Landeshauptstadt
München
**Referat für
Bildung und Sport**

Bayerischer Landtag

mit OOP, XML, HTML & Swing zu den Abgeordneten

Städtische Berufsschule für Informationstechnik

Herausgeber	Michael Niedermair Städtische Berufsschule für Informationstechnik https://www.bsinfo.eu/
Autor(en)	Michael Niedermair, Thomas Grosser
Quellen, Material, Unterstützung, ...	• Elke Niedermair • Oracle: Java-API • Oracle: Java-Tutorial • Bildarchiv Bayerischer Landtag, Foto Rolf Poss – Seite(n): 1, 2, 4, 9, 8, 8
Satz	Michael Niedermair
Version	2016-06-22 (Revision: 1651) **(BETA-Entwurf)**
Lizenz	http://creativecommons.org/licenses/by-nc-nd/3.0/de/

Inhaltsverzeichnis

1 Vorwort 5

2 Einführung 9

3 Einführung in Swing 11
3.1 Einfaches Fenster 11
3.2 Vereinfachung mit Lambda 14
3.3 Todo-Liste 15

4 XML 17
4.1 Einführung 18
4.2 Grundlagen XML 20
4.3 Übersicht: Java und XML 23

5 Testgetriebene Entwicklung 37
5.1 Wie funktioniert ein Testtreiber? 37
5.2 XML-Baum 39

6 CSV 43
6.1 Daten einspielen über CSV 43
6.2 Parser Apache-Commons-CSV 45

7 GUI mit Swing 47
7.1 Erster Baum 47
7.2 Synchrone Tabellen-Anzeige 52
7.3 Synchrone HTML-Anzeige 56

8 Anhang 59
8.1 Datencontainer 60
8.2 Datei-IO 74
8.3 Vererbung und Co. 101
8.4 Einführung in Lambda 128
8.5 Swing 135
8.6 „Dies und Das“ 152
8.7 Schülerbeispiele 155

Java 159

Stichwortverzeichnis 171

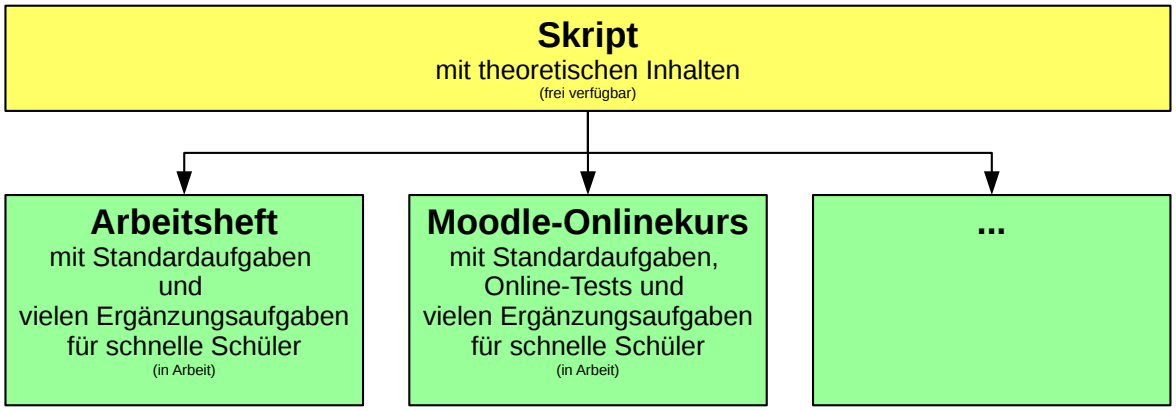
1 Vorwort

Der Unterricht bei den IT-Berufen (Fachinformatiker/-in–Anwendungsentwicklung und Fachinformatiker/-in–Systemintegration), gerade in Verkürzerklassen (2 Jahre, anstelle von 3 Jahren Ausbildungszeit) ist sehr inhomogen. Das Spektrum reicht hier von Schülern, die gerade einmal etwas Programmierung hatten, bis hin zu Studienabbrechern, die mehrere Semester Informatik studiert haben und oft an einem „Nebenfach" gescheitert sind, welches Sie nicht wirklich ernst genommen haben.

Bei solch inhomogenen Gruppen ist ein normaler Unterricht extrem schwierig. Die „Einen" können gerade dem Unterricht folgen, die „Anderen" sind mit den Aufgaben in kürzester Zeit fertig und langweilen sich dann sehr schnell.

Das Skript

Dieses Skript dient dabei als theoretische Grundlage, um Aufgaben, Lernsituationen etc. zu erstellen, die individuell auf Klassen angepasst werden können.



Das Skript hat einen Hauptteil mit den Schwerpunkten Einführung in Swing, XML, Testgetriebene Entwicklung, CSV und umfangreichere Swing-Komponenten. Im Anhang sind viele Bereiche enthalten, die „mit einfließen" und Unterstützung leisten. Je nach Klasse können hier zusätzliche Aufgaben eingebaut werden oder der Anhang nur als erneute Vertiefung und Wiederholung genutzt werden, wenn die Themen schon behandelt worden sind.

Das Skript ist der „erste Wurf" und es sind viele Ideen, Aufgaben, Übungen etc. eingeflossen, die sich so in den Jahren als Lehrer angesammelt haben. Leider sind auch viele Sachen nicht eingeflossen, da der Fertigstellungstermin immer näher rückte.

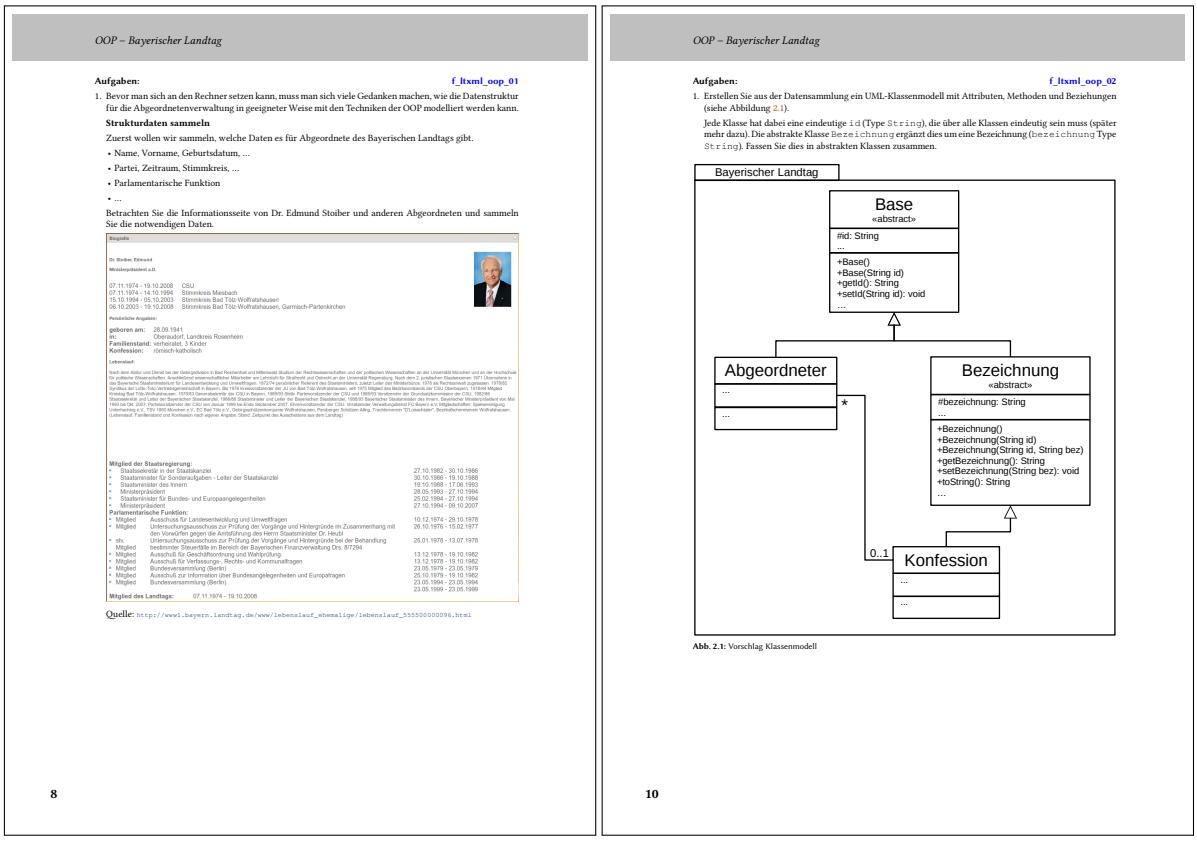
Die hier verwendeten Daten sollen auch dazu dienen, im nächster Zukunft ein weiteres Skript mit dem Thema „Bayerischer Landtag – Daten im Griff mit MySQL & SQL" zu erstellen und mit „echten" Daten Abfragen etc. zu erstellen und SQL zu üben.

Das Arbeitsheft

Zu dem Skript ist ein Arbeitsheft im Entstehen, welches gezielt auf die jeweilige Klasse mit Aufgaben, Übungen etc. angepasst wird (siehe Abbildung 1.1 auf der nächsten Seite).

Moodle-Online-Kurs

Ebenfalls ist ein Moodle-Online-Kurs im Entstehen, der nicht nur die Aufgaben enthält, sondern viele Möglichkeiten von Moodle verwendet, beispielsweise die Abgabe von Lösungen oder die Ablegung von Tests. Dabei steht das Skript als Onlineversion zu Verfügung (es wird ganz ohne Papier gearbeitet), wobei alle Beispiele als Datei eingebunden sind, so dass diese einfach in die eigene Entwicklungsumgebung eingebunden werden kann. So können die Schüler die Aufgaben nicht nur im Unterricht bearbeiten und lösen, sondern dies auch von zu Hause oder vom Betrieb aus erledigen, gerade wenn es um interessante Zusatzaufgaben geht.



OOP – Bayerischer Landtag

Aufgaben: f_ltxml_oop_01

1. Bevor man sich an den Rechner setzen kann, muss man sich viele Gedanken machen, wie die Datenstruktur für die Abgeordnetenverwaltung in geeigneter Weise mit den Techniken der OOP modelliert werden kann.

   Strukturdaten sammeln

   Zuerst wollen wir sammeln, welche Daten es für Abgeordnete des Bayerischen Landtags gibt.

   - Name, Vorname, Geburtsdatum, ...
   - Partei, Zeitraum, Stimmkreis, ...
   - Parlamentarische Funktion
   - ...

   Betrachten Sie die Informationsseite von Dr. Edmund Stoiber und anderen Abgeordneten und sammeln Sie die notwendigen Daten.

   Biografie

   Dr. Stoiber, Edmund
   Ministerpräsident a.D.

   07.11.1974 - 19.10.2008 CSU
   07.11.1974 - 14.10.1994 Stimmkreis Miesbach
   15.10.1994 - 05.10.2003 Stimmkreis Bad Tölz-Wolfratshausen
   06.10.2003 - 19.10.2008 Stimmkreis Bad Tölz-Wolfratshausen, Garmisch-Partenkirchen

   Persönliche Angaben:

   geboren am: 28.09.1941
   in: Oberaudorf, Landkreis Rosenheim
   Familienstand: verheiratet, 3 Kinder
   Konfession: römisch-katholisch

   Lebenslauf:

   Nach dem Abitur und Dienst bei der Gebirgsjägerbrigade in Bad Reichenhall und Mittenwald Studium der Rechtswissenschaften und der politischen Wissenschaften an der Universität München und an der Hochschule für Politik München. 1964 Staatsexamen, 1966 Promotion zum Dr. jur. 1968 Assessorexamen. 1968 Referent im Bundesministerium für Landesverteidigung, 1970 Referent im Bundeskanzleramt, 1971 Referent im Bundesministerium für Landesverteidigung, 1972 Referent im Bayerischen Staatsministerium für Landesentwicklung und Umweltfragen, 1972/74 persönlicher Referent des Staatsministers, Juli/Oktober Leiter des Ministerbüros, 1978 als Rechtsanwalt zugelassen, 1978/82 Syndikus der Lotto-Toto-Vertriebsgemeinschaft in Bayern, ab 1978 Kreisvorsitzender der JU von Bad Tölz-Wolfratshausen, seit 1973 Mitglied des Bezirksvorstandes der CSU Oberbayern, 1978/84 Mitglied Kreistag Bad Tölz-Wolfratshausen, 1979/93 Generalsekretär der CSU in Bayern, 1980/93 Stellv. Parteivorsitzender der CSU, seit 1980/93 Vorsitzender der Grundsatzkommission der CSU, 1982/86 Staatssekretär und Leiter der Bayerischen Staatskanzlei, 1986/88 Staatsminister und Leiter der Bayerischen Staatskanzlei, 1988/93 Bayerischer Staatsminister des Innern, Bayerischer Ministerpräsident von Mai 1993 bis Oktober 2007, Vorsitzender der Bayerischen Landesstiftung, Kuratoriumsvorsitzender der Hanns-Seidel-Stiftung, Kuratoriumsmitglied der Deutschen Stiftung für Internationale Entwicklung e.V., TSV 1860 München e.V., EC Bad Tölz e.V., Gebirgsschützenkompanie Wolfratshausen, Pfarrberger Schützen Alling, Trachtenverein "D'Loisachtaler", Bezirkstrachtenverein Wolfratshausen. (Lebenslauf, Familienstand und Konfession nach eigener Angabe. Stand: Zeitpunkt des Ausscheidens aus dem Landtag)

   Mitglied der Staatsregierung:

• Staatssekretär in der Staatskanzlei	27.10.1982 - 30.10.1986
• Staatsminister für Sonderaufgaben - Leiter der Staatskanzlei	30.10.1986 - 19.10.1988
• Staatsminister des Innern	19.10.1988 - 17.06.1993
• Ministerpräsident	28.05.1993 - 27.10.1994
• Staatsminister für Bundes- und Europaangelegenheiten	25.02.1994 - 27.10.1994
• Ministerpräsident	27.10.1994 - 06.10.2007

   Parlamentarische Funktion:

• Mitglied Ausschuss für Landesentwicklung und Umweltfragen	10.12.1974 - 29.10.1978
• Mitglied Untersuchungsausschuss zur Prüfung der Vorgänge und Hintergründe im Zusammenhang mit den Vorwürfen gegen die Amtsführung des Herrn Staatsminister Dr. Heubl	26.10.1976 - 15.02.1977
• stv. Untersuchungsausschuss zur Prüfung der Vorgänge und Hintergründe bei der Behandlung bestimmter Steuerfälle im Bereich der Bayerischen Finanzverwaltung Drs. 8/7294	25.01.1978 - 13.07.1978
• Mitglied Ausschuß für Geschäftsordnung und Wahlprüfung	13.12.1978 - 19.10.1982
• Mitglied Ausschuß für Verfassungs-, Rechts- und Kommunalfragen	13.12.1978 - 19.10.1982
• Mitglied Bundesversammlung (Berlin)	23.05.1979 - 23.05.1979
• Mitglied Ausschuß zur Information über Bundesangelegenheiten und Europafragen	25.10.1979 - 19.10.1982
• Mitglied Bundesversammlung (Berlin)	23.05.1994 - 23.05.1994
	23.05.1999 - 23.05.1999

   Mitglied des Landtags: 07.11.1974 - 19.10.2008

   Quelle: http://www1.bayern.landtag.de/www/lebenslauf_ehemalige/lebenslauf_555500000096.html

   8

OOP – Bayerischer Landtag

Aufgaben: f_ltxml_oop_02

1. Erstellen Sie aus der Datensammlung ein UML-Klassenmodell mit Attributen, Methoden und Beziehungen (siehe Abbildung 2.1).

   Jede Klasse hat dabei eine eindeutige `id` (Type `String`), die über alle Klassen eindeutig sein muss (später mehr dazu). Die abstrakte Klasse `Bezeichnung` ergänzt dies um eine Bezeichnung (`bezeichnung` Type `String`). Fassen Sie dies in abstrakten Klassen zusammen.

   Bayerischer Landtag
   Base «abstract»: #id: String; ...; +Base(); +Base(String id); +getId(): String; +setId(String id): void; ...
   Abgeordneter: ...; ...
   Bezeichnung «abstract»: #bezeichnung: String; ...; +Bezeichnung(); +Bezeichnung(String id); +Bezeichnung(String id, String bez); +getBezeichnung(): String; +setBezeichnung(String bez): void; +toString(): String; ...
   Konfession: ...; ...
   * 0..1

   Abb. 2.1: Vorschlag Klassenmodell

   10

Abb. 1.1: Aufgabenbeispiele

Daten

Die Daten der Abgeordneten stammen von den öffentlich zugänglichen Seiten des Landtages.[1] Mit Hilfe des HTML-Parsers „jsoup“ wurden die Seiten eingelesen, die Struktur ermittelt und dann gespeichert.

Eine direkte Weitergabe der Daten vom Bayerischen Landtag war leider aus rechtlichen Gründen nicht möglich.

Dank

Ein besonderer Dank für die Unterstützung geht dabei an Herrn Bernhard Seidenath, MdL mit Team und an Frau Radl (Öffentlichkeitsarbeit – Bayerischer Landtag) u. a. für die Zurverfügungstellung diverser Bilder.[2]

Für das intensive Durcharbeiten der Alpha-Version, Finden von vielen Besonderheiten und ausgiebige Tests möchten wir unseren Dank den beiden Klassen 1101VA und 11Z2VA (Jahrgang 2015/16) ausdrücken.

Ein großer Dank geht auch an die vielen Helfer und Unterstützer, die mit Rat und Tat, Ideen, Anregungen, Aufgaben etc. mit geholfen haben.

Lizenz

Das Skript steht unter der Lizenz CC-BY-NC-ND und kann entsprechend der Lizenzbedingungen verwendet werden.[3]

[1] siehe http://www1.bayern.landtag.de/webangebot1/ehem_abgeordnete.maske.jsp
[2] siehe https://www.bayern.landtag.de/fileadmin/Internet_Dokumente/Oeffarbeit_Paed_Betreuung/Nutzungsbedingungen_aktuell.pdf
[3] siehe http://creativecommons.org/licenses/by-nc-nd/3.0/de/

Beim Einsatz werden sicher viele weitere Ideen entstehen, Ungereimtheiten oder Besonderheiten, auch Fehler genannt, auftauchen. Wir werden diese sammeln und versuchen, diese am Ende des Schuljahres entsprechend einzuarbeiten. Daher würden uns über Anregungen, Kritik, Lob etc. freuen.

Wir wünschen viel Spaß und Erfolg beim Durcharbeiten

Michael Niedermair, StD

Thomas Grosser, OStR

2 Einführung

Der Bayerische Landtag stellt die Daten der Abgeordneten (von 1946 bis heute) zur Verfügung.[1]

Ihre Aufgabe ist es, die Daten entsprechend in einer GUI darzustellen.

Dazu sollen Sie eine geeignete OOP-Struktur entwickeln, die Daten aus verschiedenen CSV-Dateien in eine XML-Struktur migrieren und mit einer Swing-Oberfläche entsprechend ansprechend anzeigen.

Alle Bereiche sollen so ausgelegt werden, dass diese nachträglich leicht zu erweitern sind.

Wenn Sie mit einem Thema/Begriff nicht „klar kommen“, sehen Sie zuerst im Stichwortverzeichnis nach, ob hier entsprechende Einträge vorhanden sind.

Abb. 2.1: Anzeige in Baumansicht

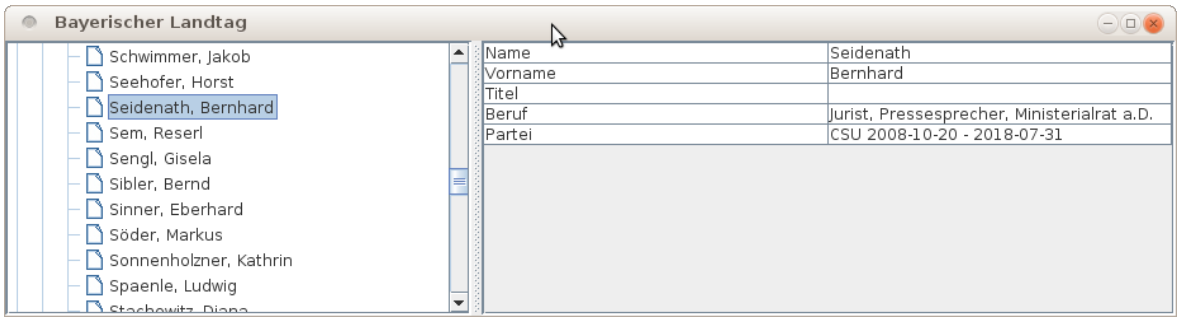


Abb. 2.2: Daten zu einem Abgeordneten (einfaches Beispiel)

Weitere Beispiele sind auf Seite 155 zu finden.

[1] siehe `http://www1.bayern.landtag.de/webangebot1/ehem_abgeordnete.maske.jsp`

Java-Version

Für alle Beispiele und Übungen ist Java 8 notwendig.

Mit dem Parameter `-version` kann jeweils von der Laufzeitumgebung bzw. vom Compiler die Version ermittelt werden.

```
java -version
javac -version
```

```
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
javac 1.8.0_91
```

Grundlagen: sollten bekannt sein!

- Java Basics
- Array, Listen und Co.
- Datei-IO
- XML
- UML

Ergänzendes Material

- **Handbuch der Java-Programmierung**
  O'Reilly Verlag, 8. Auflage 2014, 1104 Seiten, ISBN 978-3-95561-514-7
  http://www.javabuch.de/
- **Java - Der umfassende Programmierkurs**
  O'Reilly Verlag, 1. Auflage Mai 2014, 960 Seiten, ISBN 978-3-95561-540-6
- **Java ist auch eine Insel**
  Rheinwerk-Verlag, 11. Auflage 2014, 1306 Seiten, ISBN 978-3-8362-2873-2
  http://openbook.rheinwerk-verlag.de/javainsel/
  http://www.tutego.de/javabuch/
- **Java SE 8 Standard-Bibliothek**
  Rheinwerk-Verlag, 2. Auflage 2014, 1448 Seiten, ISBN 978-3-8362-2874-9
  https://www.rheinwerk-verlag.de/java-se-8-standard-bibliothek_3607/
  http://www.tutego.de/javabuch/

3 Einführung in Swing

Es gibt verschiedene GUI-Systeme, die in Java verfügbar sind:

AWT Das Abstract Window Toolkit (kurz AWT) ist von Anfang an bei Java enthalten und stellt eine quasi-plattformunabhängige grafische Oberfläche bereit.[1]

Swing Swing wurde entwickelt, um den Beschränkungen von AWT entgegenzuwirken. Dadurch können auch komplexe grafische Oberflächen wie Bäume oder Tabellen verwendet werden.[2]

SWT Das Standard Widget Toolkit (kurz SWT) ist eine Bibliothek für GUIs in Java, die auf native Bibliotheken des Betriebssystems zurückgreift. SWT wurde von IBM für die IDE Eclipse entwickelt.[3]

JavaFX JavaFX ist ein GUI-System um moderne Anwendungen mit Rich-Media-Inhalten, Audio und Video zu ermöglichen.[4]

3.1 Einfaches Fenster

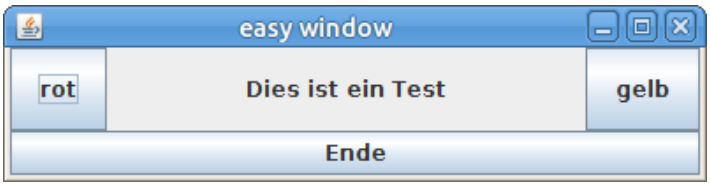


Abb. 3.1: Einfaches Swing Fenster

- Zwei Schaltflächen (Button's): Ändern die Farbe des Textes.
- Button „rot“ -> Text wird rot
- Button „gelb“ -> Text wird gelb
- Button „Ende“ -> Das Programm wird verlassen.

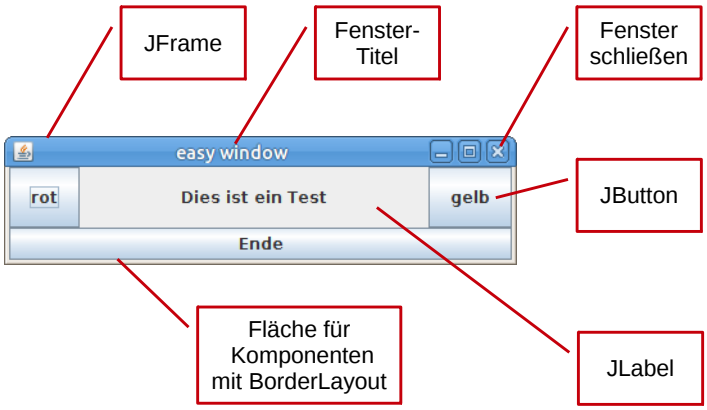


Abb. 3.2: Die Komponenten

JFrame	Das Fenster, welches eine Titelleiste mit Schaltflächen enthält.
JLabel	Eine Anzeigefläche für einzeiligen Text.
JButton	Eine Schaltfläche, die u. a. auf Mausklicks reagiert.
BorderLayout	Der Layoutmanager, der die Komponenten im Fenster anordnet. `BorderLayout` richtet die Komponenten nach Norden, Süden, Westen, Osten und zentriert aus.

[1] siehe https://de.wikibooks.org/wiki/Java_Standard:_Grafische_Oberfl%C3%A4chen_mit_AWT
[2] siehe https://de.wikibooks.org/wiki/Java_Standard:_Grafische_Oberfl%C3%A4chen_mit_Swing
[3] siehe https://de.wikipedia.org/wiki/Standard_Widget_Toolkit
[4] siehe https://de.wikipedia.org/wiki/JavaFX

Fenster-Titel Der Text, der in der Titelleiste angezeigt wird.

Fenster schließen Die Schaltfläche „X“, welche das Programm beenden soll.

Fläche für Komponenten Ein Fenster hat mehrere Zeichenflächen: z. B. eine für die Komponenten, für Aufklappmenüs, für Menüleisten, für Hintergrundbilder, …

Swing Tutorial

Von Oracle gibt es zu Swing ein sehr umfangreiches Tutorial mit vielen Beispielen. Es findet sich unter:
http://docs.oracle.com/javase/tutorial/uiswing/

Der Code im Einzelnen

```
1  /*
2   * $Id: EasyWindow.java 1588 2015-12-19 11:38:43Z michael $
3   */
4
5  package de.nm.gui.swing;
6
7  import java.awt.BorderLayout;
8  import java.awt.Color;
9  import java.awt.Container;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import java.awt.event.WindowAdapter;
13 import java.awt.event.WindowEvent;
14
15 import javax.swing.JButton;
16 import javax.swing.JFrame;
17 import javax.swing.JLabel;
```

Am Anfang befindet sich der Klassenkopfkommentar (`/* ... */`). Dieser enthält hier die Subversion-Id zur Codeverwaltung.

Mit `package` wird der Paketname (namespace) für die Klasse festgelegt. Dies bedeutet, die Datei `EasyWindow.java` befindet sich im Unterverzeichnis „de/nm/gui/swing“.

Anschließend werden alle benötigten Klassen mit `import` eingebunden.

Da Swing das „Rad nicht neu erfindet“, werden alle nicht sichtbaren Klassen von AWT, wie beispielsweise `Color`, weiter benutzt.

```
19 /**
20  * A easy window!
21  *
22  * @author <a href="mailto:mgn.schule@gmx.de">Michael Niedermair</a>
23  * @version $Revision: 1588 $
24  */
25 public class EasyWindow extends JFrame {
```

Die Klasse wird definiert. Dabei wird definiert, dass die Klasse von der Klasse `JFrame` alle Eigenschaften und Methoden erbt. Somit ist die Klasse `EasyWindow` auch ein Fenster! Über dem Klassenkopf befindet sich der `javadoc`-Kommentar (`/** ... */`), welcher mit dem Programm *javadoc* extrahiert werden kann. Die Tags (beginnen mit `@`) haben dabei eine besondere Bedeutung, u. a. werden diese bei der Ausgabe besonders formatiert.

```
27    /**
28     * main
29     */
30    public static void main(final String[] args) {
31
32       new EasyWindow();
33    }
```

In der `main`-Methode wird eine anonyme Instanz der Klasse `EasyWindow` erzeugt. Nach der Erzeugung wird automatisch der Konstruktor zur Initialisierung der Klasse aufgerufen.

```
35    /**
36     * the label with text.
37     */
38    private final JLabel label;
```

Die Klassenvariable `label` vom Typ `JLabel` wird definiert.

```
40    /**
41     * Create a new object.
42     */
43    public EasyWindow() {
44
45       super();
46
47       // title
48       setTitle("easy window");
```

Der Konstruktor dient zur Initialisierung der Klasse. Hier wird zuerst der Konstruktor der Überklasse (hier `JFrame`) mit `super()` aufgerufen. Danach wird der Titel (Überschrift) des Fensters festgelegt.

```
50       // window listener
51       addWindowListener(new WindowAdapter() {
52
53          @Override
54          public void windowClosing(final WindowEvent e) {
55             exit();
56          }
57
58       });
```

Damit die Aktion (Anwender klickt auf das 'x' im Fensterkopf) „Fenster schließen" implementiert wird, muss ein entsprechender `WindowAdapter` mit der Methode `addWindowListener()` hinzugefügt werden. Dabei wird die Methode `windowClosing()` der Klasse `WindowAdapter` überschrieben. Diese wird von der Laufzeitumgebung aufgerufen, wenn der Anwender auf 'x' klickt. In der Methode wird die Methode `exit()` (kommt später noch) aufgerufen.

```
60       // container for components and so on.
61       final Container con = getContentPane();
62
63       // layout
64       con.setLayout(new BorderLayout());
```

Die Fensterkomponenten werden in der Komponentenebene des Fensters platziert. Diese Ebene erhält man mit der Methode `getContentPane()`. Anschließend wird das Layout für diese Ebene festgelegt. Das Layout `BorderLayout` ordnet die Komponenten entsprechend der Himmelsrichtungen an (Norden, Süden, Osten, Westen und Mittig).

```
66       // components
67       label = new JLabel("Dies ist ein Test", JLabel.CENTER);
68       final JButton red = new JButton("rot");
69       final JButton yellow = new JButton("gelb");
70       final JButton exit = new JButton("Ende");
71
72       // put the components on the frame
73       con.add(label, BorderLayout.CENTER);
74       con.add(red, BorderLayout.WEST);
75       con.add(yellow, BorderLayout.EAST);
76       con.add(exit, BorderLayout.SOUTH);
```

Die Komponenten (`JLabel`, `JButton`) werden definiert und entsprechend platziert.

```
78       // add the action for the buttons
79       exit.addActionListener(new ActionListener() {
80
81          @Override
82          public void actionPerformed(final ActionEvent e) {
83             exit();
84          }
85       });
```

Mit der Methode `addActionListener()` wird die Aktion für den Button festgelegt. Dabei wird die Methode `actionPerformed()` des Interfaces `ActionListener` implementiert. Der `exit`-Button ruft dabei die Methode `exit()` auf.

```
87      red.addActionListener(new ActionListener() {
88
89         @Override
90         public void actionPerformed(final ActionEvent e) {
91            label.setForeground(Color.RED);
92         }
93      });
94
95      yellow.addActionListener(new ActionListener() {
96
97         @Override
98         public void actionPerformed(final ActionEvent e) {
99            label.setForeground(Color.YELLOW);
100        }
101     });
```

...

```
103     // set the size for the window
104     setSize(400, 100);
105
106     // make it visible
107     setVisible(true);
108  }
```

Die Größe des Fensters wird festgelegt und das Fenster wird sichtbar.

```
110  /**
111   * Exit the program.
112   */
113  private void exit() {
114
115     System.exit(0);
116  }
```

Die Methode `exit()` wird definiert. Hier wird die Laufzeitumgebung mit dem `ERRORLEVEL 0` (kein Fehler) verlassen.

3.2 Vereinfachung mit Lambda

```
78      // add the action for the buttons
79      exit.addActionListener(new ActionListener() {
80
81         @Override
82         public void actionPerformed(final ActionEvent e) {
83            exit();
84         }
85      });
```

Für den `ActionListener` wurde hier eine innere anonyme Klasse definiert. Verwendet man dafür einen Lambda-Ausdruck (ab Java 8), so kann dies deutlich vereinfacht werden (mehr dazu auf Seite 128).

```
76      // add the action for the buttons
77      exit.addActionListener((e) -> exit());
78      red.addActionListener((e) -> label.setForeground(Color.RED));
79      yellow.addActionListener((e) -> label.setForeground(Color.YELLOW));
```

3.3 Todo-Liste

In nächsten Schritt soll eine Einkaufsliste erstellt werden.

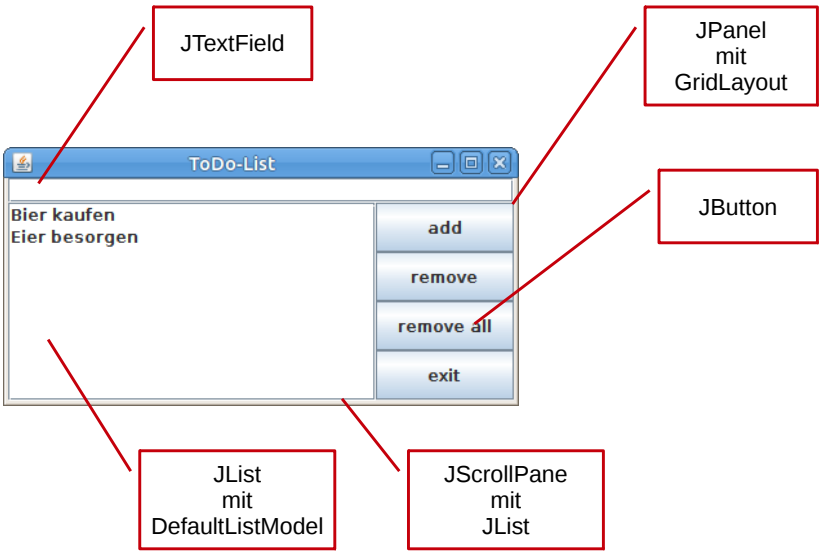


Abb. 3.3: Die Komponenten

Das Konzept, welches hinter der `JList` steckt, nennt sich MVC (Model-View-Controller) und zählt zu den Entwurfsmustern (engl. desginpattern).

Dabei wird hier eine Trennung zwischen Anzeige, Steuerung und Speicher hergestellt, so dass man die Anzeige und / oder den Speicher austauschen kann, ohne dass die andere Seite davon berührt ist. Es ist auch möglich, nicht nur einen Speicher zu verwenden, sondern die Speicherung in mehreren Bereichen (Speicher, Textdatei, DB, ...) durchzuführen.

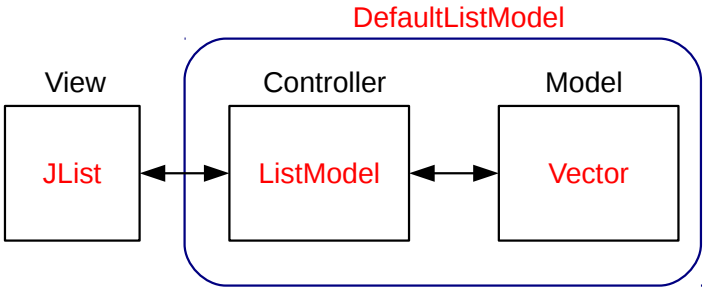


Abb. 3.4: Model-View-Controller

Verwendet man die Klasse `DefaultListModel`, so wird der Controller (hier eine Implementierung von `Listmodel`) und der Speicher (hier ein `Vector` (mehr dazu auf Seite 63)) in einem Objekt gehalten und Anfragen werden von der Klasse weitergereicht (engl. delegate → auch ein Entwurfsmuster).

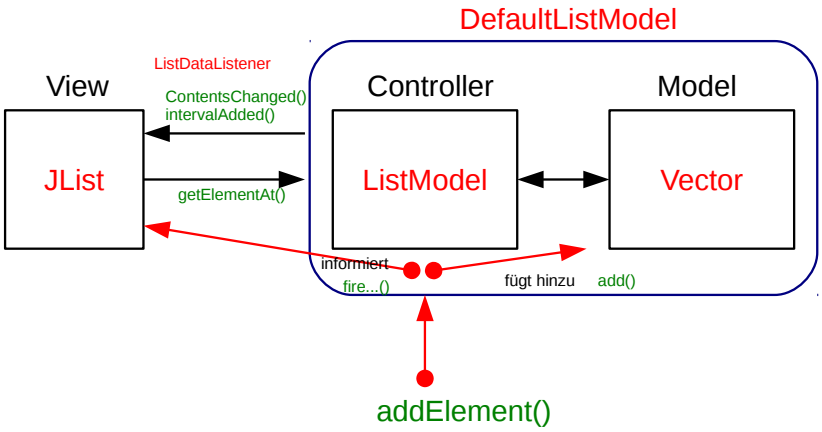


Abb. 3.5: Element hinzufügen

Über die Methode `addElement()` der Klasse `DefaultListModel` wird ein Eintrag hinzugefügt. Dieser wird dann dem `Vector` über die Methoden `add()` hinzugefügt und dann wird `JList` informiert.

`fireContentsChanged()` Ein oder mehrere Elemente wurden geändert.

`fireIntervalAdded()` Ein oder mehrere Elemente wurden hinzugefügt.

`fireIntervalRemoved()` Ein oder mehrere Elemente wurden entfernt.

Die `JList` ruft dann die Methoden `getSize()` und `getElementAt()` auf, um die Daten zu ermitteln. Zusätzlich kann hier auch Listener verwendet werden (später mehr dazu).

Codeausschnitte:

```
34   private final JList<String> list;
35   private final DefaultListModel<String> lm;
36   private final JTextField tf;
```

In der Klasse werden `JList` und `DefaultListModel` definiert. Da die Einträge der Liste Zeichenketten sind, wird als Typ jeweils `String` definiert. Danach folgt `JTextField` für die Eingabe.

```
61       // list ...
62       lm = new DefaultListModel<String>();
63       list = new JList<String>(lm);
64       final JScrollPane sp = new JScrollPane(list);
65       con.add(sp, BorderLayout.CENTER);
66
67       tf = new JTextField();
68       con.add(tf, BorderLayout.NORTH);
```

Im ersten Schritt wird das `ListModel` erzeugt. Dieses wird beim Erzeugen der `JList` als Parameter mit übergeben. Somit ist die View mit dem Controller und dem darin implementieren Speicher verbunden. Damit die Liste länger werden kann, bietet `JScrollPane` bei Bedarf vertikale bzw. horizontale Scrollleisten an.

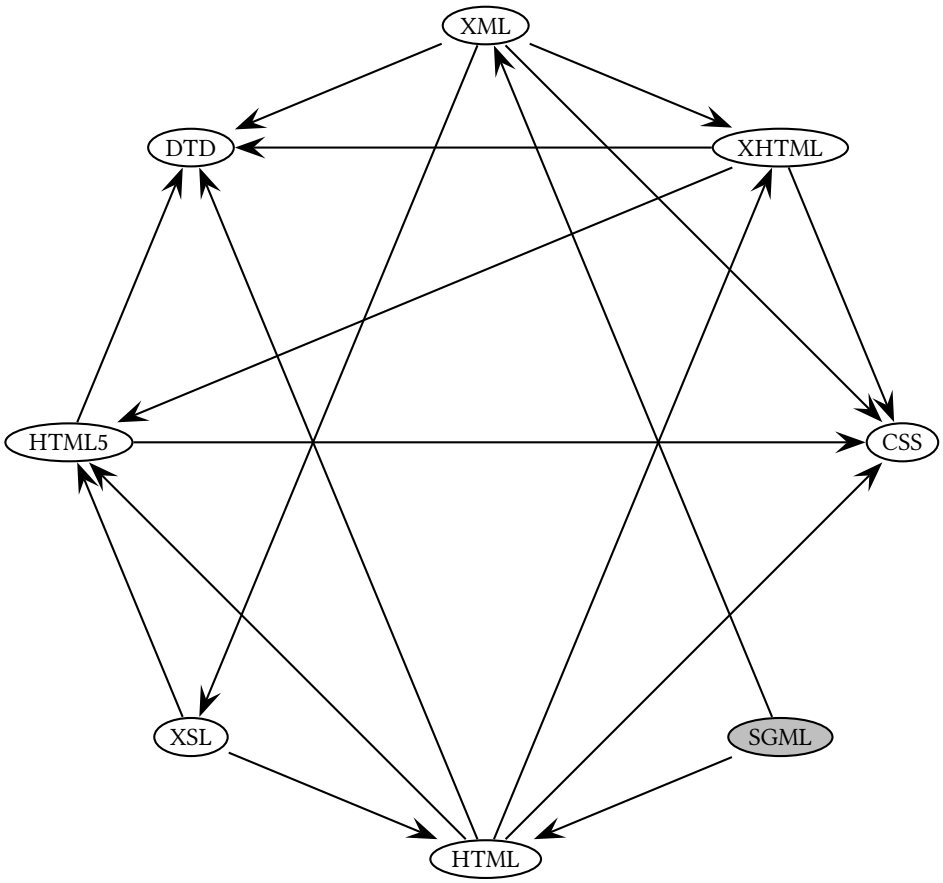
4 XML

Inhalt

4.1 Einführung 18
4.2 Grundlagen XML 20
4.2.1 Was ist XML? 20
4.2.2 Der Aufbau eines XML-Dokuments 20
4.3 Übersicht: Java und XML 23
4.3.1 APIs 23
4.3.2 JAXB 24
4.3.3 JDOM 28
4.3.4 XPATH 30

4.1 Einführung

Auszeichnungssprachen (Markup Languages) gibt es eine Menge. Die Standards sind hier HTML und XML mit den entsprechenden Erweiterungen wie beispielsweise CSS und XSL.



SGML Standard Generalized Markup Language
SGML ist eine Auszeichnungssprache, die in den 1960ern für technische Dokumentationen entwickelt worden ist (DIN-, EN- und ISO-Norm). SGML ist sehr mächtig, aber auch entsprechend komplex, weshalb es nur bei „Experten“ verwendet worden ist. Der Standard wurde 2012 ersatzlos zurückgezogen.

HTML Hypertext Markup Language
HTML wurde am Kernforschungszentrum Cern am 13. März 1989 von Tim Berners-Lee festgelegt, um Daten (Texte, Bilder, Hyperlinks, ...) zwischen Forschern auszutauschen bzw. auf diese zuzugreifen. HTML wurde dann später zur Weiterentwicklung an das W3C weitergegeben, bei dem Tim Berners-Lee den Vorsitz hat.

XML Extensible Markup Language
XML ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien. Durch seine beliebig anpassbare Struktur sind daraus sehr viele Formate, wie beispielsweise SVG für Vektorgrafiken, RSS für Webseiten-Feeds, MathML zur Beschreibung von Formeln, GPS zur Speicherung von Geodaten entstanden.

CSS Cascading Style Sheets
CSS ist eine deklarative Sprache für Stilvorlagen von strukturierten Dokumenten. Sie wird vor allem zusammen mit HTML und XML (zum Beispiel bei SVG) eingesetzt, um das Aussehen entsprechend anzupassen.

DTD Document Type Definition
DTD bzw. DOCTYPE ist ein Regelsatz, um Dokumente eines bestimmten Typs zu deklarieren.

XML Schema XML Schema Definition
XML Schema (kurz XSD) dient zur Definition von Strukturen in XML-Dokumenten, wie bei einer DTD, jedoch werden eine große Anzahl von Datentypen unterstützt.

XSL Extensible Stylesheet Language
XSL ist eine Transformationssprache zur Definition von Layouts für XML-Dokumente.

XHTML Extensible HyperText Markup Language
XHTML ist eine Neuformulierung von HTML 4 in XML. XHTML setzt dabei die Regeln von XML voraus und lässt sich somit einfacher vom Parser verarbeiten. XHTML wurde inzwischen von HTML 5 abgelöst.

HTML5 HTML5 ist die Weiterentwicklung von HTML und XHTML.

4.2 Grundlagen XML

4.2.1 Was ist XML?

Die Abkürzung XML steht für e**X**tensible **M**arkup **L**anguage. Anders als bei HTML stellt die Auszeichnungssprache XML keinen festen Satz an Sprachelementen zur Verfügung. Vielmehr kann XML beliebig mit eigenen Elementen erweitert und den eigenen Bedürfnissen angepasst werden. Dies hat zudem den Vorteil, dass die Dokumente leichter lesbar werden, da es möglich ist, den Elementen sprechende Namen zu geben.

Beispiel:

Eine Hauptüberschrift unter HTML wird durch die Zeile

```
<h1> Titel </h1>
```

definiert. XML macht auch ungeübten Lesern deutlich, worum es sich handelt.

```
<ueberschrift> Titel </ueberschrift>
```

XML stellt lediglich die Struktur der Daten und die Inhalte zur Verfügung. Wie diese Daten angezeigt oder verarbeitet werden, muss über Stylesheets geregelt werden. Dazu können entweder die aus HTML bekannten Cascading Style Sheets (CSS) oder aber die speziell für XML entworfene eXtensible Stylesheet Language (XSL) verwendet werden. Mithilfe derartiger Stylesheets können XML-Daten formatiert und am Bildschirm oder in Druckform ausgegeben werden.

Auch wenn XML dem Verfasser von XML-Dokumenten viele Freiheiten hinsichtlich der verwendeten Auszeichnungen gibt, unterliegen XML-Dokumente doch einigen Regeln, die eingehalten werden müssen.

4.2.2 Der Aufbau eines XML-Dokuments

XML–Dokumente bestehen aus verschiedenen Elementen. Jedes Element kann wiederum Unterelemente besitzen. Jedes Element besteht aus einem Start-Tag mit einem eindeutigen und gültigen Namen, danach folgt der Inhalt, den Abschluss bildet das End-Tag.

```
<tagname> Inhalt </tagname>
```

Gültigkeit des Tagnames

Damit ein Tagname gültig ist, muss er mit einem Buchstaben oder Unterstrich (_) beginnen. Danach darf der Name Buchstaben, Ziffern, Punkte, Bindestriche und Unterstriche in beliebiger Länge enthalten. Verboten sind allerdings Leerzeichen und auch Doppelpunkte werden nur in Verbindung mit Namensräumen verwendet.

Vorsicht Groß- und Kleinschreibung

XML unterscheidet sehr genau zwischen Groß- und Kleinschreibung. Nur wenn die Schreibweise des Tagnamens im End-Tag genau mit dem Start-Tag übereinstimmt, werden beide als zusammengehörig betrachtet.

End-Tag

Anders als bei HTML ist die Existenz eines End-Tags immer nötig, es sei denn, es handelt sich um leere Elemente. Andernfalls bricht der Parser die Verarbeitung des Dokuments ab. Parser sind Programme oder Programmkomponenten zur Überprüfung eines Dokuments. Außerdem bereitet der Parser die Daten so auf, dass sie von anderen Anwendungen weiterverarbeitet werden können. Es ist nicht erlaubt, dass der Parser selbständig fehlende Tags ergänzt.

Sind Elemente ineinander verschachtelt, so müssen diese in der Reihenfolge, mit der sie geöffnet wurden, umgekehrt wieder geschlossen werden, so dass alles von innen nach außen geschlossen wird. Es muss immer ein Element vorhanden sein, das alle anderen beinhaltet. Dieses wird als Wurzelelement bezeichnet. Leere

Elemente wie etwa Tags, die auf externe Bilder verweisen, müssen anders als in HTML immer mit `/>` geschlossen werden, damit sie der Parser verarbeiten kann. Außerdem ist zu beachten, dass der Attributwert in Anführungszeichen gesetzt werden muss.

```
<img src="bilddatei.gif"/>
```

Dies ist eigentlich eine Abkürzung für

```
<img src="bilddatei.gif"></img>
```

XML-Deklaration

Eine weitere Regel, die jedes XML-Dokument befolgen muss, ist die erste Zeile des Dokuments. Jedes Dokument beginnt mit der so genannten XML-Deklaration. Diese beinhaltet immer die verwendete Version, wobei zur Zeit die Version 1.0 verwendet wird. Außerdem kann der zu verwendende Zeichensatz angegeben werden und ein Attribut, das festlegt, ob es sich um ein eigenständiges Dokument handelt, das auf keine externen Dokumente zugreift. Die Fragezeichen am Anfang und am Ende der Deklaration weisen diese für den Browser als so genannte „Processing Instruction“ aus, also einen Verarbeitungshinweis, den der Browser entsprechend umsetzt.

```
<?xml version= "1.0" encoding="ISO-8859-1" standalone="yes" ?>
```

In obigem Beispiel wird der Zeichensatz `ISO-8859-1` verwendet, der unter anderem die deutschen Umlaute beinhaltet. Standardmäßig wird von XML Unicode `UTF-8` verwendet. Für andere Sprachen wie etwa türkisch oder russisch müssen die entsprechenden ISO-Zeichensätze `ISO-8859-2` bis `ISO-8859-15` verwendet werden. Welche Zeichensätze welche Sprachen beinhalten, finden Sie auf der Internetseite der ISO unter http://www.iso.org.

Entity-Referenzen

XML besitzt fünf vordefinierte Entity-Referenzen, die benötigt werden, damit Zeichendaten vom Parser nicht falsch interpretiert werden. So wird beispielsweise das Zeichen `<` immer als Beginn eines Tags interpretiert. Soll es im Text vorkommen, muss es dort mit Hilfe der entsprechenden Entity-Referenz umschrieben werden. Der Parser ersetzt die Entity-Referenz dann durch die richtigen Zeichendaten. Entity-Referenzen beginnen immer mit `&` und enden mit einem Strichpunkt. Die Tabelle 4.1 zeigt die fünf Standard-Entity-Referenzen.

Entity-Referenz	Zeichen
&	&
<	<
>	>
"	"
'	'

Tab. 4.1: Die vordefinierten Entity-Referenzen von XML

CDATA

Soll ein Text in XML dargestellt werden, der sehr viele dieser vordefinierten Entity-Referenzen enthält, gibt es eine Möglichkeit, sich die Schreibarbeit zu erleichtern, indem der Text in einen CDATA-Abschnitt eingebettet wird. CDATA steht für **C**haracter **D**ata. Solche Abschnitte werden vom Parser unverändert übernommen, so dass die Zeichen, die sonst durch die Entity-Referenzen dargestellt werden müssen, direkt eingetragen werden können. Ein CDATA-Abschnitt hat die Form

```
<![CDATA[ Inhalt beliebiger Art ]]>
```

und wird wie ein Unterelement in ein anderes Element eingebettet.

Wohlgeformte XML-Dokumente

Ein XML-Dokument, das die bisher beschriebenen Regeln befolgt, wird wohlgeformt genannt. Wohlgeformte Dokumente können vom XML-Prozessor (Parser) verarbeitet werden. Andernfalls bricht er mit einer Fehlermeldung ab.

Kommentare

Bei größeren Dokumenten kann es durchaus sinnvoll sein, in das Dokument Kommentare einzufügen, um einzelne Teile zu beschreiben und dem Leser deutlich zu machen, worum es sich handelt. Kommentare in XML beginnen stets mit `<!--` und enden mit `-->`. Alles dazwischen wird vom Parser ignoriert und nicht angezeigt.

Attribute

Eine andere Methode, einzelne Elemente näher zu beschreiben, ist die Verwendung von Attributen. Ein XML-Element kann beliebig viele Attribute besitzen, allerdings müssen diese unterschiedliche Namen haben. Die Attributwerte stehen immer in Anführungszeichen. Allgemein hat ein Element mit Attributen die Form:

```
<element attribut="attributwert"> Inhalt </element>
```

Sollen mehrere Attribute verwendet werden, werden diese einfach durch Leerzeichen voneinander getrennt.

```
<element attribut1="attributwert1" attribut2="attributwert2">
  Inhalt
</element>
```

Bei der Festlegung der Struktur Ihres Dokuments sollten Sie sich schon frühzeitig Gedanken darüber machen, welche Informationen in Form von Elementen und welche als Attribute angegeben werden. Denn Attribute haben einen entscheidenden Nachteil: Attribute können nur einfachen Text enthalten, Unterstrukturen sind anders als bei Elementen nicht möglich. Allerdings eignen sich Attribute hervorragend dazu, Informationen über ein Element einzubinden, die der Leser des Dokuments nicht braucht, die aber eventuell für weiterverarbeitende Anwendungen wichtig sind, ohne dass sie für den Leser sichtbar sind.

4.3 Übersicht: Java und XML

Um in Java auf XML zuzugreifen (lesend, teilweise auch schreibend), gibt es mehrere Möglichkeiten und APIs.

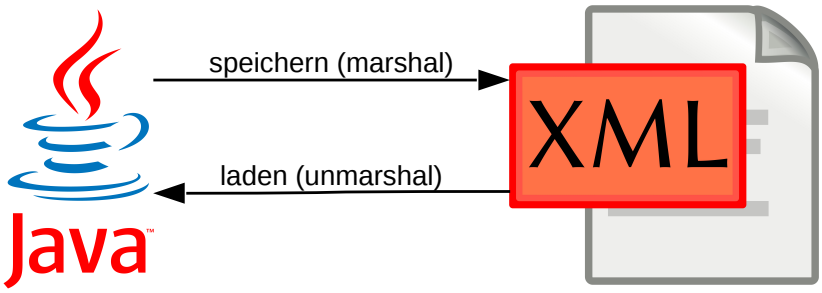


Abb. 4.1: Java-XML

4.3.1 APIs

Hier einige Möglichkeiten und Vertreter:

DOM-APIs Hier wird das gesamte XML-Dokument (XML-Baum) in den Speicher geladen (DOM - **D**ocument **O**bject **M**odel) und steht meist über die Klasse `Document` zur Verfügung, welches vom W3C definiert worden ist.[1] Zusätzliche Bibliotheken, wie JDOM[2], DOM4J[3] oder XOM[4] erweitern die Funktionalität. Zu beachten ist bei DOM, dass das gesamte XML-Dokument in den Speicher geladen wird und somit nur für kleine und mittlere XML-Dokumente geeignet ist.

Push-APIs Hier ruft der Parser (meist SAX - **S**imple **A**PI for **X**ML) nach dem Callback-Prinzip (ereignisorientiert) entsprechende Methoden auf und „meldet“ so das Auffinden von Elementen, Attributen usw. Der Parser kann hier nur in eine Richtung arbeiten, was bedeutet, dass auf ein Element nicht erneut vom Parser zugegriffen werden kann (Dieses muss bei Bedarf dann vorher gespeichert werden). Der große Vorteil dabei ist, dass nur sehr wenig Arbeitsspeicher benötigt wird und beliebig große XML-Dokumente verarbeitet werden können.

Pull-APIs Pull-Parser arbeiten ähnlich wie Push-Parser, nur dass der Parser nicht wie bei SAX Informationen an die Applikation über einen Event-Mechanismus schickt (PUSH), sondern die Anwendung holt sich die nächsten Informationen selbst, wenn sie diese benötigt (PULL). Es wird über den XML-Baum iteriert. Vertreter sind u. a. StAX[5] (**St**reaming **A**PI for **X**ML) und XPP[6] (**X**ml **P**ull **P**arser). Pull-Parser sind üblicherweise performanter als Push-Parser.

Mapping-APIs Bei der Mapping-API arbeitet der Programmierer nicht mit den XML-Rohdaten, sondern erhält nur entsprechende Objekte (Objekt-XML-Mapper). Vertreter sind u. a. JAXB[7] (**J**ava **A**rchitecture for **X**ML **B**inding), welches über Annotationen arbeiten, Castor[8] (das Mapping erfolgt über eine XML-Datei) und XStream[9] (das Mapping erfolgt dabei automatisch, kann aber auch von Hand festgelegt werden).

Da unsere XML-Dateien nicht sehr groß sind, wir ein OOP-Klassen-Modell besitzen und JAXB bei Java standardmäßig integriert ist, verwenden wir diese API.

[1] siehe http://de.wikipedia.org/wiki/Document_Object_Model
[2] siehe http://www.jdom.org/
[3] siehe http://dom4j.sourceforge.net/
[4] siehe http://www.xom.nu/
[5] siehe http://de.wikipedia.org/wiki/Streaming_API_for_XML
[6] siehe http://www.extreme.indiana.edu/xgws/xsoap/xpp/
[7] siehe http://de.wikipedia.org/wiki/Java_Architecture_for_XML_Binding
[8] siehe http://castor.codehaus.org/
[9] siehe http://xstream.codehaus.org/

4.3.2 JAXB

Bei JAXB wird das Mapping über Annotationen[10] definiert. Hier ein Auszug:[11]

`@XmlElement`

Legt fest, dass die Klasse ein XML-Element ist. Dadurch werden alle „getter/setter" zu Elementen bzw. Attributen. Standardmäßig wird der Klassenname zum Elementnamen (in Kleinbuchstaben). Über `name` kann der XML-Element-Name und über `namespace` der Namensraum festgelegt werden.

```
@XmlElement
public class Art { ... }

<art id="a1">
   <name>trocken</name>
</art>
```

`@XmlAccessorType`

Legt fest, ob Felder oder Properties gemappt werden sollen.

`PROPERTY`	Properties (getter/setter) werden gemappt (Standard)
`FIELD`	Felder werden gemappt
`PUBLIC_MEMBER`	Öffentliche Felder werden gemappt
`NONE`	Nichts wird gemappt

```
@XmlElement(name="weinart")
@XmlAccessorType(XmlAccessType.FIELD)
public class Art { ... }
```

`@XmlRootElement`

Legt fest, dass die Klasse das XML-root-Element ist.

`@XmlAttribute`

Legt fest, dass das Feld bzw. Property ein XML-Attribut wird. Über `name` kann der XML-Attribut-Name und über `namespace` der Namensraum festgelegt werden.

```
@XmlAttribute
@XmlID
public String getId() {
   return id;
}
```

`@XmlID`

Legt fest, dass auf dieses Element bzw. Attribut von woanders verwiesen wird (One-to-Many, Many-to-Many, ...).

`@XmlIDREF`

Legt fest, dass auf ein anderes Element bzw. Attribut verwiesen wird. Hier wird nur die ID gespeichert und nicht das komplette Element.

```
@XmlAttribute
@XmlIDREF
public Art getArt() {
   return art;
}

<wein art="a1" id="w1" ort="o1" type="t1">
```

`@XmlType`

Legt über `propOrder` fest, in welcher Reihenfolge die eingebetteten Elemente gespeichert werden.

```
@XmlElement
@XmlType(propOrder = { "preis", "flgroesse",
                       "rebsorten", "alk", "rz", "sa" })
public class Wein { ... }

<wein art="a1" id="w1" name="Amethyst" ort="o1" type="t1">
   <preis>0.0</preis>
   <flgroesse>0.0</flgroesse>
   <alk>0.0</alk>
   <rz>0.0</rz>
   <sa>0.0</sa>
</wein>
```

[10] aus Wikipedia: „In der Softwareentwicklung dienen Annotationen dazu, Metadaten in den Quelltext eines Programms einzubinden. Diese haben keine direkte Auswirkung auf die Übersetzung des Quelltextes, bieten aber zusätzliche Möglichkeiten im Vergleich zu einfachen Kommentaren."

[11] siehe https://jaxb.java.net/2.2.11/docs/

`@XmlElementWrapper` Legt fest, dass Elemente verschachtelt werden.

```
@XmlElementWrapper(name = "arten")
public Map<String, Art> getArten() {
   return arten;
}

<arten>
   <entry>
      <key>A-TROCKEN</key>
      <value id="A-TROCKEN" name="trocken"/>
   </entry>
   <entry>
      <key>A-HALBTROCKEN</key>
      <value id="A-HALBTROCKEN" name="halbtrocken"/>
   </entry>
</arten>
```

Beispiele

Root-Element `bsp01` mit Element und Attribut.

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Bsp01 {

   @XmlElement
   String element;

   @XmlAttribute
   String attribut;
}
```

```
<?xml version="1.0" encoding="UTF-8" ⤶
standalone="yes"?>
<bsp01 attribut="Attribut">
    <element>Element</element>
</bsp01>
```

Root-Element `root` mit Element und Attribut, wobei die Namen überschrieben werden.

```
@XmlRootElement(name = "root")
@XmlAccessorType(XmlAccessType.FIELD)
public class Bsp02 {

   @XmlElement(name = "el")
   String element;

   @XmlAttribute(name = "att")
   String attribut;
}
```

```
<?xml version="1.0" encoding="UTF-8" ⤶
standalone="yes"?>
<root att="Attribut">
    <el>Element</el>
</root>
```

Root-Element mit Liste (`List`).

```
@XmlRootElement(name = "root")
@XmlAccessorType(XmlAccessType.FIELD)
public class Bsp03 {

   @XmlElement
   List<String> liste =
      new ArrayList<String>();
}
```

```
<?xml version="1.0" encoding="UTF-8" ⤶
standalone="yes"?>
<root>
    <liste>Apfel</liste>
    <liste>Birne</liste>
    <liste>Banane</liste>
</root>
```

Root-Element mit „eingepackter“ Liste (`List`).

```
@XmlRootElement(name = "root")
@XmlAccessorType(XmlAccessType.FIELD)
public class Bsp03b {

   @XmlElement(name = "liste")
   @XmlElementWrapper(name = "muesli")
   List<String> liste = new ArrayList<⤶
   String>();
}
```

```
<?xml version="1.0" encoding="UTF-8" ⤶
standalone="yes"?>
<root>
    <muesli>
        <liste>Apfel</liste>
        <liste>Birne</liste>
        <liste>Banane</liste>
    </muesli>
</root>
```

Root-Element mit Map (`Map`).

```
@XmlRootElement(name = "root")
@XmlAccessorType(XmlAccessType.FIELD)
public class Bsp04 {

   @XmlElement
   Map<String, String> map =
      new HashMap<String, String>();
}
```

```
<?xml version="1.0" encoding="UTF-8" ⤸
standalone="yes"?>
<root>
  <map>
    <entry>
      <key>A1</key><value>Apfel</value>
    </entry>
    <entry>
      <key>A2</key><value>Birne</value>
    </entry>
    <entry>
      <key>A3</key><value>Banane</value>
    </entry>
   </map>
</root>
```

Root-Element mit Map (`Map`) und `Bsp01`.

```
@XmlRootElement(name = "root")
@XmlAccessorType(XmlAccessType.FIELD)
public class Bsp05 {

   @XmlElement
   Map<String, Bsp01> map = new HashMap<⤸
   String, Bsp01>();
}
```

```
<?xml version="1.0" encoding="UTF-8" ⤸
standalone="yes"?>
<root>
  <map>
    <entry>
      <key>A1</key>
      <value attribut="A1">
         <element>Apfel</element>
      </value>
    </entry>
    <entry>
      <key>A2</key>
      <value attribut="A2">
         <element>Birne</element>
      </value>
    </entry>
    <entry>
      <key>A3</key>
      <value attribut="A3">
         <element>Banane</element>
      </value>
    </entry>
  </map>
</root>
```

Möchte man, dass beispielsweise bei einer `Map` die Einträge nicht mit Element und `key/value` gespeichert werden, so kann man man mit einem `XmlAdapter` arbeiten.

Die Elemente werden dabei in der Klasse `MapElements` zusammengefasst und als Attribute definiert.

```
/** MapElements */
public class MapElements {
   @XmlAttribute      public String key;
   @XmlAttribute      public String value;

   private MapElements() {} // Required by JAXB

   public MapElements(final String key, final String value) {
      this.key = key;
      this.value = value;
   }
}
```

Der `MapAdapter` sorgt dafür, dass die Elemente richtig gespeichert, bzw. geladen werden.

```
/** MapAdapter */
public class MapAdapter extends XmlAdapter<MapElements[], Map<String, String>> {

   public MapAdapter() { }

   @Override
   public MapElements[] marshal(final Map<String, String> arg0) throws Exception {
      MapElements[] mapElements = new MapElements[arg0.size()];
      int i = 0;
      for (Map.Entry<String, String> entry : arg0.entrySet()) {
         mapElements[i++] = new MapElements(entry.getKey(), entry.getValue());
      }
```

```
      return mapElements;
   }

   @Override
   public Map<String, String> unmarshal(final MapElements[] arg0) throws Exception {
      Map<String, String> r = new HashMap<String, String>();
      for (MapElements mapelement : arg0) {
         r.put(mapelement.key, mapelement.value);
      }
      return r;
   }
}
```

Über `@XmlJavaTypeAdapter` wird die Adapter-Klasse festgelegt.

```
@XmlRootElement(name = "root")
@XmlAccessorType(XmlAccessType.FIELD)
public class Bsp06 {

   @XmlJavaTypeAdapter(MapAdapter.class)
   @XmlElement
   Map<String, String> map =
      new HashMap<String, String>();
}
```

```
<?xml version="1.0" encoding="UTF-8" ❯
standalone="yes"?>
<root>
    <map>
        <item key="A1" value="Apfel"/>
        <item key="A2" value="Birne"/>
        <item key="A3" value="Banane"/>
    </map>
</root>
```

XML lesen/schreiben

Damit JAXB weiß, welche Klasse (es muss die Klasse angegeben werden, bei der `@XmlRootElement` angegeben worden ist) gespeichert werden soll, wird dies über die Klasse `JAXBContext` festgelegt. Danach kann die XML-OOP-Struktur über die Klasse `Marshaller` gespeichert werden.

Im nachfolgenden Beispiel wird der XML-Baum „schön“ formatiert, einmal auf der Konsole ausgegeben und dann in der Datei `target/weingut.xml` gespeichert.

```
final JAXBContext context = JAXBContext.newInstance(Weingut.class);
final Marshaller m = context.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(wg, System.out);
m.marshal(wg, new File("target/weingut.xml"));
```

Das Lesen funktioniert sehr ähnlich.

```
final JAXBContext context = JAXBContext.newInstance(Weingut.class);
Weingut wg = (Weingut) context.createUnmarshaller().
                       unmarshal(new File("target/weingut.xml"));
```

4.3.3 JDOM

JDOM ist eine Alternative und Ergänzung zu der offiziellen Java-API und integriert die Vorteile und APIs von SAX und DOM. Wir gehen hier nur auf die Möglichkeiten ein, XML-Dokumente zu lesen und einen bestimmten Inhalt zu erhalten. JDOM bietet weitaus mehr, dies würde aber hier den Bereich sprengen.

Für die nachfolgenden Beispiele wird dabei folgende XML-Datei verwendet.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<A id="1">
   <B1 id="2">
      <C1 id="3" />
      <C2 id="4" />
   </B1>
   <B2 id="5" />
   <B3 id="6">
      <C3 id="7" />
      <C4 id="8">
         <D1 id="9">
            <E1 id="10" />
         </D1>
         <D2 id="11" />
      </C4>
      <C5 id="12" />
   </B3>
</A>
```

Für JDOM und später XPATH sind folgende Bibliotheken notwendig:

- jaxen-1.1.6.jar
- jdom-2.0.6.jar
- xml-apis.jar
- xercesImpl.jar

Dokument einlesen und root-Element ausgeben:

```
// IMPORT org.jdom2.*;
// IMPORT org.jdom2.input.*;
try {
   final SAXBuilder saxBuilder = new SAXBuilder();
   final Document document = saxBuilder.build(new File("../src/xml/demo.xml"));
   final Element root = document.getRootElement();

   System.out.println("root   : " + root.getName());
   System.out.println("mit ID : " + root.getAttributeValue("id"));

} catch (JDOMException | IOException e) {
   System.err.println(e.getMessage());
}
```

```
root   : A
mit ID : 1
```

Die Klasse `SAXBuilder` ist der eigentliche Parser, der die Arbeit im Hintergrund macht. Über die Methode `build()` wird die XML-Datei eingelesen und man erhält ein XML-Dokument, welches durch die Klasse `Document` abgebildet wird.

Über die Methode `getRootElement()` erhält man das Wurzelelement.

Die Klasse `Element` (Auszug):

`getAttribute()`	This returns the attribute for this element with the given name and within no namespace, or null if no such attribute exists.
`getAttributes()`	This returns the complete set of attributes for this element, as a List of Attribute objects in no particular order, or an empty list if there are none.
`getAttributeValue()`	This returns the attribute value for the attribute with the given name and within no namespace, null if there is no such attribute, and the empty string if the attribute value is empty.
`getChild()`	This returns the first child element within this element with the given local name and belonging to no namespace.

`getChildren()`	This returns a List of all the child elements nested directly (one level deep) within this element, as Element objects.
`getChildText()`	Returns the textual content of the named child element, or null if there's no such child.
`getName()`	Returns the (local) name of the element (without any namespace prefix).
`getText()`	Returns the textual content directly held under this element as a string.

Alle direkten Unterelemente von root ausgeben:

```
// IMPORT org.jdom2.*;
// IMPORT org.jdom2.input.*;
try {
   final SAXBuilder saxBuilder = new SAXBuilder();
   final Document document = saxBuilder.build(new File("../src/xml/demo.xml"));
   final Element root = document.getRootElement();

   System.out.println("root   : " + root.getName());
   final List<Element> liste = root.getChildren();
   final Iterator<Element> it = liste.iterator();

   while (it.hasNext()) {
      final Element child = it.next();
      System.out.println("   " + child.getName());
   }
} catch (JDOMException | IOException e) {
   System.err.println(e.getMessage());
}
```

```
root   : A
   B1
   B2
   B3
```

Will man ein bestimmtes Element, welches evtl. tief verschachtelt ist, ist der Aufwand mit JDOM recht hoch. Hier hilft bei der Suche nach bestimmten Elementen XPATH.

4.3.4 XPATH

Die XML Path Language (XPATH) ist eine Abfragesprache, um Teile eines XML-Dokumentes zu adressieren und auszuwerten.

Grundlagen

XPATH betrachtet das XML-Dokument als Baum und hat folgende Bereiche:

Knoten (nodes) stellen die XML-Bereiche dar: Elemente, Attribute , Text, Kommentare, Namensräume, …

Achsen legen fest, welche weitere Knoten ausgewählt werden soll

Prädikate schränken das Suchergebnis weiter ein

Ein XPATH-Ausdruck kann sich aus vielen Bereichen zusammensetzen. Diese werden dann mit „/“ getrennt. Beliebig viele XPATH-Ausdrücke lassen sich mit dem Pipe-Zeichen „ | “ zusammenführen.

Achse	*adressierte Knoten*	*Abkürzung*
	Dokument-Knoten	/
child	direkt untergeordnete Knoten	
parent	der direkt übergeordnete Elternknoten	./..
self	der Kontextknoten selbst (nützlich für zusätzliche Bedingungen)	.
ancestor	übergeordnete Knoten	
ancestor-or-self	übergeordnete Knoten inklusive des Kontextknotens	
descendant	untergeordnete Knoten	.//
descendant-or-self	untergeordnete Knoten inklusive des Kontextknotens	
following	im XML-Dokument nachfolgend (ohne untergeordnete Knoten des selektierten Knotens)	
following-sibling	wie following, aber zugleich vom selben parent stammend	
preceding	im XML-Dokument vorangehend (ohne übergeordnete Knoten)	
preceding-sibling	wie preceding, aber zugleich vom selben parent stammend	
attribute	Attributknoten	@
namespace	Namensraumknoten, die aus dem Attribut xmlns stammen	

Tab. 4.2: Achsen

Code:

```
// IMPORT org.jdom2.*
// IMPORT org.jdom2.input.*
// IMPORT org.jdom2.filter.*
// IMPORT org.jdom2.xpath.*
try {
   final SAXBuilder saxBuilder = new SAXBuilder();
   final Document document = saxBuilder.build(new File("../src/xml/demo.xml"));

   final XPathFactory factory = XPathFactory.instance();
   final XPathExpression<Element> xp = factory.compile("/A/child::*", Filters.element());
   final List<Element> liste = xp.evaluate(document);

   final Iterator<Element> it = liste.iterator();
   while (it.hasNext()) {
      System.out.println(it.next().getName());
   }
} catch (JDOMException | IOException e) {
   System.err.println(e.getMessage());
}
```

```
B1
B2
B3
```

Nachdem das Dokument geladen worden ist, wird die `XPathFactory` erzeugt. Danach wird der XPATH-Ausdruck erzeugt und mit der Methoden `compile()` übersetzt. Als Ergebnis sollen nur Elemente geliefert

werden (mit `Filters.element()`). Mit Hilfe der Methode `evaluate()` wird der XPATH-Ausdruck auf das Dokument angewendet und liefert als Ergebnis eine Liste zurück. Diese kann dann mit Hilfe der Klasse `Iterator` ausgegeben werden.

In nachfolgenden Beispielen wird mit dem Skript `runXPATH` das oben angepasste Java-Programm mit dem XPATH-Parameter aufgerufen und das Ergebnis ausgegeben.

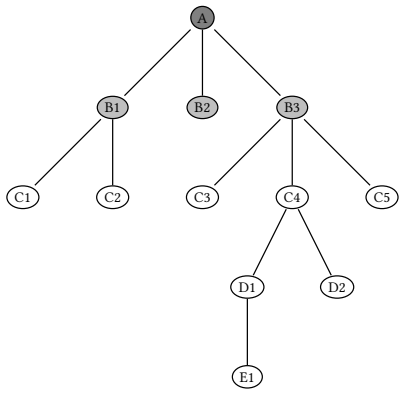
Beispiele:

`child`

Gibt die unmittelbaren untergeordneten Konten aus.
Beispiel: `/A/child::*`

```
sh ./runXPATH '/A/child::*'
```

```
B1
B2
B3
```

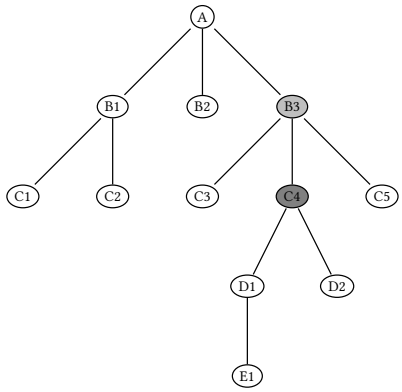


`parent`

Gibt die unmittelbaren Elternknoten aus.
Beispiel: `//C4/parent::*`

```
sh ./runXPATH '//C4/parent::*'
```

```
B3
```

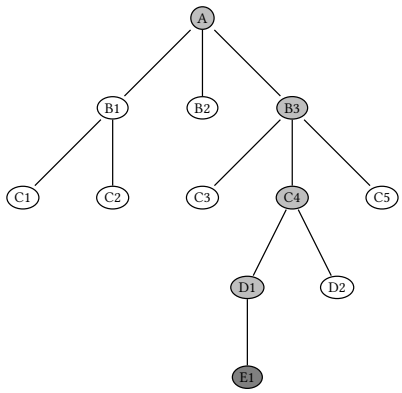


`ancestor`

Gibt die Vorfahren aus.
Beispiel: `//E1/ancestor::*`

```
sh ./runXPATH '//E1/ancestor::*'
```

```
A
B3
C4
D1
```

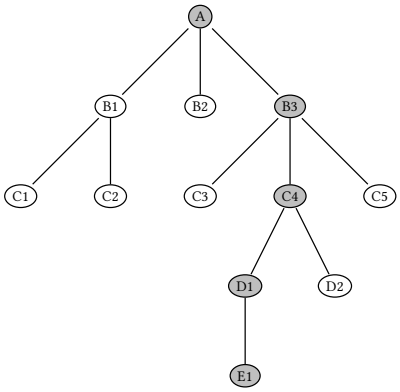


ancestor-or-self

Gibt die Vorfahren und sich selbst aus.
Beispiel: `//E1/ancestor-or-self::*`

```
sh ./runXPATH '//E1/ancestor-or-self::*'
```

```
A
B3
C4
D1
E1
```

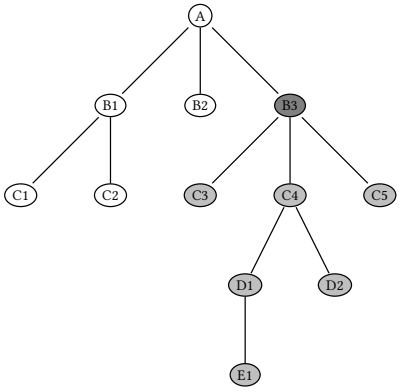


descendant

Gibt alle nachfolgenden Knoten und deren Kinder aus.
Beispiel: `//B3/descendant::*`

```
sh ./runXPATH '//B3/descendant::*'
```

```
C3
C4
D1
E1
D2
C5
```

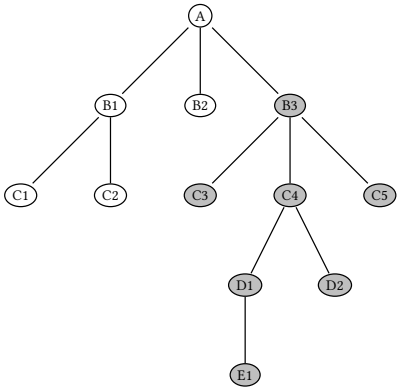


descendant-or-self

Gibt alle nachfolgenden Knoten, sich selbst und deren Kinder aus.
Beispiel: `//B3/descendant-or-self::*`

```
sh ./runXPATH '//B3/descendant-or-self::*'
```

```
B3
C3
C4
D1
E1
D2
C5
```

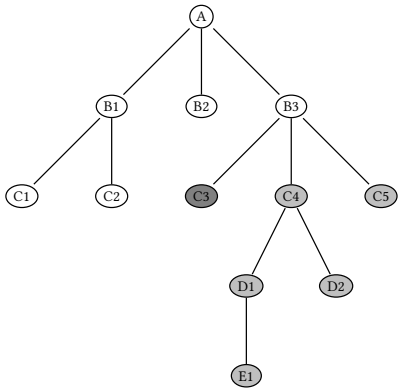


following

Gibt die nachfolgenden Knoten und seine Kinder aus.
Beispiel: `//C3/following::*`

```
sh ./runXPATH '//C3/following::*'
```

```
C4
D1
E1
D2
C5
```

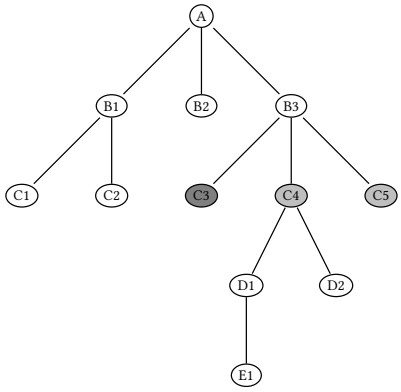


following-sibling

Gibt alle nachfolgenden Knoten in der gleichen Ebene aus.
Beispiel: `//C3/following-sibling::*`

```
sh ./runXPATH '//C3/following-sibling::*'
```

```
C4
C5
```

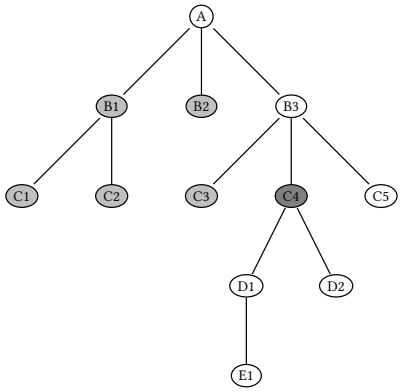


preceding

Gibt die Knoten, die vor dem aktuellen Knoten im XML-Dokument stehen, unabhängig von der Hierarchie der Knoten (ausgeschlossen Vorfahr-Knoten des aktuellen Knotens) aus.
Beispiel: `//C4/preceding::*`

```
sh ./runXPATH '//C4/preceding::*'
```

```
B1
C1
C2
B2
C3
```

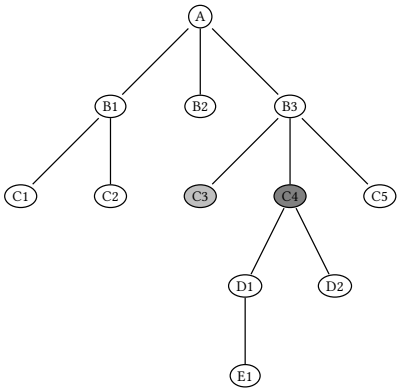


preceding-sibling

Gibt die Knoten, die vor dem aktuellen Knoten auf der gleichen Hierarchie-Ebene stehen, aus.
Beispiel: `//C4/preceding-sibling::*`

```
sh ./runXPATH '//C4/preceding-sibling::*'
```

```
C3
```

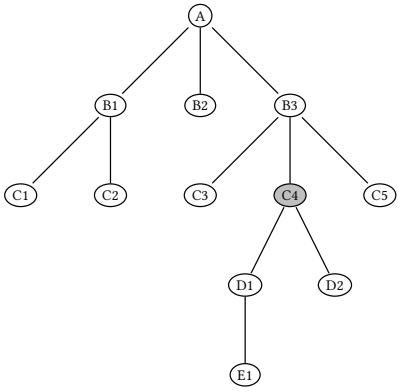


self

Gibt sich selbst aus.
Beispiel: `//C4/self::*`

```
sh ./runXPATH '//C4/self::*'
```

```
C4
```



Knotentypen

Hinter der Angabe der Achse kann nicht nur ein Ausdruck, sondern auch nachfolgende Knotentypen (siehe Tabelle 4.3) verwendet werden.

Knotentyp	*Beschreibung*
`comment()`	Ein Kommentarknoten wird ausgewählt.
`text()`	Ein Textknoten wird ausgewählt.
`processing-instruction()`	Ein PI-Knoten wird ausgewählt.
`node()`	Es wird irgendeiner der obigen Knotentypen ausgewählt.

Tab. 4.3: Knotentypen

Möchte man alle Knoten eines Dokuments auswählen, kann man zum Beispiel die folgenden Knotenmengen vereinigen.

```
/descendant-or-self::node()
/descendant-or-self::node()/attribute::*
/descendant-or-self::node()/namespace::*
```

Knoten

Die Knoten in einer Knotenmenge sind ungeordnet. Allerdings kann über die Funktion `position()` die Position eines Knotens in der aktuellen Knotenliste bestimmt werden. Die Reihenfolge der Knoten orientiert sich an der Reihenfolge, in der die Knoten im XML-Dokument auftreten. Bei vorwärtsgerichteten Achsen wird die Ursprungsreihenfolge beibehalten und man spricht von *Dokumentenordnung*. Für rückwärtsgerichtete Achsen wird die Reihenfolge umgekehrt und man spricht von *umgekehrter Dokumentenordnung*.

Abkürzungen

Da manche Achsenauswahlen sehr oft verwendet werden, hat das W3C bestimmte Abkürzungen festgelegt (Tabelle 4.4).

Abkürzung	***ausgeschriebene Syntax***
.	self::node()
..	parent::node()
name	child::name
@name	attribute::name
//	/descendant-or-self::node()/

Tab. 4.4: Abkürzungen

5 Testgetriebene Entwicklung

Im nächsten Schnitt sollen die Java-Klassen aus dem OOP-Klassenmodell erstellt werden. Damit alle die selbe Datenbasis besitzen, soll hier mit der Methode „Testgetriebene Entwicklung“[1] (engl. test driven development) gearbeitet werden.

Dazu wird ein Eclipse-Projekt nach Vorgaben erstellt, welches mit Hilfe von JUnit Tests erstellt und ausführt, um zu ermitteln, ob die erstellten Klassen den Vorgaben entsprechen.

Gehen Sie dabei wie folgt vor:

- Importieren Sie das Eclipse-Projekt (Datei `eclpise_project_ltxml.zip`) über File-Import-General-Existing Projects into Workspace-Select archive file-...
- Die Java-Quellen liegen in den beiden Verzeichnissen `src/java` (hier die Projektdateien) und `src/text` (hier die Testtreiber).
- Die erzeugten Class-Dateien (default output folder) werden in `target/classes` abgelegt.
- Als JDK-Basis verwenden Sie 1.8, also Java 8 (**Überprüfen Sie diese Einstellung!**).
- Als Endocoding verwenden Sie UTF-8 (**Überprüfen Sie diese Einstellung!**).

5.1 Wie funktioniert ein Testtreiber?

Betrachten wir zunächst den Testtreiber `T_00_KonfessionTest` für die Klasse `Konfession`.

```
/*
 * $Id: T_00_KonfessionTest.java 1576 2015-12-14 12:16:43Z michael $
 */
package de.nm.ltxml.core;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

import org.junit.Test;

/**
 * Test for {@link Konfession}.
 *
 * @version $Revision: 1576 $
 */
public class T_00_KonfessionTest {

   @Test
   public void test_01() {
      final Konfession konf = new Konfession();
      assertNotNull(konf);
      assertTrue(konf.getId() instanceof String);
      assertTrue(konf.getBezeichnung() instanceof String);
      assertEquals("??", konf.getId());
      assertEquals("??", konf.getBezeichnung());
   }

   @Test
   public void test_02() {
      final Konfession konf = new Konfession("k1", "römisch-katholisch");
      assertNotNull(konf);
      assertEquals("k1", konf.getId());
      assertEquals("römisch-katholisch", konf.getBezeichnung());
   }

   @Test
   public void test_03() {
```

[1] siehe https://de.wikipedia.org/wiki/Testgetriebene_Entwicklung

```
39        final Konfession konf = new Konfession("k2", "evangelisch");
40        assertNotNull(konf);
41        assertEquals("k2", konf.getId());
42        assertEquals("evangelisch", konf.getBezeichnung());
43     }
44
45     @Test
46     public void test_toString() {
47        final Konfession konf = new Konfession("k2", "evangelisch");
48        assertNotNull(konf);
49        assertEquals("Konfession: id=k2, bezeichnung=evangelisch",
50                 konf.toString());
51     }
52
53  }
```

Der Testtreiber ist eine normale Klasse, mit der Besonderheit, dass vor die Methoden die Annotation `@Test` geschrieben wird. Wird der Test ausgeführt (über Run as-JUnit Test), so wird jede Methode aufgerufen, vor der ein `@Test` steht[2]. Liefert eine von den vielen `assert`-Methoden einen Fehler, so wird die Methode abgebrochen und mit der nächsten Methode begonnen. In Eclipse wird dies in einer speziellen View angezeigt (siehe Abbildung 5.1). Der grüne Balken sagt aus, dass die Tests erfolgreich waren, ein roter Balken zeigt entsprechende Fehler an.

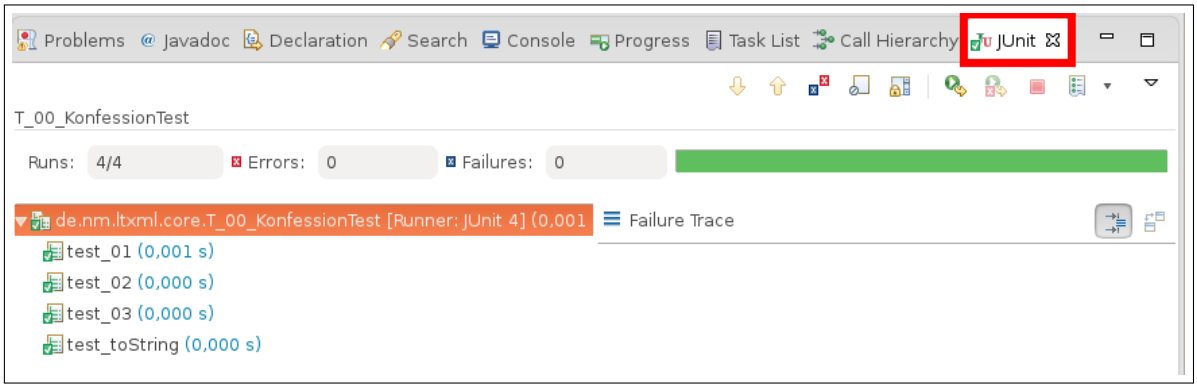


Abb. 5.1: Eclipse-View für JUnit am Beispiel T_00_KonfessionTest

Betrachten wir den ersten Test:

```
19     @Test
20     public void test_01() {
21        final Konfession konf = new Konfession();
22        assertNotNull(konf);
23        assertTrue(konf.getId() instanceof String);
24        assertTrue(konf.getBezeichnung() instanceof String);
25        assertEquals("??", konf.getId());
26        assertEquals("??", konf.getBezeichnung());
27     }
```

Zuerst wird eine Instanz der Klasse `Art` erzeugt. Danach wird über `assertNotNull()` geprüft, ob die Instanz nicht `null` ist. Danach wird geprüft, ob die Methoden `getId()` und `getBezeichnung()` als Typ einen `String` zurückgeben (mit `assertTrue()`) und die Ergebnisse mit `assertEquals()` jeweils `??` sind. Sind die fünf Tests erfolgreich, so wird die Methode als korrekt gekennzeichnet.

Aufgaben → f_ltxml_junit_01

Aufgaben → f_ltxml_test_01

Aufgaben → f_ltxml_test_02

Aufgaben → f_ltxml_test_03

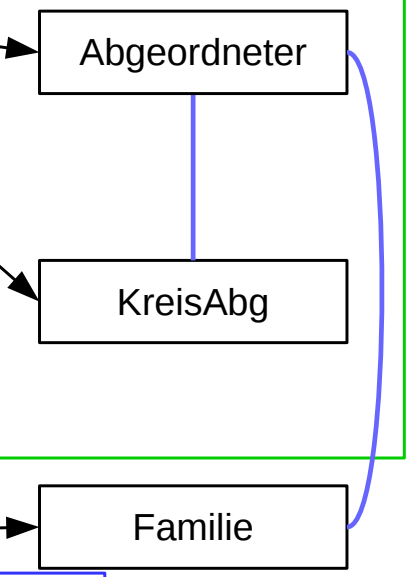
Aufgaben → f_ltxml_test_04

[2] JUnit bietet hier noch sehr viele Möglichkeiten, die aber hier zur Zeit nicht benötigt werden

5.2 XML-Baum

Nachfolgend der XML-Baum und exemplarisch die Beziehungen zwischen den entsprechenden Klassen.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<baylandtag>
    <abgeordnete>
        <abg beruf="Gaukler" fam="f1" … vorname="Pumukel" id="a1">
            <kresiabg>
                <krbg>krbg_1</krbg>
            </kresiabg>
            <ordenabg>
                <orbg>orbg_1</orbg>
            </ordenabg>
            <parfktabg>
                <pfbg>pfbg_1</pfbg>
            </parfktabg>
            <parteiabg>
                <pabg>pabg_1</pabg>
            </parteiabg>
            <staatabg>
                <stbg>stbg_1</stbg>
            </staatabg>
        </abg>
    </abgeordnete>
    <familie>
        <fam bezeichnung="ledig" id="f1"/>
    </familie>
    <konfession>
        <konf bezeichnung="römisch-katholisch" id="k1"/>
    </konfession>
```



■ ■ ■

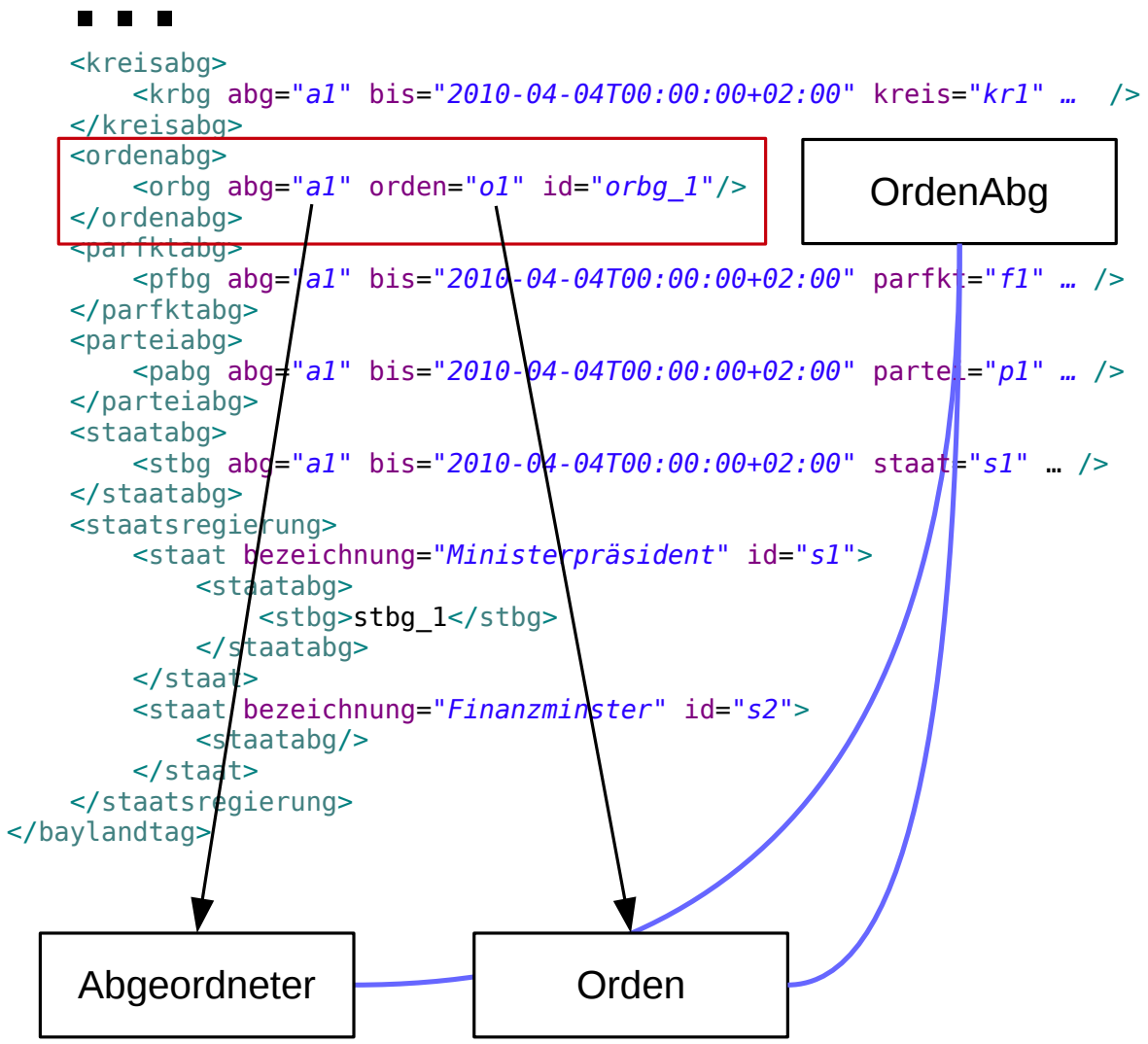
■ ■ ■

```
<kreis>
    <kr type="W" bezeichnung="Oberbayern" id="kr1">
        <kreisabg>
            <krbg>krbg_1</krbg>
        </kreisabg>
    </kr>
    <kr type="S" bezeichnung="Würzburg-Land" id="kr2">
        <kreisabg/>
    </kr>
</kreis>
<orden>
    <ord bezeichnung="Bayerischer Verdienstorden" id="o1">
        <ordenabg>
            <orbg>orbg_1</orbg>
        </ordenabg>
    </ord>
    <ord bezeichnung="Ritterkreuz" id="o2">
        <ordenabg/>
    </ord>
</orden>
<partei>
    <pa name="CSU" bezeichnung="Christlich-Soziale Union in Bayern"
        id="p1">
        <parteiabg>
            <pabg>pabg_1</pabg>
        </parteiabg>
    </pa>
    <pa name="BP" bezeichnung="Bayernpartei" id="p2">
        <parteiabg/>
    </pa>
</partei>
<parfkt>
    <pf bezeichnung="Ältestenrat" id="f1">
        <parfktabg>
            <pfbg>pfbg_1</pfbg>
        </parfktabg>
    </pf>
    <pf bezeichnung="Ausschuß für den Staatshaushalt" id="f2">
        <parfktabg/>
    </pf>
</parfkt>
```

Kreis

KreisAbg

■ ■ ■


...
<kreisabg>
<krbg abg="a1" bis="2010-04-04T00:00:00+02:00" kreis="kr1" … />
</kreisabg>
<ordenabg>
<orbg abg="a1" orden="o1" id="orbg_1"/>
</ordenabg>
<parfktabg>
<pfbg abg="a1" bis="2010-04-04T00:00:00+02:00" parfkt="f1" … />
</parfktabg>
<parteiabg>
<pabg abg="a1" bis="2010-04-04T00:00:00+02:00" partei="p1" … />
</parteiabg>
<staatabg>
<stbg abg="a1" bis="2010-04-04T00:00:00+02:00" staat="s1" … />
</staatabg>
<staatsregierung>
<staat bezeichnung="Ministerpräsident" id="s1">
<staatabg>
<stbg>stbg_1</stbg>
</staatabg>
</staat>
<staat bezeichnung="Finanzminister" id="s2">
<staatabg/>
</staat>
</staatsregierung>
</baylandtag>
OrdenAbg
Abgeordneter
Orden

6 CSV

„Das Dateiformat CSV steht für englisch Comma-separated values (seltener Character-separated values) und beschreibt den Aufbau einer Textdatei zur Speicherung oder zum Austausch einfach strukturierter Daten. Die Dateinamenserweiterung lautet .csv.

Ein allgemeiner Standard für das Dateiformat CSV existiert nicht, jedoch wird es im RFC 4180 grundlegend beschrieben. Die zu verwendende Zeichenkodierung ist ebenso wenig festgelegt; 7-Bit-ASCII-Code gilt weithin als der kleinste gemeinsame Nenner.

In CSV-Dateien können Tabellen oder eine Liste unterschiedlich langer Listen abgebildet werden.“[1]

6.1 Daten einspielen über CSV

Die Daten der Abgeordneten stammen von der öffentlich zugänglichen Seite des Landtages.[2] Mit Hilfe des HTML-Parsers jsoup[3] wurden die Seiten eingelesen, die Struktur ermittelt und dann in einer Datenbank gespeichert.

Dabei wurde nicht bei allen Feldern eine vollständige Normalisierung durchgeführt.

Daraus wurden dann nachfolgende CSV-Dateien erzeugt (hier ein Auszug).

1. `abgeordneter.csv` die Abgeordneten

```
"id","Name","Vorname","Titel","Beruf","geb_am","geb_in","ges_am","ges_in","f_id","k_id"
"1","Ach","Manfred","","Oberregierungsrat a.D.","1940-09-14","Ansbach/Mittelfranken","null","","1","1"
"2","Ackermann","Renate","","Heilerziehungspflegerin","1952-04-26","Ansbach","null","","2","2"
"3","Adametz","Rudolf","","kaufm.Angestellter","1923-03-26","Mährisch-Ostrau","1983-08-19","Kirchseeon","3","3"
"4","Adelberger","Alfons","","Realschulkonrektor","1925-04-30","Obernburg/Main","2000-06-04","","4","4"
```

2. `bild.csv` die Zuordnung der Abgeordneten zu deren Bildern

```
"Name","Vorname","geb_am","geb_in","bild"
"Ach","Manfred","14.09.1940","Ansbach/Mittelfranken","555500000000.jpg"
"Ackermann","Renate","26.04.1952","Ansbach","555500000266.jpg"
"Adametz","Rudolf","26.03.1923","Mährisch-Ostrau","555600002808.jpg"
"Adelberger","Alfons","30.04.1925","Obernburg/Main","555600002810.jpg"
```

3. `familienstand.csv` der Familienstand

```
"id","Bezeichnung"
"44","1 erwachsener Sohn"
"30","1 Kind"
"28","2 Kinder"
"48","3 Kinder"
```

4. `konfession.csv` die Konfession

```
"id","Bezeichnung"
"9","altkatholisch"
"2","evangelisch"
"5","evangelisch-lutherisch"
"8","Freidenker"
```

5. `kreis.csv` die Wahl- bzw. Stimmkreise

```
"id","Type","Bezeichnung"
"418","S","Würzburg-Land"
"419","W","Mittelfranken"
"420","W","Oberbayern"
"421","S","Stkr.Alzenau,Gemünden a.Main,Lohr a.Main/Ufr"
```

6. `orden.csv` die Orden bzw. Auszeichnungen

```
"id","Bezeichnung"
"44","Bayer. Militär-Max-Joseph-Orden"
"26","Bayer.Verfassungsmedaille in Silber"
"29","Bayerische Staatsmedaille für Verdienste um die Umwelt"
"12","Bayerische Verfassungsmedaille in Gold"
```

7. `parfkt.csv` die Parlamentarische Funktionen

---

[1] Quelle https://de.wikipedia.org/wiki/CSV_%28Dateiformat%29
[2] siehe http://www1.bayern.landtag.de/webangebot1/ehem_abgeordnete.maske.jsp
[3] siehe http://jsoup.org/

```
"id","Bezeichnung"
"359","1.stv.Mitglied"
"356","2.stv.Mitglied"
"237","7er-Kommission - Besoldungsordnung"
"173","Aids-Kommission"
```

8. `partei.csv` die Parteien

```
"id","Name","Bezeichnung"
"28","B'90/Grüne","Bündnis 90/Die Grünen"
"29","CSU","Christlich-Soziale Union in Bayern"
"30","FDP","Freie Demokratische Partei"
"31","SPD","Sozialdemokratische Partei Deutschlands"
```

9. `staatsregierung.csv` die Funktionen in der Staatsregierung

```
"id","Bezeichnung"
"62","Arbeitsminister"
"27","Finanzminster"
"3","Innenminister"
"4","Justizminister"
```

10. `zt_fkt_abg.csv` die Beziehung: Parlamentarische Funktionen – Abgeordneter

```
"pa_id","a_id","Typ","von","bis"
"1","1","Mitglied","1994-11-10","1998-09-27"
"2","1","Mitglied","1994-11-10","1998-09-27"
"3","1","Mitglied","1996-12-19","1997-12-17"
"4","1","Vorsitzender","1998-10-28","2003-10-05"
```

11. `zt_kreis_abg.csv` die Beziehung: Kreis – Abgeordneter

```
"k_id","a_id","von","bis"
"418","1","1994-10-17","2008-10-19"
"419","2","2003-10-06","2013-10-07"
"420","3","1970-12-03","1973-03-20"
"421","4","1966-12-02","1970-12-02"
```

12. `zt_ord_abg.csv` die Beziehung: Orden – Abgeordneter

```
"o_id","a_id"
"1","17"
"1","97"
"2","116"
"3","116"
```

13. `zt_partei_abg.csv` die Beziehung: Partei – Abgeordneter

```
"p_id","a_id","von","bis"
"28","2","2003-10-06","2013-10-07"
"28","50","1986-10-22","1990-10-14"
"28","50","2003-10-06","2003-10-06"
"28","110","1990-10-24","1994-10-14"
```

14. `zt_reg_abg.csv` die Beziehung: Staatsregierung – Abgeordneter

```
"s_id","a_id","von","bis"
"1","7","2013-10-10","2013-10-10"
"2","15","1947-01-10","1947-09-20"
"3","15","1947-09-20","1950-12-18"
"4","15","1957-10-16","1958-12-09"
```

6.2 Parser Apache-Commons-CSV

Apache stellt mit der Bibliothek „Commons-CSV“ einen sehr einfachen CSV-Parser bereit.[4]

Nachfolgender Testtreiber zeigt die Funktionsweise.

```
package de.nm.io.csvcommons;

import static junit.framework.TestCase.assertEquals;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.Iterator;

import org.apache.commons.csv.CSVFormat;
import org.apache.commons.csv.CSVParser;
import org.apache.commons.csv.CSVRecord;
import org.junit.BeforeClass;
import org.junit.Test;

/**
 * Test for the apache common cvs parser.
 *
 * @author <a href="mailto:mgn.schule@gmx.de">Michael Niedermair</a>
 * @version $Revision: 1503 $
 */
public class CsvTest {

   @BeforeClass
   public static void init() throws IOException {
      BufferedWriter w = new BufferedWriter(new FileWriter(adr));
      w.write("\"Name\",\"Vorname\",\"Strasse\",\"PLZ\",\"Ort\",\"Alter\"\n"
               + "\"Meier\",\"Peter\",\"Teststr. 12\",12345,\"Testhausen\",18\n"
               + "\"Müller\",\"Uschi\",\"Nimmerfallstr. 3\",\"D-12345\",\"Entenhausen
               \",17\n");
      w.close();
   }

   private static final File adr = new File("target/adr.csv");

   @Test
   public void test01() throws IOException {
      final CSVParser parser = CSVParser.parse(adr, StandardCharsets.UTF_8,
               CSVFormat.EXCEL.withHeader());

      final Iterator<CSVRecord> it = parser.iterator();
      if (it.hasNext()) {
         final CSVRecord record1 = it.next();
         assertEquals("Meier", record1.get("Name"));
         assertEquals("Peter", record1.get("Vorname"));
         assertEquals("Teststr. 12", record1.get("Strasse"));
         assertEquals("12345", record1.get("PLZ"));
         assertEquals("Testhausen", record1.get("Ort"));
         assertEquals("18", record1.get("Alter"));
      }
   }
}
```

Aufgaben → f_ltxml_csv_01

[4] siehe https://commons.apache.org/proper/commons-csv/ → commons-csv-1.1.jar

7 GUI mit Swing

Es gibt viele Speicherformen und Darstellungen, bei denen man eine Baumstruktur oder Tabelle zur Speicherung bzw. Anzeige verwendet. Darunter fallen z. B. Dateisysteme, XML-Dateien, unser Landtag und vieles mehr.

7.1 Erster Baum

Der Baum wird mit der Klasse `JTree` dargestellt. Dabei wird jeder Eintrag in dem Baum mit der Klasse `DefaultMutableTreeNode` angezeigt. Hat der Eintrag Untereinträge, so werden diese nach einem Doppelklick (aufgeklappt) angezeigt.

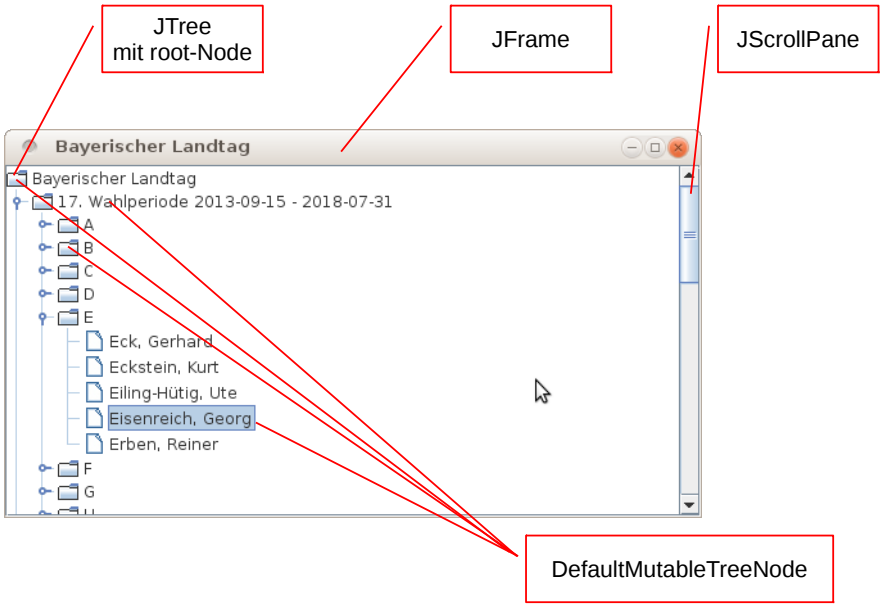


Abb. 7.1: Verwendete Klassen

Die Klasse `DefaultMutableTreeNode` bindet dabei das Interface `MutableTreeNode` und `TreeNode` ein.

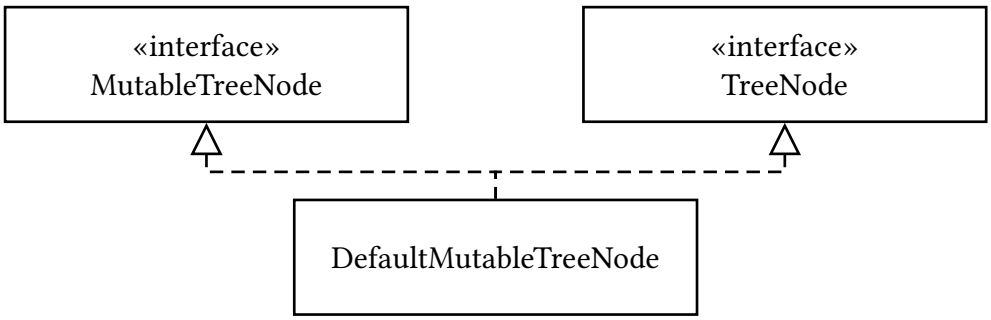


Abb. 7.2: Die Klasse `DefaultMutableTreeNode`

TreeNode

Das Interface `TreeNode` hat dabei folgende wichtige Methoden:

`children()`	Returns the children of the receiver as an Enumeration.
`getAllowsChildren()`	Returns true if the receiver allows children.
`getChildAt(...)`	Returns the child TreeNode at index childIndex.
`getChildCount()`	Returns the number of children TreeNodes the receiver contains.
`getIndex(...)`	Returns the index of node in the receivers children.
`getParent()`	Returns the parent TreeNode of the receiver.
`isLeaf()`	Returns true if the receiver is a leaf.

MutableTreeNode

Das Interface `MutableTreeNode` hat dabei folgende wichtige Methoden:

`insert(...)`	Adds child to the receiver at index.
`remove(...)`	Removes the child at index / node from the receiver.
`removeFromParent()`	Removes the receiver from its parent.
`setParent(...)`	Sets the parent of the receiver to newParent.
`setUserObject(...)`	Resets the user object of the receiver to object.

DefaultMutableTreeNode

Die Klasse `DefaultMutableTreeNode` hat dabei folgende wichtige Methoden:

`add(...)`	Removes newChild from its parent and makes it a child of this node by adding it to the end of this node's child array.
`getDepth()`	Returns the depth of the tree rooted at this node – the longest distance from this node to a leaf.
`getFirstChild()`	Returns this node's first child.
`getFirstLeaf()`	Finds and returns the first leaf that is a descendant of this node – either this node or its first child's first leaf.
`getLastChild()`	Returns this node's last child.
`getLastLeaf()`	Finds and returns the last leaf that is a descendant of this node – either this node or its last child's last leaf.
`getLeafCount()`	Returns the total number of leaves that are descendants of this node.
`getLevel()`	Returns the number of levels above this node – the distance from the root to this node.

Der Code

Betrachten wir den Code: `LtTree.java`

```
23 public class LtTree extends JFrame {
24
25    private static final String BAYERISCHER_LANDTAG = "Bayerischer Landtag";
26
27    /** main */
28    public static void main(final String[] args) throws Exception {
29       new LtTree(Util.readBayLandtag());
30    }
```

Die Klasse wird direkt von `JFrame` abgeleitet. Die Konstante dient als Fenstertitel. In der `main`-Methode wird die XML-Datei geladen und an den Konstruktor übergeben.

```
33    public LtTree(final BayLandtag lt) throws IOException {
34
35       super(BAYERISCHER_LANDTAG);
36       setDefaultCloseOperation(EXIT_ON_CLOSE);
37
38       lt.calculate();
39       final DefaultMutableTreeNode root = new DefaultMutableTreeNode(BAYERISCHER_LANDTAG);
40       GuiUtil.addWahlperiodeTreeNode(root, lt);
41
42       final JTree tree = new JTree(root);
43       final JScrollPane sp = new JScrollPane(tree);
44       sp.setPreferredSize(new Dimension(700, 500));
45
46       add(sp, BorderLayout.CENTER);
47
48       pack();
49       setVisible(true);
50    }
```

Im Konstruktor wird zuerst der Fenstertitel mit Hilfe des Aufrufs `super(...)` realisiert. Danach wird über `setDefaultCloseOperation()` festgelegt, dass bei dem „X“ im Fenstertitel das Programm und damit die Laufzeitumgebung beendet wird.

Anschließend wird das End-Datum berechnet (siehe f_ltxml_csv_02) und der root-Node wird erstellt, der als Text „Bayerischer Landtag“ erhält. Über die Klasse `GuiUtil` werden die Wahlperioden hinzugefügt.

Nun kann der Baum mit Hilfe der Klasse `JTree` erstellt werden. Als Parameter wird hier der root-Node angegeben. Da der Baum größer werden kann, wird dieser in ein `JScrollPane` gepackt, damit bei Bedarf vertikale bzw. horizontale Scrollbalken angezeigt werden können.

Die Größe des `JScrollPane` wird auf 700x500 Pixel gesetzt, dem Frame hinzugefügt und anschließend angezeigt.

Die Klasse `GuiUtil` enthält einige Hilfsmethoden.

```
61    /** add {@link Wahlperiode} to the parent. */
62    public static void addWahlperiodeTreeNode(final DefaultMutableTreeNode parent, final ↲
      BayLandtag lt) {
63
64       final Iterator<Wahlperiode> it = lt.getWpliste().iterator();
65       while (it.hasNext()) {
66          final Wahlperiode wp = it.next();
67          final DefaultMutableTreeNode tn = new DefaultMutableTreeNode(wp.getDisplayString↲
            ());
68          parent.add(tn);
69          addAbgGroupTreeNode(tn, lt, wp);
70       }
71    }
```

Die Methode `addWahlperiodeTreeNode()` fügt die Wahlperioden dem Haupteintrag hinzu. Für jeden Eintrag wird die Methode `addAbgGroupTreeNode()` aufgerufen.

```
30    /** add the first character for grouping */
31    public static void addAbgGroupTreeNode(final DefaultMutableTreeNode parent, final ↲
      BayLandtag lt,
32          final Wahlperiode wp) {
33
34       final List<Abgeordneter> abgliste = getAbg(lt, wp);
35       Collections.sort(abgliste);
36       final Set<Character> setfirstchar = getFirstChar(abgliste);
37
```

```
38      // Gruppieren
39      final Iterator<Character> it = setfirstchar.iterator();
40      while (it.hasNext()) {
41         final Character ch = it.next();
42         final DefaultMutableTreeNode tn = new DefaultMutableTreeNode(ch);
43         parent.add(tn);
44         addAbgTreeNode(tn, abgliste, ch);
45      }
46   }
```

Im nächsten Schritt werden die Abgeordneten einer Wahlperiode mit der Methode `getAbg()` ermittelt und anschließend sortiert. Für die bessere Gruppierung werden nun die vorhandenen Anfangsbuchstaben der Abgeordneten mit der Methode `getFirstChar()` bestimmt. Danach werden die Einträge für den Baum erstellt und die Abgeordneten mit der Methode `addAbgTreeNode()` hinzugefügt.

```
73   /** Return all {@link Abgeordneter} in a {@link Wahlperiode} */
74   public static List<Abgeordneter> getAbg(final BayLandtag lt, final Wahlperiode wp) {
75
76      final List<Abgeordneter> wpliste = new ArrayList<>(200);
77
78      final Iterator<KreisAbg> krabIt = lt.getSetKreisAbg().iterator();
79      while (krabIt.hasNext()) {
80         final KreisAbg krab = krabIt.next();
81
82         // System.out.println(df.format(krab.getVon()) + " " +
83         // df.format(krab.getBis()) + " --> "
84         // + df.format(wp.getVon()) + " " + df.format(wp.getBis()));
85
86         if (krab.getVon().getTime() >= wp.getVon().getTime() && krab.getVon().getTime() ↲
           <= wp.getBis().getTime()) {
87            // von im Bereich enthalten
88            wpliste.add(krab.getAbg());
89         } else if (krab.getBis().getTime() >= wp.getVon().getTime()
90               && krab.getBis().getTime() <= wp.getBis().getTime()) {
91            // bis im Bereich enthalten
92            wpliste.add(krab.getAbg());
93         }
94      }
95      return wpliste;
96   }
```

Die Methode `getAbg()` ermittelt die Abgeordneten einer bestimmten Wahlperiode. Um festzustellen, ob ein Abgeordneter in einer Wahlperiode dem Parlament angehört, wird die Beziehung der Klasse `KreisAbg` verwendet. Ist das Anfangs- bzw. Enddatum im Bereich der Wahlperiode enthalten, so wird der Abgeordnete in die Liste kopiert.

```
98   /**
99    * Return the first character of the name from {@link Abgeordneter} as
100   * {@link Set}
101   */
102  public static Set<Character> getFirstChar(final List<Abgeordneter> abgliste) {
103     final Set<Character> setchar = new TreeSet<>();
104
105     final Iterator<Abgeordneter> it = abgliste.iterator();
106     while (it.hasNext()) {
107        final Abgeordneter abg = it.next();
108        setchar.add(abg.getName().charAt(0));
109     }
110     return setchar;
111  }
```

Die Methode `getFirstChar()` ermittelt für die Gruppierung die vorhandenen Anfangsbuchstaben. Dazu wird für jeden Abgeordneten der Anfangsbuchstabe des Namens ermittelt und der Instanz der Klasse `TreeSet` hinzugefügt. Zur Erinnung: `TreeSet` sortiert die Einträge und sorgt dafür, dass keine doppelten Einträge vorhanden sind (siehe Abschnitt 8.1.3 auf Seite 68).

```
48   /** add the {@link Abgeordneter} with the fitting first char */
49   public static void addAbgTreeNode(final DefaultMutableTreeNode parent, final List<↲
     Abgeordneter> abgliste,
50         final Character ch) {
51      final Iterator<Abgeordneter> it = abgliste.iterator();
52      while (it.hasNext()) {
53         final Abgeordneter abg = it.next();
54         if (abg.getName().charAt(0) == ch) {
```

```
55              final DefaultMutableTreeNode tn = new DefaultMutableTreeNode(abg.
                getDisplayString());
56              parent.add(tn);
57          }
58      }
59  }
```

Die Methode `addAbgTreeNode()` fügt den Abgeordneten dem Baum hinzu, wenn der Anfangsbuchstabe der Gruppierung entspricht.

Aufgaben → f_ltxml_tree_01

7.2 Synchrone Tabellen-Anzeige

Im nächsten Schritt wird die Anzeige erweitert, so dass die Informationen für den Abgeordneten in einer `JTable` angezeigt werden. Grundlagen zur `JTable` finden Sie ab Seite 135.

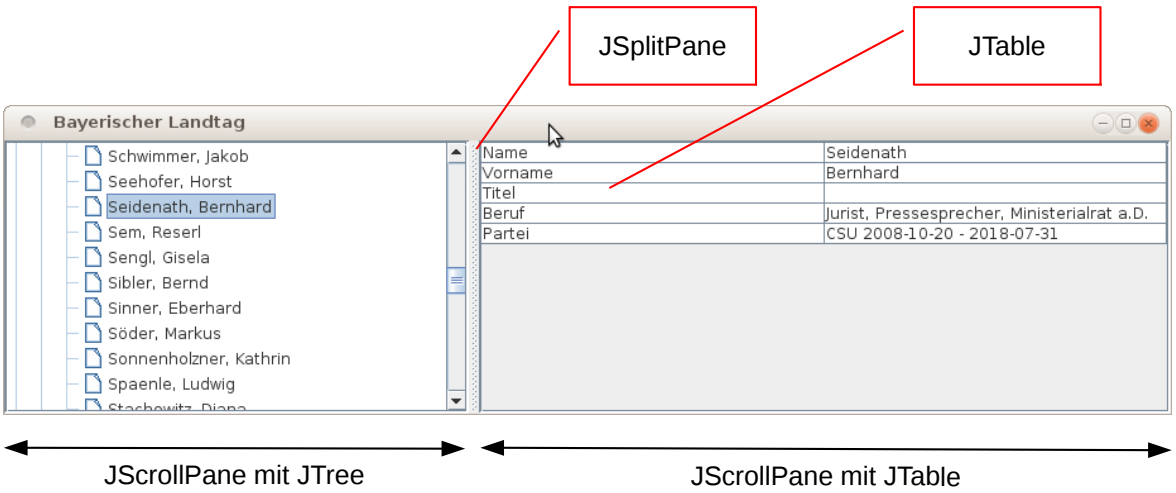


Abb. 7.3: Verwendete Klassen

Dazu sind folgende Schritte notwendig:

1. Aufteilung des Fensters in zwei Bereich mit der Klasse `JSplitPane`.
2. Einbau einer Tabelle mit der Klasse `JTable`.
3. Verwendung von abgeleiteten Klassen von `DefaultMutableTreeNode`.
4. Einbau eines `TreeSelectionListener`, der auf „Klicks“ reagiert und somit die Tabelle aktualisiert.

Der Code

Betrachten wir den Code: `LtTree.java`

```
49      final JTree tree = new JTree(root);
50      final JScrollPane spl = new JScrollPane(tree);
51      spl.setPreferredSize(new Dimension(500, 500));
52
53      tm = new LtTablelModel(lt);
54      final JTable table = new JTable(tm);
55      table.setTableHeader(null);
56      final JScrollPane spr = new JScrollPane(table);
57      spr.setPreferredSize(new Dimension(500, 500));
58
59      final JSplitPane split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, spl, spr);
60      split.setContinuousLayout(true);
61      add(split, BorderLayout.CENTER);
```

Für die Tabelle ist ein Controller notwendig. Hierzu wird die Klasse `de.nm.ltxml.gui.v2.LtTableModel` verwendet, die von `AbstractTableModel` abgeleitet ist. Da die Tabelle keinen Tabellenkopf benötigt, wird dieser über `setTableHeader` ausgeschaltet.

```
64      tree.addTreeSelectionListener(e -> {
65         final Object entry = e.getPath().getLastPathComponent();
66         if (entry instanceof DefaultMutableTreeNode) {
67            if (entry instanceof WahlperiodeMutableTreeNode) {
68               // TODO ...
69               tm.update(null);
70            } else if (entry instanceof GroupMutableTreeNode) {
71               // TODO ...
72               tm.update(null);
73            } else if (entry instanceof AbgeordneterMutableTreeNode) {
74               final AbgeordneterMutableTreeNode tn = (AbgeordneterMutableTreeNode) entry;
75               tm.update(tn.getAbg());
76            } else {
77               tm.update(null);
78            }
79         }
80      });
```

Der `TreeSelectionListener` wird bei jedem „Klick“ auf einen Eintrag im Baum tätig. Dabei wird über `e.getPath().getLastPathComponent()` der Eintrag ermittelt. Der entsprechende Typ bestimmt dabei, was auf der rechten Seite angezeigt werden soll. Über die Methode `update()` wird dem Kontroller der Tabelle der Abgeordnete mitgeteilt oder `null`, wenn „nichts“ angezeigt werden soll.

Betrachten wir den Code:
`de.nm.ltxml.gui.v2.treenode.WahlperiodeMutableTreeNode.java`

```
16 public class WahlperiodeMutableTreeNode extends DefaultMutableTreeNode {
17
18    private final Wahlperiode wp;
19
20    public WahlperiodeMutableTreeNode(final Wahlperiode wp) {
21       super(wp.getDisplayString());
22       this.wp = wp;
23    }
24
25 }
```

Die Klasse wird von `DefaultMutableTreeNode` abgeleitet und übernimmt mit dem Konstruktor jeweils das entsprechende Objekt für die Anzeige. Der anzuzeigende Text wird an den übergeordneten Konstruktor mit Hilfe der Methode `getDisplayString()` weitergegeben. Zusätzlich wird die Objektreferenz gehalten.

Warum eine weitere Klasse, in der fast nichts steht?

Auf den ersten Blick ist die Klasse nicht wirklich sinnvoll. Später, wenn Funktionen erweitert werden, beispielsweise für jeden Bereich ein eigenes Anzeige-Icon, ist diese Variante dann deutlich einfacher und übersichtlicher.

Betrachten wir den Code:
`de.nm.ltxml.gui.v2.treenode.GroupMutableTreeNode.java`

```
14 public class GroupMutableTreeNode extends DefaultMutableTreeNode {
15
16    private final Character firstchar;
17
18    public GroupMutableTreeNode(final Character firstchar) {
19       super(firstchar);
20       this.firstchar = firstchar;
21    }
22
23 }
```

Betrachten wir den Code:
`de.nm.ltxml.gui.v2.treenode.AbgeordneterMutableTreeNode.java`

```
16 public class AbgeordneterMutableTreeNode extends DefaultMutableTreeNode {
17
18    private final Abgeordneter abg;
19
20    public AbgeordneterMutableTreeNode(final Abgeordneter abg) {
21       super(abg.getDisplayString());
22       this.abg = abg;
23    }
24
25    /** @return the abg */
26    public Abgeordneter getAbg() {
27       return abg;
28    }
29
30 }
```

Betrachten wir den Code:
`de.nm.ltxml.gui.v2.GuiUtil.java`

```
64    /** add {@link Wahlperiode} to the parent. */
65    public static void addWahlperiodeTreeNode(final DefaultMutableTreeNode parent, final ⤸
      BayLandtag lt) {
66
67       final Iterator<Wahlperiode> it = lt.getWpliste().iterator();
68       while (it.hasNext()) {
69          final Wahlperiode wp = it.next();
70          final WahlperiodeMutableTreeNode tn = new WahlperiodeMutableTreeNode(wp);
71          parent.add(tn);
72          addAbgGroupTreeNode(tn, lt, wp);
73       }
74    }
```

In der Klasse `GuiUtil` müssen nur die Stellen angepasst werden, an denen die Klasse `DefaultMutableTreeNode` vorkommt. Die Klasse muss durch die oben abgeleitete Klasse ersetzt werden.

Betrachten wir den Code:
`de.nm.ltxml.gui.v2.LtTablelModel.java`

```
14    private static class Entry {
15       String s0 = "";
16       String s1 = "";
17
18       public Entry(final String s0, final String s1) {
19          this.s0 = s0;
20          this.s1 = s1;
21       }
22    }
23
24    private Abgeordneter abg = null;
25    private final ArrayList<Entry> data = new ArrayList<>();
26    private final BayLandtag lt;
27
28    public LtTablelModel(final BayLandtag lt) {
29       this.lt = lt;
30    }
```

Für die Einträge der Tabelle wird ein Datenhaltungsobjekt erzeugt. Dieses enthält zwei Zeichenketten für die beiden Spalten. Die Einträge werden dabei in einer `ArrayList` gespeichert. Zusätzlich wird eine Referenz auf die Klasse `BayLandtag` gehalten, so dass bei späteren Erweiterungen alle Daten verfügbar sind.

```
76    /** update */
77    public void update(final Abgeordneter abg) {
78       this.abg = abg;
79       data.clear();
80       if (abg != null) {
81          data.add(new Entry("Name", abg.getName()));
82          data.add(new Entry("Vorname", abg.getVorname()));
83          data.add(new Entry("Titel", abg.getTitel()));
84          data.add(new Entry("Beruf", abg.getBeruf()));
85          final Iterator<ParteiAbg> it = abg.getSetParteiAbg().iterator();
86          while (it.hasNext()) {
87             data.add(new Entry("Partei", it.next().getDisplayString()));
88          }
89       }
90       fireTableDataChanged();
91    }
```

Mit der Methode `update()` wird dem Kontroller mitgeteilt, dass sich der Abgeordnete geändert hat. Wird hier `null` übergeben, so wird „nichts“ angezeigt. Bei jedem Update wird die Liste gelöscht und mit den aktuellen Daten versorgt. Wichtig ist, dass zum Schluss über die Methode `fireTableDataChanged()` der Tabelle mitgeteilt wird, dass sich die Daten geändert haben. Ansonsten erfolgt keine Änderung der Anzeige der Tabelle.

```
32    @Override
33    public Class<?> getColumnClass(final int columnIndex) {
34       return String.class;
35    }
36
37    @Override
38    public int getColumnCount() {
39       return 2;
40    }
41
42    @Override
```

```
43    public String getColumnName(final int column) {
44       return "";
45    }
46
47    @Override
48    public int getRowCount() {
49       if (abg == null) {
50          return 0;
51       }
52       return data.size();
53    }
54
55    @Override
56    public Object getValueAt(final int rowIndex, final int columnIndex) {
57       final Entry row = data.get(rowIndex);
58       if (columnIndex == 0) {
59          return row.s0;
60       } else if (columnIndex == 1) {
61          return row.s1;
62       }
63       return "??";
64    }
65
66    @Override
67    public boolean isCellEditable(final int rowIndex, final int columnIndex) {
68       return false;
69    }
70
71    @Override
72    public void setValueAt(final Object aValue, final int rowIndex, final int columnIndex) ⤸
      {
73       // ignore
74    }
```

Nun müssen noch die Methoden des Kontrollers mit „Leben“ gefüllt werden. Der besondere Trick ist, dass die Methode `getRowCount()` 0 zurück gibt, wenn `null` übergeben wird. Somit wird die Tabelle nicht angezeigt.

Aufgaben → f_ltxml_tree_02

7.3 Synchrone HTML-Anzeige

Anstelle einer Tabelle kann auch eine `JEditorPane` für die Anzeige von HTML verwendet werden.

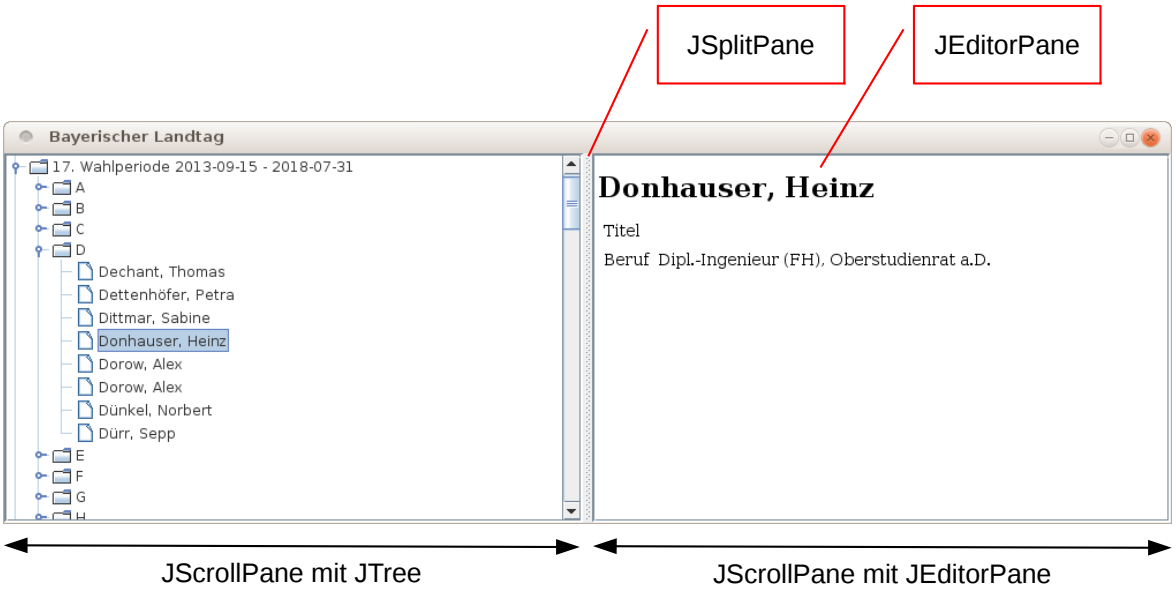


Abb. 7.4: Verwendete Klassen

Der Code

Betrachten wir den Code: `LtTree.java`

```
53      // display HTML
54      editorPane = new JEditorPane("text/html", "");
55      editorPane.setEditable(false);
56      final JScrollPane spr = new JScrollPane(editorPane);
57      spr.setPreferredSize(new Dimension(500, 500));
58
59      final JSplitPane split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, spl, spr);
60      split.setContinuousLayout(true);
61      add(split, BorderLayout.CENTER);
```

Bei der Verwendung der Klasse `JEditorPane` ist es wichtig, hier als MIME-Typ `text/html` zu verwenden.

```
64      tree.addTreeSelectionListener(e -> {
65         final Object entry = e.getPath().getLastPathComponent();
66         if (entry instanceof DefaultMutableTreeNode) {
67            if (entry instanceof WahlperiodeMutableTreeNode) {
68               // TODO ...
69               editorPane.setText("");
70            } else if (entry instanceof GroupMutableTreeNode) {
71               // TODO ...
72               editorPane.setText("");
73            } else if (entry instanceof AbgeordneterMutableTreeNode) {
74               final AbgeordneterMutableTreeNode tn = (AbgeordneterMutableTreeNode) entry;
75               editorPane.setText(GuiUtil.html(tn.getAbg()));
76            } else {
77               editorPane.setText("");
78            }
79         }
80      });
```

Beim `TreeSelectionListener` wird hier über die Methode `setText()` der HTML-Text festgelegt. Dabei wird die Anzeige sofort aktualisiert.

Die Klasse `GuiUtil` wird dann um die Methode `html()` erweitert..

```
117   public static String html(final Abgeordneter abg) {
118      final StringBuilder buf = new StringBuilder(200);
119      buf.append("<h1>").append(abg.getDisplayString()).append("</h1>");
120      buf.append("<table>");
121      buf.append("<tr><td>Titel</td><td>").append(abg.getTitel()).append("</td></tr>");
122      buf.append("<tr><td>Beruf</td><td>").append(abg.getBeruf()).append("</td></tr>");
```

```
123         buf.append("</table>");
124         return buf.toString();
125     }
```

Aufgaben (für Gute) → f_ltxml_tree_03

Aufgaben (für Gute) → f_ltxml_tree_04

8 Anhang

Im Anhang finden sich zusätzliche Unterlagen zu Themen, die Sie schon wissen sollten. Sind Sie sich nicht mehr sicher, ob das Thema bekannt ist und von Ihnen beherrscht wird, so arbeiten Sie die Seiten „zu Hause“ durch.

Inhalt

8.1 Datencontainer ... 60
8.1.1 Arrays ... 60
8.1.2 Listen – dynamische Arrays ... 63
8.1.3 Mengen – Set ... 68
8.1.4 Warteschlangen – Queue ... 69
8.1.5 Enum ... 70
8.1.6 Properties ... 71
8.1.7 Wörterbücher (Maps) ... 72
8.2 Datei-IO ... 74
8.2.1 Dateien ... 74
8.2.2 Dateien mit NIO ... 76
8.2.3 Textdateien ... 79
8.2.4 Binärdateien ... 83
8.2.5 Datenkomprimierung ... 91
8.2.6 Prüfsummen ... 98
8.3 Vererbung und Co. ... 101
8.3.1 Vererbung und Komposition ... 101
8.3.2 Polymorphie ... 111
8.3.3 Generische Programmierung ... 117
8.3.4 Java Generics ... 121
8.3.5 Abstrakte Klassen und Interfaces ... 124
8.4 Einführung in Lambda ... 128
8.4.1 Ausdruck ... 128
8.4.2 Parameterliste ... 129
8.4.3 Lambda-Body ... 130
8.4.4 Lambda Streams ... 131
8.5 Swing ... 135
8.5.1 Grundlagen der Swing-JTable ... 135
8.5.2 Java-JTree ... 144
8.6 „Dies und Das“ ... 152
8.6.1 Klassen finden ... 152
8.6.2 Zahlen formatieren ... 153
8.6.3 SwingX ... 154
8.7 Schülerbeispiele ... 155

8.1 Datencontainer

Ein zentrales Thema der modernen Informatik sind die „Datenstrukturen und Algorithmen“. Hierbei geht es darum, wie Daten vor allem im Speicher abgelegt werden, sodass darauf effizient und schnell zugegriffen werden kann.

Diese Container können wir grob in fünf Arten unterteilen:

Array Ein statisches Array, wobei jedes Element über einen Index angesprochen wird.

List Eine dynamische Liste, wobei jedes Element über einen Index angesprochen wird.

Set Eine Menge enthält jedes Element nur einmal.

Queue Eine Warteschlange nimmt Elemente auf und liefert diese in einer bestimmten Reihenfolge zurück.

Map Eine Map ist eine Menge von Schlüssel/Wert-Paaren (key-value).

8.1.1 Arrays

„Ein Feld bezeichnet in der Informatik eine Datenstruktur.“
(engl. Array für „Anordnung“, „Aufstellung“, „Reihe“, „Reihung“, „Bereich“)

Mit Hilfe eines Arrays können Daten (gleicher Datentyp) im Speicher abgelegt werden, was einen Zugriff über einen Index ermöglicht.

erzeugen

```
Datentyp[] array = new Datentyp[<groesse>];
```

Beispiel:

```
int[] a = new int[10];
int[] b = {1, 5, 7, 9};
String[] c = new String[20];
String[] d = {"Hugo", "Peter", "Uschi"};
```

- Alle Elemente eines Arrays haben den selben Datentyp.
- Die Größe kann nach der Definition nicht mehr geändert werden.
- Die Zuweisung kann auch bei der Definition erfolgen.
- Sehr schneller Zugriff.

Speicherung

Ein Array mit einfachen (primitiven) Datentypen wird wie folgt gespeichert.

```
int[] b = {1, 5, 7, 9};
```

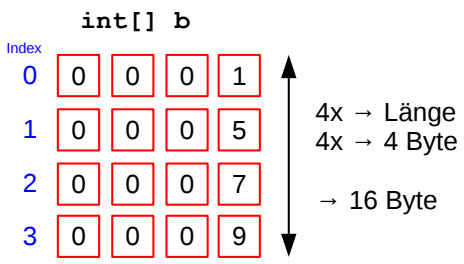


Abb. 8.1: Array

Speicheradresse

Die Speicheradresse lässt sich dabei sehr schnell berechnen und hängt nicht von der Größe des Arrays ab.

Werden komplexe Datentypen (Klassen) verwendet, so wird im Array nur die Referenzadresse des Speicherobjekts gespeichert.

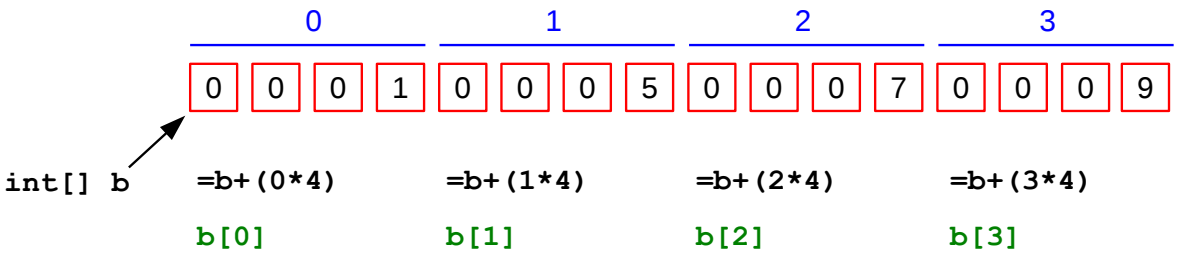


Abb. 8.2: Array im Speicher

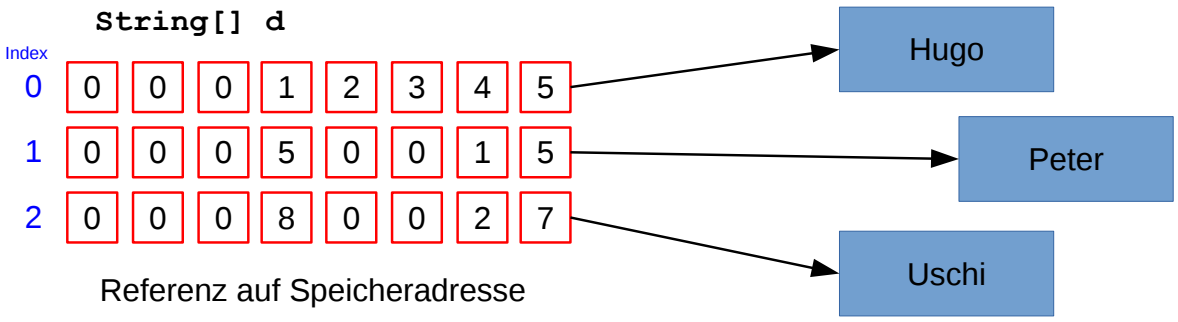


Abb. 8.3: Array im Speicher bei komplexen Datentypen

zuweisen

```
String[] c = new String[20];
c[0] = "Peter";
c[7] = "Uschi";
```

- Der Zugriffsindex beginnt bei 0.
- Einfache Datentypen werden mit ihrem Grundwert (bei `int` z. B. 0) initialisiert.
- Komplexe Datentypen werden mit `null` initialisiert.

auslesen

```
int[] b = {1, 5, 7, 9};

for (int i=0; i < b.length; i++) {
  System.out.println(i + " : " + b[i];
}
```

- Der Zugriffsindex beginnt bei 0.
- Die Länge eines Arrays wird über das Attribut `length` bestimmt.

mit foreach:

```
int[] b = {1, 5, 7, 9};

for (int x: b) {
  System.out.println(x);
}
```

- Es entfällt die Zählvariable.
- Der Wert des Arrayelements wird der Variablen (hier `x`) zugewiesen.

Da sich bei Arrays die Größe nicht ändert (und nur dann!), kann der Schleifentest mit einer Variablen durchgeführt werden.

Beispiel:

```
int[] b = {1, 5, 7, 9};

for (int i=0, n=b.length; i < n; i++) {
  System.out.println(i + " : " + b[i]);
}
```

- Braucht für die Variable `n` mehr Speicher (32 bit).
- Spart beim Durchlauf Zeit, da nicht jedesmal die Arraygröße ermittelt werden muss.

sortieren

Die Klasse `Arrays` bietet u. a. folgende Möglichkeiten:

- Array mit Werten füllen (`Arrays.fill()`)
- Array sortieren (`Arrays.sort()`)
- ...

Beispiel:

```
int[] b = {9, 30, 5, 1, 5, 7, 9};
Arrays.sort(b);
for (int i=0, n=b.length; i < n; i++) {
  System.out.println(i + " : " + b[i]);
}
```

kann wie folgt vereinfacht werden:

```
int[] b = {9, 30, 5, 1, 5, 7, 9};
Arrays.sort(b);
for (int x : b) {
  System.out.println(x);
}
```

Mehrdimensionale Arrays

In Java sind auch mehrdimensionale Arrays möglich.
Diese werden durch Arrays von Arrays realisiert.

Beispiel:

```
int[][] a = new int[ 4 ][ 8 ];
int[][] b = {{1,2}, {2,3,4}, {5, 6}};

System.out.println(b[1][2]);
```

8.1.2 Listen – dynamische Arrays

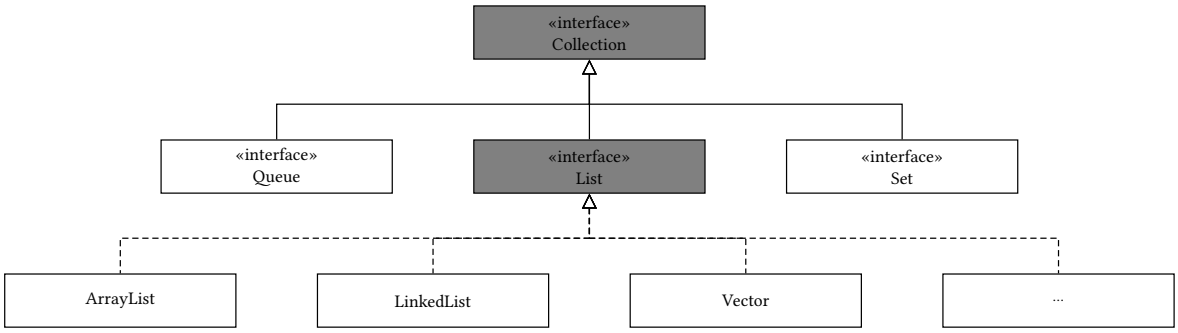
Problem: Array

- Alle Elemente eines Arrays haben den selben Datentyp.
  → Nicht wirklich ein Problem!
- Die Größe kann nach der Definition nicht mehr geändert werden.
  → Sehr großes Problem!
- Kompletter Speicherbedarf (zumindest bei primitiven Typen) wird sofort belegt!
  → Kann zu längeren Ladezeiten führen.

Lösung:

Collections bzw. **Listen**

Übersicht



Interface Collection

- Stellt allgemeine Schnittstellen für „Datensammler“ bereit.
- Die Datensammler werden durch abgeleitete Interfaces genauer spezifiziert.

Auszug:

`add()`	Appends the specified element to the this list.
`addAll()`	Adds all of the elements in the specified collection to this collection.
`clear()`	Removes all of the elements from this collection.
`contains()`	Returns true if this collection contains the specified element.
`isEmpty()`	Returns true if this collection contains no elements.
`iterator()`	Returns an iterator over the elements in this collection.
`parallelStream()`	Returns a possibly parallel Stream with this collection as its source.
`remove()`	Removes a single instance of the specified element from this collection, if it is present.
`size()`	Returns the number of elements in this collection.
`stream()`	Returns a sequential Stream with this collection as its source.
`toArray()`	Returns an array containing all of the elements in this collection.

Interface List

- Stellt eine Schnittstelle für Listen bereit.
- Listen sind Ansammlungen von Datenmengen des gleichen Typs, die über einen Index angesprochen werden.

Auszug:

`add(...)`	Appends the specified element to the end of this list.
`get(int index)`	Returns the element at the specified position in this list.

`indexOf(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`remove(int index)`	Removes the element at the specified position in this list.
`set(int index, E element)`	Replaces the element at the specified position in this list with the specified element.

ArrayList

- Die Klasse `ArrayList` verwaltet intern die Datenmengen mit einem Array, was den Zugriff sehr schnell macht.
- Beim Anlegen der `ArrayList` wird der Speicherplatz für die Anfangsgröße schon belegt.
- Wird die Arraygrenze erreicht, so wird ein neues Array erzeugt (Standardmäßig die doppelte Größe) und alle Elemente werden in das neue Array kopiert, was sehr lange dauern kann.
- Wird ein Arrayelement gelöscht, so werden alle nachfolgenden Elemente nach vorne geschoben, was je nach Position und Größe sehr lange dauern kann.

Definition:

```
List<Typ> liste = new ArrayList<Typ>();
```

- Damit man flexibel ist und die Implementierung der Liste jederzeit austauschen kann, implementiert man gegen das Interface `List` und nicht gegen die Klasse `ArrayList`.
- Damit Typsicherheit gewährleistet wird, wird der zu verwendende Datentyp angegeben.

Beispiel:

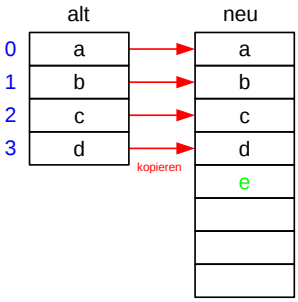
```
List<String> liste = new ArrayList<String>();

liste.add("Peter");
liste.add("Uschi");
liste.add("Hans");

System.out.println("Elemente  : " + liste.size());
System.out.println("Element 2 : " + liste.get(1));
```

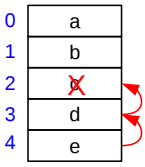
```
Elemente  : 3
Element 2 : Uschi
```

Größe erhöhen:



1. Ein neues Array wird erzeugt (meist doppelte Größe)
2. Die alten Elemente werden kopiert
3. Das neue Element wird hinzugefügt
4. Das alte Array wird zum Löschen freigegeben

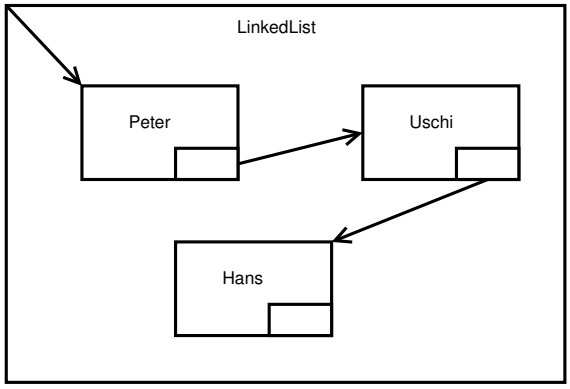
Element löschen:



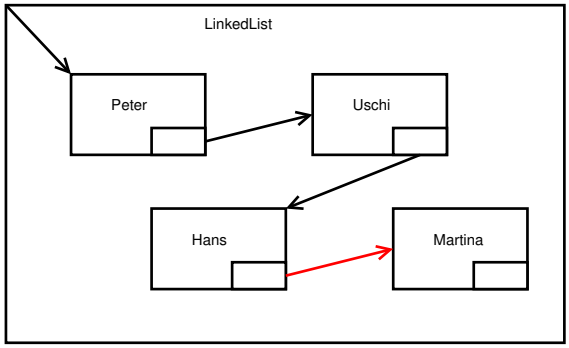
Wird ein Element gelöscht, werden alle „dahinter liegenden“ Elemente um eins „nach vorne“ kopiert.

LinkedList

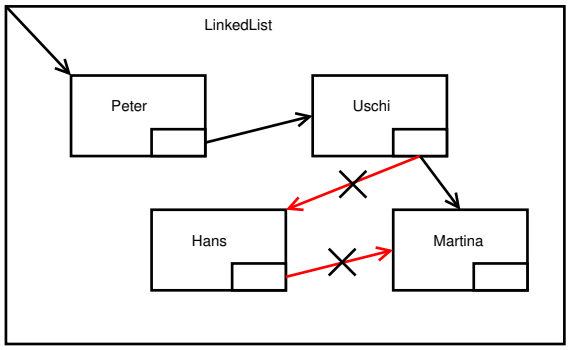
Die `LinkedList` verwendet kein Array als internen Speicher, sondern verkettet die Elemente miteinander.



- Die Liste verweist mit einem Zeiger auf das erste Element.
- Für jeden Eintrag wird das Objekt und ein Zeiger auf das nächste Element gespeichert.



- Wird ein Element hinzugefügt, so wird der Zeiger des letzten Elements auf das neue Element gesetzt.



- Wird ein Element gelöscht, so wird der Zeiger des vorherigen Elements auf das nachfolgende Element gesetzt. Der Zeiger des zu löschenden Elements wird entfernt.
- Alle Objekte, auf die keine Referenz mehr besteht, werden vom Speicherverwaltungssystem (*garbage collector*) automatisch gelöscht.

Vor- / Nachteile gegenüber der ArrayList

- Hinzufügen und Löschen von Elementen geht wesentlich schneller.
- Zu Anfang wird kein Speicher belegt.
- Zugriff auf ein Element ist langsam, da dieses vom Anfang der Kette „gesucht“ werden muss.

Beispiel:

```
List<String> liste = new LinkedList<String>();

liste.add("Peter");
liste.add("Uschi");
liste.add("Hans");

System.out.println("Elemente  : " + liste.size());
System.out.println("Element 2 : " + liste.get(1));
```

```
Elemente  : 3
Element 2 : Uschi
```

Vector

- Die Klasse `Vector` verhält sich wie die Klasse `ArrayList`, jedoch ist der Zugriff komplett synchronisiert, d. h., der Vector kann mit Threads verwendet werden, ohne dass von Hand eine Zugriffsteuerung verwendet werden muss.

- Durch die Synchronisierung ist der `Vector` aber langsamer als die `ArrayList` und sollte daher nur verwendet werden, wenn Parallzugriffe stattfinden.

Test

Analysieren Sie die Programme Test1 (Array), Test2 (Vector), Test3 (ArrayList) und Test4 (LinkedList).

Ruft man diese Programme jeweils mit einer Größe von 5000 auf, so ergeben sich folgende Zahlen:

	erzeugen	summieren	sortieren	suchen
Array	29	12	12339	1
ArrayList	28	14	13306	2
Vector	29	14	15209	1
LinkedList	28	82	869051	1

Iterator

In den obigen Beispielen wurden die „Datentöpfe“ mit einer Zählschleife ausgelesen. Eine andere Variante ist das Verwenden eines Iterators, der die Elemente schrittweise zurück liefert. Dabei muss im „Datentopf“ das Interface `Iterator` implementiert sein, was bei obigen Klassen standardmäßig der Fall ist.

`hasNext()` Returns true if the iteration has more elements.

`next()` Returns the next element in the iteration.

`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Beispiel:

```
final List<String> liste = Arrays.asList("Uschi", "Michael", "Peter", "Heidi");

final Iterator<String> it = liste.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
}
```

```
Uschi Michael Peter Heidi
```

Die Hilfklasse Collections

Das Gegenstück zur Klasse `Arrays` ist bei Collections die Klasse `Collections`.

Auszug:

`addAll()` Adds all of the specified elements to the specified collection.

`binarySearch()` Searches the specified list for the specified object using the binary search algorithm.

`fill()` Replaces all of the elements of the specified list with the specified element.

`replaceAll()` Replaces all occurrences of one specified value in a list with another.

`sort()` Sorts the specified list into ascending order, according to the natural ordering of its elements.

Comparator

„Datentöpfe“ lassen sich über die Hilfsklasse `Collections` sehr einfach sortieren. Standardmäßig wird dabei die natürliche Sortierreihenfolge verwendet. Beispielsweise bei Zahlen aufsteigend und bei Zeichenketten nach dem Unicodecodepoint des Zeichens aufsteigend – dabei wird zwischen Groß- und Kleinschreibung unterschieden.

Beispiel:

```
final List<String> liste = Arrays.asList("Uschi", "Michael", "Peter", "Heidi");

Collections.sort(liste);

final Iterator<String> it = liste.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
}
```

```
Heidi Michael Peter Uschi
```

Möchte man die Sortierreihenfolge ändern, so muss man zusätzlich beim Sortieren einen Comparator angeben, der die Reihenfolge bestimmt.

`compare()` Compares its two arguments for order.

Beispiel:

```
final List<String> liste = Arrays.asList("Uschi", "Michael", "Peter",
               "Heidi", "uschi", "peter");

Collections.sort(liste, new Comparator<String>() {
   @Override
   public int compare(final String s1, final String s2) {
      return s1.compareToIgnoreCase(s2);
   }
});

final Iterator<String> it = liste.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
}
```

```
Heidi Michael Peter peter Uschi uschi
```

Alphabetische Sortierung

Will man eine Alphabetische Sortierung, beispielsweise mit Umlauten, so muss die Klasse `Collator` verwendet werden.

Beispiel:

```
final List<String> liste = Arrays.asList("Straße", "Strasse", "Baum",
               "Bär", "Baer");

final Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.SECONDARY);// a == A, a < Ä

Collections.sort(liste, new Comparator<String>() {
   @Override
   public int compare(final String s1, final String s2) {
      return collator.compare(s1, s2);
   }
});
final Iterator<String> it = liste.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
}
```

```
Baer Bär Baum Straße Strasse
```

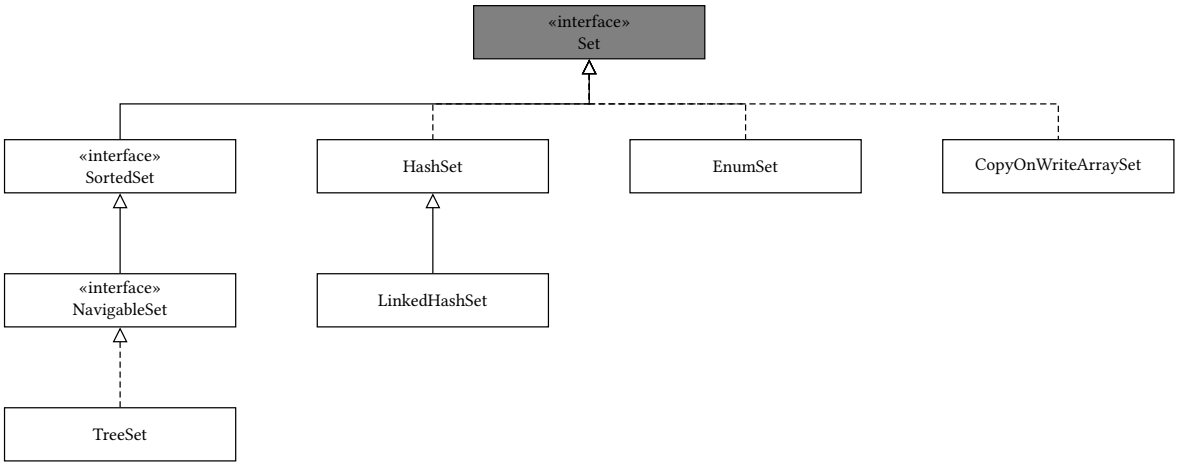
Dabei wird hier `Locale.GERMAN` verwendet, um die Sprache festzulegen. Über die Methode `setStrength()` wird festgelegt, dass beispielsweise 'a' vor 'ä', d. h. wie 'ae' einsortiert wird.

Aufgaben → f_tools_dc_list_01

8.1.3 Mengen – Set

Mengen sind Listen, die aber dafür sorgen, dass jedes Element nur einmal in der Menge auftauchen kann. Diese binden das Interface `Set` ein.

Übersicht (Auswahl)



Die Java-API bietet mehrere konkrete Implementierungen an, z. B.:

`HashSet`	Die Standardklasse für Mengen. `null` kann eingefügt werden. Es gibt keine Ordnung. Sehr schnell, da ein Hashing-Verfahren verwendet wird.
`LinkedHashSet`	Wie `HashSet`, aber die Elemente sind nach der Einfügereihenfolge geordnet.
`TreeSet`	Wie `HashSet`, allerdings werden die Elemente im Container aufsteigend angeordnet. Um die Sortierung festzulegen wird dem `TreeSet`-Konstruktor ein `Comparator`-Objekt übergeben. Wenn die aufzunehmenden Elemente die `Comparable`-Schnittstelle implementieren, kann auf das `Comparator`-Objekt verzichtet werden und die Elemente mit ihrer `compareTo()`-Methode sortiert werden. Intern wird für die Speicherung ein ausbalancierter Binärbaum verwendet.
`CopyOnWriteArraySet`	Schnelle Datenstruktur für viele lesende Operationen. Verwendet intern die Klasse `CopyOnWriteArrayList`
`NavigableSet`	Verleiht dem `TreeSet`-Container zusätzliche Methoden zum Finden von Elementen die kleiner oder größer als ein übergebenes Referenzelement sind.
`EnumSet`	Abstrakte Spezialisierung für Aufzählungstypen (`Enum`). Alle Elemente müssen vom gleichen Aufzählungstypen sein.

Beispiel HashSet

```
final HashSet<String> set = new HashSet<>();
set.add("Uschi");
set.add("Michael");
set.add("Uschi");
set.add("Michael");

final Iterator<String> it = set.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
}
```

```
Uschi Michael
```

Beispiel TreeSet

```
final List<String> liste = Arrays.asList("Uschi", "Michael", "Peter",
               "Heidi", "Uschi", "Michael");

final TreeSet<String> set = new TreeSet<>(liste);

final Iterator<String> it = set.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
}
```

```
Heidi Michael Peter Uschi
```

Beispiel TreeSet: kleiner gleich

```
final List<Integer> liste = Arrays.asList(1, 100, 80, 90, 70, 35, 3, 5);

final TreeSet<Integer> set = new TreeSet<>(liste);

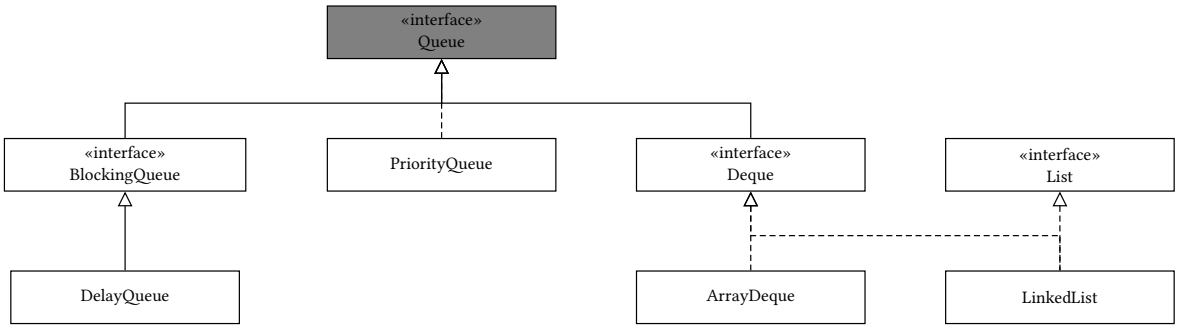
final Iterator<Integer> it = set.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + " ");
}
System.out.println("\nkleiner als 69: " + set.lower(69));
```

```
1 3 5 35 70 80 90 100
kleiner als 69: 35
```

8.1.4 Warteschlangen – Queue

Warteschlangen (Queue) zeichnen sich dadurch aus, dass sie ihre eingefügten Elemente nur in einer bestimmten, für den entsprechenden Container typischen Reihenfolge, zurückgeben.

Übersicht (Auswahl)



FIFO (First In First Out) Prinzipiell funktioniert diese Variante wie eine typische Warteschlange an der Kaufhauskasse. Man stellt sich hinten an und wer sich zuerst angestellt hat kommt auch als Erstes dran.

LIFO (Last In First Out) Ein LIFO-Beispiel aus dem alltäglichen Leben ist beispielsweise eine Umzugskiste in der Bücher gestapelt werden. Das Buch das als Letztes oben auf den Stapel gelegt wurde wird später als Erstes wieder herausgenommen.

Prioritätenwarteschlange Elementen werden Prioritätsstufen zugeordnet. Elemente mit höherer Priorität werden vor Elementen mit niedrigerer Priorität zurückgegeben. Im Alltag wäre diese Warteschlange in etwa mit einem Wartezimmer beim Arzt zu vergleichen. Patienten kommen in der Reihenfolge dran wie sie Termine vereinbart haben. Und Notfälle haben oberste Priorität.

Double Ended Queue (Deque) Es kann sowohl am Anfang als auch am Ende gelesen und geschrieben werden. Hierzu existiert eine eigene Schnittstelle die von `Queue` abgeleitet ist und spezielle Methoden für den beidseitigen Zugriff definiert: `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst()`, `removeLast()`, `peekFirst()`, `peekLast()`, `pollFirst()`, `pollLast()`, `push()`, `pop()`

Die Java-API bietet mehrere konkrete Warteschlangenimplementierungen an, z. B.:

`LinkedList` Diese Klasse implementiert nicht nur die List- sondern auch die Queue-Schnittstelle. Genauer die von Queue abgeleitete Deque-Schnittstelle. Greift man über die Methoden der Deque-Schnittstelle auf einen Linked-List-Container zu, verhält er sich wie eine Double Ended Queue.

`PriorityQueue` Implementierung einer Prioritätenwarteschlange. Die Prioritätsstufen können von den Elementen selbst oder vom Container festgelegt werden. Im ersten Fall muss die Klasse, der die Elemente angehören, die Schnittstelle `Comparable` implementieren. Soll der Container selbst die Elemente priorisieren, wird dem Konstruktor ein Vergleichsobjekt übergeben, das die Schnittstelle `Comparator` für den Typ der Elemente implementiert. Bei Elementen mit gleicher Priorität ist die Reihenfolge willkürlich.

`ArrayDeque` Hierbei handelt es sich um die Standardimplementierung einer `Deque`. Eine Warteschlange bei der man sowohl am Anfang als auch am Ende lesen und schreiben kann.

`DelayQueue` Queue, der die Elemente erst nach einer gewissen Zeit entnommen werden können.

Beispiel

```
final LinkedList<String> queue = new LinkedList<>();
queue.add("Uschi");
queue.add("Michael");
queue.add("Peter");
queue.add("Hans");

while (!queue.isEmpty()) {
   System.out.print(queue.removeFirst() + " ");
}
```

```
Uschi Michael Peter Hans
```

8.1.5 Enum

Enum sind konstante Aufzählungen („Datentöpfe“), die im Code als „Konstanten“ verwendet werden können.

Beispiel Wochentag

```
public enum WocheTag {
      MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG
}
```

```
final WocheTag x = WocheTag.MONTAG;
if (x == WocheTag.MONTAG) {
   System.out.println("Heute ist Montag!");
}
```

```
Heute ist Montag!
```

Vorteile:

- Vergleich mit '==’
- können bei switch-case verwendet werden
- Referenzen können auch `null` sein

8.1.6 Properties

Die Klasse `Properties` ist eine Sonderform der Klasse `Hashtable`, bei der Schlüssel-Werte-Paare immer vom Typ `String` sind. Die Einträge lassen sich in einer Datei speichern und wieder auslesen.

Beispiel

```
try {
   final File fp = new File("my.properties");

   final Properties props = new Properties();
   props.setProperty("User", "Pumukel");
   props.setProperty("Version", "0.02");

   props.store(new FileWriter(fp), "Beispiel Properties");

   // wieder lesen
   final Properties props2 = new Properties();
   props2.load(new FileReader(fp));
   System.out.println(props2.getProperty("User"));

} catch (final IOException e) {
   System.out.println(e.getMessage());
}
```

```
Pumukel
```

Die erzeugte Datei sieht dabei wie folgt aus:

```
#Beispiel Properties
#Tue Nov 17 12:16:51 CET 2015
Version=0.02
User=Pumukel
```

Verwendet man anstelle von `store()` die Methode `storeToXML()`, so wird keine Text- sondern eine XML-Datei erstellt. Zum Lesen verwendet man dann `loadFromXML()`.

Die XML-Datei sieht dann wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Beispiel Properties</comment>
<entry key="Version">0.02</entry>
<entry key="User">Pumukel</entry>
</properties>
```

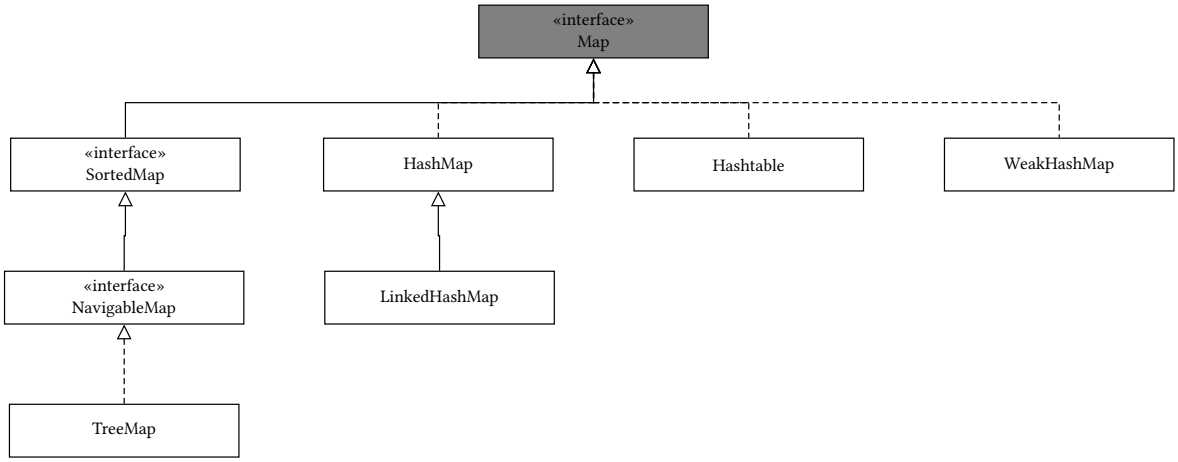
8.1.7 Wörterbücher (Maps)

Ein Wörterbuch ist ein besonderer Container (assoziativer Speicher) zur Ablage von Schlüssel/Wert-Paaren. Mithilfe des Schlüssels kann der zugehörige Wert gefunden werden. Die Suchzeit ist dabei unabhängig von der Anzahl der verwalteten Elemente.

Dies wird durch sogenanntes Hashing erreicht. Aus dem Schlüssel wird nach einem besonderen Algorithmus eine Integer-Zahl (Hashwert) berechnet, die als Index in einer Tabelle dient. An der berechneten Position steht dann der gesuchte Wert.

Wichtig dabei ist, dass der verwendete Hashing-Algorithmus gewährleistet, dass verschiedene Schlüssel zu verschiedenen Zahlen und somit Positionen in der Tabelle führen.

Übersicht (Auswahl)



Interface Map

Auszug:

`clear()`	Removes all of the mappings from this map.
`containsKey()`	Returns true if this map contains a mapping for the specified key.
`containsValue()`	Returns true if this map maps one or more keys to the specified value.
`equals()`	Compares the specified object with this map for equality.
`get()`	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
`hashCode()`	Returns the hash code value for this map.
`isEmpty()`	Returns true if this map contains no key-value mappings.
`keySet()`	Returns a Set view of the keys contained in this map.
`put()`	Associates the specified value with the specified key in this map.
`remove()`	Removes the mapping for a key from this map if it is present.
`replace()`	Replaces the entry for the specified key only if it is currently mapped to some value.
`size()`	Returns the number of key-value mappings in this map.
`values()`	Returns a Collection view of the values contained in this map.

Die Java-API bietet mehrere konkrete Implementierungen an, z. B.:

`Hashtable`	Ist Thread-sicher, erlaubt aber keine null-Schlüssel oder null-Werte.
`HashMap`	Ist nicht Thread-sicher, erlaubt aber auch null als Schlüssel oder Wert.
`LinkedHashMap`	Wie `HashMap`, zusätzlich bleibt die Reihenfolge der Schlüssel nach ihrem zeitlichen Einfügen erhalten.
`WeakHashMap`	Wie `HashMap`, aber gespeicherte Schlüssel können vom Garbage Collector entsorgt werden, wenn im Programm selbst der Schlüssel nicht mehr referenziert wird.

`TreeMap` Wie `HashMap`, allerdings werden die Elemente aufsteigend sortiert. Um die Sortierung festzulegen muss dem `TreeMap`-Konstruktor ein passendes `Comparator`-Objekt übergeben werden. Wenn die aufzunehmenden Elemente die `Comparable`-Schnittstelle implementieren, kann auf das `Comparator`-Objekt verzichtet werden und die Elemente mit ihrer `compareTo()`-Methode sortiert werden.

`NavigableMap` Die Schnittstelle verleiht der `TreeMap` zusätzliche Methoden zum Finden von Elementen oder Schlüsseln, die kleiner oder größer als ein übergebener Referenzschlüssel ist.

`EnumMap` Spezialisierung für Aufzählungstypen. Alle verwendeten Schlüssel müssen aus einer Aufzählung stammen.

Beispiel

```
final TreeMap<String, String> weine = new TreeMap<>();
weine.put("14-71", "Dunkelfelder");
weine.put("13-77", "Hower`s Cuvèe");
weine.put("11-78", "Amethyst");

final Iterator<String> it = weine.keySet().iterator();
while (it.hasNext()) {
   final String key = it.next();
   System.out.println(key + " -> " + weine.get(key));
}
```

```
11-78 -> Amethyst
13-77 -> Hower`s Cuvèe
14-71 -> Dunkelfelder
```

8.2 Datei-IO

8.2.1 Dateien

Für das Arbeiten mit Dateien/Verzeichnissen stellt Java die Klasse `File` zur Verfügung.

```
File f1 = new File(".");
File f2 = new File("/tmp");
File f3 = new File("../images/xxx");
```

- Ein konkretes File-Objekt repräsentiert einen Datei- oder Verzeichnisnamen im Dateisystem.
- Der Verweis wird durch einen Pfadnamen spezifiziert.
- Dieser kann absolut oder relativ zum aktuellen Verzeichnis angegeben werden.
- Die Angabe des Pfades ist plattformabhängig.
  - Windows: ein Backslash „\ “ trennt die Pfade
  - Linux: ein Slash „/“ trennt die Pfade
- Die Konstante `separatorChar` liefert den korrekten Wert für das aktuelle Betriebssystem.
- Evtl. Fehler (keine Schreibrechte, ...) werden dabei mit `IOException` in einem `try-catch`-Block abgefangen.

Methoden (Auszug)

`createNewFile()`	Erzeugt eine neue Datei.
`delete()`	Löscht eine Datei.
`exists()`	Testet, ob die Datei, das Verzeichnis vorhanden ist.
`getAbsolutePath()`	Gibt den absoluten Pfadnamen zurück.
`getName()`	Gibt den Namen zurück.
`getPath()`	Gibt das Verzeichnis zurück.
`isDirectory()`	Ermittelt, ob das Objekt ein Verzeichnis ist.
`isFile()`	Ermittelt, ob das Objekt eine Datei ist.
`length()`	Gibt die Dateigröße in Byte zurück.
`list()`	Gibt ein Array (`String`) mit allen Einträgen des Verzeichnisses zurück.
`listFiles()`	Gibt ein Array (`File`) mit allen Einträgen des Verzeichnisses zurück.
`mkdir()`	Erzeugt ein neues Verzeichnis.

Beispiele:

Datei nicht vorhanden

```
File nichtda = new File("gibt_es_nicht");
System.out.println("Datei "  + nichtda.getName() + " : "
   + (nichtda.exists() ? "vorhanden" : "nicht vorhanden"));
```

```
Datei gibt_es_nicht : nicht vorhanden
```

Datei erzeugen

```
try {

   File neu = new File("neue_datei.txt");
   neu.createNewFile();
   System.out.println("Datei "  + neu.getName() + " : "
      + (neu.exists() ? "vorhanden" : "nicht vorhanden"));

} catch (IOException e) {
   System.out.println("Fehler: " + e.getMessage());
}
```

```
Datei neue_datei.txt : vorhanden
```

Verzeichnisinhalt anzeigen

```
File f = new File("../images/");
System.out.println(f.getAbsolutePath());

File[] files = f.listFiles();
if (files != null) {
   int cnt=0;
   for (File file : files) {
      if (file.isDirectory()) {
         System.out.print("  D: ");
      } else {
         System.out.print("  F: ");
      }
      System.out.println(file.getName());
      if (cnt++ >20) {System.out.println("..."); break;}
   }
}
```

```
/var/local/daten/dbgui/target/cp/../images
  F: RA1_traube_06.png
  F: ltxml_oop_abg_04.pdf
  F: ltxml_tree_01.png
  F: javalinkedlist02.eps
  F: go_r.jpg
  F: javalinkedlist02.pdf
  F: POLAROID_wein_spruch_05.png
  F: RA2_wein_03.png
  F: 03_24_formular_entwurf_felder8.png
  F: 03_06_formular_entwurf.png
  F: kryonet_chat.png
  F: wein_spruch_01.jpg
  F: 03_21_formular_entwurf_felder5.png
  F: db_mv_jt_02.png
  F: swing_copywindow_beschriftung.pdf
  F: 5_1_berichte.png
  F: swing_copywindow.png
  F: 03_05_formular_assistent2.png
  F: db_tmitglied_browser.png
  F: notenliste.png
  F: java_reader.pdf
  F: swing_todolist_beschriftung.pdf
...
```

8.2.2 Dateien mit NIO

Die in Abschnitt 8.2.1 vorgestellten Konzepte gibt es seit Urbeginn von Java. Um weiteren Anforderungen gerecht zu werden, wurde mit Java 7 das „More New I/O APIs for the JavaTM Platform“ kurz „NIO.2“ erschaffen, welches vom bisherigen System `java.io` sehr unabhängig ist. Ein wesentlicher Vorteil gegenüber dem alten System ist, dass bei vielen Methoden mit Rückgabewerten festgelegt wird, ob die Operation erfolgreich war, anstelle einen Fehler zu erzeugen.

Die beiden wichtigsten Klassen sind dabei `FileSystems`, `FileSystem`, `Path` und `Files`:

`FileSystems` Enthält statische Methoden, um das aktuelle Dateisystem zu erhalten oder neue zu erstellen.

`getDefault()`	Returns the default FileSystem.
`getFileSystem(...)`	Returns a reference to an existing FileSystem.
`getFileSystem(...)`	Returns a reference to an existing FileSystem.

`FileSystem` Damit wird ein abstraktes Dateisystem beschrieben. Dies kann ein konkretes Dateisystem, wie von einer Festplatte sein oder beispielsweise ein Zip-Archiv.

`close()`	Closes this file system.
`getPath(...)`	Converts a path string, or a sequence of strings that when joined form a path string, to a Path.
`getSeparator()`	Returns the name separator, represented as a string.
`isOpen()`	Tells whether or not this file system is open.
`isReadOnly()`	Tells whether or not this file system allows only read-only access to its file stores.
`newWatchService()`	Constructs a new WatchService (optional operation).
`provider()`	Returns the provider that created this file system.

`Path` Damit wird ein Pfad zu einer konkreten Datei oder einem Verzeichnis definiert. Dabei wird genauso wie bei der Klasse `File` mit relativen oder absoluten Pfadangaben gearbeitet.

`compareTo(...)`	Compares two abstract paths lexicographically.
`endsWith(...)`	Tests if this path ends with the given path.
`getFileName()`	Returns the name of the file or directory denoted by this path as a Path object.
`getFileSystem()`	Returns the file system that created this object.
`getName(...)`	Returns a name element of this path as a Path object.
`getNameCount()`	Returns the number of name elements in the path.
`getParent()`	Returns the parent path, or null if this path does not have a parent.
`getRoot()`	Returns the root component of this path as a Path object, or null if this path does not have a root component.
`isAbsolute()`	Tells whether or not this path is absolute.
`startsWith(...)`	Tests if this path starts with the given path.
`subpath(...)`	Returns a relative Path that is a subsequence of the name elements of this path.
`toAbsolutePath()`	Returns a Path object representing the absolute path of this path.
`toFile()`	Returns a File object representing this path.
`toRealPath(...)`	Returns the real path of an existing file.
`toUri()`	Returns a URI to represent this path.

`Files` Enthält statische Methoden zum Kopieren, Umbenenen, Löschen, ...

`copy(...)`	Copy ...
`createDirectories(...)`	Creates a directory ...
`createDirectory(...)`	Creates a new directory.
`createFile(...)`	Creates a new and empty file, failing if the file already exists.
`createLink(...)`	Creates a new link (directory entry) for an existing file (optional operation).
`createSymbolicLink(...)`	Creates a symbolic link to a target (optional operation).
`createTempDirectory(...)`	Creates a new directory ...
`createTempFile(...)`	Creates a new empty file in the specified directory ...
`delete(...)`	Deletes a file.
`deleteIfExists(...)`	Deletes a file if it exists.
`exists(...)`	Tests whether a file exists.
`getAttribute(...)`	Reads the value of a file attribute.
`getLastModifiedTime(...)`	Returns a file's last modified time.
`getOwner(...)`	Returns the owner of a file.
`getPosixFilePermissions(...)`	Returns a file's POSIX file permissions.
`isDirectory(...)`	Tests whether a file is a directory.
`isExecutable(...)`	Tests whether a file is executable.
`isHidden(...)`	Tells whether or not a file is considered hidden.
`isReadable(...)`	Tests whether a file is readable.
`isRegularFile(...)`	Tests whether a file is a regular file with opaque content.
`isSameFile(...)`	Tests if two paths locate the same file.
`isSymbolicLink(...)`	Tests whether a file is a symbolic link.
`isWritable(...)`	Tests whether a file is writable.
`lines(...)`	Read all lines from a file as a Stream.
`list(...)`	Return a lazily populated Stream, the elements of which are the entries in the directory.
`move(...)`	Move or rename a file to a target file.
`newBufferedReader(...)`	Opens a file for reading, returning a BufferedReader to read text from the file ...
`newBufferedWriter(...)`	Opens or creates a file for writing, returning a BufferedWriter that may be used to write text to the file ...
`newInputStream(...)`	Opens a file, returning an input stream to read from the file.
`newOutputStream(...)`	Opens or creates a file, returning an output stream that may be used to write bytes to the file.
`notExists(...)`	Tests whether the file located by this path does not exist.
`readAllBytes(...)`	Reads all the bytes from a file.
`readAllLines(...)`	Read all lines from a file.
`setAttribute(...)`	Sets the value of a file attribute.
`setLastModifiedTime(...)`	Updates a file's last modified time attribute.
`setOwner(...)`	Updates the file owner.
`setPosixFilePermissions(...)`	Sets a file's POSIX permissions.
`size(...)`	Returns the size of a file (in bytes).
`write(...)`	Writes bytes to a file.

Beispiele

```
FileSystem fs = FileSystems.getDefault();
System.out.println(fs);
```

```
sun.nio.fs.LinuxFileSystem@15db9742
```

```
try {
   FileSystem fs = FileSystems.getDefault();
   Path p = fs.getPath("neue_datei2.txt");
   Files.deleteIfExists(p);
   Files.createFile(p);
   System.out.println("Datei :"  + p.toAbsolutePath());
   System.out.println("Owner :"  + Files.getOwner(p));
   System.out.println("Size  :"  + Files.size(p));
} catch (IOException e) {
   System.out.println("Fehler: " + e.getMessage());
}
```

```
Datei :/var/local/daten/dbgui/target/cp/neue_datei2.txt
Owner :mgn
Size  :0
```

8.2.3 Textdateien

Für Textdateien stehen die Klassen `Reader` zum Lesen und `Writer` zum Schreiben zur Verfügung.

Textdateien schreiben

Für das Schreiben in eine Textdatei wird die abstrakte Klasse `Writer` bzw. deren Ableitungen verwendet.

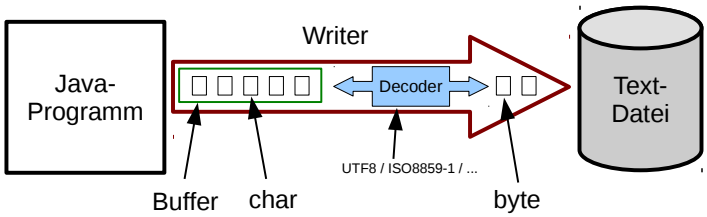
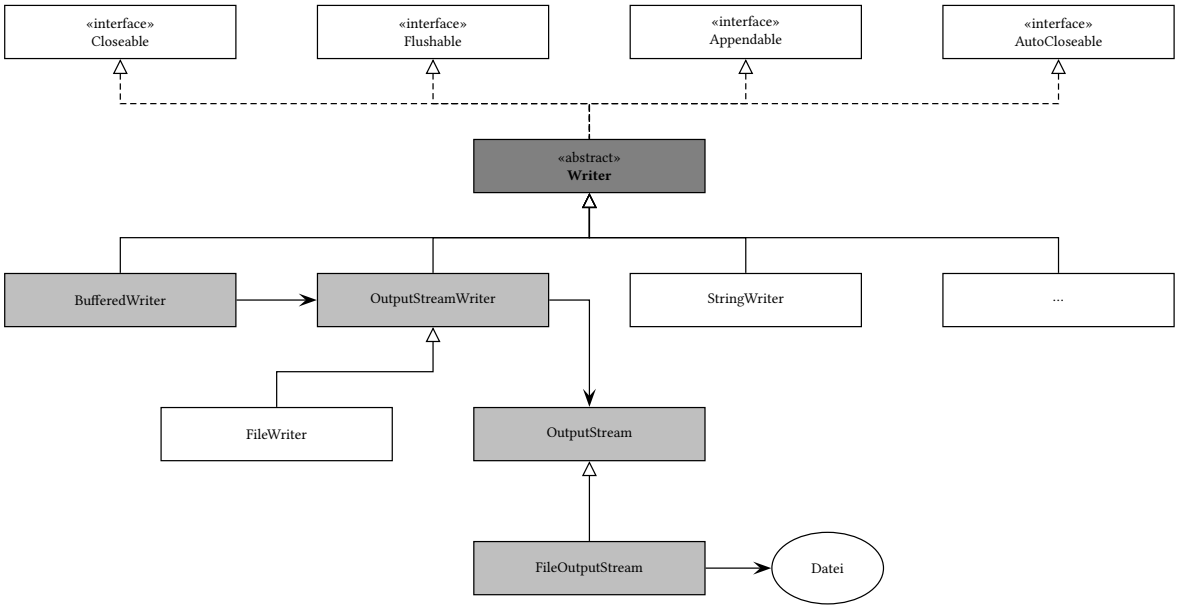


Abb. 8.4: Schreiben in eine Textdatei

Damit das Schreiben effektiver funktioniert, wird zusätzlich ein Buffer verwendet, der die geschriebenen Zeichen blockweise weiter gibt. Standardmäßig hat der Puffer (Buffer) eine Größe von 8192 Bytes, kann aber beliebig angepasst werden. Die Zeichen (`char` – 16 bit – 2 bytes) werden dabei vom `Writer` an den Decoder (hier die Klasse `OutputStreamWriter`) weitergeleitet, der die Zeichen entsprechend der Dekodiertabelle in Bytes umwandelt, die wiederum an einen `OutputStream` weiterleitet, damit die Daten in einer Datei landen.



Warum ist dies so kompliziert?

Das liegt daran, dass Text mit unterschiedlicher Kodierung, auch Encoding genannt (z. B. UTF-8, ISO8859-15, ASCII, ...) erstellt werden kann.[1] Ein normaler `String` in Java wird mit UTF-16 kodiert (16-Bit), daher hat auch ein `char` die „Breite“ von 16-Bit. Ein normales Dateisystem verwendet aber für die Speicherung immer Bytes, also 8-Bit. Somit muss die entsprechende Kodierung in ein 8-Bit Schema umgewandelt werden, welches die Klasse `OutputStreamWriter` mit einem entsprechenden Kodierer erledigt.

```
try {
   File f = new File("datei.txt");
   BufferedWriter out = new BufferedWriter(
         new OutputStreamWriter(
            new FileOutputStream(f), "UTF8"));
   String text = "Dies ist ein Text mit Umlauten: öäüß ÖÄÜ!";
   out.write(text);
   out.newLine();
   out.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Über die Verbindung `BufferedWriter` zu `OutputStreamWriter` und dann zu `FileOutputStream` wird jedes Zeichen (`char` mit 16-bit) in entsprechende Bytes umgewandelt und wenn der Puffer voll ist (oder `flush()` aus dem Interface `Flushable` bzw. `close()` aus dem Interface `AutoCloseable` bzw. `Closeable` aufgerufen wird), in die Datei geschrieben. Die beiden Methoden leiten die Aktion dabei an den nächsten `Writer`, hier `OutputStreamWriter` weiter, usw.

Ausgabe der Datei 'datei.txt' mit dem Programm hexdump:

```
hexdump -C datei.txt
```

```
00000000  44 69 65 73 20 69 73 74  20 65 69 6e 20 54 65 78  |Dies ist ein Tex|
00000010  74 20 6d 69 74 20 55 6d  6c 61 75 74 65 6e 3a 20  |t mit Umlauten: |
00000020  c3 b6 c3 a4 c3 bc c3 9f  20 c3 96 c3 84 c3 9c 21  |........ ......!|
00000030  0a                                                |.|
00000031
```

Bei der Kodierung UTF-8 werden Zahlen, normale Buchstaben etc. mit 8-Bit gespeichert, Umlaute beispielsweise mit 16-Bit, also zwei Bytes (ö=c3 b6, ä=c3 a4, ...).

Methoden von „Writer“ (Auszug)

`append()` Hängt Zeichen, Zeichenketten, Arrays etc. an.

`close()` Schließt den Stream (zuvor wird `flush()` aufgerufen).

`flush()` Leert einen Puffer.

`write()` Schreibt Zeichen, Zeichenketten, Arrays etc. in den Stream.

Methoden von „BufferedWriter“ (zusätzlich)

`newLine()` Schreibt die Zeilenendemarkierung
Windows: carriage return (CR) + newline (NL) (0x0d + 0x0a) (\r + \n)
Linux: NL (0x0x) (\n)
Mac: CR (0x0d) (\r)

Textdatei lesen

Das Lesen aus einer Datei funktioniert sehr ähnlich. Wichtig dabei ist, dass man die selbe Kodierung verwendet wie beim Speichern.

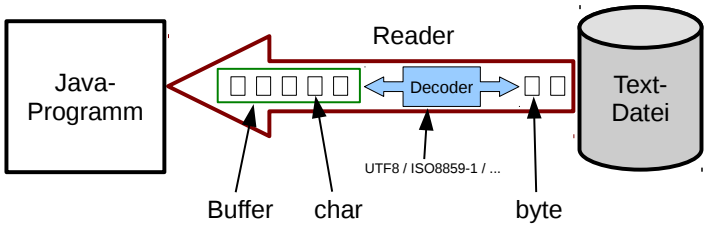
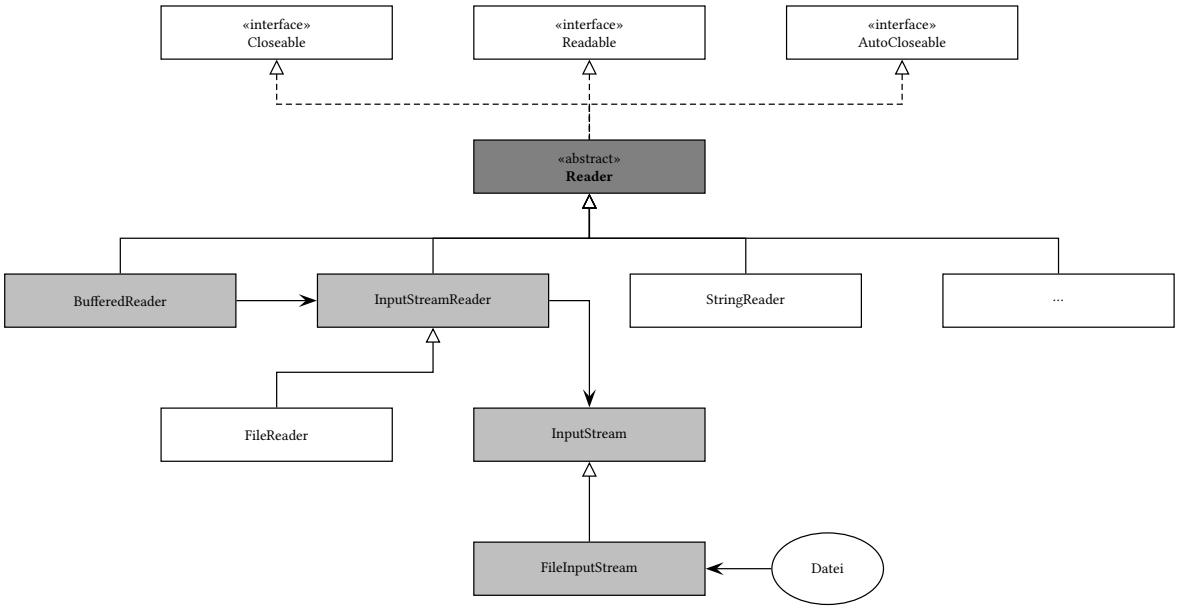


Abb. 8.5: Lesen aus einer Textdatei




```
try {
   File f = new File("datei.txt");
   BufferedReader in = new BufferedReader(
      new InputStreamReader(
         new FileInputStream(f), "UTF8"));
   String line;
   while ((line = in.readLine()) != null) {
      System.out.print(line);
   }
   in.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
Dies ist ein Text mit Umlauten: öäüß ÖÄÜ!
```

Methoden von „Reader“ (Auszug)

`read()` Liest ein Zeichen, ein Array etc. ein.

`close()` Schließt den Stream.

`reset()` Das Lesen beginnt wieder von vorne.

`skip()` Es werden n-Zeichen übersprungen.

Methoden von „BufferedReader“ (zusätzlich)

`readLine()` Liest eine komplette Zeile ein. Die Zeilenendemarkierung wird dabei automatisch erkannt.

Encoding

Nicht alle Encodings, die es gibt, werden von Java verwendet. Mit nachfolgendem Code werden die unterstützten Encodings angezeigt.

```
// IMPORT java.nio.charset.Charset
// IMPORT java.util.Iterator
// IMPORT java.util.SortedMap

SortedMap<String, Charset> cs = Charset.availableCharsets();

Iterator<String> it = cs.keySet().iterator();
while (it.hasNext()) {
   System.out.print(it.next());
   System.out.print(" ");
}
```

```
Big5 Big5-HKSCS CESU-8 EUC-JP EUC-KR GB18030 GB2312 GBK IBM-Thai IBM00858 IBM01140 ⤸
IBM01141 IBM01142 IBM01143 IBM01144 IBM01145 IBM01146 IBM01147 IBM01148 IBM01149 IBM037 ⤸
IBM1026 IBM1047 IBM273 IBM277 IBM278 IBM280 IBM284 IBM285 IBM290 IBM297 IBM420 IBM424 ⤸
IBM437 IBM500 IBM775 IBM850 IBM852 IBM855 IBM857 IBM860 IBM861 IBM862 IBM863 IBM864 IBM865⤸
 IBM866 IBM868 IBM869 IBM870 IBM871 IBM918 ISO-2022-CN ISO-2022-JP ISO-2022-JP-2 ISO-2022-⤸
KR ISO-8859-1 ISO-8859-13 ISO-8859-15 ISO-8859-2 ISO-8859-3 ISO-8859-4 ISO-8859-5 ISO⤸
-8859-6 ISO-8859-7 ISO-8859-8 ISO-8859-9 JIS_X0201 JIS_X0212-1990 KOI8-R KOI8-U Shift_JIS ⤸
TIS-620 US-ASCII UTF-16 UTF-16BE UTF-16LE UTF-32 UTF-32BE UTF-32LE UTF-8 windows-1250 ⤸
windows-1251 windows-1252 windows-1253 windows-1254 windows-1255 windows-1256 windows-1257⤸
 windows-1258 windows-31j x-Big5-HKSCS-2001 x-Big5-Solaris x-COMPOUND_TEXT x-euc-jp-linux ⤸
x-EUC-TW x-eucJP-Open x-IBM1006 x-IBM1025 x-IBM1046 x-IBM1097 x-IBM1098 x-IBM1112 x-⤸
IBM1122 x-IBM1123 x-IBM1124 x-IBM1166 x-IBM1364 x-IBM1381 x-IBM1383 x-IBM300 x-IBM33722 x-⤸
IBM737 x-IBM833 x-IBM834 x-IBM856 x-IBM874 x-IBM875 x-IBM921 x-IBM922 x-IBM930 x-IBM933 x-⤸
IBM935 x-IBM937 x-IBM939 x-IBM942 x-IBM942C x-IBM943 x-IBM943C x-IBM948 x-IBM949 x-IBM949C⤸
 x-IBM950 x-IBM964 x-IBM970 x-ISCII91 x-ISO-2022-CN-CNS x-ISO-2022-CN-GB x-iso-8859-11 x-⤸
JIS0208 x-JISAutoDetect x-Johab x-MacArabic x-MacCentralEurope x-MacCroatian x-MacCyrillic⤸
 x-MacDingbat x-MacGreek x-MacHebrew x-MacIceland x-MacRoman x-MacRomania x-MacSymbol x-⤸
MacThai x-MacTurkish x-MacUkraine x-MS932_0213 x-MS950-HKSCS x-MS950-HKSCS-XP x-mswin-936 ⤸
x-PCK x-SJIS_0213 x-UTF-16LE-BOM X-UTF-32BE-BOM X-UTF-32LE-BOM x-windows-50220 x-windows⤸
-50221 x-windows-874 x-windows-949 x-windows-950 x-windows-iso2022jp
```

System-Encoding

Über die System-Properties lässt sich festzustellen, welche Kodierung standardmäßig verwendet wird.

Der Wert `console.encoding` wird nicht von jedem Betriebssystem verwendet.

```
// IMPORT java.nio.charset.Charset

System.out.println( "os.name:          " + System.getProperty( "os.name" ) );
System.out.println( "default-charset:  " + Charset.defaultCharset() );
System.out.println( "file.encoding:    " + System.getProperty( "file.encoding" ) );
String encoding = System.getProperty("console.encoding");
if (encoding == null) {
   encoding = System.getProperty("file.encoding");
}
System.out.println( "console.encoding: " + encoding );
```

```
os.name:          Linux
default-charset:  UTF-8
file.encoding:    UTF-8
console.encoding: UTF-8
```

8.2.4 Binärdateien

Binärdateien werden mit der abstrakten Klasse `InputStream` bzw. mit deren Ableitungen gelesen und mit der abstrakten Klasse `OutputStream` bzw. mit deren Ableitungen geschrieben.

Dateien schreiben

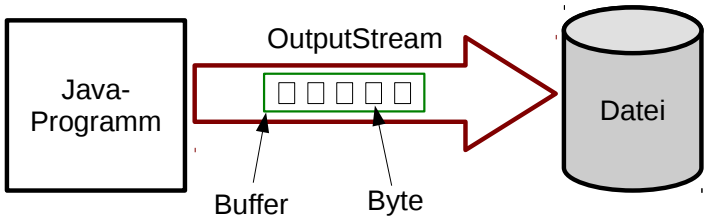


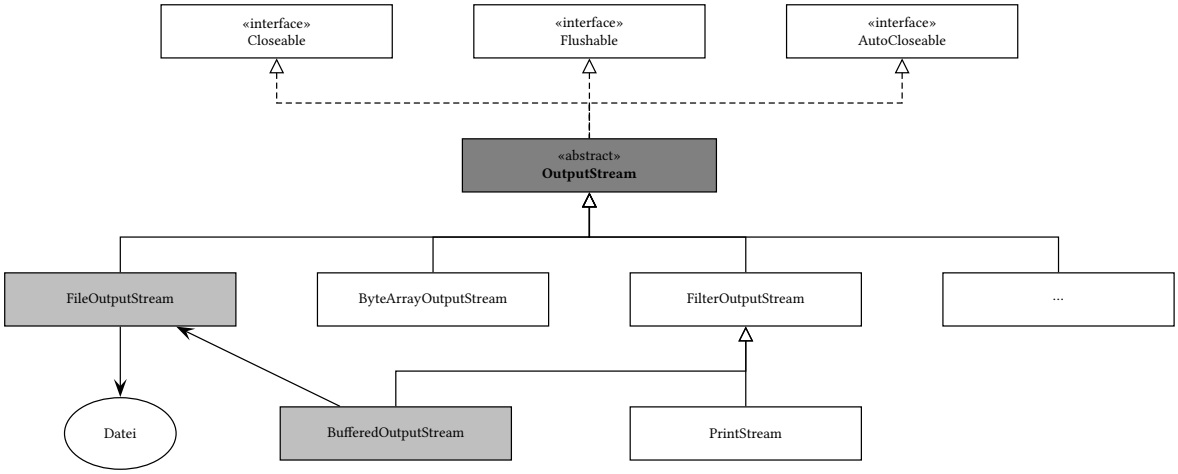
Abb. 8.6: Schreiben von Bytes

Im nachfolgenden Beispiel werden die Bytes (mit der Methode `getBytes()`) eines Strings in eine Datei geschrieben.

```
try {
   File f = new File("datei.bin");
   BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(f));
   String text = "Dies ist ein Text! Umlaute: öäüßÖÄÜ";
   out.write(text.getBytes());
   out.write('\n');
   out.close();
   System.out.println("File " + f.getName() + " created!");
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
File datei.bin created!
```

Dabei wird hier auch, wie beim `Writer` ein Buffer (Standard: 8 KiB) verwendet.



Dateien lesen

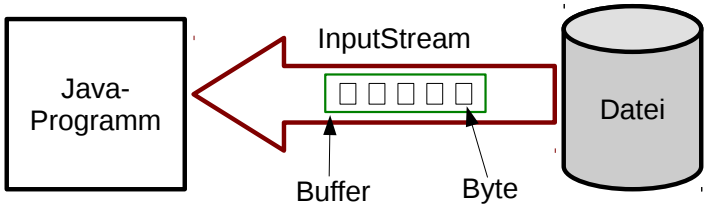


Abb. 8.7: Lesen von Bytes

Liest man die Daten wieder (hier Bytes) aus der Datei und zeigt diese an (in der Annahme, dass man wieder die selben Textzeichen erhält), wird man eine Überraschung erleben.

```
try {
   File f = new File("datei.bin");
   BufferedInputStream in = new BufferedInputStream(new FileInputStream(f));
   int ch;
   while ((ch = in.read()) != -1) {
      System.out.print((char) ch);
   }
   in.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

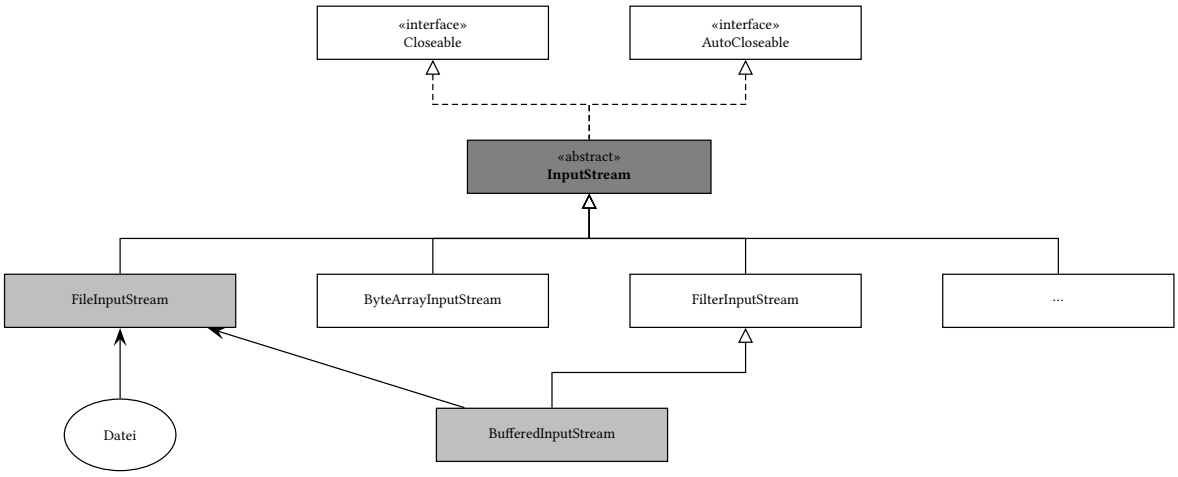
```
Dies ist ein Text! Umlaute: Ã¶Ã¤Ã¼Ã Ã Ã
```

Was ist hier passiert: Die Umlaute werden mit der Methode `getBytes()` in zwei Bytes umgewandet. Beim Lesen wird aber jeweils nur ein Byte gelesen und dann entsprechend angezeigt, was dazu führt, dass hier „Schrottzeichen“ angezeigt werden.

Zum Code:
In der Schleife wird dabei jedes einzelne Byte mit der Methode `read()` gelesen. Diese liefert das gelesene Byte als `int`-Wert zurück. Wird der Wert -1 zurückgeliefert, wird damit signalisiert, dass das Dateiende erreicht worden ist.

Warum bekomme ich ein `int` zurück, ich lese doch ein Byte?

Ein Byte (ohne Vorzeichen) umfasst den Zahlenbereich von 0 bis 255 (8 Bit). Soll aber noch zusätzlich mit dieser Information das Dateiende übertragen werden, so werden 9 Bit benötigt. In Java gibt es aber keinen einfachen Datentyp, der 9 Bit speichern kann. Der nächst größere Datentyp wäre `short` (16 Bit). Da aber die aktuellen CPU's einen Datenbus mit mindestens 32 Bit haben, also bei jedem Zugriff mindestens 32 Bit lesen, macht es keinen Unterschied, ob man das Byte im Datentyp `short` oder `int` speichert. Daher wurde der Standardtyp `int` verwendet.



Schreiben/Lesen von einfachen Datentypen

Die Klasse `DataOutputStream` dient dazu, einfache Datentypen (int, char, boolean, ...) zu speichern. Gelesen werden diese über die Klasse `DataInputStream`.

`writeBoolean(boolean v)`	Writes a boolean to the underlying output stream as a 1-byte value.
`writeByte(int v)`	Writes out a byte to the underlying output stream as a 1-byte value.
`writeBytes(String s)`	Writes out the string to the underlying output stream as a sequence of bytes.
`writeChar(int v)`	Writes a char to the underlying output stream as a 2-byte value, high byte first.
`writeChars(String s)`	Writes a string to the underlying output stream as a sequence of characters.
`writeDouble(double v)`	Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.
`writeFloat(float v)`	Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.
`writeInt(int v)`	Writes an int to the underlying output stream as four bytes, high byte first.
`writeLong(long v)`	Writes a long to the underlying output stream as eight bytes, high byte first.
`writeShort(int v)`	Writes a short to the underlying output stream as two bytes, high byte first.
`writeUTF(String str)`	Writes a string to the underlying output stream using modified UTF-8 encoding in a machine-independent manner.

Schreiben

```
try {
   File f = new File("datei_data.bin");
   DataOutputStream out = new DataOutputStream(
         new BufferedOutputStream(new FileOutputStream(f)));

   int a = 5;
   boolean x = true;
   float fl = 1.34f;

   out.writeInt(a);
   out.writeBoolean(x);
   out.writeFloat(fl);
   out.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Ein Hexdump der erzeugten Datei sieht dabei wie folgt aus:

```
hexdump -C datei_data.bin
```

```
00000000  00 00 00 05 01 3f ab 85  1f                      |.....?...|
00000009
```

Lesen

```
try {
   File f = new File("datei_data.bin");
   DataInputStream in = new DataInputStream(new BufferedInputStream(
         new FileInputStream(f)));

   int a = in.readInt();
   boolean x = in.readBoolean();
   float fl = in.readFloat();

   System.out.println(a);
   System.out.println(x);
   System.out.println(fl);
   in.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
5
true
1.34
```

`readBoolean()`	Reads one input byte and returns true if that byte is nonzero, false if that byte is zero.
`readByte()`	Reads and returns one input byte.
`readChar()`	Reads two input bytes and returns a char value.
`readDouble()`	Reads eight input bytes and returns a double value.
`readFloat()`	Reads four input bytes and returns a float value.
`readInt()`	Reads four input bytes and returns an int value.
`readLong()`	Reads eight input bytes and returns a long value.
`readShort()`	Reads two input bytes and returns a short value.
`readUnsignedByte()`	Reads one input byte, zero-extends it to type int, and returns the result, which is therefore in the range 0 through 255.
`readUnsignedShort()`	Reads two input bytes and returns an int value in the range 0 through 65535.
`readUTF()`	Reads in a string that has been encoded using a modified UTF-8 format.

Schreiben/Lesen von Objekt Datentypen

Die Klasse `ObjectOutputStream` dient dazu, einfache Datentypen (int, char, boolean, ...) und Objekte (diese werden wiederum in einfache Datentypen zerlegt) zu speichern. Gelesen werden diese über die Klasse `ObjectInputStream`.

`writeBoolean(boolean v)`	Writes a boolean to the underlying output stream as a 1-byte value.
`writeByte(int v)`	Writes out a byte to the underlying output stream as a 1-byte value.
`writeBytes(String s)`	Writes out the string to the underlying output stream as a sequence of bytes.
`writeChar(int v)`	Writes a char to the underlying output stream as a 2-byte value, high byte first.
`writeChars(String s)`	Writes a string to the underlying output stream as a sequence of characters.
`writeDouble(double v)`	Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.
`writeFloat(float v)`	Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.
`writeInt(int v)`	Writes an int to the underlying output stream as four bytes, high byte first.

`writeLong(long v)`	Writes a long to the underlying output stream as eight bytes, high byte first.
`writeShort(int v)`	Writes a short to the underlying output stream as two bytes, high byte first.
`writeUTF(String str)`	Writes a string to the underlying output stream using modified UTF-8 encoding in a machine-independent manner.
`writeObject(Object obj)`	Write the specified object to the ObjectOutputStream.

Schreiben

```
try {
   File f = new File("datei_objdata.bin");
   ObjectOutputStream out = new ObjectOutputStream(
      new BufferedOutputStream(new FileOutputStream(f)));
   String text = "Dies ist ein Text!";
   out.writeObject(text);
   out.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Ein Hexdump der erzeugten Datei sieht dabei wie folgt aus:

```
hexdump -C datei_objdata.bin
```

```
00000000  ac ed 00 05 74 00 12 44  69 65 73 20 69 73 74 20  |....t..Dies ist |
00000010  65 69 6e 20 54 65 78 74  21                       |ein Text!|
00000019
```

Beschreibung:

`ac ed`	Magic number that is written to the stream header.
`00 05`	Version number that is written to the stream header.
`74`	String
`00 12`	Länge des Strings 0x0012 = 18
`...`	UTF-8 String

Lesen

```
try {
   File f = new File("datei_objdata.bin");
   ObjectInputStream in = new ObjectInputStream(
         new BufferedInputStream(new FileInputStream(f)));
   String text = (String) in.readObject();
   System.out.println(text);
   in.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
} catch (ClassNotFoundException e) {
   System.err.println(e.getMessage());
   System.exit(2);
}
```

```
Dies ist ein Text!
```

Liest man Objekte, so muss zusätzlich der Fehler `ClassNotFoundException` abgefangen werden, da es sein kann, dass beim Lesen die entsprechende Klasse nicht gefunden wird, beispielsweise, wenn diese aus einer externen Bibliothek stammt, die nicht eingebunden ist.

`readBoolean()`	Reads one input byte and returns true if that byte is nonzero, false if that byte is zero.
`readByte()`	Reads and returns one input byte.
`readChar()`	Reads two input bytes and returns a char value.
`readDouble()`	Reads eight input bytes and returns a double value.
`readFloat()`	Reads four input bytes and returns a float value.

`readInt()`	Reads four input bytes and returns an int value.
`readLong()`	Reads eight input bytes and returns a long value.
`readShort()`	Reads two input bytes and returns a short value.
`readUnsignedByte()`	Reads one input byte, zero-extends it to type int, and returns the result, which is therefore in the range 0 through 255.
`readUnsignedShort()`	Reads two input bytes and returns an int value in the range 0 through 65535.
`readUTF()`	Reads in a string that has been encoded using a modified UTF-8 format.
`readObject()`	Read an object from the ObjectInputStream.

Schreiben/Lesen von Objekt Datentypen mit XML

Die Klasse `XMLEncoder` dient dazu, Objekte in einer XML-Datei zu speichern. Gelesen werden diese über die Klasse `XMLDecoder`.

`writeObject(Object obj)` Write an XML representation of the specified object to the output.

Schreiben

```
// IMPORT java.beans.XMLEncoder
try {
   File f = new File("datei_obj.xml");
   XMLEncoder out = new XMLEncoder(new BufferedOutputStream(new FileOutputStream(f)));
   String text = "Dies ist ein Text!";
   out.writeObject(text);
   out.writeObject(5);
   out.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Werden einfache Datentypen verwendet, werden diese automatisch in die entsprechenden Referenzklassen, auch Wrapper-Klassen genannt, (`int` $\Longrightarrow$ `Integer`) umgewandelt.[2]

Die Datei sieht dabei wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_91" class="java.beans.XMLDecoder">
 <string>Dies ist ein Text!</string>
 <int>5</int>
</java>
```

Der Vorteil gegenüber der Klasse `ObjectOutputStream` ist, dass die erzeugte Datei für den Programmierer einfacher zu lesen ist und auch sehr einfach von anderen Systemen, auch in anderen Programmiersprachen. Der Nachteil, die Datei wird meist größer und das Lesen/Schreiben dauert länger.

Lesen

```
// IMPORT java.beans.XMLDecoder
try {
   File f = new File("datei_obj.xml");
   XMLDecoder in = new XMLDecoder(new BufferedInputStream(new FileInputStream(f)));
   String text = (String) in.readObject();
   System.out.println(text);
   System.out.println(in.readObject());
   in.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
Dies ist ein Text!
5
```

`readObject()` Reads the next object from the underlying input stream.

[2] Autoboxing und Unboxing sind Verfahren zum automatischen 'Ein-' und 'Auspacken' von primitiven Datentypen in Referenztypen.

Eigene Klassen

Möchte man auf diese Weise eigene Klassen speichern bzw. lesen, so müssen diese dem JavaBean-Standard entsprechen.[3]

Eine JavaBean-Klasse muss folgende Bedingungen erfüllen.

- Öffentlicher parameterloser Konstruktor
- Serialisierbarkeit (Interface `Serializable`), wenn selbst erstelle Klassen verwendet werden.
- Öffentliche Zugriffsmethoden (public getters/getters)

Beispiel:

```
18   public static class MyBean {
19
20      private String eintrag;
21      private int anz;
22
23      public MyBean() {
24         eintrag = "";
25         anz = 1;
26      }
27
28      /** @return the anz */
29      public int getAnz() {
30         return anz;
31      }
32
33      /** @return the eintrag */
34      public String getEintrag() {
35         return eintrag;
36      }
37
38      /** * @param anz the anz to set */
39      public void setAnz(final int anz) {
40         this.anz = anz;
41      }
42
43      /** @param eintrag the eintrag to set */
44      public void setEintrag(final String eintrag) {
45         this.eintrag = eintrag;
46      }
47   }
```

Die JavaBean enthält hier die Felder `eintrag` und `anz` und die dazugehörigen getter/setter.

```
54      try {
55         final File f = new File("target/WriteXmlBean.xml");
56         final XMLEncoder out = new XMLEncoder(new BufferedOutputStream(new ⤸
           FileOutputStream(f)));
57
58         final MyBean b = new MyBean();
59         b.setEintrag("Dies ist ein Eintrag!");
60         b.setAnz(2);
61         out.writeObject(b);
62         out.close();
63      } catch (final IOException e) {
64         System.err.println(e.getMessage());
65         System.exit(1);
66      }
```

Das Lesen der Datei funktioniert ähnlich.

```
23      try {
24         final File f = new File("target/WriteXmlBean.xml");
25         final XMLDecoder in = new XMLDecoder(new BufferedInputStream(new FileInputStream(⤸
           f)));
26         final WriteXmlBean.MyBean b = (WriteXmlBean.MyBean) in.readObject();
27         System.out.println(b.getEintrag());
28         System.out.println(b.getAnz());
29         in.close();
30      } catch (final IOException e) {
31         System.err.println(e.getMessage());
32         System.exit(1);
33      }
```

[3] Alle Java-Standardklassen entsprechen dem JavaBean-Standard automatisch.

Die erzeugte Datei sieht dann wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_66" class="java.beans.XMLDecoder">
   <object class="de.nm.io.WriteXmlBean$MyBean">
      <void property="anz">
         <int>2</int>
      </void>
      <void property="eintrag">
         <string>Dies ist ein Eintrag!</string>
      </void>
   </object>
</java>
```

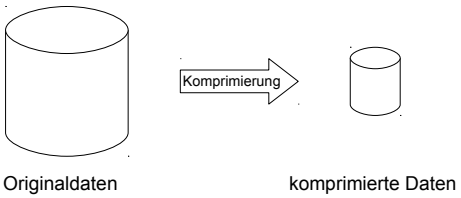
8.2.5 Datenkomprimierung

Man unterscheidet zwischen zwei Komprimierungsarten:

- **verlustbehaftete Komprimierung**
  Dabei wird der Datenstrom komprimiert und ein gewisser Qualitätsverlust wird in Kauf genommen, um eine geringere Datenmenge zu erhalten. Eine Dekomprimierung zum Original ist nicht mehr möglich. Solch ein Verfahren wird gerne bei Filmen, Audiodateien oder Bildern verwendet, z. B. bei jpg-Bildern.
  Eine verlustbehaftete Komprimierung ist z. B. für eine Datensicherung **nicht** geeignet!
- **verlustfreie Komprimierung**
  Dabei ist es jederzeit möglich, den komprimierten Datenstrom wieder in sein Original zurück zu führen.
  Nur eine verlustfreie Komprimierung ist z. B. für eine Datensicherung geeignet.

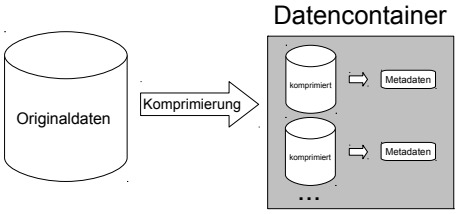
Die Daten werden auf zwei Arten komprimiert und weitergegeben:

als Datenstrom



Bei diesem Verfahren wird der Original-Datenstrom komprimiert und weitergegeben bzw. in einer Datei gespeichert. Dabei werden nur die reinen Daten komprimiert, Metadaten (bei einer Datei z. B. der Eigentümer, das Erstellungsdatum, etc.) gehen dabei verloren.

in einem Datencontainer



Hier werden nicht nur die Daten in komprimierter Form in einem Datencontainer gespeichert, sondern zu jeder gespeicherten Datei werden auch entsprechende Metadaten (z. B. Dateiname, Datum, Eigentümer, Zugriffsrechte, etc.) gespeichert. In einem Datencontainer können theoretisch beliebig viele Dateien und sogar Verzeichnisstrukturen gespeichert werden.

Eine Übersicht über viele Datenkomprimierungsprogramme findet man unter:
http://de.wikipedia.org/wiki/Liste_von_Datenkompressionsprogrammen

GZIP

- GZIP ist ein Kompressionsprogramm für Datenströme und einzelne Dateien
- Unix: `gzip datei.txt -> datei.txt.gz`
- Unix: `gunzip datei.txt.gz -> datei.txt`
- Java: `GZIPInputStream` und `GZIPOutputStream`

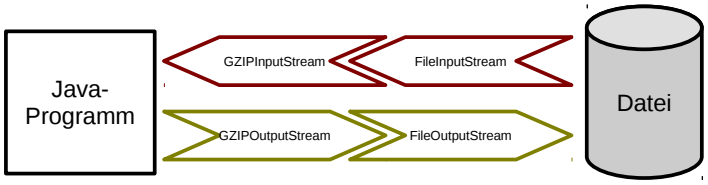


Abb. 8.8: GZIP-Streams

Als Beispiel wird hier die Datei „Die komplette Bibelausgabe Martin Luther 1912 als ASCII-Datei“[4] verwendet. Der Vorteil dabei ist, es ist eine große Textdatei mir realem Inhalt und ist lizenzfrei (Autor ist seit mehr als 70 Jahren verstorben).

```
stat -c %s Martin_Luther_Uebersetzung_1912.txt
```

```
4370012
```

Im nachfolgenden Code wird die Datei gelesen und mit GZIP komprimiert. Dabei wird jeweils ein Block von 32 KiB gelesen und geschrieben. Die Blockgröße hat bei vielen Komprimierungsalgorithmus eine wichtige Rolle. Ist diese zu klein, kann der Algorithmus ineffektiv sein. Ein zu großer Wert bringt dagegen keinen Vorteil mehr und kostet nur Speicher.

```
// IMPORT java.util.zip.GZIPOutputStream
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   File fout = new File("Martin_Luther_Uebersetzung_1912.txt.gz");
   int BUFFERSIZE = 32768;

   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   GZIPOutputStream out = new GZIPOutputStream(new FileOutputStream(fout), BUFFERSIZE);

   byte[] buffer = new byte[BUFFERSIZE];
   int bytes_read;
   while ((bytes_read = in.read(buffer)) != -1) {
      out.write(buffer, 0, bytes_read);
   }
   in.close();
   out.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Für die gepackte Datei ergibt sich nachfolgender Prozentsatz:

```
sh calkomp Martin_Luther_Uebersetzung_1912.txt Martin_Luther_Uebersetzung_1912.txt.gz
```

```
4370012 - 1384616 = 2985396 : 31%
```

[4] siehe http://bitimage.dyndns.org/german/MartinLuther-1912/Martin_Luther_Uebersetzung_1912.txt

ZIP

Das ZIP-Dateiformat (zipper ⟹ „Reißverschluss“) ist ein Containerformat für komprimierte Dateien. Dabei wird für jede Datei ein extra Eintrag mit der Klasse `ZipEntry` erzeugt und dem Stream hinzugefügt (Klasse `ZipOutputStream`). Anschließend wird der eigentliche Dateiinhalt in den Stream geschrieben. Danach muss der Eintrag wieder mit `closeEntry()` geschlossen werden.

```
// IMPORT  java.util.zip.ZipEntry
// IMPORT  java.util.zip.ZipOutputStream
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   File fout = new File("Martin_Luther_Uebersetzung_1912.zip");
   int BUFFERSIZE = 32768;

   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   ZipOutputStream out = new ZipOutputStream(new BufferedOutputStream(
      new FileOutputStream(fout), BUFFERSIZE));
   out.setLevel(9);

   ZipEntry entry = new ZipEntry(fin.getName());
   out.putNextEntry(entry);

   byte[] buffer = new byte[BUFFERSIZE];
   int bytes_read;
   while ((bytes_read = in.read(buffer)) != -1) {
      out.write(buffer, 0, bytes_read);
   }
   out.closeEntry();

   in.close();
   out.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Für die gepackte Datei ergibt sich nachfolgender Prozentsatz:

```
sh calkomp Martin_Luther_Uebersetzung_1912.txt Martin_Luther_Uebersetzung_1912.zip
```

```
4370012 - 1374348 = 2995664 : 31%
```

Will man die komprimierte Datei entpacken oder Informationen darüber erhalten, so steht hier die Klasse `ZipFile` zur Verfügung. Den eigentlichen Inhalt erhält man über die Methode `getInputStream(...)`.

```
// IMPORT  java.util.zip.ZipEntry
// IMPORT  java.util.zip.ZipFile
// IMPORT  java.util.Enumeration
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.zip");

   ZipFile zip = new ZipFile(fin);
   for (Enumeration<? extends ZipEntry> e = zip.entries(); e.hasMoreElements();) {
      ZipEntry entry = e.nextElement();
      System.out.println(entry.getName());
      System.out.println("   " + entry.getSize() + " -> " + entry.getCompressedSize());
   }
   zip.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
Martin_Luther_Uebersetzung_1912.txt
   4370012 -> 1374164
```

Verzeichnisse

Will man Unterverzeichnisse mit in das ZIP-Archiv packen, so wird das entsprechende Verzeichnis mit dem Trenner `/` vor den Dateinamen im `ZipEntry` geschrieben.

```
ZipEntry entry = new ZipEntry("verzeichnis/unterverzeichnis/" + fin.getName());
```

JAR

Jar-Archive sind Zip-Archiven sehr ähnlich. Der Unterschied besteht darin, dass hier zusätzlich eine Manifest-Datei enthalten ist. Die Klasse `JarFile` entspricht `ZipFile`, `JarEntry` entspricht `ZipEntry` und `JarOutputStream` entspricht `ZipOutputStream`.

Apache Commons Compress

Apache stellt eine Bibliothek mit dem Namen Apache Commons Compress (kurss ACC) zur Verfügung, die viele Komprimierungsarten bietet.

Eine Auswahl:

AR Standardisiertes Unix-Archivformat (ohne Komprimierung) (inzwischen von tar abgelöst).

CPIO Standardisiertes Unix-Archivformat. cpio bietet selbst keine Komprimierung, es kann aber mit compress/gzip/bzip2 kombiniert werden.

TAR Containerformat, ursprünglich „Tape-Archiver“ von Unix für Bandsicherung. Reine tar-Archive sind unkomprimiert, tar kann aber mit compress, gzip, bzip2 oder xz verwendet werden.

ZIP Containerformat für komprimierte Dateien mit Metainformationen.

JAR Containerformat (wie zip) für Java.

BZIP2 Kompressionsprogramm unter Unix. Alternative zu gzip, bietet bessere Kompression, ist jedoch zeit- und speicheraufwendiger.

GZIP GNU Kompressionsprogramm für Unix.

XZ Alternative zu bzip2. Der Kompressionsalgorithmus basiert auf dem LZMA-Verfahren und ermöglicht daher gerade bei Binärdateien höhere Kompressionsraten.

7Z 7z ist ein freies Dateiformat zur komprimierten Archivierung von Dateien. Bietet sehr hohe Datenkompressionsrate und ist geeignet für sehr großen Datenmengen, sowie Verschlüsselung nach dem AES-256-Standard.

BZIP2

BZIP2 komprimiert deutlich besser als GZIP, benötigt aber mehr Rechenleistung und ist langsamer.

```
// IMPORT org.apache.commons.compress.compressors.bzip2.BZip2CompressorOutputStream;
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   File fout = new File("Martin_Luther_Uebersetzung_1912.txt.bzip2");
   int BUFFERSIZE = 32768;

   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   BZip2CompressorOutputStream out = new BZip2CompressorOutputStream(
                  new BufferedOutputStream(new FileOutputStream(fout), BUFFERSIZE));

   byte[] buffer = new byte[BUFFERSIZE];
   int bytes_read;
   while ((bytes_read = in.read(buffer)) != -1) {
      out.write(buffer, 0, bytes_read);
   }
   in.close();
   out.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Für die gepackte Datei ergibt sich nachfolgender Prozentsatz:

```
sh calkomp Martin_Luther_Uebersetzung_1912.txt Martin_Luther_Uebersetzung_1912.txt.bzip2
```

```
4370012 - 998523 = 3371489 : 22%
```

TAR

TAR ist ein Containerformat (in Unix sehr weit verbreitet), welches ursprünglich zur Bandarchivierung verwendet worden ist. Es selbst komprimiert nicht, kann aber mit jeden Streamkomprimierer verknüpft werden, beispielsweise BZIP2.

Dabei hat TAR den selben Mechanismus wie ZIP, jedoch mit den Klassen `TarArchiveOutputStream`, `TarArchiveEntry` und `TarArchiveInputStream`.

```
// IMPORT org.apache.commons.compress.compressors.bzip2.BZip2CompressorOutputStream
// IMPORT org.apache.commons.compress.archivers.tar.TarArchiveEntry
// IMPORT org.apache.commons.compress.archivers.tar.TarArchiveOutputStream
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   File fout = new File("Martin_Luther_Uebersetzung_1912.tar.bz2");
   int BUFFERSIZE = 32768;

   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   TarArchiveOutputStream out = new TarArchiveOutputStream(
         new BZip2CompressorOutputStream(
            new BufferedOutputStream(new FileOutputStream(fout), BUFFERSIZE)));

   TarArchiveEntry entry = new TarArchiveEntry(fin);
   out.putArchiveEntry(entry);

   byte[] buffer = new byte[BUFFERSIZE];
   int bytes_read;
   while ((bytes_read = in.read(buffer)) != -1) {
      out.write(buffer, 0, bytes_read);
   }
   out.closeArchiveEntry();

   out.finish();
   out.close();
   in.close();

} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Für die gepackte Datei ergibt sich nachfolgender Prozentsatz:

```
sh calkomp Martin_Luther_Uebersetzung_1912.txt Martin_Luther_Uebersetzung_1912.tar.bz2
```

```
4370012 - 998944 = 3371068 : 22%
```

Ein Test mit `tar` zeigt folgendes Ergebnis:[5]

```
tar tjvf Martin_Luther_Uebersetzung_1912.tar.bz2
```

```
-rw-r--r-- 0/0         4370012 2016-05-24 08:05 Martin_Luther_Uebersetzung_1912.txt
```

[5] t -> Test, j -> Komprimierung mit bzip2, v -> verbose, f -> filename

7z

7z ist ein freies Dateiformat zur komprimierten Archivierung von Dateien. Es bietet sehr hohe Datenkompressionsraten und ist geeignet für sehr große Datenmengen, sowie Verschlüsselung nach dem AES-256-Standard.[6]

```
// IMPORT org.apache.commons.compress.archivers.sevenz.SevenZArchiveEntry
// IMPORT org.apache.commons.compress.archivers.sevenz.SevenZOutputFile
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   File fout = new File("Martin_Luther_Uebersetzung_1912.7z");
   int BUFFERSIZE = 32768;

   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   SevenZOutputFile out = new SevenZOutputFile(fout);

   SevenZArchiveEntry entry = out.createArchiveEntry(fin, fin.getName());
   out.putArchiveEntry(entry);

   byte[] buffer = new byte[BUFFERSIZE];
   int bytes_read;
   while ((bytes_read = in.read(buffer)) != -1) {
      out.write(buffer, 0, bytes_read);
   }
   out.closeArchiveEntry();

   out.finish();
   out.close();
   in.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Für die gepackte Datei ergibt sich nachfolgender Prozentsatz:

```
sh calkomp Martin_Luther_Uebersetzung_1912.txt Martin_Luther_Uebersetzung_1912.7z
```

```
4370012 - 1038206 = 3331806 : 23%
```

[6] Note that Commons Compress currently only supports a subset of compression and encryption algorithms used for 7z archives. For writing only uncompressed entries, LZMA2, BZIP2 and Deflate are supported - reading also supports LZMA and AES-256/SHA-256.

8.2.6 Prüfsummen

Überträgt man eine Datei, so möchte man sicherstellen, dass diese auch korrekt angekommen ist, also die Integrität der Daten sichergestellt ist. Aus diesem Grund berechnet man von der Datei eine eindeutige Prüfsumme (engl. checksum) und übermittelt diese ebenfalls. Der Empfänger kann dann erneut diese Prüfsumme ermitteln und so mit der übertragenen Prüfsumme vergleichen. Sind die beiden identisch, so hat die Übertragung einwandfrei geklappt.

Einfache Verfahren

Einfache Verfahren sind beispielsweise die Erstellung der Quersumme oder das Erstellen eines Paritätsbit. Jedoch werden damit keine Zahlendreher erkannt.

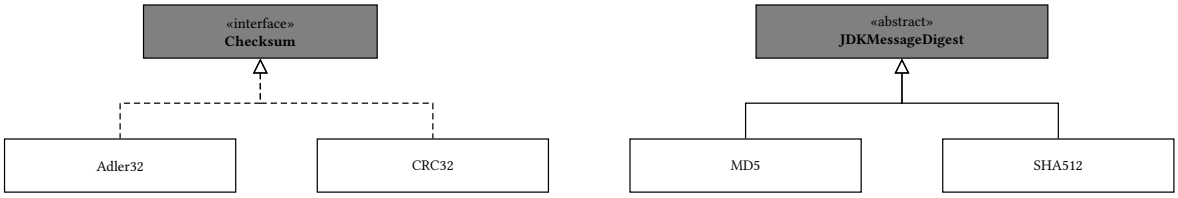
Komplexere Verfahren

Komplexere Verfahren errechnen anhand eines Prüfverfahrens eine Prüfsumme oder berechnen über eine Hashfunktion einen eindeutigen Hashwert.

Je nach Verfahren kann dieses auch verwendet werden, um beispielsweise ein Passwort in einen Hashwert umzuwandeln, um so Passwörter zu verwalten, ohne den Klartext speichern zu müssen. Man spricht dann von einer kryptographischen Hashfunktion.

In Java (ohne weitere Bibliotheken) sind die Verfahren Adler32, CRC32, MDx und SHAx implementiert.[7]

Nachfolgend die entspechende Klassenübersicht (Auswahl):



[7] Java hat hier die Bibliothek Bouncy Castle Crypto APIs eingebunden:
siehe https://www.bouncycastle.org/

Adler-32

Adler-32 ist ein einfacher, von Mark Adler entwickelter Prüfsummenalgorithmus. Er wird unter anderem von der zlib-Bibliothek benutzt, um (zufällige Übertragungs-) Fehler im komprimierten Datenstrom zu erkennen. In RFC 1950 wird der Algorithmus genau beschrieben. Der Adler-32-Algorithmus ist einfacher und lässt sich schneller berechnen als die bekannte Zyklische Redundanzprüfung (cyclic redundancy check), bietet aber auch weniger Sicherheit beim Erkennen von zufälligen Bitfehlern.[8]

```
// IMPORT java.util.zip.Adler32
// IMPORT java.util.zip.Checksum
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   int BUFFERSIZE = 32768;
   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   byte[] buffer = new byte[BUFFERSIZE];

   Checksum cs = new Adler32();
   int read = 0;
   while ((read = in.read(buffer)) > 0) {
      cs.update(buffer, 0, read);
   }
   in.close();
   System.out.println(Long.toHexString(cs.getValue()));
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
a41d9b9e
```

CRC-32

Die zyklische Redundanzprüfung (englisch cyclic redundancy check, daher meist CRC) ist ein Verfahren zur Bestimmung eines Prüfwerts für Daten, um Fehler bei der Übertragung oder Speicherung erkennen zu können. Im Idealfall kann das Verfahren sogar die empfangenen Daten selbständig korrigieren, um eine erneute Übertragung zu vermeiden. Es wurde 1961 von W. Wesley Peterson entwickelt.[9]

```
// IMPORT java.util.zip.CRC32
// IMPORT java.util.zip.Checksum
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   int BUFFERSIZE = 32768;
   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   byte[] buffer = new byte[BUFFERSIZE];

   Checksum cs = new CRC32();
   int read = 0;
   while ((read = in.read(buffer)) > 0) {
      cs.update(buffer, 0, read);
   }
   in.close();
   System.out.println(Long.toHexString(cs.getValue()));
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
6e6ac63f
```

[8] Quelle: siehe https://de.wikipedia.org/wiki/Adler-32
[9] Quelle: siehe https://de.wikipedia.org/wiki/Zyklische_Redundanzpr%C3%BCfung

MD5

Message-Digest Algorithm 5 (MD5) ist eine weit verbreitete kryptographische Hashfunktion, die aus einer beliebigen Nachricht einen 128-Bit-Hashwert erzeugt. Dies erlaubt beispielsweise die leichte Überprüfung eines Downloads auf Korrektheit. MD5 wurde 1991 von Ronald L. Rivest entwickelt. Sie gilt inzwischen nicht mehr als sicher, da es mit überschaubarem Aufwand möglich ist, unterschiedliche Nachrichten zu erzeugen, die den gleichen MD5-Hashwert aufweisen.[10]

```
// IMPORT java.security.MessageDigest;
// IMPORT java.security.NoSuchAlgorithmException;
// IMPORT java.math.BigInteger;
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   int BUFFERSIZE = 32768;
   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   byte[] buffer = new byte[BUFFERSIZE];

   MessageDigest md = MessageDigest.getInstance("MD5");
   int read = 0;
   while ((read = in.read(buffer)) > 0) {
      md.update(buffer, 0, read);
   }
   in.close();
   System.out.println(new BigInteger(1, md.digest()).toString(16));
} catch (IOException | NoSuchAlgorithmException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
6b464424500688fdf6faed1c57fd54bf
```

SHA512

SHA-2 (von englisch secure hash algorithm, sicherer Hash-Algorithmus) ist der Oberbegriff für die vier kryptologischen Hashfunktionen SHA-224, SHA-256, SHA-384 und SHA-512, die 2001 vom US-amerikanischen NIST als Nachfolger von SHA-1 standardisiert wurden.[11]

```
// IMPORT java.security.MessageDigest;
// IMPORT java.security.NoSuchAlgorithmException;
// IMPORT java.math.BigInteger;
try {
   File fin = new File("Martin_Luther_Uebersetzung_1912.txt");
   int BUFFERSIZE = 32768;
   BufferedInputStream in = new BufferedInputStream(new FileInputStream(fin), BUFFERSIZE);
   byte[] buffer = new byte[BUFFERSIZE];

   MessageDigest md = MessageDigest.getInstance("SHA-512");
   int read = 0;
   while ((read = in.read(buffer)) > 0) {
      md.update(buffer, 0, read);
   }
   in.close();
   String bi = new BigInteger(1, md.digest()).toString(16);
   System.out.println(bi.substring(0, 80) + "\n" + bi.substring(80));
} catch (IOException | NoSuchAlgorithmException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
dccf75de819f377dabe792d1c675d07f9ebd7687b4086b45ff54734feed95f257dac8e21bdcd256e
d175a2399b10fb1c8e55b92bbaa8dcf4fff9da320e5b8823
```

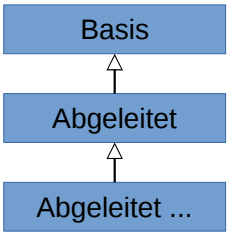
[10] Quelle: siehe https://de.wikipedia.org/wiki/Message-Digest_Algorithm_5
[11] Quelle: siehe https://de.wikipedia.org/wiki/SHA-2

8.3 Vererbung und Co.

Wird im allgemeinen Sprachgebrauch von OOP, Vererbung und Co. gesprochen, so werden hier meist mehrere Bereiche (Schlagworte) zusammengefasst:

- Vererbung
- Komposition
- Polymorphie
- Generische Programmierung
- Generics
- abstrakte Klassen
- Interfaces
- ...

8.3.1 Vererbung und Komposition



Vererbung im objektorientierten Sinne bedeutet, dass eine Klasse die in ihr definierten Elemente, also alle Attribute und Methoden, an eine andere Klasse weitervererbt. Sie deswegen jedoch einzig und allein als bequemes Mittel zur Wiederverwendung von Code zu sehen, würde ihr nicht gerecht. Mit ihr lassen sich eine Reihe weiterer objektorientierter Konzepte realisieren, wie zum Beispiel das Überschreiben von Methoden, die Polymorphie, abstrakte Klassen und der Aufbau von Klassenhierarchien.

Das Prinzip der Vererbung

Bei der Vererbung wird eine Klasse von einer bestehenden Klasse abgeleitet. Die abgeleitete Klasse erbt dadurch die Elemente der Basisklasse. Basisklasse und abgeleitete Klasse sind dabei ganz normale Klassen, welche folgende Funktionen bei der Vererbung erfüllen:

Basisklasse Sie gibt ihre Elemente weiter (vererbt sie). Man spricht in Java auch von Superklasse.

Abgeleitete Klasse Nimmt die Elemente in Empfang (erbt sie). Man spricht in Java auch von Subklasse.

Der grundlegende Mechanismus

Um eine Klasse als Basisklasse einer neuen Klasse zu verwenden, muss man lediglich bei der Definition der neuen Klasse den Namen der Basisklasse mit dem Schlüsselwort `extends` an den Namen der neuen Klasse hängen.

Jetzt verfügt die abgeleitete Klasse neben der von ihr selbst definierten Attributen und Methoden automatisch auch über die Elemente der Basisklasse.

```
static class Basis {
   int wert;

   void ausgeben()  {
      System.out.println(" Wert = " + wert);
   }
}

static class Abgeleitet extends Basis {

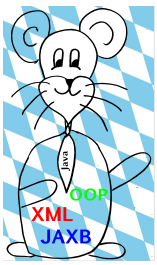
   //Das geerbte Feld wert kann direkt verwendet werden.
   void erhoehen() {
         wert += 10;
   }
}

public static void main(String[] args) {

   //Für Objekte der Klasse Abgeleitet können sowohl geerbte, als auch
   //selbst definierte Elemente benutzt werden.
   Abgeleitet obj = new Abgeleitet();

   obj.wert = 5;   //geerbt
   obj.erhoehen(); //selbst definiert
   obj.ausgeben(); //geerbt
}
```

```
 Wert = 15
```



static class

Bei den Beispielen wird die übergeordnete Klasse (alles „landet“ in einer Datei) automatisch generiert und die dargestellten Klassen als „innere Klassen“ verwendet. Daher ist hier ein `static` notwendig.

Der Sinn der Vererbung

Als Programmierer wird man im Laufe seiner Karriere eine Vielzahl von Klassen zur Repräsentation verschiedener Objekte und zur Lösung unterschiedlicher Aufgaben schreiben. Irgendwann stellt man fest, dass man für die Implementierung einer neu zu erstellenden Klasse die Funktionalität einer Klasse, die man vor langer Zeit einmal geschrieben hat, sehr gut wiederverwenden kann. Nun stellt sich die Frage, wie sich die Funktionalität einer Klasse bei der Implementierung neuer Klassen sinnvoll wiederverwenden lässt.

Die objektorientierte Programmierung bietet hier zwei unterschiedliche sich ergänzende Konzepte an:

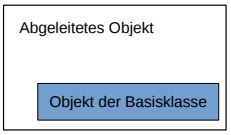
- Vererbung
- Komposition

Während bei der Vererbung die Elemente der bestehenden Klasse an die neue Klasse vererbt werden, geschieht bei der Komposition nichts anderes, als dass in der neuen Klasse ein Feld vom Typ der bestehenden Klasse definiert wird.

Wichtige Fakten ...

- Die Basisklasse vererbt alle in ihr definierten Elemente mit Ausnahme der Konstruktoren und der private-Elemente.
- Es gibt keine Möglichkeit, auf die Auswahl der vererbten Elemente einzuwirken.
- Die Basisklasse wird durch die Vererbung nicht geändert.
- Eine Klasse kann nicht gleichzeitig von zwei oder mehreren Basisklassen abgeleitet werden.
- Klassen, die als final definiert sind, können nicht als Basisklasse verwendet werden.

Geerbte Elemente bilden Unterobjekte



Werden Elemente durch Vererbung von einer Basisklasse an eine abgeleitete Klasse weitergegeben, bilden sie in der abgeleiteten Klasse eine eigenständige Untereinheit, einen Basisklassenanteil. Einerseits werden die Elemente also durch die Vererbung zu einem Teil der abgeleiteten Klasse, andererseits bleiben sie formal Basisklassenelemente.

Grundsätzlich kann man mit den geerbten Elementen genauso programmieren wie mit den Elementen, die direkt in der abgeleiteten Klasse definiert wurden. In bestimmten Situationen tritt allerdings die Basisklassenherkunft der geerbten Elemente in den Vordergrund und muss berücksichtigt werden.

Im Einzelnen betrifft dies:

- den Zugriff auf geschützte Klassenelemente (Methoden, die in der abgeleiteten Klasse definiert sind, können nur auf die geerbten Elemente zugreifen, deren Zugriffsmodifizierer dies zulassen – Tabelle 8.1 auf der nächsten Seite zeigt die vier Zugriffsebenen),
- die Instanzbildung (wird von einer abgeleiteten Klasse ein Objekt erzeugt, bilden die geerbten Basisklassenelemente in diesem ein Unterobjekt, das durch einen Konstruktor der Basisklasse initialisiert werden muss),
- die Behandlung als Basisklassenobjekt (da Objekte abgeleiteter Klassen ein Basisklassenunterobjekt enthalten, kann man sie auch als Basisklassenobjekte behandeln).

Definition in Basisklasse	*Zugriff aus Basisklasse*	*Zugriff aus abgel. Klasse*	*Zugriff aus Paket*	*Zugriff von „woanders“*
`public`	ja	ja	ja	ja
default	ja	ja gleiches Paket!	ja	nein
`protected`	ja	ja	ja	nein
`private`	ja	nein	nein	nein

Tab. 8.1: Vier Zugriffsebenen

Beispiele

```
static class A_Zaehler {

   // Die Klasse schützt Ihre Daten. Der Zähler kann nur über die Methode
   // zugängliche Methoden manipuliert werden.
   private int wert;

   public A_Zaehler() {
      wert = 0;
   }

   public void weiterdrehen() {
      ++wert;
   }

   public int abfragen() {
      return wert;
   }
}

public static void main(String[] args) {

   A_Zaehler zaehler = new A_Zaehler();

   for (int i = 0; i <= 10; i++) {
      System.out.print(zaehler.abfragen() + " " );
      zaehler.weiterdrehen();
   }
}
```

```
0 1 2 3 4 5 6 7 8 9 10
```

```
static class A_Zaehler {

   // Damit MeinZaehler den Wert manipulieren kann, muss der Zugriff auf vereerbte ⤸
   Elemente
   // erweitert werden.
   protected int wert;

   public A_Zaehler() {
      wert = 0;
   }

   public void weiterdrehen() {
      ++wert;
   }

   public int abfragen() {
      return wert;
   }
}

static class B_MeinZaehler extends A_Zaehler {

   public B_MeinZaehler(final int startwert) {
      wert = startwert;
   }

   // Erweiterung
   public void zurueckdrehen() {
      --wert;
   }
}

public static void main(String[] args) {

   A_Zaehler zaehler = new A_Zaehler();

   for (int i = 0; i <= 10; i++) {
      System.out.print(zaehler.abfragen() + " " );
      zaehler.weiterdrehen();
   }
   System.out.println();

   // Einsatz des erweiterten, eigenen Zählers
   B_MeinZaehler meinZaehler = new B_MeinZaehler(10);

   System.out.println("\nDer abgeleitete Zaehler ist auch für Countdowns geeignet:");
   for (int i = 0; i <= 10; i++) {
      System.out.print(meinZaehler.abfragen() + " ");
      meinZaehler.zurueckdrehen();
   }
}
```

```
0 1 2 3 4 5 6 7 8 9 10

Der abgeleitete Zaehler ist auch für Countdowns geeignet:
10 9 8 7 6 5 4 3 2 1 0
```

Wie initialisiert man geerbte Elemente?

Dass geerbte Elemente in einer abgeleiteten Klasse ein Unterobjekt bilden, merkt man daran, dass für die Erzeugung und Initialisierung dieser Elemente die Konstruktoren der Basisklasse und nicht die Konstruktoren der abgeleiteten Klasse zuständig sind.

Zur Erzeugung des Basisklassenunterobjekts wird also der Konstruktor der Basisklasse aufgerufen. Dass sich dies unter Umständen nicht im Quelltext widerspiegelt, liegt daran, dass der Compiler zur Erzeugung des Basisklassenunterobjekts den Standardkonstruktor der Basisklasse heranzieht.

Die Aufgabenteilung sieht also wie folgt aus:

- Zur Erzeugung der Basisklassenunterobjekte wird ein in der Basisklasse definierter Konstruktor gerufen.
- Die Einrichtung des eigenen Objekts (Initialisierung der Felder und andere Arbeiten) übernimmt der Konstruktor der abgeleiteten Klasse.

Basisklasse: Compiler erzeugt Basiskonstruktor

```
static class Basis {
   int b_wert;
}

static class Abgeleitet extends Basis {
   int abg_wert;

   Abgeleitet() {
      abg_wert = 22;
      b_wert = 11;
   }
}

public static void main(final String[] args) {

   // Es wird sowohl der Abgeleitet-Konstruktor als auch der Basisklassenkonstruktor
   // aufgerufen. Natürlich nicht sichtbar, weil der Standardkonstruktor vom Compiler
   // erzeugt wird.
   final Abgeleitet obj = new Abgeleitet();
   System.out.println("b_wert   = " + obj.b_wert);
   System.out.println("abg_wert = " + obj.abg_wert);
}
```

```
b_wert   = 11
abg_wert = 22
```

Basisklasse: mit selbst definiertem Konstruktor

```
static class Basis {
   int b_wert;
   Basis() {
      b_wert=999;
   }
}

static class Abgeleitet extends Basis {
   int abg_wert;

   Abgeleitet() {
      abg_wert = 22;
      // b_wert = 11;
   }
}

public static void main(final String[] args) {

   // Nun ist klar ersichtlich, dass zur Erzeugung des Basisklassenunterobjekts auch der
   // Basisklassenkonstruktor benutzt wird!
   final Abgeleitet obj = new Abgeleitet();
   System.out.println("b_wert   = " + obj.b_wert);
   System.out.println("abg_wert = " + obj.abg_wert);
}
```

```
b_wert   = 999
abg_wert = 22
```

Expliziter Aufruf des Basisklassenkonstruktors

Hat eine Klasse keinen Standardkonstruktor, sondern nur einen Konstruktor, der beim Aufruf die Übergabe eines oder mehrerer Argumente erwartet, steht man vor dem Problem, wie man dem Basisklassenkonstruktor Argumente übergeben kann.

Zum Aufruf des Basisklassenkonstruktors stellt Java das Schlüsselwort `super` zur Verfügung. Besitzt die Klasse also keinen Standardkonstruktor, so muss mit `super()` der Konstruktor der Basisklasse gerufen werden.

Gegebenenfalls übernimmt der Konstruktor der abgeleiteten Klasse nicht nur die Argumente für die eigenen Felder, sondern auch die Anfangswerte für die geerbten Felder und reicht diese an den Basisklassenkonstruktor weiter.

Der explizite Aufruf des Basisklassenkonstruktors ist auch dann interessant, wenn die Basisklasse über mehrere Konstruktoren verfügt. Wie bei der Überladung kann man in solchen Fällen über die Anzahl und Typen der weitergereichten Argumente auswählen, welcher Basisklassenkonstruktor verwendet werden soll.

```
static class Basis {
   int b_wert;

   Basis() {
      System.out.println("### Basis()");
      b_wert = 1;
   }
   Basis(final double p) {
      System.out.println("### Basis(double p)");
      b_wert = (int) p;
   }
   Basis(final int p) {
      System.out.println("### Basis(int p)");
      b_wert = p;
   }
}
static class Abgeleitet extends Basis {
   int abg_wert;

   Abgeleitet() {
      abg_wert = 2;
   }
   // Hier werden Argumente an den Basisklassenkonstruktor weitergereicht.
   Abgeleitet(final int param1, final double param2) {
      super(param2);
      abg_wert = param1;
   }
}

public static void main(final String[] args) {

   final Abgeleitet obj1 = new Abgeleitet();
   System.out.println("obj1: b_wert   = " + obj1.b_wert);
   System.out.println("obj1: abg_wert = " + obj1.abg_wert);

   final Abgeleitet obj2 = new Abgeleitet(22, 11.23);
   System.out.println("obj2: b_wert   = " + obj2.b_wert);
   System.out.println("obj2: abg_wert = " + obj2.abg_wert);
}
```

```
### Basis()
obj1: b_wert   = 1
obj1: abg_wert = 2
### Basis(double p)
obj2: b_wert   = 11
obj2: abg_wert = 22
```

Aufruf von Konstruktoren in Konstruktoren

Es gibt drei Techniken, mit denen man in der Definition eines Konstruktors andere Konstruktoren aufrufen kann:

`this` Mit dem Schlüsselwort `this` kann man einen anderen Konstruktor der gleichen Klasse aufrufen. Dies wird häufig verwendet um die Implementierung eines Konstruktors mit wenigen Argumenten auf einen Konstruktor mit mehreren Argumenten zurückzuführen.

`super` Mit dem Schlüsselwort `super` kann man einen Konstruktor der Basisklasse aufrufen.

`super()` Mit der Syntax `new Aussen().super()` kann man den Konstruktor einer Basisklasse aufrufen die als innere Klasse einer Klasse `Aussen` definiert ist.

Verdecken, überschreiben, überladen

Wenn man eine Klasse von einer anderen Klasse ableitet, begnügt man sich natürlich nicht damit, die Elemente der Basisklasse zu erben, sondern definiert weitere Elemente, die die Funktionalität der Basisklasse erweitern, spezialisieren oder anpassen.

Dazu kann man in der abgeleiteten Klasse gänzlich neue Elemente einführen. Es ist aber auch möglich, Elemente zu definieren, die den gleichen Namen tragen wie geerbte Basisklassenelemente.

Dann gilt es drei Fälle zu unterscheiden:

- Verdecken
- Überschreiben
- Überladung

Verdeckung

Wenn man in einer abgeleiteten Klasse ein Feld definiert, das den gleichen Namen trägt wie ein geerbtes Feld, verdeckt das neu definierte Feld im Gültigkeitsbereich der abgeleiteten Klasse das geerbte Feld.

Wir wissen, dass geerbte Elemente in den Objekten der abgeleiteten Klasse Unterobjekte bilden. Das hat Konsequenzen für die Verdeckung. Während die in der abgeleiteten Klasse definierten Methoden nämlich auf das neu definierte, verdeckte Feld zugreifen, arbeiten die geerbten Methoden weiter mit dem verdeckten Feld, das aus ihrem Basisklassenunterobjekt stammt.

Innerhalb der abgeleiteten Klassen kann man jederzeit mit dem Schlüsselwort `super` auf verdeckte Felder zugreifen. Von außen, über ein Objekt der abgeleiteten Klasse kann man ebenfalls auf verdeckte Instanzvariablen zugreifen, jedoch nicht mittels des Schlüsselwortes `super`, sondern durch Umwandlung des Objekts in den Typ der Basisklasse. Auf verdeckte statische Felder kann man wie gewohnt über den Klassennamen zugreifen.

```
static class Basis {
   String name = "basis";

   String getName() { //Um auf verdeckte Eigenschaft zugreifen zu können
      return name;
   }
}

static class Abgeleitet extends Basis {
   String name = "abgeleitet"; //Hier wird die Eigenschaft name der Basisklasse verdeckt

   void ausgeben() {
      System.out.println("Zugriff aus Abgeleitet: ");
      System.out.println("eigener  Name = " + name);       // abgeleitet
      System.out.println("geerbter Name = " + getName());  // Basis, da getName() aus ⤸
      Basis
      System.out.println("geerbter Name = " + super.name); // zweite Möglichkeit mit super
   }
}

public static void main(final String[] args) {

   final Abgeleitet obj = new Abgeleitet();

   obj.ausgeben();

   System.out.println("\nZugriff ueber Objekt:");
   System.out.println("eigener  Name = " + obj.name);           // Abgeleitet
   System.out.println("geerbter Name = " + ((Basis) obj).name); // Basis
   System.out.println("geerbter Name = " + obj.getName());      // Basis
}
```

```
Zugriff aus Abgeleitet:
eigener  Name = abgeleitet
geerbter Name = basis
geerbter Name = basis

Zugriff ueber Objekt:
eigener  Name = abgeleitet
geerbter Name = basis
geerbter Name = basis
```

Überschreibung

Auch Methoden können verdeckt werden. Allerdings spricht man dann nicht von Verdeckung, sondern von Überschreibung.

Die Überschreibung ist für die Vererbung von großer Bedeutung, denn sie erlaubt dem Entwickler der abgeleiteten Klasse das von der Basisklasse geerbte Verhalten durch Implementierung eigener Methodendefinitionen anzupassen.

Die überschreibende Methode muss den gleichen Namen, die gleiche Signatur (Argumente) und den gleichen Rückgabetyp wie die überschriebene Methode haben. Auch andere Teile des Methodenkopfs (`throws`-Definition, Zugriffsmodifizierer und Modifizierer) müssen weitestgehend identisch übernommen werden. Der Zugriff darf bei der Überschreibung aufgelockert, aber nicht verstärkt werden. Die überschreibende Methode verdeckt in der abgeleiteten Klasse die gleichnamige geerbte Methode.

```
static class Zaehler {
   protected int wert;

   public Zaehler() {
      wert = 0;
   }

   public int abfragen() {
      return wert;
   }

   // Wird von abgeleiteter Klasse überschrieben.
   public void weiterdrehen() {
      ++wert;
   }
}

static class ZweiSchrittZaehler extends Zaehler {

   public ZweiSchrittZaehler(final int startwert) {
      wert = startwert;
   }

   // Überschriebene Methode mit angepasster Implementierung
   @Override
   public void weiterdrehen() {
      wert += 2;
   }
}

public static void main(final String[] args) {

   final ZweiSchrittZaehler zaehler = new ZweiSchrittZaehler(10);

   for(int i = 0; i <= 10; ++i) {
       System.out.print(zaehler.abfragen() + " ");
       zaehler.weiterdrehen();
   }
}
```

```
10 12 14 16 18 20 22 24 26 28 30
```

Überladung

Geerbte Methoden können in der abgeleiteten Klasse auch überladen werden. Dies bedeutet nichts anderes, als dass mehrere Methoden mit dem gleichen Namen definiert werden können. Voraussetzung ist allerdings, dass die Methoden anhand ihrer Signatur (Parameterliste ...) unterscheidbar sind.

```
static class Basis {
   protected int wert = 1;

   // überladene Methode mit verschiedenen Parameterlisten
   void erhoehen() {
      ++wert;
   }
}

static class Abgeleitet extends Basis {

   // überladene Methode mit verschiedenen Parameterlisten
   void erhoehen(final int betrag) {
      wert += betrag;
   }
}

public static void main(final String[] args) {

   final Abgeleitet obj = new Abgeleitet();

   obj.erhoehen();
   System.out.println("Wert = " + obj.wert);

   obj.erhoehen(12);
   System.out.println("Wert = " + obj.wert);
}
```

```
Wert = 2
Wert = 14
```

Vererbung und objektorientiertes Design

Wann ist Vererbung gerechtfertigt?

Vererbung ist nicht allein ein Mechanismus um bestehenden Code in einer Klasse wiederzuverwenden. Sie setzt vielmehr Folgendes voraus:

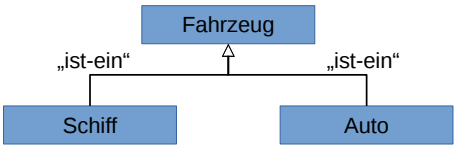
- die abgeleitete Klasse ist eine Weiterführung oder Spezialisierung der Basisklasse oder umgekehrt,
- die Basisklasse ist eine allgemeinere (abstraktere) Beschreibung der Objekte der abgeleiteten Klasse.

Letztere Bedingung wird meistens durch die „Ist ein"-Frage überprüft.

Ist ein Objekt der abgeleiteten Klasse auch ein Objekt der Basisklasse?

Wenn ja, dann ist die Vererbung sinnvoll.

Das ist im Prinzip der einfachste Fall, in dem man Vererbung benutzt. Man verfügt bereits über eine Klasse, die die Grundfunktionalität zur Verfügung stellt. Der Klasse fehlen aber bestimmte Felder und Methoden. Anstatt eine ganz neue Klasse aufzusetzen, leitet man von der Basisklasse ab und ergänzt in der abgeleiteten Klasse die zusätzlich benötigten Elemente.



Ein zweiter wichtiger Fall ergibt sich, wenn zwei Klassen etliche Elemente gemeinsam haben, jedoch keine „Ist ein"-Beziehung besteht. Das könnte daran liegen, dass die beiden Klassen sozusagen „Geschwister" sind. Somit könnten beide Klassen auf eine gemeinsame Basisklasse zurückgeführt werden.

Vererbung versus Komposition

Das Pendant zur Vererbung ist die Komposition oder Einbettung. Komposition bedeutet, dass eine Klasse Felder definiert, die vom Datentyp einer anderen Klasse stammen. Während die Vererbung eine „Ist-ein"-Beziehung darstellt entspricht die Komposition einer „Hat ein"-Beziehung.

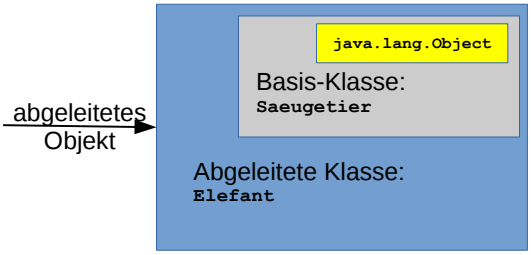
Klassenhierarchien

Grundsätzlich kann jede Klasse als Basisklasse fungieren. Es sei denn, sie wurde als `final` definiert. Daraus folgt zugleich, dass abgeleitete Klassen selbst auch wieder als Basisklassen für weitere Klassen dienen können. Da von einer Basisklasse beliebig viele Klassen abgeleitet werden dürfen, können auf diese Weise ganze Klassenhierarchien entstehen.

Mehrfachvererbung

Von einer Basisklasse können mehrere Klassen abgeleitet werden, aber eine abgeleitete Klasse kann immer nur **eine direkte** Basisklasse haben. In Java gibt es keine Mehrfachvererbung!
Um trotzdem die Vorteile der Mehrfachvererbung nutzen zu können, ohne dabei die Nachteile in Kauf zu nehmen, stellt Java Interfaces zur Verfügung.

8.3.2 Polymorphie



Beim Zugriff auf vererbte Elemente enthält jedes Objekt einer abgeleiteten Klasse auch ein Objekt seiner Basisklasse, ein Basisklassenunterobjekt. Jedes Basisklassenunterobjekt ist selbst wieder ein Unterobjekt seiner Basisklasse bis hin zum abschließenden `Object`-Unterobjekt.

Dieser Aufbau erlaubt es uns, Objekte einer abgeleiteten Klasse auch als Objekte ihrer Basisklasse zu behandeln. Insbesondere kann man ein Objekt vom Typ einer abgeleiteten Klasse in einer Objektvariablen vom Typ einer zugehörigen Basisklasse abspeichern. Der Zugriff auf Elemente abgeleiteter Klassen ist über eine Referenz vom Typ einer Basisklasse nicht möglich.

Beispiel

```
static class Saeugetier {
   void steckbrief() {
      System.out.println("Ich bin ein Saeugetier.");
   }
}

static class Elefant extends Saeugetier {
   void trompeten() {
      System.out.println("Törrrröööööö!");
   }
}

static class Gepard extends Saeugetier {
   void fauchen() {
      System.out.println("Fffffffauchhhhhhh!");
   }
}

public static void main(String[] args) {

   // Polymorphe Variable
   Saeugetier saeuger = new Saeugetier();
   saeuger.steckbrief();

   // Erst Zuweisung von Gepard
   saeuger = new Gepard();
   saeuger.steckbrief();

   // Dann Zuweisung von Elefant
   saeuger = new Elefant();
   saeuger.steckbrief();
   // Der Aufruf von steckbrief() ist möglich, da diese von der Basisklasse
   // geerbt wird.
}
```

```
Ich bin ein Saeugetier.
Ich bin ein Saeugetier.
Ich bin ein Saeugetier.
```

Beim Speichern eines Objekts einer abgeleiteten Klasse in einer Objektvariablen vom Typ einer Basisklasse, wird das Objekt auf das entsprechende Basisklassenunterobjekt reduziert. Dies bedeutet, dass man über die Basisklassenvariablen nur auf die Elemente des Objekts zugreifen kann, die dieses von der Basisklasse geerbt hat.

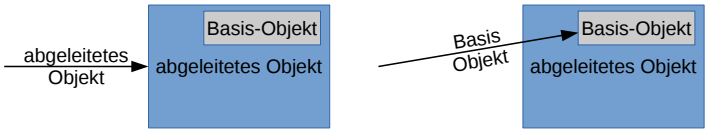


Abb. 8.9: Basisklassenunterobjekt

Die physikalische Struktur eines Objekts beim Zuweisen einer Objektvariablen einer Basisklasse bleibt trotz der Reduzierung auf das Basisklassenunterobjekts unberührt. Dies bedeutet, dass dieser Vorgang reversibel ist. Dazu muss lediglich die Objektreferenz in den Typ der abgeleiteten Klasse „gecastet“ werden.

Upcast Typumwandlung einer Objektreferenz vom Typ einer abgeleiteten Klasse in eine Basisklasse.

Downcast Typumwandlung einer Objektreferenz vom Typ einer Basisklasse in eine abgeleitete Klasse.

Bei einem Downcast muss stets eine explizite Typangabe erfolgen. Bei einem Upcast kann diese entfallen. Dies liegt daran, dass der Compiler bei einer Umwandlung von einem abgeleiteten Objekt in ein Basisklassenobjekt und einer anschließenden Umwandlung in ein anderes abgeleitetes Objekt keinen Fehler erkennen kann. Schließlich ist das zweite abgeleitete Objekt auch vom Typ der Basisklasse. Der Interpreter würde jedoch eine `ClassCastException` auslösen. Durch die explizite Typumwandlung stellt der Compiler sicher, dass sich der Programmierer über die Gefahren eines Downcasts im Klaren ist.

```
static class Saeugetier {
   void steckbrief() {
      System.out.println("Ich bin ein Saeugetier.");
   }
}

static class Elefant extends Saeugetier {
   void trompeten() {
      System.out.println("Törrrröööööö!");
   }
}

static class Gepard extends Saeugetier {
   void fauchen() {
      System.out.println("Fffffffauchhhhhhh!");
   }
}

public static void main(String[] args) {

   // Downcast immer explizit!!
   Saeugetier saeuger;
   saeuger = new Elefant();
   ((Elefant) saeuger).trompeten();
   Elefant dumbo = (Elefant)saeuger;
   dumbo.trompeten();

   // Upcast ohne Typangabe
   final Elefant elefant = new Elefant();
   saeuger = elefant;

   // Compilierbar, aber ClassCastException weil dumbo ein Elefant und
   // kein Gepard ist!!
   try {
      saeuger = new Gepard();
      dumbo = (Elefant) saeuger;
   } catch (ClassCastException e) {
      System.out.println(e.getMessage());
   }
}
```

```
Törrrröööööö!
Törrrröööööö!
Cs_ltxml_JAVA_0097$Gepard cannot be cast to Cs_ltxml_JAVA_0097$Elefant
```

Polymorphe Methoden

Von polymorphen Methoden spricht man, wenn es zu einem Methodenbezeichner mehrere Implementierungen gibt.

Dies geschieht durch:

- Überladung bzw.
- Überschreiben

Das Überladen von Methoden sollte ausreichend besprochen worden sein (mehr dazu auf Seite 106) und ist hier nicht weiter interessant. Viel spannender ist das Überschreiben von Methoden. Dies passiert, wenn eine abgeleitete Klasse eine geerbte Methode neu definiert und mit eigenem klassenspezifischen Verhalten verbindet.

Werden nun Instanzen von abgeleiteten Klassen gebildet, wird jeweils die Version der Methode gerufen, die in der entsprechenden Klasse definiert ist.

Was aber passiert, wenn eine polymorphe Methode über eine polymorphe Variable aufgerufen wird?

```
static class Saeugetier {
   void steckbrief() {
      System.out.println("Ich bin ein Saeugetier.");
   }
}

static class Elefant extends Saeugetier {
   void trompeten() {
      System.out.println("Törrrröööööö!");
   }
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Elefant.");
   }
}

static class Gepard extends Saeugetier {
   void fauchen() {
      System.out.println("Fffffffauchhhhhhh!");
   }
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Gepard.");
   }
}

public static void main(String[] args) {

   // Es wird jeweils die Methode aus dem entsprechendem Kontext gerufen
   final Saeugetier saeuger = new Saeugetier();
   saeuger.steckbrief();
   // steckbrief() ist überschrieben

   final Gepard gepard = new Gepard();
   gepard.steckbrief();
   // steckbrief() ist überschrieben
   final Elefant elefant = new Elefant();
   elefant.steckbrief();
}
```

```
Ich bin ein Saeugetier.
Ich bin ein Gepard.
Ich bin ein Elefant.
```

Der Java-Compiler ruft grundsätzlich die Methodenversion auf, die zum Typ des Objekts gehört. Lediglich Methoden, die als `private`, `static` oder `final` definiert wurden, sind an den Typ der Objektvariablen gebunden.

```
static class Saeugetier {
   void steckbrief() {
      System.out.println("Ich bin ein Saeugetier.");
   }
}

static class Elefant extends Saeugetier {
   void trompeten() {
      System.out.println("Törrrröööööö!");
   }
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Elefant.");
   }
}

static class Gepard extends Saeugetier {
   void fauchen() {
      System.out.println("Fffffffauchhhhhhh!");
   }
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Gepard.");
   }
}

public static void main(String[] args) {

   Saeugetier saeuger;
   // Java entscheidet sich grundsätzlich für die passende Methode des Objekttyps.
   // nicht die der Referenz!!!
   // Methoden die als private static oder final definiert wurden werden über die
   // Referenzvariable gefunden.
   saeuger = new Gepard();
   saeuger.steckbrief();

   saeuger = new Elefant();
   saeuger.steckbrief();
}
```

```
Ich bin ein Gepard.
Ich bin ein Elefant.
```

Methodenaufrufe in Java

Trifft der Java-Compiler auf einen Methodenaufruf, geschieht Folgendes:

- Der Compiler prüft die syntaktische Korrektheit des Methodenaufrufs.
- Der Compiler sammelt alle Methodenversionen, die für den Aufruf infrage kommen. Angefangen bei der Klasse des Objekts, über das der Methodenaufruf geschieht, bis hin zu allen Basisklassen und Schnittstellen, in denen Methoden mit gleichem Namen und passende Parameter existieren.
- Der Compiler ermittelt die geeignetste Methode und schlägt diese dem Interpreter vor. Existieren mehrere überschriebene Methodenversionen, entscheidet sich der Compiler für die Version der am weitesten abgeleiteten Klasse (maximal für die Klasse des aufrufenden Objekts). Anschließend wird der Bytecode generiert und die Methode kategorisiert (`virtual`, `nonvirtual`, `static`, ...).

Bei `virtual`-Methoden (potenziell polymorph) bestimmt der Interpreter die Klasse des Objekts (dies ist dem Compiler nicht möglich) und prüft, ob es in der Klasse eine gleichnamige Methode gibt, die die Methodendefinition aus der Klasse der aufrufenden Objektvariablen überschreibt. Findet er eine, wird sie aufgerufen. Gibt es sie nicht, so sucht er in den Basisklassen des Objekts weiter. Somit nähert sich der Interpreter immer mehr der Klasse der Objektvariablen.

```
static class Saeugetier {
   void steckbrief() {
      System.out.println("Ich bin ein Saeugetier.");
   }
}

static class Elefant extends Saeugetier {
   void trompeten() {
      System.out.println("Törrrröööööö!");
   }
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Elefant.");
   }
}

public static void main(String[] args) {

   Elefant dumbo = new Elefant();
   dumbo.steckbrief();
   ((Saeugetier)dumbo).steckbrief();
}
```

```
Ich bin ein Elefant.
Ich bin ein Elefant.
```

Polymorphes Verhalten unterdrücken

Java geht grundsätzlich davon aus, dass der Programmierer polymorphes Verhalten wünscht. Er kann polymorphes Verhalten auch nicht bestimmen. Lediglich die Definition einer Methode als `private`, `static` oder `final` verhindert einen polymorphen Aufruf.

`static` Als `static` definierte Methoden werden über den Klassennamen und nicht über die Objektvariable gerufen. Damit können sie nicht am Konzept der Polymorphie teilnehmen.

`private` Als `private` definierte Methoden können in abgeleiteten Klassen neu definiert werden. Hierbei handelt es sich jedoch weder um Überladung noch Überschreibung, sondern eben um eine neue Methode in einem anderen Gültigkeitsbereich.

`final` Der Modifizierer `final` wurde zu dem Zweck in Java aufgenommen, um eine Überschreibung zu verbieten. Somit wird auch jegliches polymorphes Verhalten unterbunden.

Sichtbarkeit ändern

Eine `protected`-Methode in der Basisklasse darf in der abgeleiteten Methoden öffentlich werden, als mit `public` gekennzeichnet werden.

```
static class Saeugetier {
   protected void steckbrief() {
      System.out.println("Ich bin ein Saeugetier.");
   }
}

static class Elefant extends Saeugetier {
   @Override
   public void steckbrief() {
      System.out.println("Ich bin ein Elefant.");
   }
}

public static void main(String[] args) {

   Elefant dumbo = new Elefant();
   dumbo.steckbrief();
   ((Saeugetier)dumbo).steckbrief();
}
```

```
Ich bin ein Elefant.
Ich bin ein Elefant.
```

Aufgaben → f_tools_poly_01
Aufgaben → f_tools_poly_02

8.3.3 Generische Programmierung

Basisklassen-Arrays

Objektvariablen sind in Java immer potentiell polymorph. Sie können sowohl auf Objekte der eigenen oder irgendeiner abgeleiteten Klasse verweisen. Dies ermöglicht es uns, Objekte aus verschiedenen abgeleiteten Klassen gemeinsam in einem Array zu verwalten.

Grundsätzlich können Arrays nur für einen bestimmten Typ definiert werden und somit auch nur diese Elemente aufnehmen. Bedient man sich aber eines Tricks und definiert ein Array für den Typ einer Basisklasse, die allen Elementen gemeinsam ist, so können in diesem Array Objekte sämtlicher abgeleiteten Klassen verwaltet werden.

```
static class Saeugetier {
   void steckbrief() {
      System.out.println("Ich bin ein Saeugetier.");
   }
}

static class Elefant extends Saeugetier {
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Elefant.");
   }
}

static class Gepard extends Saeugetier {
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Gepard.");
   }
}

public static void main(String[] args) {

   // Basisklassenarray zur Aufnahme von Saeugetieren
   final Saeugetier[] saeuger = new Saeugetier[6];

   // Zufälliges Erzeugen von Elefanten oder Geparden
   for (int i = 0; i < saeuger.length; ++i) {
      if (Math.random() < 0.5) {
         saeuger[i] = new Gepard();
      } else {
         saeuger[i] = new Elefant();
      }
   }

   System.out.println("Alle Tiere durchgehen:");
   for (int i = 0; i < saeuger.length; ++i) {
      // Die richtige Steckbriefmethode wird aufgerufen, weil
      // sie in Saeugetier definiert ist, und grundsätzlich polymorph -->
      // Java standardmäßig dynamisch gebunden!!!
      saeuger[i].steckbrief();
   }
}
```

```
Alle Tiere durchgehen:
Ich bin ein Elefant.
Ich bin ein Gepard.
Ich bin ein Elefant.
Ich bin ein Elefant.
Ich bin ein Gepard.
Ich bin ein Elefant.
```

Basisklassenparameter

Die Reduzierung abgeleiteter Objekte auf ihre Basisklassenunterobjekte erlaubt uns außerdem, die Implementierung generischer Methoden, die als Argumente Objekte beliebiger abgeleiteter Klassen akzeptieren.

```
static class Saeugetier {
   private final boolean raeuber;
   Saeugetier(final boolean raeuber) {
      this.raeuber = raeuber;
   }
   boolean getRaeuber() {
      return raeuber;
   }
   void steckbrief() {
      System.out.print("Ich bin ein Saeugetier. ");
   }
}

static class Elefant extends Saeugetier {
   Elefant() {
      super(false);
   }
   @Override
   void steckbrief() {
      System.out.print("Ich bin ein Elefant. ");
   }
}

static class Gepard extends Saeugetier {
   Gepard() {
      super(true);
   }
   @Override
   void steckbrief() {
      System.out.print("Ich bin ein Gepard.  ");
   }
}

static void alarmieren(final Saeugetier tier) {
   if (tier.getRaeuber()) {
      System.out.println("  Alarm! Ein Raeuber ist unterwegs.");
   } else {
      System.out.println("  Entwarnung! Nur ein harmloser Pflanzenfresser.");
   }
}

public static void main(final String[] args) {

   final Saeugetier[] saeuger = new Saeugetier[6];
   for (int i = 0; i < saeuger.length; ++i) {
      if (Math.random() < 0.5) {
         saeuger[i] = new Gepard();
      } else {
         saeuger[i] = new Elefant();
      }
   }
   System.out.println("Alle Tiere durchgehen:");
   for (int i = 0; i < saeuger.length; ++i) {
      saeuger[i].steckbrief();
      alarmieren(saeuger[i]);
   }
}
```

```
Alle Tiere durchgehen:
Ich bin ein Gepard.    Alarm! Ein Raeuber ist unterwegs.
Ich bin ein Gepard.    Alarm! Ein Raeuber ist unterwegs.
Ich bin ein Gepard.    Alarm! Ein Raeuber ist unterwegs.
Ich bin ein Elefant.   Entwarnung! Nur ein harmloser Pflanzenfresser.
Ich bin ein Gepard.    Alarm! Ein Raeuber ist unterwegs.
Ich bin ein Elefant.   Entwarnung! Nur ein harmloser Pflanzenfresser.
```

Typidentifizierung zur Laufzeit (RTTI)

Die beste Laufzeittypidentifizierung (**R**un**t**ime **T**ype **I**dentification) besteht darin, die Typidentifizierung dem Compiler zu überlassen, indem man überschriebene Methoden benutzt. Die Grenze beim Arbeiten mit polymorphen Objekten und ihren polymorphen Methoden ist genau dann erreicht, wenn man über eine Basisklassenvariable, die auf ein abgeleitetes Objekt verweist, auf ein Element der abgeleiteten Klasse zugreifen möchte, das nicht von der Basisklasse geerbt oder überschrieben wurde.

Ist der Typ des Objekts bekannt, so kann mittels Downcast das richtige Element ermittelt werden.

In den Fällen, in denen der Typ erst zur Laufzeit bekannt ist, muss der richtige Objekttyp ermittelt und dann umgewandelt werden.

Klassentyp eines Objekts ermitteln - instanceof

Der `instanceof`-Operator verhält sich genauso wie der Versuch einer impliziten oder expliziten Typumwandlung, nur dass die Typumwandlung simuliert und ihr Ergebnis als boolescher Wert zurückgeliefert wird.

```
static class Saeugetier {
   void steckbrief() {
      System.out.print("\nIch bin ein Saeugetier. ");
   }
}

static class Elefant extends Saeugetier {
   @Override
   void steckbrief() {
      System.out.print("\nIch bin ein Elefant.    ");
   }
}

static class Gepard extends Saeugetier {
   @Override
   void steckbrief() {
      System.out.print("\nIch bin ein Gepard.     ");
   }
}

public static void main(String[] args) {

   Saeugetier saeuger = new Saeugetier();
   saeuger.steckbrief();
   System.out.print(saeuger instanceof Saeugetier ? " Saeugetier" : "");
   System.out.print(saeuger instanceof Elefant ? " Elefant" : "");
   System.out.print(saeuger instanceof Gepard ? " Gepard" : "");

   final Gepard gepard = new Gepard();
   gepard.steckbrief();
   System.out.print(gepard instanceof Saeugetier ? " Saeugetier" : "");
   // System.out.print(gepard instanceof Elefant ? " Elefant" : "");
   // Compilerfehler: Incompatible conditional operand types Gepard and Elefant
   System.out.print(gepard instanceof Gepard ? " Gepard" : "");

   final Elefant elefant = new Elefant();
   elefant.steckbrief();
   System.out.print(elefant instanceof Saeugetier ? " Saeugetier" : "");
   System.out.print(elefant instanceof Elefant ? " Elefant" : "");
   // System.out.print(elefant instanceof Gepard ? " Gepard" : "");
   // Compilerfehler: Incompatible conditional operand types Elefant and Gepard
}
```

```
Ich bin ein Saeugetier.  Saeugetier
Ich bin ein Gepard.      Saeugetier Gepard
Ich bin ein Elefant.     Saeugetier Elefant
```

Beliebige Objekte und Klassen vergleichen – getClass()

Bei der Rückumwandlung von polymorphen Objektvariablen ist der `instanceof`-Operator immer die erste Wahl. Sollten jedoch Typen verglichen werden, bei denen der `instanceof`-Operator nicht greift, kann man auf die Methode `getClass()` aus der Klasse `Object` zurückgreifen.

- Zwei Objekte vergleichen
  `if(obj1.getClass() == obj2.getClass())`
- Objekt mit einem Klassennamen vergleichen
  `if(obj1.getClass() == KLASSENNAME.class)`

```
static class Saeugetier {
   void steckbrief() {
      System.out.println("Ich bin ein Saeugetier.");
   }
}
static class Elefant extends Saeugetier {
   void trompeten() {
      System.out.println("Törrrröööööö!");
   }
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Elefant.");
   }
}
static class Gepard extends Saeugetier {
   void fauchen() {
      System.out.println("Fffffffauchhhhhhh!");
   }
   @Override
   void steckbrief() {
      System.out.println("Ich bin ein Gepard.");
   }
}
public static void main(String[] args) {

   final Saeugetier[] saeuger = new Saeugetier[4];
   for (int i = 0; i < saeuger.length; ++i) {
      if (Math.random() < 0.5) {
         saeuger[i] = new Gepard();
      } else {
         saeuger[i] = new Elefant();
      }
   }
   System.out.println("Alle Tiere geben Laut:");
   for (int i = 0; i < saeuger.length; ++i) {
      if (saeuger[i].getClass() == Gepard.class) {
         ((Gepard) saeuger[i]).fauchen();
      } else {
         ((Elefant) saeuger[i]).trompeten();
      }
   }
   System.out.println("\nAlle Tiere geben Laut:");
   for (int i = 0; i < saeuger.length; ++i) {
      if (saeuger[i] instanceof Gepard) {
         ((Gepard) saeuger[i]).fauchen();
      } else {
         ((Elefant) saeuger[i]).trompeten();
      }
   }
}
```

```
Alle Tiere geben Laut:
Fffffffauchhhhhhh!
Törrrröööööö!
Törrrröööööö!
Fffffffauchhhhhhh!

Alle Tiere geben Laut:
Fffffffauchhhhhhh!
Törrrröööööö!
Törrrröööööö!
Fffffffauchhhhhhh!
```

8.3.4 Java Generics

Klassen parametrisieren

Eine weitere Säule der generischen Programmierung stellen sogenannte generische Datentypen oder auch „Java Generics“ dar. Sie erlauben es dem Programmierer, Klassen zu definieren, ohne sich auf die in der Klasse verwendeten Datentypen festlegen zu müssen.

Die Objekte, die aus parametrisierten Klassen erzeugt werden, sind streng typisiert. Man kann zwar Objekte für beliebige Typen erzeugen, jedoch sind sie dann an den Typ gebunden, mit dem sie typisiert wurden. Genau umgekehrt wäre es, wenn Sie eine Klasse auf der Basis des `object`-Datentyps implementieren. Da dies der oberste Basisklassentyp ist, könnten dann beliebige Typen in der Klasse benutzt werden.

Syntax

Ein parametrisierter Datentyp ist eine Klasse, die Platzhalter für mindestens einen intern verwendeten Datentyp definiert. Als Platzhalter kommen hierbei jedoch lediglich Referenztypen in Frage. Primitive Datentypen können nicht verwendet werden.

Die Platzhalter werden in spitzen Klammern an den Namen der Klasse angehängt. Wollen Sie verschiedene Typen durch einen Platzhalter repräsentieren, so muss für jeden Typ ein eigener Platzhalter definiert werden.

Definition der Klasse:

```
class GenerischeKlasse<T> {
   private T feld;

   GenerischeKlasse(T param) {
      feld = param;
   }

   T getFeld() {
      return feld;
   }
}
```

Verwendung:

```
GenerischeKlasse<String> obj1 = new GenerischeKlasse<String>();
```

Diamant Operator

Bei der Definition von Variablen mit einem generischen Datentyp verlangt die Java-Syntax offensichtlich eine redundante Wiederholung der Typangabe.

```
Map<String, String> daten = new HashMap<String, String>();
```

Ab der Version 7 kann stattdessen eine verkürzte Variablendefinition mit dem „Diamant“-Operator[12] benutzt werden.

```
Map<String, String> daten = new HashMap<>();
```

Methoden parametrisieren

Auch Methoden können unabhängig von den Klassen, in denen sie implementiert sind, parametrisiert sein. In diesem Fall taucht der Typparameter in der Signatur der jeweiligen Methode auf, nämlich in spitzen Klammern vor dem Rückgabetyp.

```
modifizierer <T> rückgabetyp methodenname(parameter){ … }
```

[12] <> - mit viel Fantasie erinnert diese Schreibweise an einen Diamanten.

Wildcard-Typ und Einschränkungen

Eine wichtige Grundregel beim Arbeiten mit Generics lautet:

Wenn eine Klasse A von Klasse B abgeleitet ist, und weiterhin eine generische Klasse C definiert ist, gilt NICHT, das C<A> ein Subtyp von C<B> ist!

Beispiel:

```
class Mensch{…

   String leseDaten(){ … }
}
class Mann extends Mensch{…

}
class Frau extends Mensch{…

}
LinkedList<Mann> lauefer = new LinkedList<Mann>();
LinkedList<Frau> laeuferinnen = new LinkedList<Frau>();
LinkedList<Mensch> alle = new LinkedList<Mensch>();…

   void listeAusgeben(LinkedList<Mensch> liste){
      for(Mensch i : liste){
         System.out.println(i.leseDaten());
      }
   }
listeAusgeben(laeufer);
// Erzeugt einen Compilerfehler, da laeufer als Linked
// List<Mann> definiert ist und dies ist eben kein
// Subtyp von LinkedList<Mensch>.
```

Der Wildcard-Typ

Abhilfe bei diesem Problem schafft der Wildcard-Typ „?“. Das „?“ steht dabei für einen beliebigen, unbekannten Typ. Eine Implementierung der obigen Methode `listeAusgeben()` würde dann wie folgt aussehen:

```
void listeAusgeben(LinkedList<?> liste){
   for(Object i : liste){
      System.out.println( ((Mensch)i).leseDaten() );
   }
}
```

Da das „?“ für jeden beliebigen Datentyp stehen kann, können „?“-Objekte ausschließlich Variablen vom Typ `Object` zugewiesen werden. Deshalb muss in der `for`-Schleife mit `Object` gearbeitet und anschließend umständlich nach Mensch gecastet werden.

Arrays von parametrisierten Klassen

Soll ein Array Objekte einer parametrisierten Klasse aufnehmen, muss es zwangsweise mit einem Wildcard-Typ erzeugt werden.

Eingeschränkte Wildcards

Ein großer Nachteil beim Einsatz von Wildcards ist, dass Wildcard-Typ-Parameter uneingeschränkt jede typisierte Form der zugehörigen parametrisierten Klasse akzeptieren. Im obigen Beispiel könnte man also der Methode `listeAusgeben()` auch ein Argument vom Typ `LinkedList<String>` übergeben. Zur Kompilierzeit würde dann allerdings die Konvertierung in Mensch innerhalb der `for`-Scheife fehlschlagen.

Abhilfe schafft hier die sogenannte eingeschränkte Wildcard.

```
? extends Klassenname [&] [Schnittstellenname] [&] [Schnittstellenname_2]
```

legt die obere Grenze fest.

```
? super Klassenname [&] [Schnittstellenname] [&] [Schnittstellenname_2]
```

legt die untere Grenze fest.

Eingeschränkte Platzhalter

Die „extends“-Schranke lässt sich außerdem auch auf Platzhalterdefinitionen in einer Klasse anwenden. Uneingeschränkte Platzhalter wie `eineKlasse<T>` können bei der Instanziierung durch jeden beliebigen Objekttyp ersetzt werden. Oft ist es jedoch wünschenswert, dass eine parametrisierte Klasse nur Objekte einer konkreten Klassenhierarchie verarbeiten kann.

Vererbung und Überladung

Beim Ableiten einer beliebigen Klasse von einer parametrisierten Klasse muss entweder angegeben werden, von welcher Typisierung der Basisklasse man ableiten möchte, oder man definiert Platzhalter der Basisklasse hinter der abgeleiteten Klasse und reicht sie an die Basisklasse weiter.

Beispiel:

```
class Basis<T> { … }
class Abgeleitet extends Basis<EinTyp> { … }

class Basis <T> { … }
class Abgeleitet<T> extends Basis<T> { … }

class Basis<T extends EineKlasse> { … }
class Abgeleitet <T extends EineKlasse> extends Basis<T> { … }
```

Ob man den (oder die) Platzhalter der Basisklasse in der abgeleiteten Klasse benutzt, bleibt einem selbst überlassen. Selbstverständlich kann man auch für die abgeleitete Klasse eigene Platzhalter definieren.

Überladung von Methoden mit Platzhaltern

Beim Überladen von parametrisierten Methoden muss zwingend darauf geachtet werden, dass es nicht zu Zweideutigkeiten kommt.

```
class Klasse<T>{…

   void eineMethode(T param) { … }
   void eineMethode(Integer param) { … }
}
```

Wird die Klasse mit `<T>` als Typ `Integer` definiert, so kommt es zu einem Fehler, da der Compiler nicht mehr entscheiden kann, welche Implementierung von `eineMethode()` er aufrufen soll.

Einschränkungen

Für jede parametrisierte Klasse existiert nur eine Implementierung. Egal, wie viele unterschiedliche Datentypen typisiert sind. Daraus folgt:

- Mit `instanceof` kann man prüfen, ob ein Objekt einer parametrisierten Klasse angehört, nicht jedoch für welchen Typ es parametrisiert wurde.
- Die Methode `getClass()` liefert für alle Objekte einer parametrisierten Klasse dieselbe Referenz. Sie kann somit ebenso nicht zur Unterscheidung von Typisierungen eingesetzt werden.

8.3.5 Abstrakte Klassen und Interfaces

Die Vererbung erlaubt es uns, Code aus Basisklassen in abgeleiteten Klassen zu verwenden. Durch das Konzept der Polymorphie ist nun klar, dass es genauso sinnvoll sein kann, eine einheitliche Schnittstelle zu vererben. Dies wird nämlich durch die Verwendung von polymorphen Methoden, die in den abgeleiteten Klassen überschrieben werden, realisiert: Der geerbte Name (Schnittstelle) wird übernommen, die Definition (der Code) wird ersetzt. Führt man diesen Gedanken fort, kommt man zu Konzepten, die ausschließlich der Vererbung von Schnittstellen dienen – den abstrakten Klassen und schließlich den Java-Schnittstellen (Interfaces).

Beispiel:

```
class Kreis {
      private double radius = 1.0;

      public Kreis()            { }
      public Kreis(double r)    { this.radius = r; }

      public double getRadius() { return radius; }
      public void setRadius(double r) {
         if (r > 0) {
            this.radius = r;
         }
      }
      public double umfang()    { return 2 * Math.PI * radius; }
      public double fläche()    { return Math.PI * radius * radius; }
}
```

```
class Rechteck  {
      private double breite = 1.0;
      private double laenge = 1.0;

      public Rechteck() { }
      public Rechteck(double b, double l) {
         this.breite = b;
         this.länge = l;
      }
      public double getBreite() { return breite; }
      public double getLänge()  { return länge; }
      }
      public void setBreite(double b) {
         if (b > 0) {
            this.breite = b;
         }
      }
      public void setLänge(double l) {
         if (l > 0) {
            this.laenge = l;
         }
      }

      public double umfang()    { return 2 * (länge + breite); }
      public double fläche()    { return länge * breite; }
}
```

Nachteil bei dieser Implementierung ist, dass es keine gemeinsame Basisklasse gibt. Möchte man beispielsweise Kreis- und Rechteckobjekte zusammen in einem Array verwalten, muss man dazu ein Array von `Object`-Elementen definieren.

```
Object[] formen = new Object[3];
formen[0] = new Kreis(3);
formen[1] = new Rechteck(10, 2);
formen[2] = new Kreis(5);
```

Die Vorteile der Polymorphie sind somit nicht nutzbar. Möchte man auf die spezielle Funktionalität der Kreis- bzw. Rechteckklassen zugreifen, so müssen die Objektreferenzen in die abgeleiteten Klassen zurückverwandelt werden.

Beispiel: Berechnung der Gesamtfläche aller Elemente des Arrays

```
double gesamtFläche = 0.0;
for(int i = 0; i < formen.length; i++){
   if(formen[i] instanceof Kreis) {
      gesamtFläche += ((Kreis) formen[i]).fläche();
   } else {
      gesamtFläche += ((Rechteck) formen[i].fläche();
```

```
   }
}
```

Zur Verbesserung definieren wir eine gemeinsame Basisklasse Form, leiten die Klassen `Rechteck` und `Kreis` davon ab und definieren dort bereits die Methoden `umfang()` und `fläche()`.

```
Form[] formen = new Object[3];
formen[0] = new Kreis(3);
formen[1] = new Rechteck(10, 2);
formen[2] = new Kreis(5);

double gesamtFläche = 0.0;
for(int i = 0; i < formen.length; i++) {
   gesamtFläche += formen[i].fläche();
}
```

Nun kann das Basisklassen-Array mit dem Typ Form definiert werden, und die polymorphen Methoden zur Berechnung der Gesamtfläche herangezogen werden.

So weit so gut. Diese Implementierung weist jedoch immer noch zwei gravierende Fehler auf:

- Die von Form vererbten Methoden haben keine sinnvolle Implementierung. Sollte die Klassenhierarchie um die Klasse Dreieck erweitert werden und dabei vergessen werden, die Methoden `fläche()` und `umfang()` zu überschreiben, so führt dies zwangsläufig zu fehlerhaften Berechnungen.
- Die Klasse Form kann instanziiert werden. Dies ist jedoch absolut unsinnig, da diese Objekte nicht einsetzbar wären.

Abstrakte Methoden

Durch die Definition einer Methode mit dem Schlüsselwort `abstract` kann diese ohne Anweisungsteil existieren.

```
abstract typ methodenname(parameter);
```

Diese Variante wird genutzt um Methodennamen als Schnittstellenvorgabe an andere Klassen zu vererben. Aufgabe der abgeleiteten Klassen ist es nun, die Methoden zu überschreiben und mit einem Anweisungsteil auszustatten.

Abstrakte Klassen

Eine Klasse, die mindestens eine abstrakte Methode enthält, bezeichnet man als abstrakte Klasse. Diese muss ebenfalls mit dem Schlüsselwort `abstract` definiert werden.

```
abstract class klassenname {…

}
```

Es ist nicht möglich, Instanzen von abstrakten Klassen zu erzeugen. Sie sind lediglich als Basisklasse einsetzbar. Eine Klasse, die von einer abstrakten Klasse abgeleitet wird, verpflichtet sich, alle abstrakten Methoden der Basisklasse zu überschreiben. Tut sie dies nicht, so wird sie ebenso zu einer abstrakten Klasse und muss dementsprechend definiert werden.

Interfaces

In Java kann eine Klasse von einer anderen Klasse abgeleitet werden. Diese kann wiederum von einer dritten Klasse abgeleitet sein und so fort. In der Realität finden sich jedoch meist komplexere Erbhierarchien. Wir selbst haben unsere Eigenschaften von Vater und Mutter geerbt. Ein Klavier kann zum einen als Möbelstück und als Musikinstrument angesehen werden. Man spricht in diesem Fall von Mehrfachvererbung. Das heißt, eine Klasse wird von zwei oder mehreren Klassen abgeleitet. **Java verbietet das Konzept der Mehrfachvererbung jedoch strikt.** Als Ersatz bietet uns Java ein Konzept, das die Probleme der Mehrfachvererbung umgeht.

Die Schnittstellen

Schnittstellen kann man sich als abstrakte Klassen vorstellen, die einzig und alleine abstrakte `public`-Methoden enthalten.

```
interface Umkehrbahr{
   Object umkehren(); //Elemente von Schnittstellen sind automatisch public
}
```

Klassen können von Schnittstellen erben. Hierdurch verpflichten sie sich, alle in der Schnittstelle deklarierten Elemente zu implementieren.

```
class Text implements Umkehrbar {
   //muss die Methode umkehren implementieren
}
```

Schnittstellen vs. Klassen

- Schnittstellen werden mit dem Schlüsselwort `interface` definiert.
- Schnittstellen enthalten nur Methodendefinitionen ohne Anweisungsblock, statische Konstanten und Typdefinitionen.
- Alle Elemente einer Schnittstelle sind automatisch `public`.
- Schnittstellen haben keinen Konstruktor.
- Schnittstellen können von anderen Schnittstellen abgeleitet werden.
- Klassen können mehrere Schnittstellen implementieren.

Schnittstellen definieren

Schnittstellen werden mit dem Schlüsselwort `interface` definiert. Anschließend folgt per Konvention die Eigenschaft oder Fähigkeit, die der Schnittstelle verliehen werden soll (`Comparable`, `Sortable`, `Zeichenbar`, ...). In den geschweiften Klammern werden die Schnittstellenelemente aufgeführt.

```
interface Zeichenbar{
   void zeichnen();
}
```

Elemente einer Schnittstelle können sein:

- Abstrakte `public`-Methoden ohne Anweisungsteil. Eine explizite Angabe der Schlüsselwörter `abstract` und `public` ist nicht notwendig.
- `public static final`-Konstanten. Die Angabe der Schlüsselwörter entfällt auch hier. Allerdings müssen Konstanten in der Schnittstelle initialisiert werden. Klassen, die die Schnittstelle implementieren, können direkt auf die Konstanten zugreifen.
- Typdefinitionen für Klassen und Schnittstellen. Definierte Typen sind automatisch `public` und `static`.

Schnittstellen vererben

Schnittstellen können genauso wie Klassen von einer Basisschnittstelle abgeleitet werden. Anders als Klassen können sie sogar mehrere Basisschnittstellen haben.

```
interface Zeichenbar extends Darstellbar, Comparable{
   void zeichnen();
}
```

Erbt eine Schnittstelle ein Feld gleichen Namens auf verschiedenen Vererbungswegen, so gibt der Compiler eine Fehlermeldung aus. Werden mehrere gleichnamige Felder jedoch über eine Vererbungslinie geerbt, so verdecken sie sich. Geerbte Methoden können in abgeleiteten Schnittstellen überladen und überschrieben werden.

Schnittstellen und Polymorphie

Schnittstellen unterstützen ebenfalls Polymorphie:

- Variablen vom Typ einer Schnittstelle können auf alle Objekte verweisen, deren Klasse die Schnittstelle implementiert.
- Über die Schnittstellentypvariable können die in der Schnittstelle definierten und geerbten Methoden des Objekts aufgerufen werden, wobei stets die im Objekttyp definierten Methodenversionen ausgeführt werden.

Mehrere Schnittstellen implementieren

Eine Klasse kann zwar nur eine direkte Basisklasse haben, sie kann aber sehr wohl mehrere Schnittstellen implementieren.

```
class Klassenname extends Basisklasse implements Schnittstelle1, Schnittstelle2 {…

}
```

Schnittstellen durch anonyme Klassen implementieren

Generische Methoden, die darauf angewiesen sind, dass die ihnen übergebenen Objekte eine bestimmte Funktionalität aufweisen, definieren ihre Parameter meist von Schnittstellentypen, in denen die gewünschten Methoden deklariert sind. So kann gewährleistet werden, dass alle an seine Methoden übergebenen Objekte über die gewünschte Funktionalität verfügen.

Beispielsweise müssen Ereignisbehandlungsmethoden zu Steuerelementen eingerichtet werden. Hierzu wird die Methode `addActionListener()`, die einen Parameter vom Typ `ActionListener` definiert, verwendet. `ActionListener` ist eine Schnittstelle, die eine einzige Methode namens `actionPerformed()` definiert.

Der Programmierer muss nun eine Klasse definieren, die `ActionListener` implementiert. Dabei muss die Methode `actionPerformed()` definiert werden, die den Code beinhaltet, der bei entsprechendem Ereignis ausgeführt werden soll. Nun wird ein Objekt der Klasse erzeugt und als Argument der Methode `addActionListener()` übergeben.

```
class Test{…

   JButton btn = new JButton();
   MyActionListener listen = new MyActionListener();
   btn.addActionListener(listen);…

   class MyActionListener implements ActionListener{
      @Override
      public void actionPerformed(ActionEvent e){…

      }
   }…

}
```

Dieses doch sehr umständliche Verfahren kann abgekürzt werden, indem man in einem Schritt eine anonyme Klasse definiert, instanziiert und das Objekt an die `addActionListener()`-Methode übergibt.

```
class Test{…

   JButton btn = new JButton();
   btn.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){…

      }
   );…

}
```

8.4 Einführung in Lambda

Mit Lambda-Ausdrücken[13] (die Kurzform wird gerne Lambdas genannt) wird ein Teil der funktionalen Programmierung in Java aufgenommen.

Dadurch lassen sich einige Aufgaben, wie beispielsweise innere anonyme Klassen, Verarbeitungen von Datencontainern wie Arrays, Collections, etc. und Parallelverarbeitung deutlich vereinfachen und mit weniger Code umsetzen.

8.4.1 Ausdruck

Lambdas kommen aus der funktionalen Programmierung und stellen einen Behälter für Sourcecode, ähnlich wie bei Methoden, jedoch ohne Namen oder ohne explizite Angabe eines Rückgabetyps, dar.

Der allgemeine Aufruf sieht dabei wie folgt aus:

```
(Parameter-Liste) -> { Ausdruck oder Anweisungen }
```

Beispiele

Verwendet man eine GUI, z. B. mit Swing, so wird die Aktion für einen Button über den `ActionListener` geregelt.

```
      final JButton klick = new JButton("klick");
      klick.addActionListener(new ActionListener() {

         @Override
         public void actionPerformed(final ActionEvent e) {
            System.exit(0);
         }
      });
```

Da das Interface `ActionListener` nur eine Methode hat (hier `actionPerformed()`), kann diese durch einen Lambda-Ausdruck ersetzt werden.

```
      final JButton klick = new JButton("klick");
      klick.addActionListener((e) -> {
         System.exit(0);
      });
```

Auch beim Sortieren von Arrays und Containern kann Lambda verwendet werden.

```
// Sortierung mit Comparator
final List<String> namen = Arrays.asList("Uschi", "Michael", "Peter",
                           "Heidi");

Collections.sort(namen, new Comparator<String>() {
   @Override
   public int compare(final String s1, final String s2) {
      return s1.compareToIgnoreCase(s2);
   }
});

// Ausgabe
final Iterator<String> it = namen.iterator();
while (it.hasNext()) {
   System.out.print(it.next() + ", ");
}
```

```
Heidi, Michael, Peter, Uschi,
```

[13] Es wird auch gerne der Begriff Closures verwendet.

Mit Lambda:

```
// Sortierung mit Comparator
final List<String> namen = Arrays.asList("Uschi", "Michael", "Peter",
                           "Heidi");

Collections.sort(namen, (s1, s2) -> s1.compareToIgnoreCase(s2));

// Ausgabe
namen.forEach(element -> System.out.print(element + ", "));
```

```
Heidi, Michael, Peter, Uschi,
```

Die Ausgabe über `forEach()` wird etwas später erläutert.

Die Vorteile sind u. a.:

- kürzerer Code
- Es muss keine innere anonyme Klasse erstellt werden.
- Weniger Arbeit beim Übersetzen und kein weiteres Laden der inneren anonymen Klasse für den „ClassLoader“.

Die Syntax von Lamdas kann sehr knapp und kurz gehalten werden, daher muss der Compiler viele Informationen selbst ermitteln.

Es stellt sich nun die Frage, wie der Compiler die richtige Methode findet?

Dies ist möglich, da es eine Menge an Interfaces gibt, die nur eine einzige Methode haben, beispielsweise `Runnable`, `Callable`, `AutoCloseable`, `Comparable`, `Iterable` ...

Diese Interfaces beschreiben die Funktionalität (Funktionstypen) und durch die eindeutige Signatur der Methode (Parametertypen, Returntyp und Exception) kann der Compiler die fehlenden Werte bestimmen. Man bezeichnet dies als Functional Interface Types bzw. SAM Types (Single Abstract Method).

Diese werden mit der Annotation `@FunctionalInterface` versehen.

Betrachten wir vorheriges Beispiel, so verwendet der Compiler nachfolgendes Interface und bestimmt daraus die entsprechenden Werte.

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);
   // ...
}
```

8.4.2 Parameterliste

Die Parameterliste ist entweder eine kommagetrennte Liste in runden Klammern oder ein einzelner Bezeichner ohne runde Klammern.

Verwendet man runde Klammern, so kann man zusätzlich auch den Parametertyp für alle Parameter angeben. Ohne Angabe des Typs versucht der Compiler automatisch diesen zu bestimmen.

- `(int x) -> x + 1`
  Ein Parameter mit expliziter Typangabe.
- `int x -> x + 2`
  Runde Klammern müssen bei Angabe des Parametertyps angegeben werden.
- `(x) -> x + 3`
  Ein Parameter ohne explizite Typangabe – der Compiler ermittelt (deduziert) den fehlenden Parametertyp aus dem Kontext.
- `x -> x + 4`
  Ein Parameter ohne explizite Typangabe. Runde Klammern können weglassen werden.
- `(int x, int y) -> x - y`
  Parameterliste mit zwei Parametern und expliziter Typangabe.
- `int x, int y -> x - y`
  Wenn man den Parametertyp angibt, müssen runde Klammern verwendet werden.
- `(x, y) -> x - y`
  Zwei Parameter ohne explizite Typangabe.

- `x, y -> x - y`
  Bei mehreren Parametern müssen runde Klammern verwendet werden.
- `(x, int y) -> x - y`
  Man darf Parameter mit und ohne Typangabe nicht mischen.
- `() -> 4711`
  Die Parameterliste darf leer sein.

8.4.3 Lambda-Body

Der Body (Rumpf) ist entweder ein einzelner Ausdruck oder eine Liste von Anweisungen in geschweiften Klammern.

- `() -> System.out.println("X")`
  Body bestehend aus einem einzigen Ausdruck.
- `(String[] args) -> (args != null) ? args.length : 0`
  Der ?–Operator ergibt auch einen einzigen Ausdruck.
- ```
  (String[] args) -> {
    if(args != null) {
      return args.length;
    } else {
      return 0;
    }
  }
  ```
  Body bestehend aus einer Anweisung. Hier müssen geschweifte Klammern verwendet werden.
- `(int x) -> x+1`
  Body bestehend aus einem einzigen Ausdruck.
- `(int x) -> return x+1`
  Mit `return` fängt eine Anweisung an und kein Ausdruck. Zusätzlich muss die `return`-Anweisung mit einem Semikolon enden, daher müssen geschweifte Klammen verwendet werden.
- `(int x) -> { return x+1;}`
  Richtig.
  ```

8.4.4 Lambda Streams

Mit Lambda Streams sind nicht IO-Streams gemeint, sondern Verarbeitung von Daten in entsprechenden Datentöpfen, wie Listen, Maps, etc.

Nachfolgendes Beispiel berechnet die Summe bei rabattierten Preisen über 50.

```
List<Double> preise = Arrays.asList(2.34, 20d, 30.33, 40d, 50d, 50.5, 6.23, 99.99);
double rabatt = 0.9;
double sum = 0;

for (Double preis : preise) {
   if (preis > 50) {
      sum += preis * rabatt;
   }
}
System.out.println(sum);
```

```
135.441
```

Die Nachteile dabei sind:

1. Initialisierung und explizites Aufsummieren der Variablen `sum` muss man selbst machen.
2. Es wird eine externe Iteration verwendet.
3. Die Verarbeitung dieses Iterators ist sequenziell, d. h. nur mit sehr großem Aufwand ist eine Parallelisierung möglich.
4. Es werden im Iterator folgende Operationen genutzt:
   - Filterung: `if (preis > 50)`
   - Berechnung Rabatt und Summenbildung

Mit Lambda-Streams sieht das Beispiel wie folgt aus:

```
List<Double> preise = Arrays.asList(2.34, 20d, 30.33, 40d, 50d, 50.5, 6.23, 99.99);
System.out.println(
  preise.stream()
  .filter(preis -> preis > 50)
  .mapToDouble(preis -> preis * 0.9)
  .sum()
);
```

```
135.441
```

Die Vorteile sind:

1. Es wird hier deklarative Programmierung angewendet:
   Es werden dabei generische Methoden und Parametrisierung durch Lambda-Ausdrucke verwendet (Entwurfsmuster „Code-as-Data“).
2. gute Lesbarkeit:
   Es werden so aussagekräftige Einteilungen gemacht, die in funktionale Aspekte aufgeteilt sind. Die Schritte sind dabei: Filterung, Transformation und Reduktion (Filter-Map-Reduce).
3. einfachere Syntax:
   Alle Bestandteile sind Ausdrücke und liefern ein Ergebnis zurück (dies wird auch „fluent API“ genannt).
4. mögliche Parallelisierung:
   Streams dieser Art können traversiert und somit parallel verarbeitet werden.
5. mögliche späte Evaluierung:
   Der Stream kann entscheiden, wann eine Evaluierung der Ausdrücke sinnvoll ist – diese kann insbesondere verzögert („lazy“) passieren.

Grundmuster der Streams

Das Grundmuster der Streams basiert auf dem Entwurfsmuster „Pipes & Filters“. Dabei handelt es sich um ein bewährtes Verfahren, welches bei Unix-Kommandos schon sehr lange angewendet wird:

```
ps -ef | grep bash | cut -c 49- | head
```

Die Ausgabe von `ps` wird mit der Pipe an `grep` zur Filterung weitergereicht. Die Ausgabe wird mit `cut` auf einen Teilbereich begrenzt (nur Spalten > 49). Danach gibt `head` die ersten Einträge aus.

Streams in Java 8 arbeiten letztlich nach demselben Prinzip und verketten Teiloperatoren zu einem Gesamtkontrollfluss auf einer Java-Collection.

Bei obigem Beispiel sind `filter` und `map` sogenannte intermediäre Operatoren, die „lazy“ ausgewertet werden. Dagegen ist `sum` ein terminaler Operator, der „eager“ ausgewertet wird.

lazy – eager

Funktionale Sprachen kann man auch nach ihrer Auswertungsstrategie unterscheiden: Bei strikter Auswertung (engl. eager bzw. strict evaluation) werden die Argumente von Funktionen zuerst ausgewertet. Dagegen werden bei der nicht-strikten Auswertung zunächst die Ausdrücke als Ganzes übergeben und dann ausgewertet.
Eine Variante der nicht-strikten Auswertung ist die Bedarfsauswertung (engl. lazy evaluation), bei der Ausdrücke erst ausgewertet werden, wenn deren Wert in einer Berechnung benötigt wird. Dadurch lassen sich z. B. unendlich große Datenstrukturen (die Liste aller natürlicher Zahlen, die Liste aller Primzahlen, etc.) definieren, und bestimmte Algorithmen vereinfachen sich.
Manche Berechnungen lassen sich mit strikter Auswertung, andere mit Bedarfsauswertung effizienter ausführen. Terminiert die strikte Auswertung eines Ausdruckes, so terminiert auch die nicht-strikte Auswertung. Hintergrund hiervon ist die Konfluenz-Eigenschaft des jeder funktionalen Sprache zugrundeliegenden λ-Kalküls, die aussagt, dass das Ergebnis der Berechnung nicht von der Reihenfolge der Auswertung abhängt.
Quelle: `https://de.wikipedia.org/wiki/Funktionale_Programmierung`

Die Streams-API bietet eine Fülle generischer Methoden an, die sich zur Verarbeitung von Collections nutzen lassen.[14]

Intermediäre Operation (Rückgabe ist immer ein `Stream`):

`distinct()`	Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
`filter()`	Returns a stream consisting of the elements of this stream that match the given predicate.
`flatMap()`	Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
`generate()`	Returns an infinite sequential unordered stream where each element is generated by the provided Supplier.
`iterate()`	Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
`limit()`	Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
`peek()`	Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
`skip()`	Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
`sorted()`	Returns a stream consisting of the elements of this stream, sorted according to natural order.

[14] siehe Java 8 API – `java.util.stream.Stream`

`map...`:

`map()` → `Stream`
Returns a stream consisting of the results of applying the given function to the elements of this stream.

`mapToDouble()` → `DoubleStream`
Returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.

`mapToInt()` → `IntStream`
Returns an IntStream consisting of the results of applying the given function to the elements of this stream.

`mapToLong()` → `LongStream`
Returns a LongStream consisting of the results of applying the given function to the elements of this stream.

Terminale Operation (Rückgabe ist funktionsabhängig):

`allMatch()` Returns whether all elements of this stream match the provided predicate.

`anyMatch(` Returns whether any elements of this stream match the provided predicate.

`count()` Returns the count of elements in this stream.

`findAny()` Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.

`findFirst()` Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.

`forEach()` Performs an action for each element of this stream.

`forEachOrdered()` Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.

`max()` Returns the maximum element of this stream according to the provided Comparator.

`min()` Returns the minimum element of this stream according to the provided Comparator.

`noneMatch()` Returns whether no elements of this stream match the provided predicate.

`reduce()` Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.

`DoubleStream`:

`average()` Returns an OptionalDouble describing the arithmetic mean of elements of this stream, or an empty optional if this stream is empty.

`count()` Returns the count of elements in this stream.

`max()` Returns an OptionalDouble describing the maximum element of this stream, or an empty OptionalDouble if this stream is empty.

`min()` Returns an OptionalDouble describing the minimum element of this stream, or an empty OptionalDouble if this stream is empty.

`sum()` Returns whether all elements of this stream match the provided predicate.

Beispiel:

```
List<Double> preise = Arrays.asList(2.34, 20d, 30.33, 40d, 50d, 50.5, 6.23, 99.99);
System.out.println("mit stream: " + preise.stream().count());
System.out.println("mit size(): " + preise.size());
System.out.println("sum()     : " + preise.stream().mapToDouble(preis -> preis).sum());
System.out.println("max()     : " + preise.stream().mapToDouble(preis -> preis).max().
getAsDouble());
System.out.println("min()     : " + preise.stream().mapToDouble(preis -> preis).min().
getAsDouble());
```

```
mit stream: 8
mit size(): 8
sum()     : 299.39
max()     : 99.99
min()     : 2.34
```

Parallele Verarbeitung

Verwendet man anstelle von `stream()` die Methode `parallelStream()`, so wird die Verarbeitung in parallele Aktionen zerlegt. Die Reihenfolge ist dabei nicht vorhersehbar.

```
List<Double> preise = Arrays.asList(2.34, 20d, 30.33, 40d, 50d, 50.5, 6.23, 99.99);
preise.parallelStream().forEach(preis -> System.out.print(preis + " "));
```

```
99.99 50.5 30.33 50.0 40.0 2.34 20.0 6.23
```

Performance

Bei Streams muss man immer die Performance im Hinterkopf haben. Nicht immer sind Streams schneller. Es kann gerade bei einfachen Operatoren sein, dass Streams deutlich langsamer sind als eine einfache Schleife. Je aufwendiger die Rechenoperation ist, desto performanter sind Streams.

Negativbeispiel: Hier wird der maximale Wert von einem `int`-Array ermittelt, welches vorher mit Zufallszahlen gefüllt worden ist.

```
int numberOfElements = 500_000;
float nstoms = 1_000_000f;
Random random = new Random();
int[] array = new int[numberOfElements];
for (int i = 0; i < numberOfElements; i++) {
   array[i] = random.nextInt();
}

long start = System.nanoTime();
int m = Integer.MIN_VALUE;
for (int i = 0; i < array.length; i++) {
   if (array[i] > m) {
      m = array[i];
   }
}
System.out.printf("for             %5.2fms\n",(System.nanoTime()-start)/ nstoms);

start = System.nanoTime();
m = Arrays.stream(array).max().getAsInt();
System.out.printf("stream          %5.2fms\n",(System.nanoTime()-start)/ nstoms);

start = System.nanoTime();
m = Arrays.stream(array).parallel().max().getAsInt();
System.out.printf("parallelstream  %5.2fms\n",(System.nanoTime()-start)/ nstoms);
System.out.printf("Kerne: %2d", Runtime.getRuntime().availableProcessors());
```

```
for              4,53ms
stream          81,09ms
parallelstream  19,39ms
Kerne:  4
```

8.5 Swing

8.5.1 Grundlagen der Swing-JTable

Für Tabellen gibt es in Swing eine eigene Klasse: `JTable`

Diese Klasse ist nach dem Model–View–Control–Konzept (MVC) aufgebaut.

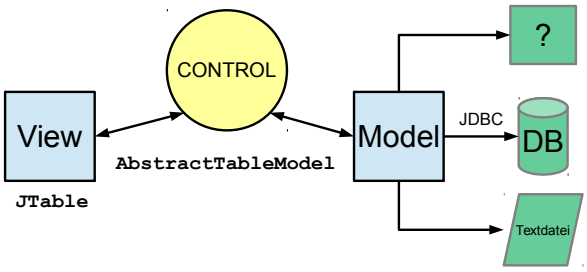
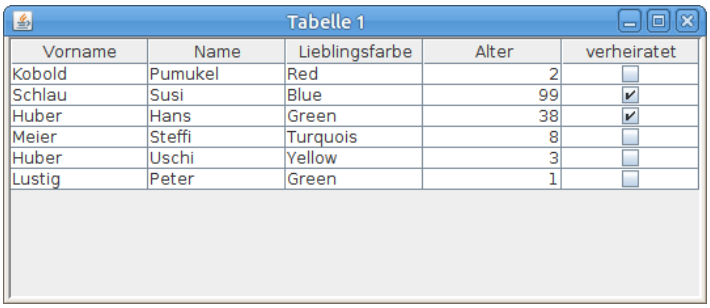


Abb. 8.10: Model–View–Control–Konzept

Einfache Tabelle



Vorname	Name	Lieblingsfarbe	Alter	verheiratet
Kobold	Pumukel	Red	2	☐
Schlau	Susi	Blue	99	☑
Huber	Hans	Green	38	☑
Meier	Steffi	Turquois	8	☐
Huber	Uschi	Yellow	3	☐
Lustig	Peter	Green	1	☐

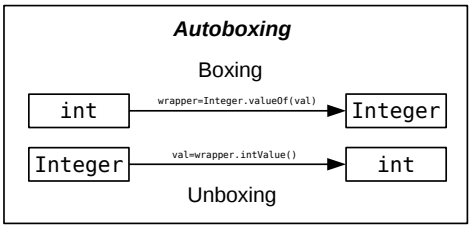
Abb. 8.11: einfache Tabelle

Bei dieser Übung wird eine Tabelle erstellt, in der es möglich ist, die Werte zu editieren. Die `JTable` übernimmt dabei mit Hilfe des Kontrollers alle Aufgaben.

Dabei kann eine `JTable` nur Objekte darstellen, einfache Datentypen müssen mit ihren „Wrapper"–Klassen[15] umgewandelt werden bzw. den Mechanismus „autoboxing" verwenden.

```
new Integer( 10 )
```

Wrapper-Klassen



Für jeden einfachen (primitiven) Datentyp gibt es eine entsprechende Wrapper-Klasse:
`int` ⟶ `Integer`, `double` ⟶ `Double`, ...

Wrapper-Klassen bieten statische Methoden zur Umwandlung eines primitiven Datentyps in einen String (dies wird Formatierung genannt) und vom String zurück in einen primitiven Datentyp (dies wird Parsen genannt).

Java wandelt automatisch bei Bedarf primitive Datentypen und Wrapper-Objekte ineinander um – dies wird auch Autoboxing genannt.

Speicher

Als Speicher wird für den Tabellen–Inhalt ein mehrdimensionales Array verwendet, für den Tabellen–Kopf ein eindimensionales Array.

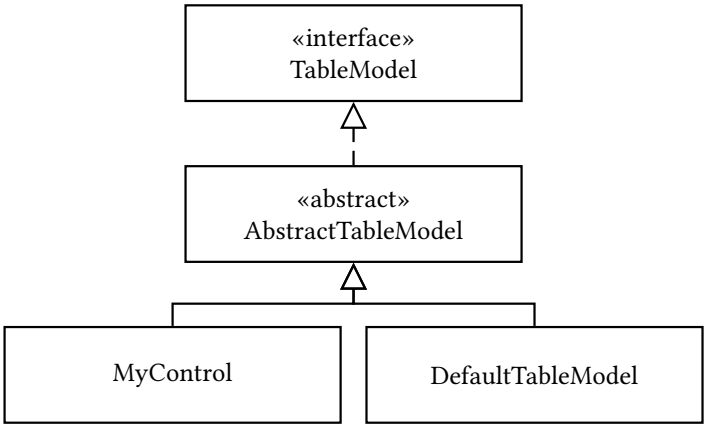
```
// Tabellen-Elemente
private final Object[][] data = { { "Kobold", "Pumukel", "Red", 2, false },
          { "Schlau", "Susi", "Blue", 99, true },
          { "Huber", "Hans", "Green", 38, true },
          { "Meier", "Steffi", "Turquois", 8, false },
          { "Huber", "Uschi", "Yellow", 3, false },
          { "Lustig", "Peter", "Green", 1, false } };
```

[15] siehe https://de.wikipedia.org/wiki/Adapter_%28Entwurfsmuster%29

```
// Tabellen-Überschrift
String[] names = { "Vorname", "Name", "Lieblingsfarbe", "Alter",
         "verheiratet" };
```

Kontroller

Als Control-Element wird die Klasse `MyControl`, die von `AbstractTableModel` (Interface `TableModel`) vererbt ist, verwendet.



Dabei ist es notwendig, die sieben Methoden zu überschreiben.

1. **getColumnCount()**
   liefert die Anzahl der Spalten zurück.
   Hierfür kann die Länge des Arrays für den Tabellen–Kopf verwendet werden.
2. **getRowCount()**
   liefert die Anzahl an Zeilen zurück.
   Hierfür kann die Länge des Arrays für die Tabellen–Daten verwendet werden.
3. **getValueAt(int row, int col)**
   liefert den Wert an Stelle row, col zurück.
4. **getColumnName(int col)**
   liefert die Spaltenüberschrift zurück.
5. **getColumnClass(int col)**
   liefert den Spaltentyp zurück.
   Dieser kann mit der Methode getClass() ermittelt werden.
6. **isCellEditable(int row, int col)**
   stellt fest, ob eine Zelle editierbar ist oder nicht.
7. **setValueAt(Object aValue, int row, int col)**
   setzt den Wert einer Zelle

```
// CONTROL
private class MyControl extends AbstractTableModel {

   // Diese Methoden müssen immer implementiert werden!!!

   @SuppressWarnings({ "unchecked", "rawtypes" })
   @Override
   public Class getColumnClass(final int col) {
      return getValueAt(0, col).getClass();
   }

   @Override
   public int getColumnCount() {
      return names.length;
   }

   @Override
   public String getColumnName(final int col) {
      return names[col];
   }

   @Override
```

```
    public int getRowCount() {
       return data.length;
    }

    @Override
    public Object getValueAt(final int row, final int col) {
       return data[row][col];
    }

    @Override
    public boolean isCellEditable(final int row, final int col) {
       return true;
    }

    @Override
    public void setValueAt(final Object aValue, final int row,
             final int col) {
       data[row][col] = aValue;
    }
 }
```

VIEW

Das View–Objekt ist hier die Tabelle. Als Parameter beim Erzeugen wird das Control–Objekt mit übergeben, so dass die Verknüpfung sofort hergestellt ist.

```
    // CONTROL
    final MyControl mc = new MyControl();

    // VIEW
    final JTable tableView = new JTable(mc);

    // Fenster ...
    final JScrollPane scrollpane = new JScrollPane(tableView);
    scrollpane.setBorder(new BevelBorder(BevelBorder.LOWERED));
    scrollpane.setPreferredSize(new Dimension(530, 200));
    getContentPane().add(scrollpane);
    pack();
    setVisible(true);
```

Tabellen–Editor

Es wäre nun schön, wenn man z. B. die Lieblingsfarbe nicht über ein Textfeld eingeben müsste, sondern diese über ein Auswahlfenster anwählen könnte. Dies ist mit der `JTable` möglich, da man für jede Spalte einen eigenen Editor definieren kann. Für die Auswahl der Farbe empfiehlt es sich, eine `JComboBox` zu wählen, die bei einem Doppelklick aufklappt und die gewünschten Farben anzeigt.

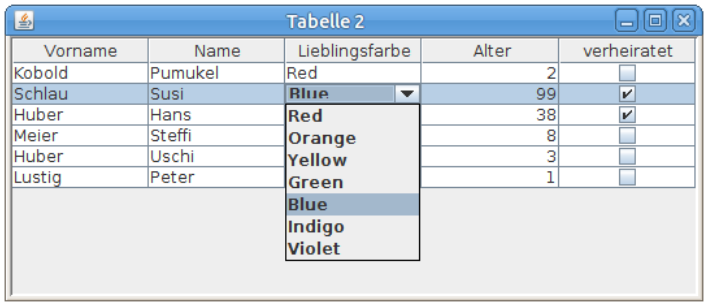


Abb. 8.12: Tabellen Editor

JComboBox

Wir definieren eine JComboBox und fügen alle Farben hinzu, die wir verwenden wollen.

```
// eine ComboBox soll als neuer Editor verwendet werden
final JComboBox<String> comboBox = new JComboBox<String>();
comboBox.addItem("Red");
comboBox.addItem("Orange");
comboBox.addItem("Yellow");
comboBox.addItem("Green");
comboBox.addItem("Blue");
comboBox.addItem("Indigo");
comboBox.addItem("Violet");
```

Spalte festlegen

Nun suchen wir die Spalten, für die der neue Editor gelten soll. Man kann hier die Spalte anhand der Spaltenüberschrift suchen lassen oder sie per Nummer festlegen.

```
// suche die Spalte, die als Überschrift "Lieblingsfarbe" hat
final TableColumn colorColumn = tableView.getColumn("Lieblingsfarbe");
```

Editor eintragen

Jetzt wird der Editor für die Spalte eingetragen.

```
// Editor eintragen
colorColumn.setCellEditor(new DefaultCellEditor(comboBox));
```

Die Klasse `DefaultCellEditor` kann standardmäßig drei verschiedene Klassen als Editor verwenden. Möchte man einen anderen Editor verwenden, so muss von `AbstractCellEditor` abgeleitet werden und die Implementierung selbst erstellt werden.

- `JCheckBox`
- `JComboBox`
- `JTextField`

Tabellen–Farbe–Tooltip–Breite

Nun wollen wir mit der Tabelle noch mehr machen:

- Farbe der Spalten ändern
- Farbe von Einträgen ändern, wenn sie bestimmten Bedingungen entsprechen
- Tooltip für Spalte und Spaltenkopf
- Breite von Spalten festlegen

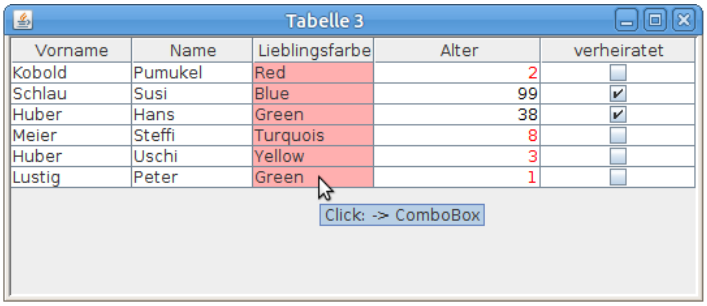


Abb. 8.13: Tabellen mit Farbe, Tooltip und Breite

Hintergrund und Tooltip

Es wird ein neuer `DefaultTableCellRenderer` angelegt, der von `JLabel` erbt. Dieser ist für das Malen der Zelle zuständig.

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ javax.swing.JComponent
              └ javax.swing.JLabel
                  └ javax.swing.table.DefaultTableCellRenderer
```

Abb. 8.14: Vererbungsbaum

Es wird sodann die Hintergrundfarbe auf "pink" gesetzt und der Tooltip festgelegt.

```
    // Einen "pink" Hintergrund und einen "tooltip" fuer die "Lieblingsfarbe"
    final DefaultTableCellRenderer colorColumnRenderer = new DefaultTableCellRenderer();

    colorColumnRenderer.setBackground(Color.pink);
    colorColumnRenderer.setToolTipText("Click: -> ComboBox");
    colorColumn.setCellRenderer(colorColumnRenderer);
```

Spalten–Tooltip

Für den Tooltip der Spalten wird zuerst ein Array mit den Texten definiert. Danach wird die Methode `createDefaultTableHeader()` der Klasse `JTable` überschrieben, der ein Objekt von `JTableHeader` mit der überschriebenen Methode `getToolTipText()` zurückliefert.

Anhand der Info, wo sich die Maus im Tabellenkopf befindet, wird der vorher definierte Text ausgewählt.

```
  // Tooltips
  String[] columnToolTips = { "Hier wird der Vorname angezeigt",
           "Hier wird der Name angezeigt", "Farbe", "Das Alter", null };
```

```
    final JTable tableView = new JTable(mc) {
       // Tooltip für den Spaltenkopf
       @Override
       protected JTableHeader createDefaultTableHeader() {
          return new JTableHeader(columnModel) {
             @Override
             public String getToolTipText(final MouseEvent e) {
                final Point p = e.getPoint();
                final int index = columnModel.getColumnIndexAtX(p.x);
                final int realIndex = columnModel.getColumn(index)
                         .getModelIndex();
                return columnToolTips[realIndex];
             }
          };
       }
    };
```

Spalten–Breite

Die Spaltenbreite lässt sich recht einfach über die Methode `setPreferredWidth()` verändern. Beachten Sie aber, wenn Sie eine Spaltenbreite verändern und die Gesamtbreite nicht verändern, so werden alle anderen Spalten bzw. nur die letzte Spalte auch angepasst.

```
// Breite der Spalte "verheiratet" setzen
final TableColumn marriedColumn = tableView.getColumn("verheiratet");
marriedColumn.setPreferredWidth(100);
```

Farbe der Spalteneinträge ändern

Um Änderungen der Spalteneinträge vornehmen zu können, benötigen wir wiederum einen `DefaultTableCellRenderer` und müssen dessen Methode `setValue()` überschreiben.

```
// Zeige die Eintraege in "Alter" in verschiedenen Farben
final TableColumn ageColumn = tableView.getColumn("Alter");
final DefaultTableCellRenderer ageColumnRenderer = new DefaultTableCellRenderer() {
   @Override
   public void setValue(final Object value) {
      final int cellValue = (value instanceof Number)
               ? ((Number) value).intValue() : 0;
      setForeground((cellValue > 18) ? Color.black : Color.red);
      setText((value == null) ? "" : value.toString());
   }
};
ageColumnRenderer.setHorizontalAlignment(JLabel.RIGHT);
ageColumn.setCellRenderer(ageColumnRenderer);
ageColumn.setPreferredWidth(110);
```

Tabellen–Besonderheiten

Jetzt wäre es natürlich noch schön, wenn bei der Spalte "Lieblingsfarbe" auch gleich diese als Hintergrund genommen würde.

Dazu müssen wir den `DefaultTableCellRenderer` `colorColumnRenderer` nur etwas abändern.

Bei der Methode `setValue()` wird die eingestellte Farbe abgefragt (hier nur "rot" und "gelb") und dann mit der Methode `setBackground()` die Hintergrundfarbe anhand der ausgewählten Farbe gesetzt und anschließend der Text gesetzt.

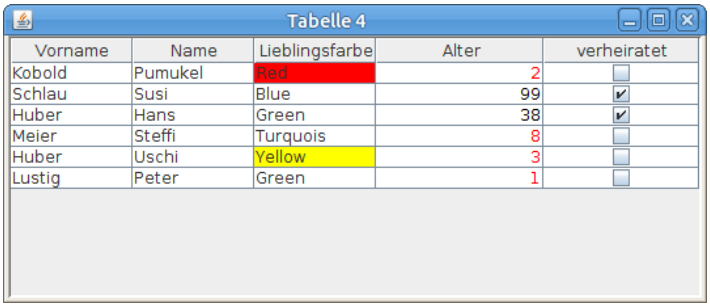


Abb. 8.15: Tabellen mit Besonderheiten

```
// Einen besonderen Hintergrund und einen "tooltip" für die "Lieblingsfarbe"
final DefaultTableCellRenderer colorColumnRenderer = new DefaultTableCellRenderer() ⤵
{
   @Override
   public void setValue(final Object value) {
      final String cellValue = (value instanceof String)
               ? ((String) value) : "";

      if (cellValue.equals("Red")) {
         setBackground(Color.red);
      } else if (cellValue.equals("Yellow")) {
         setBackground(Color.yellow);
      } else {
         setBackground(Color.white);
      }

      setText((value == null) ? "" : value.toString());
   }
};
colorColumnRenderer.setToolTipText("Click: -> ComboBox");
colorColumn.setCellRenderer(colorColumnRenderer);
```

Tabelleneditor für Farben

Für die Spalte „Lieblingsfarbe“ wäre es sinnvoll, wenn man hier als Spaltentyp nicht `String`, sondern `Color` verwendet. Dazu muss man den Typ im *Speicher* entsprechend ändern.

```
// Tabellen-Elemente
private final Object[][] data = {
         { "Kobold", "Pumukel", Color.RED, 2, false },
         { "Schlau", "Susi", Color.BLUE, 99, true },
         { "Huber", "Hans", Color.GREEN, 38, true },
         { "Meier", "Steffi", Color.MAGENTA, 8, false },
         { "Huber", "Uschi", Color.YELLOW, 3, false },
         { "Lustig", "Peter", Color.GREEN, 1, false } };
```

Der Renderer wird ebenfalls entsprechend angepasst.

```
   final DefaultTableCellRenderer colorColumnRenderer = new DefaultTableCellRenderer() ↲
   {
      @Override
      public void setValue(final Object value) {
         final Color cv = (value instanceof Color) ? ((Color) value)
                  : Color.WHITE;
         setBackground(cv);
         setText(String.format("#%02x%02x%02x", cv.getRed(), cv.getGreen(),
                  cv.getBlue()));
      }
   };
```

Tabelle 5

Vorname	Name	Lieblingsfarbe	Alter	verheiratet
Kobold	Pumukel	#ff0000	2	☐
Schlau	Susi	#0000ff	99	☑
Huber	Hans	#00ff00	38	☑
Meier	Steffi	#ff00ff	8	☐
Huber	Uschi	#ffff00	3	☐
Lustig	Peter	#00ff00	1	☐

Abb. 8.16: Verwendung von `Color`

Um die Farbe „schön“ auswählen zu können, leitet man eine Klasse von `AbstractCellEditor` ab und bindet zusätzlich das Interface `TableCellEditor` ein. Damit der Editor auf einen „klick“ reagiert, muss zusätzlich ein `JButton` mit einem `ActionListener` verwendet werden.

```
public class ColorCellEditor extends AbstractCellEditor
         implements TableCellEditor, ActionListener {

   /**
    * commad for the button.
    */
   private static final String EDIT = "edit";

   /**
    * The selected color.
    */
   private Color color;

   /**
    * The component for the table.
    */
   private final JButton button;

   /**
    * The color chooser.
    */
   private final JColorChooser colorChooser;

   /**
    * The dialog.
    */
   private final JDialog dialog;

   /**
    * Create a new object.
```

```
    */
   public ColorCellEditor() {
      button = new JButton();
      button.setActionCommand(EDIT);
      button.addActionListener(this);
      button.setBorderPainted(false);

      colorChooser = new JColorChooser();
      dialog = JColorChooser.createDialog(button, "Pick a Color", true,
               colorChooser, this, null);
   }

   @Override
   public void actionPerformed(final ActionEvent e) {
      if (EDIT.equals(e.getActionCommand())) {
         // EDIT
         colorChooser.setColor(color);
         dialog.setVisible(true);
         fireEditingStopped();
      } else {
         // OK
         color = colorChooser.getColor();
      }
   }

   @Override
   public Object getCellEditorValue() {
      return color;
   }

   @Override
   public Component getTableCellEditorComponent(final JTable table,
            final Object value, final boolean isSelected, final int row,
            final int column) {
      color = (Color) value;
      return button;
   }

}
```

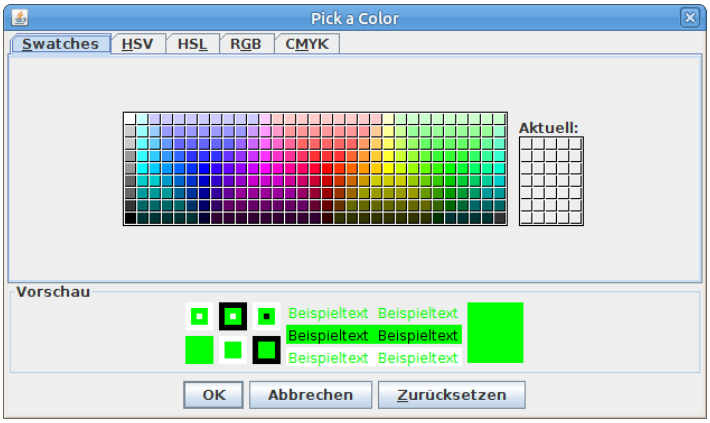


Abb. 8.17: Farbauswahl

8.5.2 Java-JTree

Es gibt viele Speicherformen und Darstellungen, bei denen man eine Baumstruktur zur Speicherung bzw. Anzeige verwendet. Darunter fallen z. B. Dateisysteme, XML-Dateien und vieles mehr.

Erster Baum

Der Baum wird mit der Klasse `JTree` dargestellt. Dabei wird jeder Eintrag in dem Baum mit der Klasse `DefaultMutableTreeNode` angezeigt. Hat der Eintrag Untereinträge, so werden diese nach einem Doppelklick (aufgeklappt) angezeigt.

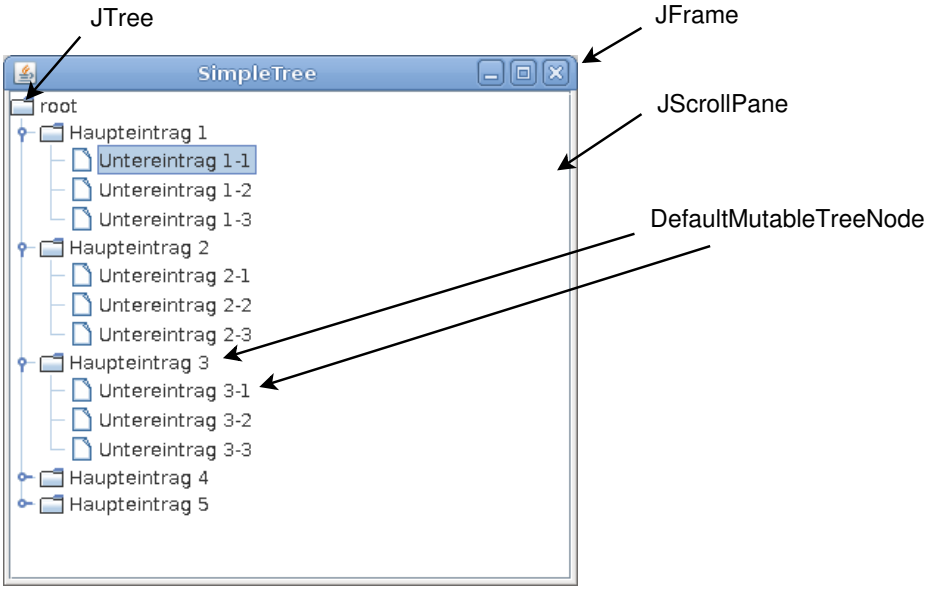


Abb. 8.18: Verwendete Klassen

FileBrowser

Eine häufig verwendete Baumstruktur ist das Dateisystem. Es hat einen Haupteintrag, viele Unterverzeichnisse und jedes Verzeichnis kann Dateien und weitere Unterverzeichnis enthalten. In den nächsten Schritten wird schrittweise ein Datei-Browser entwickelt.

Version 1

Für einen Dateieintrag wird die Klasse `FileTreeNode` erstellt, die von der Klasse `DefaultMutableTreeNode` erbt.

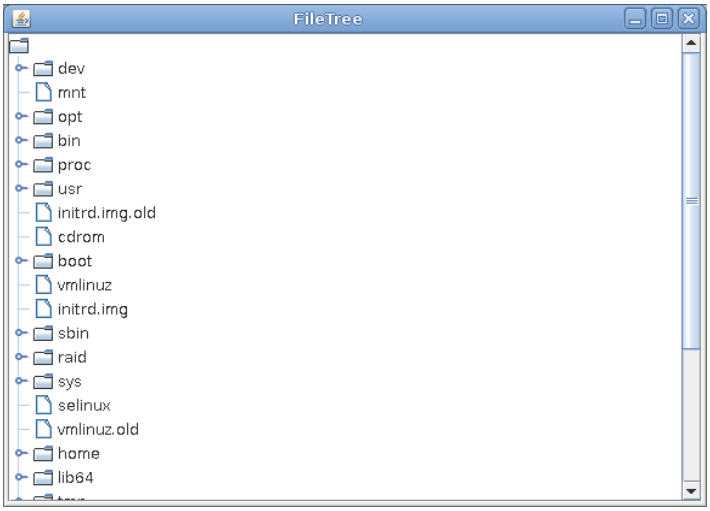


Abb. 8.19: Einfacher Datei-Browser

Bei dem herkömmlichen rekursiven Aufbau wird der komplette Verzeichnisbaum durchlaufen. Dies bedeutet, dass bei sehr vielen Verzeichnissen und Dateien das Einlesen sehr lange dauert, da jeder Verzeichniseintrag gelesen werden muss. Daher wird die Methode `toString()` überschrieben, welche vom `JTree` aufgerufen wird, um den Text für die Anzeige im Baum zu erhalten. Erst wenn dieser Text angefordert wird, wird der Inhalt des Verzeichnisses ausgelesen. Somit wird das Fenster sehr schnell aufgebaut und nur bei Bedarf, wenn der Anwender auf ein Unterverzeichnis klickt, die enthaltenen Daten gelesen.

FileExplorer

Bei vielen Baumdarstellungen ist es sinnvoll, zusätzliche Informationen in einer Tabelle anzuzeigen. Bei einem Dateibaum könnte dies z. B. der Dateityp, der Dateiname, die Größe in Bytes und vieles mehr sein.

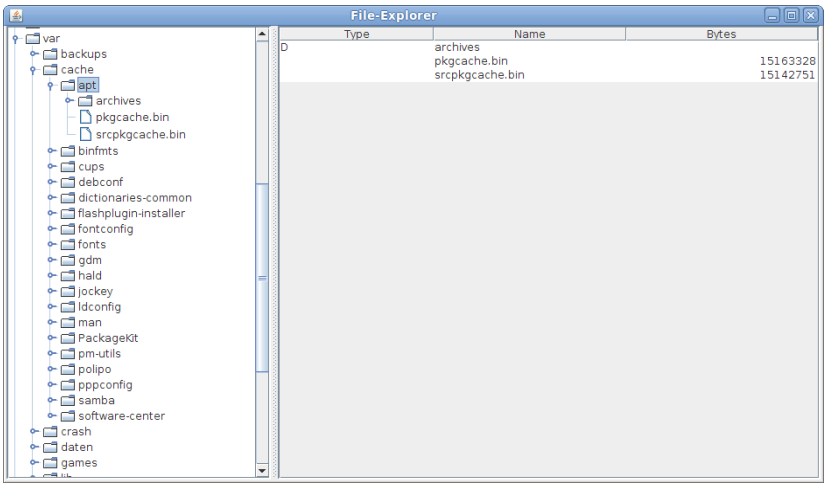


Abb. 8.20: Datei-Browser mit Baum und Tabelle

Für solch eine Kombination braucht man folgende Elemente:

- Eine Hauptklasse, die alles „zusammen hält“ (`FileExplorer`)
- Ein Modell für den Baum (`FileSystemModel`)
- Ein Modell für die Tabelle (`DirectoryModel`)
- Eine Tabelle (`FileTable`)
- Einen Baum (`FileTree`)
- Einen `TreeSelectionListener`, der Baum und Tabelle verbindet (`FileTreeListener`)
- Zwei `JScrollPane`s und eine `JSplitPane` für die Anzeige.
- Eine Klasse mit statischen Hilfemethoden.

JarViewer (einfach)

Ein JAR-Archiv hat den selben Aufbau wie ein ZIP-Archiv und wird nur durch eine zusätzliche Datei (`META-INF/MANIFEST.MF`) ergänzt.

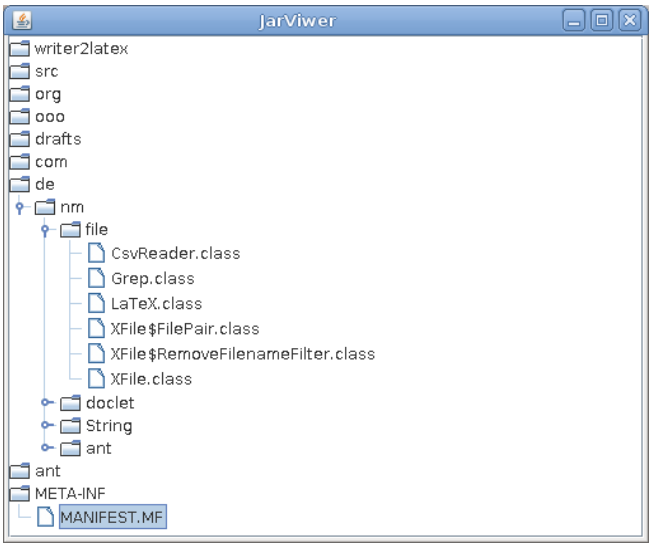


Abb. 8.21: Jar-Browser mit Baum

Der Eintrag des JAR-Archivs wird dabei mit der Klasse `JarEntryTreeNode` realisiert, welche von `DefaultMutableTreeNode` abgeleitet ist.

Da im JAR-Archiv jeder Eintrag flach gespeichert wird, d. h. der Eintrag hat alle Verzeichnisebenen und den Dateinamen in einem Eintrag, muss für den Baum jeweils die flache Ebene aufgespaltet und für jede Ebene ein eigener Eintrag im Baum erstellt werden. Dies wird in der Methode `readJar` realisiert.

FileBrowser mit integriertem Baum

Im vorherigen Beispiel wurde für die Ansicht eine geteilte Ansicht (`JSplitPane`) verwendet, in der auf der linken Seite der Baum und auf der rechten Seite die entsprechenden Tabelle angezeigt worden ist.

Eine andere Variante ist, den Baum direkt in einer Tabellenspalte anzuzeigen, was bewirkt, dass dadurch die Ansicht harmonischer wirkt.

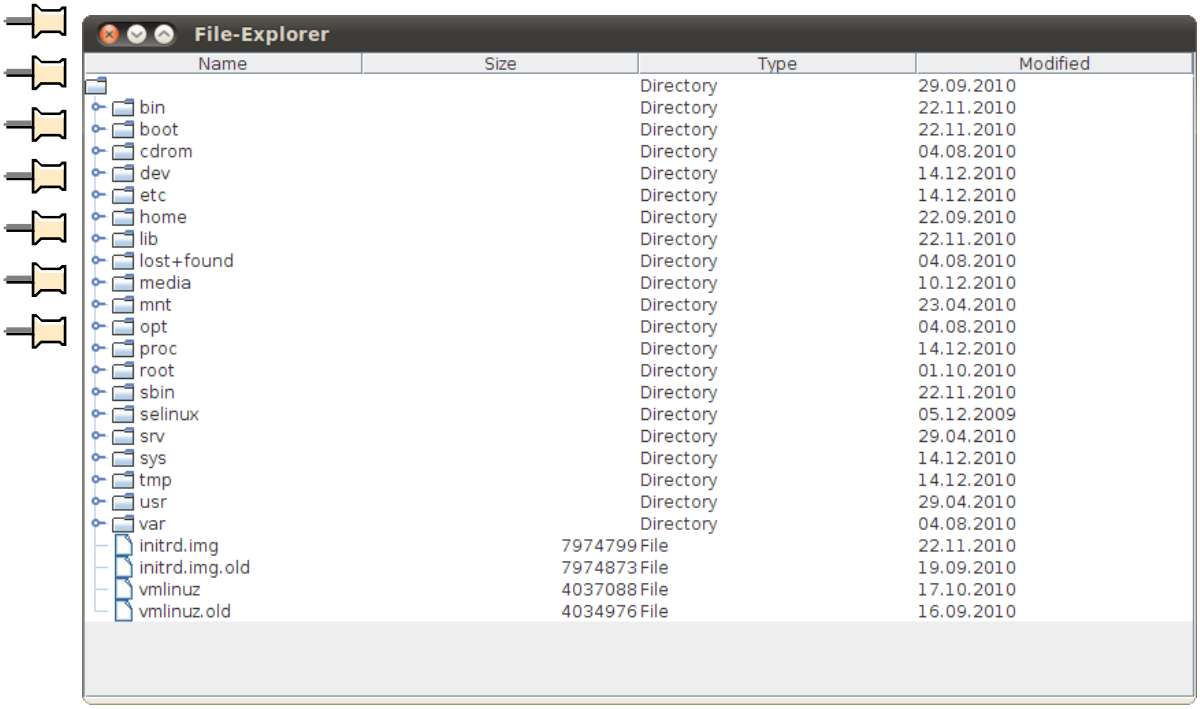


Abb. 8.22: File-Browser mit integriertem Baum in der Tabelle

Die Tabelle verwendet zum Anzeigen von Spalten die Klasse `DefaultTableCellRenderer` mit dem Interface `TableCellRenderer` als Renderer, welche von `JLabel` und `JComponent` abgeleitet ist. Prinzipiell kann jede Swing-Komponente für die Anzeige verwendet werden, also auch ein Baum mit der Klasse `JTree`, es muss nur zusätzlich das Interface `TableCellRenderer` implementiert werden.

Damit auch die Tabelle informiert wird, welchen Ast der Anwender „angeklickt" hat, muss hier zusätzlich ein Adapter (Entwurfsmuster) verwendet werden, der die Information des ausgewählten Eintrages im Baum an den Tabellenkontroller weitergibt, so dass die Daten angepasst werden können.

Abbildung 8.23 zeigt, welche Klassen für obige Ansicht verwendet werden und welche Klasse welche Klasse verwendet bzw. informiert.

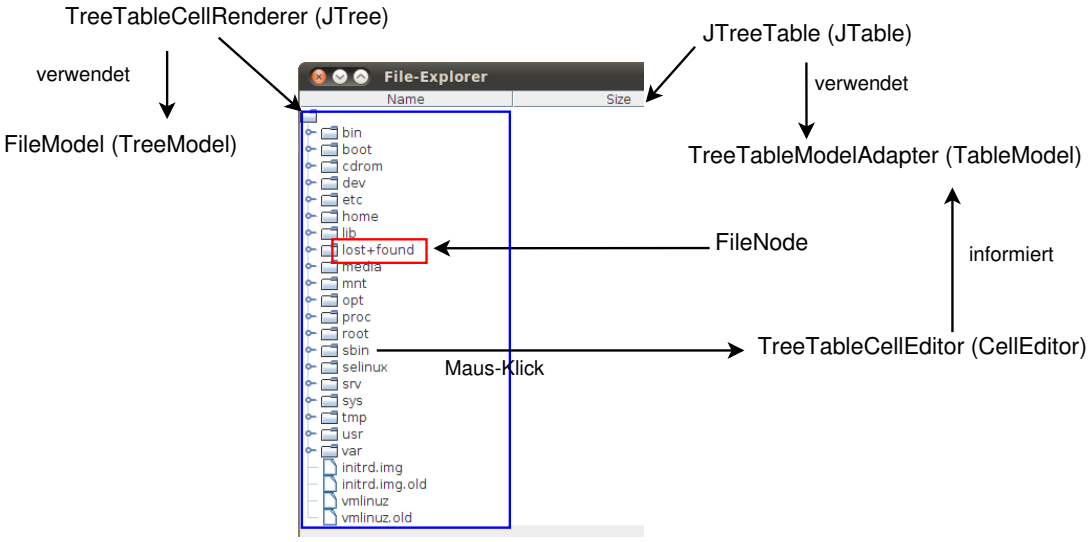


Abb. 8.23: File-Browser – Klassen

Klassenaufbau

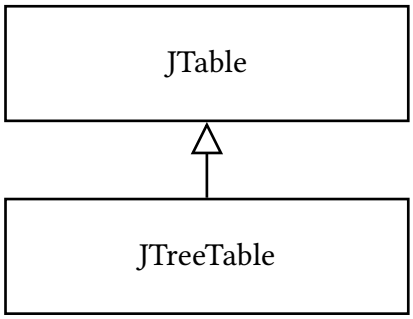


Abb. 8.24: Die Klasse `JTreeTable`

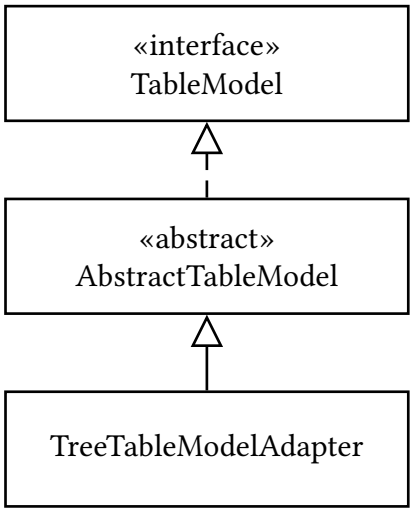


Abb. 8.25: Die Klasse `TreeTableModelAdapter`

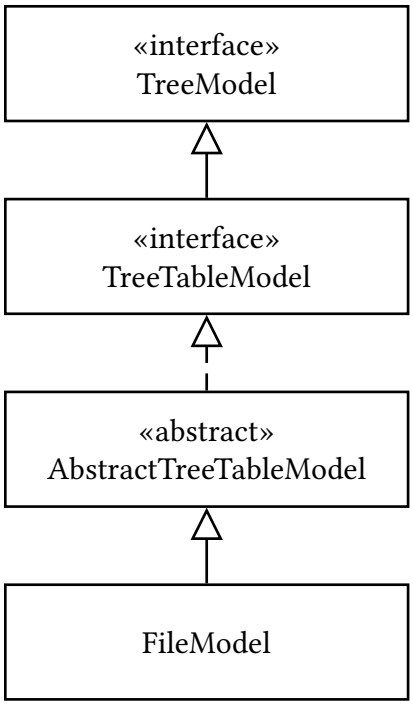


Abb. 8.26: Die Klasse `FileModel`

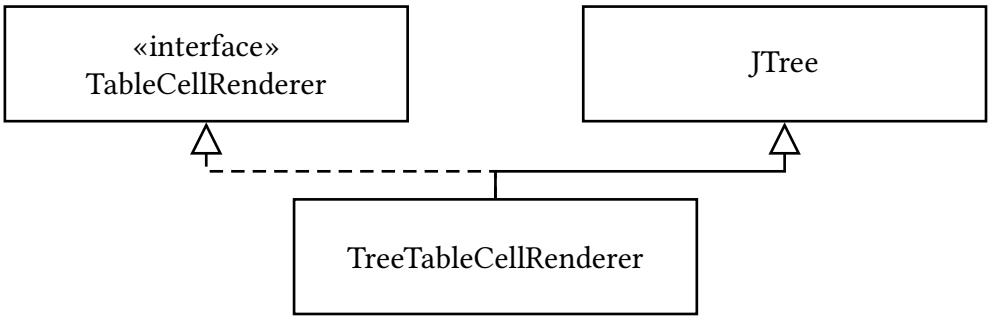


Abb. 8.27: Die Klasse `TreeTableCellRenderer`

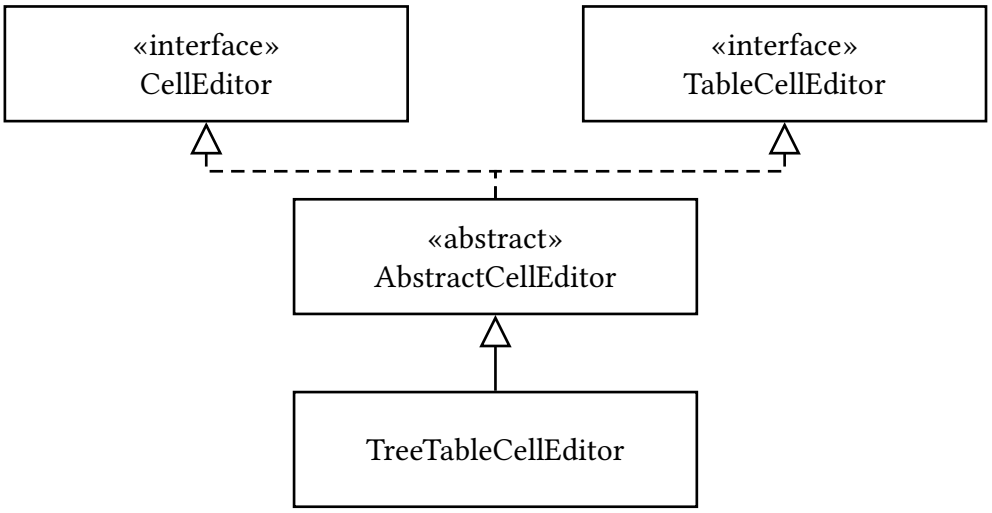


Abb. 8.28: Die Klasse `TreeTableCellEditor`

„Drag and Drop“ im Baum

Auch „Drag and Drop“ ist mit dem Baum möglich.

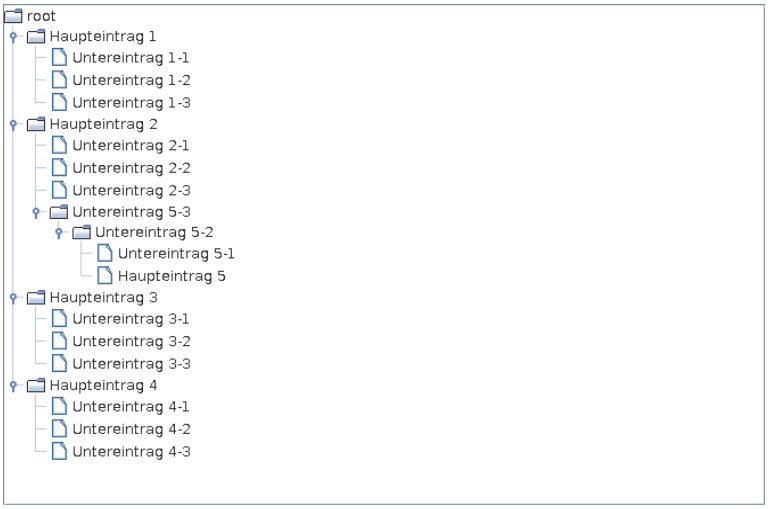


Abb. 8.29: „Drag and Drop“ im Baum

Dabei wird einfach ein Eintrag mit der Maus gepackt (linke Maustaste gedrückt halten) und an die Stelle geschoben, an der er zukünftig verweilen soll. Ob ein Eintrag verschoben werden kann oder nicht, kann dabei beliebig gesteuert werden.

Um den Baum „Drag and Drop“-fähig zu machen, sind folgende Schritte notwendig:

1. Mit der Methode `setDragEnabled()` wird die Funktion aktiviert.
2. Mit der Methode `setDropMode()` wird festgelegt, wie und was verschoben werden kann.
3. Das Ziel wird mit der Methode `setDropTarget()` festgelegt. Dabei wird die Klasse `DropTarget` verwendet, die als Parameter die Komponente erhält (hier der Baum) und eine Klasse, die das Interface `DropTargetListener` eingebunden hat.
4. Die Klasse `TreeDropTargetAdapter` erstellen, in der die Methode `drop()` überschrieben wird.

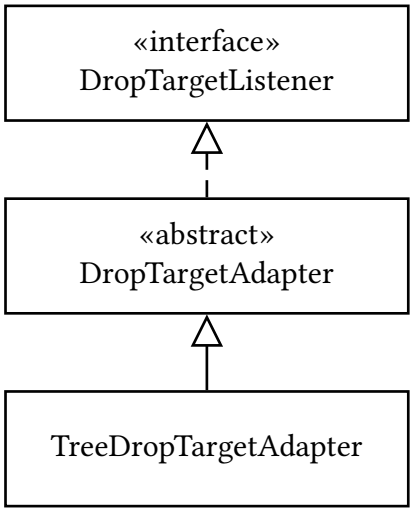


Abb. 8.30: Die Klasse `TreeDropTargetAdapter`

8.6 „Dies und Das“ ...

8.6.1 Klassen finden

Der JVM muss mitgeteilt werden, wo Klassen zu finden sind. Standardmäßig wird nur im aktuellen Verzeichnis und in den Systemverzeichnissen /-bibliotheken der Laufzeitumgebung nach Klassen gesucht.[16]

Aufrufparameter -classpath

Mit dem Parameter `-classpath` (kurs `-cp`) kann direkt beim Aufruf der JVM angegeben werden, welche zusätzlichen Suchorte verwendet werden sollen.

```
java -classpath classpath1:classpath2 MainClass
```

Der Klassenpfad enthält Wurzelverzeichnisse der Pakete und JAR-Dateien.

Beispiel:

```
java -cp "mylib.jar:bin/.:lib/*" de.nm.gui.MainClass
```

Hier werden die Klassen im JAR-Archiv `mylib.jar`, im Verzeichnis `bin` und alle JAR-Archive im Verzeichnis `lib` in den Klassenpfad aufgenommen.

Trenner

Unter Linux ist der Trenner ein Doppelpunkt ':', unter Windows ein Semikolon ';'.
Das Betriebssystem interpretiert den '*' selbst, daher muss die gesamte Pfadangabe in doppelten Anführungszeichen gesetzt werden!

Umgebungsvariable CLASSPATH

Alternativ kann auch die Umgebungsvariable CLASSPATH die Pfadangaben spezifizieren.

Linux

```
EXPORT CLASSPATH=classpath1:classpath2
java MainClass
```

Windows

```
SET CLASSPATH=classpath1;classpath2
java MainClass
```

Zu beachten ist dabei, dass `-cp`-Angaben die Einträge in CLASSPATH überschreiben.

[16] siehe https://docs.oracle.com/javase/8/docs/technotes/tools/unix/classpath.html

8.6.2 Zahlen formatieren

Mit Hilfe der Klasse `DecimalFormat` lassen sich Zahlen für die Ausgabe entsprechend formatieren.

```
DecimalFormat df = new DecimalFormat("#0.00");
System.out.println(df.format(23.1234567));
```

```
23,12
```

aus der API:

Symbol	*Location*	*Localized?*	*Meaning*
0	Number	Yes	Digit
#	Number	Yes	Digit, zero shows as absent
.	Number	Yes	Decimal separator or monetary decimal separator
-	Number	Yes	Minus sign
,	Number	Yes	Grouping separator
E	Number	Yes	Separates mantissa and exponent in scientific notation. Need not be quoted in prefix or suffix.
;	Subpattern boundary	Yes	Separates positive and negative subpatterns
%	Prefix or suffix	Yes	Multiply by 100 and show as percentage
\ u2030	Prefix or suffix	Yes	Multiply by 1000 and show as per mille value
\ u00A4	Prefix or suffix	No	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
’	Prefix or suffix	No	Used to quote special characters in a prefix or suffix, for example, `"'#'#"` formats 123 to `"#123"`. To create a single quote itself, use two in a row: `"# o''clock"`.

Ausgabe mit führenden Nullen:

```
DecimalFormat df = new DecimalFormat("00000000");
System.out.println(df.format(12345));
```

```
00012345
```

Eine weitere Möglichkeit bietet die Methode `format` der Klasse `String`.

```
System.out.println(String.format("%04x", 255));
```

```
00ff
```

```
System.out.println(String.format("%05d", 255));
```

```
00255
```

```
System.out.println(String.format("%.2f", 23.1234567));
```

```
23,12
```

8.6.3 SwingX

Mit der Erweiterung SwingX[17] (Lizenz: LGPL) stehen einem mehr Möglichkeiten zur Verfügung.

`JXDatePicker`	Standard date chooser component
`JXPanel`	Adds translucency to the standard JPanel
`JXGlassPanel`	Dismiss on click & drop shadow functionality
`JXTitledPanel`	Title bar added to the `JXPanel` ala JGoodies
`JXHyperLink`	Extends `JLabel`, adds action listener support
`JXRadioGroupy`	Extends `JLabel`, adds action listener support
`JXRadioGroup`	Simplifies dealing with `ButtonGroup` & `JRadioButton`
`JXScrollUp`	Extends `JXTitledPanel`, adds collapsing functionality
`JXStatusBar`	Enhanced status bar functionality
`JXMonthView`	Displays a month calendar
`JXImagePanel`	Displays an imageo
`JXErrorPane`	Enhanced standard error dialog
`JXEditorPane`	Enhancements to `JEditorPane`
`JXList`	Adds in place editing to `JList`
`JXTable`	Adds filtering/sorting/highlighting/column hiding to `JTable`
`JXTree`	Enhancements to `JTree`
`JXTreeTable`	Combination of `JTree` & `JTable`
`JXFindDialog`	Standard find dialog
`JXComboBox`	Enhanced `JComboBox` supporting embedded tables, etc
`JXButtonPanel`	Smart panel for displaying buttons in OS specific & local specific order

[17] siehe https://swingx.java.net/

8.7 Schülerbeispiele

Abb. 8.31: Schülerbeispiel: SJ 2015/16 BE

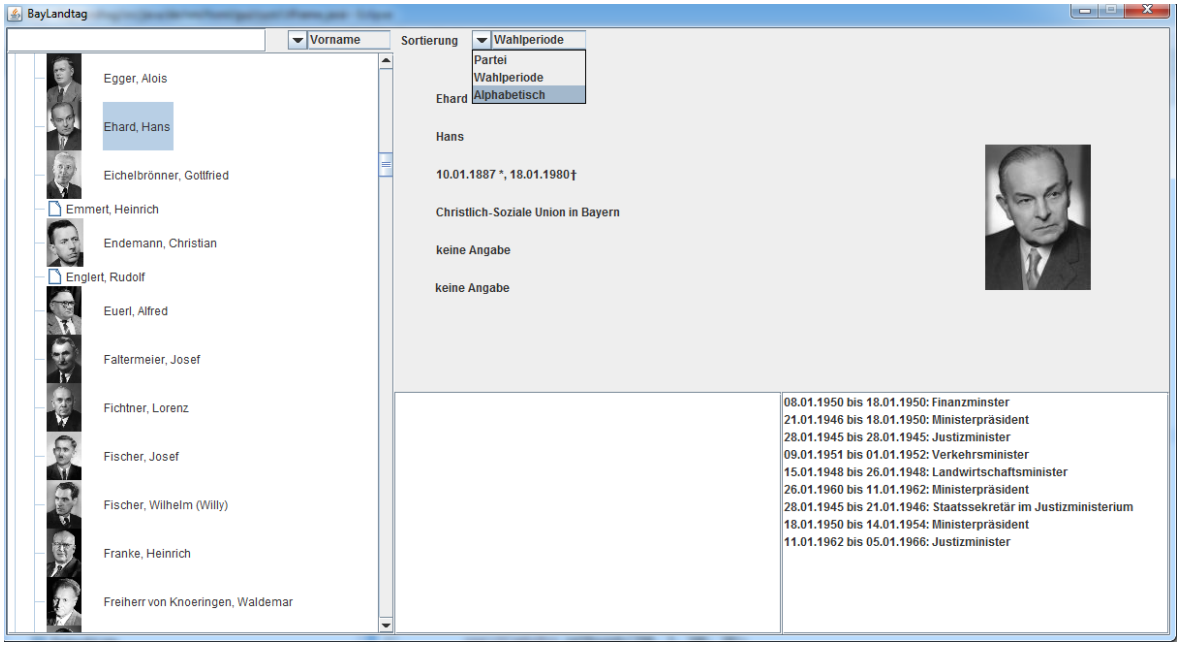


Abb. 8.32: Schülerbeispiel: SJ 2015/16 FM

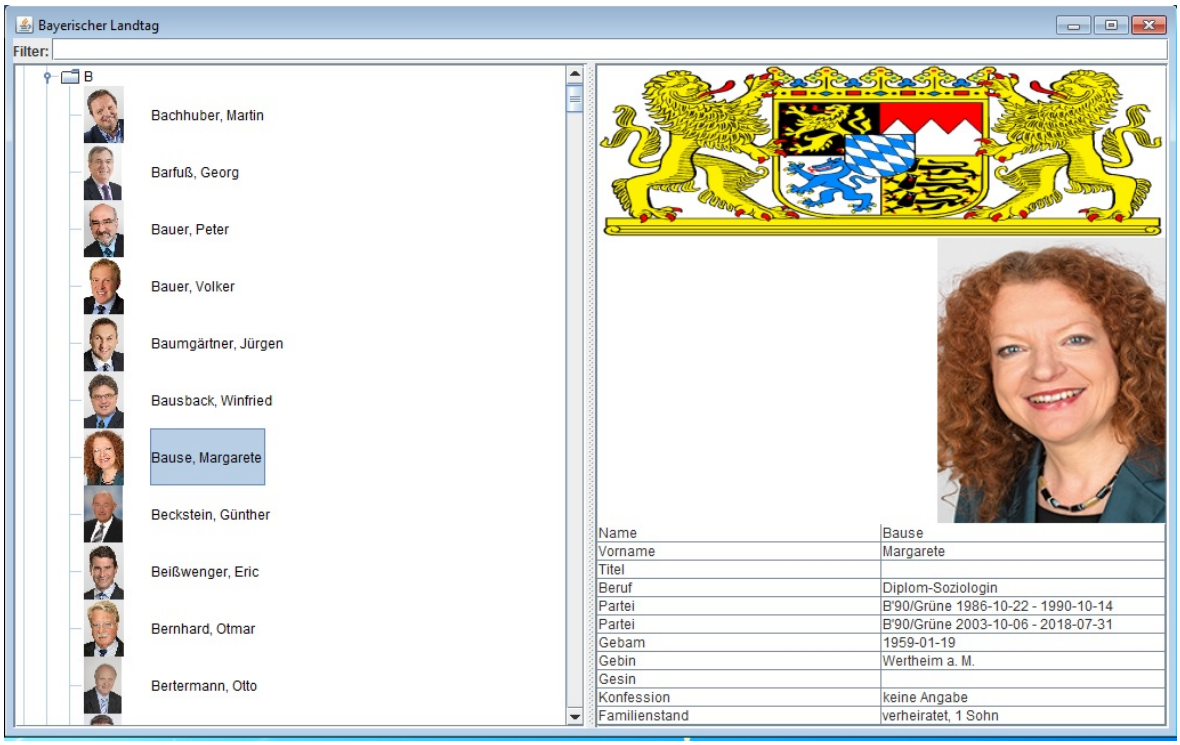


Abb. 8.33: Schülerbeispiel: SJ 2015/16 IS

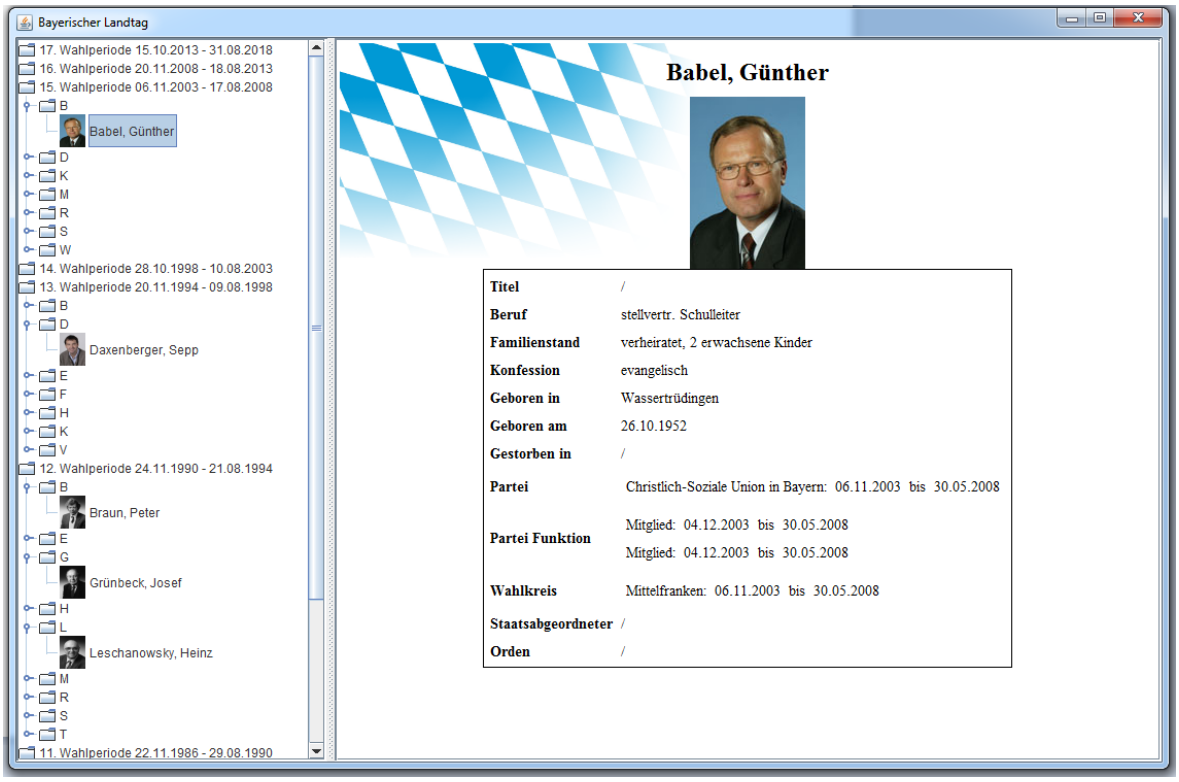


Abb. 8.34: Schülerbeispiel: SJ 2015/16 JG

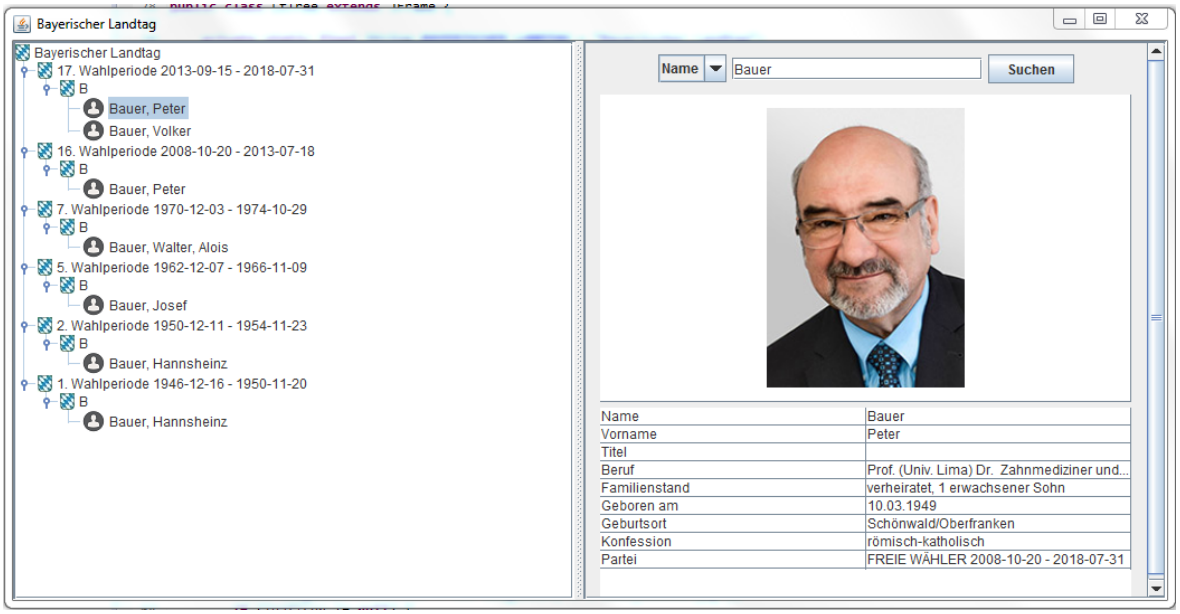


Abb. 8.35: Schülerbeispiel: SJ 2015/16 RH

Java

Symbole

@FunctionalInterface . . . 129
@Test . . . 38
@XmlAccessorType . . . 24
@XmlAttribute . . . 24
@XmlElement . . . 24
@XmlElementWrapper . . . 25
@XmlID . . . 24
@XmlIDREF . . . 24
@XmlJavaTypeAdapter . . . 27
@XmlRootElement . . . 24, 27
@XmlType . . . 24

A

AbstractButton (javax.swing)
  addActionListener() . . . 13
AbstractListModel (javax.swing)
  fireContentsChanged() . . . 16
  fireIntervalAdded() . . . 16
  fireIntervalRemoved() . . . 16
AbstractTableModel (javax.swing.table) . . . 52, 136
  fireTableDataChanged() . . . 54
ActionListener (java.awt.event) . . . 13, 128
  actionPerformed() . . . 13, 128
actionPerformed() (java.awt.event.ActionListener) . . . 13, 128
add() (java.util.Collection) . . . 63
add() (java.util.List) . . . 63
add() (java.util.Vector) . . . 15
add(...) (javax.swing.tree.DefaultMutableTreeNode) . . . 48
addActionListener() (javax.swing.AbstractButton) . . . 13
addAll() (java.util.Collection) . . . 63
addAll() (java.util.Collections) . . . 66
addElement() (javax.swing.DefaultListModel) . . . 15
addWindowListener() (java.awt.Window) . . . 13
allMatch() (java.util.stream.Stream) . . . 133
anyMatch((java.util.stream.Stream) . . . 133
append() (java.io.Writer) . . . 80
Appendable (java.io) . . . 79
ArrayDeque (java.util) . . . 70
ArrayList (java.util) . . . 54, 64
Arrays (java.util) . . . 62
  asList() . . . 66
asList() (java.util.Arrays) . . . 66
Assert (org.junit)
  assertEquals() . . . 37, 38
  assertNotNull() . . . 37, 38
  assertNull() . . . 37
  assertTrue() . . . 38
assertEquals() (org.junit.Assert) . . . 37, 38
assertNotNull() (org.junit.Assert) . . . 37, 38
assertNull() (org.junit.Assert) . . . 37
assertTrue() (org.junit.Assert) . . . 38
AutoCloseable (java.lang) . . . 79–81
average() (java.util.stream.DoubleStream) . . . 133

B

binarySearch() (java.util.Collections) . . . 66
BorderLayout (java.awt) . . . 13

BufferedReader (java.io) . . . 81
readLine() . . . 81
BufferedWriter (java.io) . . . 79, 80
close() . . . 80
flush() . . . 80
newLine() . . . 80
build() (org.jdom2.input.SAXBuilder) . . . 28

C

children() (javax.swing.tree.TreeNode) . . . 48
ClassNotFoundException (java.lang) . . . 87
clear() (java.util.Collection) . . . 63
clear() (java.util.Map) . . . 72
close() (java.io.BufferedWriter) . . . 80
close() (java.io.Reader) . . . 81
close() (java.io.Writer) . . . 80
close() (java.nio.file.FileSystem) . . . 76
Closeable (java.io) . . . 79–81
closeEntry() (java.util.zip.ZipEntry) . . . 93
Collator (java.text) . . . 67
setStrength() . . . 67
Collection (java.util)
add() . . . 63
addAll() . . . 63
clear() . . . 63
contains() . . . 63
isEmpty() . . . 63
iterator() . . . 63
parallelStream() . . . 63, 134
remove() . . . 63
size() . . . 63
stream() . . . 63, 134
toArray() . . . 63
Collections (java.util) . . . 66
addAll() . . . 66
binarySearch() . . . 66
fill() . . . 66
replaceAll() . . . 66
sort() . . . 66
Color (java.awt) . . . 12
Comparable (java.lang) . . . 68
Comparator (java.util) . . . 68
compare() . . . 67
compare() (java.util.Comparator) . . . 67
compareTo(...) (java.nio.file.Path) . . . 76
compile() (org.jdom2.xpath.XPathFactory) . . . 30
contains() (java.util.Collection) . . . 63
containsKey() (java.util.Map) . . . 72
containsValue() (java.util.Map) . . . 72
copy(...) (java.nio.file.Path) . . . 77
CopyOnWriteArrayList (java.util.concurrent) . . . 68
CopyOnWriteArraySet (java.util.concurrent) . . . 68
count() (java.util.stream.DoubleStream) . . . 133
count() (java.util.stream.Stream) . . . 133
createDirectories(...) (java.nio.file.Path) . . . 77
createDirectory(...) (java.nio.file.Path) . . . 77
createFile(...) (java.nio.file.Path) . . . 77
createLink(...) (java.nio.file.Path) . . . 77
createMarshaller() (javax.xml.bind.JAXBContext) . . . 27
createNewFile() (java.io.File) . . . 74
createSymbolicLink(...) (java.nio.file.Path) . . . 77
createTempDirectory(...) (java.nio.file.Path) . . . 77
createTempFile(...) (java.nio.file.Path) . . . 77
createUnmarshaller() (javax.xml.bind.JAXBContext) . . . 27

D

DataInputStream (java.io) . . . 85

readBoolean() . . . 86
readByte() . . . 86
readChar() . . . 86
readDouble() . . . 86
readFloat() . . . 86
readInt() . . . 86
readLong() . . . 86
readShort() . . . 86
readUnsignedByte() . . . 86
readUnsignedShort() . . . 86
readUTF() . . . 86
DataOutputStream (java.io) . . . 85
writeBoolean(boolean v) . . . 85
writeByte(int v) . . . 85
writeBytes(String s) . . . 85
writeChar(int v) . . . 85
writeChars(String s) . . . 85
writeDouble(double v) . . . 85
writeFloat(float v) . . . 85
writeInt(int v) . . . 85
writeLong(long v) . . . 85
writeShort(int v) . . . 85
writeUTF(String str) . . . 85
DecimalFormat (java.text) . . . 153
DefaultListModel (javax.swing) . . . 15
addElement() . . . 15
DefaultMutableTreeNode (javax.swing.tree) . . . 47, 48, 52–54, 144, 145
add(...) . . . 48
getDepth() . . . 48
getFirstChild() . . . 48
getFirstLeaf() . . . 48
getLastChild() . . . 48
getLastLeaf() . . . 48
getLeafCount() . . . 48
getLevel() . . . 48
DelayQueue (java.util.concurrent) . . . 70
delete() (java.io.File) . . . 74
delete(...) (java.nio.file.Path) . . . 77
deleteIfExists(...) (java.nio.file.Path) . . . 77
distinct() (java.util.stream.Stream) . . . 132
Document (org.jdom2) . . . 28
getRootElement() . . . 28
Document (org.w3c.dom) . . . 23
DoubleStream (java.util.stream)
average() . . . 133
count() . . . 133
max() . . . 133
min() . . . 133
sum() . . . 133
DropTarget (java.awt.dnd) . . . 151

E

Element (org.jdom2) . . . 28
getAttribute() . . . 28
getAttributes() . . . 28
getAttributeValue() . . . 28
getChild() . . . 28
getChildren() . . . 29
getChildText() . . . 29
getName() . . . 29
getText() . . . 29
endsWith(...) (java.nio.file.Path) . . . 76
EnumMap (java.util) . . . 73
EnumSet (java.util) . . . 68
equals() (java.util.Map) . . . 72
evaluate() (org.jdom2.xpath.XPathExpression) . . . 31

exists() (java.io.File) . . . 74
exists(...) (java.nio.file.Path) . . . 77
extends . . . 101

F

File (java.io) . . . 74, 76
  createNewFile() . . . 74
  delete() . . . 74
  exists() . . . 74
  getAbsolutePath() . . . 74
  getName() . . . 74
  getPath() . . . 74
  isDirectory() . . . 74
  isFile() . . . 74
  length() . . . 74
  list() . . . 74
  listFiles() . . . 74
  mkdir() . . . 74
FileInputStream (java.io) . . . 81
FileOutputStream (java.io) . . . 80
Files (java.nio.file) . . . 76
FileSystem (java.nio.file) . . . 76
  close() . . . 76
  getPath(...) . . . 76
  getSeparator() . . . 76
  isOpen() . . . 76
  isReadOnly() . . . 76
  newWatchService() . . . 76
  provider() . . . 76
FileSystems (java.nio.file) . . . 76
  getDefault() . . . 76
  getFileSystem(...) . . . 76
FileWriter (java.io) . . . 79
fill() (java.util.Collections) . . . 66
filter() (java.util.stream.Stream) . . . 132
final . . . 110
findAny() (java.util.stream.Stream) . . . 133
findFirst() (java.util.stream.Stream) . . . 133
fireContentsChanged() (javax.swing.AbstractListModel) . . . 16
fireIntervalAdded() (javax.swing.AbstractListModel) . . . 16
fireIntervalRemoved() (javax.swing.AbstractListModel) . . . 16
fireTableDataChanged() (javax.swing.table.AbstractTableModel) . . . 54
flatMap() (java.util.stream.Stream) . . . 132
flush() (java.io.BufferedWriter) . . . 80
flush() (java.io.Writer) . . . 80
Flushable (java.io) . . . 79, 80
forEach() (java.util.stream.Stream) . . . 133
forEachOrdered() (java.util.stream.Stream) . . . 133
format (java.lang.String) . . . 153

G

generate() (java.util.stream.Stream) . . . 132
get() (java.util.List) . . . 63
get() (java.util.Map) . . . 72
getAbsolutePath() (java.io.File) . . . 74
getAllowsChildren() (javax.swing.tree.TreeNode) . . . 48
getAttribute() (org.jdom2.Element) . . . 28
getAttribute(...) (java.nio.file.Path) . . . 77
getAttributes() (org.jdom2.Element) . . . 28
getAttributeValue() (org.jdom2.Element) . . . 28
getBytes() (java.lang.String) . . . 83
getChild() (org.jdom2.Element) . . . 28
getChildAt(...) (javax.swing.tree.TreeNode) . . . 48
getChildCount() (javax.swing.tree.TreeNode) . . . 48
getChildren() (org.jdom2.Element) . . . 29
getChildText() (org.jdom2.Element) . . . 29

getContentPane() (javax.swing.JFrame) . . . 13
getDefault() (java.nio.file.FileSystems) . . . 76
getDepth() (javax.swing.tree.DefaultMutableTreeNode) . . . 48
getElementAt() (javax.swing.ListModel) . . . 16
getFileName() (java.nio.file.Path) . . . 76
getFileSystem() (java.nio.file.Path) . . . 76
getFileSystem(...) (java.nio.file.FileSystems) . . . 76
getFirstChild() (javax.swing.tree.DefaultMutableTreeNode) . . . 48
getFirstLeaf() (javax.swing.tree.DefaultMutableTreeNode) . . . 48
getIndex(...) (javax.swing.tree.TreeNode) . . . 48
getInputStream(...) (java.util.zip.ZipFile) . . . 93
getLastChild() (javax.swing.tree.DefaultMutableTreeNode) . . . 48
getLastLeaf() (javax.swing.tree.DefaultMutableTreeNode) . . . 48
getLastModifiedTime(...) (java.nio.file.Path) . . . 77
getLeafCount() (javax.swing.tree.DefaultMutableTreeNode) . . . 48
getLevel() (javax.swing.tree.DefaultMutableTreeNode) . . . 48
getName() (java.io.File) . . . 74
getName() (org.jdom2.Element) . . . 29
getName(...) (java.nio.file.Path) . . . 76
getNameCount() (java.nio.file.Path) . . . 76
getOwner(...) (java.nio.file.Path) . . . 77
getParent() (java.nio.file.Path) . . . 76
getParent() (javax.swing.tree.TreeNode) . . . 48
getPath() (java.io.File) . . . 74
getPath(...) (java.nio.file.FileSystem) . . . 76
getPosixFilePermissions(...) (java.nio.file.Path) . . . 77
getRoot() (java.nio.file.Path) . . . 76
getRootElement() (org.jdom2.Document) . . . 28
getSeparator() (java.nio.file.FileSystem) . . . 76
getSize() (javax.swing.ListModel) . . . 16
getText() (org.jdom2.Element) . . . 29
GZIPInputStream (java.util.zip) . . . 92
GZIPOutputStream (java.util.zip) . . . 92

H

hashCode() (java.util.Map) . . . 72
HashMap (java.util) . . . 72
HashSet (java.util) . . . 68
Hashtable (java.util) . . . 71, 72
hasNext() (java.util.Iterator) . . . 66

I

indexOf() (java.util.List) . . . 64
InputStream (java.io) . . . 83
InputStreamReader (java.io) . . . 81
insert(...) (javax.swing.tree.MutableTreeNode) . . . 48
IOException (java.io) . . . 74
isAbsolute() (java.nio.file.Path) . . . 76
isDirectory() (java.io.File) . . . 74
isDirectory(...) (java.nio.file.Path) . . . 77
isEmpty() (java.util.Collection) . . . 63
isEmpty() (java.util.Map) . . . 72
isExecutable(...) (java.nio.file.Path) . . . 77
isFile() (java.io.File) . . . 74
isHidden(...) (java.nio.file.Path) . . . 77
isLeaf() (javax.swing.tree.TreeNode) . . . 48
isOpen() (java.nio.file.FileSystem) . . . 76
isReadable(...) (java.nio.file.Path) . . . 77
isReadOnly() (java.nio.file.FileSystem) . . . 76
isRegularFile(...) (java.nio.file.Path) . . . 77
isSameFile(...) (java.nio.file.Path) . . . 77
isSymbolicLink(...) (java.nio.file.Path) . . . 77
isWritable(...) (java.nio.file.Path) . . . 77
iterate() (java.util.stream.Stream) . . . 132
Iterator (java.util) . . . 31, 66
  hasNext() . . . 66

next() . . . 66
remove() . . . 66
iterator() (java.util.Collection) . . . 63

J

JarEntry (java.util.jar) . . . 94
JarFile (java.util.jar) . . . 94
JarOutputStream (java.util.jar) . . . 94
JAXB_FORMATTED_OUTPUT (javax.xml.bind.Marshaller) . . . 27
JAXBContext (javax.xml.bind) . . . 27
createMarshaller() . . . 27
createUnmarshaller() . . . 27
newInstance() . . . 27
JButton (javax.swing) . . . 13
JEditorPane (javax.swing) . . . 56
setText() . . . 56
JFrame (javax.swing) . . . 12, 49
getContentPane() . . . 13
setDefaultCloseOperation() . . . 49
JLabel (javax.swin) . . . 13
JLabel (javax.swing) . . . 13
JList (javax.swing) . . . 15
JScrollPane (javax.swing) . . . 16, 49
JSplitPane (javax.swing) . . . 52
JTable (javax.swing) . . . 52, 135
setTableHeader . . . 52
JTableHeader (javax.swing.table) . . . 139
JTree (javax.swing) . . . 47, 49, 144, 145
setDragEnabled() . . . 151
setDropMode() . . . 151
setDropTarget() . . . 151

K

keySet() (java.util.Map) . . . 72

L

length() (java.io.File) . . . 74
limit() (java.util.stream.Stream) . . . 132
lines(...) (java.nio.file.Path) . . . 77
LinkedHashMap (java.util) . . . 72
LinkedHashSet (java.util) . . . 68
LinkedList (java.util) . . . 65, 70
List (java.util) . . . 25, 64
add() . . . 63
get() . . . 63
indexOf() . . . 64
remove() . . . 64
set() . . . 64
list() (java.io.File) . . . 74
list(...) (java.nio.file.Path) . . . 77
listFiles() (java.io.File) . . . 74
Listmodel (javax.swing) . . . 15
ListModel (javax.swing)
getElementAt() . . . 16
getSize() . . . 16
loadFromXML() (java.util.Properties) . . . 71
Locale.GERMAN (java.util) . . . 67

M

Map (java.util) . . . 26
clear() . . . 72
containsKey() . . . 72
containsValue() . . . 72
equals() . . . 72
get() . . . 72
hashCode() . . . 72
isEmpty() . . . 72

keySet() . . . 72
put() . . . 72
remove() . . . 72
replace() . . . 72
size() . . . 72
values() . . . 72
map() (java.util.stream.Stream) . . . 133
mapToDouble() (java.util.stream.Stream) . . . 133
mapToInt() (java.util.stream.Stream) . . . 133
mapToLong() (java.util.stream.Stream) . . . 133
marshal() (javax.xml.bind.Marshaller) . . . 27
Marshaller (javax.xml.bind) . . . 27
JAXB_FORMATTED_OUTPUT . . . 27
marshal() . . . 27
setProperty() . . . 27
unmarshal() . . . 27
max() (java.util.stream.DoubleStream) . . . 133
max() (java.util.stream.Stream) . . . 133
min() (java.util.stream.DoubleStream) . . . 133
min() (java.util.stream.Stream) . . . 133
mkdir() (java.io.File) . . . 74
move(...) (java.nio.file.Path) . . . 77
MutableTreeNode (javax.swing.tree) . . . 47, 48
insert(...) . . . 48
remove(...) . . . 48
removeFromParent() . . . 48
setParent(...) . . . 48
setUserObject(...) . . . 48

N

NavigableMap (java.util) . . . 73
NavigableSet (java.util) . . . 68
newBufferedReader(...) (java.nio.file.Path) . . . 77
newBufferedWriter(...) (java.nio.file.Path) . . . 77
newInputStream(...) (java.nio.file.Path) . . . 77
newInstance() (javax.xml.bind.JAXBContext) . . . 27
newLine() (java.io.BufferedWriter) . . . 80
newOutputStream(...) (java.nio.file.Path) . . . 77
newWatchService() (java.nio.file.FileSystem) . . . 76
next() (java.util.Iterator) . . . 66
noneMatch() (java.util.stream.Stream) . . . 133
notExists(...) (java.nio.file.Path) . . . 77

O

ObjectInputStream (java.io) . . . 86
readBoolean() . . . 87
readByte() . . . 87
readChar() . . . 87
readDouble() . . . 87
readFloat() . . . 87
readInt() . . . 88
readLong() . . . 88
readObject() . . . 88
readShort() . . . 88
readUnsignedByte() . . . 88
readUnsignedShort() . . . 88
readUTF() . . . 88
ObjectOutputStream (java.io) . . . 86, 88
writeBoolean(boolean v) . . . 86
writeByte(int v) . . . 86
writeBytes(String s) . . . 86
writeChar(int v) . . . 86
writeChars(String s) . . . 86
writeDouble(double v) . . . 86
writeFloat(float v) . . . 86
writeInt(int v) . . . 86

writeLong(long v) . . . 87
writeObject(Object obj) . . . 87
writeShort(int v) . . . 87
writeUTF(String str) . . . 87
OutputStream (java.io) . . . 79, 83
OutputStreamWriter (java.io) . . . 79, 80

P

parallelStream() (java.util.Collection) . . . 63, 134
Path (java.nio.file) . . . 76
compareTo(...) . . . 76
copy(...) . . . 77
createDirectories(...) . . . 77
createDirectory(...) . . . 77
createFile(...) . . . 77
createLink(...) . . . 77
createSymbolicLink(...) . . . 77
createTempDirectory(...) . . . 77
createTempFile(...) . . . 77
delete(...) . . . 77
deleteIfExists(...) . . . 77
endsWith(...) . . . 76
exists(...) . . . 77
getAttribute(...) . . . 77
getFileName() . . . 76
getFileSystem() . . . 76
getLastModifiedTime(...) . . . 77
getName(...) . . . 76
getNameCount() . . . 76
getOwner(...) . . . 77
getParent() . . . 76
getPosixFilePermissions(...) . . . 77
getRoot() . . . 76
isAbsolute() . . . 76
isDirectory(...) . . . 77
isExecutable(...) . . . 77
isHidden(...) . . . 77
isReadable(...) . . . 77
isRegularFile(...) . . . 77
isSameFile(...) . . . 77
isSymbolicLink(...) . . . 77
isWritable(...) . . . 77
lines(...) . . . 77
list(...) . . . 77
move(...) . . . 77
newBufferedReader(...) . . . 77
newBufferedWriter(...) . . . 77
newInputStream(...) . . . 77
newOutputStream(...) . . . 77
notExists(...) . . . 77
readAllBytes(...) . . . 77
readAllLines(...) . . . 77
setAttribute(...) . . . 77
setLastModifiedTime(...) . . . 77
setOwner(...) . . . 77
setPosixFilePermissions(...) . . . 77
size(...) . . . 77
startsWith(...) . . . 76
subpath(...) . . . 76
toAbsolutePath() . . . 76
toFile() . . . 76
toRealPath(...) . . . 76
toUri() . . . 76
write(...) . . . 77
peek() (java.util.stream.Stream) . . . 132
PriorityQueue (java.util) . . . 70

private . . . 103
Properties (java.util) . . . 71
  loadFromXML() . . . 71
  store() . . . 71
  storeToXML() . . . 71
protected . . . 103
provider() (java.nio.file.FileSystem) . . . 76
public . . . 103
put() (java.util.Map) . . . 72

R

read() (java.io.Reader) . . . 81
Readable (java.io) . . . 81
readAllBytes(...) (java.nio.file.Path) . . . 77
readAllLines(...) (java.nio.file.Path) . . . 77
readBoolean() (java.io.DataInputStream) . . . 86
readBoolean() (java.io.ObjectInputStream) . . . 87
readByte() (java.io.DataInputStream) . . . 86
readByte() (java.io.ObjectInputStream) . . . 87
readChar() (java.io.DataInputStream) . . . 86
readChar() (java.io.ObjectInputStream) . . . 87
readDouble() (java.io.DataInputStream) . . . 86
readDouble() (java.io.ObjectInputStream) . . . 87
Reader (java.io) . . . 79, 81
  close() . . . 81
  read() . . . 81
  reset() . . . 81
  skip() . . . 81
readFloat() (java.io.DataInputStream) . . . 86
readFloat() (java.io.ObjectInputStream) . . . 87
readInt() (java.io.DataInputStream) . . . 86
readInt() (java.io.ObjectInputStream) . . . 88
readLine() (java.io.BufferedReader) . . . 81
readLong() (java.io.DataInputStream) . . . 86
readLong() (java.io.ObjectInputStream) . . . 88
readObject() (java.beans.XMLDecoder) . . . 88
readObject() (java.io.ObjectInputStream) . . . 88
readShort() (java.io.DataInputStream) . . . 86
readShort() (java.io.ObjectInputStream) . . . 88
readUnsignedByte() (java.io.DataInputStream) . . . 86
readUnsignedByte() (java.io.ObjectInputStream) . . . 88
readUnsignedShort() (java.io.DataInputStream) . . . 86
readUnsignedShort() (java.io.ObjectInputStream) . . . 88
readUTF() (java.io.DataInputStream) . . . 86
readUTF() (java.io.ObjectInputStream) . . . 88
reduce() (java.util.stream.Stream) . . . 133
remove() (java.util.Collection) . . . 63
remove() (java.util.Iterator) . . . 66
remove() (java.util.List) . . . 64
remove() (java.util.Map) . . . 72
remove(...) (javax.swing.tree.MutableTreeNode) . . . 48
removeFromParent() (javax.swing.tree.MutableTreeNode) . . . 48
replace() (java.util.Map) . . . 72
replaceAll() (java.util.Collections) . . . 66
reset() (java.io.Reader) . . . 81

S

SAXBuilder (org.jdom2.input) . . . 28
  build() . . . 28
Serializable (java.io) . . . 89
Set (java.util) . . . 68
set() (java.util.List) . . . 64
setAttribute(...) (java.nio.file.Path) . . . 77
setDefaultCloseOperation() (javax.swing.JFrame) . . . 49
setDragEnabled() (javax.swing.JTree) . . . 151
setDropMode() (javax.swing.JTree) . . . 151

setDropTarget() (javax.swing.JTree) . . . 151
setLastModifiedTime(...) (java.nio.file.Path) . . . 77
setOwner(...) (java.nio.file.Path) . . . 77
setParent(...) (javax.swing.tree.MutableTreeNode) . . . 48
setPosixFilePermissions(...) (java.nio.file.Path) . . . 77
setProperty() (javax.xml.bind.Marshaller) . . . 27
setStrength() (java.text.Collator) . . . 67
setTableHeader (javax.swing.JTable) . . . 52
setText() (javax.swing.JEditorPane) . . . 56
setUserObject(...) (javax.swing.tree.MutableTreeNode) . . . 48
size() (java.util.Collection) . . . 63
size() (java.util.Map) . . . 72
size(...) (java.nio.file.Path) . . . 77
skip() (java.io.Reader) . . . 81
skip() (java.util.stream.Stream) . . . 132
sort() (java.util.Collections) . . . 66
sorted() (java.util.stream.Stream) . . . 132
startsWith(...) (java.nio.file.Path) . . . 76
static . . . 102
store() (java.util.Properties) . . . 71
storeToXML() (java.util.Properties) . . . 71
Stream (java.util.stream)
  allMatch() . . . 133
  anyMatch(. . . 133
  count() . . . 133
  distinct() . . . 132
  filter() . . . 132
  findAny() . . . 133
  findFirst() . . . 133
  flatMap() . . . 132
  forEach() . . . 133
  forEachOrdered() . . . 133
  generate() . . . 132
  iterate() . . . 132
  limit() . . . 132
  map() . . . 133
  mapToDouble() . . . 133
  mapToInt() . . . 133
  mapToLong() . . . 133
  max() . . . 133
  min() . . . 133
  noneMatch() . . . 133
  peek() . . . 132
  reduce() . . . 133
  skip() . . . 132
  sorted() . . . 132
stream() (java.util.Collection) . . . 63, 134
String (java.lang)
  format . . . 153
  getBytes() . . . 83
StringReader (java.io) . . . 81
StringWriter (java.io) . . . 79
subpath(...) (java.nio.file.Path) . . . 76
sum() (java.util.stream.DoubleStream) . . . 133
super . . . 105–107
super() . . . 105, 106

T

TableModel (javax.swing.table) . . . 136
TarArchiveEntry (org.apache.commons.compress.archivers.tar) . . . 96
TarArchiveInputStream (org.apache.commons.compress.archivers.tar) . . . 96
TarArchiveOutputStream (org.apache.commons.compress.archivers.tar) . . . 96
this . . . 106
throws . . . 108
toAbsolutePath() (java.nio.file.Path) . . . 76
toArray() (java.util.Collection) . . . 63

toFile() (java.nio.file.Path) . . . 76
toRealPath(...) (java.nio.file.Path) . . . 76
toUri() (java.nio.file.Path) . . . 76
TreeMap (java.util) . . . 73
TreeNode (javax.swing.tree) . . . 47, 48
  children() . . . 48
  getAllowsChildren() . . . 48
  getChildAt(...) . . . 48
  getChildCount() . . . 48
  getIndex(...) . . . 48
  getParent() . . . 48
  isLeaf() . . . 48
TreeSelectionListener (javax.swing.event) . . . 52, 53, 56
TreeSet (java.util) . . . 50, 68

U

unmarshal() (javax.xml.bind.Marshaller) . . . 27

V

values() (java.util.Map) . . . 72
Vector (java.util) . . . 15, 65
  add() . . . 15

W

WeakHashMap (java.util) . . . 72
Window (java.awt)
  addWindowListener() . . . 13
WindowAdapter (java.awt.event) . . . 13
  windowClosing() . . . 13
windowClosing() (java.awt.event.WindowAdapter) . . . 13
write() (java.io.Writer) . . . 80
write(...) (java.nio.file.Path) . . . 77
writeBoolean(boolean v) (java.io.DataOutputStream) . . . 85
writeBoolean(boolean v) (java.io.ObjectOutputStream) . . . 86
writeByte(int v) (java.io.DataOutputStream) . . . 85
writeByte(int v) (java.io.ObjectOutputStream) . . . 86
writeBytes(String s) (java.io.DataOutputStream) . . . 85
writeBytes(String s) (java.io.ObjectOutputStream) . . . 86
writeChar(int v) (java.io.DataOutputStream) . . . 85
writeChar(int v) (java.io.ObjectOutputStream) . . . 86
writeChars(String s) (java.io.DataOutputStream) . . . 85
writeChars(String s) (java.io.ObjectOutputStream) . . . 86
writeDouble(double v) (java.io.DataOutputStream) . . . 85
writeDouble(double v) (java.io.ObjectOutputStream) . . . 86
writeFloat(float v) (java.io.DataOutputStream) . . . 85
writeFloat(float v) (java.io.ObjectOutputStream) . . . 86
writeInt(int v) (java.io.DataOutputStream) . . . 85
writeInt(int v) (java.io.ObjectOutputStream) . . . 86
writeLong(long v) (java.io.DataOutputStream) . . . 85
writeLong(long v) (java.io.ObjectOutputStream) . . . 87
writeObject(Object obj) (java.beans.XMLEncoder) . . . 88
writeObject(Object obj) (java.io.ObjectOutputStream) . . . 87
Writer (java.io) . . . 79
  append() . . . 80
  close() . . . 80
  flush() . . . 80
  write() . . . 80
writeShort(int v) (java.io.DataOutputStream) . . . 85
writeShort(int v) (java.io.ObjectOutputStream) . . . 87
writeUTF(String str) (java.io.DataOutputStream) . . . 85
writeUTF(String str) (java.io.ObjectOutputStream) . . . 87

X

XmlAdapter (javax.xml.bind.annotation.adapters) . . . 26
XMLDecoder (java.beans) . . . 88
  readObject() . . . 88
XMLEncoder (java.beans) . . . 88

writeObject(Object obj) 88
XPathExpression (org.jdom2.xpath)
evaluate() 31
XPathFactory (org.jdom2.xpath) 30
compile() 30

Z

ZipEntry (java.util.zip) 93, 94
closeEntry() 93
ZipFile (java.util.zip) 93, 94
getInputStream(...) 93
ZipOutputStream (java.util.zip) 93, 94

Stichwortverzeichnis

A

Abgeleitete Klasse ... 101
Abstract Window Toolkit ... 11
Adler32 ... 98
Annotation ... 24, 38
Array ... 60
  Länge bestimmen ... 61
ASCII ... 79
Attribut ... 21, 22, 101
Auszeichnungssprache ... 18
Autoboxing ... 88, 135
AWT ... 11

B

Basisklasse ... 101, 109
Basisklassenobjekt ... 102
Baum ... 47, 144
Baumstruktur ... 47, 144, 145
Beziehung
  hat-ein ... 109
  ist-ein ... 109
Binärbaum ... 68
Boxing ... 135
Buffer ... 79

C

Callback-Prinzip ... 23
Cascading Style Sheets ... 18, 20
Castor ... 23
CDATA ... 22
checksum ... 98
CLASSPATH ... 152
Closures ... 128
Comparator ... 67
Compiler ... 10
CRC ... 99
CRC32 ... 98
CSS ... 18, 20
CSV ... 43
  Bibliothek
    commons-csv ... 45

D

Datei ... 74
  erzeugen ... 75
  komprimieren ... 91
  Prüfsumme ermitteln ... 98
  Textdatei ... 79
    lesen ... 81
    schreiben ... 79
  vorhanden ... 75
Datei-Browser ... 145
Dateieintrag ... 145
Dateisysteme ... 47, 144
Datenhaltungsobjekt ... 54
Datenstruktur ... 60
Deklaration ... 21
delegate ... 15
desginpattern ... 15
DOCTYPE ... 18
Document Type Definition ... 18
DOM ... 23, 28
DOM4J ... 23
Downcast ... 112
DTD ... 18

E

eaget evaluation ... 132
Einbettung ... 109
Eintrag ... 47, 144
Element ... 20
  Wurzelelement ... 20
Encoding ... 79, 82
  ASCII ... 79
  ISO8859-15 ... 79
  System ... 82
  unterstützte anzeigen ... 82
  UTF-16 ... 79
  UTF-8 ... 79
Entity ... 21
Entwurfsmuster ... 15, 148
  Adapter ... 135, 148
  Code-as-Data ... 131
  Model–View–Control ... 135
  MVC ... 135
  Pipes & Filters ... 132
  Wrapper ... 135
Enum ... 70
Extensible HyperText Markup
  Language ... 19
Extensible Markup Language ... 18
Extensible Stylesheet Language ... 19

F

Fachinformatiker
  Anwendungsentwicklung ... 5
  Systemintegration ... 5
FIFO ... 69
Filterung ... 131
fluent ... 131
Formatierung ... 135
Functional Interface Types ... 129
Funktionale Programmierung ... 128
Funktionstypen ... 129

G

Garbage Collector ... 72
generische Programmierung ... 117
GPS ... 18

H

Hashfunktion ... 98
  kryptographisch ... 98
Hashing ... 72
Hashwert ... 72, 98
HTML ... 18
  Parser ... 43
HTML5 ... 19
Hypertext Markup Language ... 18

I

Instanzbildung ... 102
Integrität ... 98
Interface ... 110
ISO8859-15 ... 79
IT-Berufe ... 5
Iterator ... 66

J

JAR ... 147, 152
  Archiv ... 147
  Viewer ... 147
Java
  Compiler ... 10
  Generics ... 121
  Laufzeitumgebung ... 10
  Version ermitteln ... 10
JavaBean ... 89
JavaFX ... 11
JAXB ... 23
  Datei lesen ... 27
  Datei schreiben ... 27
JDOM ... 23, 28
jsoup ... 43
JUnit ... 37
JVM ... 152
  Parameter
    classpath ... 152
    cp ... 152

K

Klasse ... 101
  abgeleitet ... 101, 109, 111
  anonym ... 14
  Basisklasse ... 101, 111
  Basisklassenunterobjekt ... 111
  ClassLoader ... 152
  CLASSPATH ... 152
  Downcast ... 111
  Hierarchie ... 110
  innere Klasse ... 14
  laden ... 152
  Subklasse ... 101
  Superklasse ... 101
  Upcast ... 111
Kodierung ... 81
Kommentar ... 22
Komposition ... 102, 109
Komprimierung ... 91
  7Z ... 94, 97
  als Stream ... 91

Apache Commons Compress ... 94
AR ... 94
BZIP2 ... 94, 95
CPIO ... 94
GZIP ... 92, 94
im Kontainer ... 91
JAR ... 94
TAR ... 94, 96
verlustbehaftet ... 91
verlustfrei ... 91
XZ ... 94
ZIP ... 93, 94
Konstanten ... 70
Konstruktor ... 104

L
Lambda ... 14, 128
Body ... 130
Lambdas ... 128
Parameterliste ... 129
Streams ... 131
Laufzeittypidentifizierung ... 119
Laufzeitumgebung ... 10
lazy evaluation ... 132
LIFO ... 69
Listen
Hilfsmethoden ... 66
Iterator ... 66
sortieren ... 66
Lizenz ... 6
CC-BY-NC-ND ... 6

M
MANIFEST.MF ... 147
Markup Languages ... 18
MathML ... 18
MD5 ... 100
MDx ... 98
Mehrfachvererbung ... 110, 125
Methode ... 101
Aufruf ... 114
nonvirtual ... 114
polymorph ... 112
static ... 114
virtual ... 114
MIME ... 56
Model-View-Controller ... 15
Modifizierer ... 108
MVC ... 15, 135

N
namespace ... 12
NIO ... 76

P
Paketname ... 12
Parallelverarbeitung ... 128
Paritätsbit ... 98
Parsen ... 135
Parser ... 20
Pipe ... 30, 132
Polymorphie ... 111
Processing Instruction ... 21
Programmierung
Deklarativ ... 131
Property ... 24
Prüfsummen ... 98
Adler-32 ... 99
CRC-32 ... 99
MD5 ... 100
SHA-2 ... 100
SHA512 ... 100
Prüfverfahren ... 98
Puffer ... 79
PULL ... 23
PUSH ... 23

Q
Quersumme ... 98
Queue ... 69

R
Reduktion ... 131
Redundanzprüfung ... 99
Referenzklassen ... 88
RFC 4180 ... 43
RSS ... 18
RTTI ... 119

S
SAM ... 129
SAX ... 23, 28
SGML ... 18
SHA-2 ... 100
SHAx ... 98
Single Abstract Method ... 129
Sortierung
alphabetisch ... 67
Zeichenketten ... 66
Spezialisierung ... 109
Standard Generalized Markup
Language ... 18
Standard Widget Toolkit ... 11
StAX ... 23
strict evaluation ... 132
Stylesheet ... 20
Subklasse ... 101
Superklasse ... 101
SVG ... 18
Swing ... 11, 47
Anzeige synchronisieren ... 52, 56
Baum ... 47
Größe festlegen ... 49
HTML-Ansicht ... 56
Scrollbalken ... 49
Tabelle ... 52
Tree ... 47
SwingX ... 154
SWT ... 11

T
Tabelle ... 47
Tag ... 20
Testgetriebene Entwicklung ... 37
Transformation ... 131
Typsicherheit ... 64
Typumwandlung ... 112

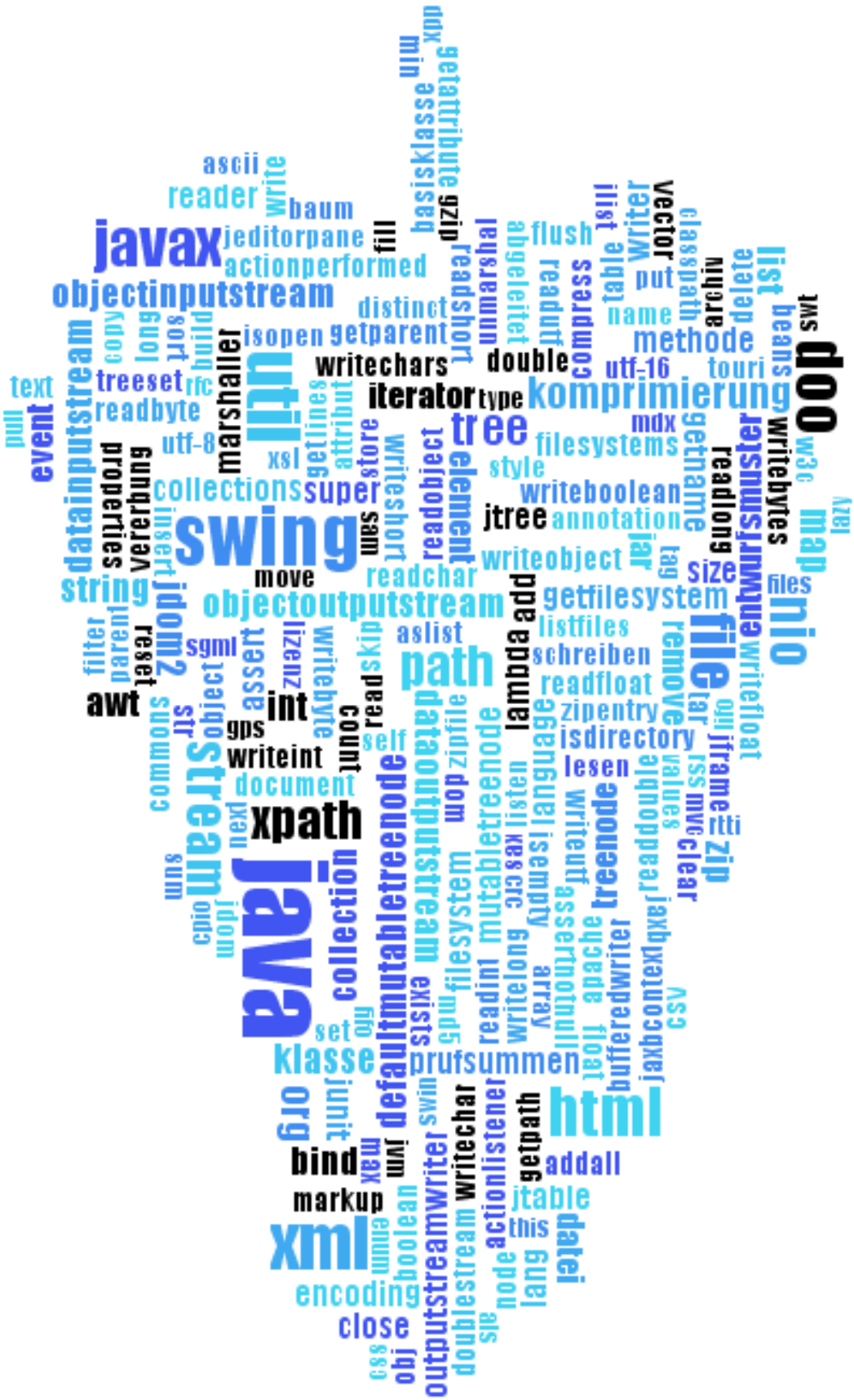
U
Überladung ... 107, 108, 112
Überschreiben ... 107, 108, 112
Unboxing ... 88, 135
Unicodecodepoint ... 66
Upcast ... 112
UTF-16 ... 79
UTF-8 ... 79, 80

V
Verarbeitungshinweis ... 21
Verdecken ... 107
Vererbung ... 101, 102, 109
Prinzip ... 101
Verzeichnis ... 74
Inhalt auflisten ... 75
Unterverzeichnis ... 145

W
W3C ... 18, 23
Warteschlangen ... 69
Wrapper-Klassen ... 88, 135
Wurzelelement ... 20

X
XHTML ... 19
XML ... 18, 20, 23
Deklaration ... 21
DOM-API ... 23
Kommentare ... 22
lesen ... 27
Mapping-API ... 23
marshal ... 27
Namensraum ... 24
Pull-API ... 23
Push-API ... 23
root-Element ... 24
schreiben ... 27
Sortierreihenfolge ... 24
unmarshal ... 27
Verknüpfungen ... 24
verschachteln ... 25
wohlgeformt ... 22
XML Path Language ... 30
XML Schema ... 18
XML Schema Definition ... 18
XOM ... 23
XPATH ... 30
Abkürzungen ... 35
comment() ... 34
Knotentypen ... 34
node() ... 34
processing-instruction() ... 34
text() ... 34
XPP ... 23
XSD ... 18
XSL ... 18–20
XStream ... 23

Z
Zahlen
formatieren ... 153
Zeichenketten
sortieren ... 66
ZIP ... 147
Archiv ... 147
Zugriffsebenen ... 102
Zugriffsmodifizierer ... 108


javax
swing
java
util
oop
komprimierung
tree
nio
file
path
stream
xpath
collection
defaultmutabletreenode
datainputstream
objectinputstream
objectoutputstream
klasse
prufsummen
html
xml
bind
markup
encoding
close
int
awt
string
event
text
reader
ascii
write
baum
jeditorpane
actionperformed
distinct
getparent
isopen
writechars
double
iterator
type
utf-16
touri
methode
name
flush
put
table
writer
vector
list
beans
delete
classpath
archiv
swt
compress
unmarshal
readdt
readshort
getattribute
basisklasse
gzip
min
dtx
fill
treeset
rfc
readbyte
utf-8
marshaller
getlines
attribut
store
writeshort
sam
readobject
element
jtree
annotation
writeboolean
collections
super
isx
style
filesystems
mdx
getname
readlong
writebytes
w3c
map
lazy
properties
vererbung
insert
jdom2
move
readchar
writeobject
tag
size
files
entwurfsmuster
writefloat
getfilesystem
listfiles
aslist
add
lambda
schreiben
readfloat
remove
zipentry
isdirectory
lesen
iframe
rss
mvc
clear
zip
rtti
values
readdouble
jaxbcontext
jaxb
asc
csv
bufferedwriter
apache
treenode
writeint
assertmontnull
array
isempty
listen
xsd
dom
zipfile
self
count
readskip
gps
writeint
document
sgml
reset
filter
parent
object
str
commons
sun
jdom
cpio
next
assert
writebyte
lizenz
datoutputstream
mutabletreenode
readint
writelong
filesystem
exists
md5
set
log
junit
org
swing
max
jvm
getpath
addall
jtable
this
dtd
actionlistener
writechar
boolean
enum
outputstreamwriter
doublestream
node
lang
datei
obj
als
ssl