (tx.isActive()){tx.rollback();}
23        pm.close();
24      }
25    }
26  }
```

Wichtig ist darauf Rücksicht zu nehmen, daß sobald wir mit externen Komponenten, in diesen Fall der Datenbank arbeiten, Ausnahmen auftreten können. Auf jeden Fall sollte man dann die Transaktion in einen konsistenen Zustand hinterlassen. Am Ende einer Datenbankoperation ist der `PersistenceManager` zu schließen. Hierzu bietet sich an, sich des `finally` eines `try`-Blockes zu bedienen.

Wir können dieses Progamm jetzt einfach starten. Dabei müssen natürlich sämtliche relevanten Jar-Dateien im Klassenpfad stehen.

```
sep@pc305-3:~/fh/webapp/student> java -cp classes/:$CLASSPATH name/panitz/sheetMusic/FirstPersistentObject
sep@pc305-3:~/fh/webapp/student>
```

Mit einem Blick in die Datenbank können wir uns tatsächlich davon überzeugen, daß die entsprechenden Objekte in der Tabelle für `WebLink` abgelegt wurden. Abbildung 16.2 zeigt die Datenbank nach ablegen der ersten persistenetn Objekte.

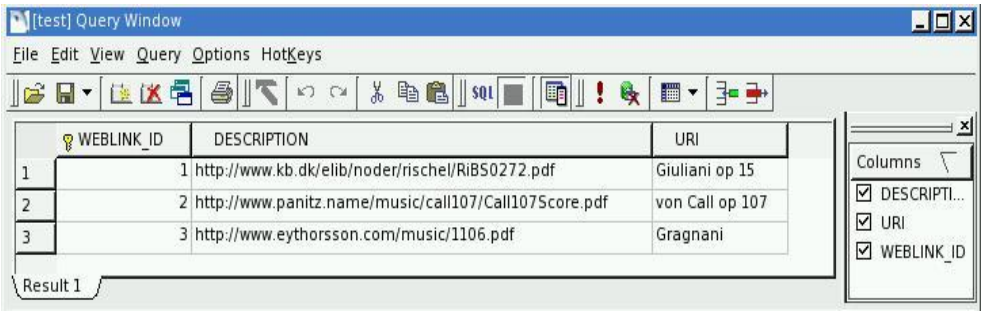


Abbildung 16.2: Datenbanktabelle mit drei Javaobjekten.

16.2.6 Objekte aus Datenbank instanziieren

Nachdem wir nun glücklich Objekte in die Datenbank geschrieben haben, wollen wir auch wieder Objekte aus der Datenbank zurück in ein laufendes Programm bekommen. Hierzu lassen sich von einem `PersistenceManager` Anfrageobjekte der Schnittstelle `Query` erzeugen. Die einfachste Anfrage an eine Datenbank kann dabei alle Objekte die von einer bestimmten Klasse gespeichert wurden erfragen.

QueryWebLinks.java

```
1  package name.panitz.sheetMusic;
2  import javax.jdo.*;
3  import java.util.Collection;
```

```
4  public class QueryWebLinks{
5    public static void main(String [] xs){
6      final PersistenceManager pm = PMManager.getPM();
7      final Transaction tx=pm.currentTransaction();
8      final Query q=pm.newQuery(WebLink.class);
9      try{
10       tx.begin();
11       for (WebLink w:(Collection<WebLink>)q.execute())
12         System.out.println(w);
13       tx.commit();
14     }finally{
15       if (tx.isActive()){tx.rollback();}
16       pm.close();
17     }
18   }
19 }
```

Auch dieses war einfacher zu Programmieren, als gedacht. Das Programm gibt tatsächlich die drei `WebLink`-Objekte aus der Datenbank aus.

```
sep@pc305-3:~/fh/webapp/student> java -cp classes/:$CLASSPATH name/panitz/sheetMusic/QueryWebLinks
<a href="http://www.panitz.name/music/call107/Call107Score.pdf">von Call op 107</a>
<a href="http://www.kb.dk/elib/noder/rischel/RiBS0272.pdf">Giuliani op 15</a>
<a href="http://www.eythorsson.com/music/1106.pdf">Gragnani Ouverture</a>
sep@pc305-3:~/fh/webapp/student>
```

Eine Schwäche des JDO APIs ist zu erkennen. Das Ergebnis der Methode `execute` ist lediglich vom Typ `Object`. Allein der Dokumentation entnehmen wir, daß es sich hier um ein `Collection`-Objekt, dessen Elemente vom Typ der Klasse sind, für die die Anfrage erzeugt wurde. Mit generischen Typen ließe sich heutzutage ein API für JDO definieren, in dem das Typsystem ausdrück, vom welchen Typ das Ergebnis einer Anfrage ist. Hierzu wäre die Schnittstelle `Query` generisch zu definieren über den Ergebnistyp.

16.2.7 Modifizieren persistenter Objekte

Wir haben bisher gesehen, wie sich Objekte in die Datenbank speichern lassen und ebenso, wie diese wieder aus der Datenbank geladen werden können. Wie sieht es aus, mit der Möglichkeit, Daten zu manipulieren. Versuchen wir dieses einmal naiv:

ModifyWebLinksDirect.java

```
1  package name.panitz.sheetMusic;
2  import javax.jdo.*;
3  import java.util.Collection;
4  public class ModifyWebLinksDirect{
5    public static void main(String [] xs){
6      final PersistenceManager pm = PMManager.getPM();
7      final Transaction tx=pm.currentTransaction();
8      final Query q=pm.newQuery(WebLink.class);
9      try{
10       tx.begin();
```

```
      for (WebLink w:(Collection<WebLink>)q.execute())
        w.description=w.description.toUpperCase();
      tx.commit();
    }finally{
      if (tx.isActive()){tx.rollback();}
      pm.close();
    }
  }
}
```

Lassen wir dieses Programm jedoch laufen und betrachten anschließend die gespeicherten Objekte in der Datenbank, so werden wir enttäuscht feststellen, daß in der Datenbank die Objekte nicht verändert wurden:

```
sep@pc305-3:~/fh/webapp> java -cp classes/:$CLASSPATH name/panitz/sheetMusic/FirstPersistentObject
sep@pc305-3:~/fh/webapp> java -cp classes/:$CLASSPATH name/panitz/sheetMusic/QueryWebLinks
<a href="http://www.panitz.name/music/call107/Call107Score.pdf">von Call op 107</a>
<a href="http://www.kb.dk/elib/noder/rischel/RiBS0272.pdf">Giuliani op 15</a>
<a href="http://www.eythorsson.com/music/1106.pdf">Gragnani Ouverture</a>
sep@pc305-3:~/fh/webapp/student>
```

Der Grund liegt darin, daß JPOX keine Gelegenheit hatte, den Code für die einfache Zuweisung an das Feld `description` derart zu verändern, daß die Zuweisung einen direkten Effekt auf die Datenbank hat. Geschieht diese Zuweisung allerdings direkt in der Klasse `WebLink`, so hat der Enhancer die Möglichkeit diesen Code entsprechend einzufügen. Daher sollten wir zumindest eine `set`-Methode für Felder, die auf der Datenbank modifiziert werden sollen vorsehen. Dann kann der Enhancer diese `set`-Methode derart verändern, daß das Objekt persistent verändert wird:

WebLink.java

```
  public void setDescription(String des){description=des;}
}
```

Versuchen wir also jetzt die Modifikation mit Hilfe der neuen Methode:

ModifyWebLinks.java

```
package name.panitz.sheetMusic;
import javax.jdo.*;
import java.util.Collection;
public class ModifyWebLinks{
  public static void main(String [] xs){
    final PersistenceManager pm = PMManager.getPM();
    final Transaction tx=pm.currentTransaction();
    final Query q=pm.newQuery(WebLink.class);
    try{
      tx.begin();
      for (WebLink w:(Collection<WebLink>)q.execute())
        w.setDescription(w.description.toUpperCase());
      tx.commit();
    }finally{
      if (tx.isActive()){tx.rollback();}
```

```
      pm.close();
    }
  }
}
```

Und siehe da, jetzt ändern sich auch die Objekte in der Datenbank:

```
sep@pc305-3:~/fh/webapp/student> java -cp classes/:$CLASSPATH name/panitz/sheetMusic/ModifyWebLinks
sep@pc305-3:~/fh/webapp/student> java -cp classes/:$CLASSPATH name/panitz/sheetMusic/QueryWebLinks
<a href="http://www.panitz.name/music/call107/Call107Score.pdf">VON CALL OP 107</a>
<a href="http://www.kb.dk/elib/noder/rischel/RiBS0272.pdf">GIULIANI OP 15</a>
<a href="http://www.eythorsson.com/music/1106.pdf">GRAGNANI OUVERTURE</a>
sep@pc305-3:~/fh/webapp/student>
```

16.2.8 Persistent verschiener Typen

Mapping für String-Objekte

Composer.java

```
package name.panitz.sheetMusic;
import java.util.*;
import javax.jdo.*;

public class Composer {
  private String firstName;
  public void setFirstName(String fN){firstName=fN;}
  public String getFirstName(){return firstName;}

  private String lastName;
  public void setLastName(String lastName){this.lastName=lastName;}
  public String getLastName(){return lastName;}

  public Date born;
  public Date death;

  private String biography;
  public void setBiography(String bio){biography=bio;}
  public String getBiography(){return biography;}

  Composer(String fN,String lN,Date b,Date d,String bio){
    firstName=fN;lastName=lN;born=b;death=d;biography=bio;};

  Composer(String fN,String lN,String bio){
    this(fN,lN,null,null,bio);};

  public String toString(){return lastName+", "+firstName;}

  static public Composer
        getForName(PersistenceManager pm,String fName,String lName){
    final Query q
```

```
      = pm.newQuery(Composer.class,"firstName == fName && lastName==lName");
    q.declareParameters("String fName,String lName");
    q.setUnique(true);
    return (Composer)q.execute(fName,lName);
  }
}
```

package..jdo

```
  <class name="Composer" />
  </package>
</jdo>
```

FirstPersistentComposer.java

```
package name.panitz.sheetMusic;
import javax.jdo.*;
import java.util.*;
public class FirstPersistentComposer{
 public static void main(String [] xs){
  final String bio
  ="Mauro Giuliani (July 27, 1781  May 8, 1828) was an Italian guitarist "
  +"and composer. Aside from Fernando Sor, Giuliani is reckoned as the "
  +"other leading guitar virtuoso of the 19th century. "
  +"Mauro Giuliani, one of the greatest exponents of the Nineteenth "
  +"Century guitar, a first rank virtuoso and a great composer for guitar, "
  +"was born in Bisceglie, on July 27, 1781.";

  final Composer giuliani= new Composer("Giuliani","Mauro",bio);
  final PersistenceManager pm = PMManager.getPM();
  final Transaction tx=pm.currentTransaction();
  try{
    tx.begin();
    pm.makePersistent(giuliani);
    tx.commit();
  }finally{
    if (tx.isActive()){tx.rollback();}
    pm.close();
  }
  final PersistenceManager pm2 = PMManager.getPM();
  final Query q=pm2.newQuery(Composer.class);
  final Transaction tx2=pm2.currentTransaction();
  try{
    tx2.begin();
    System.out.println(
     ((Collection<Composer>)q.execute()).iterator().next().getBiography());
    tx2.commit();
  }finally{
    if (tx2.isActive()){tx2.rollback();}
    pm2.close();
  }
 }
}
```

```
sep@pc305-3:~/fh/webapp/student> java -cp classes/:$CLASSPATH name/panitz/sheetMusic/FirstPersistentComposer
Mauro Giuliani (July 27, 1781 ? May 8, 1828) was an Italian guitarist and composer. Aside from\
 Fernando Sor, Giuliani is reckoned as the other leading guitar virtuoso of the 19th century. \
Mauro Giuliani, one of the greatest exponents of the Nineteenth Cen
sep@pc305-3:~/fh/webapp/student>
```

package.jdo

```
    <class name="Composer" >
      <field name="biography" persistence-modifier="persistent">
        <column name='name' jdbc-type='BLOB'/>
      </field>
    </class>
```

Aufzählungstypen

Instrument.java

```
package name.panitz.sheetMusic;
import javax.jdo.*;
import java.util.*;
import java.lang.reflect.*;

public final class Instrument{
  public static final Instrument Guitar   = new Instrument("Guitar");
  public static final Instrument Violin   = new Instrument("Violin");
  public static final Instrument Viola    = new Instrument("Viola");
  public static final Instrument Cello    = new Instrument("Cello");
  public static final Instrument Piano    = new Instrument("Piano");
  public static final Instrument Cembalo  = new Instrument("Cembalo");
  public static final Instrument Recorder = new Instrument("Recorder");
  public static final Instrument Horn     = new Instrument("Horn");
  public static final Instrument Flute    = new Instrument("Flute");
  public static final Instrument Oboe     = new Instrument("Oboe");
  public static final Instrument Bassoon  = new Instrument("Bassoon");
  public static final Instrument Clarinet = new Instrument("Clarinet");
  public static final Instrument Trumpet  = new Instrument("Trumpet");
  public static final Instrument Harp     = new Instrument("Harp");
  public static final Instrument Voice    = new Instrument("Voice");

  public String toString(){return name;}

  private String name;
  private Instrument(String n){name=n;}

  public static Instrument[] instruments;
  static {staticInitInstruments();}
  static private void staticInitInstruments(){
    final Field[] fields = Instrument.class.getFields();
    final List<Instrument> ins=new ArrayList<Instrument>();
    for (final Field f:fields){
      if (f.getType().equals(Instrument.class))
```

```
35          try{
36            ins.add((Instrument)f.get(null));
37          }catch(IllegalAccessException _){}
38        }
39        instruments=ins.toArray(new Instrument[0]);
40      }
41
42      public  boolean equals(Object other){
43        return  null!=other
44             && other instanceof Instrument
45             && ((Instrument)other).name.equals(name);
46      }
47
48      static public Instrument getForName(PersistenceManager pm,String name){
49        final Query q = pm.newQuery(Instrument.class,"name == n");
50        q.declareParameters("String n");
51        q.setUnique(true);
52        return  (Instrument)q.execute(name);
53      }
54
55      static public void populateDB(){
56        final PersistenceManager pm = PMManager.getPM();
57        final Transaction tx=pm.currentTransaction();
58        try{
59          tx.begin();
60          for (Instrument instr:instruments)
61            if (getForName(pm,instr.name)==null) pm.makePersistent(instr);
62          tx.commit();
63        }catch (Exception e){
64        }finally{
65          if (tx.isActive()){tx.rollback();}
66          pm.close();
67        }
68      }
69
70      public static void main(String [] args){populateDB();}
71
72      static public Collection<Object>
73                    getPersistentObjects(PersistenceManager pm){
74        return getPersistentObjects(pm,Instrument.class);}
75
76      static public Collection<Object>
77                    getPersistentObjects(PersistenceManager pm,Class klasse){
78        final Transaction tx=pm.currentTransaction();
79        try{
80          tx.begin();
81          final Query q = pm.newQuery(klasse,"true");
82          return (Collection<Object>)q.execute();
83        }catch (Exception e){
84        }finally{
```

```
      if (tx.isActive()){tx.rollback();}
      pm.close();
    }
    return null;
  }

  public static String htmlOptionFields(Collection<Object> os,Object selected)
    final StringBuffer optionBuf = new StringBuffer();
    optionBuf.append
      ("<option "+(selected==null¨ selected=\"true\"":"")+">NONE</option>\n");
    for (Object o:os){
      optionBuf.append
        ("<option "+(o.equals(selected)¨ selected=\"true\"":"")+">"
                  +o+"</option>\n");
    }
    return optionBuf.toString();
  }
}
```

Genre.java

```
package name.panitz.sheetMusic;
import javax.jdo.*;
import java.util.*;
import java.lang.reflect.*;

public final class Genre {
  public static final Genre Sonata    = new Genre("Sonata");
  public static final Genre Variations= new Genre("Variations");
  public static final Genre Cappriccio= new Genre("Cappriccio");
  public static final Genre Etude     = new Genre("Etude");
  public static final Genre Rondo     = new Genre("Rondo");
  public static final Genre Pollaca   = new Genre("Pollaca");
  public static final Genre Fantasie  = new Genre("Fantasie");
  public static final Genre Valse     = new Genre("Valse");

  private String name;
  private Genre(String n){name=n;}
  public String toString(){return name;}

  public static Genre[] genres;
  static {staticInitGenres();}
  static private void staticInitGenres(){
    final Field[] fields = Genre.class.getFields();
    final List<Genre> ins=new ArrayList<Genre>();
    for (final Field f:fields){
      if (f.getType().equals(Genre.class))
        try{
          ins.add((Genre)f.get(null));
        }catch(IllegalAccessException _){}
```

```
    }
    genres=ins.toArray(new Genre[0]);
  }

  public  boolean equals(Object other){
    return  null!=other
         && other instanceof Genre
         && ((Genre)other).name.equals(name);
  }

  static public Genre getForName(PersistenceManager pm,String name){
    final Query q = pm.newQuery (Genre.class,"name == n");
    q.declareParameters("String n");
    q.setUnique(true);
    return  (Genre)q.execute(name);
  }

  static public void populateDB(){
    final PersistenceManager pm = PMManager.getPM();
    final Transaction tx=pm.currentTransaction();
    try{
      tx.begin();
      for (Genre genre:genres)
        if (getForName(pm,genre.name)==null) pm.makePersistent(genre);
      tx.commit();
    }catch (Exception e){
    }finally{
      if (tx.isActive()){tx.rollback();}
      pm.close();
    }
  }

  static public Collection<Object>
                 getPersistentObjects(PersistenceManager pm){
    return Instrument.getPersistentObjects(pm,Genre.class);}

  public static void main(String [] args){populateDB();}
}
```

package.jdo

```
    <class name="Instrument" />
    <class name="Genre" />
```

Mapping für Sammlungsobjekte

Opus.java

```
package name.panitz.sheetMusic;
import java.util.*;
import javax.jdo.*;
```

```
public class Opus {
  private String title;
  public String getTitle(){return title;}
  public void setTitle(String t){title=t;}

  private String opusNo;
  public String getOpusNo(){return opusNo;}
  public void setOpusNo(String oN){opusNo=oN;}

  private String comment;
  public String getComment(){return comment;}
  public void setComment(String c){comment=c;}

  private List<Instrument> instruments;
  public List<Instrument> getInstruments(){return instruments;}
  public void setInstruments(List<Instrument> is){instruments=is;}

  private List<WebLink> links;
  public List<WebLink> getLinks(){return links;}
  public void setLinks(List<WebLink> wls){links=wls;}

  Genre genre;
  Composer composer;

  private Opus(){}

  Opus(String t,Genre g,Composer c,String op,String com
    ,List<Instrument> ins,List<WebLink> ls){
    title=t;genre=g;composer=c;opusNo=op;comment=com;instruments=ins;links=ls;

  public String toString(){
    return composer+": "+title+" "+instruments+" "+links;
  }

  static public Opus getForComposerAndTitle
         (PersistenceManager pm,Composer c,String t){
    final Query q = pm.newQuery (Opus.class,"composer == c && title==t");
    q.declareParameters("Composer c,String t");
    q.setUnique(true);
    return (Opus)q.execute(c,t);
  }
  static public Opus getForComposerAndTitleAndOpus
         (PersistenceManager pm,Composer c,String t,String o){
    final Query q = pm.newQuery (Opus.class
                                ,"composer == c && title==t && opusNo ==o");
    q.declareParameters("Composer c,String t,String o");
    q.setUnique(true);
    return (Opus)q.execute(c,t,o);
  }
```

```
}
```

package.jdo

```
    <class name="Opus">
      <field name="instruments" persistence-modifier="persistent">
        <collection element-type="name.panitz.sheetMusic.Instrument"/>
        <join/>
      </field>
      <field name="links" persistence-modifier="persistent">
        <collection element-type="name.panitz.sheetMusic.WebLink"/>
        <join/>
      </field>
    </class>
  </package>
</jdo>
```

16.2.9 Komplexere Anfragen

16.3 Servlets

In diesen Kapitel betrachten wir die Möglichkeit mit Javaprogrammen die Funktionalität eines Webservers zu erweitern. Damit lassen sich dynamische Webseiten erzeugen. Eine Webadresse wird vom Server umgeleitet auf ein Javaprogramm, das die entsprechende Antwortseite erzeugt.

Dieses Kapitel versucht ein einfaches Kochbuch zum Schreiben von Webanwendungen zu sein. Wir benutzen als Datenquelle für die Webanwendung die per JDO erzeugte Datenbank aus dem letzten Kapitel.

16.3.1 Servlet Container

Standardmäßig unterstützt ein Webserver nicht die Fähigkeit Javaprogramme für bestimmte URLs aufzurufen. Hierzu ist der Webserver zunächst um eine Komponente zu erweitern, die ihn dazu befähigt. Eine solche Komponente wird *servlet container* bezeichnet. Einer der gebräuchlichsten *servlet container* ist der *tom cat* (`http://jakarta.apache.org/tomcat/`) . Dieser kann genutzt werden um z.B. den *apache* Webserver zu erweitern, so daß auf ihm Servlets laufen können. Der *tom cat* selbst ist jedoch auch bereits ein eigener Webserver und kann, so wie wir es in diesem Kapitel tun, auch als eigenständiger Webserver betrieben werden.

16.3.2 Struktur einer Webapplikation

Der *tom cat* kommt mit einem *ant* Skript zum Bauen einer Webapplikation. Neben den üblichen Zielen wie `compile` enthält dieses Skript die Ziele `install`, `remove` und `reload` um eine Webapplikation auf einen laufenden *tom cat* zu instalieren.

Dieses *ant* Skript erwartet eine bestimmte Ordnerstruktur, der folgenden Form:

```
applikationsname
  |--- src
  |--- docs
  |--- web
        |--- WEB-INF
        |--- images
  |--- build
  |--- dist
```

Im Order `src` sind alle Java-sourcen zu plazieren. Im Order `web` stehen die einzelnen Webseiten für die Webapplikation. Im Unterordner `WEB-INF` ist die Hauptkonfigurationsdatei `web.xml`. In dieser sind die einzelnen Servlets mit ihren zugehörigen Javaklassen und URLs verzeichnet.

16.3.3 Servletkonfiguration

Die wichtigste Konfigurationsdatei für eine Webapplikation ist die Datei `web.xml`. In ihr wird die Webapplikation mit ihren Servlets beschrieben. Für jedes Servlet wird ein Name vergeben. Dieser Name wird gebunden an eine Servletklasse, die die entsprechende Funktionalität zur Verfügung stellt und an die URL über den auf das Servlet zugegriffen wird. Das Format der Datei `web.xml` ist in einer DTD definiert. In unseren Beispiel sehen wir vier Servlets vor:

web.xml

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>Sheet Music</display-name>
  <description>Guitar music resources on the web.</description>
```

Das erste Servlet soll alle in der Datenbank gespeicherten Resourcen anzeigen. Es wird durch die Servletklasse `ListAll` realisiert werden und über den Anhang `/uri` an die URI des Webservers erreicht werden:

web.xml

```
  <servlet>
    <servlet-name>ListServlet</servlet-name>
    <servlet-class>name.panitz.sheetMusic.ListAll</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ListServlet</servlet-name>
    <url-pattern>/list</url-pattern>
  </servlet-mapping>
```

Das zweite Servlet soll es ermöglichen, einen neuen Komponisten in die Datenbank hinzuzufügen:

web.xml

```
  <servlet>
   <servlet-name>AddComposerServlet</servlet-name>
   <servlet-class>name.panitz.sheetMusic.AddComposer</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AddComposerServlet</servlet-name>
    <url-pattern>/addComposer</url-pattern>
  </servlet-mapping>
```

Das dritte Servlet soll es ermöglichen einen neuen Eintrag von einen Link auf Noten im Netz in die Datenbank einzufügen. Dabei soll ein Komponist auf einem Formular aus der Datenbank ausgewählt werden können, für den dann ein neuer Eintrag auf ein Werk von ihm im Web eingetragen werden kann. Hierfür werden wir zwei getrennte Servlets benötigen. Zunächst das Servlet, in dem aus einer Liste aller Komponisten aus der Datenbank einer ausgewählt werden kann:

web.xml

```
  <servlet>
    <servlet-name>AddOpusTemplate</servlet-name>
    <servlet-class
      >name.panitz.sheetMusic.AddOpusTemplate</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AddOpusTemplate</servlet-name>
    <url-pattern>/addOpusTemplate</url-pattern>
  </servlet-mapping>
```

Und schließlich das Servlet, in dem ein neues Werk zu dem zuvor ausgewähltne Komponisten zugefügt werden kann.

web.xml

```
  <servlet>
    <servlet-name>AddOpus</servlet-name>
    <servlet-class>name.panitz.sheetMusic.AddOpus</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AddOpus</servlet-name>
    <url-pattern>/addOpus</url-pattern>
  </servlet-mapping>
</web-app>

```

Für unsere Webapplikation sehen wir noch eine minimale Einstiegswebseite vor, die wir vorerst nicht als Servlet zu realisieren brauchen.

index.html

```
<html>
<head>
<title>Sheet Music for Classical Guitar</title>
```

```
4  </head>
5  <body bgcolor="white">
6
7   <h1>Sheet Music for Classical Guitar</h1>
8
9   <ul>
10   <li><a href="list">List all resources</a>.</li>
11   <li><a href="addComposer.html">add a new composer entry</a>.</li>
12   <li><a href="addOpusTemplate">add a new opus for an existing
13        composer</a>.</li>
14  </ul>
15 </body>
16 </html>
```

16.3.4 Servletklassen

Bisher haben wir die Anwendungslogik und die Persistenzschicht unserer Anwendung implementiert. Wir haben beschrieben, welche Dienste unsere Webanwendung über Servlets anbieten soll. Es sind nun die eigentlichen Servletklassen zu schreiben. Leider ist das Servlet API nicht Bestandteil der Java Bibliotheken aus dem Standard SDK. Die Jar-Datei und Dokumentation für das Servlet-API ist separat herunterzuladen. In der Distribution von *tomcat* ist das Servlet-API enthalten.

Ein Servlet ist eine Klasse, die die Schnittstelle `javax.servlet.Servlet` implementiert. Hierin finden sich fünf Methoden, zwei die Informationen über dass Servlet angeben, eine die bei der Initialisierung des Servlets vom Servletcontainer ausgeführt wird, eine die ausgeführt wird, wenn der Servletcontainer den Dienst abstellt und schließlich eine, die den eigentlichen Dienst beschreibt:

```
1 void destroy();
2 ServletConfig getServletConfig();
3 java.lang.String getServletInfo();
4 void init(ServletConfig config);
5 void service(ServletRequest req, ServletResponse res);
```

Zum Schreiben von Servlets für HTTP wird nicht direkt diese Schnittstelle implementiert sondern eine abstrakte Klasse erweitert, die Klasse `javax.servlet.http.HttpServlet`, die ihrerseits die Schnittstelle Servlet implementiert. `HttpServlet` ist zwar eine abstrakte Klasse, es können also nicht direkt Objekte dieser Klasse instanziiert werden, aber die Klasse hat keine abstrakten Methoden.

Generierung von Seiten

Die einfachste Form eines Servlets schickt eine dynamisch erzeugte HTML-Seite an den Client. Hierzu ist die Methode `doGet` der Klasse `HttpServlet` zu überschreiben. Die Methode hat zwei Argumente:

- ein `HttpServletRequest`, in dem Informationen über den Aufruf des Servlets zu finden sind.

- eine `HttpServletResponse`, die zur Erzeugung der Antwortseite benutzt wird.

Auf der `HttpServletResponse` läßt sich der Typ des Inhalts setzen, in den meisten Fällen wahrscheinlich `text/html`. Weiterhin enthält es einen `PrintWriter`, auf den der Inhalt der zu generierenden Seite auszugeben ist.

Unter Berücksichtigung der Ausnahme, die auftreten können, läßt sich nun leicht ein Servlet implementieren. Die folgende Klasse realisiert das Servlet zum Auflisten der gespeicherten Adressen.

Zunächst importieren wir alle notwendigen Pakete und definieren eine Unterklasse von `HttpServlet`:

ListAll.java

```
6  package name.panitz.sheetMusic;
7
8  import java.io.*;
9
10 import javax.servlet.ServletException;
11 import javax.servlet.http.*;
12
13 import java.util.*;
14 import javax.jdo.*;
15
16 public final class ListAll extends HttpServlet {
```

In dieser sei dann die Methode `doGet` überschrieben:

ListAll.java

```
1  package name.panitz.sheetMusic;
2    public void doGet(HttpServletRequest request
3                      ,HttpServletResponse response)
4          throws IOException, ServletException {
```

Dem Antwortobjekt wird darin zunächst gesagt, daß wir eine Html-Antwort erzeugen wollen:

ListAll.java

```
5      response.setContentType("text/html");
```

Dann läßt sich von dem Antwortobjekt ein `PrintWriter`-Objekt geben, auf dem als String nach und nach die Antwort geschrieben werden kann:

ListAll.java

```
6      PrintWriter writer = response.getWriter();
7
8      writer.println("<html>");
9      writer.println("<head>");
10     writer.println("<title>Links</title>");
11     writer.println("</head>");
12     writer.println("<body bgcolor=\"white\">");
```

Jetzt holen wir uns schließlich den Inhalt der zu schreibenen Seite aus der Datenbank und fügen die entsprechende Information nach und nach an die Antwort an:

ListAll.java

```
13      try{
14       final PersistenceManager pm = PMManager.getPM();
15       final Transaction tx=pm.currentTransaction();
16       try{
17        tx.begin();
18        final Query qComposer = pm.newQuery(Composer.class,"true");
19        qComposer.setOrdering("lastName ASC");
20        final Collection<Composer> comps
21          = (Collection<Composer>)qComposer.execute();
22        writer.println("<table>");
23        for (Composer comp:comps){
24          writer.println(ToHtml.toHTMLString(comp,pm));
25          writer.println("<p />\n\n");
26        }
27        writer.println("</table>");
28        tx.commit();
```

Wir tragen auf einfachste Weise dem Rechnung, daß hierbei einiges schief gehen kann und wir in Form von Ausnahmen davon erfahren.

ListAll.java

```
29       }catch (Exception e){
30         writer.println(e);
31         e.printStackTrace(writer);
32       }finally{
33         if (tx.isActive()){tx.rollback();}
34         pm.close();
35       }
36     }catch(Exception e){writer.println(e);e.printStackTrace(writer);}
37
38      writer.println("</body>");
39      writer.println("</html>");
40    }
41  }
```

Damit ist das erste Servlet realisiert.

Lesen von Parametern

Im letzten Abschnitt haben wir ein Servlet geschrieben, das eine Html-Seite generiert. Dabei werden keine vom Client übermittelten Informationen genutzt. Jetzt wollen wir ein Servlet schreiben, das Benutzereingabe in einem `form`-Element auf der Webseite erhält und benutzt.

Einen neuen Komponisten Hinzufügen Das erste Servlet mit dieser Funktionalität soll es ermöglichen einen neuen Komponisten in die Datenbank einzutragen.

Hierzu brauchen wir zunächst ein einfaches Formular in Form einer Html-Seite, in der dieDaten eingegeben werden können:

addComposer.html

```
<html><body bgcolor="white">
<h1>Add a new composer entry</h1>
<form action="addComposer" method=POST>

last name:
<input type="text" size="20" name="lastname">
<br>
first name:
<input type="text" size="20" name="firstname">
<br>
biography:
<textarea type="text" rows="20" cols="60" name="biography">
<br>

<input type="submit">
</form>
</body>
</html>
```

Um jetzt auf die durch ein `POST` vom Client an den Webserver übermittelte Daten zugreifen zu können, überschreiben wir die Methode `doPost` der Klasse `HttpServlet`. Hier können wir auf dem `HttpServletRequest`-Objekt mit der Methode `getParameter` auf den Wert eines mitgeschickten Eingabefeldes zugreifen, und dieses für die Antwort benutzen.

AddComposer.java

```
package name.panitz.sheetMusic;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.Writer;

import javax.servlet.ServletException;
import javax.servlet.http.*;

import javax.jdo.*;
import java.util.*;

public final class AddComposer extends HttpServlet {
```

Die Methode `doPost` hat zunächst einmal die gleichen Möglichkeiten, wie die Methoden `doGet`, um eine Antwort zu generieren:

AddComposer.java

```
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
```

```
16          throws IOException, ServletException {
17      response.setContentType("text/html");
18      final PrintWriter writer = response.getWriter();
19
20      writer.println("<html>");
21      writer.println("<head>");
22      writer.println("<title>Added Composer</title>");
23      writer.println("</head>");
24      writer.println("<body bgcolor=white>");
```

Zusätzlich lassen sich die Werte der eingegebenen Parameter aus dem `HttpServletRequest`-Objekt erfragen. Wir nutzen sie, um ein neues `Composer`-Ojekt zu erzeugen:

AddComposer.java

```
25      try{
26        final String lastname = request.getParameter("lastname");
27        final String firstname = request.getParameter("firstname");
28        final String biography = request.getParameter("biography");
29        final Composer comp
30         =new Composer(firstname,lastname,null,null,biography);
31
```

Dieses können wir schließlich persitent in die Datenbank schreiben.

AddComposer.java

```
32        final PersistenceManager pm = PMManager.getPM();
33        final Transaction tx=pm.currentTransaction();
34        try{
35          tx.begin();
36          pm.makePersistent(comp);
37          tx.commit();
38        }finally{
39          if (tx.isActive()){tx.rollback();}
40          pm.close();
41        }
42        writer.println("added: "+lastname+", "+firstname);
43
```

Abschließend kümmern wir uns noch um eventuelle Ausnahmen beim Lesen der Parameter und Schreiben Antwort, um schließlich das generierte Html-Dokument zum Ende zu führen:

AddComposer.java

```
44      }catch(Exception e){writer.println(e);}
45      writer.println("</body>");
46      writer.println("</html>");
47    }
```

Auch die Methode `doGet` sei noch überschrieben, für den Fall, daß keine Daten über ein `POST` übermittelt wurden

AddComposer.java

```
48    public void
49     doGet(HttpServletRequest request,HttpServletResponse response)
50                            throws IOException, ServletException{
51      response.setContentType("text/html");
52      PrintWriter out = response.getWriter();
53      out.println("GET Request. No Form Data Posted");
54    }
55  }
```

16.4 Java Server Pages

In den obigen Servletbeispielen konnte man schon sehen, daß ein Servlet zwei sehr unterschiedliche Aufgabe wahrnimmt:

- die angefragten Daten berechnen.
- die angefragte Html-Seite mit den berechneten Daten generieren.

Dieses sind zwei sehr unterschiedliche Aufgaben, die wahrscheinlich in der Regel auch personell getrennt entwickelt werden. Das Schreiben der Html-Seite für die Ausgabe kommt bei Servlets etwas zu kurz. Es wird nur Java-Code geschrieben. Die Ausgabe muß mühsam mit `println`-Aufrufen generiert werden.

Diesem Problem begegnen *Java Server Pages(JSP)*. Diese sehen im Prinzip wie Html-Seiten aus, in denen an ausgesuchten Stellen kleine Java Code Schnipsel stehen. Der Java Code berechnet an diesen Stellen, wie der Wert für die Webseite dort berechnet wird.

Die Syntax für JSP ist recht umfangreich und wir werden hier nur ein kleines Beispiel betrachten:

> **Beispiel:**
> Die Seite zum Auflisten der Komponisten in unserer Webapplikation läßt sich wesentlich kürzer als JSP schreiben.

Interessant ist das Ausführungsmodell von JSP. Aus der JSP-Seite wird eine Javaklasse generiert, die mit einem Javakompilierer übersetzt wird. Die generierte Klasse wird dann als Servlet geladen. All dieses wird vom Servlet Conainer übernommen. Eine JSP braucht nicht von Hand übersetzt zu werden. Allerdings bekommt man auch erst vom Servlet Container einer Fehlermeldung, wenn das generierte Servlet nicht übersetzbar ist.

16.5 XML

Kapitel 17

Scala, das Java der Zukunft?

Anhang A

Javawerkzeuge

A.1 Klassenpfad und Java-Archive

Bisher haben wir Javaklassen einzeln übersetzt und jeweils pro Klasse eine Datei `.class` erhalten. Ein Javaprogramm als solches läßt sich damit gar nicht genau eingrenzen. Ein Javaprogramm ist eine Klasse mit einer Hauptmethode zusammen mit der Menge aller Klassen, die von der Hauptmethode zur Ausführung benötigt werden. Selbst Klassen, die zwar im gleichen Paket wie die Hauptklasse mit der Hauptmethode liegen, werden nicht unbedingt vom Programm benötigt und daher auch nicht geladen. Ein Programm liegt also in vielen verschiedenen Dateien verteilt. Das kann unhandlich werden, wenn wir unser Programm anderen Benutzern bereitstellen wollen. Wir wollen wohl kaum mehrere hundert Dateien auf einen Server legen, die ein potentieller Kunde dann herunterlädt.

Aus diesem Grunde bietet Java eine Möglichkeit an, die Klassen eines Programms oder einer Bibliothek zusammen in einer Datei zu bündeln. Hierzu gibt es `.jar`-Dateien. `Jar` steht für *Java archive*. Die Struktur und Benutzung der `.jar`-Dateien leitet sich von den seit alters her in Unix bekannten `.tar`-Dateien ab, wobei `tar` für *tape archive* steht und ursprünglich dazu gedacht war, Dateien gemeinsam auf einen Tonband-Datenträger abzuspeichern.

In einem Javaarchiv können nicht nur Klassendateien gespeichert werden, sondern auch Bilder und Sounddateien, die das Programm benötigt, und zusätzlich Versionsinformationen.

A.1.1 Benutzung von jar-Dateien

`jar` ist zunächst einmal ein Programm zum Verpacken von Dateien und Ordnern in eine gemeinsame Datei, ähnlich wie es auch das Programm `zip` macht. Das Programm `jar` wird in einer Javainstallation mitgeliefert und kann von der Kommandozeile gestartet werden. Ein Aufruf ohne Argumente führt zu einer Hilfemeldung:

```
sep@swe10:~/fh/prog2> jar
Syntax: jar {ctxu}[vfm0Mi] [JAR-Datei] [Manifest-Datei] [-C dir] Dateien ...
Optionen:
    -c  neues Archiv erstellen
    -t  Inhaltsverzeichnis für Archiv auflisten
    -x  benannte (oder alle) Dateien aus dem Archiv extrahieren
    -u  vorhandenes Archiv aktualisieren
```

```
    -v  ausführliche Ausgabe für Standardausgabe generieren
    -f  Namen der Archivdatei angeben
    -m  Manifestinformationen aus angegebener Manifest-Datei einbeziehen
    -0  nur speichern; keine ZIP-Komprimierung verwenden
    -M  keine Manifest-Datei für die Einträge erstellen
    -i  Indexinformationen für die angegebenen JAR-Dateien generieren
    -C  ins angegebene Verzeichnis wechseln und folgende Datei einbeziehen
Falls eine Datei ein Verzeichnis ist, wird sie rekursiv verarbeitet.
Der Name der Manifest-Datei und der Name der Archivdatei müssen
in der gleichen Reihenfolge wie die Flags ''m'' und ''f'' angegeben werden.

Beispiel 1: Archivieren von zwei Klassendateien in einem Archiv
mit dem Namen classes.jar:
       jar cvf classes.jar Foo.class Bar.class
Beispiel 2: Verwenden der vorhandenen Manifest-Datei ''meinmanifest''
und Archivieren aller
           Dateien im Verzeichnis foo/ in ''classes.jar'':
       jar cvfm classes.jar meinmanifest -C foo/ .

sep@swe10:~/fh/prog2>
```

Wie man sieht, gibt es eine Reihe von Optionen, um mit `jar` die Dateien zu erzeugen, ihren Inhalt anzuzeigen oder sie wieder auszupacken.

Erzeugen von Jar-Dateien

Das Kommando zum Erzeugen einer Jar-Datei hat folgende Form:

```
jar cf jar-file input-file(s)
```

Die Befehlsoptionen im einzelnen:

- Das `c` steht dafür, das eine Jar-Datei erzeugt werden soll.
- Das `f` steht dafür, daß der Dateiname der zu erzeugenden Jar-Datei folgt.
- `jar-file` ist der Name, den die zu erzeugende Datei haben soll. Hier nimmt man üblicherweise die Erweiterung `.jar` für den Dateinamen.
- `input-file(s)` ist eine Liste beliebiger Dateien und Ordner. Hier kann auch die aus der Kommandozeile bekannte `*`-Notation benutzt werden.

  Beispiel:
  Der Befehl

  ```
  jar cf myProg.jar *.class
  ```

  verpackt alle Dateien mit der Erweiterung `.class` in eine `Jar`-Datei namens `myProg.jar`.

Anzeige des Inhalts einer Jar-Datei

Das Kommando zur Anzeige der in einer Jar-Datei gespeicherten Dateien hat folgende Form:

```
jar tf jar-file
```

Die Befehlsoptionen im einzelnen:

- Das `t` steht dafür, das eine Auflistung der enthaltenen Dateien angezeigt werden[1] soll.
- Das `f` steht dafür, daß der Dateiname der benötigten Jar-Datei folgt.
- `jar-file` ist der Name der existierenden Jar-Datei.

Extrahieren des Inhalts der Jar-Datei

Der Befehl zum Extrahieren des Inhalts einer Jar-Datei hat folgende schematische Form:

```
jar xf jar-file [archived-file(s)]
```

Die Befehlsoptionen im einzelnen:

- Die `x`-Option gibt an, daß Dateien aus einer Jar-Datei extrahiert werden sollen.
- Das `f` gibt wieder an, daß der Name einer JAR-Datei folgt.
- `jar-file` ist der Name einer Jar-Datei, aus der die entsprechenden Dateien zu extrahieren sind.
- Optional kann noch eine Liste von Dateinamen folgen, die angibt, welche Dateien extrahiert werden sollen. Wird diese Liste weggelassen, so werden alle Dateien extrahiert.

Bei der Extraktion der Dateien werden die Dateien in dem aktuellen Verzeichnis der Kommandozeile gespeichert. Hierbei wird die originale Ordnerstruktur der Dateien verwendet, sprich Unterordner, wie sie in der Jar-Datei verpackt wurden, werden auch als solche Unterordner wieder ausgepackt.

Ändern einer Jar Datei

Schließlich gibt es eine Option, die es erlaubt, eine bestehende Jar-Datei in ihrem Inhalt zu verändern. Der entsprechende Befehl hat folgendes Format:

```
jar uf jar-file input-file(s)
```

Die Befehlsoptionen im einzelnen:

- Die Option `u` gibt an, daß eine bestehende Jar-Datei geändert werden soll.
- Das `f` gibt wie üblich an, daß der Name einer Jar-Datei folgt.
- Eine Liste von Dateinamen gibt an, daß diese zur Jar-Datei hinzuzufügen sind.

Dateien, die bereits in der Jar-Datei enthalten sind, werden durch diesen Befehl mit der neuen Version überschrieben.

[1] `t` steht dabei für *table*.

A.1.2 Der Klassenpfad

Jar-Dateien sind nicht nur dazu gedacht, daß damit Javaanwendungen einfacher ausgetauscht werden können, sondern der Javainterpreter kann Klassen direkt aus der Jar-Datei starten. Eine Jar-Datei braucht also nicht ausgepackt zu werden, um eine Klasse darin auszuführen. Es ist dem Javainterpreter lediglich mitzuteilen, daß er auch in einer Jar-Datei nach den entsprechenden Klassen suchen soll.

Angabe des Klassenpfades als Option

Der Javainterpreter hat eine Option, mit der ihm angegeben werden kann, wo er die Klassen suchen soll, die er zur Ausführung der Anwendung benötigt. Die entsprechende Option des Javainterpreters heißt `-cp` oder auch `-classpath`, wie die Hilfmeldung des Javainterpreters auch angibt:

```
sep@swe10:~/fh/prog2> java -help -cp
Usage: java [-options] class [args...]
           (to execute a class)
   or  java -jar [-options] jarfile [args...]
           (to execute a jar file)

where options include:

    -cp -classpath <directories and zip/jar files separated by :>
                  set search path for application classes and resources
```

Es läßt sich nicht nur eine Jar-Datei für den Klassenpfad angeben, sondern mehrere, und darüber hinaus auch Ordner im Dateisystem, in denen sich die Paketstruktur der Javaanwendung bis hin zu deren Klassendateien befindet. Die verschiedenen Einträge des Klassenpfades werden bei der Suche einer Klasse von vorne nach hinten benutzt, bis die Klasse im Dateisystem oder in einer Jar-Datei gefunden wurde. Die verschiedenen Einträge des Klassenpfades werden durch einen Doppelpunkt getrennt.

Angabe des Klassenpfades als Umgebungsvariable

Implizit existiert immer ein Klassenpfad in Ihrem Betriebssystem als eine sogenannte Umgebungsvariable. Sie können den Wert dieser Umgebungsvariable abfragen und ändern. Die Umgebungsvariable, die Java benutzt, um den Klassenpfad zu speichern, heißt `CLASSPATH`. Deren Wert benutzt Java, wenn kein Klassenpfad per Option angegeben wird.

Windows und Unix unterscheiden sich leicht in der Benutzung von Umgebungsvariablen. In Unix wird der Wert einer Umgebungsvariable durch ein vorangestelltes Dollarzeichen bezeichnet, also `$CLASSPATH`, in Windows wird sie durch Prozentzeichen eingeschlossen also, `%CLASSPATH%`.

Abfrage des Klassenpfades In der Kommandozeile kann man sich über den aktuellen Wert einer Umgebungsvariablen informieren.

Unix In Unix geschieht dieses leicht mit Hilfe des Befehls `echo`, dem die Variable in der Dollarnotation folgt:

```
sep@swe10:~/fh/prog2> echo $CLASSPATH
.:/home/sep/jarfiles/log4j-1.2.8.jar:/home/sep/jarfiles:
sep@swe10:~/fh/prog2>
```

Windows In Windows hingegen benutzt man den Konsolenbefehl `set`, dem der Name der Umgebungsvariablen folgt.

```
set CLASSPATH
```

Setzen des Klassenpfades Innerhalb einer Eingabeaufforderung kann für diese Eingabeaufforderung der Wert einer Umgebungsvariablen geändert werden. Auch hierin unterscheiden sich Unix und Windows marginal:

Unix

> **Beispiel:**
> Wir fügen dem Klassenpfad eine weitere Jar-Datei an ihrem Beginn an:
>
> ```
> sep@swe10:~/fh/prog2> export CLASSPATH=~/jarfiles/jugs.jar:$CLASSPATH
> sep@swe10:~/fh/prog2> echo $CLASSPATH
> /home/sep/jarfiles/jugs.jar:.:/home/sep/jarfiles/log4j-1.2.8.jar:/home/sep/jarfiles:
> ```

Windows In Windows werden Umgebungsvariablen auch mit dem `set`-Befehl geändert. Hierbei folgt dem Umgebungsvariablennamen mit einem Gleichheitszeichen getrennt der neue Wert.

> **Beispiel:**
> Der entsprechende Befehl des letzten Beispiels ist in Windows:
> `set CLASSPATH=~\jarfiles\jugs.jar:%CLASSPATH%`

A.2 APT

A.3 Applets testen mit: appletviewer

A.4 native2ascii

A.5 Klassendateien analysieren mit javap

A.6 Parsergeneratoren

A.6.1 javacc

A.6.2 antlr

Anhang B

Jugs: Java interpetiert

Anhang C

Generierung von Klassen für XML DTDs

C.1 Introduction

Parsers are a well understood concept. Usually a parser is used to read some text according to a context free grammar. It is tested if the text can be produced with the rules given in the grammar. In case of success a result tree is constructed. Different strategies for parsing can be applied. Usually a parser generator program is used. A textual representation of the grammar controls the generation of the parser. In functional programming languages parsers can easily be hand coded by way of parser cominators[HM96]. We will show, how to apply the concept of parser combinators to XML document type decriptions in Java.

C.1.1 Functional Programming

The main building block in functional programming is a function definition. Functions are first class citizens. They can be passed as argument to other functions, whithout being wrapped within some data object. Functions which take other function as argument are said to be of *higher order*. A function that creates a new result function from different argument functions is called a *combintor*.

Parser Combinators

A parser is basically a function. It consumes a token stream and results a list of several parse results. An empty list result can be interpreted as, no successfull parse[Wad85].

The following type definition characterizes a parser in Haskell.

HsParse.hs

```
type Parser result token = [token] -> [(result,[token])]
```

The type `Parser` is generic over the type of the token and the type of a result. A parser is a function. The result is a pair: the first element of the pair is the result of the parse, the second

element is the list of the remaining token (the parser will have consumed some tokens from the token list).

In functional programming languages a parser can be written by way of combinators. A parser consists of several basic parsers which are combined to a more complex parser. Simple parsers test if a certain token is the next token in the token stream.

The basic parsers, which tests exactely for one token, can in Haskell be written as:

HsParse.hs

```
getToken tok [] = []
getToken tok (x:xs)
  |tok==x = [(x,xs)]
  |otherwise = []
```

There are two fundamental ways to combine parsers to more complex parser:

- the sequence of two parsers.
- the alternativ of two parsers.

In functional programming languages combinator functions can be written, to implement these two ways to construct more complex parsers.

In the alternativ combination of two parser, first the first parser is applied to the token stream. Only if this does not succeed, the second parser is applied to the token stream.

In Haskell this parser can be implemented as:

HsParse.hs

```
alt p1 p2 toks
  |null res1 = p2 toks
  |otherwise = res1
 where
  res1 = p1 toks
```

In the sequence combination of two parsers, first the first parser is applied to the token stream and then the second parser is applied to the next token stream of the results of the first parse.

In Haskell this parser can be implemented as:

HsParse.hs

```
seqP p1 p2 toks = concat
  [[((rs1,rs2),tks2) |(rs2,tks2)<-(p2 tks1) ]
  |(rs1,tks1)<-p1 toks]
```

The result of a sequence of two parser is the pair of the two partial parses. Quite often a joint result is needed. A further function can be provided to apply some function to a parse result in order to get a new result:

HsParse.hs

```
mapP f p toks = [(f rs,tks)|(rs,tks)<-p toks]
```

With these three combinators basically all kinds of parsers can be defined.

Beispiel:
A very simple parser implementation in Haskell:

HsParse.hs

```
parantheses = mapP (\((x1,x2),x3) -> x2)
     (getToken '(' `seqP` a `seqP` getToken ')')

a = getToken 'a' `alt` parantheses

main = do
  print (parantheses "((((a))))")
  print (parantheses "((((a)))")
  print (parantheses "((((a))))bc")
```

The program has the following output:

```
sep@linux:~/fh/xmlparslib/examples> src/HsParse
[('a',"")]
[]
[('a',"bc")]
sep@linux:~/fh/xmlparslib/examples>
```

In the first call the string could complete be parsed, in the second call the parser failed and in the third call the parser succeeded but two further tokens where not consumed by the parser.

Further parser combinators can be expressed with the two combinators above. Typical combinators define the repetitive application of a parser, which is expressed by the symbols `+` and `*` in the extended Backus-Naur-form[NB60].

C.1.2 XML

XML documents are tree like structured documents. Common parser libraries are available to parse the textual representation of an XML document and to build the tree structure. In object orientated languages a common modell for the resulting tree structure is used, the *distributed object modell (DOM)*[**?**].

Document Type Definition

The document type of an XML document describes, which tags can be used within the document and which children these elements are allowed to have. The document type defiition (DTD) in XML is very similar to a context free grammar. [1] A DTD describes what kind of children elements a certain element can have in a document. In a DTD we find the typical elements of grammars:

- sequences of elements, denoted by a comma `,`.

[1] Throughout this paper we will ignore XML attributes.

- alternatives of elements donoted by the vertical bar `|`.
- several kinds of repetition denoted by `+`, `*` and `?`.

Consider the following DTD, which we will use as example throughout this paper:

album.dtd

```
<!DOCTYPE musiccollection SYSTEM "musiccollection.dtd" [
  <!ELEMENT musiccollection (lp|cd|mc)*>
  <!ELEMENT lp (title,artist,recordingyear?,track+,note*)>
  <!ELEMENT cd (title,artist,recordingyear?,track+,note*)>
  <!ELEMENT mc (title,artist,recordingyear?,track+,note*)>
  <!ELEMENT track (title,timing?)>
  <!ELEMENT note (#PCDATA|author)*>

  <!ELEMENT timing (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT artist (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT recordingyear (#PCDATA)>
]>
```

The following XML document is build according to the structure defined in this DTD.

mymusic.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<musiccollection>
  <lp>
    <title>White Album</title>
    <artist>The Beatles</artist>
    <recordingyear>1968</recordingyear>
    <track><title>Revolution 9</title></track>
    <note><author>sep</author>my first lp</note>
  </lp>
  <cd>
    <title>Open All Night</title>
    <artist>Marc Almond</artist>
    <track><title>Tragedy</title></track>
    <note>
     <author>sep</author>
     Marc sung tainted love in the bandSoft Cell</note>
  </cd>
</musiccollection>
```

Another much simpler example is the following document:

simple.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<musiccollection/>
```

As can be seen, DTDs are very similar to grammars. And in fact, the same techniques can be applied for DTDs as for grammars. In [**?**] a Haskell combinator library for DTDs is presented. In the next sections we will try to apply these techniques in Java.

C.1.3 Java

In Java the main building block is a class, which defines a set of objects. A class can contain methods. These methods can represent functions. It is therefore natural to represent a Parser type by way of some class and the different parser combinators by way of subclasses in Java.

Generic Types

As we have seen in functional programming, parser combinators make heavily use of generic types. Fortunatly with release 1.5 of Java generic types are part of Java's type system. This makes the application of parser combinator techniques in Java tracktable.

C.2 Tree Parser

As we have seen a DTD is very similar to a grammar. The main difference is, that a grammar describes the way tokens are allowed in a token stream, whereas a DTD describes which elements can occur in a tree structure. Therefore a parser which we write for a DTD will not try to test, if a token stream can be build with the rules, but if a certain tree structure can be build with the rules in the DTD. What we get is a parser, which takes a tree and not a stream as input.

C.2.1 Parser Type

We will now define classes to represent parsers over XML trees in Java.

First of all, we need some class to represent the result of a parse. In the Haskell application above we used a pair for the result of a parse. The first element of the pair represented the actual result data constructed, the second element the remaining token stream. In our case, we want to parse over XML trees. Therefore the token List will be a list of XML nodes as defined in DOM. We use the utility class `Tuple2` as defined in [Pan04] to represent pairs.

Tuple1.java

```
package name.panitz.crempel.util;

public class Tuple1<t1> {
  public t1 e1;
  public Tuple1(t1 a1){e1=a1;}
  String parenthes(Object o){return "("+o+")";}
  String simpleToString(){return e1.toString();}
  public String toString(){return parenthes(simpleToString());}
  public boolean equals(Object other){
    if (! (other instanceof Tuple1)) return false;
    return e1.equals(((Tuple1)other).e1);
  }
}
```

Tuple2.java

```
package name.panitz.crempel.util;

```

```
public class Tuple2<t1,t2> extends Tuple1<t1>{
  public t2 e2;
  public Tuple2(t1 a1,t2 a2){super(a1);e2=a2;}
  String simpleToString(){
    return super.simpleToString()+","+e2.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple2)) return false;
    return super.equals(other)&& e2.equals(((Tuple2)other).e2);
  }
}
```

Tuple3.java

```
package name.panitz.crempel.util;

public class Tuple3<t1,t2,t3> extends Tuple2<t1,t2>{
  public t3 e3;
  public Tuple3(t1 a1,t2 a2,t3 a3){super(a1,a2);e3=a3;}
  String simpleToString(){
    return super.simpleToString()+","+e3.toString();}
  public boolean equals(Object other){
    if (! (other instanceof Tuple3)) return false;
    return super.equals(other)&& e3.equals(((Tuple3)other).e3);
  }
}
```

Function1.java

```
package name.panitz.crempel.util;

public interface Function1<arg,result>{result eval(arg a);}
```

Visitor.java

```
package name.panitz.crempel.util;

public interface Visitor<arg,result>
                      extends Function1<arg,result>{
}
```

The following class represents the result of a parse.

ParseResult.java

```
package name.panitz.crempel.util.xml.parslib;

import name.panitz.crempel.util.Tuple2;
import java.util.List;
import org.w3c.dom.Node;

public class ParseResult<a> extends Tuple2<a,List<Node>>{
  public ParseResult(a n,List<Node> xs){super(n,xs);}
```

```
  public boolean failed(){return false;}
}
```

This class is generic over the actual result type. We added a method for testing, whether the parse was successful.

We define a special subclass for parse results, which denotes unsuccessful parses.

Fail.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import org.w3c.dom.Node;

public class Fail<a> extends ParseResult<a>{
  public Fail(List<Node> xs){super(null,xs);}
  public boolean failed(){return true;}
}
```

Now where we have defined what a parse result looks like, we can define a class to describe a parser. A parser is some object, which has a method `parse`. This maethod takes a list of DOM nodes as argument and returns a parse result:

Parser.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.Node;
import name.panitz.crempel.util.Maybe;
import name.panitz.crempel.util.Tuple2;
import name.panitz.crempel.util.Function1;

public interface  Parser<a>{
  public ParseResult<a> parse(List<Node> xs);
```

C.2.2 Combinators

Now, where we have a parser class, we would like to add combinator function to this class. We add the following combinators:

- `seq` to create a sequence of this parser with some other parser.
- `choice` to create an alternative of this or some other parser.
- `star` to create a zero or more repetition parser of this parser.
- `plus` to create a one or more repetition parser of this parser.

- `query` to create a zero or one repetition parser of this parser.
- `map` to create a parser from this parser, which modifies the parse result with some further method.

This leads to the following method signatures in the interface `Parser`:

Parser.java

```
  public <b> Parser<Tuple2<a,b>> seq(Parser<b> p2);
  public <b extends a> Parser<a> choice(Parser<b> p2);
  public Parser<List<a>> star();
  public Parser<List<a>> plus();
  public Parser<Maybe<a>> query();
  public <b> Parser<b> map(Function1<a,b> f);
}
```

The sequence combinator creates a parser which has a pair of the two partial results as common result. The alternative parser will only allow a second parser, which has the same result type as this parser. This will lead to some complication later. Results of repetitions of more than one time are represented as lists of the partial results. For the optional repretition of zero or one time we use the class `Maybe` as of [Pan04] result type. It has two subclasses, `Just` to denote there is such a result and `Nothing` to denote the contrary.

In the following we assume an abstract class `AbstractParser`, which implements these functions. We will give the definition of `AbstractParser` later in this paper.

Optional Parser

Let us start with the parser which is optional as denoted by the question mark `?` in a DTD. We write a class to represent this parser. This class contains a parser. In the method `parse`, this given parser is applied to the list of nodes. If this fails a `Nothing` object is returned, otherwise a `Just` object, which wrappes the result of the successful parse, is returned.

We get the following simple Java class:

Optional.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import name.panitz.crempel.util.Maybe;
import name.panitz.crempel.util.Nothing;
import name.panitz.crempel.util.Just;
import org.w3c.dom.Node;

public  class Optional<a> extends  AbstractParser<Maybe<a>> {
  final Parser<a> p;
  public Optional(Parser<a> _p){p=_p;}
  public ParseResult<Maybe<a>> parse(List<Node> xs){
    final ParseResult<a> res = p.parse(xs);
    if (res.failed())
      return new ParseResult<Maybe<a>>(new Nothing<a>(),xs);
```

```
    return new ParseResult<Maybe<a>>(new Just<a>(res.e1),res.e2);
  }
}
```

Sequence

Next we implement a Java class which represents the sequence of two parsers. This class contains two inner parsers. The first one gets applied to the list of nodes, in case of success, the second is applied to the remaining list of the successful first parse.

We get the following simple Java class:

Seq.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import name.panitz.crempel.util.Tuple2;
import org.w3c.dom.Node;

public  class Seq<a,b> extends  AbstractParser<Tuple2<a,b>> {
  final Parser<a> p1;
  final Parser<b> p2;

  public Seq(Parser<a> _p1,Parser<b> _p2){p1=_p1;p2=_p2;}

  public ParseResult<Tuple2<a,b>> parse(List<Node> xs){
    ParseResult<a> res1 = p1.parse(xs);
    if (res1.failed())
      return new Fail<Tuple2<a,b>>(xs);
    ParseResult<b> res2 = p2.parse(res1.e2);
    if (res2.failed())
      return new Fail<Tuple2<a,b>>(xs);
    return new ParseResult<Tuple2<a,b>>
                 (new Tuple2<a,b>(res1.e1,res2.e1),res2.e2);
  }
}
```

Choice

The alternative parser of parser can be defined in almost exactly the way as has been done in the intrductory Haskell example. First the first parser ist tried. If this succeeds its result is used, otherwise the second parser is applied. However, we need to create new objects of classes `ParseResult` and `Fail`. This is due to the fact that even if `b extends c`, `ParseResult<b>` does not extend `ParseResult<c>`.

Choice.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import java.util.ArrayList;
```

```
import name.panitz.crempel.util.Tuple2;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class  Choice<c,a extends c,b extends c>
  extends  AbstractParser<c> {
  final Parser<a> p1;
  final Parser<b> p2;

  public Choice(Parser<a> _p1,Parser<b> _p2){p1=_p1;p2=_p2;}

  public ParseResult<c> parse(List<Node> xs){
    final ParseResult<a> res1 =  p1.parse(xs);
    if (res1.failed()){
      final ParseResult<b> res2 = p2.parse(xs);
      if (res2.failed()) return new Fail<c>(xs);
      return new  ParseResult<c>(res2.e1,res2.e2);
    }
    return new  ParseResult<c>(res1.e1,res1.e2);
  }
}
```

Repetitions

Reptetitive parsers try to apply a parser several times after another. We define one common class for repetitive parses. Subclasses will differentiate if at least one time the parser needs to be applied successfully.

Repetition.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.Node;
```

We need an parser to be applied in a repetition and a flag to signify, if the parser needs to be applied sucessfulle at least one time:

Repetition.java

```
public  class Repetition<a> extends  AbstractParser<List<a>> {
  final Parser<a> p;
  final boolean atLeastOne;
  public Repetition(Parser<a> _p,boolean one){
    p=_p;atLeastOne=one;
  }
```

Within the method `parse` the parser is applied is long as it does not fail. The results of the parses are added to a common result list.

Repetition.java

```
  public ParseResult<List<a>> parse(List<Node> xs){
    final List<a> resultList = new ArrayList<a>();
    int i = 0;

    while (true){
      final ParseResult<a> res = p.parse(xs);
      xs=res.e2;

      if (res.failed()) break;
      resultList.add(res.e1);
    }

    if (resultList.isEmpty()&& atLeastOne)
      return new Fail<List<a>>(xs);
    return new ParseResult<List<a>>(resultList,xs);
  }
}
```

Simple subclasses can be defined for the two kinds of repetition in a DTD.

Star.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;

public class Star<a> extends Repetition<a> {
  public Star(Parser<a> p){super(p,false);}
}
```

Plus.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;

public class Plus<a> extends Repetition<a> {
  public Plus(Parser<a> p){super(p,true);}
}
```

Map

Eventually we define the parser class for modifying the result of a parser. To pass the function to be applied to the parser we use an object, which implements the interface `Function1` from [Pan04].

The implementation is straight forward. Apply the parser and in case of success create a new parse result object from the original parse result.

Map.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import org.w3c.dom.Node;
import name.panitz.crempel.util.Function1;

public class Map<a,b> extends AbstractParser<b> {
  final Parser<a> p;
  final Function1<a,b> f;

  public Map(Function1<a,b> _f,Parser<a> _p){f=_f;p=_p;}

  public ParseResult<b> parse(List<Node> xs){
    final ParseResult<a> res = p.parse(xs);
    if (res.failed()) return new Fail<b>(xs);
    return new ParseResult<b>(f.eval(res.e1),res.e2);
  }
}
```

Abstract Parser Class

Now where we have classes for all kinds of parser combinators, we can use these in an abstract parser class, which implements the combinator methods of the parser interface. We simply instantiate the corresponding parser classes.

AbstractParser.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import name.panitz.crempel.util.Tuple2;
import name.panitz.crempel.util.Maybe;
import name.panitz.crempel.util.Function1;

public abstract class AbstractParser<a> implements Parser<a>{
  public <b> Parser<Tuple2<a,b>> seq(Parser<b> p2){
    return new Seq<a,b>(this,p2);
  }

  public <b extends a> Parser<a> choice(Parser<b> p2){
    return new Choice<a,a,b>(this,p2);
  }

  public Parser<List<a>> star(){return new Star<a>(this);}
  public Parser<List<a>> plus(){return new Plus<a>(this);}
  public Parser<Maybe<a>> query(){return new Optional<a>(this);}
  public <b> Parser<b> map(Function1<a,b> f){
    return new Map<a,b>(f,this);
  }
```

```
}
```

Note that the method `choice` cannot be written as general as we would like to. We cannot specify that we need one smallest common superclass of the two result types.

C.2.3 Atomic Parsers

Now we are able to combine parsers to more complex parsers. We are still missing the basic parser like the function `getToken` in our Haskell parser above. In XML documents there are several different basic units: PCDATA, Elements and Empty content.

PCDATA

The basic parser, which checks for a PCDATA node simply checks the node type of the next node. If there is a next node of type text node, then the parser succeeds and consumes the token. Otherwise it fails. We define this parser with result type `String`. The resulting `String` will contain the contents of the text node.

PCData.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import org.w3c.dom.Node;

public class PCData extends AbstractParser<String> {

  public ParseResult<String> parse(List<Node> xs){
    if (xs.isEmpty()) return new Fail<String>(xs);
    final Node x = xs.get(0);
    if (x.getNodeType()==Node.TEXT_NODE)
      return new ParseResult<String>
                  (x.getNodeValue(),xs.subList(1,xs.size()));
System.out.println("expected pcdata but got: "+x);
    return new Fail<String>(xs);
  }
}
```

Empty

A very simple parser, does always succeed with some result and does not consume any node from the input.

Return.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import org.w3c.dom.Node;

```

```
public class Return<a> extends AbstractParser<a> {
  final a returnValue;
  public Return(a r){returnValue=r;}
  public ParseResult<a> parse(List<Node> xs){
    return new ParseResult<a>(returnValue,xs);
  }
}
```

Element

The probably most interesting parser reads an element node and proceeds with the children of the element node. The children of the element node are processed with a given parser.

Therefore the element parser needs two arguments:

- the name of the element to be read.
- the parser to be applied to the children of the element.

Closure.java

```
package name.panitz.crempel.util;

public interface Closure<result>{
  public result eval();
}
```

Element.java

```
package name.panitz.crempel.util.xml.parslib;

import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import name.panitz.crempel.util.Closure;


public class Element<a> extends AbstractParser<a> {
  Parser<a> p=null;
  private final Closure<Parser<a>> pc;
  final String elementName;

  public Element(String n,Parser<a> _p){elementName=n;p=_p;pc=null;}
  public Element(String n,Closure<Parser<a>> _p){elementName=n;pc=_p;}

  public Parser<a> getP(){if (p==null) p=pc.eval(); return p;}
```

For simplicity reasons, we will neglect text nodes from list of nodes, which consist solely of whitespace:

Element.java

```
  public ParseResult<a> parse(List<Node> xs){
    if (xs.isEmpty()) {return new Fail<a>(xs);}
    Node x = xs.get(0);
    while (   x.getNodeType()==Node.TEXT_NODE
           && x.getNodeValue().trim().length()==0
          ){
      xs=xs.subList(1,xs.size());
      if (xs.isEmpty()) return new Fail<a>(xs);

      x = xs.get(0);
    }
```

Then we compare the node name with the expected node name:

Element.java

```
    final String name = x.getNodeName();
System.out.println(name+" <-> "+elementName);
    if (!name.equals(elementName))
      return new Fail<a>(xs);
```

Eventually we get the children of the node and apply the given parser to this list.

Element.java

```
    final List<Node> ys = nodelistToList(x.getChildNodes());
    ParseResult<a> res = getP().parse(ys);

    if( res.failed()) return new Fail<a>(xs);
```

If this succeeds we ensure that only neglectable whitespace nodes follow in the list of nodes

Element.java

```
    for (Node y :res.e2){
      if (y.getNodeType()!=Node.TEXT_NODE
              || y.getNodeValue().trim().length()!=0)
          return new Fail<a>(xs);
    }
```

Eventually the result can be constructed.

Element.java

```
    return new ParseResult<a>(res.e1,xs.subList(1,xs.size()));
  }
```

The following method has been used to convert DOM node list into a java list of nodes.

Element.java

```
  static public List<Node> nodelistToList(NodeList xs){
    List<Node> result = new ArrayList<Node>();
```

```
    for (int i=0;i<xs.getLength();i=i+1){
      result.add(xs.item(i));
    }
    return result;
  }
}
```

C.2.4 Building parsers

We have got everything now to implement parsers for a DTD in Java.

Defining a Parser

We can write a parser, which checks if a certain XML document is build according to the type definition as defined for our music collection example. As result for the parse we simply return some `String` object.

MusiccollectionParser.java

```
package name.panitz.crempel.test;

import java.util.List;
import org.w3c.dom.Node;
import name.panitz.crempel.util.*;
import name.panitz.crempel.util.xml.parslib.*;

public class MusiccollectionParser extends AbstractParser<String>{
```

First of all we define some objects of different `Function1` types, which are needed to construct the final result of the parses. First an object for concatenating two `String` objects:

MusiccollectionParser.java

```
  final private Function1<Tuple2<String,String>,String> concat
    = new Function1<Tuple2<String,String>,String>(){
         public String eval(Tuple2<String,String> p){
           return p.e1+p.e2;
         }
      };
```

The second function retries a `String` object from a `Maybe<String>` object.

MusiccollectionParser.java

```
  final private Function1<Maybe<String>,String> getString
    = new Function1<Maybe<String>,String>(){
         public String eval(Maybe<String> p){
           if (p instanceof Nothing) return "";
           return ((Just<String>)p).getJust();
         }
      };
```

Eventually we provide a function object, which concatenates a list of `String` object into a single `String` object.

MusiccollectionParser.java

```
  final private Function1<List<String>,String> concatList
    = new Function1<List<String>,String>(){
         public String eval(List<String> xs){
           final StringBuffer result = new StringBuffer();
           for (String x:xs) result.append(x);
           return result.toString();
         }
      };
```

Now we can define parsers for the elements as defined in the dtd. The most simple parsers are for those elements, which have only text nodes as children:

MusiccollectionParser.java

```
  final private Parser<String> recordingyear
    = new Element<String>("recordingyear",new PCData());
  final private Parser<String> artist
    = new Element<String>("artist",new PCData());
  final private Parser<String> title
    = new Element<String>("title",new PCData());
  final private Parser<String> timing
    = new Element<String>("timing",new PCData());
  final private Parser<String> author
    = new Element<String>("author",new PCData());
```

Parsers for the remaining elements are combined from these basic parsers.

MusiccollectionParser.java

```
  final private Parser<String> note
    = new Element<String>
       ("note",author.seq(new PCData()).map(concat));

  final private Parser<String> track
    = new Element<String>("track"
        ,title.seq(timing.query().map(getString)).map(concat));

  final private Parser<String> content
    =    title
        .seq(artist).map(concat)
        .seq(recordingyear.query().map(getString)).map(concat)
        .seq(track.plus().map(concatList)).map(concat)
        .seq(note.star().map(concatList)).map(concat);

  final private Parser<String> lp
    = new Element<String>("lp",content);
  final private Parser<String> cd
    = new Element<String>("cd",content);
```

```
  final private Parser<String> mc
    = new Element<String>("mc",content);

  final private Parser<String> musiccollection
   = new Element<String>
      ("musiccollection",lp.choice(cd).choice(mc)
                            .star().map(concatList));

  public ParseResult<String> parse(List<Node> xs){
    return musiccollection.parse(xs);
  }
}
```

Starting the parser

We have a tree parser over XML documents of type `musiccollection`. This parser can be applied to a list of dom objects:

MainParser.java

```
package name.panitz.crempel.test;

import name.panitz.crempel.util.*;
import name.panitz.crempel.util.xml.parslib.*;

import org.w3c.dom.*;
import java.util.*;
import java.io.*;
import javax.xml.parsers.*;

public class MainParser{
  public static void main(String [] args){
    System.out.println(pFile(args[0],args[1]));
  }

  public static Object pFile(String parserClass,String fileName){
    try{
      DocumentBuilderFactory factory
        = DocumentBuilderFactory.newInstance();

      factory.setIgnoringElementContentWhitespace(true);
      factory.setCoalescing(true);
      factory.setIgnoringComments(true);

      Document doc
        =factory.newDocumentBuilder()
        .parse(new File(fileName)) ;
      doc.normalize();
      List<Node> xs = new ArrayList<Node>();
      xs.add(doc.getChildNodes().item(0));
```

```
31
32        Parser<Object> p
33         = (Parser<Object>)Class.forName(parserClass).newInstance();
34
35        Tuple2 res = p.parse(xs);
36
37        return res.e1;
38      }catch (Exception e){e.printStackTrace();return null;}
39    }
40  }
```

The parser checks the document for its structure and returns the text of the document:

```
sep@linux:~/fh/xmlparslib/examples> java -classpath classes/  name.panitz.crempel.test.MainParser
  name.panitz.crempel.test.MusiccollectionParser src/mymusic.xml
White AlbumThe Beatles1968Revolution 9sepmy first lpOpen All NightMarc AlmondTragedysepMarc sung tainted love
sep@linux:~/fh/xmlparslib/examples>
```

C.3 Building a Tree Structure

In the last section we presented a library for writing tree parsers for a certain DTD. The actual parser needed to be hand coded for the given DTD. As has been seen this was very tedious and boring mechanical hand coding, which can be made automatically. As a matter of fact, a parser can easily be generated, but for the result type of the parser. In this sections we will develop a program that will generate tree classes for a DTD and a parser, which has an object of these tree classes as result.

C.3.1 Java types for DTDs

C.3.2 An algebraic type for DTDs

In this section we will make heavy use of the algebraic type framework as presented in [**?**]. First of all we define an algebraic type to represent a definition clause in a DTD (i.e. the right hand side of an element definition).

In a DTD there exist the following building blocks:

- PCData
- some element name
- Any
- Empty
- one or more repetition
- zero or more repetition
- zero or one repetition

- a list of choices
- a list of sequence items

We can express this in the following algebraic type definition.

DTDDef.java

```
package name.panitz.crempel.util.xml.dtd.tree;
import name.panitz.adt.*;

public @Data class DTDDef {
  @Constr public void DTDPCData(){}
  @Constr public void DTDTagName(String tagName){}
  @Constr public void DTDAny(){}
  @Constr public void DTDEmpty(){}{}
  @Constr public void DTDPlus(DTDDef dtd){}
  @Constr public void DTDStar(DTDDef dtd){}
  @Constr public void DTDQuery(DTDDef dtd){}
  @Constr public void DTDSeq(java.util.List<DTDDef> seqParts){}
  @Constr public void DTDChoice(java.util.List<DTDDef> choiceParts){}
}
```

For this algenbraic type definition we get generated a set of tree classes and a visitor framework.

Simple visitors vor DTDDef

As a first exercise we write two simple visitor classes for the type `DTDDef`.

Show The first visitor returns a textual representation of the `DTD` object.

DTDShow.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import java.util.List;

public class DTDShow extends DTDDefVisitor<String>{
  public String show(DTDDef def){return ((DTDDefAdt)def).welcome(this);}

  public String visit(DTDPCData _){return "#PCDATA";};
  public String visit(DTDTagName x){return x.tagName;}
  public String visit(DTDEmpty x){return "Empty";}
  public String visit(DTDAny x){return "Any";}
  public String visit(DTDPlus x){return show(x.dtd)+"+";};
  public String visit(DTDStar x){return show(x.dtd)+"*";};
  public String visit(DTDQuery x){return show(x.dtd)+"?";};
  public String visit(DTDSeq s){
    return aux(this,",",s.seqParts);}
  public String visit(DTDChoice c){
```

```
    return aux(this,"|",c.choiceParts);}

  private String aux
              (DTDShow visitor,String sep,List<DTDDef> parts){
    StringBuffer result=new StringBuffer("(");
    boolean first = true; for (DTDDef x:parts){
      if (first) first=false; else result.append(sep);
      result.append(show(x));
    }
    result.append(")");
    return result.toString();
  }
}
```

ShowType Now we write a visitor, which returns in Java syntax the Java type, which we associate to a certain DTD construct:

ShowType.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.*;
import java.util.List;
import java.util.ArrayList;

public class ShowType extends DTDDefVisitor<String>{
  public String visit(DTDPCData x){return "String";}
  public String visit(DTDTagName x){return x.tagName;}
  public String visit(DTDEmpty x){return "Object";}
  public String visit(DTDAny x){return "Object";}
  public String visit(DTDPlus x){return "List<"+showType(x.dtd)+">";}
  public String visit(DTDStar x){return "List<"+showType(x.dtd)+">";}
  public String visit(DTDQuery x){return "Maybe<"+showType(x.dtd)+">";}
  public String visit(DTDSeq x){
    return listAsType((List<DTDDef>) x.seqParts);}

  public String visit(DTDChoice x){
    List<DTDDef> parts = x.choiceParts;
    if (parts.size()==1) return showType(parts.get(0));
    StringBuffer result=new StringBuffer("Choice");
    for (DTDDef y:((List<DTDDef>) parts))
      result.append("_"+typeToIdent(showType(y)));
    return result.toString();
  }

  private  String listAsType(List<DTDDef> xs){
    int size=xs.size();

    if (size==1) return showType(xs.get(0));
    StringBuffer result = new StringBuffer();
```

```
    for (Integer i:new FromTo(2,size)){
      result.append("Tuple2<");
    }
    boolean first=true;
    for (DTDDef dtd:xs){
      if (!first) result.append(",");
      result.append(showType(dtd));
      if (!first) result.append(">");
      first=false;
    }
    return result.toString();
  }

  public  String showType(DTDDef def){
    return ((DTDDefAdt)def).welcome(this);}

  public  static String sShowType(DTDDef def){
    return new ShowType().showType(def);}

  static public String typeToIdent(String s ){
    return s.replace('<','_').replace('>','_').replace('.','_');
  }
}
```

C.3.3 Generation of tree classes

In this section we write a

GenerateADT.java

```
package name.panitz.crempel.util.xml.dtd;

//import name.panitz.crempel.util.adt.parser.ADT;
import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.*;
//import name.panitz.crempel.util.adt.*;
import java.util.List;
import java.util.ArrayList;
import java.io.Writer;
import java.io.StringReader;
import java.io.FileWriter;
import java.io.StringWriter;
import java.io.IOException;

public class GenerateADT extends DTDDefVisitor<String>{
  final String elementName;
  public GenerateADT(String e){elementName=e;}

  public String visit(DTDPCData x){
   return "  public @Constr void Con(String pcdata){}";}
```

```
21    public String visit(DTDTagName x){
22      final String typeName = ShowType.sShowType(x);
23      return "  public @Constr void Con("+typeName+" the"+typeName+"){}";}
24    public String visit(DTDEmpty x){return "";}
25    public String visit(DTDAny x){return "";}
26    public String visit(DTDPlus x){
27      return "  public @Constr void Con(List<"+ShowType.sShowType(x.dtd)+"> xs){
28    public String visit(DTDStar x){
29      return "  public @Constr void Con(List<"+ShowType.sShowType(x.dtd)+"> xs){
30    public String visit(DTDQuery x){
31      return "  public @Constr void Con(Maybe<"+ShowType.sShowType(x.dtd)+"> xs)
32    public String visit(DTDSeq x){
33      StringBuffer result = new StringBuffer("public @Constr void Con(");
34      boolean first = true;
35      for (DTDDef dtd :x.seqParts){
36        if (!first) result.append(",");
37        final String typeName = ShowType.sShowType(dtd);
38        result.append(typeName+" the"+ShowType.typeToIdent(typeName));
39
40        first=false;
41      }
42      result.append("){}");
43      return result.toString();
44    }
45    public String visit(DTDChoice x){
46      StringBuffer result = new StringBuffer();
47      for (DTDDef dtd :x.choiceParts){
48        String typeName = ShowType.sShowType(dtd);
49        final String varName = ShowType.typeToIdent(typeName);
50
51        result.append("\n  public @Constr void C"+elementName+varName
52                                +"("+typeName+" the"+varName+"){}");
53      }
54      return result.toString();
55    }
56
57    public static String generateADT(String element,DTDDef def){
58     return ((DTDDefAdt)def).welcome(new GenerateADT(element));}
59
60    public static void  generateADT
61       (String paket,String path,List<Tuple3<Boolean,String,DTDDef>> xs){
62      try {
63        for (Tuple3<Boolean,String,DTDDef>x:xs){
64          Writer out = new FileWriter(path+"/"+x.e2+".java");
65          out.write("package "+paket+";\n");
66          out.write("import java.util.List;\n");
67          out.write("import name.panitz.crempel.util.Maybe;\n");
68          out.write("import name.panitz.adt.*;\n");
69          out.write("public @Data class "+x.e2+"{\n");
70          out.write(generateADT(x.e2,x.e3));
```

```
71          out.write("}");
72          out.close();
73         }
74        }catch (Exception e){e.printStackTrace();}
75    }
76  }
```

C.3.4 Generation of parser code

ParserCode.java

```
1  package name.panitz.crempel.util.xml.dtd;
2
3  import name.panitz.crempel.util.xml.dtd.tree.*;
4  import name.panitz.crempel.util.*;
5  //import name.panitz.crempel.util.adt.*;
6  import java.util.List;
7  import java.util.ArrayList;
8  import java.io.Writer;
9  import java.io.StringWriter;
10 import java.io.IOException;
11
12 public class ParserCode extends DTDDefVisitor<String>{
13   final String elementName;
14   public ParserCode(String e){elementName=e;}
15
16   public String visit(DTDPCData x){return "new PCData()";}
17   public String visit(DTDTagName x){return "getV"+x.tagName+"()";}
18   public String visit(DTDEmpty x){return "new Return(null)";}
19   public String visit(DTDAny x){return null;}
20   public String visit(DTDPlus x){return parserCode(x.dtd)+".plus()";}
21   public String visit(DTDStar x){return parserCode(x.dtd)+".star()";}
22   public String visit(DTDQuery x){return parserCode(x.dtd)+".query()";}
23   public String visit(DTDSeq x){
24     StringBuffer result = new StringBuffer();
25     boolean first = true;
26     for (DTDDef dtd:(List<DTDDef>) x.seqParts){
27       if (!first){result.append(".seq(");}
28       result.append(parserCode(dtd));
29       if (!first){result.append(")");}
30       first=false;
31     }
32     return result.toString();
33   }
34
35   public String visit(DTDChoice x){
36     final List<DTDDef> xs =  x.choiceParts;
37     if (xs.size()==1) {
38       final DTDDef ch = xs.get(0);
39       final String s  = parserCode(ch);
```

```
      return s;
    }
    StringBuffer result = new StringBuffer();
    boolean first = true;
    for (DTDDef dtd:xs){
      final String argType = ShowType.sShowType(dtd);
      final String resType = elementName;
      if (!first){result.append(".choice(");}
      result.append(parserCode(dtd));
      result.append(".<"+resType+">map(new Function1<");
      result.append(argType);
      result.append(",");
      result.append(resType);
      result.append(">(){");
      result.append("\n    public "+resType+" eval("+argType+" x){");

      String typeName = ShowType.sShowType(dtd);
      final String varName = ShowType.typeToIdent(typeName);
      result.append("\n      return ("+resType+")new C"
             +elementName+varName
             +"(x);");
      result.append("\n    }");
      result.append("\n  })");
      if (!first){result.append(")");}
      first=false;
    }
    return result.toString();
  }

  public String parserCode(DTDDef def){
    return ((DTDDefAdt)def).welcome(this);}

  public static String parserCode(DTDDef def,String n){
   return new ParserCode(n).parserCode(def);}
}
```

WriteParser.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import name.panitz.crempel.util.*;
//import name.panitz.crempel.util.adt.*;
import java.util.List;
import java.util.ArrayList;
import java.io.Writer;
import java.io.StringWriter;
import java.io.IOException;

public class WriteParser extends DTDDefVisitor<String>{
```

```
13    final String elementName;
14    final boolean isGenerated  ;
15    String contentType = null;
16    String typeDef = null;
17    String fieldName = null;
18    String getterName = null;
19
20    public WriteParser(String e,boolean g){elementName=e;isGenerated=g;}
21
22    private void start(DTDDef def,StringBuffer result){
23      contentType = ShowType.sShowType(def);
24      typeDef = !isGenerated?elementName:contentType;
25      fieldName = "v"+elementName;
26      getterName = "getV"+elementName+"()";
27
28      result.append("\n\n  private Parser<"+typeDef+"> "+fieldName+" = null;");
29      result.append("\n  public Parser<"+typeDef+"> "+getterName+"{");
30      result.append("\n    if ("+fieldName+"==null){");
31      result.append("\n      "+fieldName+" = ");
32      if (!isGenerated) {
33        result.append("new Element<"+typeDef+">(\""+typeDef+"\"");
34        result.append("\n     ,");
35        result.append("new Closure<Parser<"+typeDef+">>(){public Parser<"+typeDe
36      }
37
38    }
39
40    private String f(DTDDef def){
41      StringBuffer result=new StringBuffer();
42      start(def,result);
43      result.append(ParserCode.parserCode(def,elementName));
44      if (!isGenerated){
45        result.append("\n     .<"+typeDef+">map(new Function1");
46        result.append("<"+contentType+","+typeDef+">(){");
47        result.append("\n       public "+typeDef+" eval("+contentType+" x){");
48        result.append("\n         return new "+typeDef+"(x);");
49        result.append("\n       }");
50        result.append("\n     })");
51      }
52      end(def,result);
53      return result.toString();
54    }
55
56    private void end(DTDDef def,StringBuffer result){
57      if (!isGenerated){
58        result.append("\n;}}//end of closure\n");
59        result.append(")");
60
61      }
62      result.append(";");
```

```
63      result.append("\n    }");
64      result.append("\n    return "+fieldName+";");
65      result.append("\n  }");
66    }
67
68    private String startend(DTDDef def){
69      StringBuffer result=new StringBuffer();
70      start(def,result);
71      result.append(ParserCode.parserCode(def,elementName));
72      end(def,result);
73      return result.toString();
74    }
75
76    public String visit(DTDPCData x){return f(x);}
77    public String visit(DTDTagName x){return f(x);}
78    public String visit(DTDEmpty x){return f(x);}
79    public String visit(DTDAny x){return null;}
80    public String visit(DTDPlus x){return f(x);}
81    public String visit(DTDStar x){return f(x);}
82    public String visit(DTDQuery x){return f(x);}
83    public String visit(DTDChoice x){return startend(x);}
84    public String visit(DTDSeq x){
85      StringBuffer result = new StringBuffer();
86      start(x,result);
87      result.append(ParserCode.parserCode(x,elementName));
88      result.append(".map(new Function1<"
89                  +ShowType.sShowType(x)+","+elementName+">(){");
90      result.append("\n    public "+elementName);
91      result.append(" eval("+ShowType.sShowType(x) +" p){");
92      result.append("\n      return new "+elementName);
93      unrollPairs(x.seqParts.size(),"p",result);
94      result.append(";");
95      result.append("\n    }");
96      result.append("\n  }");
97      result.append(")");
98      end(x,result);
99      return result.toString();
100   }
101
102   private void unrollPairs(int i,String var,StringBuffer r){
103     String c = var;
104     String result="";
105     for (Integer j :new FromTo(2,i)){
106        result=","+c+".e2"+result;
107        c= c+".e1";
108     }
109     result=c+result;
110     r.append("(");
111     r.append(result);
112     r.append(")");
```

```
113   }
114
115   public static String writeParser
116                        (DTDDef def,String n,boolean isGenerated){
117     return new WriteParser(n,isGenerated).writeParser(def);}
118
119   public  String writeParser(DTDDef def){
120     return ((DTDDefAdt)def).welcome(this);}
121 }
```

C.3.5 Flattening of a DTD definition

FlattenResult.java

```
1  package name.panitz.crempel.util.xml.dtd;
2  import name.panitz.crempel.util.xml.dtd.tree.DTDDef;
3  import name.panitz.crempel.util.Tuple3;
4  import name.panitz.crempel.util.Tuple2;
5  import java.util.List;
6  import java.util.ArrayList;
7
8
9  public class FlattenResult
10    extends Tuple2<DTDDef,List<Tuple3<Boolean,String,DTDDef>>>{
11   public FlattenResult(DTDDef dtd,List<Tuple3<Boolean,String,DTDDef>> es){
12     super(dtd,es);
13   }
14
15   public FlattenResult(DTDDef dtd){
16     super(dtd,new ArrayList<Tuple3<Boolean,String,DTDDef>>() );
17   }
18 }
```

DTDDefFlatten.java

```
1  package name.panitz.crempel.util.xml.dtd;
2  import name.panitz.crempel.util.xml.dtd.IsAtomic.*;
3  import name.panitz.crempel.util.xml.dtd.tree.*;
4  import name.panitz.crempel.util.Tuple3;
5  import name.panitz.crempel.util.Tuple2;
6  import name.panitz.crempel.util.Function1;
7  import java.util.List;
8  import java.util.ArrayList;
9  import java.util.TreeSet;
10 import java.util.Comparator;
11
12 public class DTDDefFlatten extends DTDDefVisitor<FlattenResult>{
13   final String elementName;
14   final boolean isGenerated;
15   private int counter = 0;
16   private String getNextName(){
```

```
17      counter=counter+1;
18      return elementName+"_"+counter;
19    }
20
21    public DTDDefFlatten(boolean g,String n){elementName=n;isGenerated=g;}
22
23    public FlattenResult visit(DTDPCData x){
24      return single(x);}
25    public FlattenResult visit(DTDTagName x){
26      return single(x);}
27    public FlattenResult visit(DTDEmpty x){
28      return single(x);}
29    public FlattenResult visit(DTDAny x){
30      return single(x);}
31    public FlattenResult visit(DTDPlus x){
32      if (IsAtomic.isAtomic(x.dtd)) return single(x);
33      return flattenModified(elementName,x.dtd
34            ,new Function1<DTDDef,DTDDef>(){
35               public DTDDef eval(DTDDef dtd){return new DTDPlus(dtd);}
36             });
37    }
38    public FlattenResult visit(DTDStar x){
39      if (IsAtomic.isAtomic(x.dtd)) return single(x);
40      return
41       flattenModified(elementName,x.dtd
42            ,new Function1<DTDDef,DTDDef>(){
43               public DTDDef eval(DTDDef dtd){return new DTDStar(dtd);}
44             });
45    }
46    public FlattenResult visit(DTDQuery x){
47      if (IsAtomic.isAtomic(x.dtd)) return single(x);
48      return flattenModified(elementName,x.dtd
49            ,new Function1<DTDDef,DTDDef>(){
50               public DTDDef eval(DTDDef dtd){return new DTDQuery(dtd);}
51             });
52    }
53    public FlattenResult visit(DTDSeq x){
54      return flattenPartList(x.seqParts
55            ,new Function1<List<DTDDef>,DTDDef>(){
56               public DTDDef eval(List<DTDDef> dtds){
57                 return new DTDSeq(dtds);}
58             });
59    }
60    public FlattenResult visit(DTDChoice x){
61      return flattenPartList(x.choiceParts
62            ,new Function1<List<DTDDef>,DTDDef>(){
63               public DTDDef eval(List<DTDDef> dtds){
64  //System.out.println("the new choice"+dtds);
65                 return new DTDChoice(dtds);}
66             });
```

```
67    }
68
69   private FlattenResult single(DTDDef x){return new FlattenResult(x);}
70
71   private FlattenResult flattenModified
72           (final String orgElem,DTDDef content
73           ,Function1<DTDDef,DTDDef> constr){
74     List<Tuple3<Boolean,String,DTDDef>> result
75      = new ArrayList<Tuple3<Boolean,String,DTDDef>>();
76     if (needsNewElement(content)){
77 //    System.out.println("owo needs new element: "+content );
78       final String newElemName
79         = ShowType.typeToIdent(ShowType.sShowType(content));
80
81       result.add(new Tuple3<Boolean,String,DTDDef>
82          (true,newElemName,content));
83       return new FlattenResult
84                 (constr.eval(new DTDTagName(newElemName)),result);
85     }
86 //    System.out.println("does not need new element");
87     FlattenResult innerRes  = flatten(content);
88 //    System.out.println(innerRes);
89     return new FlattenResult(constr.eval(innerRes.e1),innerRes.e2);
90   }
91
92   private FlattenResult flattenPartList
93       (List<DTDDef> parts,Function1<List<DTDDef>,DTDDef> constr){
94     final List<Tuple3<Boolean,String,DTDDef>> result
95       = new ArrayList<Tuple3<Boolean,String,DTDDef>>();
96     if (parts.size()==1) {return single(parts.get(0));}
97
98     List<DTDDef> newParts = new ArrayList<DTDDef>();
99     for (DTDDef part:parts){
100      if (IsAtomic.isAtomic(part)) {//System.out.println("atomic part:"+part);
101            newParts.add(part);}
102      else if (needsNewElement(part)){
103        final String newElemName
104          = ShowType.typeToIdent(ShowType.sShowType(part));
105        result.add(new Tuple3<Boolean,String,DTDDef>
106             (true,newElemName,part));
107        newParts.add(new DTDTagName(newElemName));
108      }else{
109        FlattenResult innerRes  = flatten(part);
110        newParts.add(innerRes.e1);
111        result.addAll(innerRes.e2);
112      }
113    }
114    return new FlattenResult(constr.eval(newParts),result);
115  }
116
```

```
117   static private boolean needsNewElement(DTDDef d){
118     return
119      (d instanceof DTDSeq && ((DTDSeq)d).seqParts.size()>1)
120             ||
121      (d instanceof DTDChoice &&((DTDChoice)d).choiceParts.size()>1);
122   }
123
124   static public List<Tuple3<Boolean,String,DTDDef>>
125        flattenDefList(List<Tuple3<Boolean,String,DTDDef>> defs){
126     boolean changed = true;
127     List<Tuple3<Boolean,String,DTDDef>> result = defs;
128     while (changed){
129       Tuple2<Boolean,List<Tuple3<Boolean,String,DTDDef>>> once
130         = flattenDefListOnce(result);
131       changed=once.e1;
132       result=once.e2;
133     }
134
135     TreeSet<Tuple3<Boolean,String,DTDDef>> withoutDups
136       = new TreeSet<Tuple3<Boolean,String,DTDDef>>(
137          new Comparator<Tuple3<Boolean,String,DTDDef>>(){
138            public int compare(Tuple3<Boolean,String,DTDDef> o1
139                               ,Tuple3<Boolean,String,DTDDef> o2){
140               return o1.e2.compareTo(o2.e2);
141            }
142          });
143     withoutDups.addAll(result);
144
145     result = new ArrayList<Tuple3<Boolean,String,DTDDef>> ();
146     result.addAll(withoutDups);
147     return result;
148   }
149
150   private static Tuple2<Boolean,List<Tuple3<Boolean,String,DTDDef>>>
151        flattenDefListOnce(List<Tuple3<Boolean,String,DTDDef>> defs){
152
153     final List<Tuple3<Boolean,String,DTDDef>> result
154      = new ArrayList<Tuple3<Boolean,String,DTDDef>>();
155
156     boolean changed = false;
157
158     for (Tuple3<Boolean,String,DTDDef> def:defs){
159       final FlattenResult singleResult
160         = new DTDDefFlatten(def.e1,def.e2).flatten(def.e3);
161       changed=changed||singleResult.e2.size()>0;
162       result.addAll(singleResult.e2);
163       result.add(
164        new Tuple3<Boolean,String,DTDDef>(
165          singleResult.e2.isEmpty()&&def.e1,def.e2,singleResult.e1));
166     }
```

```
    return new Tuple2<Boolean,List<Tuple3<Boolean,String,DTDDef>>>
                    (changed,result);
  }

  public static FlattenResult flatten(DTDDef def,String n){
   return new DTDDefFlatten(false,n).flatten(def);}

  public  FlattenResult flatten(DTDDef def){
   return ((DTDDefAdt)def).welcome(this);}
}


```

IsAtomic.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;

public class IsAtomic extends DTDDefVisitor<Boolean>{
  public Boolean visit(DTDPCData x){return true;}
  public Boolean visit(DTDTagName x){return true;}
  public Boolean visit(DTDEmpty x){return true;}
  public Boolean visit(DTDAny x){return true;}
  public Boolean visit(DTDPlus x){return isAtomic(x.dtd);}
  public Boolean visit(DTDStar x){return isAtomic(x.dtd);}
  public Boolean visit(DTDQuery x){return isAtomic(x.dtd);}
  public Boolean visit(DTDSeq x){return false;}
  public Boolean visit(DTDChoice x){return false;}

  public static Boolean isAtomic(DTDDef def ){
   return ((DTDDefAdt)def).welcome(new IsAtomic());}
}
```

C.3.6 Main generation class

GenerateClassesForDTD.java

```
package name.panitz.crempel.util.xml.dtd;

import name.panitz.crempel.util.xml.dtd.tree.*;
import java.util.List;
import java.util.ArrayList;
import java.io.*;
import name.panitz.crempel.util.*;

public class GenerateClassesForDTD{
  public static void generateAll
     (String paket,String path,String n,List<Tuple2<String,DTDDef>>dtds){
    List<Tuple3<Boolean,String,DTDDef>> xs
      = new ArrayList<Tuple3<Boolean,String,DTDDef>>();
```

```
    for (Tuple2<String,DTDDef> t:dtds)
      xs.add(new Tuple3<Boolean,String,DTDDef>(false,t.e1,t.e2));

    xs = DTDDefFlatten.flattenDefList(xs);

    System.out.println("vereinfacht und flachgemacht");
    for (Tuple3<Boolean,String,DTDDef> t:xs){
      System.out.println(t.e2);
      System.out.println(t.e3);
      System.out.println("");
    }

    final String parserType = dtds.get(0).e1;
    try{
      Writer out = new FileWriter(path+"/"+n+"Parser"+".java");
      out.write("package "+paket+";\n");

      out.write("import name.panitz.crempel.util.xml.parslib.*;\n");
      out.write("import name.panitz.crempel.util.*;\n");
      out.write("import java.util.*;\n");
      out.write("import org.w3c.dom.Node;\n\n");

      out.write("public class "+n+"Parser ");
      out.write("extends AbstractParser<"+parserType+">{\n");

      out.write("public ParseResult<"+parserType+"> ");
      out.write("parse(List<Node> xs){");
      out.write("  return getV"+parserType+"().parse(xs);}\n\n");

      for (Tuple3<Boolean,String,DTDDef> x :xs)
        out.write(WriteParser.writeParser(x.e3,x.e2,x.e1));

      out.write("}");
      out.close();
    }catch (IOException e){e.printStackTrace();}
    GenerateADT.generateADT(paket,path,xs);
  }
}
```

C.3.7 Main generator class

We provide the main class for generating the parser and algebraic type for a given dtd. Two arguments are passed on the command line. The file name of the DTD file and a package for the generated classes.

MainDTDParse.java

```
package name.panitz.crempel.util.xml.dtd.parser;

import java.io.*;
```

```
4  import java.util.*;
5  import name.panitz.crempel.util.*;
6  import name.panitz.crempel.util.xml.dtd.tree.*;
7  import name.panitz.crempel.util.xml.dtd.*;
8
9  public class MainDTDParse {
10   public static void main(String [] args){
11     try{
12       final String dtdFileName = args[0];
13       final String packageName = args[1].replace('/','.');
14
15       File f = new File(dtdFileName);
16       final String path
17           = f.getParentFile()==null?":f.getParentFile().getPath();
18
19       final DTD parser = new DTD(new FileReader(f));
20
21       final Tuple3<List<Tuple2<String,DTDDef>>,List<String>,String> dtd
22         = parser.dtd();
23
24       for (Tuple2<String,DTDDef> t:dtd.e1){
25         System.out.println(t.e1);
26         System.out.println(t.e2);
27         System.out.println("");
28       }
29
30       GenerateClassesForDTD
31           .generateAll(packageName,path,dtd.e3,dtd.e1);
32    }catch (Exception _){_.printStackTrace();System.out.println(_);}
33   }
34 }
```

```
sep@linux:~/fh/xmlparslib/examples/src/name/panitz/album> java -c
lasspath /home/sep/fh/xmlparslib/examples/classes/:/home/sep/fh/java1.5/exa
mples/classes/:/home/sep/fh/adt/examples/classes/ name.panitz.crempel.util.
xml.dtd.parser.MainDTDParse album.dtd name.panitz.album
package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class Choice_String_author{

  CChoice_String_authorString(String theString);
  CChoice_String_authorauthor(author theauthor);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class Choice_lp_cd_mc{

  CChoice_lp_cd_mclp(lp thelp);
  CChoice_lp_cd_mccd(cd thecd);
  CChoice_lp_cd_mcmc(mc themc);}

package name.panitz.album;
```

```
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class artist{
Con(String pcdata);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class author{
Con(String pcdata);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class cd{
Con(title thetitle,artist theartist,Maybe<recordingyear>
 theMaybe_recordingyear_,List<track> theList_track_,List<note> theList_note_);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class lp{
Con(title thetitle,artist theartist,Maybe<recordingyear>
 theMaybe_recordingyear_,List<track> theList_track_,List<note> theList_note_);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class mc{
Con(title thetitle,artist theartist,Maybe<recordingyear>
 theMaybe_recordingyear_,List<track> theList_track_,List<note> theList_note_);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class musiccollection{
Con(List<Choice_lp_cd_mc> xs);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class note{
Con(List<Choice_String_author> xs);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class recordingyear{
Con(String pcdata);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class timing{
Con(String pcdata);}

package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class title{
Con(String pcdata);}
```

```
package name.panitz.album;
import java.util.List;
import name.panitz.crempel.util.Maybe;
data class track{
Con(title thetitle,Maybe<timing> theMaybe_timing_);}

sep@linux:~/fh/xmlparslib/examples/src/name/panitz/album> ls
CChoice_String_authorString.java    lpVisitor.java
CChoice_String_authorauthor.java    mc.java
CChoice_lp_cd_mccd.java             mcVisitable.java
CChoice_lp_cd_mclp.java             mcVisitor.java
CChoice_lp_cd_mcmc.java             musiccollection.java
Choice_String_author.java           musiccollectionParser.java
Choice_String_authorVisitable.java  musiccollectionVisitable.java
Choice_String_authorVisitor.java    musiccollectionVisitor.java
Choice_lp_cd_mc.java                note.java
Choice_lp_cd_mcVisitable.java       noteVisitable.java
Choice_lp_cd_mcVisitor.java         noteVisitor.java
album.dtd                           recordingyear.java
artist.java                         recordingyearVisitable.java
artistVisitable.java                recordingyearVisitor.java
artistVisitor.java                  timing.java
author.java                         timingVisitable.java
authorVisitable.java                timingVisitor.java
authorVisitor.java                  title.java
cd.java                             titleVisitable.java
cdVisitable.java                    titleVisitor.java
cdVisitor.java                      track.java
lp.java                             trackVisitable.java
lpVisitable.java                    trackVisitor.java
sep@linux:~/fh/xmlparslib/examples/src/name/panitz/album>
```

C.3.8 JaxB

```
sep@linux:~/fh/xmlparslib/> ~/jwsdp-1.3/jaxb/bin/xjc.sh -dtd
  album.dtd -p name.panitz.jaxb.album
parsing a schema...
compiling a schema...
name/panitz/jaxb/album/impl/runtime/GrammarInfo.java
name/panitz/jaxb/album/impl/runtime/AbstractUnmarshallingEventHandlerImpl.java
name/panitz/jaxb/album/impl/runtime/PrefixCallback.java
name/panitz/jaxb/album/impl/runtime/Discarder.java
name/panitz/jaxb/album/impl/runtime/ValidatableObject.java
name/panitz/jaxb/album/impl/runtime/SAXUnmarshallerHandlerImpl.java
name/panitz/jaxb/album/impl/runtime/ContentHandlerAdaptor.java
name/panitz/jaxb/album/impl/runtime/ValidatorImpl.java
name/panitz/jaxb/album/impl/runtime/UnmarshallerImpl.java
name/panitz/jaxb/album/impl/runtime/GrammarInfoFacade.java
name/panitz/jaxb/album/impl/runtime/XMLSerializable.java
name/panitz/jaxb/album/impl/runtime/UnmarshallingEventHandler.java
name/panitz/jaxb/album/impl/runtime/DefaultJAXBContextImpl.java
name/panitz/jaxb/album/impl/runtime/SAXMarshaller.java
name/panitz/jaxb/album/impl/runtime/GrammarInfoImpl.java
name/panitz/jaxb/album/impl/runtime/MSVValidator.java
name/panitz/jaxb/album/impl/runtime/UnmarshallableObject.java
name/panitz/jaxb/album/impl/runtime/SAXUnmarshallerHandler.java
name/panitz/jaxb/album/impl/runtime/ErrorHandlerAdaptor.java
name/panitz/jaxb/album/impl/runtime/NamespaceContext2.java
name/panitz/jaxb/album/impl/runtime/Util.java
name/panitz/jaxb/album/impl/runtime/UnmarshallingEventHandlerAdaptor.java
name/panitz/jaxb/album/impl/runtime/ValidationContext.java
```

```
name/panitz/jaxb/album/impl/runtime/ValidatingUnmarshaller.java
name/panitz/jaxb/album/impl/runtime/MarshallerImpl.java
name/panitz/jaxb/album/impl/runtime/XMLSerializer.java
name/panitz/jaxb/album/impl/runtime/UnmarshallingContext.java
name/panitz/jaxb/album/impl/runtime/NamespaceContextImpl.java
name/panitz/jaxb/album/impl/ArtistImpl.java
name/panitz/jaxb/album/impl/AuthorImpl.java
name/panitz/jaxb/album/impl/CdImpl.java
name/panitz/jaxb/album/impl/JAXBVersion.java
name/panitz/jaxb/album/impl/LpImpl.java
name/panitz/jaxb/album/impl/McImpl.java
name/panitz/jaxb/album/impl/MusiccollectionImpl.java
name/panitz/jaxb/album/impl/NoteImpl.java
name/panitz/jaxb/album/impl/RecordingyearImpl.java
name/panitz/jaxb/album/impl/TimingImpl.java
name/panitz/jaxb/album/impl/TitleImpl.java
name/panitz/jaxb/album/impl/TrackImpl.java
name/panitz/jaxb/album/Artist.java
name/panitz/jaxb/album/Author.java
name/panitz/jaxb/album/Cd.java
name/panitz/jaxb/album/Lp.java
name/panitz/jaxb/album/Mc.java
name/panitz/jaxb/album/Musiccollection.java
name/panitz/jaxb/album/Note.java
name/panitz/jaxb/album/ObjectFactory.java
name/panitz/jaxb/album/Recordingyear.java
name/panitz/jaxb/album/Timing.java
name/panitz/jaxb/album/Title.java
name/panitz/jaxb/album/Track.java
name/panitz/jaxb/album/jaxb.properties
name/panitz/jaxb/album/bgm.ser
sep@linux:~/fh/xmlparslib/examples/src/name/panitz/album>
```

C.4 Example: Generating Java Classes for a DTD

C.5 Example

In this section we give an example of how to develop a program by taking as starting point a definition of the processed data by way of a DTD.

C.5.1 DTD

The DTD is a real life example. It is the document type for some theatre plays. It is taking from a distribution of Shakespeare plays made by Jon Bosak.

play.dtd

```
<!DOCTYPE PLAY SYSTEM "play.dtd" [
<!ELEMENT PLAY
   (TITLE, FM, PERSONAE, SCNDESCR, PLAYSUBT, INDUCT?
   ,PROLOGUE?, ACT+, EPILOGUE?)>
<!ELEMENT TITLE    (#PCDATA)>
<!ELEMENT FM       (P+)>
<!ELEMENT P        (#PCDATA)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
```

```
<!ELEMENT PGROUP   (PERSONA+, GRPDESCR)>
<!ELEMENT PERSONA  (#PCDATA)>
<!ELEMENT GRPDESCR (#PCDATA)>
<!ELEMENT SCNDESCR (#PCDATA)>
<!ELEMENT PLAYSUBT (#PCDATA)>
<!ELEMENT INDUCT
   (TITLE, SUBTITLE*, (SCENE+|(SPEECH|STAGEDIR|SUBHEAD)+))>
<!ELEMENT ACT
   (TITLE, SUBTITLE*, PROLOGUE?, SCENE+, EPILOGUE?)>
<!ELEMENT SCENE
   (TITLE, SUBTITLE*, (SPEECH | STAGEDIR | SUBHEAD)+)>
<!ELEMENT PROLOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT EPILOGUE (TITLE, SUBTITLE*, (STAGEDIR | SPEECH)+)>
<!ELEMENT SPEECH   (SPEAKER+, (LINE | STAGEDIR | SUBHEAD)+)>
<!ELEMENT SPEAKER  (#PCDATA)>
<!ELEMENT LINE     (#PCDATA | STAGEDIR)*>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ELEMENT SUBHEAD  (#PCDATA)>]>
```

C.5.2 Converting to LaTeX

We write an example application, which converts the plays into LaTeXformat. Therefore visitor classes are written for the generated data classes.

PLAYToLaTeXVisitor.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;

import java.util.List;
import de.tfhberlin.crempel.util.Maybe;
import java.io.*;

public  class PLAYToLaTeXVisitor
                 extends PLAYVisitor<StringWriter>{
  StringWriter out=new StringWriter();

  public StringWriter eval(PLAY pl){
    final TITLE title=pl.getTheTITLE();
    final FM fm=pl.getTheFM();
    final PERSONAE personae=pl.getThePERSONAE();
    final SCNDESCR scndesc =pl.getTheSCNDESCR();
    final PLAYSUBT playsubt=pl.getThePLAYSUBT();
    final Maybe<INDUCT> induct = pl.getTheMaybe_INDUCT_();
    final Maybe<PROLOGUE> prologue= pl.getTheMaybe_PROLOGUE_();
    final List<ACT> acts=pl.getTheList_ACT_();
    final Maybe<EPILOGUE> epilogue = pl.getTheMaybe_EPILOGUE_();

```

```
    out.write("\\documentclass[10pt,a4paper]{article}\n");
    out.write("\\usepackage[english]{babel}\n");
    out.write("\\begin{document}\n");
    out.write("\\title{"+title.getPcdata()+"}\n");
    for (ACT act:acts){
      act.visit(new ACTToLaTeXVisitor(out));
    }
    out.write("\\end{document}\n");

    return out;
  }

  public static void main(String [] args) throws Exception{
    final String arg= args[0];
    PLAY play
     = (PLAY)de.tfhberlin.crempel.test.MainParser.pFile
          ("examples.theatre.PLAYParser",arg);
    StringWriter tex = play.visit(new PLAYToLaTeXVisitor());
    Writer out
      = new FileWriter(
                  arg.substring(0,arg.lastIndexOf('.'))+".tex");
    out.write(tex.toString());
    out.close();
  }
}
```

ACTToLaTeXVisitor.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;

import java.util.List;
import de.tfhberlin.crempel.util.Maybe;
import java.io.*;

public class ACTToLaTeXVisitor extends ACTVisitor<StringWriter>{
  StringWriter out;
  public ACTToLaTeXVisitor(StringWriter o){out=o;}
  public StringWriter eval(ACT act){
    final TITLE title = act.getTheTITLE();;
    final List<SUBTITLE> subtitles = act.getTheList_SUBTITLE_();
    final Maybe<PROLOGUE> prologue=act.getTheMaybe_PROLOGUE_();
    final List<SCENE> scenes=act.getTheList_SCENE_();
    final Maybe<EPILOGUE> epilogues=act.getTheMaybe_EPILOGUE_();

    out.write("\\section*{"+title.getPcdata()+"}\n");
    for (SCENE scene:scenes){
      scene.visit(new SceneToLaTeXVisitor(out));
    }
```

```
    return out;
  }}
```

SceneToLaTeXVisitor.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;

import java.util.List;
import de.tfhberlin.crempel.util.Maybe;
import java.io.*;

public  class SceneToLaTeXVisitor extends SCENEVisitor<StringWriter>{
  StringWriter out;
  public SceneToLaTeXVisitor(StringWriter o){out=o;}

  public StringWriter eval(SCENE sc){
    final TITLE title=sc.getTheTITLE();
    final List<SUBTITLE> subtitles=sc.getTheList_SUBTITLE_();
    final List<Choice_SPEECH_STAGEDIR_SUBHEAD> spDirSubheads
     =sc.getTheList_Choice_SPEECH_STAGEDIR_SUBHEAD_();
    out.write("\\subsection*{"+title.getPcdata()+"}\n");
    for (Choice_SPEECH_STAGEDIR_SUBHEAD spDirSubhead:spDirSubheads)
      spDirSubhead.visit(new SpeechStageDirSubheadToLaTeX(out));
    return out;
  }}
```

SpeechStageDirSubheadToLaTeX.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;
import java.io.*;
import java.util.List;
import de.tfhberlin.crempel.util.Maybe;

public  class SpeechStageDirSubheadToLaTeX
 extends Choice_SPEECH_STAGEDIR_SUBHEADVisitor<StringWriter>{

  StringWriter out;
  public SpeechStageDirSubheadToLaTeX(StringWriter o){out=o;}

  public StringWriter eval
                    (CChoice_SPEECH_STAGEDIR_SUBHEADSPEECH sp){
    SPEECH speech = sp.getTheSPEECH();
    out.write("\\begin{description}\n");
    out.write("\\item[");
    for (SPEAKER speaker:speech.getTheList_SPEAKER_()){
      out.write(" "+speaker.getPcdata());
    }
```

```
    out.write("]");
    List<Choice_LINE_STAGEDIR_SUBHEAD> lineStagedirSubheads
      = speech.getTheList_Choice_LINE_STAGEDIR_SUBHEAD_();

    for (Choice_LINE_STAGEDIR_SUBHEAD lineStagedirSubhead
        :lineStagedirSubheads){
      lineStagedirSubhead.visit(new LineStageDirSubheadToLaTeX(out));
    }


    out.write("\\end{description}\n");
    return out;
  }
  public StringWriter eval
                (CChoice_SPEECH_STAGEDIR_SUBHEADSTAGEDIR st){
    STAGEDIR stageDir = st.getTheSTAGEDIR(); ;
    return out;
  }
  public StringWriter eval
                (CChoice_SPEECH_STAGEDIR_SUBHEADSUBHEAD subH){
    SUBHEAD subhead = subH.getTheSUBHEAD(); ;
    return out;
  }
  public StringWriter eval(Choice_SPEECH_STAGEDIR_SUBHEAD xs){
    throw new RuntimeException(xs.getClass().toString());
  }}

```

LineStageDirSubheadToLaTeX.java

```
package examples.theatre;

import de.tfhberlin.crempel.util.Visitor;
import java.io.*;
import java.util.List;
import de.tfhberlin.crempel.util.Maybe;

public  class LineStageDirSubheadToLaTeX
 extends Choice_LINE_STAGEDIR_SUBHEADVisitor<StringWriter>{

  StringWriter out;
  public LineStageDirSubheadToLaTeX(StringWriter o){out=o;}

  public StringWriter eval(CChoice_LINE_STAGEDIR_SUBHEADLINE sp){
    LINE line = sp.getTheLINE();
    for (Choice_String_STAGEDIR stringStageDir:line.getXs()){
      if (stringStageDir instanceof CChoice_String_STAGEDIRString){
        out.write
         (((CChoice_String_STAGEDIRString) stringStageDir)
                                    .getTheString());
```

```
        out.write("\\\\");
      }
    }
    return out;
  }
  public StringWriter eval
            (CChoice_LINE_STAGEDIR_SUBHEADSTAGEDIR st){
    STAGEDIR stageDir = st.getTheSTAGEDIR(); ;
    return out;
  }
  public StringWriter eval
            (CChoice_LINE_STAGEDIR_SUBHEADSUBHEAD subH){
    SUBHEAD subhead = subH.getTheSUBHEAD(); ;
    return out;
  }
}
```

C.6 Conclusion

We applied techniques as known from functional programming to Java. Algebraic types and parser combinators enables us to write a complicated library tool, which generates classes and parsers for a given `DTD`. Java's generic types provide some good means to express these concepts in a static type safe manner.

With *JaxB (JavaTM Architecture for XML Binding)* Sun provides a library and tool which addresses the same problem: for a given schema generate classes and a parser. However, the resulting classes and parsers in *JaxB* do not provide the visitor pattern for the resulting class and do not use the concepts as known from functional programming.

C.7 Javacc input file for DTD parser

In this section you can find a simple not complete parser for DTDs as input grammar for the parser generator *javacc*.

dtd.jj

```
options {STATIC=false;}

PARSER_BEGIN(DTD)
package name.panitz.crempel.util.xml.dtd.parser;

import name.panitz.crempel.util.*;
import name.panitz.crempel.util.xml.dtd.tree.*;
import java.util.List;
import java.util.ArrayList;
import java.io.FileReader;

public class DTD {}
PARSER_END(DTD)
```

```
14
15 TOKEN :
16 {<ELEMENTDEC: "<!ELEMENT">
17 |<DOCTYPE:    "<!DOCTYPE">
18 |<ATTLIST:    "<!ATTLIST">
19 |<REQUIRED:   "#REQUIRED">
20 |<IMPLIED:    "#IMPLIED">
21 |<EMPTY: "EMPTY">
22 |<PCDATA: "#PCDATA">
23 |<CDATA: "CDATA">
24 |<ANY: "ANY">
25 |<SYSTEM: "SYSTEM">
26 |<PUBLIC: "PUBLIC">
27 |<GR: ">">
28 |<QMARK: "?">
29 |<PLUS: "+">
30 |<STAR: "*">
31 |<#NameStartCar: [":","A"-"Z","_","a"-"z"
32                 ,"\u00C0"-"\u00D6"
33                 ,"\u00D8"-"\u00F6"
34                 ,"\u00F8"-"\u02FF"
35                 ,"\u0370"-"\u037D"
36                 ,"\u037F"-"\u1FFF"
37                 ,"\u200C"-"\u200D"
38                 ,"\u2070"-"\u218F"
39                 ,"\u2C00"-"\u2FEF"
40                 ,"\u3001"-"\uD7FF"
41                 ,"\uF900"-"\uFDCF"
42                 ,"\uFDF0"-"\uFFFD"]>
43 //                  ,"\u10000"-"\uEFFFF"]>
44 |<#InnerNameChar: ["-", ".","0"-"9", "\u00B7"
45                  ,"\u0300"-"\u036F"
46                  ,"\u203F"-"\u2040"]>
47 |<#NameChar: <InnerNameChar>|<NameStartCar>>
48 |<Name: <NameStartCar> (<NameChar>)* >
49
50 |<#ALPHA:       ["a"-"z","A"-"Z","_",".”]>
51 |<#NUM:                ["0"-"9"]>
52 |<#ALPHANUM:    <ALPHA> | <NUM>>
53 |<EQ: "=">
54 |<BAR: "|">
55 |<LPAR: "(">
56 |<RPAR: ")">
57 |<LBRACKET: "{">
58 |<RBRACKET: "}">
59 |<LSQBRACKET: "[">
60 |<RSQBRACKET: "]">
61 |<LE: "<">
62 |<SEMICOLON: ";">
63 |<COMMA: ",">
```

```
64  |<QUOTE: "\"">
65  |<SINGLEQUOTE: "'">
66  }
67
68  SKIP :
69  {< ["\u0020","\t","\r","\n"] >}
70
71  //"
72
73  void externalID():
74  {}
75  {<SYSTEM> systemLiteral()
76   |<PUBLIC> systemLiteral() systemLiteral()
77  }
78
79  void systemLiteral():{}
80  {<QUOTE><Name><QUOTE>|<SINGLEQUOTE><Name><SINGLEQUOTE>
81  }
82
83  Tuple3<List<Tuple2<String,DTDDef>>,List<String>,String> dtd():
84  {
85    Token nameToken;
86    List<Tuple2<String,DTDDef>> els = new ArrayList<Tuple2<String,DTDDef>>();
87    List<String> atts = new ArrayList<String>();
88    Tuple2<String,DTDDef> el;
89  }
90  {<DOCTYPE> nameToken=<Name> externalID() <LSQBRACKET>
91
92     (el=elementdecl(){ els.add(el);}
93     |AttlistDecl())*
94   <RSQBRACKET><GR>
95    {return new Tuple3<List<Tuple2<String,DTDDef>>,List<String>,String>
96                  (els,atts,nameToken.toString());}
97  }
98
99  Tuple2<String,DTDDef> elementdecl():
100 {Token nameToken;
101  DTDDef content;}
102 {<ELEMENTDEC> nameToken=<Name>  content=contentspec()  <GR>
103   {return new Tuple2<String,DTDDef>(nameToken.toString(),content);}
104 }
105
106
107 DTDDef contentspec():
108 {DTDDef result;}
109 {  <EMPTY>{result=new DTDEmpty();}
110  | <ANY>{result=new DTDAny();}
111  | (<LPAR>(result=Mixed()
112           |result=children()))
113  { return result;}
```

```
114 }
115
116 DTDDef children():
117 { List<DTDDef> cps;
118   DTDDef cp;
119   Modifier mod = Modifier.none;
120   boolean wasChoice = true;
121 }
122 {cp=cp()
123   (cps=choice()| cps=seq(){wasChoice=false;})
124    {cps.add(0,cp);}
125    <RPAR>(<QMARK>{mod=Modifier.query;}
126      |<STAR>{mod=Modifier.star;}
127      |<PLUS>{mod=Modifier.plus;})?
128   {DTDDef result=wasChoice?ParserAux.createDTDChoice(cps)
129                           :ParserAux.createDTDSeq(cps);
130    return mod.mkDTDDef(result);
131   }
132 }
133
134 List<DTDDef> choice():
135 { DTDDef cp;
136   List<DTDDef> result = new ArrayList<DTDDef>();}
137 {(<BAR> cp=cp() {result.add(cp);} )+
138   {return result;}
139 }
140
141 DTDDef cp():
142 { Token nameToken=null;
143   List<DTDDef> cps=new ArrayList<DTDDef>();
144   DTDDef cp;
145   Modifier mod=Modifier.none;
146   boolean wasChoice = true;
147   boolean wasTagName = false;
148 }
149 {((nameToken = <Name>{wasTagName=true;})
150  |(<LPAR>cp=cp()
151          (cps=choice()|cps=seq(){wasChoice=false;})
152     {cps.add(0,cp);}
153    <RPAR>
154   )
155  )
156      (<QMARK>{mod=Modifier.query;}
157      |<STAR>{mod=Modifier.star;}
158      |<PLUS>{mod=Modifier.plus;})?
159 {
160  DTDDef result;
161  if (wasTagName) result=new DTDTagName(nameToken.toString());
162  else result=wasChoice?ParserAux.createDTDChoice(cps)
163                       :ParserAux.createDTDSeq(cps);
```

```
164  return mod.mkDTDDef(result);
165 }
166 }
167
168 List<DTDDef> seq():
169 {  List<DTDDef> result = new ArrayList<DTDDef>();
170    DTDDef cp;
171 }
172 {(<COMMA> cp=cp(){result.add(cp);} )*
173   {return result;}
174 }
175
176 DTDDef Mixed():
177 {Token nameToken;
178  List<DTDDef> xs = new ArrayList<DTDDef>();
179  Modifier mod=Modifier.none;
180 }
181 {  <PCDATA> {xs.add(new DTDPCData());}
182   ((<BAR> nameToken=<Name>
183     {xs.add(new DTDTagName(nameToken.toString()));}
184    )* <RPAR>(<STAR>{mod=Modifier.star;})?)
185 {return mod.mkDTDDef(ParserAux.createDTDChoice(xs));}
186
187 }
188
189 void AttlistDecl():
190 {Token nameToken;}
191 {<ATTLIST> nameToken=<Name> (AttDef() )*
192 {}}
193
194 void AttDef():
195 {Token nameToken;
196  boolean isRequired;}
197 {nameToken=<Name> AttType() isRequired=DefaultDecl()
198   {}
199 }
200
201 void AttType():
202 {}
203 { StringType()}//|TokenizedType()|EnumeratedType()}
204
205 void StringType():
206 {}
207 {<CDATA>}
208
209
210 boolean DefaultDecl():
211 { boolean isRequired=false;}
212
213 {(<REQUIRED>{isRequired=true;} | <IMPLIED>)
```

```
//| (('#FIXED' S)? AttValue)
{return isRequired;}
}
```

ParserAux.java

```
package name.panitz.crempel.util.xml.dtd.parser;

import name.panitz.crempel.util.xml.dtd.tree.*;
import java.util.List;

public class ParserAux{
  static public DTDDef createDTDSeq(List<DTDDef> xs){
    return new DTDSeq(xs);
  }
  static public DTDDef createDTDChoice(List<DTDDef> xs){
    return new DTDChoice(xs);
  }
}
```

Modifier.java

```
package name.panitz.crempel.util.xml.dtd.tree;

public enum Modifier { none, star, plus, query;

  public String toString(){
    switch(this) {
          case star:   return "*";
          case plus:   return "+";
          case query:  return "?";
    }
    return "";
  }

  public DTDDef mkDTDDef(DTDDef dtd){
    switch(this) {
      case star:   return new DTDStar(dtd);
      case plus:   return new DTDPlus(dtd);
      case query:  return new DTDQuery(dtd);
    }
    return dtd;
  }
}
```

Anhang D

Hilfklasssen zu Swing

SwingWorker.java

```
package name.panitz.swing.threadTest;
import javax.swing.SwingUtilities;

/**
 * This is the 3rd version of SwingWorker (also known as
 * SwingWorker 3), an abstract class that you subclass to
 * perform GUI-related work in a dedicated thread.  For
 * instructions on using this class, see:
 *
 * http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html
 *
 * Note that the API changed slightly in the 3rd version:
 * You must now invoke start() on the SwingWorker after
 * creating it.
 */
public abstract class SwingWorker {
    private Object value;  // see getValue(), setValue()
    private Thread thread;

    /**
     * Class to maintain reference to current worker thread
     * under separate synchronization control.
     */
    private static class ThreadVar {
        private Thread thread;
        ThreadVar(Thread t) { thread = t; }
        synchronized Thread get() { return thread; }
        synchronized void clear() { thread = null; }
    }

    private ThreadVar threadVar;

    /**
```

```
     * Get the value produced by the worker thread, or null if it
     * hasn't been constructed yet.
     */
    protected synchronized Object getValue() {
        return value;
    }

    /**
     * Set the value produced by worker thread
     */
    private synchronized void setValue(Object x) {
        value = x;
    }

    /**
     * Compute the value to be returned by the <code>get</code> method.
     */
    public abstract Object construct();

    /**
     * Called on the event dispatching thread (not on the worker thread)
     * after the <code>construct</code> method has returned.
     */
    public void finished() {
    }

    /**
     * A new method that interrupts the worker thread.  Call this method
     * to force the worker to stop what it's doing.
     */
    public void interrupt() {
        Thread t = threadVar.get();
        if (t != null) {
            t.interrupt();
        }
        threadVar.clear();
    }

    /**
     * Return the value created by the <code>construct</code> method.
     * Returns null if either the constructing thread or the current
     * thread was interrupted before a value was produced.
     *
     * @return the value created by the <code>construct</code> method
     */
    public Object get() {
        while (true) {
            Thread t = threadVar.get();
            if (t == null) {
                return getValue();
```

```
            }
            try {
                t.join();
            }
            catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // propagate
                return null;
            }
        }
    }


    /**
     * Start a thread that will call the <code>construct</code> method
     * and then exit.
     */
    public SwingWorker() {
        final Runnable doFinished = new Runnable() {
           public void run() { finished(); }
        };

        Runnable doConstruct = new Runnable() {
            public void run() {
                try {
                    setValue(construct());
                }
                finally {
                    threadVar.clear();
                }

                SwingUtilities.invokeLater(doFinished);
            }
        };

        Thread t = new Thread(doConstruct);
        threadVar = new ThreadVar(t);
    }

    /**
     * Start the worker thread.
     */
    public void start() {
        Thread t = threadVar.get();
        if (t != null) {
            t.start();
        }
    }
}
```

Anhang E

Fragen und Antworten als Lernhilfe

- **Ist Java eine kompilierte oder interpretierte Sprache?**

  Im Prinzip beides: Ein Compiler übersetzt den Quelltext in .class-Dateien, die Code für eine virtuelle Maschine enthalten. Ein Interpreter ist in der Lage diesen Code auszuführen.

- **Wie ist ein Javaprogramm strukturiert?**

  Pro Datei eine Klasse (oder Schnittstelle oder Aufzählung), die den Namen der Datei trägt.

- **Gibt es einen Präprozessor?**

  Nein.

- **Gibt es einen Linker?**

  Nein, aber die virtuelle Maschine hat einen Klassenlader, der den Code der einzelnen Klassen bei Bedarf lädt.

- **Welche Namenskonventionen gibt es?**

  Klassennamen beginnen mit einem Großbuchstaben, globale Konstanten sind komplett in Großbuchstaben geschrieben. Pakete, Parameter, Variablen beginnen mit einem Kleinbuchstaben. Bei Bezeichnern aus mehreren Wörter fängt das nächste Wort immer mit einem Großbuchstaben an.

- **Ist Java statisch getypt?**

  Ja, der Compiler überprüft, ob das Programm korrekt getypt ist.

- **Gibt es auch dynamische Typüberprüfung zur Laufzeit?**

  Ja, zum einem verifiziert die virtuelle Maschine noch einmal den Code, zum anderen bewirkt eine Typzusicherung (cast) eine Laufzeittypüberprüfung.

- **Kann ich sicherstellen, daß die Typzusicherung während der Laufzeit nicht fehlschlägt.**

  Ja, indem vor der Typzusicherung der Test mit `instanceof` gemacht wird.

- **Wie schreibt man Unterprogramme oder Funktionen?**

  Immer innerhalb einer Klasse. Sie werden als Methoden bezeichnet.

- **Was sind die Eigenschaften einer Klasse?**

  Zum einen die Felder (auch als Attribute) bezeichnet, in denen Referenzen auf Objekte abgelegt werden können. Zum anderen die Methoden und Konstruktoren.

- **Was ist das `this`-Objekt?**

  Damit wird das Objekt bezeichnet, in dem sich eine Eigenschaft befindet.

- **Was ist Vererbung?**

  Jede Klasse hat genau eine Oberklasse, die in der `extends`-Klausel angegeben wird. Objekte können alle Eigenschaften die in ihrer Oberklasse zur Verfügung stehen benutzen.

- **Was ist wenn ich keine Oberklasse angebe?**

  Dann ist automatisch die Klasse `Object` die Oberklasse.

- **Was sind Konstruktoren?**

  Konstruktoren sind der Code einer Klasse, der beim Erzeugen von neuen Objekten ausgeführt wird und in der Regel die Felder des Objektes initialisiert.

- **Wie werden Konstruktoren definiert?**

  Ähnlich wie Methoden. Sie haben den Namen der Klasse und keinen Rückgabetyp.

- **Wie werden Konstuktoren aufgerufen?**

  Durch das Schlüsselwort `new` gefolgt von dem Klassennamen.

- **Hat jede Klasse einen Konstruktor?**

  Ja.

- **Und wenn ich keinen Konstruktor für meine Klasse schreibe?**

  Dann fügt Java einen leeren Konstruktor ohne Parameter ein.

- **Was ist der Unterschied zwischen statischen und nicht statischen Eigenschaften einer Klasse?**

  Eine statische Eigenschaft ist nicht an spezielle Objekte gebunden. Sie hat daher auch kein `this`-Objekt. Eine statische Eigenschaft existiert nur einmal für alle Objekte einer Klasse. Nicht-statische Methoden werden auch als Objektmethoden bezeichnet.

- **Gibt es auch statische Konstruktoren?**

  Ja, pro Klasse genau einen, der keine Parameter hat.

- **Kann ich den statischen Konstruktor auch selbst definieren?**

  Ja, mit dem Schlüssenwort `static` gefolgt von in geschweiften Klammern eingeschlossenen Code.

- **Was bedeutet Überschreiben von Methoden?**

  Methoden, die es in der Oberklasse bereits gibt in einer Unterklasse neu zu definieren.

- **Kann ich auch Konstruktoren überschreiben?**

  Nein.

- **Was bezeichnet man als Polymorphie?**

  Wenn eine Methode in verschiedenen Unterklassen einer Klasse überschrieben wird.

- **Kann ich in einer überschreibenen Methode, die überschriebene Methode aufrufen?**

  Ja, indem man das Schlüsselwort `super` benutzt und mit einem Punkt abgetrennt den eigentlichen Methodenaufruf folgen läßt.

- **Kann ich Konstruktoren der Oberklasse aufrufen.**

  Ja, aber nur im Konstruktor als erste Anweisung. Hier muß sogar der Aufruf eines Konstruktors der Oberklasse stehen. Dieser Aufruf wird durch das Schlüsselwort `super` gefolgt von der Parameterliste gemacht.

- **Was ist, wenn ich im Konstruktor keinen Aufruf an einen Konstruktor der Oberklasse schreiben.**

  Dann generiert Java den Aufruf eines Konstruktors der Oberklasse ohne Parameter als erste Anweisung in den Konstruktor. Sollte so ein parameterloser Konstruktor nicht existieren, dann gibt es allerdings einen Folgefehler.

- **Was ist späte Bindung (late binding)?**

  Beim Aufruf von Objektmethoden wird immer der Methodencode ausgeführt, der in der Klasse implementiert wurde, von der das Objekt, auf dem diese Methode aufgerufen wurde, erzeugt wurde. Es wird also immer die überschreibende Version einer Methode benutzt.

- **Was sind virtuelle Methoden und gibt es sie in Java?**

  In C++ werden Methoden, für die späte Bindung gewünscht ist als virtuell markiert. In Java funktioniert die späte Bindung für alle Objektmethoden.

- **Funktioniert späte Bindung auch für Felder?**

  Nein.

- **Funktioniert späte Bindung auch für statische Methoden?**

  Nein.

- **Funktioniert späte Bindung auch für Konstruktoren?**

  Nein.

- **Was sind überladene Methoden?**

  Methoden gleichen Namens in einer Klasse, die sich in Typ/Anzahl der Parameter unterscheiden.

- **Können auch Konstruktoren überladen werden?**

  Ja.

- **Gibt es das Prinzip von später Bindung auch für die verschiedenen überladenen Versionen einer Methode?**

  Nein! Die Auflösung, welche der überladenen Versionen einer Methode ausgeführt wird, wird bereits statisch vom Compiler vorgenommen und nicht dynamisch während der laufzeit.

- **Was sind abstrakte Klassen?**

  Klassen, die als `abstract` deklariert sind. Nur abstrakte Klassen können abstrakte Eigenschaften enthalten.

- **Und was sind abstrakte Eigenschaften?**

  Das sind Methoden, die keinen Methodenrumpf haben.

- **Können abstrakte Klassen Konstruktoren haben?**

  Ja, allerdings können von abstrakten Klassen keine Objekte mit `new` erzeugt werden.

- **Wie kann ich dann Objekte einer abstrakten Klassen erzeugen und wozu haben die dann Konstruktoren?**

  Indem Objekte einer nicht abstrakten Unterklasse mit `new` erzeugt werden. Im Konstrukor der Unterklasse wird ein Konstruktor der abstrakten Oberklasse aufgerufen.

- **Kann eine Klasse mehrere abstrakte Oberklassen haben?**

  Nein, auch abstrakte Klassen sind Klassen und es gilt die Regel: jede Klasse hat genau eine Oberklasse.

- **Kann ich in einer abstrakten Klasse abstrakte Methoden aufrufen?**

  Ja, im Rumpf einer nicht-abstrakten Methode können bereits abstrakte Klassen aufgerufen werden.

- **Was sind Schnittstellen?**

  Man kann Schnittstellen als abstrakte Klassen ansehen, in denen jeder Methode abstrakt ist.

- **Warum gibt es dann Schnittstellen?**

  Schnittstellen gelten nicht als Klassen. Eine Klasse kann nur eine Oberklasse haben, aber zusätzlich mehrere Schnittstellen implementieren.

- **Wie wird deklariert, daß eine Klasse eine Schnitstelle implementiert?**

  Durch die `implements`-Klausel in der Klassendeklaration. In ihr können mehrere Komma getrennte Schnittstellen angegeben werden.

- **Kann eine Schnittstelle eine Oberklasse haben?**

  Nein.

- **Kann eine Schnittstelle weitere Oberschnittstellen haben.**

  Ja, diese werden in der `extends`-Klausel angegeben.

- **Muß eine Klasse die eine Schnittstelle implementiert alle Methoden der Schnittstelle implementieren?**

  Im Prinzip ja, jedoch nicht, wenn die Klasse selbst abstrakt ist.

- **Haben Schnittstellen auch Objektfelder?**

  Nein! Nur Methoden und statische Felder.

- **Gibt es Zeiger?**

  Explizit nicht. Implizit wird außer bei Werten primitiver Typen immer mit Referenzen auf Objekte gearbeitet.

- **Sind alle Daten in Java Objekte?**

  Nein, es gibt 8 primitive Typen, deren Daten keine Objekte sind. Es gibt aber zu jeden dieser 8 primitiven Typen eine Klasse, die die Daten entsprechend als Objekt speichern.

- **Welches sind die primitiven Typen?**

  byte, short, int, long, double, float, char, boolean

- **Sind Strings Objekte?**

  Ja, die Klasse String ist eine Klasse wie Du und ich.

- **Sind Reihungen (arrays) Objekte?**

  Ja, und sie haben sogar ein Attribut, das ihre Länge speichert.

- **Was sind generische Typen?**

  Klassen oder Schnittstellen, in denen ein oder mehrere Typen variabel gehalten sind.

- **Wie erzeugt man Objekte einer generischen Klasse?**

  Indem beim Konstruktoraufruf in spitzen Klammern konkrete Typen für die Typvariablen angegeben werden.

- **Was passiert, wenn ich für generische Typen die spitzen Klammern bei der Benutzung weglasse?**

  Dann gibt der Compiler eine Warnung und nimmt den allgemeinsten Typ für die Typvariablen an. Meistens ist das der Typ `Object`.

- **Was sind generische Methoden?**

  Methoden in denen ein oder mehrere Parametertypen variabel gehalten sind.

- **Muß ich bei der Benutzung von generischen Methoden die konkreten Typen für die Typvariablen mit angeben?**

  Anders als beim Konstruktoraufruf für generische Klassen nicht.

- **Was passiert, wenn ich sie nicht angebe?**

  Dann inferriert der Typchecker die konkreten Typen, sofern das möglich ist.

- **Was sind typische Beispiel für generische Typen?**

  Alle Sammlungsklassen und Abbildungsklassen im Paket `java.util`, zB die Klassen `ArrayList`,`LinkedList`, `HashSet`, `Vector` oder die Schnittstellen `List`, `Map`, `Set`.

- **Wo wir gerade dabei sind. Was sollte ich bei der Klasse `Vector` beachten?**

  In 90% der Fälle ist ein Objekt der Klasse `ArrayList` einem Objekt der Klasse `Vector` vorzuziehen. `Vector` ist eine sehr alte Klasse. Ihre Objekte sind synchronisiert, die anderen Sammlungsklassen nicht, es lassen sich von den anderen Sammlungsklassen allerdings synchronisierte Kopien machen. Am besten `Vector` gar nie benutzen.

- **Was ist automatisches Boxing und Unboxing?**

  Die Konvertierung von Daten primitiver Typen in Objekte der korrespondierenden Klassen und umgekehrt wird automatisch vorgenommen.

- **Welche zusammengesetzten Befehle gibt es?**

  if, while, for, switch

- **Was hat es mit der besonderen for-Schleife auf sich?**

  Es handelt sich um eine sogenannte for-each-Schleife. Syntaktisch trennt hier ein Doppelpunkt die lokale Schleifenvariable das Sammlungsobjekt.

- **Für welche Objekte kann die for-each-Schleife benutzt werden?**

  Für alle Objekte, die die Schnittstelle `Iterable` implementieren und für Reihungen.

- **Bedeutet das, ich kann die for-each Schleife auch für meine Klassen benutzen?**

  Ja, genau, man muß nur die Schnittstelle `Iterable` hierzu implementieren.

- **Können Operatoren überladen oder neu definiert werden?**

  Nein.

- **Was sind Ausnahme?**

  Objekte von Unterklassen der Klasse `Exception`. Zusätzlich gibt es die Klasse `Error` und die gemeinsame Klasse`Throwable`.

- **Wozu sind Ausnahmen gut?**

  Um in bestimmten Situationen mit einem `throw` Befehl den Programmfluß abzubrechen um eine besondere Ausnahmesituation zu signalisieren.

- **Was für Objekte dürfen in eine `throw` geworfen werden?**

  Nur Objekte einer Unterklasse von `Throwable`.

- **Wie werden geworfene Ausnahme behandelt?**

  Indem man sie innerhalb eines `try-catch`-Konstruktes wieder abfängt.

- **Kann man auf unterschiedliche Ausnahmen in einem `catch` unterschiedlich reagieren?**

  Ja, einfach mehrere `catch` untereinanderschreiben. Das als erstes zutreffende `catch` ist dann aktiv.

- **Was ist das mit dem `finally` beim `try-catch`?**

  Hier kann Code angegeben werden, der immer ausgeführt werden soll, egal ob und welche Ausnahme aufgetreten ist. Dieses ist sinnvoll um eventuell externe Verbindungen und ähnliches immer sauber zu schließen.

- **Darf man beliebig Ausnahmen werfen?**

  Ausnahmen, die nicht in einem `catch` abgefangen werden müssen in der `throws`-Klausel einer Methode deklariert werden.

- **Alle?**

  Nein, Ausnahmeobjekte von `RuntimeException` dürfen auch geworfen werden, wenn sie nicht in der `throws`-Klausel einer Methode stehen.

- **Wozu sind Pakete da?**

  Unter anderen damit man nicht Milliarden von Klassen in einem Ordner hat und dabei die Übersicht verliert.

- **Ist mir egal. Pakete sind mir zu umständlich.**

  Das ist aber dumm, wenn man Klassen gleichen Namens aus verschiedenen Bibliotheken benutzen will. Die kann man nur unterscheiden, wenn sie in verschiedenen Paketen sind.

- **OK, ich sehs ein. Wie nenne ich mein Paket?**

  Am besten die eigene Webdomain rückwärsts nehmen also für uns an der FH: `de.fhwiesbaden.informatik.panitz.meineApplikation`

- **Da fehlt doch der Bindestrich.**

  Der ist kein gültiges Zeichen innerhalb eines Bezeichners in Java (es ist ja der Minusoperator). Bindestriche in Webadressen sind eine recht deutsche Krankheit.

- **Muß ich imports machen um Klassen aus anderen Paketen zu benutzen?**

  Nein. Das macht die Sache nur bewuemer, weil sonst der Klassenname immer und überall komplett mit seinem Paket angegeben werden muß, auch beim Konstruktoraufruf oder z.B. `instanceof`.

- **Machen Import-Anweisungen das Programm langsamer oder größer?**

  Nein! Sie haben insbesondere nichts mit includes in C gemein. Sie entsprechen eher dem `using namespace` aus C++.

- **Da gibt es doch auch noch `public`, `private` und `protected`.**

  Jaja, die Sichtbarkeiten. Hinzu kommt, wenn man keine Sichtbarkeit hinschreibt. `public` heißt von überall aufrufbar, `protected` heißt in Unterklassen und gleichem Paket aufrufbar, `package` das ist wenn man nichts hinschreibt, heißt nur im gleichen Paket sichtbar, und `private` nur in der Klasse.

- **Und dann war da noch das `final`.**

  Das hat zwei Bedeutungen: bei Variablen und Feldern, daß sie nur einmal einen Wert zugewiesen bekommen, bei Klassen, daß keine Unterklassen von der Klasse definiert werden

- **Ich will GUIs programmieren.**

  Wunderbar, da gibt es zwei Bibliotheken: Swing und AWT.

- **Na toll, warum zwei?**

  Historisch.

- **Und welche sol ich jetzt benutzen?**

  Swing!

- **Also kann ich alles über AWT vergessen, insbesondere das Paket `java.awt`?**

  Nein. Swing benutzt große Teile aus AWT. Insbesondere hat jede Swingkomponente eine Oberklasse aus AWT. Aber auch die Ereignisbehandlungsklassen werden vollständig aus AWT benutzt.

- **Und woran erkenne ich jetzt, wann ich Komponente aus AWT bnutzen muß, obwohl ich in Swing programmiere?**

  Alle graphischen AWT-Komponenten haben ein Swing Pendant, das mit dem dem Buchstaben `J` beginnt, z.B. `javax.swing.JButton` im Gegensatz zu `java.awt.Button`.

- **Sonst noch paar Tipps, zum benutzen von Swing?**

  Keine eigenen Threads schreiben! Nicht die Methode `paint` sondern `paintComponent` überschreiben.

- **Toll, wenn ich keine Threads schreiben soll, wie mache ich dann z.B. zeitgesteuerte Ereignisse in Swing?**

  Da gibt es die Klasse `javax.swing.Timer` für.

- **Ich will XML verarbeiten.**

  Kein Problem, alles vorhanden, soll es das DOM-Api oder SAX sein?

- **Weiß nicht? Was ist der Unterschied?**

  In DOM wird eine Baumstruktur für ein XML-Dokument im Speicher aufgebaut. Mit der kann man alles machen, was man mit einem Baum machen kann.

- **Super. Wozu ist dann SAX?**

  Oft braucht man nie die ganze Baumstruktur, sondern muß das Dokument nur einmal von vorne nach hinten durcharbeiten. Dabei hilft SAX

- **Ist dann effizienter schätze ich?**

  Ganz genau, insbesondere wird viel weniger Speicher benötigt.

- **Na ich weiß nicht, ob ich jetzt genügend weiß für die Klausur?**

  Das weiß ich auch nicht, über alles haben wir hier nicht gesprochen. Ich denke aber es reicht an Wissen aus.

Anhang F

Wörterliste

In dieser Mitschrift habe ich mich bemüht, soweit existent oder naheliegend, deutsche Ausdrücke für die vielen in der Informatik auftretenden englischen Fachbegriffe zu benutzen. Dieses ist nicht aus einem nationalen Chauvinismus heraus, sondern für eine flüssige Lesbarkeit des Textes geschehen. Ein mit sehr vielen englischen Wörtern durchsetzter Text ist schwerer zu lesen, insbesondere auch für Menschen, deren Muttersprache nicht deutsch ist.

Es folgen hier Tabellen der verwendeten deutschen Ausdrücke mit ihrer englischen Entsprechung.

Deutsch	**Englisch**
Abbildung	map
Attribut	modifier
Aufrufkeller	stack trace
Ausdruck	expression
Ausnahme	exception
Auswertung	evaluation
Bedingung	condition
Befehl	statement
Behälter	container
Blubbersortierung	bubble sort
Datei	file
Deklaration	declaration
Eigenschaft	feature
Erreichbarkeit	visibility
Fabrikmethode	factory method
faul	lazy
Feld	field
geschützt	protected
Hauruckverfahren	bruteforce
Implementierung	implementation
Interpreter	interpreter
Keller	stack
Klasse	class
Kommandozeile	command line

Menge	set
Methode	method
Modellierung	design
nebenläufig	concurrent
Oberklasse	superclass
Ordner	folder
Paket	package
privat	private
Puffer	buffer
Reihung	array
Rumpf	body
Rückgabetyp	return type
Sammlung	collection
Schleife	loop
Schnittstelle	interface
Sichtbarkeit	visibility
Steuerfaden	thread
strikt	strict
Strom	stream
Typ	type
Typzusicherung	type cast
Unterklasse	subclass
Verklemmung	dead lock
Verzeichnis	directory
vollqualifizierter Name	fully qualified name
Zeichen	character
Zeichenkette	string
Zuweisung	assignment
öffentlich	public
Übersetzer	compiler

Englisch	**Deutsch**
array	Reihung
assignment	Zuweisung
body	Rumpf
bruteforce	Hauruckverfahren
bubble sort	Blubbersortierung
buffer	Puffer
character	Zeichen
class	Klasse
collection	Sammlung
command line	Kommandozeile
compiler	Übersetzer
concurrent	nebenläufig
condition	Bedingung
container	Behälter
dead lock	Verklemmung

declaration	Deklaration
design	Modellierung
directory	Verzeichnis
evaluation	Auswertung
exception	Ausnahme
expression	Ausdruck
factory method	Fabrikmethode
feature	Eigenschaft
field	Feld
file	Datei
folder	Ordner
fully qualified name	vollqualifizierter Name
implementation	Implementierung
interface	Schnittstelle
interpreter	Interpreter
lazy	faul
loop	Schleife
map	Abbildung
method	Methode
modifier	Attribut
package	Paket
private	privat
protected	geschützt
public	öffentlich
return type	Rückgabetyp
set	Menge
stack	Keller
stack trace	Aufrufkeller
statement	Befehl
stream	Strom
strict	strikt
string	Zeichenkette
subclass	Unterklasse
superclass	Oberklasse
thread	Steuerfaden
type	Typ
type cast	Typzusicherung
visibility	Sichtbarkeit
visibility	Erreichbarkeit

Anhang G

Gesammelte Aufgaben

Aufgabe 1 Schreiben Sie das obige Programm mit einen Texteditor ihrer Wahl. Speichern Sie es als `FirstProgram.java` ab. Übersetzen Sie es mit dem Java-Übersetzer `javac`. Es entsteht eine Datei `FirstProgram.class`. Führen Sie das Programm mit dem Javainterpreter `java` aus. Führen Sie dieses sowohl einmal auf Linux als auch einmal unter Windows durch.

Aufgabe 2 Schreiben sie ein Programm, das ein Objekt der Klasse `Minimal` erzeugt und auf dem Bildschirm ausgibt. Hierzu ersetzen sie einfach im Programm `Answer` die `42` durch den Ausdruck `new Minimal()`

Aufgabe 3 Schreiben Sie für die vier Karteikarten in der Modellierung eines Bibliotheksystems entsprechende Klassen mit den entsprechenden Feldern.

Aufgabe 4 Schreiben Sie Klassen, die die Objekte des Bibliotheksystems repräsentieren können:

- Personen mit Namen, Vornamen, Straße, Ort und Postleitzahl.
- Bücher mit Titel und Autor.
- Datum mit Tag, Monat und Jahr.
- Buchausleihe mit Ausleiher, Buch und Datum.

Hinweis: der Typ, der ganze Zahlen in Java bezeichnet, heißt `int`.

a) Schreiben Sie geeignete Konstruktoren für diese Klassen.

b) Schreiben Sie für jede dieser Klassen eine Methode `public String toString()` mit dem Ergebnistyp. Das Ergebnis soll eine gute textuelle Beschreibung des Objektes sein.[1]

c) Schreiben Sie eine Hauptmethode in einer Klasse `Main`, in der Sie Objekte für jede der obigen Klassen erzeugen und die Ergebnisse der `toString`-Methode auf den Bildschirm ausgeben.

[1] Sie brauchen noch nicht zu verstehen, warum vor dem Rückgabetyp noch ein Attribut `public` steht.

Aufgabe 5 Suchen Sie auf Ihrer lokalen Javainstallation oder im Netz auf den Seiten von Sun (`http://www.javasoft.com`) nach der Dokumentation der Standardklassen von Java. Suchen Sie die Dokumentation der Klasse `String`. Testen Sie einige der für die Klasse `String` definierten Methoden.

Aufgabe 6 Ergänzen Sie jetzt die Klasse `Person` aus der letzten Aufgabe um ein statisches Feld `letzerVorname` mit einer Zeichenkette, die angeben soll, welchen Vornamen das zuletzt erzeugte Objekt vom Typ `Person` hatte. Hierzu müssen Sie im Konstruktor der Klasse Person dafür sorgen, daß nach der Zuweisung der Objektfelder auch noch das Feld `letzerVorname` verändert wird. Testen Sie in einer Testklasse, daß sich tatsächlich nach jeder Erzeugung einer neuen Person dieses Feld verändert hat.

Aufgabe 7 In dieser Aufgabe sollen Sie eine Gui-Klasse benutzen und ihr eine eigene Anwendungslogik übergeben.

Gegeben seien die folgenden Javaklassen, wobei Sie die Klasse `Dialogue` nicht zu analysieren oder zu verstehen brauchen:

ButtonLogic.java

```
1 class ButtonLogic {
2   String getDescription(){
3     return "in Großbuchstaben umwandeln";
4   }
5   String eval(String x){return x.toUpperCase();}
6 }
```

Dialogue.java

```
1 import javax.swing.*;
2 import java.awt.event.*;
3 import java.awt.*;
4 class Dialogue extends JFrame{
5
6   final ButtonLogic logic;
7
8   final JButton button;
9   final JTextField inputField  = new JTextField(20) ;
10  final JTextField outputField = new JTextField(20) ;
11  final JPanel p = new JPanel();
12
13  Dialogue(ButtonLogic l){
14    logic = l;
15    button=new JButton(logic.getDescription());
16    button.addActionListener
17     (new ActionListener(){
18        public void actionPerformed(ActionEvent _){
19          outputField.setText
20             (logic.eval(inputField.getText().trim()));
21        }
22      });
23    p.setLayout(new BorderLayout());
24    p.add(inputField,BorderLayout.NORTH);
```

```
    p.add(button,BorderLayout.CENTER);
    p.add(outputField,BorderLayout.SOUTH);
    getContentPane().add(p);
    pack();
    setVisible(true);
  }
}
```

TestDialogue.java

```
class TestDialogue {
  public static void main(String [] _){
    new Dialogue(new ButtonLogic());
  }
}
```

a) Übersetzen Sie die drei Klassen und starten Sie das Programm.

b) Schreiben Sie eine Unterklasse der Klasse `ButtonLogic`. Sie sollen dabei die Methoden `getDescription` und `eval` so überschreiben, daß der Eingabestring in Kleinbuchstaben umgewandelt wird. Schreiben Sie eine Hauptmethode, in der Sie ein Objekt der Klasse `Dialogue` mit einem Objekt Ihrer Unterklasse von `ButtonLogic` erzeugen.

c) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie römische Zahlen in arabische Zahlen umwandeln können. Testen Sie Ihr Programm.

d) Schreiben Sie jetzt eine Unterklasse der Klasse `ButtonLogic`, so daß Sie im Zusammenspiel mit der Guiklasse `Dialogue` ein Programm erhalten, in dem Sie arabische Zahlen in römische Zahlen umwandeln können. Testen Sie Ihr Programm.

e) Schreiben Sie jetzt ein Guiprogramm, daß eine Zahl aus ihrer Darstellung zur Basis 10 in eine Darstellung zur Basis 2 umwandelt. Testen Sie.

Aufgabe 8 Starten Sie folgendes Javaprogramm:

TestInteger.java

```
class TestInteger {
  public static void main(String [] _){
    System.out.println(2147483647+1);
    System.out.println(-2147483648-1);
  }
}
```

Erklären Sie die Ausgabe.

Aufgabe 9 Ergänzen Sie ihre Klasse `Ausleihe` um eine Methode `void verlaengereEinenMonat()`, die den Rückgabetermin des Buches um einen Monat erhöht.

Aufgabe 10 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 11 Schreiben Sie mit den bisher vorgestellten Konzepten ein Programm, das unendlich oft das Wort `Hallo` auf den Bildschirm ausgibt. Was beobachten Sie, wenn sie das Programm lange laufen lassen?

Aufgabe 12 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 13 Modellieren und schreiben Sie eine Klasse, die ein Bankkonto darstellt. Auf das Bankkonto sollen Einzahlungen und Auszahlungen vorgenommen werden können. Es gibt einen maximalen Kreditrahmen. Das Konto soll also nicht beliebig viel in die Miese gehen können. Schließlich muß es eine Möglichkeit geben, Zinsen zu berechnen und dem Konto gutzuschreiben.

Aufgabe 14 Schreiben Sie jetzt die Methode zur Berechnung der Fakultät, indem Sie eine Iteration und nicht eine Rekursion benutzen.

Aufgabe 15 Schreiben Sie eine Methode
`static String darstellungZurBasis(int x,int b)`, die als Parameter eine Zahl x und eine zweite Zahl b erhält. Sie dürfen annehmen, daß $x > 0$ und $1 < b < 11$. Das Ergebnis soll eine Zeichenkette vom Typ `String` sein, in der die Zahl x zur Basis b dargestellt ist. Testen Sie ihre Methode mit unterschiedlichen Basen.

Hinweis: Der zweistellige Operator `%` berechnet den ganzzahligen Rest einer Division. Bei einem geschickten Umgang mit den Operatoren `%, /` und `+` und einer `while`-Schleife kommen Sie mit sechs Zeilen im Rumpf der Methode aus.

Aufgabe 16 Schreiben Sie eine Methode `static int readIntBase10(String str)`. Diese Methode soll einen String, der nur aus Ziffern besteht, in die von ihm repräsentierte Zahl umwandeln. Benutzen sie hierzu die Methode `charAt` der `String`-Klasse, die es erlaubt, einzelne Buchstaben einer Zeichenkette zu selektieren.

Aufgabe 17 Für die Lösung dieser Aufgabe gibt es 3 Punkte, die auf die Klausur angerechnet werden. Voraussetzung hierzu ist, daß die Lösung mir in der Übung gezeigt und erklärt werden kann.

In dieser Aufgabe sollen Sie eine Klasse für römische Zahlen entwickeln.

a) Schreiben Sie eine Klasse `Roman`. Diese Klasse soll eine natürliche Zahl darstellen.

b) Schreiben Sie für Ihre Klasse `Roman` einen Konstruktor, der ein Stringobjekt als Parameter hat. Dieser Stringparameter soll eine römische Zahl darstellen. Der Konstruktor soll diese Zahl lesen und in einem Feld des Typs `int` abspeichern.

c) Implementieren Sie die Methode `public String toString()` für Ihre Klasse `Roman`, die die intern gespeicherte Zahl als römische Zahl dargestellt zurückibt.

d) Fügen Sie ihrer Klasse `Roman` die folgenden Methoden für arithmetische Rechnungen hinzu.

- `Roman add(Roman other)`
- `Roman sub(Roman other)`
- `Roman mul(Roman other)`
- `Roman div(Roman other)`

e) Testen Sie Ihre Klasse `Roman`.

Aufgabe 18 (3 Punkte)

In dieser Aufgabe soll ein Spielbrett für das Spiel Vier gewinnt implementiert werden. Laden Sie hierzu die Datei
vier.zip (`http://www.panitz.name/prog1/load/vier.zip`)

a) Schreiben Sie eine Klasse `VierImplementierung`, die die Schnittstelle `VierLogik` entsprechend der Dokumentation implementiert.

b) Schreiben Sie folgende Hauptmethode und starten Sie diese. Sie sollten jetzt in der Lage sein, über die Eingabekonsole Vier gewinnt zu spielen.

```
1 public static void main(String[] args) {
2   new VierKonsole().spiel(new VierImplementierung());
3 }
```

Suchen Sie sich einen Spielpartner und versuchen Sie, gegen ihn zu gewinnen.

Aufgabe 19 In dieser Aufgabe sollen Sie ein Programm schreiben, das nach und nach die Primzahlen ausgibt.

a) Schreiben sie eine Unterklasse `From` der Klasse `FromTo`, die von einem Startwert an in Einerschritten ganze Zahlen zurückgibt und für die `hasNext` immer wahr ist.

b) Schreiben Sie eine Klasse `Sieb`, die die Schnittstelle `IntIterator` implementiert. Die Klasse soll zwei Felder haben: eine ganze Zahl und ein weiteres Objekt, das die Schnittstelle `IntIterator` implementiert. Der Konstruktor habe die Signatur:
`public Sieb(int i,IntIterator it)`
Die Methode `next` soll das nächste Element des inneren Iterators `it` zurückgeben, das nicht durch die Zahl `i` teilbar ist.

Beispiel: `PrintIntIteratorElements.print(new Sieb(3,new From(1)))` gibt alle nicht durch 3 teilbaren natürlichen Zahlen auf dem Bildschirm aus.

c) Übersetzen Sie ihren Code mit folgender Klasse:

```
package name.panitz.iterator;
class PrimIterator implements IntIterator{
  private IntIterator sieb = new From(2);
  public boolean hasNext(){return sieb.hasNext();}
  public int next(){
    int result = sieb.next();
    sieb= new Sieb(result,sieb);
    return result;
  }
}
```

Lassen Sie sich alle Werte von `PrimIterator` mit der Methode `PrintIntIteratorElements.print` ausgeben.

Aufgabe 20 Nehmen Sie beide der in diesem Kapitel entwickelten Umsetzungen von Listen und fügen Sie ihrer Listenklassen folgende Methoden hinzu. Führen Sie Tests für diese Methoden durch.

a) `a last()`: gibt das letzte Element der Liste aus.

b) `List<a> concat(List<a> other)` bzw.: `Li<a> concat(Li<a> other)`: erzeugt eine neue Liste, die erst die Elemente der `this`-Liste und dann der `other`-Liste hat, es sollen also zwei Listen aneinander gehängt werden.

c) `a elementAt(int i)`: gibt das Element an einer bestimmten Indexstelle der Liste zurück. Spezifikation:

$$\begin{aligned} elementAt(Cons(x, xs), 1) &= x \\ elementAt(Cons(x, xs), n+1) &= elementAt(xs, n) \end{aligned}$$

Aufgabe 21 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 22 Diese Aufgabe soll mir helfen, Listen für Ihre Leistungsbewertung zu erzeugen.

a) Implementieren Sie für Ihre Listenklasse eine Methode `String toHtmlTable()`, die für Listen Html-Code für eine Tabelle erzeugt, z.B:

```
<table>
  <tr>erstes Listenelement</tr>
  <tr>zweites Listenelement</tr>
  <tr>drittes Listenelement</tr>
</table>
```

b) Nehmen Sie die Klasse `Student`, die Felder für Namen, Vornamen und Matrikelnummer hat. Implementieren Sie für diese Klasse eine Methode `String toTableRow()`, die für Studenten eine Zeile einer Html-Tabelle erzeugt:

```
Student s1 = new Student("Müller","Hans",167857);
System.out.println(s1.toTableRow());
```

soll folgende Ausgabe ergeben:

```
<td>Müller</td><td>Hans</td><td>167857</td>
```

Ändern Sie die Methode `toString` so, daß sie dasselbe Ergebnis wie die neue Methode `toTableRow` hat.

c) Legen Sie eine Liste von Studenten an, sortieren Sie diese mit Hilfe der Methode `sortBy`nach Nachnamen und Vornamen und erzeugen Sie eine Html-Seite, die die sortierte Liste anzeigt.

Sie können zum Testen die folgende Klasse benutzen:

HtmlView.java

```
package name.panitz.data.list;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.basic.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

public class HtmlView extends JPanel {

  //example invocation
  public static void main(String s[]) {
    HtmlView view = new HtmlView();
    view.run();
    view.setText("<h1>hallo</h1>");
  }

  JFrame frame;
  JTextPane ausgabe = new JTextPane();

  public HtmlView() {
    ausgabe.setEditorKit(new HTMLEditorKit());
    add(ausgabe);
  }

  void setText(String htmlString){
    ausgabe.setText(htmlString);
    frame.pack();
    ausgabe.repaint();
  }
```

```
31
32    void run(){
33      frame = new JFrame("HtmlView");
34      frame.getContentPane().add(this);
35      frame.pack();
36      frame.setVisible(true);
37    }
38  }
```

Aufgabe 23 Erzeugen Sie ein Baumobjekt, das einen Stammbaum Ihrer Familie darstellt. Die Knoten und Blätter sind mit Namen markiert. Die Wurzel ist mit Ihren Namen markiert. Kinderknoten sind mit den leiblichen Eltern der Person eines Knotens markiert.

Aufgabe 24 Testen Sie die in diesem Abschnitt entwickelten Methoden auf Bäumen mit dem in der letzten Aufgabe erzeugten Baumobjekt.

Aufgabe 25 Schreiben Sie die folgenden weiteren Methoden für die Klasse `Tree`. Schreiben Sie Tests für diese Methoden.

a) `List leaves()`: erzeugt eine Liste der Markierungen an den Blättern des Baumes.

b) `String show()`: erzeugt eine textuelle Darstellung des Baumes. Jeder Knoten soll eine eigene Zeile haben. Kinderknoten sollen gegenüber einem Elternknoten eingerückt sein.

Beispiel:

```
william
  charles
    elizabeth
    phillip
  diana
    spencer
```

Tipp: Benutzen Sie eine Hilfsmethode
`String showAux(String prefix)`,
die den String `prefix` vor den erzeugten Zeichenketten anhängt.

c) `boolean contains(a o)`: ist wahr, wenn eine Knotenmarkierung gleich dem Objekt `o` ist.

d) `int maxDepth()`: gibt die Länge des längsten Pfades von der Wurzel zu einem Blatt an.

e) `List<a> getPath(a o)`: gibt die Markierungen auf dem Pfad von der Wurzel zu dem Knoten, der mit dem Objekt `o` markiert ist, zurück. Ergebnis ist eine leere Liste, wenn ein Objekt gleich `o` nicht existiert.

f) `java.util.Iterator<a> iterator()`: erzeugt ein Iterator über alle Knotenmarkierungen des Baumes.
Tipp: Benutzen Sie bei der Umsetzung die Methode `flatten`.

g) `public boolean equals(Object other)`: soll genau dann wahr sein, wenn `other` ein Baum ist, der die gleiche Struktur und die gleichen Markierungen hat.

h) `<b>boolean sameStructure(Tree<a> other)`: soll genau dann wahr sein, wenn der Baum `other` die gleiche Struktur hat. Die Markierungen der Knoten können hingegen unterschiedlich sein.

Aufgabe 26 Schreiben Sie eine Methode `List<Tree<a>> siblings()`, die die Geschwister eines Knotens als Liste ausgibt. In dieser Liste der Geschwister soll der Knoten selbst nicht auftauchen.

Aufgabe 27 Schreiben Sie eine modifizierende Methode
`void deleteChildNode(int n)`,
die den n-ten Kindknoten löscht, und stattdessen die Kinder des n-ten Kindknotens als neue Kinder mit einhängt.
Beispiel:

```
        a                 wird durch                          a
       / \                deleteChildNode(1)                 /|\
      /   \               zu:                               / | \
     b     c                                               /  |  \
    / \   / \                                             /   |   \
   d   e  f  g                                           b    f    g
         /|\                                            /|   /|\
        h i j                                          d e  h i j
```

Aufgabe 28 Überschreiben Sie die Methoden `addLeaf` und `deleteChildNode` in der Klasse `BinTree`, so daß sie nur eine Ausnahme werfen, wenn die Durchführung der Modifikation dazu führen würde, daß das Objekt, auf dem die Methode angewendet wird, anschließend kein Binärbaum mehr wäre.

Aufgabe 29 Schreiben Sie analog zur Methode `flatten` in der Klasse `Tree` eine Methode `List<a> postorder()`, die die Knoten eines Baumes in Postordnung linearisiert.

Aufgabe 30 Rechnen Sie auf dem Papier ein Beispiel für die Arbeitsweise der Methode `military()`.

Aufgabe 31 Zeichnen Sie schrittweise die Baumstruktur, die im Programm `TestSearchTree` aufgebaut wird.

Aufgabe 32 Erzeugen Sie ein Objekt des typs `SearchTree` und fügen Sie nacheinander die folgenden Elemente ein:
`"anna","berta","carla","dieter","erwin","florian","gustav"`
Lassen Sie anschließend die maximale Tiefe des Baumes ausgeben.

Aufgabe 33

a) Schreiben sie entsprechend die Methode:
static public SearchTree rotateLeft(SearchTree t)

b) Schreiben Sie in der Klasse `BinTree` die entsprechende modifizierende Methode `rotateLeft()`

c) Testen Sie die beiden Rotierungsmethoden. Testen Sie, ob die Tiefe der Kinder sich verändert und ob die Inordnung gleich bleibt. Testen Sie insbesondere auch einen Fall, in dem die `NullPointerException` abgefangen wird.

Aufgabe 34 Übersetzen und starten Sie die Klasse `Finalize` und beobachten Sie, wie und wann Java Objekte aus dem Speicher löscht.

Aufgabe 35 Schreiben Sie ein Programm `FileConvert` zum Konvertieren von Textdateien in eine andere Kodierung. Dem Programm sollen über die Kommandozeilenparameter der Name der Eingabedatei, der Name der Ausgabedatei und der Name der benutzten Codierung übergeben werden. Ein möglicher Aufruf wäre also:

```
linux:~/>java FileConvert test.txt konvertiertTest.txt utf-8
```

Lassen Sie eine deutsche Textdatei mit Umlauten in eine Datei mit der Codierung utf-8 konvertieren. Betrachten Sie die Ergebnisdatei. Was stellen Sie fest?

Aufgabe 36 Erweitern Sie die Klasse `Li`, so daß Sie Ihre Listenobjekte in Dateien schreiben und wieder aus Dateien lesen können. Testen Sie ihre Implementierung.

Aufgabe 37 Studieren Sie die Dokumentation von `java.io.RandomAccessFile` und schreiben Sie einige Testbeispiele zur Benutzung dieser Klasse.

Aufgabe 38 Schreiben Sie ein XML Dokument, daß nach den Regeln der obigen DTD gebildet wird.

Aufgabe 39 Die folgenden Dokumente sind kein wohlgeformetes XML. Begründen Sie, wo der Fehler liegt, und wie dieser Fehler behoben werden kann.

a)
```
<a>to be </a><b> or not to be</b>
```

Lösung

Kein top-level Element, das das ganze Dokument umschließt.

b)
```
<person geburtsjahr=1767>Ferdinand Carulli</person>
```

Lösung

Attributwerte müssen in Anführungszeichen stehen.

c)
```
<line>lebt wohl<br/>
<b>Gott weiß, wann wir uns wiedersehen</line>
```

Lösung

`<b>` wird nicht geschlossen.

d)
```
<kockrezept><!--habe ich aus dem Netz
  <name>Saltimbocca</name>
  <zubereitung>Zutaten aufeinanderlegen
               und braten.</zubereitung>
</kockrezept>
```

Lösung

Kommentar wird nicht geschlossen.

e)
```
<cd>
  <artist>dead&alive</artist>
  <title>you spin me round</title></cd>
```

Lösung

Das Zeichen & muß als character entity `&` geschrieben werden.

Aufgabe 40 Gegeben sind das folgende XML-Dokument:

TheaterAutoren.xml
```
<?xml version="1.0" encoding="iso-8859-1" ?>
<autoren>
<autor>
  <person>
    <nachname>Shakespeare</nachname>
    <vorname>William</vorname>
  </person>
  <werke>
    <opus>Hamlet</opus>
    <opus>Macbeth</opus>
    <opus>King Lear</opus>
  </werke>
</autor>
<autor>
  <person>
    <nachname>Kane</nachname>
```

```
    <vorname>Sarah</vorname>
  </person>
  <werke>
    <opus>Gesäubert</opus>
    <opus>Psychose 4.48</opus>
    <opus>Gier</opus>
  </werke>
</autor>
</autoren>
```

a) Schreiben Sie eine DTD, das die Struktur dieses Dokuments beschreibt.

Lösung

AutorenType.dtd

```
<!DOCTYPE autoren SYSTEM "AutorenType.dtd" [
<!ELEMENT autoren (autor+)>
<!ELEMENT autor (person,werke)>
<!ELEMENT werke (opus*)>
<!ELEMENT opus (#PCDATA)>
<!ELEMENT person (nachname,vorname)>
<!ELEMENT nachname (#PCDATA)>
<!ELEMENT vorname (#PCDATA)>]>
```

b) Entwerfen Sie Java Schnittstellen, die Objekte ihrer DTD beschreiben.

Lösung

Autoren.java

```
package name.panitz.xml.exercise;
import java.util.List;
public interface Autoren {
  List<Autor> getAutorList();
}
```

Autor.java

```
package name.panitz.xml.exercise;
public interface Autor {
  Person getPerson();
  Werke  getWerke();
}
```

Person.java

```
package name.panitz.xml.exercise;
public interface Person {
  Nachname getNachname();
  Vorname  getVorname();
}
```

Werke.java

```
package name.panitz.xml.exercise;
import java.util.List;
public interface Werke {
  List<Opus> getOpusList();
}
```

HasJustTextChild.java

```
package name.panitz.xml.exercise;
public interface HasJustTextChild {
  String getText();
}
```

Opus.java

```
package name.panitz.xml.exercise;
public interface Opus extends HasJustTextChild {}
```

Nachname.java

```
package name.panitz.xml.exercise;
public interface Nachname extends HasJustTextChild {}
```

Vorname.java

```
package name.panitz.xml.exercise;
public interface Vorname extends HasJustTextChild {}
```

c) Schreiben Sie ein XSLT-Skript, das obige Dokument in eine Html Liste der Werke ohne Autorangabe transformiert.

Lösung

WerkeListe.xsl

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
     <html><head><title>Werke</title></head>
       <body><xsl:apply-templates /></body>
     </html>
  </xsl:template>

  <xsl:template match="autoren">
     <ul><xsl:apply-templates select="autor/werke/opus"/></ul>
  </xsl:template>

  <xsl:template match="opus">
     <li><xsl:apply-templates/></li>
  </xsl:template>
</xsl:stylesheet>
```

d) Schreiben Sie eine Javamethode
`List<String> getWerkListe(Autoren autoren);`,
die auf den Objekten Ihrer Schnittstelle für `Autoren` eine Liste von Werknamen erzeugt.

Lösung

GetWerke.java

```
package name.panitz.xml.exercise;
import java.util.List;
import java.util.ArrayList;

public class GetWerke{
  public List<String> getWerkListe(Autoren autoren){
    List<String> result = new ArrayList<String>();
    for (Autor a:autoren.getAutorList()){
      for (Opus opus: a.getWerke().getOpusList())
        result.add(opus.getText());
    }
    return result;
  }
}
```

Aufgabe 41 Gegeben sei folgendes XML-Dokument:

Foo.xml

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<x1><x2><x5>5</x5><x6>6</x6></x2><x3><x7>7</x7></x3>
<x4><x8/><x9><x10><x11></x11></x10></x9></x4></x1>
```

a) Zeichnen Sie dieses Dokument als Baum.

b) Welche Dokumente selektieren, die folgenden XPath-Ausdrücke ausgehend von der Wurzel dieses Dokuments

- `//x5/x6`
- `/descendant-or-self::x5/..`
- //x5/ancestor::*
- /x1/x2/following-sibling::*
- /descendant-or-self::text()/parent::node()

Aufgabe 42 Schreiben Sie eine Methode
`List<Node> getLeaves(Node n);`,
die für einen DOM Knoten, die Liste aller seiner Blätter zurückgibt.

Lösung

GetLeaves.java

```
package name.panitz.xml.exercise;
import java.util.List;
import java.util.ArrayList;
import org.w3c.dom.*;

public class GetLeaves{
```

```
  public List<Node> getLeaves(Node n){
    List<Node> result = new ArrayList<Node>();
    NodeList ns = n.getChildNodes();
    if (ns.getLength()==0){
      result.add(n); return result;
    }
    for (int i=0;i<ns.getLength();i++){
      result.addAll(getLeaves(ns.item(i)));
    }
    return result;
  }
}
```

Aufgabe 43 Die Methode `actionPerformed` der Schnittstelle `ActionListener` bekommt ein Objekt der Klasse `ActionEvent` übergeben. In dieser gibt es eine Methode `getWhen`, die den Zeitpunkt, zu dem das Ereignis aufgetreten ist als eine Zahl kodiert zurückgibt. Mit dieser Zahl können Sie die Klasse `java.util.Date` instanziieren, um ein Objekt zu bekommen, das Datum und Uhrzeit enthält.

Schreiben Sie jetzt eine Unterklasse von `JTB`, so daß beim Drücken des Knopfes, jetzt die aktuelle Uhrzeit im Textfeld erscheint.

Aufgabe 44 Schreiben Sie eine kleine Guianwendung mit einer Textfläche und drei Knöpfen, die einen erweiterten Zähler darstellt:

- einen Knopf zum Erhöhen eines Zählers.
- einen Knopf zum Verringern des Zählers um 1.
- einen Knopf zum Zurücksetzen des Zählers auf 0.

Aufgabe 45 Schreiben Sie Ihr Programm eines Zählers mit drei Knöpfen jetzt so, daß sie innere Klassen benutzen.

Aufgabe 46 Schreiben Sie eine GUI-Komponente `StrichKreis`, die mit geraden Linien einen Kreis entsprechend untenstehender Abbildung malt. Der Radius des Kreises soll dabei dem Konstruktor der Klasse `StrichKreis` als `int`-Wert übergeben werden.

Testen Sie Ihre Klasse mit folgender Hauptmethode:

```
public static void main(String [] args) {
  StrichKreis k = new StrichKreis(120);
  JFrame f = new JFrame();
  JPanel p = new JPanel();
  p.add(new StrichKreis(120));
  p.add(new StrichKreis(10));
  p.add(new StrichKreis(60));
  p.add(new StrichKreis(50));
```

```
9     p.add(new StrichKreis(70));
10    f.getContentPane().add(p);
11
12    f.pack();
13    f.setVisible(true);
14  }
```

Hinweis: In der Klasse `java.lang.Math` finden Sie Methoden trigeometrischer Funktionen. Insbesondere `toRadians` zum Umrechnen von Gardwinkel in Bogenmaß sowie `sin` und `cos`.

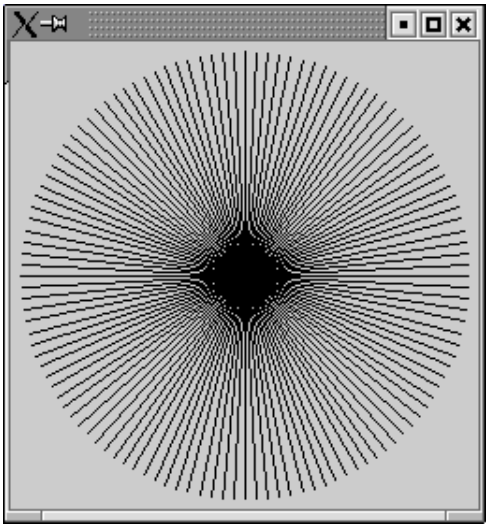


Abbildung G.1: Kreis mit Durchmesserlinien.

Aufgabe 47 Erweitern Sie die Strichkreisklasse aus der letzten Aufgabe, so daß jeder Strich in einer anderen Farbe gezeichnet wird.

Aufgabe 48 Implementieren Sie eine Analoguhr. Das Ziffernblatt dieser Uhr können Sie dabei mit den Methoden auf einem `Graphics`-Objekt zeichnen, oder Sie können versuchen eine Bilddatei zu laden und das Ziffernblatt als Bild bereitstellen. Als Anregung für ein Ziffernblatt können Sie sich das Bild des Ziffernblatts der handgefertigten Sekundenuhr (`http://www.panitz.name/prog2/examples/images/wiebking.jpg`) des Uhrmachers G.Wiebking herunterladen.

Aufgabe 49 Schreiben Sie eine Animation, in der zwei Kreise sich bewegen. Die Kreise sollen an den Rändern der Spielfläche abprallen und sie sollen ihre Richtung ändern, wenn sie sich berühren.

Aufgabe 50 Schreiben Sie eine Animation, deren Verhalten über Mausklicks beeinflussen können.

Aufgabe 51 Erweitern Sie die obige Grammatik so, daß sie mit ihr auch geklammerte arithmetische Ausdrücke ableiten können. Hierfür gibt es zwei neue Terminalsymbolde: `(` und `)`.

Schreiben Sie eine Ableitung für den Ausdruck: `1+(2*20)+1`

Aufgabe 52 Betrachten Sie die einfache Grammatik für arithmetische Ausdrücke aus dem letzen Abschnitt. Zeichnen Sie einen Syntaxbaum für den Audruck `1+1+2*20`.

Aufgabe 53 **(2 Punkte)** In dieser Aufgabe sollen Sie den Parser für arithmetische Ausdrücke um geklammerte Ausdrücke erweitern. Hierzu sei die Grammatik für die Regel *multExpr* wie folgt geändert:

multExpr ::= *atomExpr multOp multExpr*|*atomExpr*
atomExpr ::= `(`*addExpr*`)`|*zahl*

Hierzu sei die entsprechende zusätzliche Tokenaufzählung für die zwei Klammersymbole gegeben:

Parentheses.java

```
package name.panitz.parser;
public enum Parentheses implements Token{lpar,rpar;}
```

Zusätzlich gegeben sei die die folgende Funktion:

DoParentheses.java

```
package name.panitz.parser;

public class DoParentheses
  implements
    Function<Pair<Parentheses,Pair<Integer,Parentheses>>,Integer>{
  public Integer
            apply(Pair<Parentheses,Pair<Integer,Parentheses>> x){
    return x.getE2().getE1();
  }
}
```

a) Schreiben Sie eine Klasse `AtomExprEval`, die `Parser<Integer>` implementiert und der Regel für *atomExpr* entspricht.

b) Ändern Sie die Klasse *MultExprEval*, so daß sie der geänderten Grammatikregel entspricht.

c) Erweitern Sie die Klasse `ArithTokenizer`, so daß er auch die beiden Klammersymbole erkennt.

Aufgabe 54 Fügen Sie der Datei `sms.jjt` folgende Zeilen am Anfang ein:

```
options {
   MULTI=true;
   STATIC=false;
}
```

Generieren Sie mit `jjtree` und `javac` den Parser neu. Was hat sich geändert?

Klassenverzeichnis

Abbildung, 7-11
About, 15-19
AbstractParser, C-12
ACTToLaTeXVisitor, C-39
AddComposer, 16-24, 16-25
addComposer, 16-24
AddExprEval, 14-24
AddExprTree, 14-27
AddOpEval, 14-23
AddOpTree, 14-26
AddPrefix, 9-32
ADT, 9-24–9-30
AdtGen, 9-23
AdtGenFact, 9-23
album, C-4
Alt, 14-18, 14-19
AMovable, 12-50
AnimatedJPanel, 12-48
Animation, 12-48
AnonymousCounter, 12-11
Answer, 1-6
Apfel, 5-8
Apfelmaennchen, 12-20–12-22
ApfelWithMouse, 12-39, 12-40
arith1, 9-36
arith2, 9-36
ArithParsTreeBuilder, 14-31
ArithTokenizer, 14-29–14-31
ArrayToString, 3-33
ASimpleMovable, 12-51
AssertFac, 8-1
Assign, 9-3
AutomaticBoxing, 3-10
Autor, 10-53, G-12
Autoren, 10-53, G-12
AutorenType, 10-52, G-12
Axes, 10-18–10-22
AxisType, 10-17

Ball, 12-53
BallGame, 15-5
BallGameClient, 15-6
BallGameImpl, 15-5
BallGameServer, 15-6
BetterUseOfImageIcon, 12-63
BinFunction, 14-21
BinTree, 6-50–6-53
Birne, 5-8
BorderLayoutTest, 12-8
Bottom, 3-17
BouncingBall, 12-49, 12-50
Box, 5-2
BreakTest, 3-21
BubbleSort, 3-37
BufferedCopy, 7-19
ButtonLogic, 2-18, G-2

Calculate, 9-6
CallFullNameMethod, 2-9
CallStaticTest, 2-11
CastTest, 2-22
Catch1, 4-24
Catch2, 4-25
Catch3, 4-26
Catch4, 4-27
Catch5, 4-29
CatchOutOfBounds, 4-32
Choice, C-9
Clone, 7-4
ClosingApfelFrame, 12-40
Closure, C-14
CodeFragment, 10-14
CollectMax, 5-6
CollectMaxOld, 5-6
ColorTest, 12-18
Company, 9-19
company, 9-8
Complex, 12-19, 12-20
ComponentOverview, 12-41–12-43
ComponentToFile, 12-25–12-27
Composer, 16-11
CondExpr, 3-26
Cons, 6-8, 6-13
Constr, 9-22
ContainsNoA, 6-20
ContTest, 3-21
Copy, 7-16
Copy2, 7-17
CountApplet, 12-13
countCode, 10-47
Counter, 12-6
CounterListener, 12-5
CounterPanel, 12-13
countKapitel, 10-47

CountNodes, 10-10

Data, 9-22
DateClient, 15-3
DateServer, 15-2
Deadlock, 11-8
Dept, 9-19
Dialogue, 2-18, G-2
DialogueLogic, 4-11
DialogueLogics, 4-13
Dic, 7-12
Dictionary, 7-11
DisplayTree, 12-32–12-36
DisplayTreeTest, 12-37
DoIntOp, 14-23
DomDepth, 10-12
DoNodeTest, 10-24, 10-25
DoOpTree, 14-27
DoParentheses, 14-33, G-17
DoTest, 3-18
DoubleTest, 3-7
dtd, C-42
DTDDef, C-20
DTDDefFlatten, C-28
DTDShow, C-20
DU, 9-20

Element, C-14, C-15
ElseIf, 3-14
Employee, 9-20
Empty, 6-8
EncodedCopy, 7-18
Epsilon, 14-16
EQ, 5-8
Eq, 9-10
EqualVsIdentical, 7-2
ErsteFelder, 2-5
ErsteFelder2, 2-5
Euroschein, 3-40
EvalArith, 14-24
EvalKlip, 9-31
EvalXPath, 10-35–10-38
Example, 15-9
exampleProgs, 10-48
exampleProgs1, 10-47
exampleProgs2, 10-47

fac, 10-47
Fail, C-7
fak, 9-31
FifthThrow, 4-23
FilterCondition, 6-18
Finalize, 7-5
Finally, 4-31
FirstArray, 3-32
FirstIf, 3-13
FirstIf2, 3-13
FirstLocalFields, 2-8
FirstPersistentComposer, 16-12
FirstPersistentObject, 16-7
FirstProgram, 1-6
FirstThrow, 4-19
Flatten, 9-12
FlattenResult, C-28
FlowLayoutTest, 12-7
FontTest, 12-24
Foo, 10-55, G-14
ForBreak, 3-21
ForTest, 3-19
FourthThrow, 4-23
FromTo, 3-36, 4-17
FromToStep, 4-16
Function, 14-19
Function1, 9-14, C-6
Function2, 9-14

G2DTest, 12-28
GenerateADT, C-22
GenerateClassesForDTD, C-32
Genre, 16-15
Get, 11-10
GetCode, 10-15
GetDate, 15-1
GetDateImpl, 15-2
GetIntOp, 14-22
GetLeaves, 10-55, G-14
GetOpTree, 14-25
GetPersonName, 2-9
GetSet1, 11-9
GetSet2, 11-10
GetSet3, 11-11
GetSimpleProg, 9-4
GetVars, 9-32
GetWerke, 10-54, G-14
GPair, 5-4
GreaterX, 6-22
GridLayoutTest, 12-9
GuiHangs, 12-45
GuiHangsNoLonger, 12-60

HasJustTextChild, 10-53, G-13
HelloMobileWorld, 15-11, 15-12
HsParse, C-1–C-3
HtmlDialogue, 4-14
HtmlDialogueTest, 4-15
htmlfac, 10-47
HtmlView, 6-26, G-7

ImageIcon, 12-61, 12-62
ImagePaddle, 12-58
ImagePong, 12-59
Increase, 9-12
IncreaseReflect, 9-13
IncreaseSalary, 9-9
index, 16-20
InnerCounter, 12-10
InstanceOfTest, 2-21
Instrument, 16-13
IntIterator, 4-16
IntOp, 14-21
IsAtomic, C-32
Iterate, 7-9
IterTage, 3-40

JT, 12-4
JTB, 12-4
JTreeTest, 12-31
JustLogMe, 8-2

Klip, 9-30
KlipParser, 9-33–9-35
Ko, 5-10
Kovarianz, 3-34

LazyBool, 3-30
LessEqualX, 6-21
Li, 6-9, 6-11, 6-12, 6-14, 6-15, 6-19, 6-22, 6-24
LiIterator, 7-9
LineStageDirSubheadToLaTeX, C-41
List, 6-7, 6-13
ListAll, 16-22, 16-23
ListTest, 5-11
ListUsage, 7-7
LongString, 6-20
Loop, 9-4

MainDTDParse, C-33
MainParser, C-18
ManualBoxing, 3-10
Map, 14-19, 14-20, C-11
MBall, 15-14
Minimal, 2-4
Minus, 3-26
MMovable, 15-13
MobilePong, 15-18
Modifier, C-47
ModifyWebLinks, 16-10
ModifyWebLinksDirect, 16-9
MoreFinally, 4-31
Movable, 12-51
MPaddle, 15-15
MSimpleMovable, 15-13
Mult, 9-4
MultExprEval, 14-23
MultExprTree, 14-27
MultOpEval, 14-22
MultOpTree, 14-26
MusiccollectionParser, C-16, C-17
MusicDB, 15-8, 15-9
MyClass, 4-1
MyComp, 9-8
mymusic, C-4
MyNonPublicClass, 4-6
MyPublicClass, 4-6

Nachname, 10-53, G-13
NegativeNumberException, 4-23
NegativeOrderingCondition, 6-23
NewIteration, 5-14
NoClone, 7-3
Node, 9-3
NodeTest, 10-23
NonThrow, 4-20
NormalizeXPath, 10-40
NoteListIterator, 10-11
Num, 9-3
NumberTooLargeException, 4-27

ObjectArray, 3-32
OldBox, 5-1
OldIteration, 5-13
Operatorauswertung, 3-27
Optional, C-8
OpToken, 14-21
OpTree, 14-25
Opus, 10-53, 16-16, G-13
OrderingCondition, 6-23
Overloaded, 9-2
Overridden, 9-2

Paar, 7-10

package, 16-3, 16-13, 16-16, 16-18
package., 16-12
PackageTest, 4-9
Paddle, 12-54
Pair, 9-33, 14-17
Parentheses, 14-32, G-17
Parser, 14-15, C-7, C-8
ParserAux, C-47
ParserCode, C-24
ParseResult, 14-15, C-6
ParseToken, 14-16
ParseXML, 10-10
PCData, C-13
Person, 9-20, 10-53, G-12
Person1, 2-7
Person2, 2-12
PersonExample1, 2-6
PersonExample2, 2-7
play, C-37
PLAYToLaTeXVisitor, C-38
Plus, C-11
PMManager, 16-6
Pong, 12-54
PongApplet, 12-57
PongCanvas, 15-15
PrintIntIteratorElements, 4-16
PrintTwoNumbers, 11-4
PrivateTest, 4-8
ProtectedTest, 4-9
PU, 9-20
PunktVorStrich, 3-11

Query, 9-14, 9-15
Query2, 9-18
QuerySome, 9-17
QueryWebLinks, 16-8

ReaderIterator, 5-14
Reihenfolge, 3-27, 3-28
Relation, 6-23
RemoveAbbreviation, 10-31–10-34
Repetition, C-10
Return, C-13
ReturnString, 7-6
ReverseComparator, 7-13
Rhyme, 7-13
Rounded, 3-8

SafePrintTwoNumbers, 11-6
SafeSwap, 11-6
SafeTwoNumbers, 11-5
Salary, 9-20
SalaryBill, 9-15
SaxCountNodes, 10-14
SaxParse, 10-13
SceneToLaTeXVisitor, C-40
SchachFeld, 3-35
SearchTree, 6-55, 6-58, 6-60, 6-61
SecondArray, 3-33
SecondThrow, 4-21
SelectNodes, 10-14
Seq, 9-3, 14-17, 14-18, C-9
Set, 11-9
Show, 9-10
Show2, 9-16
ShowFile, 7-16
ShowKlip, 9-30
ShowNodeTest, 10-23
ShowType, C-21
ShowXPath, 10-30
simple, C-4
SimpleAnimation, 12-52
SimpleGraphics, 12-15
SimpleMovable, 12-52
SimpleThread, 11-2
SkriptTree, 6-48
SleepingThread, 11-2
SmileBall, 12-57
sms, 14-34, 14-35
SortStringLi, 6-16, 6-17
SpeechStageDirSubheadToLaTeX, C-40
Square, 3-11
StackTest, 3-24
StackTrace, 4-30
Star, C-11
StaticTest, 2-11
StrictBool, 3-30
StringBox, 5-5
StringLengthLessEqual, 6-24
StringLessEqual, 6-23
StringOrdering, 6-15
StringStartsWithA, 6-20
StringUtil, 4-5
StringUtilMethod, 2-6
StringVsStringBuffer, 7-6
Student, 2-13, 2-14
StudentOhneVererbung, 2-12
Sub, 9-4
SubUnit, 9-20
Summe, 3-14

Summe2, 3-17
Summe3, 3-19
Summe4, 3-20
Swap, 11-4
SwingWorker, D-1
Switch, 3-23

T, 9-21
Tage, 3-39
TerminationTest, 3-28, 3-29
TestArrayList, 4-3
Testbool, 3-9
TestboolOperator, 3-12
TestboolOperator2, 3-12
TestDialogue, 2-18, G-3
TestePerson1, 2-8
TestEQ, 5-8
TestEq, 9-11
TestEvalArith, 14-24
TestFilter, 6-20
TestFirstLi, 6-10
TestFirstLi2, 6-11
TestFirstList, 6-8
TestFlatten, 9-13
TestFromTo, 4-17
TestImport, 4-4
TestImport2, 4-4
TestImport3, 4-5
TestIncrease, 9-12
TestIncreaseReflect, 9-13
TestInteger, 3-5, G-3
TestKlip, 9-32
TestLateBinding, 2-17
TestLength, 6-12
TestListLength, 6-13
TestPaket, 4-1
TestQualifier, 10-39
TestReaderIterator, 5-15
TestSalaryBill, 9-15
TestSearchTree, 6-56
TestShow, 9-10
TestSortBy, 6-25
TestStudent, 2-15
TestStudent1, 2-15
TestStudent2, 2-16
TestToXML, 9-16, 9-17
TestToXML2, 9-19
TestTreeArith, 14-28
TextUtils, 5-15
TheaterAutoren, 10-52, G-11
ThrowIndex, 4-21
ToHTMLString, 4-12
Token, 14-15
ToUpper, 4-13
ToUpperCase, 4-12
ToXML, 9-16
ToXML2, 9-18
Trace, 5-12
Transform, 9-11
Transformer, 9-11
TransformReflect, 9-13
Tree, 6-32–6-34, 6-36–6-48, 9-21, 12-30, 12-31
TreeArith, 14-28
TreeCondition, 6-37
TreeToFile, 12-38
TSize, 9-22
Tuple1, C-5
Tuple2, C-5
Tuple3, C-6
TwoNumbers, 11-3

Uhr, 12-46, 12-47
UniPair, 5-5
UpperConversion, 4-13
UseBox, 5-3
UseCollectMax, 5-7
UseImageIcon, 12-62
UseOldBox, 5-2
UseOldBoxError, 5-2
UsePair, 5-4
UsePublic, 4-6
UseSimpleGraphics, 12-15
UseStringBox, 5-5
UseStringUtil, 4-6
UseThis, 2-11
UseUniPair, 5-5

Var, 9-3
VarParams, 2-10
Vergleich, 3-12
ViewApplet, 12-14
ViewPong, 12-57
VisibilityOfFeatures, 4-8
Visitor, 9-4, C-6
Vorname, 10-54, G-13
VorUndNach, 3-18

WarnList, 5-11
web, 16-19, 16-20

WebLink, 16-2, 16-10
Werke, 10-53, G-12
WerkeListe, 10-54, G-13
WhileTest, 3-16
Window1, 12-2
Wochentage, 3-38
WriteParser, C-25
WriteReadObject, 7-20

XPath, 10-29
XPathResult, 10-35
XSLT, 10-46

Zahl, 14-20
ZahlEval, 14-22
ZahlTree, 14-25

Abbildungsverzeichnis

2.1 Modellierung einer Person. 2-2
2.2 Modellierung eines Buches. 2-3
2.3 Modellierung eines Datums. 2-3
2.4 Modellierung eines Ausleihvorgangs. 2-3

4.1 Ein Gui-Dialog mit Html-Ausgabe. 4-15

6.1 Schachtel Zeiger Darstellung einer dreielementigen Liste. 6-4
6.2 Schachtel Zeiger Darstellung der Funktionsanwendung von concat auf zwei Listen. 6-5
6.3 Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt. 6-5
6.4 Schachtel-Zeiger-Darstellung nach dem zweiten Reduktionsschritt. 6-6
6.5 Schachtel Zeiger Darstellung des Ergebnisses nach der Reduktion. 6-6
6.6 Modellierung von Listen mit drei Klassen. 6-7
6.7 Modellierung der Filterbedingungen. 6-19
6.8 Modellierung von Bäumen mit einer Klasse. 6-31
6.9 Modellierung von Bäumen mittels einer Klassenhierarchie. 6-32
6.10 Schematische Darstellung der Militärordnung. 6-54
6.11 Nicht balanzierter Baum. 6-59
6.12 Baum nach Rotierungsschritt. 6-59
6.13 Baum nach weiteren Rotierungsschritt. 6-60

10.1 Anzeige der per XSLT-Skript generierten HTML-Seite. 10-45

12.1 Ein erstes Fenster. 12-3
12.2 Fenster mit Textfläche 12-4
12.3 Fenster mit Textfläche und Knopf 12-5
12.4 Einfache graphische Komponente. 12-16
12.5 Kreis mit Durchmesserlinien. 12-18

12.6 Erste farbige Graphik. 12-19
12.7 Apfelmännchen. 12-23
12.8 Verschiedene Fontgrößen. 12-26
12.9 Transformationen auf Text in Graphics2D. 12-30
12.10 Baumdarstellung mit JTree. 12-33
12.11 Baum graphisch dargestellt mit DisplayTree. 12-38
12.12 Überblick über einfache Komponenten. 12-43
12.13 Überblick über komplexere Komponenten. 12-44
12.14 Eine einfache Digitaluhr. 12-47

14.1 Ableitungsbaum für a moon orbits mars. 14-8
14.2 Ableitungsbaum für mercury is a planet. 14-8
14.3 Ableitungsbaum für planet orbits a phoebus. 14-9

15.1 Handysimulation mit WTK2.2. 15-14

16.1 Datenbank nach Erzeugen des Schemas von JPOX. 16-6
16.2 Datenbanktabelle mit drei Javaobjekten. 16-8

G.1 Kreis mit Durchmesserlinien. G-16

Literaturverzeichnis

[Arn04] Arnaud Le Hors and Philippe Le Hégaret and Lauren Wood and Gavin Nicol and Jonathan Robie and Mike Champion and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification Version 1.0. W3C Recommendation, April 2004. `http://www.w3.org/TR/P3P/`.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) 1.0. W3C Recommendation, November 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116.xml`.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions of Information Theory*, 2:113–124, 1956.

[CWH00] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial Third Edition*. Addison-Wesley Publishing Company, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[GKS01] Ulrich Grude, Chritsoph Knabe, and Andreas Solymosi. Compilerbau. Skript zur Vorlesung, TFH Berlin, 2001. `www.tfh-berlin.de/~grude/SkriptCompilerbau.pdf`.

[Gru01] Ulrich Grude. Einführung in gentle. Skript, TFH Berlin, 2001. `www.tfh-berlin.de/~grude/SkriptGentle.pdf`.

[Gru03] Ulrich Grude. Einf"uhrung in die Programmierung mit Java. Skript, TFH Berlin, 2003. `www.tfh-berlin.de/~grude/SkriptJava1SS03.pdf`.

[HM96] Graham Hutton and Eric Meijer. Monadic parser combinators. submitted for publication,
`www.cs.nott.ac.uk/Department/Staff/gmh/bib.html`, 1996.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler compiler. Technical Report 39, Bell Laboratories, 1975.

[Ler97] Xavier Leroy. The Caml Light system release 0.73. Institut National de Recherche en Informatique et Automatique, 1 1997.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. IT Press, Cambridge, Massachusetts, 1990.

[NB60] Peter Naur and J. Backus. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, may 1960.

[Oa04] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[OW97] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.

[Pan00] Sven Eric Panitz. Generische Typen in Bolero. *Javamagazin*, 4 2000.

[Pan04] Sven Eric Panitz. Erweiterungen in Java 1.5. Skript zur Vorlesung, TFH Berlin, 2004. `www.panitz.name/java1.5/index.html`.

[Pan05] Sven Eric Panitz. The Scrap Your Boilerplate Pattern in Java. Draft; Online since March 2005 `panitz.name/paper/boilerplate/index.html`, 3 2005.

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15. IEEE Computer Society, 1998.

[PJ03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[Pv97] M. J. Plasmeijer and M van Eekelen. *Concurrent Clean Language Report 1.2*. University of Nijmegen, 1997.

[T. 04] T. Bray, and al. XML 1.1. W3C Recommendation, February 2004. `http://www.w3.org/TR/xml11`.

[Vis01] Joost Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 270–282. ACM Press, 2001.

[Wad85] Phil Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer, 1985.

[Wad00] Philip Wadler. A formal semantics of patterns in XSLT, March 2000.

[Wor99] World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.