Bernhard Baltes-Götz

Einführung in das Programmieren mit Java

2004.02.25

Herausgeber:	Universitäts-Rechenzentrum Trier
	Universitätsring 15
	D-54286 Trier
	WWW: http://www.uni-trier.de/urt/urthome.shtml
	E-Mail: urt@uni-trier.de
	Tel.: (0651) 201-3417, Fax.: (0651) 3921
Leiter:	Prof. Dr.-Ing. Manfred Paul
Autor:	Bernhard Baltes-Götz (E-Mail: baltes@uni-trier.de)
Druck:	Druckerei der Universität Trier

Copyright © 2003; URT

Vorwort

Dieses Manuskript entstand als Begleitlektüre zum Java-Einführungskurs, den das Universitäts-Rechenzentrum Trier (URT) im Sommersemester 2003 angeboten hat.

Lernziele

Java ist zwar mit dem Internet groß geworden, hat sich jedoch mittlerweile als universelle, für vielfältige Zwecke einsetzbare Programmiersprache etabliert. Im Kurs geht es nicht primär um Kochrezepte zur Erstellung lebendiger Webseiten, sondern um die systematische Einführung in das Programmieren mit Java. Dabei werden wichtige Konzepte und Methoden der Softwareentwicklung vorgestellt, wobei die objektorientierte Programmierung einen großen Raum einnimmt.
Der Java-Einsatz im Internet soll aber ebenfalls berücksichtigt werden.

Voraussetzungen bei den Leser(innen)

- EDV-Allgemeinbildung
  Dass die Leser(innen) wenigstens durchschnittliche Erfahrungen bei der *Anwendung* von Computerprogrammen haben sollten, versteht sich von selbst.
  Im Text wird zwar als Entwicklungsumgebung das Programm JCreator unter MS-Windows benutzt, doch sind die Kursinhalte weitestgehend betriebssystem-unabhängig.
- Programmierkenntnisse werden *nicht* vorausgesetzt.
  Leser(innen) *mit* Programmiererfahrung werden sich bei den ersten Abschnitten eventuell etwas langweilen.
- Motivation
  Generell ist mit einem erheblichen Zeitaufwand bei der Lektüre und bei der aktiven Auseinandersetzung mit dem Stoff (z.B. durch das Lösen von Übungsaufgaben) zu rechnen.

Software zum Üben

Für die unverzichtbaren Übungen sollte ein Rechner zur Verfügung stehen, auf dem die Standard Edition des Java 2 Software Development Kits (Java 2 SDK SE) der Firma Sun in einer Version ab 1.2 mitsamt der zugehörigen Dokumentation installiert ist. Nähere Hinweise zum Bezug und zur Verwendung der Software folgen in Abschnitt 2.

Dateien zum Kurs

Dieses Kursmanuskript ist zusammen mit den behandelten Beispielen und Lösungsvorschlägen zu einigen Übungsaufgaben im Internet ausgehend von der Startseite der Universität Trier (www.uni-trier.de) auf folgendem Weg zu finden:

Weitere Serviceangebote > EDV-Dokumentationen > Elektronische Publikationen >
> Programmentwicklung > Einführung in das Programmieren mit Java

Im Campusnetz der Universität Trier sind die Dateien noch bequemer über eine Netz-Freigabe zugänglich, nachdem man sich bei einem Windows-Rechner mit Einbindung in die NT-Domäne URT angemeldet hat. Führen Sie dort nach

Start > Ausführen

den Befehl

k baltes

aus, um die Netz-Freigabe als Laufwerk **K:** in Ihr Windows-System einzubinden.
Anschließend finden Sie im Verzeichnis

K:\Java

die Datei **java.pdf** mit dem Manuskript (im PDF-Format) sowie das Unterverzeichnis **BspUeb** mit Beispielen und Lösungsvorschlägen zu Übungsaufgaben.

Trier, im September 2003 Bernhard Baltes-Götz

1 EINLEITUNG 1

1.1 Beispiel für die objektorientierte Softwareentwicklung mit Java 1
1.1.1 Objektorientierte Analyse und Modellierung 1
1.1.2 Objektorientierte Programmierung 3
1.1.3 Algorithmen 5
1.1.4 Die main()-Methode 5
1.1.5 Ausblick auf Anwendungen mit graphischer Benutzerschnittstelle 6

1.2 Quellcode, Bytecode und Maschinencode 7

1.3 Java als Programmiersprache und als Klassenbibliothek 8

1.4 Zentrale Merkmale der Java-Technologie 9

1.5 Ausblick auf die Erstellung von Java-Applets 11

2 WERKZEUGE ZUM ENTWICKELN VON JAVA-PROGRAMMEN 13

2.1 Java-Entwicklung mit Texteditor und SDK 13
2.1.1 Editieren 13
2.1.2 Kompilieren 15
2.1.3 Ausführen 16
2.1.4 Pfad für class-Dateien setzen 16
2.1.5 Programmfehler beheben 17

2.2 Java-Entwicklung mit dem JCreator LE 2.x 18

2.3 Übungsaufgaben zu Abschnitt 2 21

3 ELEMENTARE SPRACHELEMENTE 22

3.1 Einstieg 22
3.1.1 Aufbau einer Java-Applikation 22
3.1.2 Kommentare 24
3.1.3 Bezeichner 25
3.1.4 Einfache Ausgabe bei Konsolenanwendungen 26
3.1.5 Übungsaufgaben zu Abschnitt 3.1 26

3.2 Variablen und primitive Datentypen 27
3.2.1 Primitive Datentypen 29
3.2.2 Variablendeklaration, Initialisierung und Wertzuweisung 30
3.2.3 Blöcke und Gültigkeitsbereiche für lokale Variablen 31
3.2.4 Finalisierte Variablen (Konstanten) 33
3.2.5 Literale 34
3.2.5.1 Ganzzahl-Literale 34
3.2.5.2 Gleitkomma-Literale 34
3.2.5.3 char-Literale 35
3.2.5.4 Zeichenketten-Literale 36
3.2.5.5 boolean-Literale 36
3.2.6 Übungsaufgaben zu Abschnitt 3.2 37

3.3 Einfache Eingabe bei Konsolenanwendungen 38
3.3.1 Beschreibung der Klasse `Simput` 38
3.3.2 Simput-Installation für die SDK-Werkzeuge und JCreator 39
3.3.3 Übungsaufgabe zu Abschnitt 3.3 40

3.4 Operatoren und Ausdrücke 41
3.4.1 Arithmetische Operatoren 41
3.4.2 Methodenaufrufe 44
3.4.3 Vergleichsoperatoren 44
3.4.4 Logische Operatoren 45
3.4.5 Bitorientierte Operatoren 47
3.4.6 Typumwandlung (Casting) 48
3.4.7 Zuweisungsoperatoren 50
3.4.8 Konditionaloperator 51
3.4.9 Auswertungsreihenfolge 51
3.4.10 Über- und Unterlauf 54
3.4.11 Übungsaufgaben zu Abschnitt 3.4 55

3.5 Anweisungen 57
3.5.1 Überblick 57
3.5.2 Bedingte Anweisung und Verzweigung 58
3.5.2.1 if-Anweisung 58
3.5.2.2 switch-Anweisung 60
3.5.3 Wiederholungsanweisung 62
3.5.3.1 Zählergesteuerte Schleife (for) 62
3.5.3.2 Bedingungsabhängige Schleifen 63
3.5.3.2.1 while-Schleife 63
3.5.3.2.2 do-Schleife 64
3.5.3.3 Schleifen(durchgänge) vorzeitig beenden 64
3.5.4 Übungsaufgaben zu Abschnitt 3.5 65

4 KLASSEN UND OBJEKTE 68

4.1 Überblick, historische Wurzeln, Beispiel 68
4.1.1 Einige Kernideen und Vorzüge der OOP 68
4.1.2 Strukturierte Programmierung und OOP 69
4.1.3 Auf-`Bruch` zu echter Klasse 70

4.2 Instanzvariablen (Eigenschaften) 72
4.2.1 Deklaration 72
4.2.2 Gültigkeitsbereich und Ablage im Hauptspeicher 73
4.2.3 Initialisierung 73
4.2.4 Zugriff durch eigene und fremde Methoden 74

4.3 Methoden 74
4.3.1 Methodendefinition 75
4.3.1.1 Methodenkopf, Formalparameter 75
4.3.1.2 Rückgabewerte, return-Anweisung 77
4.3.1.3 Methodenrumpf 78
4.3.2 Methodenaufruf, Aktualparameter 78
4.3.3 Methoden überladen 79

4.4 Objekte 80
4.4.1 Referenzvariablen definieren, Klassen als Datentypen 80
4.4.2 Objekte erzeugen 81
4.4.3 Überflüssige Objekte entfernen, Garbage Collector 82

4.5 Konstruktoren 84

4.6 Referenzen 85
4.6.1 Referenzparameter 85
4.6.2 Rückgabewerte vom Referenztyp 86
4.6.3 this als Referenz auf das aktuelle Objekt 87

4.7 Klassenbezogene Eigenschaften und Methoden 87
4.7.1 Klassenvariablen 87
4.7.2 Klassenmethoden 88
4.7.3 Wiederholung zu den Variablentypen in Java 89

4.8 Rekursive Methoden 90

4.9 Aggregation 92

4.10 Übungsaufgaben zu Abschnitt 1 94

5 ELEMENTARE KLASSEN 97

5.1 Arrays (Felder) 97
5.1.1 Arrays deklarieren 97
5.1.2 Arrays erzeugen 98
5.1.3 Arrays benutzen 99
5.1.4 Beispiel: Beurteilung des Java-Pseudozufallszahlengenerators 100
5.1.5 Initialisierungslisten 102
5.1.6 Objekte als Array-Elemente 103
5.1.7 Mehrdimensionale Felder 103
5.1.8 Übungsaufgaben zu Abschnitt 2.1 105

5.2 Zeichenketten 106
5.2.1 Die Klasse String für konstante Zeichenkette 106
5.2.1.1 Implizites und explizites Erzeugen von String-Objekten 106
5.2.1.2 Der interne String-Pool 107
5.2.1.3 String als Read Only - Klasse 108
5.2.1.4 Methoden für String-Objekte 109
5.2.1.4.1 Verketten von Strings 109
5.2.1.4.2 Vergleichen von Strings 109
5.2.1.4.3 Länge einer Zeichenkette 111
5.2.1.4.4 Zeichen extrahieren, suchen oder ersetzen 111
5.2.1.4.5 Groß-/Kleinschreibung normieren 112
5.2.2 Die Klasse StringBuffer für veränderliche Zeichenketten 113
5.2.3 Übungsaufgaben zu Abschnitt 2.2 114

5.3 Verpackungs-Klassen für primitive Datentypen 116
5.3.1 Einweg-Verpackungen 116
5.3.2 Konvertierungs-Methoden 118
5.3.3 Konstanten mit Grenzwerten 118
5.3.4 Character-Methoden zur Zeichen-Klassifikation 118
5.3.5 Übungsaufgaben zu Abschnitt 2.3 119

6 PAKETE 120

6.1 Das API der Java 2 Standard Edition 120

6.2 Pakete erstellen 123
6.2.1 package-Anweisung und Paketordner 123
6.2.2 Unterpakete 124
6.2.3 Paketunterstützung in der JCreator-Entwicklungsumgebung 126

6.3 Pakete verwenden 127
6.3.1 Verfügbarkeit der Dateien 127
6.3.2 Namensregeln 127

6.4 Zugriffsmodifikatoren 129
6.4.1 Zugriffsschutz für Klassen 129
6.4.2 Zugriffsschutz für Variablen und Methoden 129

6.5 Java-Archivdateien 130
6.5.1 Archivdateien mit jar erstellen 131
6.5.2 Archivdateien verwenden 132
6.5.3 Ausführbare JAR-Dateien 133

6.6 Übungsaufgaben zu Abschnitt 3 133

7 VERERBUNG UND POLYMORPHIE 136

7.1 Definition einer abgeleiteten Klasse 138

7.2 Der Zugriffsmodifikator protected 139

7.3 super-Konstruktoren und Initialisierungs-Sequenzen 140

7.4 Überschreiben und Überdecken 141
7.4.1 Methoden überschreiben 141
7.4.2 Finalisierte Methoden und Klassen 143
7.4.3 Elementvariablen überdecken 144

7.5 Polymorphie 144

7.6 Abstrakte Methoden und Klassen 146

7.7 Übungsaufgaben zu Abschnitt 1 147

8 AUSNAHME-BEHANDLUNG 149

8.1 Unbehandelte Ausnahmen 149

8.2 Ausnahmen abfangen 151
8.2.1 Die try-Anweisung 151
8.2.2 Programmablauf bei der Ausnahmebehandlung 152
8.2.3 Vergleich mit der traditionellen Ausnahmebehandlung 155
8.2.4 Diagnostische Ausgaben 156

8.3 Ausnahme-Klassen in Java 157

8.4 Obligatorische und freiwillige Ausnahmebehandlung 158

8.5 Ausnahmen auslösen (throw) und deklarieren (throws) 159

8.6 Ausnahmen definieren 161

8.7 Übungsaufgaben zu Abschnitt 2 162

9 INTERFACES 163

9.1 Interfaces definieren 163

9.2 Interfaces implementieren 165

9.3 Interfaces als Referenz-Datentypen verwenden 168

9.4 Übungsaufgaben zu Abschnitt 3 168

10 EIN-/AUSGABE ÜBER DATENSTRÖME 170

10.1 Grundprinzipien 170
10.1.1 Datenströme 170
10.1.2 Taxonomie der Stromverarbeitungsklassen 172
10.1.3 Zum guten Schluss 172
10.1.4 Aufbau und Verwendung der Transformationsklassen 173

10.2 Verwaltung von Dateien und Verzeichnissen 175
10.2.1 Verzeichnis anlegen 175
10.2.2 Dateien explizit erstellen 175
10.2.3 Informationen über Dateien und Ordner 176
10.2.4 Verzeichnisinhalte auflisten 176
10.2.5 Umbenennen 177
10.2.6 Löschen 177

10.3 Klassen zur Verarbeitung von Byte-Strömen 178
10.3.1 Die OutputStream-Hierarchie 178
10.3.1.1 Überblick 178
10.3.1.2 FileOutputStream 178
10.3.1.3 DataOutputStream 180
10.3.1.4 BufferedOutputStream 181
10.3.1.5 PrintStream 183
10.3.2 Die InputStream-Hierarchie 185

10.4 Klassen zur Verarbeitung von Character-Strömen 186
10.4.1 Die Writer-Hierarchie 186
10.4.1.1 Überblick 186
10.4.1.2 Brückenklassen und Encodings, OutputStreamWriter 187
10.4.1.3 Implizite und explizite Pufferung, BufferedWriter 189
10.4.1.4 PrintWriter 190
10.4.1.5 Umlaute in Java-Konsolenanwendungen unter Windows 192
10.4.1.6 FileWriter 193
10.4.1.7 Sonstige Writer-Subklassen 194
10.4.2 Die Reader-Hierarchie 194
10.4.2.1 Überblick 194
10.4.2.2 FileReader 195
10.4.2.3 Eingaben an der Konsole 196

10.5 Objektserialisierung 197

10.6 Empfehlungen zur Verwendung der IO-Klassen 199
10.6.1 Textdaten in sequentielle Dateien schreiben 199
10.6.2 Textdaten aus sequentiellen Dateien lesen 200
10.6.3 Ausgabe auf die Konsole 200
10.6.4 Tastatureingaben 201
10.6.5 Objekte schreiben und lesen 201
10.6.6 Primitive Datentypen binär in eine sequentielle Datei schreiben 202
10.6.7 Primitive Datentypen aus einer sequentiellen Binärdatei lesen 202

10.7 Missbilligte Methoden 203

10.8 Übungsaufgaben zu Abschnitt 1 204

11 GUI-PROGRAMMIERUNG MIT SWING/JFC 207

11.1 Java Foundation Classes 207

11.2 Elementare Klassen im Paket javax.swing 208
11.2.1 Komponenten und Container 208
11.2.2 Top-Level Container 210

11.3 Beispiel für eine Swing-Anwendung 211

11.4 Swing-Komponenten, Teil 1 213
11.4.1 Label 213
11.4.2 Befehlsschalter 214
11.4.3 Zubehör für Swing-Komponenten 214
11.4.4 Standarddialoge 215

11.5 Die Layout-Manager der Container 216
11.5.1 BorderLayout 217
11.5.2 GridLayout 218
11.5.3 FlowLayout 219
11.5.4 Freies Layout 219

11.6 Ereignisbehandlung 220
11.6.1 Das Delegationsmodell 220
11.6.2 Ereignistypen und Ereignisklassen 221
11.6.3 Ereignisempfänger registrieren 222
11.6.4 Adapterklassen 223
11.6.5 Schließen von Fenstern und GUI-Programmen 224
11.6.6 Optionen zur Definition von Ereignisempfängern 225
11.6.6.1 Innere Klassen als Ereignisempfänger 225
11.6.6.2 Anonyme Klassen als Ereignisempfänger 226
11.6.6.3 Do-It-Yourself – Ereignisbehandlung 227
11.6.7 Tastatur- und Mausereignisse 227
11.6.7.1 KeyEvent 227
11.6.7.2 MouseEvent 228

11.7 Swing-Komponenten, Teil 2 229
11.7.1 Einzeilige Text-Komponenten 229
11.7.2 Kontrollkästchen und Optionsfelder 231
11.7.3 Kombinationsfelder 233
11.7.4 Ein (fast) kompletter Editor als Swing-Komponente 234
11.7.5 Menüs 235
11.7.6 Dateiauswahldialog 236

11.8 Look & Feel umschalten 236

11.9 Übungsaufgaben zu Abschnitt 1 238

12 APPLETS 241

12.1 Stammbaum der Applet-Basisklasse 242

12.2 Applet-Start via Browser oder Appletviewer 243

12.3 Methoden für kritische Lebensereignisse 245

12.4 Sonstige Methoden für die Applet-Browser-Kooperation 247
12.4.1 Parameter übernehmen 247
12.4.2 Browser-Statuszeile ändern 248
12.4.3 Andere Webseiten öffnen 248

12.5 Das Java-Browser-Plugin 251

12.6 Übungsaufgaben zu Abschnitt 2 251

13 MULTIMEDIA 253

13.1 Grafik 253
13.1.1 Die Klasse Graphics 253
13.1.2 Das Koordinatensystem der Zeichenfläche 254
13.1.3 Organisation der Grafikausgabe 255
13.1.3.1 System-initiierte Aktualisierung 255
13.1.3.2 Programm-initiierte Aktualisierung 258
13.1.3.3 Details zur Grafikausgabe in Swing 260

13.2 Sound 263

13.3 Übungsaufgaben zu Abschnitt 3 265

14 THREADS 266

14.1 Threads erzeugen 266

14.2 Threads synchronisieren 270
14.2.1 Monitore 270
14.2.2 Koordination per wait() und notify() 271

14.3 Das Interface Runnable 272

14.4 Threads unterbrechen 274

14.5 Threads stoppen 275

14.6 Thread-Lebensläufe 277
14.6.1 Scheduling und Prioritäten 277
14.6.2 Zustände von Threads 279

14.7 Sonstige Thread-Themen 279
14.7.1 Daemon-Threads 279
14.7.2 Deadlocks 280
14.7.3 Threads und Swing 281

14.8 Zusammenfassung 282

14.9 Übungsaufgaben zu Abschnitt 1 283

15 LITERATUR 286

16 ANHANG 287

16.1 Operatorentabelle 287

16.2 Lösungsvorschläge zu den Übungsaufgaben 288
Abschnitt 2 (Werkzeuge zum Entwickeln von Java-Programmen) 288
Abschnitt 3 (Elementare Sprachelemente) 288
Abschnitt 3.1 (Einstieg) 288
Abschnitt 3.2 (Variablen und primitive Datentypen) 289
Abschnitt 3.3 (Einfache Eingabe bei Konsolenanwendungen) 290
Abschnitt 3.4 (Operatoren und Ausdrücke) 290
Abschnitt 3.5 (Anweisungen) 291
Abschnitt 4 (Klassen und Objekte) 292
Abschnitt 5 (Elementare Klassen) 293
Abschnitt 5.1 (Arrays) 293
Abschnitt 5.2(Zeichenketten) 293
Abschnitt 5.3 (Verpackungs-Klassen für primitive Datentypen) 293
Abschnitt 6 (Pakete) 294
Abschnitt 7 (Vererbung und Polymorphie) 294
Abschnitt 8 (Ausnahme-Behandlung) 295

Abschnitt 9 (Interfaces) 295
Abschnitt 10 (Ein-/Ausgabe über Datenströme) 295
Abschnitt 11 (GUI-Programmierung mit Swing/JFC) 296
Abschnitt 12 (Applets) 297
Abschnitt 13 (Multimedia) 297
Abschnitt 14 (Threads) 297

1 Einleitung

1.1 Beispiel für die objektorientierte Softwareentwicklung mit Java

In diesem Abschnitt soll eine Vorstellung davon vermittelt werden, was ein Computerprogramm (in Java) ist, und wie man es erstellt. Dabei kommen einige Grundbegriffe der Informatik zur Sprache, wobei eine ausführliche Behandlung allerdings nicht möglich ist.
Ein Computerprogramm besteht im Wesentlichen (von Bildern, Klängen und anderen Ressourcen einmal abgesehen) aus einer Menge von wohlgeformten und wohlgeordneten Definitionen und Anweisungen zur Bewältigung einer bestimmten Aufgabe. Dazu muss das Programm ...

- den betroffenen Gegenstandsbereich modellieren,
- Algorithmen realisieren, die in endlich vielen Schritten und unter Verwendung von endlich vielen Ressourcen (z.B. Speicher) bestimmte Ausgangszustände in akzeptable Zielzustände überführen.

Wir wollen präzisere und komplettere Definitionen zum komplexen Begriff eines Computerprogramms den Lehrbüchern überlassen (siehe z.B. Echtle & Goedicke 2000, Goll et al. 2000) und stattdessen ein Beispiel betrachten, um einen Einstieg in die Materie zu finden.
Bei der Suche nach einem geeigneten Java-Einstiegsbeispiel tritt allerdings ein Dilemma auf:

- Einfache Beispiele sind für das Programmieren mit Java nicht besonders repräsentativ, z.B. ist von der Objektorientierung außer einem gewissen Formalismus eigentlich nichts vorhanden.
- Repräsentative Java-Programme sind wegen ihrer Länge und Komplexität (aus der Sicht des Anfängers) als Einstiegsbeispiel nicht gut geeignet.

Im folgenden Beispielprogramm wird trotz angestrebter Einfachheit *nicht* auf objektorientiertes Programmieren (OOP) verzichtet. Seine Aufgabe besteht darin, für Brüche mit ganzzahligem Zähler und Nenner elementare Operationen auszuführen (Kürzen, Addieren), womit es etwa einem Schüler beim Anfertigen der Hausaufgaben (zur Kontrolle der eigenen Lösungen) nützlich sein kann.

1.1.1 Objektorientierte Analyse und Modellierung

Einer objektorientierten Programmentwicklung geht die **objektorientierte Analyse** der Aufgabenstellung voran. Dabei versucht man, alle beteiligten **Objekt-Sorten** zu identifizieren und definiert für sie jeweils eine **Klasse**, die durch **Eigenschaften** (**Instanzvariablen**) und **Handlungskompetenzen** (**Methoden**) gekennzeichnet ist. Dass jedes Objekt gleich in eine Klasse („Schublade“) gesteckt wird, mögen die Anhänger einer ausgeprägt individualistischen Weltanschauung bedauern. Auf einem geeigneten Abstraktionsniveau betrachtet lassen sich jedoch die meisten Objekte der realen Welt ohne großen Informationsverlust in Klassen einteilen.
In unserem einfachen Beispiel kann man sich bei der objektorientierten Analyse wohl auf die Klasse der *Brüche* beschränken. Beim möglichen Ausbau des Programms zu einem Bruchrechnungs-Trainer kommen jedoch weitere Klassen hinzu (z.B. Aufgabe, Übungsaufgabe, Testaufgabe).

Dass Zähler und Nenner die zentralen Eigenschaften eines Bruchs sind, bedarf keiner Begründung. Sie werden in der Klassendefinition durch ganzzahlige Variablen (Java-Datentyp **int**) repräsentiert:

- `zaehler`
- `nenner`

Im objektorientierten Paradigma ist jede Klasse für die Manipulation ihrer Eigenschaften selbst verantwortlich. Diese sollen **eingekapselt** und vor direktem Zugriff durch Objekte aus fremden Klassen geschützt sein. So ist sicher gestellt, dass nur sinnvolle Änderungen der Eigenschaften möglich

sind. Außerdem wird aus später zu erläutenden Gründen die Produktivität der Softwareentwicklung gefördert.

Die Handlungskompetenzen (Methoden) einer Klasse bilden demgegenüber ihre öffentlich zugängliche **Schnittstelle**. Ihre Objekte sind in der Lage, auf eine Reihe von **Nachrichten** mit einem bestimmten Verhalten zu reagieren. In unserem Beispiel sollte die Klasse `Bruch` z.B. eine Methode zum Kürzen besitzen. Dann kann einem konkreten `Bruch`-Objekt die Nachricht zugestellt werden, diese Methode auszuführen.
Sich unter einem `Bruch` ein Objekt vorzustellen, das Nachrichten empfängt und mit einem passenden Verhalten beantwortet, ist etwas gewöhnungsbedürftig. In der realen Welt sind Brüche, die sich selbst auf Signal hin kürzen, nicht unbedingt alltäglich, wenngleich möglich (z.B. als didaktisches Kinderspielzeug). Das objektorientierte Modellieren eines Gegenstandbereiches ist also nicht unbedingt eine direkte Abbildung, sondern eine Rekonstruktion. Einerseits soll der Gegenstandsbereich im Modell gut repräsentiert sein, um dem Programmentwickler und seinem Auftraggeber den Umgang mit dem fertigen Programm zu erleichtern. Andererseits soll eine möglichst stabile, gut erweiterbare und wieder verwendbare Software entstehen.

Um fremden Objekten trotz Datenkapselung die Veränderung einer Eigenschaft zu erlauben, müssen entsprechende Methoden (mit geeigneten Kontrollmechanismen) angeboten werden. Unsere `Bruch`-Klasse sollte wohl über Methoden zum Verändern von Zähler und Nenner verfügen. Bei einer geschützten Eigenschaft ist auch der direkte *Lese*zugriff ausgeschlossen, so dass im `Bruch`-Beispiel auch noch Methoden zum Ermitteln von Zähler und Nenner ratsam sind. Eine konsequente Umsetzung der Datenkapselung erzwingt also eventuell eine ganze Serie von Methoden zum Lesen und Setzen von Eigenschaftswerten.
Mit diesem Aufwand werden aber erhebliche Vorteile realisiert:

- **Sicherheit**
  Die Eigenschaften sind vor unsinnigen und gefährlichen Zugriffen geschützt, wenn Veränderungen nur über die vom Klassendesigner entworfenen Methoden möglich sind.
- **Effizienz**
  Bei der Entwicklung großer Softwaresysteme unter Beteiligung zahlreicher Programmierer ermöglicht die Datenkapselung eine perfekte Modularisierung. Der Klassendesigner trägt die volle Verantwortung dafür, dass die von ihm entworfenen Methoden korrekt arbeiten. Andere Programmierer müssen beim Verwenden der Klasse lediglich die Methoden kennen. Das Innenleben einer Klasse kann vom Designer nach Bedarf geändert werden, ohne dass andere Programmbestandteile angepasst werden müssen.

Nach obigen Überlegungen sollten die Objekte unserer `Bruch`-Klasse folgende Methoden beherrschen:

- `setzeZaehler(int zpar), setzeNenner(int npar)`
  Das Objekt wird beauftragt, seinen `zaehler` bzw. `nenner` auf einen bestimmten Wert zusetzen. Ein direkter Zugriff auf die Eigenschaften soll fremden Klassen nicht erlaubt sein (Datenkapselung). Bei dieser Vorgehensweise kann das Objekt z.B. verhindern, dass sein Nenner auf 0 gesetzt wird.
- `gibZaehler(), gebNenner()`
  Das `Bruch`-Objekt wird beauftragt, den Wert seiner Zähler- bzw. Nenner-Eigenschaft mitzuteilen. Diese Methoden sind erforderlich, weil ein direkter Zugriff auf die Eigenschaften nicht vorgesehen ist.
- `kuerze()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` zu kürzen. Welcher Algorithmus dazu benutzt wird, bleibt dem Objekt bzw. dem Klassen-Designer überlassen.

- `addiere(Bruch b)`
  Das Objekt wird beauftragt, den als Parameter übergebenen Bruch zum eigenen Wert zu addieren.
- `frage()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` beim Anwender via Konsole (Eingabeaufforderung, „DOS-Fenster“) zu erfragen.
- `zeige()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` auf der Konsole anzuzeigen.

Um die durch objektorientierte Analyse gewonnene Modellierung eines Gegenstandsbereichs standardisiert und übersichtlich zu beschreiben, wurde die **Unified Modeling Language** (UML) entwickelt. Hier wird eine Klasse durch ein Rechteck mit drei Abschnitten dargestellt:

- Oben steht der Name der Klasse.
- In der Mitte stehen die Eigenschaften (Attribute, Instanzvariablen).
  Nach dem Namen einer Eigenschaft gibt man ihren Datentyp (siehe unten) an.
- Unten stehen die Methoden (Handlungskompetenzen).
  In Anlehnung an eine in vielen Programmiersprachen (wie z.B. Java) übliche Syntax zur Methodendefinition gibt man für die Argumente eines Methodenaufrufs sowie für den Rückgabewert (falls vorhanden) den Datentyp an. Was mit letzten Satz genau gemeint ist, werden Sie bald erfahren.

Für die `Bruch`-Klasse erhält man folgende Darstellung:

Bruch
zaehler: int nenner: int
setzeZaehler(int zpar) setzeNenner(int npar):boolean gibZaehler():int gebNenner():int kuerze() addiere(Bruch b) frage():boolean zeige()

Sind bei einer Anwendung mehrere Klassen beteiligt, dann sind auch die Beziehungen zwischen den Klassen wesentliche Bestandteile des Modells.

1.1.2 Objektorientierte Programmierung

In unserem einfachen Beispielprojekt soll nun die `Bruch`-Klasse in der Programmiersprache Java kodiert werden, wobei die Eigenschaften (Attribute, Instanzvariablen) zu deklarieren und die Methoden zu implementieren sind. Es resultiert der so genannte **Quellcode**, der am besten in einer Textdatei namens **Bruch.java** untergebracht wird.
Zwar sind Ihnen die meisten Details dieser Klassendefinition selbstverständlich jetzt noch fremd, doch sind die Variablen-Deklarationen und Methoden-Implementationen als zentrale Bestandteile leicht zu erkennen:

```
class Bruch {
    private int zaehler;
    private int nenner = 1;

    void setzeZaehler(int zpar) {zaehler = zpar;}

    boolean setzeNenner(int n) {
        if (n != 0)   {
            nenner = n;
            return true;
        } else
            return false;
    }

    int gibZaehler() {return zaehler;}

    int gibNenner() {return nenner;}

    void kuerze() {
        // größten gemeinsamen Teiler mit dem Euklidischen Algorithmus bestimmen
        if (zaehler != 0) {
            int ggt = 0;
            int az = Math.abs(zaehler);
            int an = Math.abs(nenner);
            do {
                if (az == an)
                    ggt = az;
                else
                    if (az > an)
                        az = az - an;
                    else
                        an = an - az;
            } while (ggt == 0);

            zaehler /= ggt;
            nenner  /= ggt;
        }
    }

    void addiere(Bruch b) {
        zaehler = zaehler*b.nenner + b.zaehler*nenner;
        nenner = nenner*b.nenner;
        kuerze();
    }

    boolean frage() {
        int n;
        System.out.print("Zaehler: ");
        zaehler = Simput.gint();
        System.out.print("Nenner : ");
        n = Simput.gint();
        if (n != 0) {
            nenner = n;
            return true;
        }
        else {
            System.out.println("Der Nenner darf nicht 0 werden.");
            return false;
        }
    }
```

```
    void zeige() {
        System.out.println("     " + gibZaehler() + "\n" +
                           "  -----\n" +
                           "     " + gibNenner() + "\n");
    }
}
```

1.1.3 Algorithmen

Am Anfang von Abschnitt 1.1 wurden mit der Modellierung des Gegenstandbereichs und der Realisierung von Algorithmen zwei wichtige Aufgaben der Softwareentwicklung genannt, von denen die letztgenannte bisher kaum zur Sprache kam. Auch im weiteren Verlauf des Kurses wird die explizite Diskussion von Algorithmen (z.B. hinsichtlich Voraussetzungen, Korrektheit, Terminierung und Aufwand) keinen großen Raum einnehmen im Vergleich zur Auseinandersetzung mit Java als Sprache und Klassenbibliothek.
Im Einführungsbeispiel treten überwiegend einfache Methoden auf, bei denen man kaum von *Algorithmen* sprechen wird. Eine Ausnahme stellt die Methode `kuerze()` dar, wo über den **euklidischen Algorithmus** der größte gemeinsame Teiler (ggT) von Zähler und Nenner eines Bruchs bestimmt wird, durch den zum optimalen Kürzen beide Zahlen zu dividieren sind. Beim euklidischen Algorithmus wird die leicht zu beweisende Aussage genutzt, dass für zwei natürliche Zahlen u und v ($u > v$) der ggT gleich dem ggT von v und (u-v) ist. Dieses Ergebnis wird in `kuerze()` folgendermaßen ausgenutzt:

> Es wird geprüft, ob Zähler und Nenner identisch sind. Trifft dies zu, ist der ggT gefunden (identisch mit Zähler und Nenner). Anderenfalls wird die größere der beiden Zahlen durch deren Differenz ersetzt, und mit diesem verkleinerten Problem startet das Verfahren neu.

Man erhält auf jeden Fall in endlich vielen Schritten zwei identische Zahlen und damit den ggT.

Der beschriebene Algorithmus eignet sich dank seiner Einfachheit gut für das Einführungsbeispiel, ist aber in Bezug auf den erforderlichen Berechnungsaufwand nicht überzeugend. In einer Übungsaufgabe zu Abschnitt 3.5 werden Sie eine erheblich effizientere Variante implementieren.

1.1.4 Die main()-Methode

Bislang wurde aufgrund der objektorientierten Analyse des Aufgabenbereichs eine Klasse entworfen und in Java realisiert. Daraus lassen sich im Programmablauf selbständige, eigenverantwortliche Objekte erstellen, die etliche Methoden beherrschen und nach Eintreffen entsprechender Nachrichten ausführen.
Wir benötigen nun noch eine Art Skript, das Objekte aus der `Bruch`-Klasse erzeugt und diesen Objekten Befehle zustellt, die (zusammen mit dem Verhalten des Anwenders) den Programmablauf voranbringen. In klassischer Terminologie könnte man hier vom „Hauptprogramm“ reden. Aus objektorientierter Perspektive wird die Hauptrolle jedoch von der `Bruch`-Klasse gespielt, weil große Teile der zum Lösen des Problems benötigten Handlungskompetenz (z.B. Aufgabe erfragen, Bruch kürzen, Lösung präsentieren) in die `Bruch`-Klassendefinition eingegangen sind.
Das angesprochene Skript zur Ablaufsteuerung kann in Java auf unterschiedliche Weise realisiert werden. Eine typische Lösung besteht darin, eine eigene Klasse zu definieren, die aber nicht als Bauplan für Objekte dient, sondern eine so genannte *Klassenmethode* namens **main()** zur Ablaufsteuerung enthält. Wie Sie bald im Detail erfahren, wird ein Java-Programm grundsätzlich über die **main**()-Klassenmethode einer beteiligten Klasse in Gang gesetzt. In unserem Beispiel wäre es durchaus möglich, die `Bruch`-Klasse um eine solche Methode zu erweitern. Indem wir eine andere Klasse zum Starten verwenden, wird u.a. gleich demonstriert, wie leicht das Hauptergebnis unserer Arbeit (die `Bruch`-Klasse) für thematisch verwandte Projekte genutzt werden kann.

Im Beispiel soll die „Starterklasse“ den Namen `BruchRechnung` erhalten. In ihrer **main()**-Methode werden 2 Objekte (Instanzen) aus der Klasse `Bruch` erzeugt und mit dem Ausführen verschiedener Methoden beauftragt:

<table>
<tr><th>Quellcode</th><th>Ein- und Ausgabe</th></tr>
<tr><td>

```
class BruchRechnung {
   public static void main(String[] args)
   {
      Bruch b1 = new Bruch(), b2 = new Bruch();

      System.out.println("1. Bruch");
      b1.frage();
      b1.kuerze();
      b1.zeige();

      System.out.println("\n2. Bruch");
      b2.frage();
      b2.kuerze();
      b2.zeige();

      System.out.println("\nSumme");
      b1.addiere(b2);
      b1.zeige();
   }
}
```

</td><td>

```
1. Bruch
Zaehler: 20
Nenner : 84
    5
  -----
    21


2. Bruch
Zaehler: 12
Nenner : 36
    1
  -----
    3


Summe
    4
  -----
    7
```

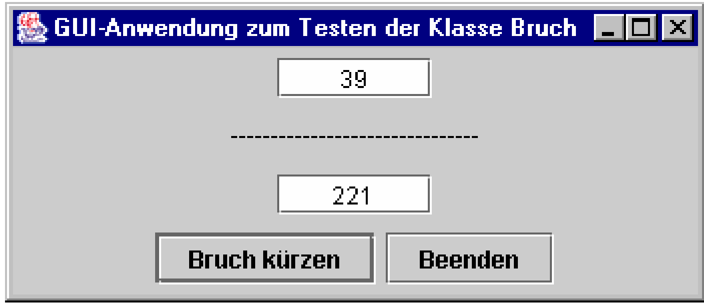
</td></tr>
</table>

Wir haben zur Lösung der Aufgabe zwei Klassen mit folgender Aufgabenverteilung definiert:

- Die Klasse `Bruch` enthält den Bauplan für die wesentlichen Akteure im Aufgabenbereich. Dort alle Eigenschaften und Handlungskompetenzen von Brüchen zu konzentrieren, hat folgende Vorteile:
  - Die Klasse kann in verschiedenen Programmen eingesetzt werden (Wiederverwendbarkeit). Dies fällt vor allem deshalb so leicht, weil die Objekte Handlungskompetenzen (Methoden) besitzen **und** alle erforderlichen Instanzvariablen (Attribute) mitbringen.
  - Beim Umgang mit den `Brüchen` sind wenig Probleme zu erwarten, weil nur klasseneigene Methoden Zugang zu kritischen Attributen haben (Datenkapselung). Sollten doch Fehler auftreten, sind die Ursachen in der Regel schnell identifiziert.
- Die Klasse `BruchRechnung` dient nicht als Bauplan für Objekte, sondern enthält eine Klassenmethode **main()**, die beim Programmstart aufgerufen wird und dann für einen speziellen Einsatz von `Bruch`-Objekten sorgt. Mit einer Wiederverwendbarkeit des `BruchTest`-Quellcodes in anderen Projekten ist kaum zu rechnen.

1.1.5 Ausblick auf Anwendungen mit graphischer Benutzerschnittstelle

Das obige Beispielprogramm arbeitet der Einfachheit halber mit einer konsolen-orientierten Ein- und Ausgabe. Nachdem wir in dieser übersichtlichen Umgebung grundlegende Sprachelemente erarbeitet haben, werden wir uns auch mit der Programmierung von graphischen Benutzerschnittstellen beschäftigen. In folgendem Programm zum Kürzen von Brüchen wird die oben definierte Klasse `Bruch` verwendet, ihre Methoden `frage()` und `zeige()` sind jedoch durch grafikorientierte Varianten ersetzt worden:



Mit dem Quellcode zur Gestaltung der graphischen Oberfläche könnten sie im Moment noch nicht allzu viel anfangen. Am Ende des Kurses werden Sie derartige Anwendungen aber mit Leichtigkeit erstellen.

In diesem Abschnitt sollten Sie einen ersten Eindruck von der Softwareentwicklung mit Java gewinnen. Alle dabei erwähnten Konzepte der objektorientierter Programmierung und technischen Details der Realisierung in Java werden bald systematisch behandelt und sollten Ihnen daher im Moment noch keine Kopfschmerzen bereiten.

1.2 Quellcode, Bytecode und Maschinencode

Eben haben Sie Java als eine Programmiersprache kennen gelernt, die Ausdrucksmittel zur Modellierung von Anwendungsbereichen bzw. zur Formulierung von Algorithmen bereitstellt. Unter einem „Programm“ wurde dabei der vom Entwickler zu formulierende *Quellcode* verstanden.
Während Sie derartige Programme bald mit Leichtigkeit lesen und begreifen werden, kann die CPU eines Rechners nur einen *speziellen* Satz von Befehlen verstehen, die als Folge von Nullen und Einsen (= *Maschinencode*) formuliert werden müssen. Die ebenfalls CPU-spezifische Assemblersprache stellt eine für Menschen lesbare Form des Maschinencodes dar. Die CPU holt sich einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, heutzutage immerhin mehrere hundert Millionen Befehle pro Sekunde.
Ein Quellcode-Programm muss also erst in Maschinencode übersetzt werden, damit es von einem Rechner ausgeführt werden kann. Dies geschieht bei Java aus Gründen der Portabilität und Sicherheit auf eine besondere Weise:

Kompilieren: Quellcode → Bytecode

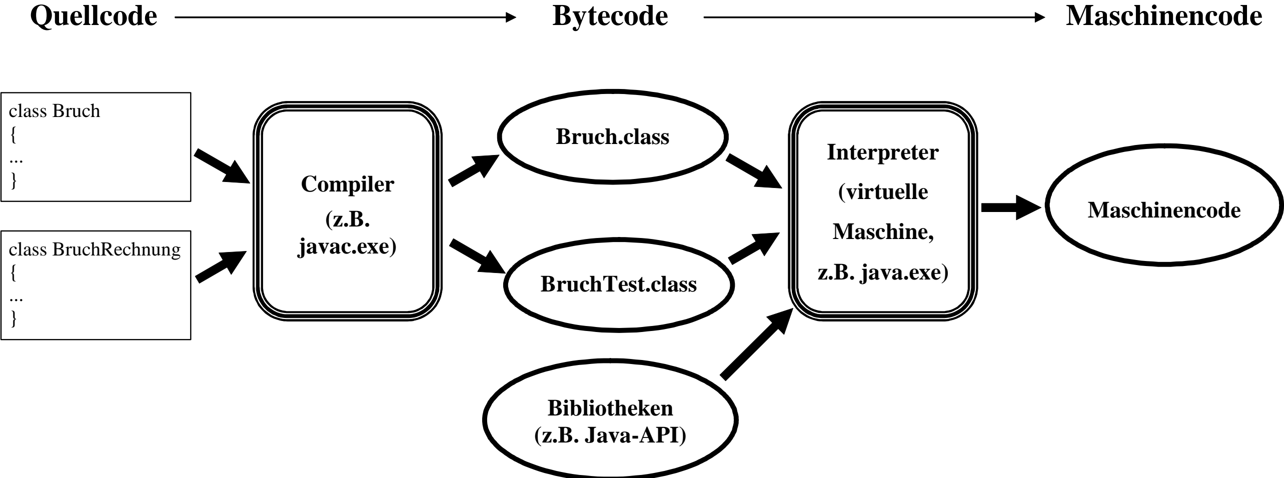
Der (z.B. mit einem beliebigen Editor verfasste) Quellcode wird vom **Compiler** in einen maschinen-unabhängigen **Bytecode** übersetzt. Dieser besteht aus den Befehlen einer von der Firma Sun definierten **virtuellen Maschine**, die sich durch ihren vergleichsweise einfachen Aufbau gut auf aktuelle Hardware-Architekturen abbilden lässt.
Wenngleich der Bytecode von den heute üblichen Prozessoren noch nicht direkt ausgeführt werden kann, hat er doch bereits die meisten Verarbeitungsschritte auf dem Weg vom Quell- zum Maschinencode durchlaufen. Sein Name geht darauf zurück, dass die Instruktionen der virtuellen Maschine jeweils genau ein Byte (= 8 Bit) lang sind.
Weil Bytecode kompakter ist als Maschinencode, eignet er sich gut für die Übertragung via Internet.
Quellcode-Dateien tragen in Java die Namenserweiterung „**.java**“, Bytecode-Dateien die Erweiterung „**.class**“.

Interpretieren: Bytecode → Maschinencode

Für jede Betriebssystem-Plattform mit Java-Unterstützung muss ein (naturgemäß plattformabhängiger) **Interpreter** erstellt werden, der den Bytecode zur Laufzeit in die jeweilige Maschinensprache übersetzt (z.B. das Tool **java** aus dem Java 2 SDK). Dabei findet auch eine Bytecode-**Verifikation** statt, um potentiell gefährliche Aktionen zu verhindern. Die eben erwähnte Bezeichnung *virtuelle*

Maschine (engl.: **J**ava **V**irtual **M**achine, JVM) verwendet man auch für die an der Ausführung von Java-Programmen beteiligte Software. Man benötigt also für jede reale Maschine eine vom jeweiligen Wirtsbetriebssystem abhängige JVM, um den Java-Bytecode plattformunabhängig zu machen. Mittlerweile kommen bei der Ausführung von Java-Programmen leistungssteigernde Techniken (**J**ust-**in**-**T**ime – Compiler, **HotSpot** – Compiler mit Analyse des Laufzeitverhaltens) zum Einsatz, welche die Bezeichnung *Interpreter* fraglich erscheinen lassen. Allerdings ändert sich nichts an der Aufgabe, aus dem plattformunabhängigen Bytecode den zur aktuellen Hardware passenden Maschinencode zu erzeugen. So wird wohl keine Verwirrung gestiftet, wenn in diesem Manuskript weiterhin vom *Interpreter* die Rede ist.

In der folgenden Abbildung sind die Dateinamen zum `Bruch`-Beispiel und außerdem die Namen des Compilers **javac.exe** und des Interpreters **java.exe** aus dem Java 2 Software Development Kit (SDK) eingetragen:



1.3 Java als Programmiersprache und als Klassenbibliothek

Damit die Programmierer nicht das Rad (und ähnliche Dinge) ständig neu erfinden müssen, bietet Java in seinem **API** (**A**pplication **P**rogram **I**nterface) eine große Bibliothek mit fertigen Klassen für nahezu alle Routineaufgaben. Im Abschnitt über Pakete werden die wichtigsten API-Bestandteile grob skizziert. An eine systematische Behandlung ist des enormen Umfangs wegen nicht zu denken.

An dieser Stelle soll geklärt werden, dass die Java-Technologie einerseits auf einer Programmiersprache mit einer bestimmten Syntax und Semantik basiert, dass andererseits aber die Funktionalität im Wesentlichen von einer umfangreichen Standardbibliothek beigesteuert wird, deren Klassen in jeder virtuellen Java-Maschine zur Verfügung stehen.

Die Java-Designer waren bestrebt, sich auf möglichst wenige, elementare Sprachelemente zu beschränken und alle damit bereits formulierbaren Konstrukte in der Standardbibliothek unterzubringen. Es resultierte eine sehr kompakte Sprache (siehe Gosling et al. 2000), die seit ihrer Veröffentlichung im Jahr 1995 kaum geändert werden musste.

Neue Funktionalitäten wurden in der Regel durch eine Erweiterung der Java-Klassenbibliothek realisiert. Hier gab es erhebliche Änderungen, so dass die Firma Sun seit der Bibliotheksversion 1.2 offiziell von der **Java 2** – Plattform spricht.[1]

[1] Nach dieser nicht perfekt intuitiven Bezeichnungsweise arbeiten wir also derzeit mit der Java 2 Plattform, Standard Edition (J2SE), Version 1.4.1.

Einige Klassen sind mittlerweile auch schon als **deprecated** (überholt, nicht mehr zu benutzen) eingestuft worden. Gelegentlich stehen für eine Aufgabe verschiedene Lösungen aus unterschiedlichen Entwicklungsstadien zur Verfügung, was Einsteigern den Umgang mit der Klassenbibliothek erschwert.

Neben der sehr umfangreichen Standard-Klassenbibliothek, die integraler Bestandteil der Java-Plattform ist, sind aus diversen Quellen unzählige Java-Klassen für diverse Problemstellungen verfügbar.

1.4 Zentrale Merkmale der Java-Technologie

Die Programmiersprache Java wurde ab 1990 von einem Team der Firma Sun unter Leitung von James Gosling entwickelt (Gosling et al. 2000). Nachdem erste Pläne zum Einsatz in Geräten aus dem Bereich der Unterhaltungselektronik (z.B. Set-Top-Boxen) wenig Erfolg brachten, orientierte man sich stark am boomenden Internet. Das bisher auf die Darstellung von Texten und Bildern beschränkte WWW (Word Wide Web) wurde um die Möglichkeit bereichert, kleine Java-Programme (*Applets* genannt) von einem Server zu laden und ohne Installation im Fenster des lokalen Browsers auszuführen.
Ein erster Durchbruch gelang 1995, als die Firma Netscape die Java-Technologie in die Version 2.0 ihres WWW-Navigators integrierte. Kurze Zeit später wurden mit der Version 1.0 des Java Development Kits Werkzeuge zum Entwickeln von Java-Applets und –Applikationen frei verfügbar. Mittlerweile hat sich Java als moderne, objektorientierte und für vielfältige Zwecke einsetzbare Programmiersprache etabliert.

Die Java-Entwickler haben sich stark an der Programmiersprache **C++** orientiert, so dass sich Umsteiger von dieser sowohl im Windows- als auch im UNIX-Bereich weit verbreiteten Sprache schnell in Java einarbeiten können. Wesentliche Ziele bei der Weiterentwicklung waren Einfachheit, Robustheit, Sicherheit und Portabilität.
Auf den Aufwand einer systematischen Einordnung von Java im Ensemble der verschiedenen Programmiersprachen bzw. Softwaretechnologien wird hier verzichtet (siehe z.B. RRZN 1999, Goll et al 2000, S. 15). Jedoch sollen wichtige Eigenschaften beschrieben werden, weil sie eventuell relevant sind für die Entscheidung zum Einsatz der Sprache und zur weiteren Teilnahme am Kurs:

Objektorientierung

Java wurde als objektorientierte Sprache konzipiert und erlaubt im Unterschied zu hybriden Sprachen wie C++ und Delphi außerhalb von Klassendefinitionen praktisch keine Anweisungen.
Der objektorientierten Programmierung geht eine objektorientierte *Analyse* voraus, die alle bei einer Problemstellung involvierten Objekt identifizieren möchte. Unter einem Objekt kann man sich grob einen *Akteur* mit *Eigenschaften* (internen, gekapselten Datenelemente) und *Handlungskompetenzen* (Methoden) vorstellen. Auf dem Weg der Abstraktion fasst man identische oder zumindest sehr ähnliche Objekte zu Klassen zusammen.
Java ist sehr gut dazu geeignet, das Ergebnis einer objektorientierten Analyse in ein Programm umzusetzen. Dazu definiert man die beteiligten Klassen und erzeugt aus diesen Prototypen die benötigten Objekte. Deren Interaktion miteinander, mit dem Anwender und mit anderen Systembestandteilen sorgt für den Programmablauf.

In unserem Einleitungsbeispiel wurde einiger Aufwand in Kauf genommen, um einen realistischen Eindruck von objektorientierter Programmierung (OOP) zu vermitteln. Oft trifft man auf Einleitungsbeispiele, die zwar angenehm einfach aufgebaut sind, aber außer gewissen Formalitäten kaum Merkmale der objektorientierten Programmierung aufweisen. In den Abschnitten 2 und 3 werden auch wir solche pseudo-objektorientierten (POO-) Programme benutzen, um elementare Sprachelemente in möglichst einfacher Umgebung kennen zu lernen. Aus den letzten Ausführungen ergibt

sich u.a., dass Java zwar eine objektorientierte Programmierweise nahe legen und unterstützen, aber nicht erzwingen kann.

Portabilität

Die Portabilität von Java resultiert vor dem Hintergrund der in Abschnitt 1.2 beschriebenen Übersetzungsprozedur daraus, dass sich Bytecode-Interpreter relativ gut für aktuelle Rechner-Plattformen implementieren lassen.
Man mag einwenden, dass sich der Quellcode vieler Programmiersprachen (z.B. C++) auf verschiedenen Rechnerplattformen kompilieren lässt. Diese Quellcode-Kompatibilität ist jedoch auf einfache Anwendungen mit textorientierter Benutzerschnittstelle beschränkt und stößt selbst dort auf manche Detailprobleme (z.B. durch verschiedenen Zeichensätze). C++ wird zwar auf vielen verschiedenen Plattformen eingesetzt, doch kommen dabei in der Regel plattformabhängige Funktions- bzw. Klassenbibliotheken zum Einsatz.
Bei Java besitzt hingegen die Standard-Laufzeitumgebung mit ihren insgesamt ca. 3000 Klassen bereits weit reichende Fähigkeiten für die Gestaltung graphischer Benutzerschnittstellen, für Datenbank- und Netzwerkzugriffe usw., so dass plattformunabhängige Anwendungen mit modernem Funktionsumfang und Design realisiert werden können. Einer Java-Anwendung stehen zudem auf allen Plattformen dieselben Zeichensätze zur Verfügung (mit UNICODE-Technologie).

Sicherheit

Auch der Sicherheit dient die oben beschriebene Übersetzungsprozedur, weil ein als Bytecode übergebenes Programm durch den beim Empfänger installierten Interpreter vor der Ausführung recht effektiv auf unerwünschte Aktivitäten geprüft werden kann.

Robustheit

Zur Robustheit von Java trägt u.a. der Verzicht auf Merkmale von C++ bei, die erfahrungsgemäß zu Fehlern verleiten, z.B.:

- Pointerarithmetik
- Überladen von Operatoren
- Mehrfachvererbung

Außerdem wird der Programmierer zu einer gründlichen Fehlerbehandlung gezwungen. Der Compiler **javac** besteht darauf, dass beim Ausführen von Methoden alle bekannten Ausnahmebedingungen abgefangen werden. Von den sonstigen Maßnahmen zur Förderung der Stabilität ist vor allem noch die Feldgrenzen-Überwachung zu erwähnen.

Einfachheit

Schon im Zusammenhang mit der Robustheit wurden einige fehleranfällige C++ - Bestandteile erwähnt, die in Java bewusst weggelassen wurden (explizite Pointer, Mehrfachvererbung, Überladen von Operatoren). Zur Vereinfachung führt auch der Verzicht auf Headerdateien und Präprozessor-Anweisungen.
Wenn man dem Programmierer eine Aufgabe komplett abnimmt, kann er dabei keine Fehler mehr machen. In diesem Sinn wurde in Java der so genannte **Garbage Collector** („Müllsammler“) implementiert, der den Speicher nicht mehr benötigter Objekte automatisch frei gibt. Im Unterschied zu C++, wo die Freigabe durch den Programmierer über den **free**-Operator zu erfolgen hat, sind Speicherlöcher damit ausgeschlossen.
Mittlerweile gibt es auch etliche Java-Entwicklungsumgebungen, die eine graphische Gestaltung von Benutzeroberflächen analog zu Visual Basic oder Delphi erlauben (z.B. Suns’s Forte for Java, Visual Age von IBM, JBuilder von Borland). In diesem Kurs können solche **RAD**-Werkzeuge (**R**apid **A**pplication **D**evelopment) allerdings nicht berücksichtigt werden.

Insgesamt ist Java im Vergleich zu C++ deutlich einfacher zu beherrschen und damit für Einsteiger eher zu empfehlen.

Multithreaded-Architektur

Java unterstützt Anwendungen mit mehreren, parallel laufenden Ausführungsfäden (Threads). Solche Anwendungen bringen erhebliche Vorteile für den Benutzer, der z.B. mit einem Programm interagieren kann, während es im Hintergrund aufwändige Berechnungen ausführt oder auf die Antwort eines Netzwerk-Servers wartet.

Verteilte Anwendungen

Java ist besonders kommunikationsfreudig. Neben den zum Herunterladen via Internet bestimmten Applets gibt es weitere Möglichkeiten, verteilte Anwendungen auf Basis des TCP/IP – Protokolls zu realisieren. Die Kommunikation kann über Sockets, CGI-Aufrufe per URL, **RMI** (**R**emote **M**ethod **I**nvocation) oder **SOAP** (**S**imple **O**bject **A**ccess **P**rotocol) erfolgen. Diese Begriffe müssen Sie übrigens jetzt nicht verstanden haben. Wenn sie später im Kurs relevant werden, folgt eine Erklärung.

Performanz

Der durch Sicherheit (Bytecode-Verifikation), Stabilität (z.B. Garbage Collector) und Portabilität verursachte Performanznachteil von Java-Programmen (z.B. gegenüber C++) ist durch die Entwicklung leistungsfähiger virtueller Java-Maschinen mittlerweile weitgehend irrelevant geworden, wenn es nicht gerade um performanz-kritische Anwendungen (z.B. Spiele) geht.

Dynamisches Laden

Die in einem Java-Programm benutzen Fremdklassen werden von der virtuellen Maschine dynamisch geladen (auch über Netzverbindungen).

1.5 Ausblick auf die Erstellung von Java-Applets

Wir werden uns die Grundlagen der objektorientierten Java-Programmierung im Rahmen von eigenständigen Java-*Anwendungen* erarbeiten. Diese unterscheiden sich in einigen (z.B. sicherheitsrelevanten) Merkmalen von den Java-*Applets*, die durch einen WWW-Browser von einem WWW-Server geholt und dann ausgeführt werden.
Wenn Sie möglichst schnell ein attraktives Java-Applet zur Aufwertung einer WWW-Seite schreiben möchten, müssen Sie also in den ersten Abschnitten dieses Kurses eine Durststrecke überstehen. Damit Sie dabei nicht den Glauben an Ihre Chance zum Erstellen attraktiver Applets verlieren, wollen wir einige Demos genießen, die mit dem Java 2 SDK geliefert werden.
Auf der Startseite der SDK 1.4 - Online-Dokumentation (http://java.sun.com/j2se/1.4.2/docs/) finden Sie den Link

Demonstration Applets and Applications

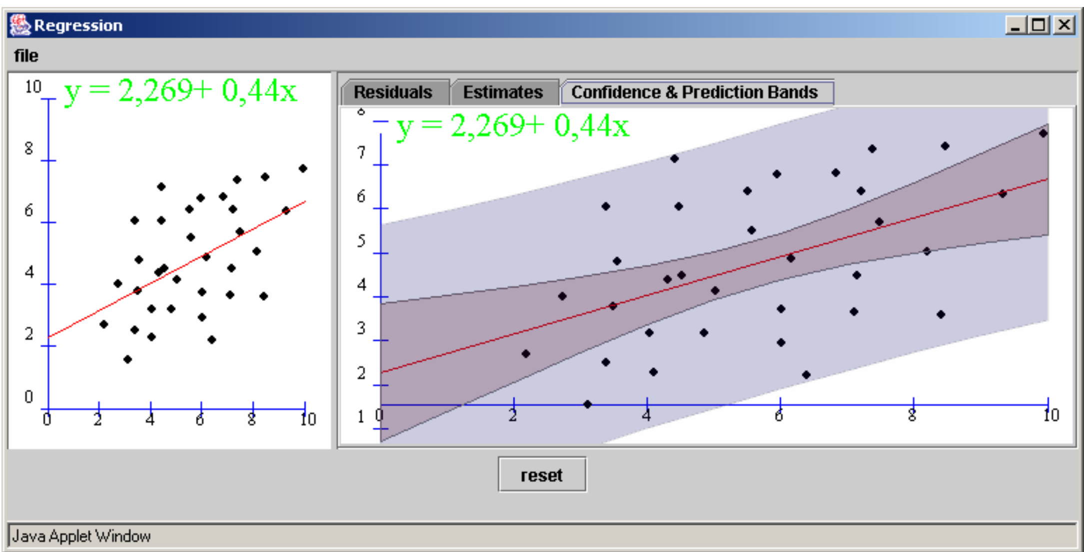
Er führt zu zahlreichen Demo-Applets, die von den aktuellen Browsern ausgeführt werden können, z.B.:

Auch die Java-Quellcodedateien sind verfügbar, so dass dem Lernen durch Nachahmen und Experimentieren nichts im Wege steht.

Wer neben Java auch noch Wahrscheinlichkeitstheorie und Statistik lernen möchte, wird auf folgender Webseite fündig:

http://www.math.csusb.edu/faculty/stanton/m262/index.html

Hier wird demonstriert, wie mit Java interaktive Webseiten (E-Learning!) erstellt werden können, z.B.:



Dieses Applet startet in einem eigenen Fenster.

2 Werkzeuge zum Entwickeln von Java-Programmen

In diesem Abschnitt werden kostenlose Werkzeuge zum Entwickeln von Java-Applikationen bzw. – Applets beschrieben.
Zunächst beschränken wir uns puristisch auf einen Texteditor und das **Software Development Kit (Standard Edition)** der Firma Sun[1]. In dieser sehr übersichtlichen „Entwicklungsumgebung“ werden die grundsätzlichen Arbeitsschritte und einige Randbedingungen besonders deutlich.
Anschließend gönnen wir uns aber doch etwas mehr Luxus in Form der kostenlos verfügbaren Java-Entwicklungsumgebung **JCreator LE**, die auf dem JDK aufsetzt und neben einem guten Editor (z.B. mit Syntaxhervorhebung) noch weitere Arbeitserleichterungen bietet (z.B. einen Klasseninspektor).
Bei größeren Projekten sollte die Verwendung einer aufwändigeren Entwicklungsumgebung (z.B. mit visuellen Werkzeugen zur Gestaltung der Benutzeroberfläche) in Erwägung gezogen werden.

2.1 Java-Entwicklung mit Texteditor und SDK

Für die Betriebsysteme Windows, Linux und Solaris kann das aktuelle Java 2 SDK samt Installationsanleitung und Dokumentation über folgende Webseite bezogen werden:

http://java.sun.com/j2se/

2.1.1 Editieren

Wir verfassen den Quellcode mit einem beliebigen Texteditor, unter Windows z.B. mit **Notepad** (**Editor**). Um das Erstellen, Compilieren und Ausführen von Java-Programmen ohne großen Aufwand üben zu können, erstellen wir das unvermeidliche **Hallo**-Programm, das vom oben beschriebenen POO-Typ ist (pseudo-objektorientiert):

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Hallo {
   public static void main(String[] args) {
       System.out.println("Hallo Allerseits!");
   }
}
```

</td><td>

```
Hallo Allerseits!
```

</td></tr>
</table>

Im Unterschied zu *hybriden* Programmiersprachen wie C++ und Delphi, die neben der objektorientierten auch die rein prozedurale Programmiertechnik unterstützen, verlangt Java auch für solche Trivialprogramme eine Klassendefinition.

Das POO-Programm ist zwar nicht „vorbildlich“, eignet sich aber aufgrund seiner Kürze zum Erläutern wichtiger Regeln, an die Sie sich so langsam gewöhnen müssen. Alle Themen werden aber später noch einmal systematischer und ausführlicher behandelt:

- Nach dem Schlüsselwort **class** folgt der frei wählbare Klassenname. Hier ist wie bei allen Bezeichnern zu beachten, dass Java streng zwischen Groß- und Kleinbuchstaben unterscheidet.
- Dem Kopf der Klassendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf.

[1] Mit der Version 1.2 hat Sun die ursprüngliche Bezeichnung **JDK** (**J**ava **D**evelopment **K**it) durch **SDK** (**S**oftware **D**evelopment **K**it) ersetzt. Gleichzeitig wurde der gesamten Java-Plattform die Generationsbezeichnung 2 verliehen. Nach dieser nicht ganz einfachen Sprachregelung liefert also Sun im Moment für seine Programmiersprache Java 2 das Entwicklungspaket SDK 1.4.x aus.

- Weil die `Hallo`-Klasse startfähig sein soll, muss sie eine Methode namens **main()** besitzen. Diese wird vom Interpreter beim Programmstart ausgeführt und dient in der Regel dazu, Objekte zu erzeugen.
- Die Definition der Methode **main()** wird von drei Schlüsselwörtern eingeleitet, deren Bedeutung für Neugierige hier schon erwähnt wird:
  - **public**
    Wie eben erwähnt, wird die Methode **main()** beim Programmstart vom Interpreter gesucht und ausgeführt. Weil es sich beim Interpreter aus Sicht des Programms um einen *externen* Akteur handelt, muss (zumindest ab SDK 1.4.x) die Methode **main()** explizit über den Modifikator **public** für die Öffentlichkeit frei gegeben werden.
    Bei aufmerksamer Lektüre der (z.B. im Internet) zahlreich vorhandenen Java-Beschreibungen stellt man fest, dass viele Autoren nicht nur die Methode **main()**, sondern auch die zugehörige *Klasse* als **public** definieren, z.B.:

```
public class Hallo {
  public static void main(String[] args) {
    System.out.println("Hallo Allerseits!");
  }
}
```

    Diese Praxis erscheint zwar plausibel und systematisch, wird aber vom Java-Compiler bzw. –Interpreter *nicht* gefordert und stellt daher eine vermeidbare Mühe dar. Bei der Wahl einer Regel für dieses Manuskript habe ich mich am Verhalten der Java-Urheber orientiert: Gosling et at. (2000) lassen bei ausführbaren Klassen den Modifikator **public** systematisch weg. Wir werden später klare und unvermeidbare Gründe für die Verwendung des Klassen-Modifikators **public** kennen lernen.
  - **static**
    Mit diesem Modifikator wird **main()** als **Klassenmethode** gekennzeichnet. Im Unterschied zu den *Instanz*methoden der Objekte gehören die Klassenmethoden, oft auch als *statische Methoden* bezeichnet, zur Klasse und können ohne vorherige Objekt-Kreation ausgeführt werden.
    Die beim Programmstart automatisch ausgeführte **main()**–Methode der Startklasse muss auf jeden Fall durch den Modifikator **static** als Klassenmethode gekennzeichnet werden. In einem objektorientierten Programm hat sie insbesondere die Aufgabe, die ersten Objekte zu erzeugen (siehe unsere Klasse `Bruchrechnung` auf Seite 6).
  - **void**
    Die Methode **main()** erhält den Typ **void**, weil sie keinen Rückgabewert liefert.
- In der Parameterliste einer Methode kann die gewünschte Arbeitsweise näher spezifiziert werden. Wir werden uns später ausführlich mit diesem wichtigen Thema beschäftigen und beschränken uns hier auf zwei Hinweise:
  - *Für Neugierige und/oder Vorgebildete*
    Der **main()**-Methode werden über ein Feld mit Stringelementen die Spezifikationen übergeben, die der Anwender in der Kommandozeile beim Programmstart angegeben hat. In unserem Beispiel kümmert sich die Methode **main()** allerdings nicht um solche Anwenderwünsche.
  - *Für Alle*
    Bei der **main()**-Definition ist die im Beispiel verwendete Parameterliste obligatorisch, weil der Interpreter ansonsten die Methode beim Programmstart nicht erkennt und sich ungefähr so äußert:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

    Den Parameter*namen* (im Beispiel: `args`) darf man allerdings beliebig wählen.
- Dem Kopf einer Methodendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf mit Variablendeklarationen und Anweisungen.

- In der **main**()-Methode unserer `Hallo`-Klasse wird die **println**()-Methode des vordefinierten Objektes **System.out** dazu benutzt, einen Text an die Standardausgabe zu senden. Zwischen dem Objekt- und dem Methodennamen steht ein Punkt. Bei dem Methodenaufruf handelt sich um eine Anweisung, die folglich mit einem Semikolon abzuschließen ist.

Es dient der Übersichtlichkeit, zusammengehörige Programmteile durch eine gemeinsame Einrücktiefe zu kennzeichnen. Man realisiert die Einrückungen am einfachsten mit der Tabulatortaste, aber auch Leerzeichen sind erlaubt. Für den Compiler sind die Einrückungen irrelevant.

Speichen Sie Ihr Quellprogramm unter dem Namen `Hallo.java` in einem geeigneten Verzeichnis, z.B. in

U:\Eigene Dateien\Java\Kurs\Hallo

Beachten Sie bitte:

- Der Dateinamensstamm (vor dem letzten Punkt) sollte mit dem Klassennamen übereinstimmen.
- Die Dateinamenserweiterung muss unbedingt **.java** lauten.
- Unter Windows ist beim Dateinamen die Groß-/Kleinschreibung zwar irrelevant, doch sollte auch hier aus ästhetischen Gründen und im Hinblick auf eine mögliche Weitergabe der Quellcodedatei auf Konsistenz mit dem Klassennamen geachtet werden.

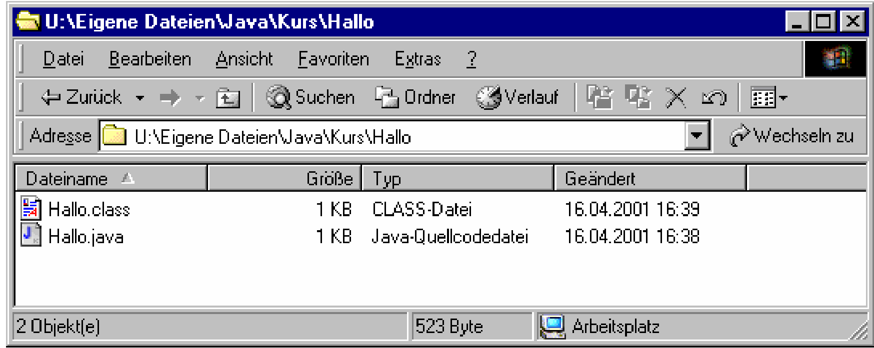
2.1.2 Kompilieren

Öffnen Sie ein Konsolenfenster, und wechseln Sie in das Verzeichnis mit dem neu erstellten Quellprogramm `Hallo.java`.
Lassen Sie das Programm vom SDK-Compiler **javac** übersetzen:

```
javac Hallo.java
```

Damit dieser Aufruf von jedem Verzeichnis aus klappt, muss nach der Java 2 SDK-Installation das **bin**-Unterverzeichnis (mit dem Compiler **javac**) in die Definition der Umgebungsvariablen PATH- aufgenommen werden.

Falls keine Probleme auftreten, meldet sich der Rechner nach kurzer Bedenkzeit mit einem neuen Kommando-Prompt zurück, und die Quellcodedatei `Hallo.java` erhält Gesellschaft durch die Bytecodedatei `Hallo.class`, z.B.:



Beim Kompilieren einer Quellcodedatei werden auch alle darin benutzten Klassen neu übersetzt, falls deren Bytecodedatei fehlt oder älter als die zugehörige Quellcodedatei ist. Sind etwa im Bruchrechnungsbeispiel die Quellcodedateien **Bruch.java** *und* **BruchRechnung.java** geändert worden, dann genügt folgender Compileraufruf, um beide neu zu übersetzen:

```
javac BruchRechnung.java
```

2.1.3 Ausführen

Lassen Sie das Programm (bzw. die Klasse) `Hallo.class` vom SDK-Interpreter ausführen:

```
java Hallo
```

Bitte beachten:

- Damit dieser Aufruf von jedem Verzeichnis aus klappt, muss nach der Java 2 SDK-Installation das **bin**-Unterverzeichnis (mit dem Interpreter **java**) in die Definition der Umgebungsvariablen PATH- aufgenommen werden.
- Die Namenserweiterung „class" wird **nicht** angegeben. Wer's doch tut, erhält keinen Fleißpunkt, sondern eine Fehlermeldung:
  ```
  U:\Eigene Dateien\Java\Kurs\Hallo>java Hallo.class
  Exception in thread "main" java.lang.NoClassDefFoundError: Hallo/class
  ```
- Beim Aufruf des Interpreters wird der Name der auszuführenden Klasse als Parameter angegeben. Weil es sich dabei um einen Java-Bezeichner handelt, muss auch die Groß-/Kleinschreibung mit der Klassendeklaration (in der Datei `Hallo.java`) übereinstimmen (auch unter Windows!).
- Die Version des installierten Interpreters kann mit folgenden Kommando ermittelt werden:
  ```
  U:\Eigene Dateien\Java\Kurs\Hallo>java -version
  java version "1.4.1"
  Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1-b21)
  Java HotSpot(TM) Client VM (build 1.4.1-b21, mixed mode)
  ```
  Wir erfahren, dass Suns **HotSpot - Client VM** (Virtual Maschine) im Einsatz ist. Sie stellt sich beim Übersetzen des Java-Bytecodes in Maschinencode besonders geschickt an (Analyse des Laufzeitverhaltens) und ist älteren JVMs (Java Virtual Maschine), die mit einem **JIT**-Compiler (**J**ust-**i**n-**T**ime) arbeiten, deutlich überlegen sein.

Zum *Ausführen* eines Java-Programms wird *nicht* das vollständige Java 2 SDK benötigt, sondern lediglich die virtuelle Java-Maschine (Java VM) mit dem Interpreter **java.exe.** Dieses - auch als **Java Runtime Environment** (JRE) bezeichnete - Produkt ist bei Sun separat erhältlich (http://java.sun.com/getjava/de/) und darf weiter gegeben werden, um eigene Java-Programme auf fremden Rechnern lauffähig zu machen. Das Java 2 SDK enthält natürlich auch die Laufzeitumgebung (im Verzeichnis **...\jre**), so dass diese nicht zusätzlich installiert werden muss.

Das beim Einsatz von **java.exe** unvermeidliche Konsolenfenster kann beim Starten einer Java-Anwendung mit graphischer Benutzeroberfläche störend wirken. Unter MS-Windows bietet sich als Alternative der Starter **javaw.exe** an, der auf ein Konsolenfenster verzichtet. Trägt man z.B. das folgende Kommando zum Starten des in Abschnitt 1.1 vorgestellten graphischen Bruchrechnungsprogramms als Ziel einer Verknüpfungsdatei ein, dann ist das Programm anschließend per Doppelklick zu starten (ohne Auftritt eines Konsolenfensters):

```
javaw BruchRechnungGui
```

2.1.4 Pfad für class-Dateien setzen

Compiler und Interpreter benötigen Zugriff auf die Bytecodedateien zu allen Klassen, die im zu übersetzenden Quellcode bzw. im auszuführenden Programm angesprochen werden. Mit Hilfe der Umgebungsvariablen CLASSPATH kann man eine Liste von Pfaden spezifizieren, die durchsucht werden sollen, z.B.:

```
C:\User>echo %classpath%
.;C:\WINNT\java\classes;U:\Eigene Dateien\Java\class
```

Befinden sich alle benötigten Klassen entweder in einer Java-API-Bibliothek (z.B. **...\jre\lib\rt.jar**) oder im aktuellen Verzeichnis, dann wird ab der SDK-Version 1.2 keine CLASSPATH-Umgebungsvariable benötigt. Ist sie jedoch vorhanden (z.B. von irgend einem Installationsprogramm unbemerkt angelegt), dann werden außer den API-Bibliotheken nur die angegebenen Pfade berücksichtigt. Dies führt oft zu Problemen bei CLASSPATH-Variablen, die das aktuelle Verzeichnis nicht enthalten, z.B.:

```
U:\Eigene Dateien\Java\Hallo>java Hallo
Exception in thread "main" java.lang.NoClassDefFoundError: Hallo
```

In diesem Fall muss das aktuelle Verzeichnis (z.B. dargestellt durch einen einzelnen Punkt, s.o.) in die CLASSPATH-Pfadliste aufgenommen werden.

Wenn sich nicht alle benötigten **class**-Dateien im aktuellen Verzeichnis befinden, und auch nicht auf die CLASSPATH-Variable vertraut werden soll, können die nach **class**-Dateien zu durchsuchenden Pfade auch in den Startkommandos für Compiler und Interpreter über die **classpath**-Option angegeben werden, z.B.:

```
javac -classpath ".;U:\Eigene Dateien\Java\Class" BruchRechnung.java
javaw -classpath "U:\Eigene Dateien\Java\Bruch\gui" BruchRechnungGui
```

Auch hier muss der aktuelle Pfad ausdrücklich aufgelistet werden, wenn er in die Suche einbezogen werden soll.
Ein Vorteil der Kommandozeilenoption gegenüber der Umgebungsvariablen CLASSPATH besteht darin, dass für jede Anwendung eine eigene Pfadliste eingestellt werden kann.
Mit dem Verwenden der Kommandozeilenoption wird eine eventuell vorhandene CLASSPATH-Umgebungsvariable für den gestarteten Compiler- oder Interpreterlauf deaktiviert.

2.1.5 Programmfehler beheben

Die vielfältigen Fehler, die wir mit naturgesetzlicher Unvermeidlichkeit beim Programmieren machen, kann man einteilen in:

- **Syntaxfehler**
  Diese verstoßen gegen eine Syntaxregel der verwendeten Programmiersprache, werden vom Compiler gemeldet und sind daher relativ leicht zu beseitigen.
- **Semantikfehler**
  Hier liegt kein Syntaxfehler vor, aber das Programm verhält sich anders als erwartet.

Die Java-Entwickler haben dafür gesorgt (z.B. durch strenge Typisierung, Beschränkung der impliziten Typanpassung, Zwang zur Behandlung von Ausnahmen), dass möglichst viele Fehler vom Compiler aufgedeckt werden können.
Wir wollen am Beispiel eines provozierten Syntax-Fehlers überprüfen, ob der SDK-Compiler hilfreiche Fehlermeldungen produziert. Wenn im `Hallo`-Programm der Bezeichner **System** fälschlicherweise mit kleinem Anfangsbuchstaben geschrieben wird, meldet der Compiler:

```
U:\Eigene Dateien\Java\Hallo>javac Hallo.java
Hallo.java:5: package system does not exist
                 system.out.print("Hallo Allerseits!");
                       ^
1 error
```

Der Compiler hat die fehlerhafte Stelle sehr gut lokalisiert:

- Die Fehlermeldung wird eingeleitet mit dem Fundort: Datei **Hallo.java**, Zeile 5
- Der Compiler listet die betroffene Zeile und markiert, an welcher Stelle er damit Probleme hat.

Sie können sich allerdings nicht darauf verlassen, dass die Lokalisation stets so perfekt klappt und müssen eventuell auch die vorhergehenden Zeilen inspizieren.
Auch die Fehler*beschreibung* fällt im Beispiel ziemlich eindeutig aus. Nach Erfahrungen mit Programmiersprachen wie Visual Basic oder Delphi müssen Sie sich eventuell noch daran gewöhnen, dass in Java die Groß-/Kleinschreibung signifikant ist.

Zum Abschluss wollen wir noch mit einem Fehler der zweiten Art experimentieren und im Kopf der **main**()-Methodendefinition den Modifikator **static** streichen, der Klassenmethoden kennzeichnet (vgl. Abschnitt 2.1.1). Während der *Compiler* am veränderten Programm nichts zu bemängeln hat, klagt der *Interpreter*:

```
U:\Eigene Dateien\Java\Hallo>java Hallo
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Ein Verständnis der Fehlermeldung `NoSuchMethodError` setzt schon etwas mehr Java - Know How voraus: Die Methode **main**() ist zwar vorhanden, allerdings mangels **static**-Deklaration für den Interpreter „unsichtbar“.
Ansonsten ist der Fehler relativ harmlos, weil er die Ausführung des Programms verhindert. Übler sind Fehler, die zu Abstürzen oder falschen Ergebnissen führen.

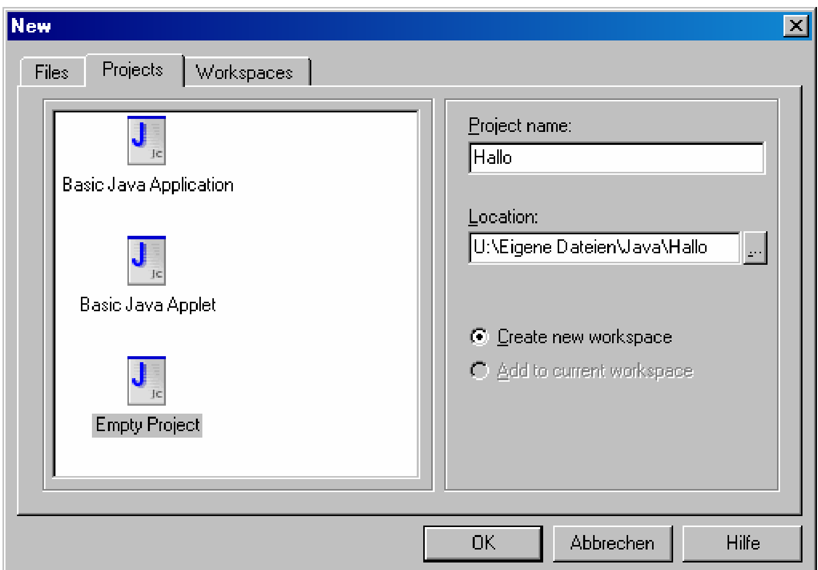
2.2 Java-Entwicklung mit dem JCreator LE 2.x

Auf die Dauer ist das Hantieren mit Editor und Kommandozeile beim Entwickeln von Java-Software recht umständlich, zumal bei Editoren ohne spezielle Unterstützung für Java-Programmierer (z.B. durch Syntaxhervorhebung, intelligentes Einrücken).
Mit der kostenlosen Java-Entwicklungsumgebung **JCreator LE 2.x** der Firma Xinox Software können wir uns das Leben erheblich erleichtern. Das schlanke, unproblematisch zu bedienende und sehr flink agierende Programm ist über folgende Webseite zu beziehen:

http://www.jcreator.com/

JCreator LE setzt Suns Java 2 SDK voraus, das daher zuerst installiert werden sollte.

Um das `Hallo`-Programm per JCreator zu erstellen, legt man nach **File > New** auf der Registerkarte **Projects** der Dialogbox **New** ein neues Projekt an:



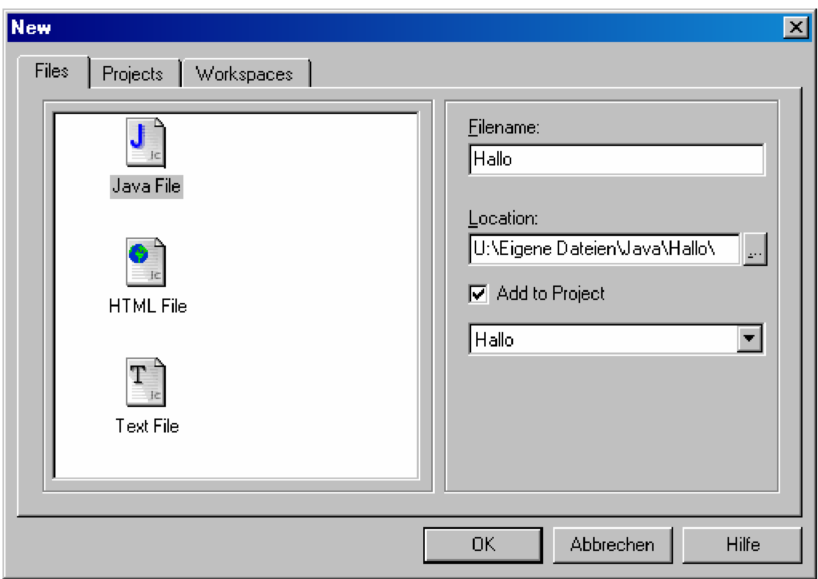
Von den angebotenen Projekttypen ist für uns **Empty Project** am besten geeignet:

- Wir wollen kein Applet, sondern eine Applikation erstellen.
- Die Option **Basic Java Application** liefert den Quellcode für eine elementare GUI-Anwendung, woran wir im Moment noch nicht interessiert sind.

Bei der in diesem Kurs vorgesehenen Arbeitsweise ist beim Anlegen eines neuen Projektes die Option **Create new workspace** stets gegenüber der eventuell verfügbaren Alternative **Add to current workspace** zu bevorzugen.
In dem unter **Location** genannten Pfad wird ein Unterordner mit dem angegebenen Projektnamen angelegt. Dort landen neben der Quellcode- und der Bytecodedatei auch die vom JCreator zur Projektverwaltung verwendeten Dateien:

- **jcp**-Datei (**J**Creator **P**roject).
  Enthält Informationen über ein Projekt (z.B. eine Anwendung oder ein Paket). Das Projekt kann per Doppelklick auf die **jcp**-Datei geöffnet werden. Es ist allerdings sinnvoller, ein JCreator - Projekt über den zugehörigen Arbeitsbereich zu öffnen:
- **jcw**-Datei (**J**Creator **W**orkspace).
  Jedes Projekt gehört zu einem Arbeitsbereich, dessen Eigenschaften in einer **jcw**-Datei verwaltet werden. Grundsätzlich kann ein Arbeitsbereich mehrere Projekte enthalten. Weil wir davon keinen Gebrauch machen werden, können Sie die Begriff *Projekt* und *Arbeitsbereich* als äquivalent betrachten.
  Um die Arbeit an einem vorhandenen Projekt fortzusetzen, öffnet man am besten den zugehörigen Arbeitsbereich, z.B. per Doppelklick auf seine **jcw**-Datei.

Wir quittieren die **New**-Dialogbox und öffnen sie dann erneut (per **File > New**), um auf der Registerkarte **Files** eine leere Quellcodedatei anzulegen:



Als **Filename** sollte hier der geplante Klassenname eingetragen werden. Nach dem Quittieren der Dialogbox steht ein Editor-Fenster zur Verfügung:

Hallo.java *
```
1 class Hallo {
2     public static void main(String[] args) {
3         System.out.print("Hallo Allerseits!");
4     }
5 }
```

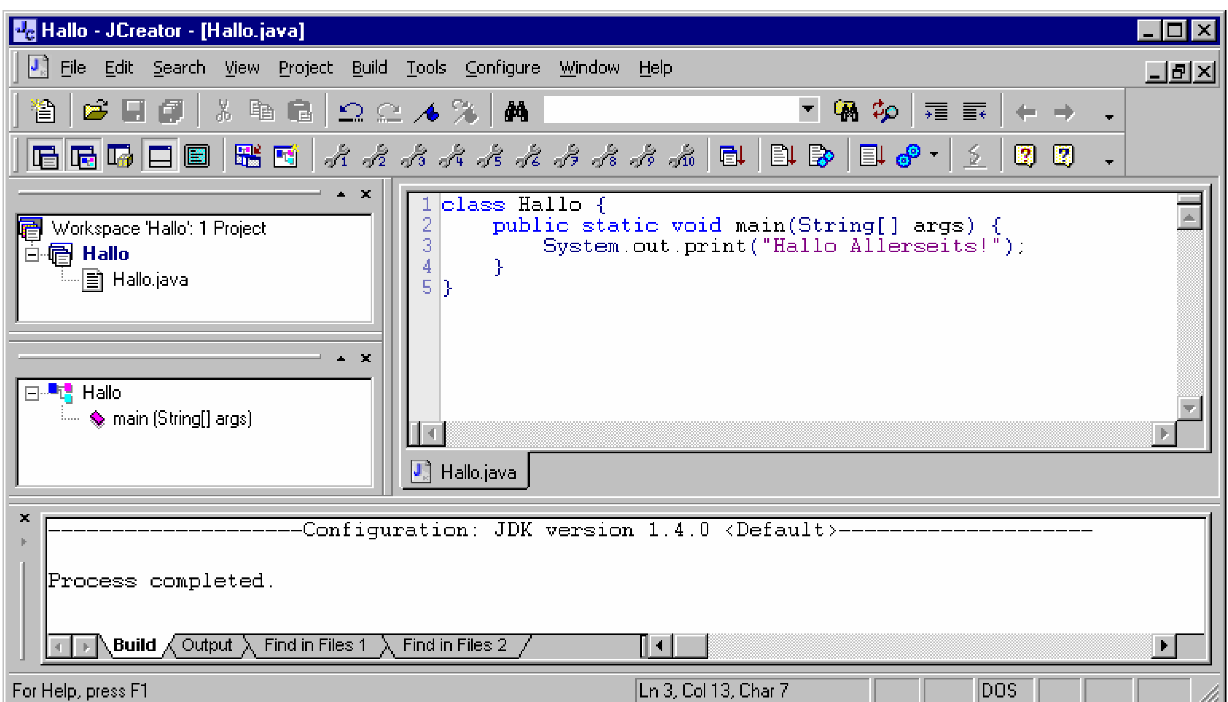
Welche Vorteile der JCreator-Editor im Vergleich zum Notepad bietet, wird bereits beim Eintippen des `Hallo`-Quellcodes deutlich:

- Verschiedene Syntaxbestandteile werden farblich hervorgehoben, fest definierte Schlüsselwörter z.B. in Blau. Nach kurzer Eingewöhnungszeit werden Sie an der fehlenden Blaufärbung sofort erkennen, dass Ihnen beim Eintippen eines Schlüsselwortes (z.B. **class**) ein Fehler unterlaufen sein muss.
- Befindet sich die Schreibmarke auf oder neben einer Klammer, so wird diese *samt Gegenstück* farblich markiert. Damit sind z.B. fehlende oder überzählige Blockklammern leichter auszumachen.
- Beim Einfügen neuer Zeilen wird die Einrückebene meist korrekt gewählt.

Neben dem Editorbereich, der auch mit *mehreren* Dateien umgehen kann, bietet der Arbeitsplatz noch weitere Fenster:

- **Dateiansicht** (bei Bedarf mit **View > Toolbars > FileView** oder zu öffnen)
- **Klassenansicht** (bei Bedarf mit **View > Toolbars > ClassView** oder zu öffnen)
  Hier erscheinen die Eigenschaften und Methoden der im aktuellen Editorfenster definierten Klassen.
- **Paketansicht** (bei Bedarf mit **View > Toolbars > PackageView** oder zu öffnen)
  Funktional verwandte Klassen fasst man zu Paketen zusammen, die auch eine wichtige Rolle bei der Verwaltung von Zugriffsrechten spielen. In unserem Beispiel wird die Klasse `Hallo` dem automatisch definierten Paket **Default** zugerechnet. Wir werden uns später ausführlich mit Paketen beschäftigen.
- **Ausgabe** (bei Bedarf mit **View > Toolbars > Output** oder zu öffnen)
  Hier erscheinen u.a. die Fehlermeldungen des Compilers. Nach einem Doppelklick auf die erste Zeile einer Compiler-Fehlermeldung (mit der Ortsangabe) wird übrigens im Editor sofort die betroffenen Anweisung markiert.

Nach dem Kompilieren der `Hallo`-Klasse zeigt sich folgendes Bild:



Zum Kompilieren und Ausführen von Programmen bietet der JCreator u.a. folgende Kommandos:

- **Build > Compile File** oder übersetzt die Quelle im aktiven Editorfenster.
- **Build > Compile Project** oder oder **F7** übersetzt alle Quellen des Projektes.
- **Build > Execute Project** oder oder **F5** führt das Projekt aus.

Nach einem Compiler-Lauf finden sich im `Hallo`-Projektordner folgende Dateien:

U:\Eigene Dateien\Java\Hallo

Dateiname	Größe	Typ	Geändert	Attribute
Hallo.class	1 KB	CLASS-Datei	29.04.2003 01:02	A
Hallo.java	1 KB	Java Source File	29.04.2003 00:45	A
Hallo.jcp	1 KB	JCreator Project File	29.04.2003 00:08	A
Hallo.jcw	1 KB	JCreator Workspace...	29.04.2003 00:08	A
src_hallo.txt	1 KB	Textdatei	29.04.2003 01:02	A

5 Objekt(e) 1,29 KB Arbeitsplatz

2.3 Übungsaufgaben zu Abschnitt 2

1) Experimentieren Sie mit dem `Hallo`-Beispielprogramm, z.B. indem Sie weitere Ausgabeanweisungen ergänzen.

2) Beseitigen Sie die Fehler in folgender Variante des `Hallo`-Programms:

```
class Hallo {
    static void mein(String[] args) {
        System.out.println("Hallo Allerseits!);
    }
```

3) Welche von den folgenden Aussagen sind **falsch**?

- Beim Starten eines Java-Programms muss man den Namen der auszuführenden Klasse samt Extension angeben (**.class**).
- Beim Übersetzen einer Java-Klasse muss man den Dateinamen samt Extension angeben (**.java**).
- Damit der Aufruf von **javac.exe** von jedem Verzeichnis aus klappt, muss das **bin**-Unterverzeichnis der Java 2 SDK - Installation in die Definition der Umgebungsvariablen PATH aufgenommen werden.
- Damit der Java 1.4-Compiler die API-Klassen findet, muss deren Pfad in die Umgebungsvariable CLASSPATH eingetragen werden.

3 Elementare Sprachelemente

3.1 Einstieg

3.1.1 Aufbau einer Java-Applikation

Bevor wir im Rahmen von möglichst einfachen Beispiel-Programmen elementare Sprachelemente kennen lernen, soll unser bisheriges Wissen über die Struktur von Java-Programmen[1] zusammengefasst werden:

- Ein Java-Programm besteht aus **Klassen**.
  Meist verwendet man für den Quellcode einer Klasse jeweils eine eigene Datei. Der Compiler erzeugt auf jeden Fall für jede Klasse eine eigene Bytecodedatei.
  Unser Bruchrechnungs-Beispiel in Abschnitt 1.1 besteht aus den beiden Klassen `Bruch` und `BruchRechnung`.
- Von den Klassen eines Programms muss mindestens eine **startfähig** sein.
  Dazu benötigt sie eine **Methode** mit dem Namen **main,** dem Rückgabetyp **void**, einer bestimmten Parameterliste **(String[] args)** und den Modifikatoren **public** und **static**. Diese wird beim Starten der Klasse vom Interpreter ausgeführt. Daher muss sie der Öffentlichkeit zugänglich (Modifikator **public**) und als Klassenmethode unabhängig von der Existenz konkreter Objekte ausführbar sein (Modifikator **static**). Auch die Startklasse selbst muss öffentlich sichtbar sein, was aber auch ohne explizite Verwendung des Modifikators **public** der Fall ist.
  Beim Bruchrechnungs-Beispiel in Abschnitt 1.1 ist die Klasse `BruchRechnung` startfähig.
  In den POO-Beispielen von Abschnitt 3 existiert jeweils nur eine Klasse, die infolgedessen startfähig sein muss.
- Sowohl die Klassen- als die Methodendefinitionen bestehen aus:
  - **Kopf**
    In Abschnitt 2.1.1 finden sich im Zusammenhang mit dem `Hallo`-Beispiel nähere Erläuterungen.
  - **Rumpf**
    Dieser besteht aus einem Block mit beliebig vielen Anweisungen, mit denen z.B. Variablen definiert oder verändert werden. Ein Klassen- oder Methodenrumpf wird durch geschweifte Klammern begrenzt.
- Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms. In Java sind bis auf wenige Ausnahmen alle Anweisungen mit einem **Semikolon** abzuschließen.
- Zur **Formatierung** von Java-Programmen haben sich Konventionen entwickelt, die wir bei passender Gelegenheit besprechen werden. Der Compiler ist hinsichtlich der Formatierung sehr tolerant und beschränkt sich auf folgende Regeln:
  - Die einzelnen Bestandteile einer Definition oder Anweisung müssen in der richtigen Reihenfolge stehen.
  - Zwischen zwei Sprachbestandteilen muss im Prinzip ein Trennzeichen stehen, wobei das Leerzeichen, das Tabulatorzeichen und der Zeilenumbruch erlaubt sind. Diese Trennzeichen dürfen sogar in beliebigen Anzahlen und Kombinationen auftreten. Zeichen mit festgelegter Bedeutung wie z.B. ";", "(", "+", ">" sind *selbstbegrenzend*, d.h. vor und nach ihnen sind keine Trennzeichen nötig (aber erlaubt).

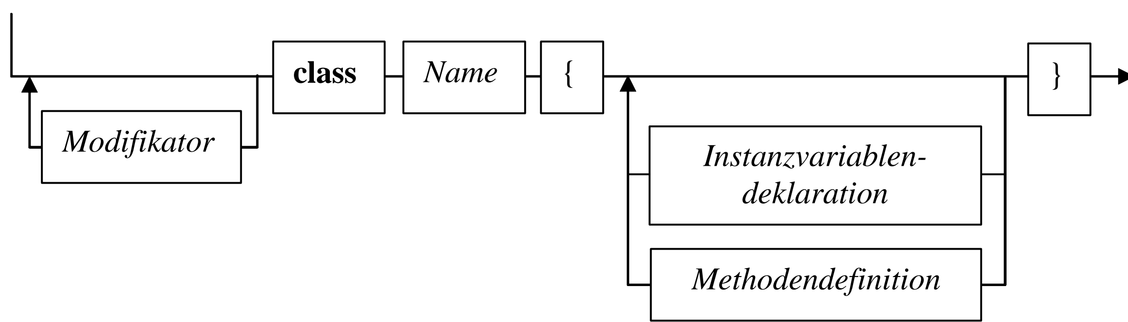
[1] Hier ist ausdrücklich von Java-*Programmen* (alias -*Applikationen*) die Rede. Bei den später vorzustellenden Java-*Applets* ergeben sich einige Abweichungen.

Um für eine Java-Definition oder -Anweisung die erlaubten Bildungsvorschriften genau zu beschreiben, werden wir im Kurs u.a. so genannte **Syntaxdiagramme** einsetzen, für die folgende Vereinbarungen gelten:

- Für **terminale Sprachbestandteile**, die aus einem Syntaxdiagramm exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind durch *kursive* Schrift gekennzeichnet.

In den Syntaxdiagrammen zur Klassen- und zur Methodendefinition tauchen einige Bestandteile auf, die noch nicht systematisch beschrieben wurden (z.B. Modifikator, Variablen- und Parameterdeklaration), so dass Sie bei Verständnisproblemen nicht allzu lange grübeln sollten:

Klassendefinition



In unserer `Bruch`-Klasse wurden die Instanzvariablen `zaehler` und `nenner` sowie die Methoden `setzeZaehler()`, `setzeNenner()`, `gibZaehler()`, `gibNenner()`, `kuerze()`, `addiere()`, `frage()` und `zeige()` definiert.

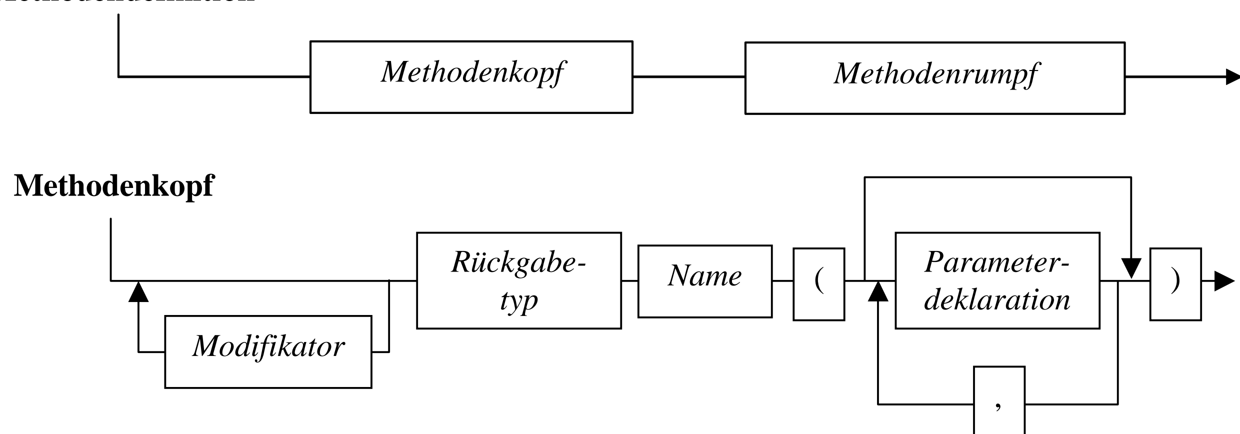
Ob man die öffnende geschweifte Klammer an das Ende der Kopfzeile setzt oder an den Anfang der nächsten Zeile, ist Geschmacksache:

```
class Hallo {
   public static void main(String[] par) {
        System.out.print("Hallo");
   }
}
```

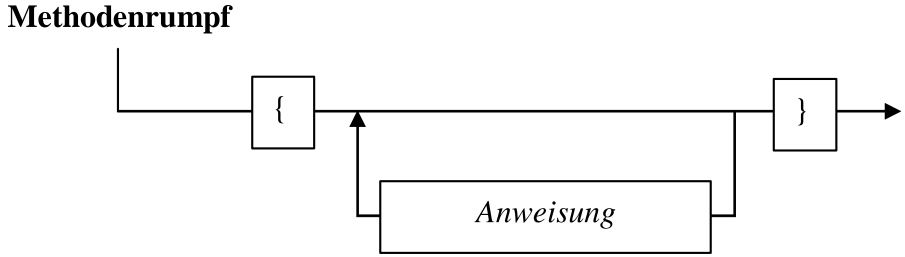
```
class Hallo
{
   public static void main(String[] par)
   {
        System.out.print("Hallo");
   }
}
```

Dies gilt analog auch für die Methodendefinition:

Methodendefinition



Methodenrumpf



Während der Beschäftigung mit elementaren Java-Sprachelementen werden wir der Einfachheit halber mit einer relativ untypischen, jedenfalls nicht wirklich objektorientierten Programmstruktur arbeiten, die Sie schon aus dem `Hallo`-Beispiel kennen. Es ist zwar eine Klassendefinition vorhanden, doch diese dient nur als Hülle für eine „Hauptfunktion“ (im Sinne der altehrwürdigen Programmiersprache C), die in der Klassenmethode **main**() untergebracht wird. Es werden keine weiteren Methoden definiert und auch keine Objekte erzeugt, allerdings durchaus vordefinierte Objekte benutzt (z.B. **System.out** in **Hallo.java**).
Bei den Demonstrations- bzw. Übungsprogrammen in Abschnitt 3 werden wir also folgende Programmstruktur benutzen:

```
class Prog {
    public static void main(String[] args) {
        //Platz für elementare Sprachelemente
    }
}
```

Manche Autoren sind der Meinung, dass derartige Programme, die wir in Abschnitt 1.4 als *pseudo-objektorientiert* bezeichnet haben, keinen optimalen Einstieg in die objektorientierte Programmierung darstellen. Ein erlaubtes Hilfsmittel sind sie m.E. aber doch, weil die spätere Behandlung von Klassen und Objekten auf einem soliden begrifflichen und motivationalen Fundament aufbauen wird. Damit die POO-Beispiele Ihrem Programmierstil nicht prägen, wurde an den Beginn des Kurses ein Beispiel gestellt (Bruchrechnung), das bereits etliche OOP-Prinzipien realisiert.

3.1.2 Kommentare

Java bietet drei Möglichkeiten, den Quelltext zu kommentieren:

- Alle Zeichen von „//“ bis zum Ende der Zeile gelten als Kommentar, wobei kein Terminierungszeichen erforderlich ist, z.B.:

  ```
  private int zaehler; // wird automatisch mit 0 initialisiert
  ```

  Hier wird eine Variablendeklarationsanweisung in der selben Zeile kommentiert.

- Zwischen einer Einleitung durch `/*` und einer Terminierung durch `*/` kann sich ein ausführlicher Kommentar auch über mehrere Zeilen erstrecken, z.B.:

  ```
  /*
     Hier könnte ein Kommentar zur anschließend
     definierten Klasse stehen.
  */
  public class Beispiel {
     .     .     .
  }
  ```

  Ein mehrzeiliger Kommentar eignet sich u.a. auch dazu, einen Programmteil (vorübergehend) zu deaktivieren, ohne ihn löschen zu müssen.

- Vor der Definition bzw. Deklaration von Klasse, Interfaces (s.u.), Methoden oder Variablen darf ein **Dokumentationskommentar** stehen, eingeleitet mit `/**` und beendet mit `*/`, z.B.:

```
/**
  Hier könnte ein Dokumentationskommentar zur anschließend
  definierten Klasse stehen.
*/
public class Beispiel {
  .     .     .
}
```

Er kann mit dem SDK-Werkzeug **javadoc** in eine HTML-Datei extrahiert werden. Die systematische Dokumentation wird über Tags für Methoden-Parameter, Rückgabewerte etc. unterstützt. Nähere Informationen finden Sie z.B. in der Dokumentation zum Java-SDK über

Tool Documentation > javadoc [Win32]

3.1.3 Bezeichner

Für Klassen, Methoden, Variablen, Parameter und sonstige Elemente eines Java-Programms benötigen wir eindeutige Namen, für die folgende Regeln gelten:

- Die Länge eines Bezeichners ist nicht begrenzt.
- Das erste Zeichen muss ein Buchstabe, Unterstrich oder Dollar-Zeichen sein, danach dürfen außerdem auch Ziffern auftreten.
- Java-Programme werden intern im **Unicode**-Zeichensatz dargestellt. Daher erlaubt Java im Unterschied zu den meisten anderen Programmiersprachen in Bezeichnern auch Umlaute oder sonstige nationale Sonderzeichen, die als Buchstaben gelten.
- Die Groß-/Kleinschreibung ist signifikant.
  Für den Java-Compiler sind also z.B.
  ```
  Anz    anz  ANZ
  ```
  grundverschiedene Bezeichner.
- Die folgenden **reservierten Wörter** dürfen nicht als Bezeichner verwendet werden:
  abstract, boolean, break, byte, case, catch, char, class, const[1], continue, default, do, double, else, extends, false, final, finally, float, for, goto[1], if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while

  Wenn Sie versehentlich eines der Java - Schlüsselwörter als Bezeichner verwenden, wird sich der Compiler in der Regel beschweren, z.B.:
  ```
  Fakul.java:9: <identifier> expected
                     int i, break;
                              ^
  ```
- Bezeichner müssen innerhalb ihres Kontextes (s.u.) eindeutig sein.

[1] Diese Schlüsselwörter sind reserviert, werden aber derzeit nicht unterstützt. Im Falle von **goto** wird sich an diesem Zustand wohl auch nichts ändern.

3.1.4 Einfache Ausgabe bei Konsolenanwendungen

Wie Sie bereits an einigen Beispielen beobachten konnten, lässt sich eine Konsolenausgabe in Java recht bequem über die Methodenaufrufe **System.out.print()** oder **System.out.println()** erzeugen.[1] Im Unterschied zu **print()** schließt **println()** die Ausgabe automatisch mit einer Zeilenschaltung ab. Beide Methoden erwarten ein einziges Argument, wobei erlaubt sind:

- Zeichenfolgen, in doppelte Anführungszeichen eingeschlossen

  Beispiel: `System.out.println("Hallo Allerseits!");`
- Sonstige Ausdrücke (siehe Abschnitt 3.4)

  Beispiele:
  - `System.out.println(ivar);`
    Hier wird der Wert der Variablen `ivar` ausgegeben.
  - `System.out.println(i==13);`
    An die Möglichkeit, als **println()**-Parameter, nahezu beliebige Ausdrücke anzugeben, müssen sich Einsteiger erst gewöhnen. Hier wird der Wert eines *Vergleichs* (der Variablen `i` mit der Zahl 13) ausgegeben. Bei Identität erscheint auf der Konsole das Schlüsselwort **true**, sonst **false**.

Besonders angenehm ist die Möglichkeit, mehrere Teilausgaben mit dem „+"-Operator zu verketten, z.B.:

```
System.out.println("Ergebnis: " + netto*MWST);
```

Im Beispiel wird der numerische Wert von `netto*MWST` in eine Zeichenfolge gewandelt und dann mit `"Ergebnis: "` verknüpft.
Eine einfache Möglichkeit zur Tastatur*ein*gabe wird später bei passender Gelegenheit vorgestellt.

3.1.5 Übungsaufgaben zu Abschnitt 3.1

1) Welche **main()**-Varianten sind zum Starten einer Applikation geeignet?

```
public static void main (String[] argz) { … }
public void main (String[] args) { … }
public static int main (String[] args) { … }
public static void main (String args[]) { … }
```

2) Welche von den folgenden Bezeichnern sind unzulässig?

4you maiLink else Lösung b_____

3) Wie ist das fehlerhafte „Rechenergebnis" in folgendem Programm zu erklären?

Quellcode	Ausgabe

`class Prog {`
`   public static void main(String[] args) {`
`      System.out.println("3.3 + 2 = " + 3.3 + 2);`
`   }`
`}`
	`3.3 + 2 = 3.32`

Sorgen Sie mit einem Paar runder Klammern dafür, dass die gewünschte Ausgabe erscheint.

[1] Für eine genauere Erläuterung reichen unsere bisherigen OOP-Kenntnisse noch nicht ganz aus. Wer aus anderen Quellen Vorkenntnisse besitzt, kann die folgenden Sätze vielleicht jetzt schon verdauen:
Wir benutzen bei der Konsolenausgabe die im Paket **java.lang** definierte und damit automatisch in jedem Java-Programm verfügbare Klasse **System**. Deren Variablen sind **static**, können also verwendet werden, ohne ein Objekt aus der Klasse **System** zu erzeugen. U.a. befindet sich unter den **System**-Klassenvariablen ein Objekt namens **out** aus der Klasse **PrintStream**. Es beherrscht u.a. die Methoden **print()** und **println()**, die jeweils ein einziges Argument von beliebigem Datentyp erwarten und zur Standardausgabe befördern.

3.2 Variablen und primitive Datentypen

Ein Programm muss in der Regel zahlreiche Daten mehr oder weniger lange im Arbeitsspeicher des Rechners ablegen. Zum Speichern eines Wertes (z.B. einer Zahl) wird eine so genannte **Variable** verwendet, worunter Sie sich einen **benannten und typisierten Speicherplatz** vorstellen können. Eine Variable erlaubt über ihren Namen den lesenden oder schreibenden Zugriff auf den zugeordneten Platz im Arbeitsspeicher, z.B.:

```
class Prog {
    public static void main(String[] args) {
    int ivar = 4711;                          //schreibender Zugriff auf ivar
    System.out.println(ivar);                 //lesender Zugriff auf ivar
    }
}
```

Um die Details bei der Verwaltung der Variablen im Arbeitsspeicher müssen wir uns nicht kümmern, da wir schließlich mit einer problemorientierten Programmiersprache arbeiten.
Allerdings verlangt Java beim Umgang mit Variablen im Vergleich zu anderen Programmier- oder Skriptsprachen einige Sorgfalt, letztlich mit dem Ziel, Fehler zu vermeiden:

- Variablen müssen **explizit deklariert** werden.
  Wenn Sie versuchen, eine nicht-deklarierte Variable zu verwenden, beschwert sich der Compiler, z.B.:

  ```
  U:\Eigene Dateien\Java\Prog\Prog.java:3: cannot resolve symbol
  symbol  : variable i
  location: class Test
                  i = 5;
                  ^
  ```

  Durch den Deklarationsswang werden z.B. Programmfehler wegen falsch geschriebener Variablennamen verhindert.
- Java ist eine **streng typisiert**
  Jede Variable besitzt einen **Datentyp**, der die erlaubten Werte sowie die zulässigen Operationen festlegt

In obigem Beispiel wird die Variable `ivar` vom Typ **int** deklariert, der ganze Zahlen im Bereich von –2147483648 bis 2147483647 als Werte aufnehmen kann.

Die Variable `ivar` erhält auch gleich den **Initialisierungswert** 4711. Auf diese oder andere Weise müssen Sie jeder lokal (innerhalb einer Methode) definierten Variablen einen Wert zuweisen, bevor Sie zum ersten Mal lesend darauf zugreifen (vgl. Abschnitt 3.2.2).

Als wichtige Eigenschaften einer Java-Variablen halten wir fest:

- **Name**
- **Datentyp**
  Damit sind festgelegt: Wertebereich, zulässige Operationen, Speicherplatzbedarf
- **aktueller Wert**
- **Ort im Hauptspeicher**
  Im Unterschied zu anderen Programmiersprachen (z.B. C++) spielt in Java die Verwaltung von Speicheradressen keine Rolle. Zur Förderung unserer EDV-Allgemeinbildung werden wir uns jedoch gelegentlich dafür interessieren, in welchen Hauptspeicherbereichen eines Programms die verschiedenen Java-Variablen landen.

In Java unterscheiden sich Variablen nicht nur hinsichtlich des Datentyps, sondern auch hinsichtlich der Zuordnung zu einer Methode, zu einem Objekt oder zu einer Klasse:

- **Lokale Variablen**
  Sie werden innerhalb einer Methode deklariert. Ihre Gültigkeit beschränkt sich auf die Methode bzw. auf einen Block innerhalb der Methode (siehe Abschnitt 3.2.3).
- **Instanzvariablen** (Elementvariablen, Eigenschaften)
  Jedes Objekt (synonym: jede Instanz) einer Klasse verfügt über einen vollständigen Satz der Instanzvariablen (Eigenschaften) der Klasse. So besitzt z.B. jedes Objekt der Klasse `Bruch` einen `zaehler` und einen `nenner`. Auf Instanzvariablen kann in allen Methoden der Klasse zugegriffen werden. Wenn entsprechende Rechte eingeräumt wurden, ist dies auch in Methoden fremder Klasen möglich.
- **Klassenvariablen**
  Diese Variablen beziehen sich auf eine gesamte Klasse, nicht auf einzelne Instanzen. Z.B. hält man oft in einer Klassenvariablen fest, wie viele Objekte einer Klasse bereits erzeugt worden sind. Auch für Klassenvariablen gilt, dass neben den klasseneigenen Methoden bei entsprechender Rechtevergabe auch Methoden fremder Klassen zugreifen dürfen.

In Abschnitt 3 werden wir ausschließlich mit lokalen Variablen arbeiten. Im Zusammenhang mit der systematischen Behandlung der objektorientierten Programmierung werden die Instanzvariablen (Eigenschaften) und die Klassenvariablen ausführlich erläutert.

Dann kommen auch so genannte *Referenzvariablen* zur Sprache, die keinen der im aktuellen Abschnitt 3.2.1 darzustellenden primitiven Datentypen besitzen, sondern ein Objekt aus einer bestimmten (z.B. auch benutzerdefinierten) Klasse ansprechen. In der **main**()-Methode der Klasse `BruchRechnung` werden z.B. die Referenzvariablen `b1` und `b1` aus der Klasse `Bruch` deklariert:

```
Bruch b1 = new Bruch(), b2 = new Bruch();
```

Sie erhalten als Initialisierungswert jeweils eine Referenz auf ein neu erzeugtes `Bruch`-Objekt.

In diesem Abschnitt wurden zwei Einteilungskriterien für Java-Variablen vorgestellt, die gemeinsam folgendes Klassifikations-Schema ergeben:

		Einteilung nach Zuordnung		
		Lokale Variable	Instanzvariable	Klassenvariable
Einteilung nach Datentyp (Inhalt)	Primitiver Datentyp			
	Referenzvariable			

Im Unterschied zu anderen Programmiersprachen (z.B. C++) ist es in Java **nicht** möglich, so genannte *globale* Variablen außerhalb von Klassen zu definieren.

Die Erläuterungen zur Einteilung der Java-Variablen sind etwas kompliziert geraten. Bis jetzt müssen Sie nur die Ausführungen verstanden haben, die für *lokale* Variablen von *primitivem* Datentyp relevant sind.

3.2.1 Primitive Datentypen

Als *primitiv* bezeichnet man in Java die auch in älteren Programmiersprachen bekannten *Standardtypen* zur Aufnahme einzelner Werter (Zeichen, Zahlen oder Wahrheitswerte).
Es stehen die folgenden primitiven Datentypen zur Verfügung:

Typ	Beschreibung	Werte	Speicherbedarf in Bit
byte	Diese Variablentypen speichern ganze Zahlen. Beispiel: `int alter = 31;`	–128 … 127	8
short		–32768 … 32767	16
int		–2147483648 ... 2147483647	32
long		–9223372036854775808 … 9223372036854775807	64
float	Variablen vom Typ **float** können Fließkommazahlen mit einer Genauigkeit von mindestens 7 Dezimalstellen speichern. Beispiel: `float pi = 3.141593f;` Float-Literale (s.u.) benötigen den Suffix **f** (oder **F**).	Minimum: $-3.4028235 \cdot 10^{38}$ Maximum: $3.4028235 \cdot 10^{38}$ Kleinster Betrag: $1.4 \cdot 10^{-45}$	32 (1 für das Vorzeichen, 8 für den Exponenten, 23 für die Mantisse)
double	Variablen vom Typ **double** können Fließkommazahlen mit einer Genauigkeit von mindestens 15 Dezimalstellen speichern. Beispiel: `double pi = 3.1415926535898;`	Minimum: $-1.7976931348623157 \cdot 10^{308}$ Maximum: $1.7976931348623157 \cdot 10^{308}$ Kleinster Betrag: $4.9 \cdot 10^{-324}$	64 (1 für das Vorzeichen, 11 für den Exponenten, 52 für die Mantisse)
char	Variablen vom Typ **char** dienen zum Speichern eines Unicode-Zeichens. Im Speicher landet aber nicht die Gestalt eines Zeichens, sondern seine Nummer im Zeichensatz. Daher zählt **char** zu den integralen (ganzzahligen) Datentypen. Soll ein Zeichen als Wert zugewiesen werden, so ist es mit *einfachen* Anführungszeichen einzurahmen. Beispiel: `char zeichen = 'j';`	Unicode-Zeichen	16
boolean	Variablen vom Typ **boolean** können Wahrheitswerte annehmen. Beispiel: `boolean cond = false;`	**true**, **false**	1

Zunächst soll der für EDV-Einsteiger potentiell missverständliche Begriff *Fließ- bzw. Gleitkommazahl* erläutert werden:

- Als Dezimaltrennzeichen ist in Java stets ein *Punkt* zu verwenden. Durch die übliche Übersetzung des englischen Begriffs *floating point number* mit Gleit*komma*zahl entsteht eine hier Unklarheit.
- Von *Gleit*kommazahlen spricht man deshalb, weil in der allgemeinen Form

  $$\pm m \cdot b^{\pm e}$$

  mit der Mantisse *m*, der Basis *b* und dem Exponenten *e* die Position des Dezimaltrennzeichens vom Exponenten abhängt, z.B.:

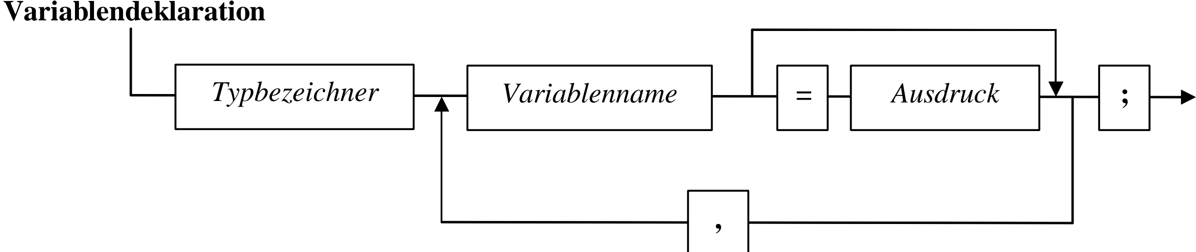
  $1{,}25 = 1{,}25 \cdot 10^{0} = 125 \cdot 10^{-2} = 0{,}0125 \cdot 10^{2}$

  Wenn wir mit einfachen Kommazahlen (ohne Exponent) arbeiten, hat das Dezimaltrennzeichen natürlich keine Veranlassung und keine Berechtigung zum Gleiten.

Ein Vorteil von Java besteht darin, dass die Wertebereiche der elementaren Datentypen auf allen Plattformen identisch sind, worauf man sich bei anderen Programmiersprachen (z.B. C/C++) nicht verlassen kann.
Im Vergleich zu C/C++ fällt noch auf, dass der Einfachheit halber auf *vorzeichenfreie* Datentypen verzichtet wurde.
Neben den primitiven Datentypen werden wir noch *Referenzdatentypen* kennen lernen, über die sich Objekte aus einer bestimmten Klasse ansprechen lassen. Man kann jede Klasse (aus dem Java-API übernommen oder selbst definiert) als Datentyp verwenden, also Referenzvariablen dieses Typs deklarieren. Auch Zeichenfolgen und Felder (Zusammenfassungen von mehreren gleichartigen Einzelvariablen) sind in Java stets Objekte.
Die abwertend klingende Bezeichnung *primitiv* darf keinesfalls so verstanden werden, dass elementare Datentypen nach Möglichkeit in Java-Programmen zu vermeiden wären. Sie sind als Bausteine für die höheren Datentypen und als lokale Variablen in Methoden unverzichtbar.

3.2.2 Variablendeklaration, Initialisierung und Wertzuweisung

Obwohl Sie die Regeln für die **Variablendeklarations-Anweisung** sicher leicht aus den obigen Beispielen abstrahieren könnten, wird ein Syntaxdiagramm dazu angegeben:



Neben den primitiven Datentypen können auch Klassen (aus dem Java-API oder selbst definiert) in einer Variablendeklaration verwendet werden, z.B.:

```
int i;
Bruch b1 = new Bruch();
```

Im zweiten Beispiel wird per **new**-Operator ein `Bruch`-Objekt erzeugt und dessen Adresse in die Referenzvariable `b1` geschrieben. Derartige Details der objektorientierten Java-Programmierung müssen Sie aber jetzt noch nicht verstehen, und auch mit dem Sprachelement *Ausdruck* werden wir uns noch ausführlich beschäftigen.

Wir betrachten vorläufig nur *lokale* Variablen, die innerhalb einer Methode existieren. Diese können an beliebiger Stelle im Methoden-Quellcode deklariert werden, aus nahe liegenden Gründen jedoch vor ihrer ersten Verwendung.

Es hat sich eingebürgert, Variablennamen mit einem Kleinbuchstaben beginnen zu lassen, um sie im Quelltext gut von den Bezeichnern für Klassen oder Konstanten (s.u.) unterscheiden zu können.

Neu deklarierte Variablen kann man optional auch gleich **initialisieren**, also auf einen gewünschten Wert setzen.
Weil *lokale* Variablen nicht automatisch initialisiert werden, muss man ihnen unbedingt vor dem ersten lesenden Zugriff einen Wert zuweisen. Auch im Umgang des Compilers mit uninitialisierten lokalen Variablen zeigt sich das Bemühen der Java-Designer um robuste Programme. Während C++ - Compiler in der Regel nur warnen, produzieren Java-Compiler eine Fehlermeldung und erstellen keinen Bytecode. Dieses Verhalten wird durch folgendes Programm demonstriert:

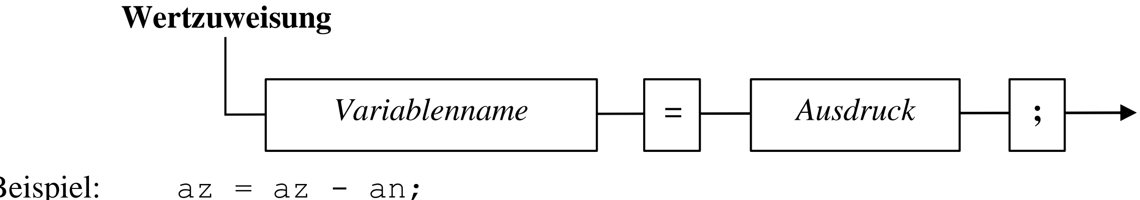
```
class Prog {
    public static void main(String[] args) {
        int argument;
        System.out.print("Argument =  " + argument);
    }
}
```

Der Compiler meint dazu:

```
Prog.java:4: variable argument might not have been initialized
                System.out.print("Argument =  " + argument);
                                                  ^
1 error
```

Weil Instanz- und Klassenvariablen automatisch mit dem Standardwert ihres Typs initialisiert werden (s.u.), ist in Java dafür gesorgt, dass alle Variablen stets einen definierten Wert haben.

Um den Wert einer Variablen im weiteren Pogrammablauf zu verändern, verwendet man eine **Wertzuweisung**, die zu den einfachsten und am häufigsten benötigten Anweisungen gehört:



Beispiel: `az = az - an;`

U.a. haben Sie mittlerweile auch schon zwei Sorten von Java-Anweisungen kennen gelernt: Variablendeklaration und Wertzuweisung.

3.2.3 Blöcke und Gültigkeitsbereiche für lokale Variablen

Wie Sie bereits wissen, besteht der Rumpf einer Methodendeklaration aus einem Block mit beliebig vielen Anweisungen, abgegrenzt durch geschweifte Klammern. Innerhalb des Methodenrumpfes können weitere Anweisungsblöcke gebildet werden, wiederum durch geschweifte Klammen begrenzt:

```
{
        Anweisung;
        .  .  .
        Anweisung;
}
```

Man spricht hier auch von einer **Block- bzw. Verbundanweisung,** und diese kann überall stehen, wo eine einzelne Anweisung erlaubt ist. Unter den Anweisungen eines Blockes dürfen sich selbstverständlich auch wiederum Blockanweisungen befinden. Einfacher ausgedrückt: Blöcke dürfen geschachtelt werden. Oft treten Blöcke zusammen mit Bedingungen oder Wiederholungen auf, z.B.:

Quellcode

```
class Prog {
    public static void main(String[] args) {
        int wert1 = 1;
        System.out.println("Wert1 = " + wert1);

        if (wert1 == 1) {
            int wert2 = 2;
            System.out.println("Wert2 = " + wert2);
        }

    //System.out.println("Wert2 = " + wert2);

    }
}
```

Ausgabe

```
Wert1 = 1
Wert2 = 2
```

Anweisungsblöcke haben einen wichtigen Effekt auf die Gültigkeit der darin deklarierten Variablen: Eine lokale Variable existiert von der deklarierenden Zeile bis zur schließenden Klammer des lokalsten Blockes. Nur in diesem **Gültigkeitsbereich** (engl. scope) kann sie über ihren Namen angesprochen werden. Beim Verlassen des Blockes wird der belegte Speicher frei gegeben, so dass die im obigen Beispiel auskommentierte Zeile folgende Fehlermeldung produzieren würde:

```
Prog.java:16: cannot resolve symbol
symbol  : variable wert2
location: class Prog
        System.out.println("Wert2 = " + wert2);
                                        ^
1 error
```

Bei hierarchisch geschachtelten Blöcken ist es in Java *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Mit dem Verzicht auf diese kaum sinnvolle C++ - Option haben die Java-Designer eine Möglichkeit beseitigt, schwer erkennbare Programmierfehler zu begehen.

Bei der übersichtlichen Gestaltung von Java-Programmen ist das Einrücken von Anweisungsblöcken sehr zu empfehlen. In Bezug auf die räumliche Anordnung (vor allem der einleitenden Blockklammer) haben sich unterschiedliche Stile entwickelt, z.B.:

```
if (wert1 == 1)
{
    int wert2 = 2;
    System.out.println(wert2);
}
```

```
if (wert1 == 1) {
    int wert2 = 2;
    System.out.println(wert2);
}
```

Wählen bzw. entwerfen Sie sich eine Regel, und halten Sie diese möglichst durch (zumindest innerhalb eines Programms).

Bei vielen Texteditoren (allerdings nicht beim Windows-Notepad) kann ein markierter Block aus mehreren Zeilen mit

<Tab> komplett nach rechts eingerückt

und mit

<Umschalt>+<Tab> komplett nach links ausgerückt

werden.

Spezialisierte Editoren für Programmierer bieten noch mehr Komfort, indem sie z.B. beim Markieren einer Blockklammer das Gegenstück farblich hervorheben.

3.2.4 Finalisierte Variablen (Konstanten)

In der Regel sollten auch die im Programm benötigten konstanten Werte (z.B. für den Mehrwertsteuersatz) in einer Variablen abgelegt und im Quelltext über ihren Variablennamen angesprochen werden, denn:

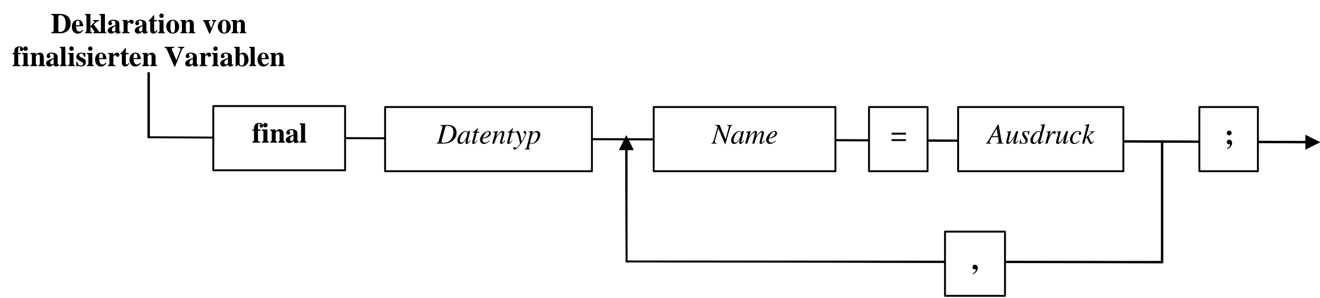
- Bei einer späteren Änderung des Wertes ist nur die Quellcode-Zeile mit der Variablendeklaration und -initialisierung betroffen.
- Der Quellcode ist leichter zu lesen.

Beispiel:

Quellcode	Ausgabe

<pre>class Prog {
   public static void main(String[] args) {
      final double MWST = 1.16;
      double netto = 100.0, brutto;
      brutto = netto * MWST;
      System.out.println("Brutto: " + brutto);
   }
}</pre>
	`Brutto: 115.99999999999999`

Variablen, die nach ihrer Initialisierung im gesamten Programmverlauf garantiert den selben Wert besitzen sollen, deklariert man als **final**:



Die Unterschiede im Vergleich zur gewöhnlichen Variablendeklaration:

- Am Anfang steht der Typqualifizierer **final**.
- Eine finalisierte Variable muss natürlich initialisiert werden.

Man schreibt üblicherweise die Namen von finalisierten Variablen (Konstanten) komplett in Großbuchstaben, um sie im Quelltext gut von anderen Sprachelementen unterscheiden zu können.

3.2.5 Literale

Die im Programmcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.2.4 wissen, ist es oft sinnvoll, Literale *innerhalb von Konstanten-Deklarationen* zu verwenden, z.B.:

```
final double MWST = 1.16;
```

Auch die Literale besitzen in Java stets einen Datentyp, wobei in Bezug auf die primitiven Datentypen einige Regeln zu beachten sind:

3.2.5.1 Ganzzahl-Literale

Ganzzahlige Werte haben in Java den Datentyp **int**, wenn nicht durch den Suffix **L** oder **l** der Datentyp **long** erzwungen wird, z.B.:

```
final long BETRAG = 2147483648L;
```

Bei der Notation sind weitere Feinheiten zu beachten. So liefern die harmlos wirkenden Anweisungen:

```
int i = 11, j = 011;
System.out.println("i = " + i + ", j = " + j);
```

ein auf den ersten Blick verblüffendes Ergebnis:

```
i = 11, j = 9
```

Durch ein anderes Beispiel:

```
int i = 8, j = 08;
System.out.println("i = " + i + ", j = " + j);
```

kommen wir dem Rätsel auf die Spur. Hier meldet der Compiler:

```
Prog.java:5: integer number too large: 08
                final int i = 8, j = 08;
                                     ^
```

Die führende Null ist keinesfalls irrelevant, sondern kündigt dem Compiler einen Wert im *Oktalsystem* an, er interpretiert „011“ als:

$$11_{\text{Oktal}} = 1 \cdot 8 + 1 \cdot 1 = 9$$

Weil im Oktalsystem nur die Ziffern 0 bis 7 verwendet werden, beschwert sich der Compiler zu Recht gegen die Ziffernfolge „08“.

Mit dem Präfix „0x“ (oder auch „0X“) werden ganzzahlige Werte im *Hexadezimalsystem* ausgedrückt. Die Anweisungen:

```
int i = 11, j = 0x11;
System.out.println("i = " + i + ", j = " + j);
```

liefern die Ausgabe:

```
i = 11, j = 17
```

3.2.5.2 Gleitkomma-Literale

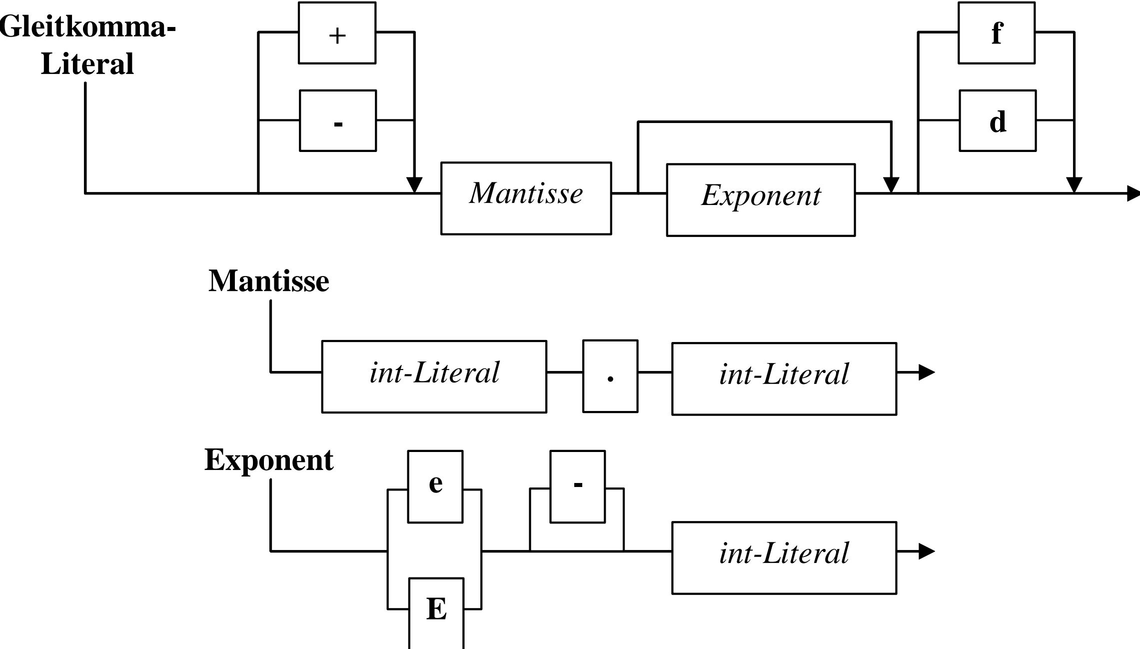
Zahlen mit Dezimalpunkt oder Exponent sind in Java vom Typ **double**, wenn nicht durch den Suffix **F** oder **f** der Datentyp **float** erzwungen wird.
Der Java-Compiler achtet bei Wertzuweisungen streng auf die Typkompatibilität. Z.B. führt die folgende Zeile:

```
final float PI = 3.141593;
```

zu der Fehlermeldung:

```
Prog.java:6: possible loss of precision
found    : double
required: float
                final float PI = 3.141593;
                                 ^
1 error
```

Hinsichtlich der Schreibweise von Gleitkomma-Literalen bietet Java etliche Möglichkeiten, von denen die wichtigsten in folgenden Syntaxdiagrammen dargestellt werden:



Bei den in Gleitkomma-Literalen auftretenden **int**-Literalen sind die oben beschriebenen Präfixe (0, 0x, 0X) und Suffixe (L, l) *nicht* erlaubt.

Beispiele für gültige Gleitkomma-Literale:

```
9.78f
0.45875e-20
```

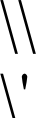
3.2.5.3 char-Literale

char-Literale werden in Java mit einfachen Hochkommata eingerahmt. Dabei sind erlaubt:

- **Unicode-Zeichen**
  Nicht erlaubt: Einfaches Hochkomma und Rückwärts-Schrägstrich (\)
- **Escape-Sequenzen**
  Hier dürfen einem einleitenden Rückwärts-Schrägstrich u.a. folgen:
  - Ein Befehls-Zeichen, z.B.:

Neue Zeile	`\n`
Horizontaler Tabulator	`\t`

  - Eines der aus syntaktischen Gründen verbotenen Zeichen (einfaches Hochkomma und Rückwärts-Schrägstrich), um es auf diese Weise doch angeben zu können:

    
    `\\`
    `\'`

- Eine Unicode-Escapesequenz mit hexadezimaler Zeichennummer
  Weil die ASCII-Zeichen mit ihren gewohnten Nummer in den Unicode-Zeichensatz übernommen wurden, kann z.B. auf folgende Weise ein Ton (ASCII-Nummer 7) erzeugt werden:
  ```
  System.out.println('\u0007');
  ```

Java-Konsolenanwendungen haben unter Windows einen Schönheitsfehler, von dem die erheblich relevanteren grafikorientieren Anwendungen *nicht* betroffen sind:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
  public static void main(String[] args) {
    System.out.println("Übler Ärger macht böse!");
  }
}</pre></td><td><pre>■bler —rger macht b÷se!</pre></td></tr>
</table>

Die schlechte Behandlung von Umlauten bei Konsolenanwendungen unter Windows geht auf folgende Ursachen zurück:

- Windows arbeitet in Konsolenfenstern DOS-konform mit dem ASCII-Zeichensatz, in grafikorientierten Anwendungen hingegen mit dem ANSI-Zeichensatz.
- Die virtuelle Java-Maschine arbeitet unter Windows grundsätzlich mit dem ANSI-Zeichensatz.

Java bietet durchaus eine Lösung für das Problem, die im Wesentlichen aus einer Ausgabestrom-konvertierungsklasse besteht und später im passenden Kontext dargestellt wird. Da wir Konsolenanwendungen nur als relativ einfache Umgebung zum Erlernen grundlegender Techniken und Konzepte verwenden und letztlich grafikorientierte Anwendungen entwickeln wollen, werden wir den Schönheitsfehler vorübergehend tolerieren.

3.2.5.4 Zeichenketten-Literale

Zeichenketten-Literale werden (im Unterschied zu **char**-Literalen) durch *doppelte* Hochkommata begrenzt. Hinsichtlich der erlaubten Zeichen und der Escape-Sequenzen gelten die Regeln für **char**-Literale analog, wobei natürlich das einfache und das doppelte Hochkomma ihre Rollen tauschen.

3.2.5.5 boolean-Literale

Als Literale vom Typ **boolean** sind nur die beiden reservierten Wörter **true** und **false** erlaubt, z.B.:

```
boolean cond = true;
```

3.2.6 Übungsaufgaben zu Abschnitt 3.2

1) In folgendem Programm wird einer **char**-Variablen eine *Zahl* zugewiesen, die sie offenbar unbeschädigt an eine **int**-Variable weitergeben kann, wobei ihr Inhalt von **println()** aber als Buchstabe ausgegeben wird. Wie erklären sich diese Merkwürdigkeiten?

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
   public static void main(String args[]) {
       char z = 99;
       int i = z;
       System.out.println("z = "+z+"\ni = "+i);
   }
}</pre></td><td><pre>z = c
i = 99</pre></td></tr>
</table>

Wie kann man das Zeichen *c* über eine Unicode-Escapesequenz ansprechen?

2) Warum meldet der Compiler hier einen möglichen Verlust an Genauigkeit?

<table>
<tr><th>Quellcode</th><th>Fehlermeldung</th></tr>
<tr><td><pre>class Prog {
  public static void main(String args[]) {
    int i = 7l;
    System.out.println(i);
  }
}</pre></td><td><pre>possible loss of precision</pre></td></tr>
</table>

3) Wieso klagt der Compiler über ein unbekanntes Symbol, obwohl die Variable `i` korrekt deklariert worden ist?

<table>
<tr><th>Quellcode</th><th>Fehlermeldung</th></tr>
<tr><td><pre>class Prog {
  public static void main(String[] args) {{
    int i = 2;
  }
    System.out.println(i);
  }
}</pre></td><td><pre>Prog.java:5: cannot resolve symbol
symbol : variable i
location: class Prog
System.out.println(i);
                   ^</pre></td></tr>
</table>

4) Schreiben Sie bitte ein Java-Programm, das folgende Ausgabe macht:

```
Dies ist ein Java-Zeichenkettenliteral:
   "Hallo"
```

5) Beseitigen Sie bitte alle Fehler in folgendem Programm:

```
class Prog {
    static void main(String[] args) {
        float PI = 3,141593;
        double radius = 2,0;
        System.out.printline("Der Flaecheninhalt betraegt: + PI * radius * radius);
    }
}
```

3.3 Einfache Eingabe bei Konsolenanwendungen

3.3.1 Beschreibung der Klasse `Simput`

Für die Übernahme von Tastatureingaben ist in Java eine ziemlich wasserdichte Methodologie vorgeschrieben, wie das folgende Beispiel zur Berechnung der Fakultät zeigt:

```
import java.io.*;
class Fakul {
    public static void main(String[] args) {
        int i, argument;
        double fakul = 1.0;
        BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));
        System.out.print("Argument:  ");
        try {
            argument = Integer.parseInt(in.readLine());
            for (i = 1; i <= argument; i++)
              fakul = fakul * i;
            System.out.println("Fakultaet: " + fakul);
        } catch(Exception e) {
            System.out.println("Falsche Eingabe");
        }
    }
}
```

Hier kommen aufwändige Programmiertechniken zum Einsatz, mit denen wir uns momentan noch nicht beschäftigen wollen. Damit wir bis zum Erlernen dieser Techniken in Demonstrations- bzw. Übungsprogrammen nicht auf Tastatureingaben verzichten müssen, habe ich eine Klasse namens `Simput` mit Methoden zur bequemen Übernahme von Tastatureingaben erstellt. Die zugehörige Bytecode-Datei **Simput.class** findet sich bei den Übungs- und Beispielprogrammen zum Kurs (Verzeichnis **...\BspUeb\Anwendungen\Simput**, weitere Ortsangaben im Vorwort) sowie im folgenden Ordner auf dem Laufwerk P im Campusnetz der Universität Trier:

P:\Prog\JavaSoft\JDK\Erg\class

Mit Hilfe der Klassenmethode `Simput.gint()` lässt sich das Fakultätsprogramm etwas einfacher realisieren. Die dabei verwendete **for**-Wiederholungsanweisung wird in Abschnitt 3.5.3 behandelt.

```
class Fakul {
    public static void main(String[] args) {
        int i, argument;
        double fakul = 1.0;
        System.out.print("Argument:  ");
        argument = Simput.gint();
        for (i = 1; i <= argument; i++)
            fakul = fakul * i;
        System.out.println("Fakultaet: " + fakul);
    }
}
```

`Simput.gint()` erwartet vom Benutzer eine per **<Enter>**-Taste quittierte Eingabe und versucht, diese als **int**-Literal zu interpretieren. Im Erfolgsfall wird das Ergebnis als `gint()`-Rückgabewert an die Variable `argument` übergeben. Anderenfalls erhält der Benutzer eine Fehlermeldung, und es wird eine Problembeschreibung in zwei Klassenvariablen (`Simput.status` und `Simput.errdes`) hinterlassen, die in obigem Beispiel von der aufrufenden Methode `main()` allerdings nicht überprüft werden. Wir werden in vielen Beispielen der folgenden Abschnitte den `gint()`-Rückgabewert der Kürze halber ebenfalls ohne Status-Kontrolle benutzen. Bei Anwendungen für den praktischen Einsatz sollte aber wie in folgender Variante des Fakultätsprogramms eine Überprüfung vorgenommen werden. Die dazu erforderliche **if**-Anweisung wird in Abschnitt 3.5.2 behandelt.

Quellcode	Ein- und Ausgabe

`class Fakul`
`{`
`   public static void main(String args[])`
`   {`
`      int i, argument;`
`      double fakul = 1.0;`
 	
`      System.out.print("Argument:  ");`
`      argument = Simput.gint();`
 	
`      if (Simput.status == true) {`
`       for (i = 1; i <= argument; i += 1)`
`            fakul = fakul * i;`
`       System.out.println("Fakultaet: " + fakul);`
`      }`
`      else`
`       System.out.println(Simput.errdes);`
`   }`
`}`
	`Argument:  ff`
	`Falsche Eingabe!`
	
	`Eingabe konnte nicht konvert. werden.`

Neben `gint()` besitzt die Klasse `Simput` noch analoge Methoden für alternative Datentypen, u.a.:

- `static public char gchar()`
  Liest ein Zeichen von der Konsole
- `static public double gdouble()`
  Liest eine Gleitkommazahl von der Konsole

Außerdem sind Methoden mit einer moderneren Fehlerbehandlung über den Exception-Mechanismus vorhanden, auf die wir aus didaktischen Gründen vorläufig noch verzichten.

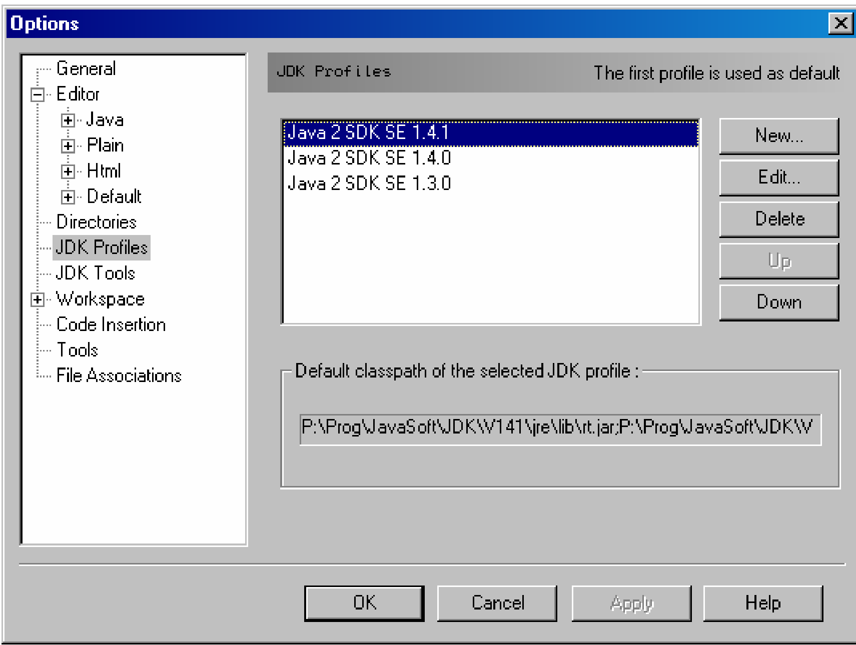
3.3.2 Simput-Installation für die SDK-Werkzeuge und JCreator

Damit beim Übersetzen per **javac.exe** und/oder beim Ausführen per **java.exe** die `Simput`-Klasse mit ihren Methoden verfügbar ist, muss die Datei **Simput.class** entweder im aktuellen Verzeichnis liegen oder über den CLASSPATH erreichbar sein, wenn sie nicht bei jedem Compiler- oder Interpreteraufruf per **classpath**-Kommandozeilenoption zugänglich gemacht werden soll.

Bei Verwendung der Java-Entwicklungsumgebung JCreator LE 2.x ist die CLASSPATH-Umgebungsvariable unwirksam, weil JCreator LE beim Start des JDK-Compilers bzw. –Interpreters grundsätzlich die Kommandozeilenoption „-classpath" benutzt (vgl. Abschnitt 2.1.4). Man kann aber leicht dafür sorgen, dass bei der Kommandozeilenoption alle benötigten Pfade einbezogen werden. Dazu ist nach

Configure > Options > JDK Profiles

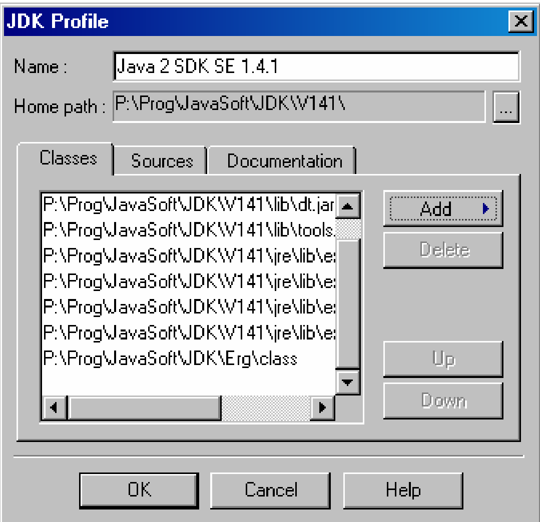
das verwendete JDK-Profil zu erweitern:



Markieren Sie in der **Options**-Dialogbox das gewünschte Profil, und fordern Sie mit **Edit** die Dialogbox **JDK Profile** an. Bei aktivem Registerblatt **Classes** kann über

Add > Add Path

ein zusätzlicher Pfad eingetragen werden, z.B.:



3.3.3 Übungsaufgabe zu Abschnitt 3.3

Erproben Sie die in Abschnitt 3.3 beschriebene Klasse `Simput`:

- Führen Sie nötigenfalls die in Abschnitt 3.3.2 beschriebene Installation aus.
- Welchen Wert liefert `gint()` bei einer ungültigen Eingabe zurück?
- Testen Sie die Methoden `gdouble()` und `gchar()`.

3.4 Operatoren und Ausdrücke

Im Zusammenhang mit der Variablendeklaration und der Wertzuweisung haben wir das Sprachelement *Ausdruck* ohne Erklärung benutzt; diese soll nun nachgeliefert werden. In Abschnitt 3.4 werden wir Ausdrücke als wichtige Bestandteile von Java-Anweisungen recht detailliert untersuchen. Große Begeisterungsstürme werden dabei wohl nicht aufbrausen, doch im Sinne einer zügigen und fehlerfreien Softwareentwicklung kommen wir an den handwerklichen Grundlagen nicht vorbei.

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf handelt es sich um einen Ausdruck. Dieser besitzt einen **Datentyp** und einen **Wert**.

Beispiel: `1.5`
Dies ist ein Ausdruck mit dem Typ **double** und dem Wert 1.5.

Besteht ein Ausdruck allerdings aus einem Methodenaufruf mit dem Pseudo-Rückgabetyp **void** (siehe unten), dann liegt kein Wert vor.

Mit Hilfe diverser Operatoren entstehen komplexe Ausdrücke, wobei Typ und Wert von den Argumenten und den Operatoren abhängen.

Beispiel: `2.0 + 1.5`
Hier resultiert der **double**-Wert 3.5.

In der Regel beschränken sich Java-Operatoren darauf, aus ihren Argumenten (Operanden) einen Wert zu ermitteln und für die weitere Verarbeitung zur Verfügung zu stellen. Einige Operatoren haben jedoch zusätzlich einen (Neben-)Effekt auf eine als Argument fungierende Variable. Dann sind in der betreffenden Operandenrolle auch nur Ausdrücke erlaubt, die für eine Variable stehen.

Die meisten Java-Operatoren verarbeiten *zwei* Operanden (Argumente) und heißen daher **zweistellig** bzw. **binär**.

Beispiel: `a + b`
Der Additionsoperator wird mit dem „+"-Symbol bezeichnet. Er erwartet zwei numerische Argumente.

Manche Operatoren begnügen sich mit einem Argument und heißen daher **einstellig** bzw. **unär**.

Beispiel: `-a`
Die Vorzeichenumkehr wird mit dem „-"-Symbol bezeichnet. Sie erwartet *ein* numerisches Argument.

Wir werden auch noch einen dreistelligen Operator kennen lernen.

Weil als Argumente einer Operation auch *Ausdrücke* von passendem Ergebnistyp erlaubt sind, können beliebig komplexe Ausdrücke aufgebaut werden. Unübersichtliche Exemplare sollten jedoch als potentielle Fehlerquellen vermieden werden.

3.4.1 Arithmetische Operatoren

Weil die arithmetischen Operatoren für die vertrauten Grundrechenarten der Schulmathematik zuständig sind, müssen ihre Operanden (Argumente) einen Ganzzahl- oder Gleitkomma-Typ haben (**byte**, **short**, **int**, **long**, **char**, **float** oder **double**). Die resultieren Ausdrücke haben wiederum einen numerischen Ergebnistyp und werden daher gelegentlich als **numerische Ausdrücke** bezeichnet.

Dass Computer beim Umgang mit den Grundrechenarten so ihre Eigenarten haben, zeigt das folgende Programm:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>
class Prog {
   public static void main(String[] args) {
       int i = 2, j = 3;
       double a = 2.0;
       System.out.println(i/j);
       System.out.println(a/j);
   }
}
</pre></td><td><pre>
0
0.6666666666666666
</pre></td></tr>
</table>

Es hängt von den Datentypen der Operanden ab, ob die **Ganzzahl**- oder die **Gleitkommarithmetik** zum Einsatz kommt. Statt diese beiden Begriffe systematisch zu erläutern, werden nur zwei wichtige Eigenschaften angegeben:

- Bei der Ganzzahl-Division werden Nachkommastellen abgeschnitten, was gelegentlich durchaus erwünscht ist.
- Die Gleitkomma-Arithmetik kann mit Nachkommastellen umgehen, hat aber gelegentlich kleine Genauigkeitsprobleme, wie schon an einem Beispiel in Abschnitt 3.2.4 zu sehen war. Für

  100 * 1.16

  erhalten wir das Ergebnis

  115.99999999999999

Wie der vom Compiler gewählte Arithmetik-Typ und damit auch der Ergebnis-Datentyp von den Datentypen der Argumente abhängen, ist in folgender Tabelle beschrieben:

<table>
<tr><th colspan="2">Datentypen der Operanden</th><th>Datentyp des Ergebniswertes</th></tr>
<tr><td rowspan="2">Kein Operand hat einen Gleitkommatyp (float oder double), so dass mit Ganzzahl-Arithmetik gerechnet wird</td><td>Beide Operanden sind vom Datentyp int.</td><td>int</td></tr>
<tr><td>Mindestens ein Operand hat den Datentyp long.</td><td>long</td></tr>
<tr><td rowspan="2">Mindestens ein Operand hat einen Gleitkommatyp, so dass mit Gleitkomma-Arithmetik gerechnet wird.</td><td>Mindestens ein Operand hat den Typ float, keiner hat den Typ double.</td><td>float</td></tr>
<tr><td>Mindestens ein Operand hat den Datentyp double.</td><td>double</td></tr>
</table>

Bevor ein ganzzahliges Argument an der Gleitkomma-Arithmetik teilnimmt, wird es automatisch in einen Gleitkommatyp gewandelt (vgl. Abschnitt 3.4.6).

In der folgenden Tabelle mit allen arithmetischen Operatoren stehen *num*, *num1* und *num2* für numerische Ausdrücke, *var* vertritt eine numerische Variable:

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
-num	**Vorzeichenumkehr**	`int i = -2;` `System.out.println(-i);`	2
num1 + *num2*	**Addition**	`int i = 2, j = 3;` `System.out.println(i+j);`	5
num1 – *num2*	**Subtraktion**	`double a = 2.6, b = 1.1;` `System.out.println(a-b);`	1.5
num1 * *num2*	**Multiplikation**	`int i = 4, j = 5;` `System.out.println(i*j);`	20
num1 / *num2*	**Division**	`System.out.println(8.0/5);` `System.out.println(8/5);`	1.6 1
num1 % *num2*	**Modulo (Restwert)** Das Ergebnis von *num1* % *num2* ist der Rest der Division von *num1* durch *num2*.	`int i = 9, j = 5;` `System.out.println(i%j);`	4

++*var*
−−*var*
	Präinkrement bzw. dekrement
	Als Argumente sind hier nur Variablen erlaubt.
	++*var* liefert *var* + 1
	erhöht *var* um 1
	−−*var* liefert *var* − 1
	verringert *var* um 1
		`int i = 4;`
		`double a = 0.2;`
		`System.out.println(++i +"\n"+`
		`--a);`
			5
			-0.8

var++ *var*−−	**Postinkrement bzw. -dekrement** Als Argumente sind hier nur Variablen erlaubt. *var*++ liefert *var* erhöht *var* um 1 *var*−− liefert *var* verringert *var* um 1	`int i = 4;` `System.out.println(i++ +" "+ i);`	4 5

Bei den Prä- und Postinkrement- bzw. dekrementoperatoren ist zu beachten, dass sie *zwei* Effekte haben:

- Das Argument wird ausgelesen, um den Wert des Ausdrucks zu ermitteln.
- Der Wert des Argumentes wird verändert.
  Wegen dieses „Nebeneffektes“ sind Prä- und Postinkrement- bzw. dekrementausdrücke im Unterschied zu sonstigen arithmetischen Ausdrücken bereits vollständige Anweisungen (vgl. Abschnitt 3.5.1):

Quellcode	Ausgabe

`class Prog {`
`   public static void main(String[] args) {`
`      int i = 12;`
`      i++;`
`      System.out.println(i);`
`   }`
`}`
	`13`

`class Prog {`
`   public static void main(String[] args) {`
`      int i = 12;`
`      i+1;`
`      System.out.println(i);`
`   }`
`}`
	`Test.java:4: not a statement`
	`                i+1;`
	`                 ^`
	
	`1 error`

3.4.2 Methodenaufrufe

Mit den arithmetischen Operatoren lassen sich nur elementare mathematische Probleme lösen. Darüber hinaus stellt Java eine große Zahl mathematischer Standardfunktionen (z.B. Potenzfunktion, Logarithmus, Wurzel, trigonometrische und hyperbolische Funktionen) über Methoden der Klasse **Math** im API-Paket **java.lang** zur Verfügung. In folgendem Programm wird die Methode **pow()** zur Potenzberechnung genutzt:

Quellcode	Ausgabe

`class Power {`
`   public static void main(String[] args) {`
`      System.out.println(4 * Math.pow(2, 3));`
`   }`
`}`
	`32.0`

Alle Methoden der Klasse **Math** sind **static**, können also verwendet werden, ohne ein Objekt aus der Klasse **Math** zu erzeugen. Später werden wir uns ausführlich mit der Verwendung von Methoden aus Java-Standardpaketen befassen.

In syntaktischer Hinsicht halten wir fest, dass ein Methodenaufruf einen Ausdruck darstellt und auch als Argument komplexerer Ausdrücke verwendet werden darf, sofern die Methode einen passenden Rückgabewert liefert.

3.4.3 Vergleichsoperatoren

Bislang haben wir *numerische* Ausdrücke betrachtet, die mit Hilfe von arithmetischen Operatoren aus einfachen Elementen aufgebaut werden. Durch Verwendung von *Vergleichsoperatoren* und *logischen Operatoren* entstehen Ausdrücke mit dem Ergebnistyp **boolean**, die als **logische Ausdrücke** bezeichnet werden sollen. Sie können die booleschen Werte **true** (wahr) und **false** (falsch) annehmen und eignen sich dazu, *Bedingungen* zu formulieren, z.B.:

```
if (arg > 0)
   System.out.println(Math.log(arg));
```

Ein **Vergleich** ist ein besonders einfach aufgebauter logischer Ausdruck, bestehend aus zwei Ausdrücken und einem Vergleichsoperator.
In der folgenden Tabelle mit den von Java unterstützten Vergleichsoperatoren stehen *expr1* und *expr2* für beliebige, miteinander vergleichbare Ausdrücke sowie *num1* und *num2* für numerische Ausdrücke (vom Datentyp **byte**, **short**, **int**, **long**, **char**, **float** oder **double**):

Operator	Bedeutung	Beispiel	
		Programmfragment	**Ausgabe**
expr1 == expr2	**Gleichheit**	`int i = 2, j = 3;` `System.out.println(i==j);`	`false`

expr1 != expr2	**Ungleichheit**	`int i = 2, j = 3;`
		`boolean erg = i!=j;`
		`System.out.println(erg);`
			`true`

num1 > num2	**größer**	`System.out.println(3 > 2);`	`true`
num1 < num2	**kleiner**	`System.out.println(3 < 2);`	`false`
num1 >= num2	**größer oder gleich**	`System.out.println(3 >= 3);`	`true`
num1 <= num2	**kleiner oder gleich**	`System.out.println(3 <= 2);`	`false`

Achten Sie unbedingt darauf, dass der Identitätsoperator durch **zwei** „="-Zeichen ausgedrückt wird. Einer der häufigsten Java-Programmierfehler besteht darin, beim Identitätsoperator nur ein Gleichheitszeichen zu schreiben. Dabei kommt es in der Regel nicht zu einer Compiler-Fehlermeldung, sondern zu einem unerwarteten Verhalten des Programms.
Im ersten **println**()-Aufruf des folgenden Programms wird das Ergebnis eines Vergleichs auf die Konsole geschrieben[1]:

Quellcode

```
class Prog {
   public static void main(String[] args) {
      int i = 1;
      System.out.println(i == 2);
      System.out.println(i);
   }
}
```

Ausgabe

```
false
1
```

Nach dem Entfernen eines Gleichheitszeichens wird aus dem logischen Ausdruck ein *Wertzuweisungsausdruck* (siehe Abschnitt 3.4.7) mit dem Datentyp **int** und dem Wert 2:

Quellcode

```
class Prog {
   public static void main(String[] args) {
      int i = 1;
      System.out.println(i = 2);
      System.out.println(i);
   }
}
```

Ausgabe

```
2
2
```

Der versehentlich gebildete Ausdruck sorgt nicht nur für eine unerwartete Ausgabe, sondern verändert auch den Wert den Variablen *i*, was im weiteren Verlauf eines Programms recht unangenehm werden kann.

3.4.4 Logische Operatoren

Durch Anwendung der logischen Operatoren auf bereits vorhandene logische Ausdrücke kann man neue, komplexere logische Ausdrücke erstellen.
Die Wirkungsweise der logischen Operatoren wird in **Wahrheitstafeln** beschrieben (*la1* und *la2* seien logische Ausdrücke):

[1] Wir wissen schon aus Abschnitt 3.1.4, dass **println()** einen beliebigen Ausdruck verarbeiten kann.

Argument *la1*	**Negation** **!** *la1*
true	false
false	true

<table>
<tr><th>Argument 1
la1</th><th>Argument 2
la2</th><th>Logisches UND
la1 && la2
la1 & la2</th><th>Log. ODER
la1 || la2
la1 | la2</th><th>Exkl. ODER
la1 ^ la2</th></tr>
<tr><td>true</td><td>true</td><td>true</td><td>true</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>false</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>true</td><td>false</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>false</td><td>false</td><td>false</td><td>false</td></tr>
</table>

In der folgenden Tabelle gibt es noch wichtige Erläuterungen und Beispiele:

<table>
<tr><th rowspan="2">Operator</th><th rowspan="2">Bedeutung</th><th colspan="2">Beispiel</th></tr>
<tr><th>Programmfragment</th><th>Ausgabe</th></tr>
<tr><td>! la1</td><td>Negation
Der Wahrheitswert wird umgekehrt.</td><td>boolean erg = true;
System.out.println(!erg);</td><td>false</td></tr>
<tr><td>la1 && la2</td><td>Logisches UND (mit bedingter Auswertung)
la1 && la2 ist genau dann wahr, wenn beide Argumente wahr sind. Ist la1 falsch, wird la2 nicht ausgewertet.</td><td>int i = 3;
boolean erg = false && i++ > 3;
System.out.println(erg + " " + i);</td><td>false 3</td></tr>
<tr><td>la1 & la2</td><td>Logisches UND (mit unbedingter Auswertung)
la1 & la2 ist genau dann wahr, wenn beide Argumente wahr sind. Es werden aber auf jeden Fall beide Ausdrücke ausgewertet.</td><td>int i = 3;
boolean erg = false & i++ > 3;
System.out.println(erg + " " + i);</td><td>false 4</td></tr>
<tr><td>la1 || la2</td><td>Logisches ODER (mit bedingter Auswertung)
la1 || la2 ist genau dann wahr, wenn mindestens ein Argument wahr ist. Ist la1 wahr, wird la2 nicht ausgewertet.</td><td>int i = 3;
boolean erg = true || i++ > 3;
System.out.println(erg + " " + i);</td><td>true 3</td></tr>
<tr><td>la1 | la2</td><td>Logisches ODER (mit unbedingter Auswertung)
la1 | la2 ist genau dann wahr, wenn mindestens ein Argument wahr ist. Es werden aber auf jeden Fall beide Ausdrücke ausgewertet.</td><td>int i = 3;
boolean erg = true | i++ > 3;
System.out.println(erg + " " + i);</td><td>true 4</td></tr>
</table>

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
la1 ^ *la2*	**Exklusives logisches ODER** *la1* **^** *la2* ist genau dann wahr, wenn genau ein Argument wahr ist, wenn also die Argumente verschiedene Wahrheitswerte haben.	`boolean erg = true ^ true;` `System.out.println(erg);`	`false`

Der Unterschied zwischen den beiden logischen UND-Operatoren **&&** und **&** bzw. zwischen den beiden logischen ODER-Operatoren **||** und **|** ist für Einsteiger vielleicht etwas unklar, weil man spontan den nicht ausgewerteten logischen Ausdrücken keine Bedeutung beimisst. Allerdings ist es in Java erlaubt, „Nebeneffekte“ in einen logischen Ausdruck einzubauen, wie das Beispiel

```
false & i++ > 3
```

zeigt. Der Postinkrementoperator erhöht beim Auswerten des rechten Ausdrucks den Wert der Variablen `i`. Eine solche Auswertung wird jedoch in der folgenden Variante des Beispiels unterdrückt, weil bereits nach Auswertung des linken Ausdrucks das Gesamtergebnis **false** fest steht:

```
false && i++ > 3
```

Wie der Tabelle auf Seite 53 zu entnehmen ist, unterscheiden sich die beiden UND-Operatoren **&&** und **&** bzw. die beiden ODER-Operatoren **||** und **|** auch hinsichtlich der Auswertungspriorität.

Um die Verwirrung noch ein wenig zu steigern, werden die Zeichen **&** und **|** im nächsten Abschnitt noch für bitorientierte Operatoren verwendet. Hier hätte man in Java durch ein stärkeres Abrücken von der C++ - Syntax mehr zur Vermeidung von Fehlern tun können.
Man kann sich aber leicht dadurch aus der Affäre ziehen, dass man für das logische UND bzw. ODER ausschließlich die bedingt auswertenden Operatoren **&&** bzw. **||** verwendet.

3.4.5 Bitorientierte Operatoren

Über unseren momentanen Bedarf hinaus gehend bietet Java einige Operatoren zur bitweisen Manipulation von Variableninhalten. Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe z.B. Java-Tutorial, SUN Inc. 2003) beschränken wir uns auf ein Beispielprogramm, das zudem generell nützliche Einblicke in die Speicherung von **char**-Daten im Computerspeicher vermitteln kann. Allerdings sind Beispiel und zugehörige Erläuterungen mit einigen technischen Details belastet. Wenn Ihnen der Sinn momentan nicht danach steht, können Sie den aktuellen Abschnitt ohne Sorge um den weiteren Kurserfolg an dieser Stelle verlassen.
Das Programm **CBit** liefert die Unicode-Kodierung zu einem vom Benutzer erfragten Zeichen. Dabei kommt die Methode `gchar()` aus unserer Bequemlichkeitsklasse `Simput` zum Einsatz, die das vom Benutzer eingetippte und mit **Enter** quittierte Zeichen abliefert (vgl. Abschnitt 0). Außerdem wird mit der **for**-Schleife eine Wiederholungsanweisung benutzt, die erst einige Abschnitte später offiziell vorgestellt wird:

Quellcode	Ausgabe

`class CBit {`
`   public static void main(String[] args) {`
`      char cbit;`
`      System.out.print("Zeichen: ");`
`      cbit = Simput.gchar();`
`      System.out.print("Unicode: ");`
`      for(int i = 15; i >= 0; i--) {`
`         if ((1 << i & cbit) != 0)`
`            System.out.print("1");`
`         else`
`            System.out.print("0");`
`      }`
`      System.out.println();`
`   }`
`}`
	`Zeichen: x`
	`Unicode: 0000000001111000`

Der **Links-Shift-Operator <<** in:

```
1 << i
```

verschiebt die Bits in der binären Repräsentation der Ganzzahl 1 um `i` Stellen nach links.
Von den 32 Bit, die ein **int**-Wert insgesamt belegt (s.o.), interessieren im Augenblick nur die rechten 16. Bei der 1 erhalten wir:

```
0000000000000001
```

Im 10. Schleifendurchgang (`i` = 6) geht dieses Muster z.B. über in:

```
0000000001000000
```

Nach dem Links-Shift- kommt der **bitweise UND-Operator** zum Einsatz:

```
1 << i & cbit
```

Das Operatorzeichen **&** wird leider in doppelter Bedeutung verwendet: Wenn beide Argumente vom Typ **boolean** sind, wird **&** als *logischer* Operator interpretiert (siehe Abschnitt 3.4.4). Sind jedoch (wie im vorliegenden Fall) beide Argumente von integralem Typ, was auch für den Typ **char** zutrifft, dann wird **&** als UND-Operator für Bits aufgefasst. Er erzeugt dann ein Bitmuster, das an der Stelle **j** eine 1 enthält, wenn *beide* Argumentmuster an dieser Stelle den Wert 1 haben, und anderenfalls eine 0.
Bei `cbit = 'x'` ist das Unicode-Bitmuster

```
0000000001111000
```

zu bearbeiten, und `1 << i & cbit` liefert z.B. bei `i` = 6 das Muster:

```
0000000001000000
```

Das von `1 << i & cbit` erzeugte Bitmuster hat den Typ **int** und kann daher mit der 0 verglichen werden:

```
(1 << i & cbit) != 0
```

Dieser logische Ausdruck wird im `i`-ten Schleifendurchgang genau dann wahr, wenn das korrespondierende Bit in der Binärdarstellung des untersuchten Zeichens den Wert 1 hat.

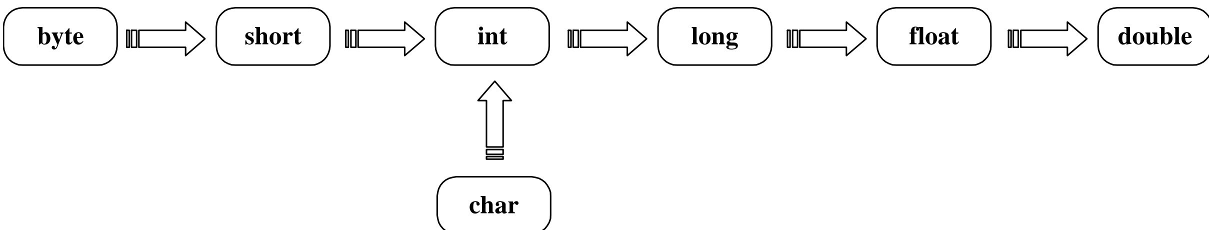
3.4.6 Typumwandlung (Casting)

Beim Auswerten des Ausdrucks

```
2.0/7
```

trifft der Divisions-Operator auf ein **double**- und auf ein **int**-Argument, so dass nach der Tabelle in Abschnitt 3.4.1 die Gleitkomma-Arithmetik zum Einsatz kommt. Dabei wird für das **int**-Argument eine **automatische (implizite) Wandlung** in den Datentyp **double** vorgenommen.

Java nimmt bei Bedarf für primitive Datentypen die folgenden **erweiternden Konvertierungen** zu einem allgemeineren Typ automatisch vor:



Im folgenden Programm wird u.a. die recht verblüffende automatische Konversion von **char** nach **int** demonstriert:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class ImplCast {
    public static void main(String[] args) {
        int i = 2, j = 3;
        System.out.println('x'/2);

        System.out.println("Wg. i=" + i + " und j=" + j);
        System.out.println("ist i>j " + (i>j));
    }
}</pre></td><td><pre>60
Wg. i=2 und j=3
ist i>j false</pre></td></tr>
</table>

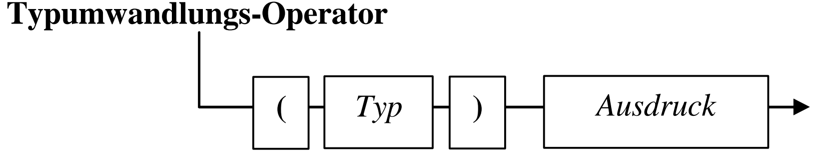
Da String in Java kein primitiver Datentyp ist, sondern eine Klasse, fehlt in obiger Abbildung die automatische Wandlung beliebiger „druckbarer" Datentypen in Strings, die wir beim Aufruf der **print**()- bzw. **println**()-Methode schon mehrfach benutzt haben.

Gelegentlich gibt es gute Gründe, über den **Casting-Operator** eine *explizite* Typumwandlung zu erzwingen. Im folgenden Programm wird z.B. mit `(int) 'x'` die **int**-erpretation des (aus Abschnitt 3.4.5 bekannten) Bitmusters zum kleinen „x" ausgegeben, damit Sie nachvollziehen können, warum das letzte Programm beim „Halbieren" dieses Zeichens auf den Wert 60 kam:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class ExplCast {
    public static void main(String[] args) {
        double a = 3.14159;
        System.out.println((int) 'x');
        System.out.println((int) a);
        System.out.println((int) 1.45367e50);
    }
}</pre></td><td><pre>120
3
2147483647</pre></td></tr>
</table>

In der zweiten Ausgabeanweisung des Beispielprogramms wird per Casting-Operation der ganzzahlige Anteil eines **double**-Wertes ermittelt.
Wie die dritte Ausgabe zeigt, sind bei einer explizit angeforderten **einschränkenden Konvertierung** schwere Programmierfehler möglich, wenn die Wertebereiche der beteiligten Variablen bzw. Datentypen nicht beachtet werden.

Die Java-Syntax zur expliziten Typumwandlung wurde unverändert von der Programmiersprache C übernommen:



3.4.7 Zuweisungsoperatoren

Bei den ersten Erläuterungen zu Wertzuweisungen blieb aus didaktischen Gründen unerwähnt, dass in Java eine Wertzuweisung als *Ausdruck* aufgefasst wird, dass wir es also mit einem binären (zweistelligen) Operator „**=**“ zu tun haben, für den folgende Regeln gelten:

- Auf der linken Seite muss eine Variable stehen.
- Auf der rechten Seite muss ein Ausdruck stehen, der einen zum Variablentyp kompatiblen Wert liefert.
- Der zugewiesene Wert stellt auch den Ergebniswert des Ausdrucks dar.

Analog zum Verhalten des Inkrement- bzw. Dekrementoperators sind auch beim Zuweisungsoperator zwei Effekte zu unterscheiden:

- Die als linkes Argument fungierende Variable wird verändert.
- Es wird ein Wert für den Ausdruck produziert.

In folgendem Beispiel fungiert ein Zuweisungsausdruck als Parameter für einen **println**()-Methodenaufruf:

Quellcode	Ausgabe

`class Prog {`
`    public static void main(String[] args) {`
`    int ivar = 13;`
`        System.out.println(ivar = 4711);`
`    }`
`}`
	`4711`

Beim Auswerten des Ausdrucks entsteht der an **println()** zu übergebende Wert *und* die Variable `ivar` wird verändert.

Selbstverständlich kann eine Zuweisung auch als Operand in einen übergeordneten Ausdruck integriert werden, z.B.:

Quellcode	Ausgabe

`class AssignOp {`
`    public static void main(String[] args) {`
`        int i = 2, j = 4;`
`        i = j = j * i;`
`        System.out.println(i + " " + j);`
`    }`
`}`
	`8 8`

Weil bei mehrfachem Auftreten des Zuweisungsoperators eine Abarbeitung von **rechts nach links** erfolgt (vgl. Tabelle in Abschnitt 3.4.9), passiert im letzten Beispiel folgendes:

- `j * i` liefert das Ergebnis 8, das der Variablen `j` zugewiesen wird und gleichzeitig den Wert des Ausdrucks `j = j * i` darstellt.
- In der zweiten Zuweisung (bei Betrachtung von rechts nach links) wird der Wert des Ausdrucks `j = j * i` an die Variable `i` übergeben.

Wie wir spätestens seit Abschnitt 3.2.2 wissen, stellt ein Zuweisungsausdruck (z.B. `a = 3`) bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für die

die Prä- und Postinkrementausdrücke (vgl. Abschnitt 3.4.1) sowie für Methodenaufrufe, jedoch **nicht** für die anderen Ausdrücke, die in Abschnitt 3.4 vorgestellt werden.

Für die häufig benötigten Zuweisungen nach dem Muster `j = j * i` (eine Variable erhält einen neuen Wert, an dessen Konstruktion sie selbst mitwirkt), bietet Java spezielle Zuweisungsoperatoren für Schreibfaule, die gelegentlich auch als **Aktualisierungsoperatoren** bezeichnet werden. In der folgenden Tabelle steht *var* für eine numerische Variable (**byte**, **short**, **int**, **long**, **char**, **float** oder **double**) und *expr* für einen typkompatiblen Ausdruck:

Operator	Bedeutung	Beispiel	
		Programmfragment	Neuer Wert[1]
var **+=** *expr*	*var* erhält den neuen Wert *var* + *expr*.	`int i = 2;` `i += 3;`	5
var **-=** *expr*	*var* erhält den neuen Wert *var* − *expr*.	`int i = 10, j = 3;` `i -= j*j;`	1
var ***=** *expr*, *var* **/=** *expr*	*var* erhält den neuen Wert *var* * *expr* bzw. *var* / *expr*.	`int i = 10;` `i /= 5;`	2
var **%=** *expr*	*var* erhält den neuen Wert *var* % *expr*.	`int i = 10;` `i %= 5;`	0

[1] Die Spalte zeigt jeweils den neuen Wert der aktualisierten Variablen `i`.

3.4.8 Konditionaloperator

Wiederum eher für Schreibfaule als für Anfänger geeignet ist der **Konditionaloperator**, mit dessen Hilfe sich die bedingungsabhängige Entscheidung zwischen zwei Ausdrücken sehr kompakt formulieren lässt. Eine Besonderheit des Konditionaloperators besteht darin, dass er *drei* Argumente verarbeitet, welche durch die Zeichen **?** und **:** sortiert werden. In der folgenden Tabelle steht *la* für einen logischen Ausdruck, *expr1* und *expr2* stehen für Ausdrücke mit einem zuweisungskompatiblem Typ:

Syntax	Erläuterung	Beispiel	
		Programmfragment	Ausgabe

la **?** *expr1* **:** *expr2*	Ist der logische Ausdruck *la* wahr, liefert der Konditional-Operator den Wert von *expr1*, anderenfalls den Wert von *expr2*.	`int i = 12, j = 3, k;`
		`k = (i > j)?i:j;`
		`System.out.println(k);`
			12

3.4.9 Auswertungsreihenfolge

Bisher haben wir Ausdrücke mit *mehreren* Operatoren und das damit verbundene Problem der Auswertungs*reihenfolge* nach Möglichkeit gemieden. Nun werden die Regeln vorgestellt, nach denen der Java-Compiler komplexe Ausdrücke mit mehreren Operatoren auswertet:

1) Priorität

Zunächst entscheidet die Priorität der Operatoren (siehe Tabelle unten) darüber, in welcher Reihenfolge die Auswertung vorgenommen wird. Z.B. hält sich Java bei numerischen Ausdrücken an die mathematische Regel „Punktrechnung geht vor Strichrechnung“.

2) Auswertungsrichtung

Bei gleicher Priorität wird ein zweites Attribut der beteiligten Operatoren relevant: ihre Auswertungsrichtung[1]. In der Regel werden gleichrangige Operatoren von Links nach Rechts ausgewertet, doch haben wir beim Zuweisungsoperator schon eine Ausnahme kennen gelernt. Um jede Unklarheit zu vermeiden, ist in Java für Operatoren gleicher Priorität stets auch die selbe Auswertungsrichtung festgelegt.

Bei manchen Operatoren gilt das mathematische Assoziativitätsgesetz, so dass die Reihenfolge der Auswertung irrelevant ist, z.B.:

$$(3 + 2) + 1 = 6 = 3 + (2 + 1)$$

Anderen Operatoren fehlt diese nette Eigenschaft, z.B.:

$$(3 - 2) - 1 = 0 \neq 3 - (2 - 1) = 2$$

Weil beim binären Minus-Operator die Auswertung von links nach rechts erfolgt, hat der Ausdruck

$$3 - 2 - 1$$

in Java den Wert 0.

3) Klammern

Wenn die aus obigen Regeln resultierende Auswertungsfolge nicht zum gewünschten Ergebnis führt, kann mit runden Klammern steuernd eingegriffen werden: Die eingeklammerten Teilausdrücke werden „von innen nach außen“ ausgewertet.

In der folgenden Tabelle sind die bisher behandelten Operatoren in absteigender Priorität (von oben nach unten) mit Auswertungsrichtung aufgelistet. Gruppen von Java-Operatoren mit gleicher Priorität sind durch fette horizontale Linien begrenzt. Sie verfügen dann stets auch über die selbe Auswertungsrichtung. In der **Operanden**-Spalte werden die zulässigen Datentypen der Argument-Ausdrücke mit Hilfe der folgenden Platzhalter beschrieben:

N	Ausdruck mit numerischem Datentyp (**byte**, **short**, **int**, **long**, **char**, **float**, **double**)
I	Ausdruck mit integralem (ganzzahligem) Datentyp (**byte**, **short**, **int**, **long**, **char**)
L	logischer Ausdruck
B	Ausdruck mit beliebigem kompatiblem Datentyp
S	String
V	Variable
V_n	Variable mit numerischem Datentyp (**byte**, **short**, **int**, **long**, **char**, **float**, **double**)

[1] In der Informatik spricht man auch von *Bindungsrichtung* oder *Assoziativität*.

Operator	Bedeutung	Operanden	Richtung
()	Methodenaufruf		$L \rightarrow R$
!	Negation	L	$L \leftarrow R$
++, --	Prä- oder Postinkrement bzw. -dekrement	V_n	
-	Vorzeichenumkehr	N	
(*type*)	Typumwandlung	B	
*, /	Punktrechnung	N, N	$L \rightarrow R$
%	Modulo	N, N	
+, -	Strichrechnung	N, N	$L \rightarrow R$
+	Stringverkettung	S, B oder B, S	
<<, >>	Links- bzw. Rechts-Shift	I, I	$L \rightarrow R$
>, <, >=, <=	Vergleichsoperatoren	N, N	$L \rightarrow R$
==, !=	Gleichheit, Ungleichheit	B, B	$L \rightarrow R$
&	Bitweises UND	I, I	$L \rightarrow R$
&	Logisches UND (mit unbedingter Auswertung)	L, L	
^	Exklusives logisches ODER	L, L	$L \rightarrow R$
|	Bitweises ODER	I, I	$L \rightarrow R$
|	Logisches ODER (mit unbedingter Auswertung)	L, L	
&&	Logisches UND (mit bedingter Auswertung)	L, L	$L \rightarrow R$
||	Logisches ODER (mit bedingter Auswertung)	L, L	$L \rightarrow R$
? :	Konditionaloperator	L, B, B	$L \leftarrow R$
=	Wertzuweisung	V, B	

Operator	Bedeutung	Operanden	Richtung
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung	V_n, N	L ← R

Im Anhang finden Sie eine erweiterte Version dieser Tabelle, die zusätzlich alle Operatoren enthält, die im weiteren Verlauf des Kurses noch behandelt werden.

3.4.10 Über- und Unterlauf

Wie wir inzwischen wissen, haben die numerischen Datentypen einen bestimmten Wertebereich. Liegt das Ergebnis eines Ausdrucks außerhalb des Zieltyp-Wertebereichs, ist ein korrektes Speichern nicht möglich. Bei der Auswertung eines komplexen Ausdrucks kann ein Wertebereichs-Problem auch schon „unterwegs" auftreten. Im betroffenen Programm ist mit einem mehr oder weniger gravierenden Fehlverhalten zu rechnen, so dass ein solcher Über- oder Unterlauf unbedingt vermieden bzw. rechtzeitig diagnostiziert werden muss.

Leider dürfen Sie von einem Java - Programm *nicht* erwarten, dass es im Falle einer „übergelaufenen" **Ganzzahlvariablen** mit einem Laufzeitfehler seine Tätigkeit einstellt, um Schlimmeres zu vermeiden. Das folgende Programm

```
class ORI {
    public static void main(String[] args) {
        int i = 2147483647, j = 5, k;
        k = i + j;
        System.out.println(i+" + "+j+" = "+k);
    }
}
```

liefert ohne jede Warnung oder Selbstzweifel das fragwürdige Ergebnis:

```
2147483647 + 5 = -2147483644
```

Oft kann ein Überlauf durch Wahl eines geeigneten Datentyps verhindert werden. Im letzten Beispiel sollte für die kritische Variable an Stelle von **int** der Typ **long** verwendet werden:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Verz {
    public static void main(String[] args) {
        long i = 2147483647, j = 5, k;
        k = i + j;
        System.out.println(i+" + "+j+" = "+k);
    }
}</pre></td><td><pre>2147483647 + 5 = 2147483652</pre></td></tr>
</table>

Ein Überlauf kann auch bei **Gleitkommavariablen** auftreten, wenngleich hier der zulässige Wertebereich weit größer ist. Das Programm

```
class ORD {
    public static void main(String[] args) {
        double bigd = Double.MAX_VALUE, smalld = Double.MIN_VALUE;
        System.out.println(bigd *= 10.0);
        System.out.println(bigd -= bigd);
        System.out.println(smalld /= 10.0);
    }
}
```

liefert die Ausgabe:

```
Infinity
NaN
0.0
```

Bei den ersten beiden Ausdrücken treten spezielle Gleitkomma-Werte auf:

Infinity positiv Unendlich
NaN undefinierter Wert („Not a Number“)
Im Beispiel hat sich Java zu Recht geweigert, „∞ - ∞“ zu berechnen.

Über die finalisierten Klassenvariablen **Double.MAX_VALUE** und **Double.MIN_VALUE** der Klasse **Double** aus dem Paket **java.lang** kann man den betragsmäßig größten bzw. kleinsten Wert abfragen, der in einer **double**-Variablen gespeichert werden kann. Später folgen nähere Erläuterungen zur Klasse **Double**.

Insgesamt ist festzuhalten, dass in Java bei der *Gleitkomma*-Arithmetik kritische Werte sinnvoll behandelt bzw. protokolliert werden, während bei der *Ganzzahl*-Arithmetik stillschweigend mit Unfug weitergerechnet wird. Wenn mit kritischen Größenordnungen zu rechnen ist, sollten daher Gleitkommavariablen zum Einsatz kommen.

Bei Gleitkommavariablen ist auch ein **Unterlauf** möglich, wobei eine Zahl mit sehr kleinem Betrag (bei **double**: $< 4.9 \cdot 10^{-324}$) nicht mehr dargestellt werden kann. Im diesem Fall rechnet ein Java - Programm mit dem Wert 0.0 weiter, was in der Regel akzeptabel ist.

Bei der Kalkulation des Unter- bzw. Überlaufrisikos darf man sich nicht auf die Endergebnisse von Ausdrücken beschränken, sondern muss auch alle Zwischenschritte berücksichtigen. Im folgenden Beispiel wird ein **double**-Argument durch 10 dividiert und anschließend logarithmiert. Für das Argument $1.5 \cdot 10^{-323}$ wird als Funktionswert erwartet:

$$\log\left(\frac{1.5 \cdot 10^{-323}}{10}\right) = \log(1.5 \cdot 10^{-323}) - \log(10) \approx -743.34 - 2.30 = -745.64$$

Wegen eines Unterlaufs während der Berechnung kommt das Programm aber zum Ergebnis -∞:

Quellcode:

```
class Prog {
  public static void main(String[] args) {
    double doub = 1.5e-323;
    System.out.println(Math.log(doub/10));
    System.out.println((Math.log(doub)-Math.log(10)));
  }
}
```

Ausgabe:

```
-Infinity
-745.6440447257072
```

3.4.11 Übungsaufgaben zu Abschnitt 3.4

1) Welche Werte und Datentypen besitzen die folgenden Ausdrücke?

```
6/4*2.0
(int)6/4.0*3
3*5+8/3%4*5
```

2) Welche Werte erhalten in folgendem Programmfragment die Integer-Variablen `erg1` und `erg2`?

```
int i = 2, j = 3, erg1, erg2;
erg1 = (i++ == j ? 7 : 8) % 2;
erg2 = (++i == j ? 7 : 8) % 2;
```

3) Welche Wahrheitsweite erhalten in folgendem Programm die boolschen Variablen `la1` **bis** `la4`?

```
class OpPref {
    public static void main(String[] args) {
        boolean la1, la2, la3, la4;
        int i = 3;
        char c = 'n';

        la1 = (2 > 3) && (2 == 2) ^ (1 == 1);
        System.out.println(la1);

        la2 = ((2 > 3) && (2 == 2)) ^ (1 == 1);
        System.out.println(la2);

        la3 = i > 0 && i * 4 == 012;
        System.out.println(la3);

        la4 = !(i > 0 || c == 'j');
        System.out.println(la4);
    }
}
```

Tipp: Die Negation von zusammengesetzten Ausdrücken ist recht unangenehm zu lesen. Mit Hilfe der Regeln von **DeMorgan** kommt man zu äquivalenten Ausdrücken, die leichter zu interpretieren sind:

!(*la1* && *la2*) = !*la1* || !*la2*
!(*la1* || *la2*) = !*la1* && !*la2*

4) Erstellen Sie ein Java-Programm, das den Exponentialfunktionswert e^x zu einer vom Benutzer eingegebenen Zahl *x* bestimmt und ausgibt, z.B.:

Eingabe: `Argument: 1`
Ausgabe: `exp(1.0) = 2.7182818284590455`

Hinweise:

- Suchen Sie mit Hilfe der SDK-Dokumentation zur Klasse **Math** im API-Paket **java.lang** eine passende Methode.
- Zum Einlesen des Argumentes können Sie die Methode `gdouble()` aus unserer Bequemlichkeitsklasse `Simput` verwenden, die eine vom Benutzer (mit oder ohne Dezimalpunkt) eingetippte und mit **Enter** quittierte Zahl als **double**-Wert abliefert.

5) Kann bei Ganzzahlvariablen ein Unterlauf auftreten?

6) Erstellen Sie ein Java-Programm, das einen DM-Betrag entgegen nimmt und diesen in Euro konvertiert. In der Ausgabe sollen ganzzahlige, korrekt gerundete Werte für Euro und Cent erscheinen, z.B.:

Eingabe: `DM-Betrag: 321`
Ausgabe: `164 Euro und 12 Cent`

Hinweise:

- Umrechnungsfaktor: 1 Euro = 1.95583 DM
- Zum Einlesen des DM-Betrages können Sie die Methode `gdouble()` aus unserer Bequemlichkeitsklasse `Simput` verwenden.

3.5 Anweisungen

Wir haben uns in diesem Kurs zunächst mit (lokalen) **Variablen** und primitiven **Datentypen** vertraut gemacht. Dann haben wir gelernt, aus Variablen, Literalen und Methodenaufrufen mit Hilfe von **Operatoren** komplexe **Ausdrücken** zu bilden. Diese wurden entweder mit der Methode **System.out.println()** auf dem Bildschirm ausgegeben oder in Wertzuweisungen verwendet. Unsere Beispielprogramme bestanden im Wesentlichen aus einer Sequenz von **Anweisungen**, die wir nach bedarfsgerecht eingeführten Regeln gebildet haben. Nun werden wir uns systematisch mit dem allgemeinen Begriff einer Java-Anweisung und vor allem mit wichtigen Spezialfällen befassen.

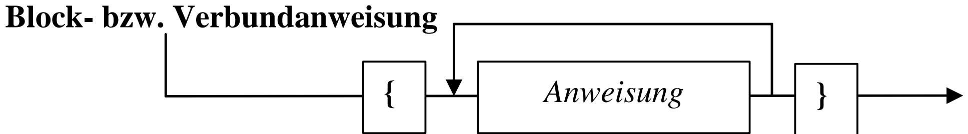
3.5.1 Überblick

Ausführbare Programmteile, die in Java stets als Methoden von Klassen zu realisieren sind, bestehen aus Anweisungen (engl. statements), die nacheinander ausgeführt werden.

Am Ende von Abschnitt 3.5 werden Sie die folgenden Typen von Anweisungen kennen:

- **Variablendeklarationsanweisungen**
  Die Variablendeklarationsanweisung wurde schon in Abschnitt 3.2.2 eingeführt.
  Beispiel: `int i = 1, k;`
- **Ausdrucksanweisungen**
  Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
    - Wertzuweisungen (vgl. Abschnitte 3.2.2 und 3.4.7)
      Beispiel: `k = i + j;`
    - Prä- bzw. Postinkrement- oder -dekrementoperationen
      Beispiel: `i++;`
      Hier ist nur der „Nebeneffekt" des Ausdrucks `i++` von Bedeutung. Sein Wert bleibt ungenutzt.
    - Methodenaufrufe
      Beispiel: `System.out.println(la1);`
      Besitzt die aufgerufene Methode einen Rückgabewert (siehe unten), wird dieser ignoriert.
- **Leere Anweisung**
  Beispiel: `;`
  Die durch ein einsames (nicht anderweitig eingebundenes) Semikolon ausgedrückte *leere* Anweisung hat keinerlei Effekte und kann dann verwendet werden, wenn die Syntax eine Anweisung verlangt, aber nichts geschehen soll.
- **Anweisungen zur Ablaufsteuerung**
  Die bisherigen Beispielprogramme bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmlauf alle nacheinander ausgeführt wurden. Oft möchte man jedoch z.B.
    - die Ausführung einer Anweisung oder eines Anweisungsblocks von einer Bedingung abhängig machen
    - oder einen Anweisungsblock wiederholt ausführen lassen.

  Für solche Zwecke stellt Java etliche Verzweigungs- bzw. Wiederholungsanweisungen zur Verfügung, die bald ausführlich behandelt werden.
- **Anweisungsblöcke**
  Einen Block von Anweisungen, die durch geschweifte Klammern zusammengefasst bzw. abgegrenzt werden, bezeichnet man als **Verbund- bzw. Blockanweisung**:



Weil ein Block wiederum eine Anweisung darstellt, ist er syntaktisch überall zugelassen, wo eine Anweisung erlaubt ist.
Wie gleich näher erläutert wird, fasst man z.B. *dann* mehrere Abweisungen zu einem Block zusammen, wenn diese Anweisungen unter einer gemeinsamen Bedingung ausgeführt werden sollen. Es wäre ja sehr unpraktisch, die selbe Bedingung für jede betroffene Anweisung wiederholen zu müssen.
In Abschnitt 3.2.3 haben wir uns im Zusammenhang mit dem Gültigkeitsbereich für lokale Variablen bereits mit Anweisungsblöcken beschäftigt.

Blockanweisungen sowie Anweisungen zur Ablaufsteuerung enthalten andere Anweisungen und werden daher auch als **zusammengesetzte Anweisungen** bezeichnet.

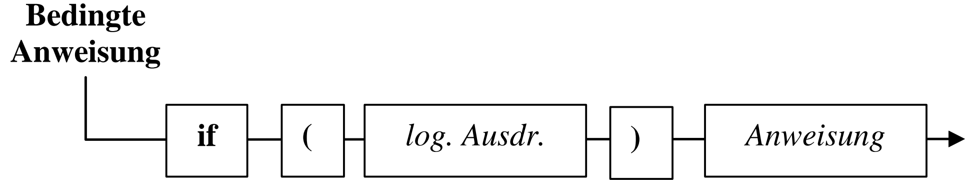
Anweisungen werden durch ein Semikolon abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

Im weiteren Verlauf von Abschnitt 3.5 werden Anweisungen zur Ablaufsteuerung behandelt, die man oft auch als *Kontrollstrukturen* bezeichnet.

3.5.2 Bedingte Anweisung und Verzweigung

3.5.2.1 if-Anweisung

Mit der folgenden Syntax sorgt man dafür, dass eine Anweisung nur dann ausgeführt wird, wenn ein logischer Ausdruck den Wert **true** annimmt:



In diesem Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` den Wert 0 besitzt:

```
if (anz == 0)
   System.out.println("Die Anzahl muss > 0 sein!");
```

Der Zeilenumbruch zwischen dem logischen Ausdruck und der (Unter-)Anweisung dient nur der Übersichtlichkeit und ist für den Compiler irrelevant.

Soll auch etwas passieren, wenn der logische Ausdruck den Wert **false** besitzt, erweitert man die **if**-Anweisung um eine **else**-Klausel.

Zur Beschreibung der **if-else** - Anweisung wird an Stelle eines Syntaxdiagramms eine alternative Darstellungsform gewählt, die sich am typischen Java-Quellcode-Layout orientiert:

if (*logischer Ausdruck***)**
    Anweisung 1
else
    Anweisung 2

Wie bei den Syntaxdiagramme gilt auch für diese Form der Syntaxbeschreibung:

- Für **terminale Sprachbestandteile**, die exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind durch *kursive* Schrift gekennzeichnet.

Während die Syntaxbeschreibung im Quellcode-Layout sehr übersichtlich ist, bietet das Syntaxdiagramm den Vorteil, bei komplizierter, variantenreicher Syntax alle zulässigen Formulierungen kompakt und präzise als Pfade durch das Diagramm zu beschreiben.

Im folgenden **if-else** - Beispiel wird der natürliche Logarithmus zu einer Zahl geliefert, falls diese positiv ist. Anderenfalls erscheint eine Fehlermeldung mit Alarmton (Unicode-Escape-Sequenz `\u0007`). Das Argument wird vom Benutzer über die `Simput`-Methode `gdouble()` erfragt (vgl. Abschnitt 3.3).

Quellcode	Ausgabe

<pre>class Verz {
   public static void main(String[] args) {
      System.out.print("Argument: ");
      double arg = Simput.gdouble();
      if (arg > 0)
         System.out.println("ln("+arg+") = "
                         +Math.log(arg));
      else
         System.out.println("\u0007Argument <= 0!");
   }
}</pre>
	<pre>Argument: 2
	ln(2.0) =
	0.6931471805599453</pre>

Als bedingt auszuführende Anweisung ist durchaus wiederum eine **if**- bzw. **if-else**-Anweisung erlaubt, so dass sich komplexere Fallunterscheidungen formulieren lassen:

if (*logischer Ausdruck 1***)**
    Anweisung 1
else if (*logischer Ausdruck 2***)**
    Anweisung 2
  . . .
  . . .
else if (*logischer Ausdruck k***)**
    Anweisung k
else
    Default-Anweisung

Enthält die letzte **if**-Anweisung eine **else**-Klausel, so wird die zugehörige Anweisung ausgeführt, wenn alle logischen Ausdrücke den Wert **false** haben. Die Wahl der Bezeichnung *Default-Anweisung* in der Syntaxdarstellung orientierte sich an der im Anschluss vorzustellenden **switch**-Anweisung.

Beim Schachteln von bedingten Anweisungen kann es zum genannten **dangling-else** - Problem kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Codefragment lässt die Einrückung vermuten, dass der Programmierer die **else**-Klausel der *ersten* **if**-Anweisung zuordnen wollte:

```
if (i > 0)
   if (j > i) k = j;
else
   k = i;
```

Der Compiler ordnet sie jedoch dem in Aufwärtsrichtung nächstgelegenen **if** zu, das nicht durch Blockklammern abgeschottet ist und noch keine **else**-Klausel besitzt. Im Beispiel bezieht er die **else**-Klausel also auf die *zweite* **if**-Anweisung.

Mit Hilfe von Blockklammern (nötigenfalls auch für eine einzelne Anweisung) kann die gewünschte Zuordnung erzwungen werden:

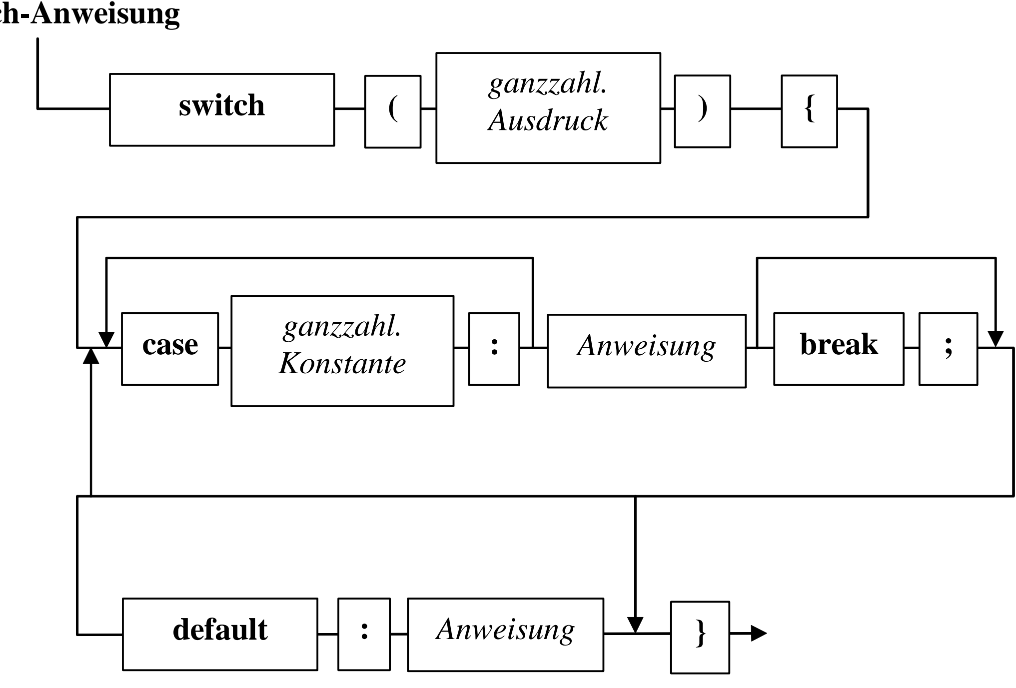
```
if (i > 0)
   {if (j > i) k = j;}
else
   k = i;
```

Alternativ könnte man auch dem zweiten **if** eine **else**-Klausel spendieren und dabei eine leere Anweisung verwenden:

```
if (i > 0)
   if (j > i) k = j;
   else;
else
   k = i;
```

3.5.2.2 switch-Anweisung

Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit von einem ganzzahligen Ausdruck vom Typ **byte**, **short**, **char** oder **int** (nicht **long**!) vorgenommen werden soll, dann ist die Mehrfachauswahl per **switch**-Anweisung weitaus handlicher als eine verschachtelte **if**-**else**-Konstruktion.
Der Genauigkeit halber wird die **switch**-Anweisung mit einem Syntaxdiagramm beschrieben. Wer die Syntaxbeschreibung im Quellcode-Layout bevorzugt, kann ersatzweise einen Blick auf das gleich folgende Beispiel werfen.



Stimmt der Wert des Ausdrucks mit der ganzzahligen Konstanten (finalisierte Variable oder Literal) zu einer **case**-Marke überein, dann wird die zugehörige Anweisung ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisung. Nach dem Ausführen einer „angesprungenen" Anweisung wird die **switch**-Konstruktion jedoch nur dann verlassen, wenn der Fall mit einer **break**-Anweisung abgeschlossen wird. Ansonsten werden auch noch die Anweisungen der nächsten Fälle (ggf. inkl. **default**) ausgeführt, bis der „Durchfall" nach unten entweder durch eine **break**-Anweisung gestoppt wird, oder die **switch**-Anweisung endet. Mit dem etwas gewöhnungsbedürftigen **Durchfall**-Prinzip

kann man für geeignet angeordnete Fälle sehr elegant kumulative Effekte kodieren, aber auch ärgerliche Programmierfehler durch vergessene **break**-Anweisungen produzieren.
Soll für mehrere Auswahlwerte dieselbe Anweisung ausgeführt werden, setzt man die zugehörigen **case**-Marken unmittelbar hintereinander und lässt die Anweisung auf die letzte Marke folgen. Leider gibt es keine Möglichkeit, eine *Serie* von Fällen durch Angabe der Randwerte festzulegen.
Im folgenden Beispielprogramm wird die Persönlichkeit des Benutzers mit Hilfe seiner Farb- und Zahlpräferenzen analysiert. Während bei einer Vorliebe für Rot oder Schwarz die Diagnose sofort fest steht, wird bei den restlichen Farben auch die Lieblingszahl berücksichtigt:

```
class PerST {
   public static void main(String[] args) {
      char farbe = args[0].charAt(0);
      int  zahl = Character.getNumericValue(args[1].charAt(0));

      switch (farbe) {
         case 'r':  System.out.println("Sie sind ein emotionaler Typ."); break;
         case 'g':
         case 'b': {
                       System.out.println("Sie scheinen ein sachlicher Typ zu sein");
                       if (zahl%2 == 0)
                          System.out.println("Sie haben einen geradlinigen Charakter.");
                       else
                          System.out.println("Sie machen wohl gerne krumme Touren.");
                    }
                    break;
         case 's':  System.out.println("Nehmen Sie nicht Alles so tragisch."); break;
         default:   System.out.println("Offenbar mangelt es Ihnen an Disziplin.");
      }
   }
}
```

Das Programm `PerST` demonstriert nicht nur die **switch**-Anweisung, sondern auch die Verwendung von **Kommandozeilen-Optionen** über den **String[]**-Parameter der **main()**-Funktion. Benutzer des Programms sollen beim Start ihre Lieblingsfarbe und ihre Lieblingszahl (aus dem Wertebereich von 0 bis 9) angeben, wobei die Farbe folgendermaßen durch einen Buchstaben zu kodieren ist:

r	für Rot
g	für Grün
b	für Blau
s	für Schwarz

Wer die Farbe Blau und die Zahl 7 bevorzugt, muss das Programm also folgendermaßen starten:

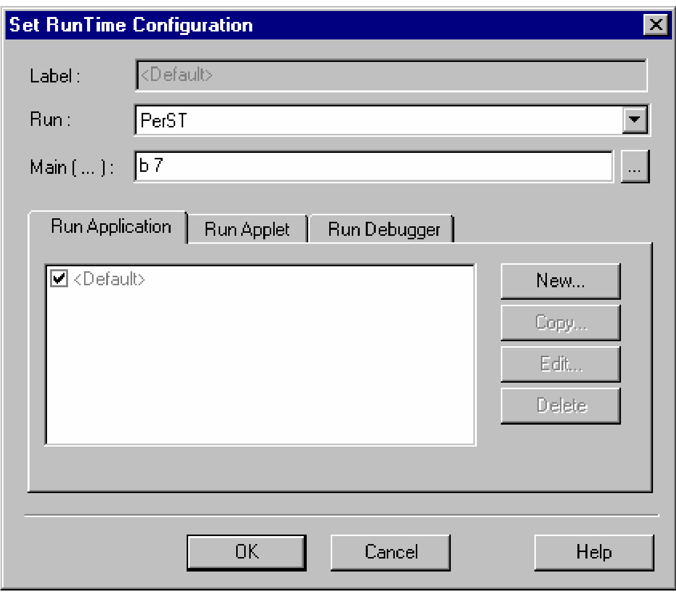
```
java PerST b 7
```

Man benötigt jeweils nur eine Java-Zeile, um die Kommandozeilen-Optionen in eine **char**- bzw. **int**-Variable zu übertragen. Die zugehörigen Erklärungen werden Sie in Kürze mit Leichtigkeit verstehen:

- Das erste Element des Zeichenketten-Feldes `args` (Nummer 0) wird (als Objekt der Klasse **String**) aufgefordert, die Methode **charAt()** auszuführen, welche das erste Zeichen abliefert. Im Programm wird dieses Zeichen der **char**-Variablen `farbe` zugewiesen.
- Analog wird auch zum zweiten Element des Zeichenketten-Feldes `args` (Nummer 1) das erste Zeichen ermittelt. Dieses wird mit Hilfe der Klassenmethode **getNumericValue()** aus der Klasse **Character** in eine Zahl vom Datentyp **int** gewandelt und anschließend der Variablen `zahl` zugewiesen.

Soll das Programm in der JCreator-Entwicklungsumgebung gestartet werden, können die „Kommandozeilen"-Optionen folgendermaßen vereinbart werden:

- Menübefehl **Build > Runtime Configuration**
- In der Dialogbox **RunTime Configuration** markiert man **<default>** und klickt auf **Edit**.
- In der Dialogbox **Set RunTime Configuration** trägt man die Optionen ins Textfeld **Main(...)** ein, z.B.:



3.5.3 Wiederholungsanweisung

Eine Wiederholungsanweisung (oder schlicht: Schleife) kommt zum Einsatz, wenn eine Anweisung *mehrfach* ausgeführt werden soll, wobei sich in der Regel schon der Gedanke daran verbietet, die Anweisung entsprechend oft in den Quelltext zu schreiben.
Bei leicht vereinfachender Betrachtung kann man hinsichtlich der Schleifensteuerung unterscheiden:

- **Zählergesteuerte Schleife** (**for**)
  Die Anzahl der Wiederholungen steht typischerweise schon vor Schleifenbeginn fest. Zur Ablaufsteuerung wird am Anfang eine Zählvariable initialisiert und dann bei jedem Durchlauf aktualisiert (z.B. inkrementiert). Die zur Schleife gehörige Anweisung wird ausgeführt, solange die Zählvariable eine festgelegte Grenze nicht überschreitet.
- **Bedingungsabhängige Schleife** (**while**, **do**)
  Bei jedem Schleifendurchgang wird eine Bedingung überprüft, und das Ergebnis entscheidet über das weitere Vorgehen:
    - **true**: die zur Schleife gehörige Anweisung ein weiteres mal ausführen
    - **false**: Schleife beenden

  Bei der *kopf*gesteuerten **while**-Schleife wird die Bedingung *vor Beginn* eines Durchgangs geprüft, bei der *fuß*gesteuerten **do**-Schleife hingegen *am Ende*.

Die gesamte Konstruktion aus Schleifensteuerung und (Verbund-)anweisung stellt in syntaktischer Hinsicht *eine* zusammengesetzte Anweisung dar.

3.5.3.1 Zählergesteuerte Schleife (for)

Die Anweisung einer **for**-Schleife wird ausgeführt, solange eine Bedingung erfüllt ist, die normalerweise auf eine ganzzahlige Indexvariable Bezug nimmt. Auf das Schlüsselwort **for** folgt der von runden Klammern umgebene Schleifenkopf, wo die Initialisierung der Indexvariablen, die Spezifikation der Fortsetzungsbedingung und die Aktualisierungsvorschrift untergebracht werden können. Am Ende steht die wiederholt auszuführende (Verbund-)Anweisung:

for (*Initialisierung*; *Bedingung*; *Aktualisierung*)   *Anweisung*

Zu den drei Bestandteilen des Schleifenkopfes sind einige Erläuterungen erforderlich, wobei hier etliche weniger typische bzw. sinnvolle Möglichkeiten weggelassen werden:

- ***Initialisierung***
  In der Regel wird man sich auf *eine* Indexvariable beschränken und dabei einen Ganzzahl-Typ wählen. Diese Variable wird im Schleifenkopf auf einen Startwert gesetzt und nötigenfalls dabei auch deklariert, z.B.:

  Quellcode

  ```
  class ForLoop {
      public static void main(String[] args) {
          int n = 10;
          double sq = 0.0;
          for (int i = 1; i <= n; i++) sq += i*i;
          System.out.println("Quadratsumme = " + sq);
      }
  }
  ```

  Ausgabe

  ```
  Quadratsumme = 385.0
  ```

  Deklarationen und Initialisierungen werden vor dem ersten Durchlauf ausgeführt.
  Eine im Initialisierungsteil deklarierte Variable ist lokal bzgl. der **for**-Schleife, steht also nur in deren Anweisung(sblock) zur Verfügung.
- ***Bedingung***
  Üblicherweise wird eine Ober- oder Untergrenze für die Indexvariable gesetzt, doch erlaubt Java beliebige logische Ausdrücke.
  Die Bedingung wird *zu Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so findet eine weitere Wiederholung des Anweisungsteils statt, anderen falls wird die **for**-Anweisung verlassen. Folglich kann es auch passieren, dass überhaupt kein Durchlauf zustande kommt.
- ***Aktualisierung***
  Die Aktualisierung wird *am Ende* eines Schleifendurchgangs (nach Ausführung der Anweisung) vorgenommen.

Zu den (zumindest stilistisch) bedenklichen Konstruktionen, die der Compiler klaglos umsetzt, gehören **for**-Schleifenköpfe ohne Initialisierung oder ohne Aktualisierung, wobei die trennenden Strichpunkte trotzdem zu setzen sind. In solchen Fällen ist die Umlaufzahl einer **for**-Schleife natürlich nicht mehr aus dem Schleifenkopf abzulesen. Dies gelingt auch dann nicht, wenn in der Schleifenanweisung eine Indexvariable modifiziert wird.

3.5.3.2 Bedingungsabhängige Schleifen

Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift dieses Abschnitts nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Java - Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems benutzt.
Unter der aktuellen Abschnittsüberschrift diskutiert man traditionsgemäß die **while**- und die **do**-Schleife.

3.5.3.2.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschreiben kann: Wer im Kopf einer **for**-Schleife auf Initialisierung und Aktualisierung verzichten möchte, ersetzt besser das Schlüsselwort **for** durch **while** und erhält dann folgende Syntax:

while (*Bedingung*)
    Anweisung

Wie bei der **for**-Anweisung wird die Bedingung *zu Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so wird der Anweisungsteil ausgeführt, anderenfalls wird die **while**-Anweisung verlassen, eventuell noch vor dem ersten Durchgang.

Im obigen Beispielprogramm kann man nach minimalen Änderungen auch eine **while**-Schleife verwenden:

Quellcode	Ausgabe

<pre>class WhiLoop {
   public static void main(String[] args) {
      int n = 10, i = 1;
      double sq = 0.0;
      while (i <= n) {
          sq += i*i;
          i++;
      }
      System.out.println("Quadratsumme = " + sq);
   }
}</pre>
	Quadratsumme = 385.0

3.5.3.2.2 do-Schleife

Bei der **do**-Schleife wird die Fortsetzungsbedingung *am Ende* der Schleifendurchläufe geprüft, so dass wenigstens *ein* Durchlauf stattfindet:

do
  Anweisung
while (*Bedingung*)**;**

do-Schleifen werden seltener benötigt als **while**-Schleifen, sind aber z.B. dann von Vorteil, wenn man vom Benutzer eine Eingabe mit bestimmten Eigenschaften einfordern möchte. In folgendem Codesegment kommt die statische Methode `gchar()` aus der Klasse `Simput` zum Einsatz, die ein vom Benutzer eingetipptes und mit **Enter** quittiertes Zeichen als **char**-Wert abliefert:

```
char antwort;
do {
   System.out.println("Soll das Programm beendet werden (j/n)? ");
   antwort = Simput.gchar();
} while (antwort != 'j' && antwort != 'n' );
```

Bei einer **do**-Schleife mit Anweisungs*block* sollte man die **while**-Klausel unmittelbar hinter die schließende Blockklammer setzen (in derselben Zeile), um sie optisch von einer selbständigen **while**-Anweisung abzuheben.

Bei jeder Wiederholungsanweisung (**for**, **while** oder **do**) kann es in Abhängigkeit von der verwendeten Bedingung passieren, dass der Anweisungsteil unendlich oft ausgeführt wird. Endlosschleifen sind als gravierende Programmierfehler unbedingt zu vermeiden. Befindet sich ein Programm in diesem Zustand muss es mit Hilfe des Betriebssystems abgebrochen werden, bei unseren Konsolenanwendungen unter Windows z.B. über die Tastenkombination **<Strg>+<C>**.

3.5.3.3 Schleifen(durchgänge) vorzeitig beenden

Mit der **break-**Anweisung, die uns schon als Bestandteil der **switch**-Anweisung begegnet ist, kann eine **for**-, **while**- oder **do**-Schleife vorzeitig verlassen werden. Mit der **continue**-Anweisung veran-

lasst man Java, den aktuellen Schleifendurchgang zu beenden und sofort mit dem nächsten zu beginnen. In der Regel kommen **break** und. **continue** im Rahmen einer Auswahlanweisung zum Einsatz, z.B. in folgendem Programm zur (relativ primitiven) Primzahlendiagnose:

```
class Primitiv {
   public static void main(String[] args) {
      boolean tg;
      int i, mpk, zahl;
      System.out.println("Einfacher Primzahlendetektor\n");
      while (true) {
         System.out.print("Zu untersuchende Zahl > 2 oder 0 zum Beenden: ");
         zahl = Simput.gint();
         if (Simput.status == false || (zahl <= 2  &&  zahl != 0)) {
            System.out.println("Keine Zahl oder illegaler Wert!\n");
            continue;
         }
         if (zahl == 0) break;
         tg = false;
         mpk = (int) Math.sqrt(zahl);  //maximaler Primzahlenkandidat
         for (i = 2; i <= mpk; i++)
            if (zahl % i == 0) {
               tg = true;
               break;
            }
         if (tg)
            System.out.println(zahl + " ist keine Primzahl (Teiler: " + i + ").\n");
         else
            System.out.println(zahl + " ist eine Primzahl.\n");
      }
   System.out.println("\nVielen Dank fuer den Einsatz dieser Software!");
   }
}
```

Man hätte die **continue**- bzw. **break**-Anweisungen zwar vermeiden können, doch werden bei dem vorgeschlagenen Verfahren die lästigen Spezialfälle (falsche Eingaben, 0 als Terminierungssignal) auf besonders übersichtliche Weise abgehakt, bevor der Kernalgorithmus startet.

Zum Kernalgorithmus der obigen Primzahlendiagnose sollte vielleicht noch erläutert werden, warum die Suche nach einem Teiler des Primzahlkandidaten bei dessen Wurzel enden kann:
Sei d ein Teiler der positiven, ganzen Zahl z, d.h. es gibt eine Zahl k (≥ 2) mit

$$z = k \cdot d$$

Dann ist auch k ein Teiler von z, und es gilt:

$$d \leq \sqrt{z} \quad \text{oder} \quad k \leq \sqrt{z}$$

Anderenfalls wäre das Produkt $k \cdot d$ ja größer als z. Wir haben also folgendes Ergebnis: Wenn eine Zahl z keinen Teiler kleiner oder gleich $\sqrt{z}$ hat, dann ist es eine Primzahl.

Zur Berechnung der Wurzel verwendet das Beispielprogramm die Methode **sqrt()** aus der Klasse **Math**, über die man sich bei Bedarf in der SDK-Dokumentation informieren kann.

3.5.4 Übungsaufgaben zu Abschnitt 3.5

1) In einer Lotterie gilt folgender Gewinnplan:

- Durch 13 teilbare Losnummern gewinnen 100 Euro.
- Losnummern, die nicht durch 13 teilbar sind, gewinnen immerhin noch einen Euro, wenn sie durch 7 teilbar sind.

Wird in folgendem Codesegment für Losnummern in der Variablen `losNr` der richtige Gewinn ermittelt?

```
if (losNr % 13 != 0)
   if (losNr % 7 == 0)
```

```
        System.out.println("Das Los gewinnt einen Euro!");
  else
     System.out.println("Das Los gewinnt 100 Euro!");
```

2) Warum liefert dieses Programm widersprüchliche Auskünfte über die boolsche Variable `b`?

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
  public static void main(String[] args) {
    boolean b = false;
    if (b = false)
      System.out.println("b ist false");
    else
      System.out.println("b ist true");
    System.out.println("\nKontr.ausg. von b: "+b);
  }
}</pre></td><td><pre>b ist true

Kontr.ausg. von b: false</pre></td></tr>
</table>

3) Das folgende (relativ sinnfreie) Programm soll Buchstaben mit 1 beginnend nummerieren:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
  public static void main(String[] args) {
    char bst = 'a';
    byte nr = 0;

    switch (bst) {
      case 'a': nr = 1;
      case 'b': nr = 2;
      case 'c': nr = 3;
    }
    System.out.println("Zu "+bst+
      " gehoert die Nummer "+nr);
  }
}</pre></td><td><pre>Zu a gehoert die Nummer 3</pre></td></tr>
</table>

Warum liefert es zum Buchstaben *a* die Nummer 3, obwohl für diesen Fall die Anweisung `nr = 1` vorhanden ist?

4) Erstellen Sie eine Variante des Primzahlen-Testprogramms aus Abschnitt 3.5.3.3, das ohne **break**- bzw. **continue** auskommt.

5) Wie oft wird die folgende **while**-Schleife ausgeführt?

```
class Prog {
  public static void main(String[] args) {
    int i = 0;
    while (i < 100);
    {
      i++;
      System.out.println(i);
    }
  }
}
```

6) Verbessern Sie das im Übungsaufgaben-Abschnitt 3.4.11 in Auftrag gegebene Programm zur DM-Euro - Konvertierung so, dass es nicht für jeden Betrag neu gestartet werden muss. Vereinbaren Sie mit dem Benutzer ein geeignetes Verfahren für den Fall, dass er das Programm doch irgendwann einmal beenden möchte.

7) Bei einem **double**-Wert sind mindestens 15 signifikante Dezimalstellen garantiert (siehe Abschnitt 3.2.1). Folglich kann ein Rechner die **double**-Werte 1,0 und $1{,}0 + 0{,}5^i$ ab einem bestimmten Exponenten *i* nicht mehr voneinander unterscheiden. Bestimmen Sie mit einem Testprogramm den größten ganzzahligen Index *i*, für den man noch erhält:

$$1{,}0 + 0{,}5^i > 1{,}0$$

8) In dieser Aufgabe sollen Sie verschiedene Varianten von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen *u* und *v* implementieren und die Performanzunterschiede messen.
Verwenden Sie als ersten Kandidaten den im Einführungsbeispiel zum Kürzen von Brüchen (Methode `kuerze()`) benutzten Algorithmus (siehe Abschnitt 1.1.3). Sein offensichtliches Problem besteht darin, dass bei stark unterschiedlichen Zahlen *u* und *v* sehr viele Subtraktions-Operationen erforderlich werden.
In der meist benutzten Variante des Euklidischen Verfahrens wird diese Problem vermieden, weil an Stelle der Subtraktion die Modulo-Operation zum Einsatz kommt, basierend auf folgendem Theorem: Für zwei natürliche Zahlen *u* und *v* ist der ggT gleich dem ggT von *u* und *u* % *v* (*u* modulo *v*).
Der alternative Euklidische Algorithmus läuft folgendermaßen ab:

> Es wird geprüft, ob *u* durch *v* teilbar ist. Trifft dies zu, ist der *v* der ggT. Anderenfalls ersetzt man:
> *u* durch *v*
> *v* durch *u* % *v*
> und startet das Verfahren neu.

Um den Zeitaufwand für beide Varianten zu messen, eignet sich die statische Methode

> **public static long currentTimeMillis()**

aus der Klasse **System** im Paket **java.lang** (siehe API-Dokumentation). Sie liefert die aktuelle Zeit in Millisekunden (seit dem 1. Januar 1970).

Für die Beispielwerte *u* = 999000999 und *v* = 36 liefern beide Euklid-Varianten sehr verschiedene Ergebnisse:

ggT-Bestimmung mit Euklid (Differenz)	ggT-Bestimmung mit Euklid (Modulo)
Erste Zahl: 999000999	Erste Zahl: 999000999
Zweite Zahl: 36	Zweite Zahl: 36
ggT: 9	ggT: 9
Benoet. Zeit: **63** Millisek.	Benoet. Zeit: **0** Millisek.

4 Klassen und Objekte

Klassen sind die wesentlichen Bestandteile von Java-Programmen. Primär handelt es sich hier um Baupläne für Objekte, die mit Eigenschaften (Instanzvariablen) und Handlungskompetenzen (Methoden) ausgestattet werden.
In einem Programm werden natürlich nicht nur Klassen definiert, sondern auch **Objekte** aus diversen (selbst definierten und vorgegebenen) Klassen erzeugt. Diese führen auf eine Botschaft hin die zugehörige Methode aus und liefern entsprechende Ergebnisse zurück. Im Idealfall entwickelt sich der Programmablauf als „kompetente und freundliche“ Interaktion zwischen Objekten.

In der Hoffnung, dass die bisher präsentierten Eindrücke von der objektorientierten Programmierung (OOP) neugierig gemacht und nicht abgeschreckt haben, kommen wir nun zur systematischen Behandlung dieser Softwaretechnologie. Auf die in Abschnitt 1 gewürdigte objektorientierte *Analyse* kann dabei aus Zeitgründen kaum eingegangen werden. Insbesondere bei größeren Projekten ist der Einsatz von objektorientierten Analyse- und Entwurfstechniken sehr empfehlenswert (siehe z.B. Balzert 1999).

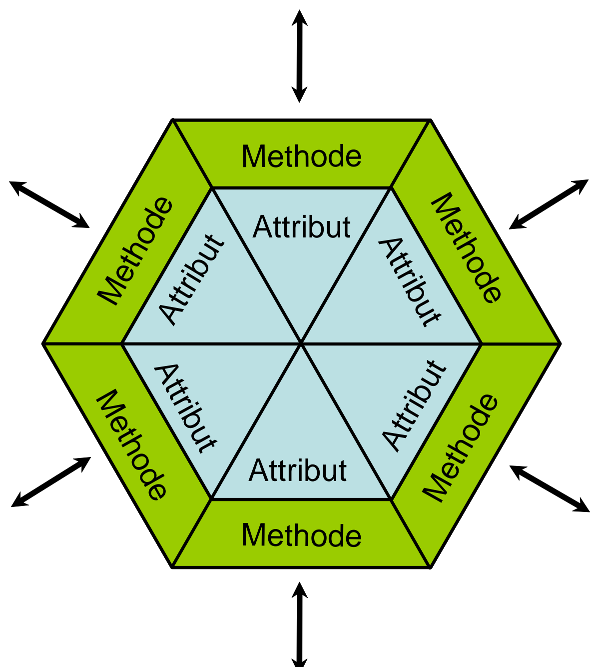
4.1 Überblick, historische Wurzeln, Beispiel

4.1.1 Einige Kernideen und Vorzüge der OOP

In diesem Abschnitt werden einige Kernideen bzw. Vorzüge ausgespart, die mit den bisherigen Beispielen noch nicht plausibel vermittelt werden können (z.B. Polymorphie).

Datenkapselung

In der OOP wird die traditionelle Trennung von Daten und Operationen aufgegeben. Hier besteht ein Programm aus **Klassen**, die durch Eigenschaften *und* zugehörige Methoden definiert sind.
Eine Klasse wird in der Regel ihre Eigenschaften gegenüber anderen Klassen verbergen (**Datenkapselung**, **information hiding**) und so vor ungeschickten Zugriffen schützen. Ihre Dienstleistungen sind dann ausschließlich über die Methoden der Schnittstelle verfügbar. Dies kommt in der folgenden, an Goll et al. (2000) angelehnten, Abbildung zum Ausdruck:



Auf diese Weise ergeben sich gravierende Vorteile:

- **Erhöhte Sicherheit durch Schutz der Eigenschaften**
  Direkte Zugriffe auf die Instanzvariablen einer Klasse sollen den klasseneigenen Methoden vorbehalten bleiben, die vom Designer der Klasse sorgfältig entworfen wurden. Damit sollten Programmierfehler seltener werden.
  Wer als Programmierer eine Klasse *verwendet*, braucht sich um den inneren Aufbau nicht zu kümmern, hat also relativ wenig Einarbeitungsaufwand und Fehlerrisiko.
  In unserem `Bruch`-Beispiel müssen wir als Klassen-Designer dafür sorgen, dass unter keinen Umständen der Nenner eines Bruches auf 0 gesetzt wird. Anwender unserer Klasse können einen Nenner einzig über die Methode `setzeNenner()` verändern und auf diesem Wege kein Unheil anrichten.
  Wenn doch ein Programmfehler wegen pathologischer Werte von Instanzvariablen auftritt, ist er relativ leicht zu lokalisieren, wenn nur wenige Methoden verantwortlich sein können.
- **Flexibilität, Anpassungsfähigkeit**
  Datenkapselung schafft günstige Voraussetzungen für die Wartung bzw. Verbesserung einer Klassendefinition. Solange die Methoden der Schnittstelle unverändert bzw. kompatibel bleiben, kann die interne Architektur einer Klasse ohne Nebenwirkungen auf andere Programmteile beliebig geändert werden.
- **Produktive Teamarbeit durch abgeschottete Verantwortungsbereiche**
  In großen Projekten können mehrere Programmierer nach der gemeinsamen Entwicklung von Schnittstellen relativ unabhängig an verschiedenen Klassen arbeiten.

Vererbung

Zu den Vorzügen der Klassen als statische Gebilde gesellt sich in der OOP ein höchst kreatives **Vererbungsverfahren**, das beste Voraussetzungen für die Erweiterung und Wiederverwendung von Software schafft: Bei der Definition einer neuen Klasse können alle Eigenschaften (Instanzvariablen) und Handlungskompetenzen (Methoden) einer Basisklasse übernommen werden. Es ist also leicht, ein Softwaresystem um neue Klassen mit speziellen Leistungen zu erweitern. Durch systematische Anwendung des Vererbungsprinzips entstehen mächtige Klassenhierarchien, die in beliebigen Projekten einsetzbar sind. Eine Klasse kann direkt genutzt werden (durch Erzeugen von Objekten) oder als Basisklasse dienen.

Realitätsnahe Modellierung

Klassen sind aber nicht nur ideale Bausteine für die rationelle Konstruktion von Softwaresystemen, sondern erlauben auch eine gute **Abbildung des Anwendungsbereichs**. In der zentralen Projektphase der objektorientierten Analyse und Modellierung sprechen Softwareentwickler und Auftraggeber dieselbe Sprache, so dass Missverständnisse und sonstige Kommunikationsprobleme weitgehend vermieden werden.
Neben den Klassen zur Modellierung von Akteuren oder Ereignissen des realen Anwendungsbereichs sind bei einer typischen Java-Anwendung aber auch zahlreiche Klassen beteiligt, die Akteure oder Ereignisse der virtuellen Welt des Computers repräsentieren (z.B. Bildschirmfenster, Mausklicks).

4.1.2 Strukturierte Programmierung und OOP

In vielen klassischen Programmiersprachen (z.B. C oder Pascal) werden zur Strukturierung von Programmen zwei Techniken angeboten, die in überarbeiteter Form auch bei der OOP genutzt werden:

- **Unterprogrammtechnik**
  Bei diesem Grundprinzip des Softwareentwurfs geht es darum, ein Gesamtproblem in unabhängige Teilprobleme aufzuteilen, die jeweils in einem eigenen *Unterprogramm* gelöst werden. Wird die von einem Unterprogramm erbrachte Leitung wiederholt (an verschiedenen Stellen eines Programms) benötigt, muss jeweils nur ein Aufruf mit dem Namen des Unterprogramms eingefügt werden. Durch diese Strukturierung ergeben sich kompaktere und übersichtlichere Programme, die leichter analysiert, korrigiert und erweitert werden können. Praktisch alle traditionellen Programmiersprachen unterstützen solche *Unterprogramme* (Subroutinen, Funktionen, Prozeduren), und meist stehen auch umfangreiche Bibliotheken mit Subroutinen für diverse Standardaufgaben zur Verfügung. Beim Einsatz einer Unterprogrammsammlung klassischer Art muss der Programmierer passende Daten bereitstellen, auf die dann vorgefertigte Funktionen losgelassen werden. Der Programmierer hat also seine Datensammlung *und* die Kollektion der verfügbaren Unterprogramme zu verwalten und zu koordinieren.
- **Problemadäquate Datentypen**
  Zusammengehörige Daten unter *einem* Variablennamen ansprechen zu können, vereinfacht das Programmieren erheblich. Mit den *Strukturen* der Programmiersprache C oder den *Records* der Programmiersprache Pascal lassen sich problemadäquate Datentypen konstruieren, die Elemente eines beliebigen, bereits bekannten Typs, enthalten dürfen. So eignet sich etwa für ein Programm zur Adressenverwaltung ein neu definierter Datentyp mit Datenelementen für Name, Vorname, Telefonnummer etc. Alle Daten eines Falles lassen sich dann in *einer* Variablen vom selbst definierten Typ speichern. Dies vereinfacht z.B. das Lesen, Kopieren oder Schreiben solcher Daten.

Die Strukturen bzw. Records der älteren Programmiersprachen werden in der OOP durch *Klassen* ersetzt, wobei diese Datentypen nicht nur durch eine Anzahl von *Eigenschaften* (Instanzvariablen beliebigen Typs) charakterisiert sind, sondern auch *Handlungskompetenzen* (Methoden) besitzen, welche die Aufgaben der Funktionen bzw. Prozeduren der älteren Programmiersprachen übernehmen.

Im Vergleich zur strukturierten Programmierung bietet die OOP u.a.:

- optimierte Modularisierung mit Zugriffsschutz
  Die Daten sind übersichtlich und sicher in Objekten gekapselt, während sie bei traditionellen Programmiersprachen entweder als globale Variablen allen Missgriffen ausgeliefert sind oder zwischen Unterprogrammen „wandern“ (Goll et al. 2000, S. 21), was bei Fehlern zu einer aufwändigen Suche entlang der Verarbeitungssequenz führen kann.
- bessere Abbildung des Anwendungsbereichs
- rationellere (Weiter-)Entwicklung von Software durch die Vererbungstechnik
- mehr Bequemlichkeit für Bibliotheksbenutzer
  Jede rationelle Softwareproduktion greift in hohem Maß auf Bibliotheken mit bereits vorhandenen Lösungen zurück. Dabei sind die Klassenbibliotheken der OOP deutlich einfacher zu verwenden als klassische Funktionsbibliotheken.

4.1.3 Auf-`Bruch` zu echter Klasse

In den Beispielprogrammen von Abschnitt 3 wurde mit den Klassendefinitionen lediglich eine in Java unausweichliche formale Anforderung an ausführbare Programmeinheiten erfüllt. Das in Abschnitt 1.1 vorgestellte Bruchrechnungsprogramm realisiert hingegen wichtige Prinzipien der objektorientierten Programmierung. Es wird nun wieder aufgegriffen und in verschiedenen Varianten bzw. Ausbaustufen als Beispiel verwendet.
Das Programm soll Schüler beim Erlernen der Bruchrechnung unterstützen. Eine objektorientierte Analyse der Problemstellung ergab, dass in einer elementaren Ausbaustufe des Programms lediglich

eine Klasse zur Repräsentation von Brüchen benötigt wird. Später sind weitere Klassen (z.B. Aufgabe, Übungsaufgabe, Testaufgabe, Lernepisode, Testepisode, Fehler, Schüler) zu ergänzen.

Die folgende `Bruch`-Klassendefinition ist im Vergleich zur Variante in Abschnitt 1.1 geringfügig verbessert worden:

- Als zusätzliche Eigenschaft erhält jeder Bruch ein `etikett` vom Datentyp **String**. Damit wird eine beschreibende Zeichenfolge verwaltet, die z.B. beim Aufruf der Methode `zeige()` neben anderen Eigenschaften auf dem Bildschirm erscheint.
- Weil die `Bruch`-Klasse ihre Eigenschaften kapselt, also fremden Klassen keine direkten Zugriffe erlaubt, muss sie auch für das `etikett` zum Lesen bzw. Setzen jeweils eine Methode bereitstellen.

Bei den unveränderten Methoden ist der Quellcode verkürzt wiedergegeben:

```
class Bruch {
    private int zaehler;           // wird automatisch mit 0 initialisiert
    private int nenner = 1;        // wird manuell mit 1 initialisiert
    private String etikett = "";  // die Ref.typ-Init. auf null wird ersetzt, siehe Text

    void setzeZaehler(int zpar) {zaehler = zpar;}

    boolean setzeNenner(int n) {.       .      .}

    void setzeEtikett(String epar) {
        int rind = epar.length();
        if (rind > 40)
            rind = 40;
        etikett = epar.substring(0, rind);
    }

    int gibZaehler() {return zaehler;}

    int gibNenner() {return nenner;}

    String gibEtikett()    {return etikett;}

    void kuerze() {.       .      .}

    void addiere(Bruch b) {.      .      .}

    boolean frage() {.     .      .}

    void zeige() {
        String luecke = "";
        for (int i=1;i<=etikett.length();i++)
            luecke = luecke + " ";
        System.out.println(" " + luecke  + "    " + zaehler + "\n" +
                           " " + etikett + " -----\n" +
                           " " + luecke  + "    " + nenner + "\n");
    }
}
```

Zunächst soll darauf hingewiesen werden, dass der Klassenname `Bruch` einer allgemein akzeptierten Java-Konvention folgend mit einem Großbuchstaben beginnt.

Hinsichtlich der Dateiverwaltung wird vorgeschlagen:

- Die `Bruch`-Klassendefinition sollte in einer eigenen Datei gespeichert werden.
- Den Namen dieser Datei sollte man aus dem Klassennamen durch Anhängen der Extension „.java“ bilden.

Im Unterschied zur Einleitung wird die `Bruch`-Klassendefinition anschließend gründlich erläutert. Dabei machen die Instanzvariablen relativ wenig Mühe, weil wir viele Details schon von den *loka-*

len Variablen kennen. Bei den Methoden gibt es mehr Neues zu lernen, so dass wir uns in Abschnitt 4.3 auf elementare Themen beschränken und später noch wichtige Ergänzungen behandeln.

4.2 Instanzvariablen (Eigenschaften)

Die Instanzvariablen einer Klasse besitzen viele Gemeinsamkeiten mit den *lokalen* Variablen, die wir im Abschnitt über elementare Sprachelemente ausschließlich verwendet haben, doch gibt es auch wichtige Unterschiede, die im Mittelpunkt des aktuellen Abschnitts stehen.
Unsere Demo-Klasse `Bruch` besitzt nach der Erweiterung um ein beschreibendes Etikett folgende Instanzvariablen (Eigenschaften):

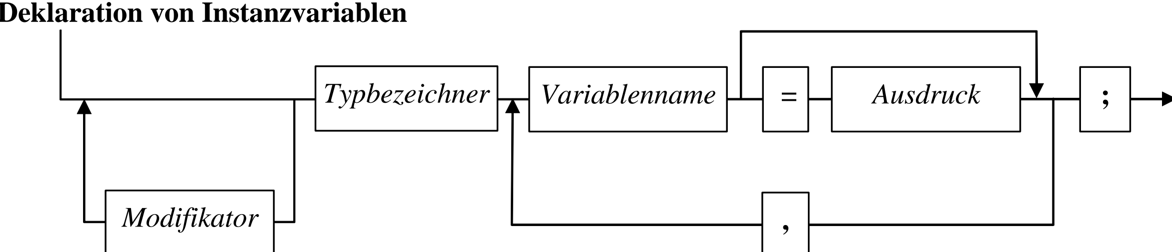
- `zaehler` (Datentyp **int**)
- `nenner` (Datentyp **int**)
- `etikett` (Datentyp **String**)

Jedes nach dem `Bruch`-Bauplan geschaffene Objekt wird einen vollständigen, individuellen Satz dieser Variablen erhalten.

4.2.1 Deklaration

Während lokale Variablen im Anweisungsteil einer Methode deklariert werden, erscheinen die Deklarationen der Instanzvariablen in der Klassendefinition *außerhalb* jeder Methodendefinition.

In der Regel gibt man beim Deklarieren von Instanzvariablen einen Modifikator zur Spezifikation der **Schutzstufe** an (siehe unten), so dass die Syntax im Vergleich zur Deklaration einer lokalen Variablen entsprechend erweitert werden muss.



Im Beispiel wird für alle Instanzvariablen mit dem Modifikator **private** die maximale Stufe der Datenkapselung angeordnet:

```
private int zaehler;
private int nenner = 1;
private String etikett = "";
```

Um fremden Klassen trotzdem einen (allerdings kontrollierten!) Zugang zu den `Bruch`-Instanzvariablen zu ermöglichen, wurden etliche Zugriffs-Methoden in die Klassendefinition aufgenommen (z.B. `setzeNenner()`).

Auf den ersten Blick erscheint die Datenkapselung nur beim Nenner eines Bruches gerechtfertigt. Doch auch bei den restlichen Instanzvariablen bringt sie (potentiell) Vorteile:

- Zugunsten einer übersichtlichen Bildschirmausgabe soll das Etikett auf 40 Zeichen beschränkt bleiben. Mit Hilfe der Zugriffsmethode `setzteEtikett()` kann dies gewährleistet werden.
- In den Zugriffsmethoden für `zaehler` und `nenner` kann man neben der Null-Überwachung für den Nenner noch weitere Wert-Restriktionen einbauen (z.B. auf positive Zahlen). Es sind sogar situationsabhängige Filtereffekte denkbar.
- Oft kann der Datentyp von gekapselten Instanzvariablen geändert werden, ohne dass die Schnittstelle inkompatibel zu vorhandenen Programmen wird, welche die Klasse benutzen. Im Fall der `Bruch`-Eigenschaften `zaehler` und `nenner` ist der Spielraum für Schnittstellen-kompatible Typanpassungen aber eher gering.

Trotz der überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden. Sie ist überflüssig, wenn bei einer Eigenschaft Lese- und Schreibzugriff erlaubt sind und eine Änderung des Datentyps nicht in Frage kommt.

4.2.2 Gültigkeitsbereich und Ablage im Hauptspeicher

Von den lokalen Variablen einer Methode unterscheiden sich die Instanzvariablen einer Klasse auch in Bezug auf den Gültigkeitsbereich und die Ablage im Hauptspeicher:

- Eine lokale Variable existiert, während die Methode ausgeführt wird, genauer: bis zum Verlassen des lokalsten Blockes. Sie wird auf dem so genannten **Stack** (deutsch: Stapel) gespeichert. Innerhalb des für ein Programm verfügbaren Speichers dient dieses Segment zur Verwaltung von Methodenaufrufen.
- Ein Satz mit allen Instanzvariablen einer Klasse wird für jedes neue Objekt erzeugt. Wenn das Objekt seine Existenz beendet, werden seine Variablen wieder beseitigt.
  Die Instanzvariablen eines neuen Objektes landen in einem Bereich des Programmspeichers, der als **Heap** (deutsch: Haufen) bezeichnet wird.

4.2.3 Initialisierung

Während bei lokalen Variablen *keine* Initialisierung stattfindet, erhalten die Instanzvariablen eines neuen Objektes automatisch folgende Startwerte:

Datentyp	**Initialisierung**
byte, **short**, **int**, **long**	0
float, **double**	0.0
char	'\u0000'
boolean	**false**
Referenz auf Objekt	**null**

Im Beispiel wird nur die automatische `zaehler`-Initialisierung unverändert übernommen:

- Beim `nenner` eines Bruches wäre die Initialisierung auf 0 bedenklich.
- Wie noch näher zu erläutern sein wird, ist **String** in Java *kein* primitiver Datentyp, sondern eine Klasse. Variablen von diesem Typ können einen Verweis auf ein Objekt aus dieser Klasse aufnehmen. Solange kein zugeordnetes Objekt existiert, hat die **String**-Variable den Wert **null**, zeigt also auf Nichts. Weil dies im Beispielprogramm zu Problemen führt, wird

ein **String**-Objekt mit einer leeren Zeichenfolge erstellt. Das Erzeugen des **String**-Objektes erfolgt *implizit*, indem der Stringvariablen `etikett` ein Zeichenfolgen-Literal zugewiesen wird.

4.2.4 Zugriff durch eigene und fremde Methoden

In den Methoden einer Klasse können die Instanzvariablen des aktuellen (die Methode ausführenden) Objektes über ihren Namen angesprochen werden, was z.B. in der `Bruch`-Methode `zeige()` zu beobachten ist:

```
System.out.println(" " + luecke  + "   " + zaehler + "\n" +
                   " " + etikett + " -----\n" +
                   " " + luecke  + "   " + nenner + "\n");
```

Im Beispiel zeigt sich syntaktisch kein Unterschied zwischen dem Zugriff auf die Instanzvariablen (`zaehler`, `nenner`, `etikett`) und dem Zugriff auf die lokale Variable (`luecke`).
Gelegentlich kann es sinnvoll sein, dem Instanzvariablennamen über das Schlüsselwort **this** eine Referenz auf das aktuelle Objekt voranzustellen (siehe Abschnitt 4.6.3).

Beim Zugriff auf eine Instanzvariablen eines *anderen* Objektes derselben Klasse muss dem Variablennamen eine Referenz auf das Objekt vorangestellt werden, wobei die Bezeichner durch den **Punktoperator** getrennt werden. In folgendem Beispiel überträgt ein `Bruch`-Objekt den eigenen `nenner`-Wert in die korrespondierende Instanzvariable eines anderen `Bruch`-Objektes, das über die Referenzvariable `bc` angesprochen wird:

```
bc.nenner = nenner;
```

Direkte Zugriffe auf die Instanzvariablen eines Objektes durch Methoden *fremder* Klassen sind zwar nicht grundsätzlich verboten, verstoßen aber gegen das Prinzip der Datenkapselung, das in der OOP von zentraler Bedeutung ist. Würden die `Bruch`-Instanzvariablen ohne den Modifikator **private**, also ohne Datenkapselung, deklariert, dann könnte z.B. der Nenner eines Bruches in der **main**()-Methode der (fremden) Klasse `BruchRechnung` direkt angesprochen werden:

```
System.out.println("Nenner von b: " + b.nenner);
b.nenner = 0;
```

In der von uns tatsächlich realisierten `Bruch`-Definition werden solche Zu- bzw. Fehlgriffe jedoch vom Compiler verhindert, z.B.:

```
BruchRechnung.java:8: nenner has private access in Bruch
                b1.nenner = 0;
                  ^
1 error
```

4.3 Methoden

In einer Klassendefinition werden Objekte entworfen, die eine Anzahl von Verhaltenskompetenzen (Methoden) besitzen, die von anderen Programmbestandteilen über festgelegte Kommunikationsregeln genutzt werden können. Objekte sind also Dienstleister, die eine Reihe von Nachrichten interpretieren und mit passendem Verhalten beantworten können.

Wie es im Objekt drinnen aussieht, geht dabei niemand etwas an. Seine Instanzvariablen (Eigenschaften) sind bei konsequenter Datenkapselung der Außenwelt nicht bekannt. Jeder Austausch findet über die Methoden statt, die daher auch die *Schnittstelle* eines Objektes bilden. Soll eine Eigenschaft trotz Datenkapselung zugänglich sein, werden entsprechende Methoden zum Lesen bzw.

Verändern erforderlich. Unsere `Bruch`-Klasse besitzt Lese- und Schreibmethoden für alle Eigenschaften (`setzeZaehler()`, `setzeNenner()`, `setzeEtikett()`, `gibZaehler()`, `gibNenner()`, `gibEtikett()`), dies ist jedoch nicht bei jeder Klasse erforderlich bzw. wünschenswert.

Beim Aufruf einer Methode ist oft über so genannte *Parameter* die gewünschte Verhaltensweise festzulegen, und bei vielen Methoden wird dem Aufrufer ein *Rückgabewert* geliefert, z.B. mit der angeforderten Information.

Ziel der Klassendefinition sind kompetente, einfach und sicher einsetzbare Objekte, die reale Objekte aus dem Aufgabenbereich einer Software gut repräsentieren. Wenn ein Programmierer z.B. ein Objekt aus unserer `Bruch`-Klasse verwendet und auf dem Bildschirm erscheinen lassen möchte, soll er sich nicht um die Ausgabe der einzelnen Eigenschaften kümmern müssen. Statt dessen soll das Objekt die Kompetenz zur geeigneten „Selbstdarstellung" besitzen und auf eine Botschaft hin realisieren. Dazu verfügt die Bruch-Klasse über die Methode `zeige()`:

```
void zeige() {
      String luecke = "";
      for (int i=1;i<=etikett.length();i++)
            luecke = luecke + " ";
      System.out.println(" " + luecke  + "    " + zaehler + "\n" +
                         " " + etikett + " -----\n" +
                         " " + luecke  + "    " + nenner + "\n");
}
```

Hier kommt neben den Instanzvariablen der `Bruch`-Klasse auch noch eine lokale Variable vom Typ **String** zum Einsatz, für die eine bequeme Verkettungs-Operation definiert ist (wie bei Zeichenketten-Literalen).
Da nur eine Bildschirmausgabe bezweckt ist, liefert `zeige()` (wie die Startmethode **main()**) keinen Rückgabewert, so dass im Kopf der Methodendefinition der Typ **void** anzugeben ist.
Weil auch nur eine einzige Arbeitsweise der `zeige()`-Methode vorgesehen ist, sind keine Parameter nötig, wobei aber die leere Parameterliste aber keinesfalls fehlen darf, weil sie in Java zur Unterscheidung zwischen Variablen- und Methodennamen dient.

In der `zeige()` - Implementierung wird die Länge des Etiketts zu einem Bruch mit der **String**-Methode **length()** ermittelt. Damit gelingt eine brauchbar formatierte Konsolenausgabe von etikettierten Brüchen, z.B.:

```
                13
  Beispiel:  -----
                221
```

Leere Rückgaben und Parameterlisten sind keinesfalls die Regel, so dass wir uns näher mit den Kommunikationsregeln beim Methodenaufruf beschäftigen müssen.

Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, sind die Methoden einer Klasse nur einmal vorhanden. Sie befinden sich in einem Bereich des programmeigenen Speichers, der als **Method Area** bezeichnet wird.

4.3.1 Methodendefinition

4.3.1.1 Methodenkopf, Formalparameter

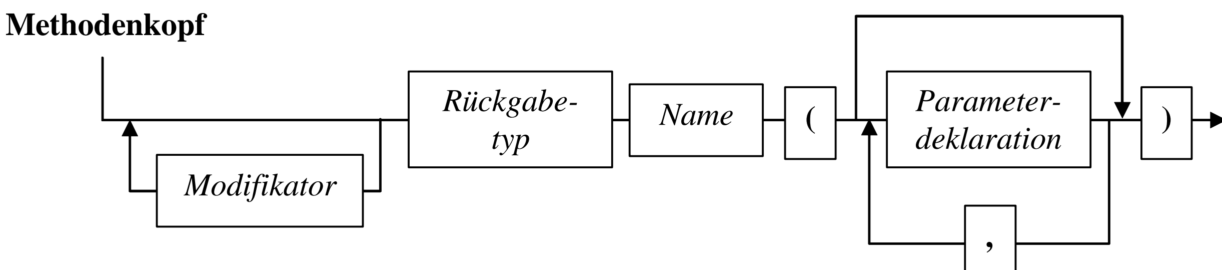
Im Kopf der Methodendefinition wird über so genannte **Formalparameter** festgelegt, welche Argumente eine Methode zur Spezifikation ihrer Arbeitsweise kennt. Beim späteren Aufruf einer Me-

thode ist eine korrespondierende Liste von so genannten **Aktualparametern** anzugeben. In den Anweisungen des Methodenrumpfs sind die Formalparameter wie lokale Variablen zu verwenden, die mit den Werten der Aktualparameter initialisiert wurden.

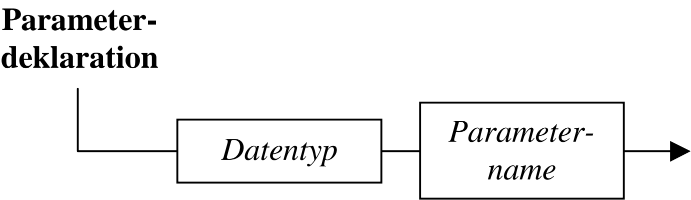
Für jeden Formalparameter sind folgende Angaben zu machen:

- **Datentyp**
  Es sind beliebige Typen erlaubt (primitive, Objektreferenzen).
- **Name**
  Damit sich die Parameternamen von anderen Bezeichnern (z.B. Namen von Instanzvariablen) unterscheiden, hängen manche Programmierer ein Suffix an, z.B. „par" oder einen Unterstrich.
- **Position**
  Die beim späteren Aufruf der Methode übergebenen Aktualparameter werden gemäß ihrer Reihenfolge den Formalparametern zugeordnet.

Das schon im Abschnitt über elementare Sprachelemente begonnene Syntaxdiagramm zum Methodenkopf



soll nun durch eine Explikation des Begriffs *Parameterdeklaration* vervollständigt werden:



Der Datentyp eines Formalparameters muss auch dann explizit angegeben werden, wenn er mit dem Typ des Vorgängers übereinstimmt.

Als erstes Beispiel betrachten wir die `Bruch`-Methode `SetzeNenner()`, die den einzigen Parameterwert (Datentyp **int**) als neuen Nenner einträgt, falls dieser von 0 verschieden ist.

```
boolean setzeNenner(int n) {
    if (n != 0) {
        nenner = n;
        return true;
    } else
        return false;
}
```

Als Beispiel mit etwas längerer Parameterliste betrachten wir noch folgende Variante der `Bruch`-Methode `addiere()` Das beauftragte Objekt soll den via Parameterliste (`zpar`, `npar`) übergebenen Bruch zu seinem eigenen Wert addieren und optional (Parameter `autokurz`) das Resultat gleich kürzen:

```
void addiere(int zpar, int npar, boolean autokurz) {
      zaehler = zaehler*npar + zpar*nenner;
      nenner = nenner*npar;
      if (autokurz)
```

```
            kuerze();
}
```

Auch bei einer Methoden-Definition kann per Modifikator der voreingestellte **Zugriffsschutz** verändert werden. Dies ist bei keiner `Bruch`-Methode geschehen, so dass hier gilt: Die Methoden sind in allen Klassen verwendbar, die zum selben *Paket* gehören. In konkreten Fall handelt es sich um das namenlose *Standardpaket*, dem alle Klassen ohne explizite Paketzuordnung angehören.
Soll eine Klasse startfähig sein, muss sie über eine Methode namens **main()** mit Schutzstufe **public** verfügen, damit sie vom Java-Interpreter ausgeführt werden kann.
Wie Sie merken, sind genauere Erläuterungen zu Zugriffsmodifikatoren für Instanzvariablen, Methoden und Klassen erst sinnvoll, nachdem wir uns ausführlich mit Paketen beschäftigt haben.

Nicht alle Modifikatoren im Methodenkopf dienen dem Zugriffsschutz: Die **main()**-Methode einer startfähigen Klasse muss nicht nur als **public**, sondern auch als **static** deklariert werden, damit sie als Klassenmethode (siehe Abschnitt 4.7) ausgeführt werden kann, bevor ein Objekt der Klasse existiert.

4.3.1.2 Rückgabewerte, return-Anweisung

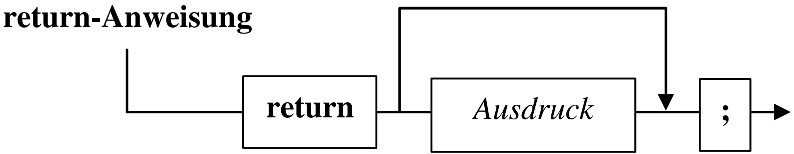
Bei der Definition einer Methode muss festgelegt werden, von welchem Datentyp ihre Antwort an die aufrufende Programmeinheit ist.
Erfolgt *keine* Rückgabe, ist der Ersatztyp **void** anzugeben. Ein zugehöriger Methodenaufruf stellt dann *keinen* Ausdruck dar, wird aber durch ein nachgestelltes Semikolon zur Anweisung.

Als Beispiel betrachten wir die `Bruch`-Methode `SetzeNenner()`, die den Aufrufer durch einen Rückgabewert vom Datentyp **boolean** darüber informiert erfährt, ob sein Auftrag ausgeführt werden konnte (**true**) oder nicht (**false**):

```
boolean setzeNenner(int n) {
    if (n != 0) {
        nenner = n;
        return true;
    } else
        return false;
}
```

Ist der Rückgabetyp einer Methode von **void** verschieden, dann muss im Rumpf dafür gesorgt werden, dass jeder mögliche Ausführungspfad mit einer **return**-Anweisung endet, die einen Wert passenden Typs übergibt.
Die **return**-Anweisung beendet die Ausführung der Methode und legt über einen Ausdruck passenden Typs den Rückgabewert fest:

return-Anweisung



Bei Methoden ohne Rückgabewert ist die **return**-Anweisung nicht unbedingt erforderlich, kann jedoch (in der Variante *ohne* wert-genererierenden Ausdruck) dazu verwendet werden, die Methode vorzeitig zu beenden (z.B. im Rahmen einer bedingten Anweisung).

Soll eine Methode mehr als nur einen Wert von primitivem Datentyp zurückliefern, dann muss ein Klassentyp benutzt werden (siehe Abschnitt 4.6).

4.3.1.3 Methodenrumpf

Über die Verbundanweisung, die den Rumpf einer Methode bildet, haben Sie in Abschnitt 4.3.1 bereits gelernt:

- Hier werden die Formalparameter wie lokale Variablen verwendet. Ihre Besonderheit besteht darin, dass sie bei jedem Methodenaufruf über Aktualparameter (s. u.) von der rufenden Programmeinheit initialisiert werden, so dass diese den Ablauf der Methode beeinflussen kann.
- Die **return**-Anweisung dient zur Rückgabe von Werten an den Aufrufer und/oder zum Beenden einer Methodenausführung.

Ansonsten können beliebige Anweisungen unter Verwendung von elementaren und objektorientierten Sprachelementen eingesetzt werden, um den Zweck einer Methode zu implementieren.
Die eben als Beispiel betrachtete `Bruch`-Methode `SetzeNenner()` beschränkt sich auf die (bedingungsabhängige) Änderung einer Instanzvariablen. Im allgemeinen wird eine Methode etwas mehr Aktivität entfalten und sich dabei auch nicht unbedingt auf die Veränderung von lokalen Variablen und objekteigenen Instanzvariablen beschränken. Über eine als Parameter übergebene Referenzvariable (siehe Abschnitt 4.6.1) sind z.B. auch Einwirkungen auf andere Objekte möglich, am besten wiederum durch Methodenaufrufe. Dabei kann es sich um andere Objekte der eigenen Klasse oder um Objekte beliebiger anderer Klassen handeln.

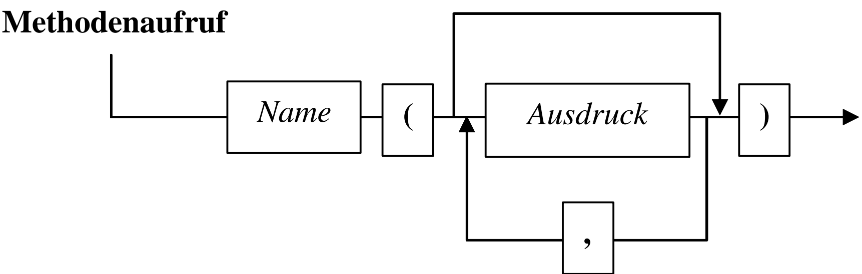
4.3.2 Methodenaufruf, Aktualparameter

Die *Verwendung* eines Objektes sollte nach der reinen Lehre ausschließlich im Aufrufen seiner Methoden bestehen, z.B.:

```
b1.zeige();
```

Nach objektorientierter Denkweise wird hier eine *Botschaft* an ein Objekt geschickt: „b1, zeige dich!“.
Als Syntaxregel ist festzuhalten, dass zwischen dem Objektnamen (genauer: dem Namen der Referenzvariablen) und dem Methodennamen der **Punktoperator** zu stehen hat.

Beim Aufruf einer Methode folgt dem Namen die in runde Klammern eingeschlossene Liste mit den **Aktualparametern**, wobei es sich um eine synchron zur Formalparameterliste geordnete Serie von Ausdrücken passenden Typs handeln muss.



Es muss grundsätzlich eine Parameterliste angegeben werden, ggf. eben eine leere.

Als Beispiel betrachten wir einen Aufruf der in Abschnitt 4.3.1.1 vorgestellten Variante der `Bruch`-Methode `addiere()`:

```
b.addiere(1, 2, true);
```

Liefert eine Methode einen Wert zurück, stellt ihr Aufruf einen *Ausdruck* dar und kann auch als Argument in komplexeren Ausdrücken auftreten, z.B.:

```
if (!b1.setzeNenner(pn))
  System.out.println("Mit dem Nenner gab es ein Problem.");
```

Durch ein angehängtes Semikolon wird jeder Methodenaufruf zur vollständigen Anweisung (siehe erstes Beispiel). Auch eine Methode *mit* Rückgabewert kann als selbständige Anweisung eingesetzt werden, wobei der Rückgabewert ignoriert wird, z.B.:

```
b1.setzeNenner(pn);
```

Soll in einer Methodenimplementierung vom aktuell handelnden Objekt eine andere Methode derselben Klasse ausgeführt werden, so wird beim Aufruf *keine* Objektbezeichnung angegeben. In beiden Varianten der `Bruch`-Methode `addiere()` soll das beauftragte Objekt den via Parameterliste übergebenen Bruch zu seinem eigenen Wert addieren und das Resultat eventuell gleich kürzen. Zum Kürzen kommt natürlich die bereits vorhandene `Bruch`-Methode zum Einsatz. Weil sie vom gerade agierenden Objekt auszuführen ist, wird keine Objektbezeichnung angegeben:

```
void addiere(int zpar, int npar, boolean autokurz) {
      zaehler = zaehler*npar + zpar*nenner;
      nenner = nenner*npar;
      if (autokurz)
      kuerze();
}
```

Wer gerne auch solche Methodenaufrufe mit der Syntax

Empfänger.Botschaft

realisieren möchte, kann mit dem Schlüsselwort **this** das aktuelle Objekt ansprechen:

```
this.kuerze();
```

4.3.3 Methoden überladen

Die in Abschnitt 4.3.1.1 vorgestellte `addiere()`-Variante kann problemlos in der `Bruch`-Klassendefinition neben der dort bereits vorhandenen `addiere()`-Variante koexistieren, weil beide Methoden unterschiedliche Parameterlisten besitzen. Man spricht hier von einer *Überladung*. Von einer Methode unterschiedliche Varianten in eine Klassendefinition aufzunehmen, lohnt sich z.B. in folgenden Situationen:

- Für verschiedene Datentypen (z.B. **double** und **int**) sind analoge Methoden vorhanden. So gibt es z.B. in der Klasse **Math** im API-Paket **java.lang** folgende Methoden, um den Betrag einer Zahl zu berechnen:

  static double abs(double a)
  static float abs(float a)
  static int abs(int a)
  static long abs(long a)

- Für eine Methode sollen unterschiedliche umfangreiche Parameterlisten angeboten werden, sodass zwischen einer bequem aufrufbaren Standardausführung (z.B. mit leerer Parameterliste) und einer individuell gestalteten Ausführungsvariante gewählt werden kann.

4.4 Objekte

Jede Java – Anwendung benötigt eine Startklasse mit statischer **main()**-Methode, die vom Interpreter beim Programmstart automatisch aufgerufen wird. Statt die `Bruch`-Definition mit einer **main()**-Methode auszustatten, haben wir eine separate „Hauptprogramm-Klasse" `BruchRechnung` definiert, deren **main()**-Methode ein Objekt aus der Klasse `Bruch` erzeugt und mit Aufträgen versorgt:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class BruchRechnung {
    public static void main(String[] args) {
        Bruch b = new Bruch();
        b.setzeZaehler(4);
        b.setzeNenner(16);
        b.kuerze();
        b.setzeEtikett("Der gekuerzte Bruch:");
        b.zeige();
    }
}</pre></td><td><pre>   1
Der gekuerzte Bruch: -----
                        4</pre></td></tr>
</table>

Hinsichtlich der Dateiverwaltung wird vorgeschlagen:

- Auch die `BruchRechnung`-Klassendefinition sollte in einer eigenen Datei gespeichert werden.
- Den Namen dieser Datei sollte man aus dem Klassennamen durch Anhängen der Extension „.java" bilden.

Beim Compilieren der Quellcode-Dateien des Projektes entstehen die Bytecode-Dateien **Bruch.class** und **BruchRechnung.class**. Der Compiler erzeugt übrigens grundsätzlich für jede Klasse eine eigene Bytecode-Datei, so dass aus einer Quellcode-Datei eventuell mehrere Bytecode-Dateien entstehen.

Beim Compilieren von **BruchRechnung.java** wird die dort benutzte Klassendatei **Bruch.class** bei Bedarf automatisch aus ihrer Quellcode-Datei erzeugt, wobei der Compiler aus der ihm bekannten Klassenbezeichnung (hier `Bruch`) den Dateinamen **Bruch.java** konstruiert. Wenn Sie obigen Vorschlag zur Benennung der Quellcode-Dateien nicht befolgen, müssen Sie beide Dateien einzeln übersetzen lassen.

4.4.1 Referenzvariablen definieren, Klassen als Datentypen

Um die Wirkungsweise der Variablendeklaration

```
Bruch b = new Bruch();
```

zu erklären, beginnen wir mit einer einfacheren Variante:

```
Bruch b;
```

Hier wird die **Referenzvariable** `b` definiert, und als Datentyp wird unsere `Bruch`-Klasse verwendet.

Zunächst halten wir fest, dass *Klassen als Datentypen* verwendet werden können. Damit haben wir bisher folgende Datentypen kennen gelernt:

- Primitive Typen (**boolean**, **char**, **byte**, ..., **double**)
- Klassentypen
  Es kommen Klassen aus dem Java-API und selbst definierte Klassen in Frage.

4.4.2 Objekte erzeugen

Die Referenzvariable `b` kann als Wert nicht etwa ein `Bruch`-Objekt (mit den oben genannten Instanzvariablen) aufnehmen, sondern einen **Verweis** (eine **Referenz**) auf einen Ort im Heap-Bereich des Speichers, wo sich ein `Bruch`-Objekt befindet.
Da `b` in einer Methode deklariert wird, handelt es sich um eine lokale Variable ohne Initialisierungswert.
Damit der Variablen `b` ein Verweis auf ein `Bruch`-Objekt als Wert zugewiesen werden kann, muss ein solches erst erzeugt werden, was in Java der **new**-Operator übernimmt, z.B. in folgendem Ausdruck:

```
new Bruch()
```

Als Operanden erwartet **new** einen Klassennamen, dem eine Parameterliste zu folgen hat, weil er hier als *Konstruktor* (s. u.) aufzufassen ist.

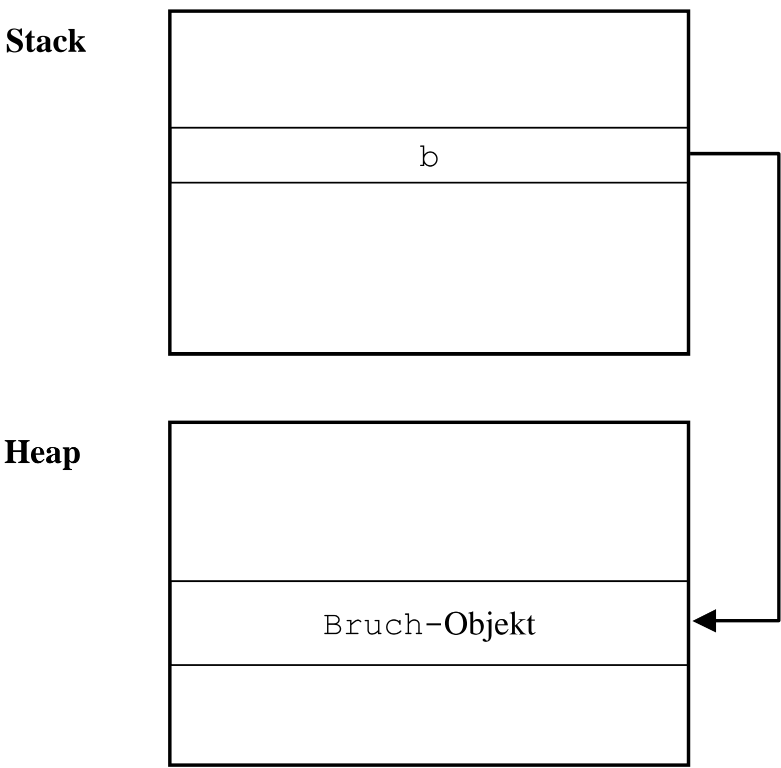
Der **new**-Operator liefert als Wert eine Referenz, die einen Zugriff auf das neue Objekt (seine Eigenschaften und Methoden) erlaubt. In der **main**()-Methode des Bruchrechnungsprogramms schreiben wir die vom **new**-Operator gelieferte Referenz mit dem Zuweisungsoperator in die Referenzvariable `b`.

Während lokale Variablen bereits beim Aufruf einer Methode (also unabhängig vom konkreten Ablauf) im **Stack**-Bereich des programmeigenen Hauptspeichers angelegt werden, entstehen *Objekte* (mit ihren Instanzvariablen) erst bei der Auswertung des **new**-Operators. Sie erscheinen auch nicht auf dem Stack, sondern werden im **Heap**-Bereich des programmeigenen Hauptspeichers angelegt. Wie viele Objekte dynamisch erzeugt werden, hängt vom konkreten Ablauf einer Methode ab (und vom verfügbaren Heap-Speicher). Die Anzahl der bei einem Methodenaufruf erzeugten Objekte ist durchaus nicht durch die Anzahl der lokalen Referenzvariablen der Methode beschränkt.

Nach Ausführung der Anweisung

```
Bruch b = new Bruch();
```

in der `BruchRechnung` - Methode **main()** zeigt die lokale Referenzvariable `b` auf ein `Bruch`-Objekt:



4.4.3 Überflüssige Objekte entfernen, Garbage Collector

Wenn keine Referenz mehr auf ein Objekt zeigt, wird es vom **Garbage Collector** (Müllsammler) der virtuellen Maschine automatisch entsorgt, und die belegten Ressourcen (z.B. Speicher) werden frei gegeben. Im Unterschied zu vielen anderen Programmiersprachen (z.B. C++) sind in Java wichtige Fehlerquellen trocken gelegt:

- Weil der Programmierer keine Verpflichtung (und Berechtigung) zum Entsorgen von Objekte hat, kann es nicht zu Programmabstürzen durch Zugriff auf voreilig vernichtete Objekte kommen.
- Es entstehen keine **Speicherlöscher** (memory leaks). Diese kommen z.B. in C++ dann zu Stande, wenn dynamisch angeforderter Speicher nicht explizit (mit **delete**) frei gegeben wird.

Sollen die Objekte einer Klasse vor dem Entsorgen noch spezielle Aufräumaktionen durchführen, dann muss eine Methode namens **finalize()** definiert werden, die vom Garbage Collector aufgerufen wird, z.B.:

```
protected void finalize() throws Throwable {
    super.finalize();
    System.out.println("Objekt Nr. "+nummer+" finalisiert");
}
```

In dieser Methodendefinition tauchen einige Bestandteile auf, die bald ausführlich zur Sprache kommen und hier ohne großes Grübeln hingenommen werden sollten:

- `super.finalize();`
  Bereits die Urahnklasse **java.lang.Object**, von der alle Java-Klassen abstammen, verfügt über eine **finalize()**-Methode. Überschreibt man diese in einer abgeleiteten Klasse, so sollte am Anfang der eigenen Implementation die überschriebene Variante aufgerufen werden, wobei das Schlüsselwort **super** die Basisklasse anspricht.

- `protected`
  In der Klasse **Object** ist für **finalize()** die Schutzstufe **protected** festgelegt, und dieser Zugriffsschutz darf beim Überschreiben der Methode nicht verschärft werden. Die ohne Angabe eines Modifikators voreingestellte Schutzstufe *Paket* enthält gegenüber **protected** eine Einschränkung und ist daher verboten.
- `throws Throwable`
  Die **finalize()**-Methode der Klasse **Object** löst ggf. eine Ausnahme aus der Klasse **Throwable** aus. Diese muss von der eigenen **finalize()**-Implementierung entweder abgefangen oder weitergereicht werden, was durch den Zusatz **throws Throwable** im Kopf anzumelden ist.

Der Garbage Collector wird im Normalfall nur tätig, wenn die virtuelle Maschine Speicher benötigt gerade nichts wichtigeres zu tun hat, so dass der genaue Zeitpunkt für die Entsorgung eines Objektes kaum vorhersehbar ist. Mit der Methode **System.gc()** kann man aber den sofortigen Einsatz des Müllsammlers erzwingen, z.B. vor ressourcen-intensiven Aktionen eines Programms.

In der Regel müssen Sie sich um das Entsorgen überflüssiger Objekte nicht kümmern, also weder eine **finalize()**-Methode für eigene Klassen definieren, noch die **System**-Methode **gc()** aufrufen.

In unseren bisherigen Beispielen verschwindet die letzte (und einzige) Referenz auf ein `Bruch`-Objekt mit dem Beenden der Methode, in der das Objekt erstellt wurde. Es ist jedoch durchaus möglich (und normal), dass ein Objekt die erzeugende Methode überlebt (siehe Abschnitt 4.6.2).

Andererseits kann man ein Objekt zu jedem beliebigen Zeitpunkt „aufgeben“, indem man alle Referenzen entfernt. Dazu setzt man die entsprechenden Referenzvariablen entweder auf den Wert **null** oder weist ihnen eine andere Referenz zu, z.B.:

Quellcode

```
class BruchRechnung {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(), b2 = new Bruch();
      b1.setzeZaehler(1); b1.setzeNenner(2);
      b2.setzeZaehler(3); b2.setzeNenner(4);
      b1.zeige();      b2.zeige();
      b1 = null;
      b2 = new Bruch();
      b2.setzeZaehler(5); b2.setzeNenner(6);
      System.gc();
      b2.zeige();
   }
}
```

Ausgabe

```
   1
 -----
   2

   3
 -----
   4

Bruch@1a16869 finalisiert
Bruch@1cde100 finalisiert
   5
 -----
   6
```

Die Ausgaben

```
Bruch@1a16869 finalisiert
Bruch@1cde100 finalisiert
```

stammen von folgender `finalize()`-Methode:

```
protected void finalize() throws Throwable {
   super.finalize();
   System.out.println(this+" finalisiert");
}
```

Hier wird mit dem Schlüsselwort **this** (vgl. Abschnitt 4.6.3) das aktuelle Objekt angesprochen. Bei der automatischen Konvertierung der Referenz in eine Zeichenfolge wird die vom Laufzeitsystem verwaltete Objektbezeichnung zu Tage fördert.

Dank Garbage Collector spart der Java-Programmierer lästige Arbeit und vermeidet Speicherlöcher. Außerdem entfällt mit der Notwendigkeit, obsoleten Speicher explizit frei zu geben, auch die Gefahr, dies zu früh zu tun. In C++ kann man im Übereifer per **delete** ohne Weiteres ein Objekt beseitigen, auf das noch Referenzen zeigen. Die so entstandenen Zeiger ins Nichts sorgen dann für Programmabstürze oder für unerklärliches Fehlverhalten.

4.5 Konstruktoren

In diesem Abschnitt werden spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten zum Einsatz kommen. Auch bei diesen so genannten *Konstruktoren* kann man die Botschafts-Metapher für Methoden aufrecht erhalten, muss aber den Adressaten auswechseln:

- Bei den oben behandelten Methodenaufrufen wird ein bestimmtes Objekt aufgefordert, eine Aktion durchzuführen. Wir haben es mit *objektbezogenen* Methoden zu tun.
- Nun geht es um Anforderungen an die virtuelle Maschine, ein neues Objekt aus einer bestimmten Klasse zu erzeugen. Wir haben es mit *klassenbezogenen* Methoden zu tun.
  Bei Konstruktoren sind einige syntaktische Besonderheiten zu beachten, auch gegenüber anderen klassenbezogenen Methoden (vgl. Abschnitt 4.7).

Wie Sie bereits wissen, wird zum Erzeugen von Objekten der **new**-Operator verwendet. Als Operand ist der vom Laufzeitsystem auszuführende Konstruktor der gewünschten Klasse anzugeben. Hat der Programmierer zu einer Klasse keinen Konstruktor definiert, dann kommt ein **Standard-Konstruktor** zum Einsatz. Weil dieser keine Parameter besitzt, ergibt sich sein Aufruf aus dem Klassennamen durch Anhängen einer leeren Parameterliste, z.B.:

```
Bruch b2 = new Bruch();
```

Der Standard-Konstruktor ist übrigens *nicht* für die oben erwähnten automatischen Initialisierungen von Instanzvariablen verantwortlich und entfaltet auch sonst keine große Aktivität.
In aller Regel ist es sinnvoll, *explizit* einen Konstruktor zu definieren, der das individuelle Initialisieren der Instanzvariablen von neuen Objekten erlaubt. Dabei sind folgende Regeln zu beachten:

- Der Konstruktor trägt den selben Namen wie die Klasse.
- Es darf eine Parameterliste definiert werden, was zum Zweck der Initialisierung ja auch unumgänglich ist.
- Der Konstruktor liefert grundsätzlich keinen Rückgabewert, und es wird *kein* Typ angegeben, auch *nicht* der Ersatztyp **void**, mit dem wir bei gewöhnlichen Methoden den Verzicht auf einen Rückgabewert dokumentieren müssen.
- Sobald man einen eigenen Konstruktor definiert, steht der Standard-Konstruktor *nicht* mehr zur Verfügung.
- Ist weiterhin ein paramameterfreier Konstruktor erwünscht, so muss dieser *zusätzlich* definiert werden.
- Es sind generell beliebig viele Konstruktoren möglich, die alle den selben Namen und jeweils eine individuelle Parameterliste haben müssen. Das Überladen von Methoden (vgl. Abschnitt 4.3.3) ist also auch bei Konstruktoren erlaubt.

Die folgende Variante unseres Beispielprogramms enthält einen expliziten Konstruktor mit Initialisierungsmöglichkeit und einen zusätzlichen, parameterfreien Konstruktor mit leerem Anweisungsteil:

```
class Bruch {
    private int zaehler;
    private int nenner = 1;
    private String etikett = "";
```

```
    Bruch(int zpar, int npar, String epar) {
        zaehler = zpar;
        nenner  = npar;
        etikett = epar;
    }

    Bruch() {}

    void setzeZaehler(int zpar) {zaehler = zpar;}
    boolean setzeNenner(int n) {.       .       .}
    void setzeEtikett(String epar) {.   .       .}
    int gibZaehler() {return zaehler;}
    int gibNenner() {return nenner;}
    String gibEtikett()    {return etikett;}
    void kuerze() {.       .       .}
    void addiere(Bruch b) {.       .       .}
    boolean frage() {.     .       .}
    void zeige(){.  .      .}
}
```

In der folgenden Variante des Testprogramms werden beide Konstruktoren eingesetzt:

Quellcode	Ausgabe

`class BruchRechnung {`
`   public static void main(String[] args) {`
`      Bruch b1 = new Bruch(1, 2, "b1 = ");`
`      Bruch b2 = new Bruch();`
 	
`      b1.zeige();`
`      b2.zeige();`
`   }`
`}`
	`          1`
	` b1 =  -----`
	`          2`
	
	`   0`
	` -----`
	`   1`

Konstruktor-Aufrufe sind ausschließlich über den **new**-Operator erlaubt.

4.6 Referenzen

In diesem Abschnitt wird sich erneut zeigen, dass gute Referenzen nützlich sind.

4.6.1 Referenzparameter

Wir haben schon festgehalten, dass die Formalparameter einer Methode wie *lokale Variablen* funktionieren, die mit den Werten der Aktualparameter initialisiert werden. Bei Parametern von *primitiven Datentyp* wirken sich Änderungen innerhalb einer Methode daher *nicht* auf die rufende Programmeinheit aus.
Bei einem Parameter vom *Referenztyp* wird ebenfalls der Wert des Aktualparameters (eine Objektreferenz) beim Methodenaufruf in den Formalparameter kopiert. Es wird jedoch keinesfalls eine Kopie des referenzierten Objekts (auf dem Heap) erstellt, so dass Formal- und Aktualparameter auf dasselbe Objekt zeigen.
Speziell bei speicherintensiven Objekten hat die in Java realisierte Referenzparametertechnik den Vorteil, dass beim Methodenaufruf Zeit und Speicherplatz gespart werden.
Von den beiden `addiere()` – Methoden des `Bruch`-Projektes verfügt die ältere Variante über einen Referenzparameter:

```
void addiere(Bruch b) {
      zaehler = zaehler*b.nenner + b.zaehler*nenner;
      nenner = nenner*b.nenner;
      kuerze();
}
```

Mit dieser Botschaft wird ein Objekt beauftragt, den via Referenzparameter spezifizierten Bruch zu seinem eigenen Wert zu addieren (und das Resultat gleich zu kürzen).

Zähler und Nenner des fremden `Bruch`-Objektes können per Referenzparameter und Punktoperator trotz Schutzstufe **private** direkt angesprochen werden, weil der Zugriff in einer `Bruch`-Methode stattfindet.
Dass in einer `Bruch`-Methodendefinition ein Referenzparameter vom Typ `Bruch` verwendet wird, ist übrigens weder „zirkulär" noch ungewöhnlich.

In `addiere()` bleibt das per Referenzparameter ansprechbare Objekt unverändert. Sofern entsprechende Zugriffsrechte vorliegen, was bei Referenzparametern vom eigenen Typ stets der Fall ist, kann eine Methode das Referenzparameter-Objekt aber durchaus auch verändern. Wir könnten z.B. die `Bruch`-Klasse um eine Methode `duplW()` erweitern, die ein Objekt beauftragt, seinen Zähler und Nenner auf ein anderes `Bruch`-Objekt zu übertragen, das per Referenzparameter bestimmt wird:

```
void duplW(Bruch bc) {
    bc.sz(zaehler);
    bc.sn(nenner);
}
```

In folgendem Programm wird das `Bruch`-Objekt `b1` beauftragt, die `duplW()`-Methode auszuführen, wobei als Parameter eine Referenz auf das Objekt `b2` übergeben wird:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class BruchRechnung {
    public static void main(String[] args) {
        Bruch b1 = new Bruch(1, 2, "b1 = ");
        Bruch b2 = new Bruch(5, 6, "b2 = ");

        b1.zeige();
        b2.zeige();
        b1.duplW(b2);
        System.out.println("Nach duplW():\n");
        b2.zeige();
    }
}</pre></td><td><pre>   1
 b1 = -----
         2

         5
 b2 = -----
         6

Nach duplW():

         1
 b2 = -----
         2</pre></td></tr>
</table>

Hier liegt *kein* Verstoß gegen das Prinzip der Datenkapselung vor, weil der Zugriff durch eine klasseneigene Methode erfolgt, die vom Klassendesigner gut konzipiert sein sollte.

4.6.2 Rückgabewerte vom Referenztyp

Bisher haben die innerhalb von Methoden erzeugten Objekte das Ende der Methode nicht überlebt. Weil keine Referenz außerhalb der Methode existierte, wurden die Objekte dem Garbage Collector überlassen. Soll ein methoden-intern kreiertes Objekt überleben, muss also eine Referenz außerhalb der Methode geschaffen werden, was z.B. über einen Rückgabewert mit Referenztyp geschehen kann.
Als Beispiel erweitern wir die `Bruch`-Klasse um die Methode `klone()`, welche ein Objekt beauftragt, einen neuen Bruch anzulegen, mit den eigenen Instanzvariablen zu initialisieren und die Referenz an den Aufrufer abzuliefern:[1]

```
Bruch klone() {
      return new Bruch(zaehler, nenner, etikett);
}
```

[1] Im Zusammenhang mit den Schnittstellen (Interfaces) werden Sie später noch eine etwas bequemere Möglichkeit zum Klonen kennen lernen.

In folgendem Programm wird nur *ein* Objekt erzeugt. Das von `b2` referenzierte Objekt entsteht in der von `b1` ausgeführten Methode `klone()`.

Quellcode	Ausgabe

`class BruchRechnung {`
`    public static void main(String[] args) {`
`        Bruch b1 = new Bruch(1, 2, "b1 = ");`
`        b1.zeige();`
`        Bruch b2 = b1.klone();`
`        b2.zeige();`
`    }`
`}`
	`         1`
	`b1 =  -----`
	`         2`
	
	`         1`
	`b1 =  -----`
	`         2`

4.6.3 this als Referenz auf das aktuelle Objekt

Gelegentlich ist es erforderlich, dass ein Objekt sich selbst ansprechen kann. Dies ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Methode wie eine Referenzvariable benutzt werden kann. In folgendem Beispiel ermöglicht die **this**-Referenz die Verwendung von Formalparameternamen, die mit den Namen von Instanzvariablen übereinstimmen:

```
Bruch(int zaehler, int nenner, String etikett) {
      this.zaehler = zaehler;
      this.nenner  = nenner;
      this.etikett = etikett;
}
```

Später werden Sie noch weit relevanteren **this**-Verwendungsmöglichkeiten begegnen.

4.7 Klassenbezogene Eigenschaften und Methoden

4.7.1 Klassenvariablen

Neben den Instanzvariablen und -methoden unterstützt Java auch *klassenbezogene* Variablen und Methoden. In unserem Beispiel kann eine klassenbezogene Variable dazu dienen, die Anzahl der bisher erzeugten `Bruch`-Objekte aufzunehmen:

```
class Bruch {
    private int zaehler;
    private int nenner = 1;
    private String etikett = "";

    static private int anzahl;

    Bruch(int zpar, int npar, String epar) {
        zaehler = zpar;
        nenner  = npar;
        etikett = epar;
        anzahl++;
    }

    Bruch() {anzahl++;}

    .   .   .
    .   .   .

    static int hanz() {return anzahl;}
}
```

Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, die beim Erzeugen des Objektes auf dem Heap angelegt werden, existiert eine klassenbezogene Variab-

le nur *einmal*. Sie wird beim Laden der Klasse in der Method Area des programmeigenen Speichers angelegt und automatisch genau so initialisiert wie eine Instanzvariable (vgl. Abschnitt 4.2.3). Syntaktisch wird eine Klassenvariable durch den Modifikator **static** gekennzeichnet. Von klasseneigenen Methoden kann sie wie eine Instanzvariable des aktuellen Objekts angesprochen werden. In unserem Beispiel wird die (automatisch auf 0 initialisierte) Klassenvariable `anzahl` in den Konstruktoren inkrementiert.

Sofern Methoden fremder Klassen überhaupt der direkte Zugriff auf eine Klassenvariable gewährt wird, müssen diese dem Variablennamen beim Zugriff einen Präfix aus Klassennamen und Punktoperator voranstellen, z.B.:

```
System.out.println("Bisher wurden "+Bruch.anzahl+" Brueche erzeugt");
```

Soll der direkte Zugriff klasseneigenen Methoden vorbehalten bleiben, ist bei der Deklaration der Modifikator **private** anzugeben, wobei uns erstmals eine Variablendeklaration mit zwei Modifikatoren begegnet (siehe obigen Quellcode).

In der folgenden Tabelle werden wichtige Unterschiede zwischen Klassen- und Instanzvariablen erläutert:

	Instanzvariablen	**Klassenvariablen**
Deklaration	ohne Modifikator **static**	mit Modifikator **static**
Zuordnung	Jedes Objekt besitzt einen eigenen Satz mit allen Instanzvariablen.	Klassenbezogene Variablen sind nur einmal vorhanden.
Existenz	Instanzvariablen werden beim Erzeugen des Objektes angelegt und initialisiert. Sie werden ungültig, wenn das Objekt nicht mehr referenziert ist.	Klassenvariablen werden beim Laden der Klasse angelegt und initialisiert. Sie werden beim Entladen der Klasse ungültig.

Wir verwenden übrigens seit Beginn des Kurses in fast jedem Programm die Klassenvariable **out** aus der Klasse **System** (im Paket **java.lang**). Diese ist vom Referenztyp und zeigt auf ein Objekt der Klasse **PrintStream**, dem wir unsere Ausgabeaufträge übergeben.

4.7.2 Klassenmethoden

Es ist vielfach sinnvoll oder gar erforderlich, einer Klasse Handlungskompetenzen (Methoden) zu verschaffen, die nicht von der Existenz konkreter Objekte abhängen. So muss z.B. beim Start einer Java-Klasse deren Methode **main()** ausgeführt werden, bevor irgend ein Objekt existiert. Das Erzeugen von Objekten gehört gerade zu den typischen Aufgaben der Methode **main()**, wobei es sich nicht unbedingt um Objekte der eigenen Klasse handeln muss.
Sofern Klassenmethoden vorhanden sind, könnte man übrigens auch eine Klasse als *Akteur* auf der objektorientierten Bühne betrachten.
Sind *ausschließlich* Klassenmethoden vorhanden, ist das Erzeugen von Objekten kaum sinnvoll. Es kann durch den Zugriffsmodifikator **private** für den Konstruktor verhindert werden.
Auch das Java-API enthält etliche Klassen, die ausschließlich klassenbezogene Methoden besitzen und damit nicht zum Erzeugen von Objekten konzipiert sind. Mit der Klasse **Math** aus dem API-Paket **java.lang** haben wir ein wichtiges Beispiel bereits in Abschnitt 3.4.1 kennen gelernt. Dort wurde auch demonstriert, wie die **Math**-Klassenmethode **pow()** von einer fremden Klasse benutzt werden kann:

```
System.out.println(4 * Math.pow(2, 3));
```

Dem Methodennamen wird ein Präfix aus Klassennamen und Punktoperator vorangestellt.

Oft ist es sinnvoll, klassenbezogene Kompetenzen mit objektbezogenen zu kombinieren, speziell bei Klassen, die neben Instanzvariablen auch Klassenvariablen besitzen. Da unsere `Bruch`-Klasse mittlerweile über eine Klassenvariable für die Anzahl der erzeugten Objekte verfügt, bietet sich die Definition einer Klassenmethode an, mit der diese Anzahl ermittelt werden kann.

Die Definition einer Klassenmethode wird (analog zum Vorgehen bei Klassenvariablen) mit dem Modifikator **static** eingeleitet, z.B.:

```
static int hanz() {return anzahl;}
```

Mit Hilfe der `Bruch`-Klassenmethode `hanz()` kann unsere Klasse `BruchRechnung` nun feststellen, wie viele Brüche bereits erzeugt wurden:

Quellcode

```
class BruchRechnung {
  public static void main(String[] args) {
    System.out.println(Bruch.hanz() + " Brueche erzeugt");
    Bruch b1 = new Bruch(1, 2, "Bruch 1");
    Bruch b2 = new Bruch(5, 6, "Bruch 2");
    b1.zeige();
    b2.zeige();
    System.out.println(Bruch.hanz() + " Brueche erzeugt");
  }
}
```

Ausgabe

```
0 Brueche erzeugt
          1
 Bruch 1 -----
          2

          5
 Bruch 2 -----
          6

2 Brueche erzeugt
```

Eine Klassenmethode kann natürlich auch von anderen Methoden der eigenen Klasse (objekt- oder klassenbezogen) aufgerufen werden, wobei der Klassenname nicht angegeben werden muss.

In früheren Abschnitten waren mit *Methoden* stets *objektbezogene* Methoden (*Instanzmethoden*) gemeint. Dies soll auch weiterhin so gelten.

4.7.3 Wiederholung zu den Variablentypen in Java

Mittlerweile haben wir verschiedene Variablentypen kennen gelernt, wobei die Sortenbezeichnung unterschiedlich motiviert war. Um einer möglichen Verwirrung vorzubeugen, bietet dieser Abschnitt eine Zusammenfassung bzw. Wiederholung.
Die folgenden Begriffe sollten Ihnen keine Probleme mehr bereiten:

- **Lokale Variablen ...**
  werden in Methoden vereinbart,
  landen auf dem Stack,
  werden **nicht** automatisch initialisiert,
  existieren, bis der innerste Block endet.
  Mögliche Typen: primitive Datentypen, Klassentypen (Objektreferenzen)
- **Instanzvariablen ...**
  werden außerhalb jeder Methodendeklaration vereinbart,
  landen (als Bestandteile von Objekten) auf dem Heap,
  werden automatisch initialisiert,
  existieren, solange das Objekt referenziert ist.
  Mögliche Typen: primitive Datentypen, Klassentypen (Objektreferenzen)

- **Klassenvariablen ...**
  werden außerhalb jeder Methodendeklaration mit dem Modifikator **static** vereinbart,
  landen (als Bestandteile von Klassen) in der method area,
  werden automatisch initialisiert,
  existieren, solange die Klasse geladen ist.
  Mögliche Typen: primitive Datentypen, Klassentypen (Objektreferenzen)
- **Referenzvariablen ...**
  sind durch ihren speziellen *Inhalt* (Referenz auf ein Objekt) gekennzeichnet. Es kann sich sowohl um lokale Variablen (z.B. `b1` in der **main()**-Methode von `BruchRechnung`) als auch um Instanzvariablen (z.B. `etikett` in der `Bruch`-Definition) handeln.

Eine Variable ist gekennzeichnet durch ...

- **Datentyp**
  Hinsichtlich des Variableninhalts sind Werte von primitivem Datentyp und Objektreferenzen zu unterscheiden.
- **Zuordnung**
  Eine Variable kann zu einem Objekt (Instanzvariable), zu einer Klasse (statische Variable) oder zu einer Methode (lokale Variable) gehören. Damit sind weitere Eigenschaften wie Ablageort, Lebensdauer und Initialisierung festgelegt (siehe oben).

Aus den Dimensionen Datentyp und Zuordnung ergibt sich eine (2 × 3)-Matrix zur Einteilung der Java-Variablen, die schon im Abschnitt über elementare Sprachelemente zu sehen war:

		Einteilung nach **Zuordnung**		
		Lokale Variable	Instanzvariable	Klassenvariable
Einteilung nach **Datentyp** (Inhalt)	Primitiver Datentyp			
	Referenzvariable			

4.8 Rekursive Methoden

Innerhalb einer Methode darf man selbstverständlich nach Belieben *andere* Methoden aufrufen. Es ist aber auch zulässig und in vielen Situationen sinnvoll, dass eine Methode *sich selbst* aufruft. Solche *rekursiven* Aufrufe erlauben eine elegante Lösung für ein Problem, das sich sukzessive auf stets einfachere Probleme des selben Typs reduzieren lässt, bis man schließlich zu einem direkt lösbaren reduzierten Problem gelangt. Zu einem rekursiven Algorithmus (per Selbstaufruf einer Methode) existiert stets auch ein iterativer Algorithmus (per Wiederholungsanweisung).
Als Beispiel betrachten wir die Ermittlung des größten gemeinsamen Teilers zu zwei natürlichen Zahlen, die in der `Bruch`-Methode `kuerze()` benötigt wird. Sie haben bereits zwei iterative Realisierungen des Euklidischen Lösungsverfahrens kennen gelernt: In Abschnitt 1.1 wurde ein sehr einfacher Algorithmus benutzt, den Sie später in einer Übungsaufgabe (siehe Abschnitt 3.5.4) durch einen effizienteren Algorithmus ersetzen sollten. In diesem Abschnitt betrachten wir die effizientere Variante, wobei zur Vereinfachung der Darstellung der GGT-Algorithmus vom restlichen Kürzungsverfahren getrennt wird:

```
int ggTi(int a, int b) {
    int rest;
    do {
        rest = a % b;
        a = b;
        b = rest;
    } while (rest > 0);
    return a;
}

void kuerze() {
    if (zaehler != 0) {
        int teiler = ggTi(Math.abs(zaehler), Math.abs(nenner));
        zaehler /= teiler;
        nenner  /= teiler;
    }
}
```

Die iterative GGT-Methode `ggTi()` kann durch folgende rekursive Variante `ggTr()` ersetzt werden:

```
int ggTr(int a, int b) {
    int rest = a % b;
    if (rest == 0)
        return b;
    else
        return ggTr(b, rest);
}
```

Statt eine Schleife zu benutzen, arbeitet die rekursive Methode nach folgender Logik:

- Ist der Parameter *a* durch den Parameter *b* (restfrei) teilbar, dann ist *b* der GGT, und der Algorithmus ist beendet.
- Anderenfalls wird das Problem, den GGT von *a* und *b* zu finden, auf ein einfacheres Problem zurückgeführt, und die Methode `ggTr()` ruft sich selbst mit neuen Aktualparametern erneut auf. Bei der Reduktion, die auch im iterativen Algorithmus analog benutzt wird, wird ein einfacher Satz der mathematischen Zahlentheorie ausgenutzt:

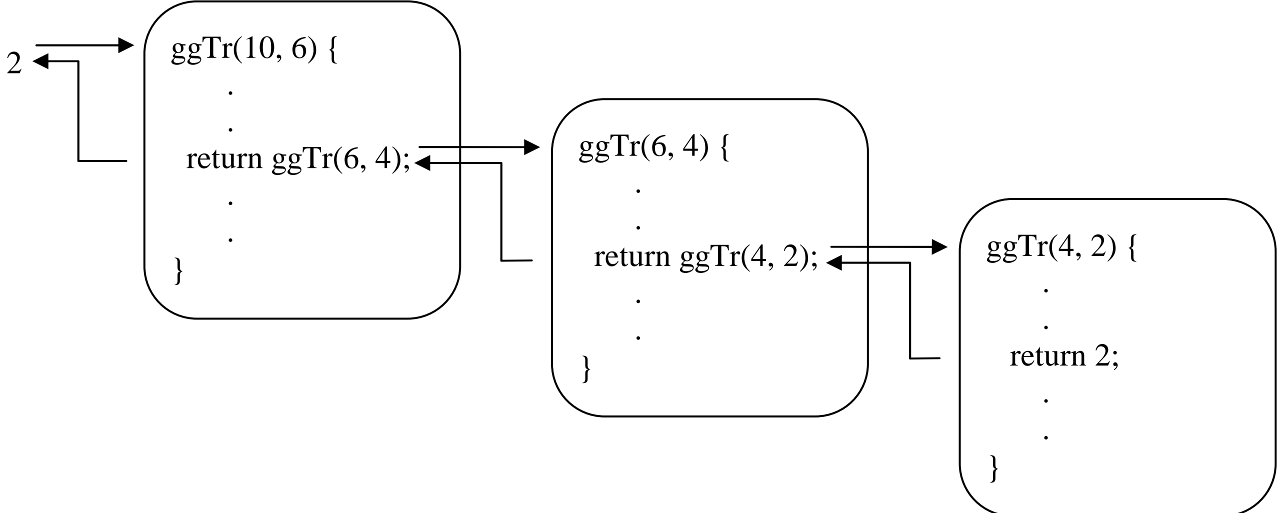
  Für natürliche Zahlen *a* und *b* mit $a > b$ gilt:

  x ist gemeinsamer Teiler von *a* und *b*

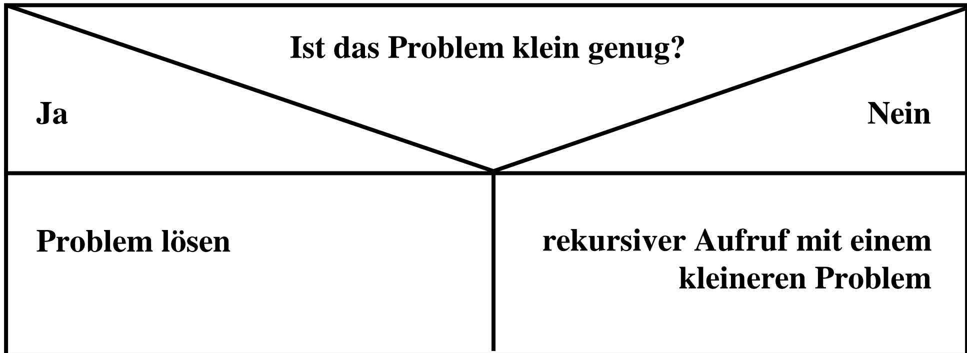
  x ist gemeinsamer Teiler von *b* und *a* % *b*

  Damit ist der GGT von *a* und *b* identisch mit dem GGT von *b* und *a* % *b*.

Wird die Methode `ggTr()` z.B. mit den Argumenten 10 und 6 aufgerufen, kommt es zu folgender Aufrufverschachtelung:



Generell läuft ein rekursiver Algorithmus nach folgender Logik ab:



Wird bei einem fehlerhaften Algorithmus der linke Zweig in diesem Struktogramm nie erreicht, erschöpfen die geschachtelten Methodenaufrufe bald die Stack-Kapazität, und es kommt zu einer Ausnahme, z.B.:

```
Exception in thread "main" java.lang.StackOverflowError
        at Bruch.ggTr(Bruch.java:46)
```

Rekursive Algorithmen lassen sich zwar oft eleganter formulieren als die iterativen Alternativen, benötigen aber durch die hohe Zahl von Methodenaufrufen in der Regel mehr Rechenzeit.

4.9 Aggregation

In der aktuellen `Bruch`-Definition ist eine Instanzvariable vom Referenztyp **String** vorhanden, und in Abschnitt 4.7.3 wurde festgehalten, dass generell für Instanzvariablen auch Klassentypen in Frage kommen. Damit ist es z.B. möglich, vorhandene Klassen als Bestandteile von neuen, komplexeren Klassen zu verwenden. Neben der später noch ausführlich zu behandelnden Vererbung ist diese *Aggregation* von Klassen eine sehr effektive Technik zur Wiederverwendung von Software bzw. zum Aufbau von komplexen Softwaresystemen. Außerdem ist sie im Sinne einer realitätsnahen Modellierung unverzichtbar, denn auch ein reales Objekt (z.B. eine Firma) enthält andere Objekte[1] (z.B. Mitarbeiter, Kunden), die ihrerseits wiederum Objekte enthalten (z.B. ein Gehaltskonto und einen Terminkalender bei den Mitarbeitern) usw.

Wegen der großen Bedeutung der Aggregation soll ihr noch ein ausführliches Beispiel gewidmet werden. Wir erweitern das Bruchrechnungsprogramm um eine Klasse namens `Aufgabe`, die Trainingssitzungen unterstützen soll. In der `Aufgabe`-Klassendefinition tauchen mehrere Instanzvariablen vom Typ `Bruch` auf:

```
class Aufgabe {
   private Bruch b1, b2, lsg, antwort;
   private char op;
   private double dlsg;

   Aufgabe(char op_, int b1Z, int b1N, int b2Z, int b2N) {
      op = op_;
      b1 = new Bruch(b1Z, b1N, "1. Argument:");
      b2 = new Bruch(b2Z, b2N, "2. Argument:");
      lsg = new Bruch(b1Z, b1N, "Das korrekte Ergebnis:");
      antwort = new Bruch();
      init();
   }
```

[1] Die betroffenen Personen mögen den Fachterminus *Objekt* nicht persönlich nehmen.

```
   private void init() {
      switch (op) {
         case '+': lsg.addiere(b2);
                break;
         case '*': lsg.multipliziere(b2);
                break;
      }
      dlsg = (double) lsg.gibZaehler() / lsg.gibNenner();
   }

   boolean korrekt() {
      double dwert = (double) antwort.gibZaehler()/antwort.gibNenner();
      return Math.abs(dwert-dlsg) < 1.0e-5 ? true : false;
   }

   void zeige(int was) {
      switch (was) {
         case 1: System.out.println("   " + b1.gibZaehler() +
                 "                 " + b2.gibZaehler());
                 System.out.println(" ------     " + op + "     -----");
                 System.out.println("   " + b1.gibNenner() +
                 "                 " + b2.gibNenner());
                 break;
         case 2: lsg.zeige(); break;
         case 3: antwort.zeige(); break;
      }
   }

   void frage() {
      System.out.println("\nBerechne bitte:\n");
      zeige(1);
      System.out.print("\nWelchen Zaehler hat dein Ergebnis:     ");
      antwort.setzeZaehler(Simput.gint());
      System.out.println("                                    ------");
      System.out.print("Welchen Nenner hat dein Ergebnis:      ");
      antwort.setzeNenner(Simput.gint());
   }

   void neueWerte(char op_, int b1Z, int b1N, int b2Z, int b2N) {
      op = op_;
      b1.setzeZaehler(b1Z); b1.setzeNenner(b1N);
      b2.setzeZaehler(b2Z); b2.setzeNenner(b2N);
      lsg.setzeZaehler(b1Z); lsg.setzeNenner(b1N);
      init();
   }
}
```

In einer `Aufgabe` dienen die `Bruch`-Objekte folgenden Zwecken:

- `b1` und `b2` werden dem Anwender (in der Methode `frage()`) im Rahmen eines Arbeitsauftrags vorgelegt, z.B. zum Addieren.
- In `antwort` landet der Lösungsversuch des Anwenders.
- In `lsg` steht das korrekte Ergebnis

In folgendem Programm wird die neue Klasse für ein Bruchrechnungstraining eingesetzt:

```
class BruchRechnung {
   public static void main(String[] args) {
      Aufgabe auf = new Aufgabe('+', 1, 2, 2, 5);
      auf.frage();
      if (auf.korrekt())
         System.out.println(" Gut!");
      else
         auf.zeige(2);
      auf.neueWerte('*', 3, 4, 2, 3);
      auf.frage();
```

```
        if (auf.korrekt())
            System.out.println(" Gut!");
        else
            auf.zeige(2);
    }
}
```

Man kann immerhin schon ahnen, wie die praxistaugliche Endversion einmal arbeiten wird:

```
Berechne bitte:

   1                2
 ------     +     -----
   2                5

Welchen Zaehler hat dein Ergebnis:     3
                                     ------
Welchen Nenner hat dein Ergebnis:      7
                                 9
 Das korrekte Ergebnis: -----
                                10


Berechne bitte:

   3                2
 ------     *     -----
   4                3

Welchen Zaehler hat dein Ergebnis:     6
                                     ------
Welchen Nenner hat dein Ergebnis:      12
 Gut!
```

4.10 Übungsaufgaben zu Abschnitt 4

1) Erstellen Sie eine Klasse für zweidimensionale Vektoren, die mindestens über Instanzmethoden mit folgenden Leistungen erfügt:

- **Länge** ermitteln

  Der Betrag eines Vektors $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ist definiert durch:

  $$|x| := \sqrt{x_1^2 + x_2^2}$$

  Verwenden Sie die Klassenmethode **Math.sqrt()**, um die Quadratwurzel aus einer **double**-Zahl zu berechnen.

- Vektor auf Länge 1 **normieren**
  Dazu dividiert man beide Komponenten durch die Länge des Vektors, denn mit

  $\tilde{x} := (\tilde{x}_1, \tilde{x}_2)$ sowie $\tilde{x}_1 := \frac{x_1}{\sqrt{x_1^2 + x_2^2}}$ und $\tilde{x}_2 := \frac{x_2}{\sqrt{x_1^2 + x_2^2}}$ gilt:

  $$|\tilde{x}| := \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

- Vektoren (komponentenweise) **addieren**

  Die Summe der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

$$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

- **Skalarprodukt** zweier Vektoren ermitteln

  Das Skalaprodukt der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

$$x \cdot y := x_1 y_1 + x_2 y_2$$

- **Winkel** zwischen zwei Vektoren in Grad ermitteln
  Für den Kosinus des Winkels, den zwei Vektoren x und y im mathematischen Sinn (links herum) einschließen, gilt:

$$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$

  Um daraus den Winkel in Grad zu ermitteln, können Sie die Klassenmethoden **Math.acos()** und **Math.toDegrees()** verwenden.

- **Rotation** eines Vektors um einen bestimmten Winkelgrad
  Mit Hilfe der Rotationsmatrix

$$D := \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

  kann der Vektor x um den Winkel α (im Bogenmaß!) gedreht werden:

$$x' = D\,x = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos(\alpha)\,x_1 - \sin(\alpha)\,x_2 \\ \sin(\alpha)\,x_1 + \cos(\alpha)\,x_2 \end{pmatrix}$$

  Die trigonometrischen Funktionen können mit den Klassenmethoden **Math.cos()** und **Math.sin()** berechnet werden. Für die Umwandlung von Winkelgraden in Bogenmaß steht die Methode **Math.toRadians()** bereit.

Erstellen Sie ein Demonstrationsprogramm, das Ihre Vektor-Klasse verwendet.
In folgendem Beispiel wurde für eine formatiere Ausgabe der **double**-Werte gesorgt:

```
Vektor 1:               (1,00; 0,00)
Vektor 2:               (1,00; 1,00)

Laenge von Vektor 1:    1,00
Laenge von Vektor 2:    1,41

Winkel:                 45,00 Grad

Um wie viel Grad soll Vektor 2 gedreht werden: 45

Neuer Vektor 2          (0,00; 1,41)
Neuer Vektor 2 normiert (0,00; 1,00)

Summe der Vektoren      (1,00; 1,00)
```

Dazu wurde ein Objekt aus der Klasse **DecimalFormat** im Paket **java.text** verwendet:

```
DecimalFormat df = new DecimalFormat("0.00");
```

Seine **format()**-Methode liefert zu einem **double**-Wert einen String unter Verwendung des im Konstruktor angegebenen Formates, z.B.:

```
System.out.println("Laenge von Vektor 1:     "
                  + df.format(v1.betrag()));
```

Um die Klasse nutzen zu können, müssen Sie entweder den Namen vollqualifiziert (mit Paketnamen, siehe unten) angeben (**java.text.DecimalFormat**) oder am Anfang Ihres Demonstrationsprogramms die Klasse **DecimalFormat** importieren:

```
import java.text.DecimalFormat;
```

2) Experimentieren Sie mit dem Garbage Collector, indem Sie in einer Java-Anwendung zahlreiche Objekte erzeugen und sofort wieder frei geben. Die Objekte sollten aus einer Klasse mit folgenden Merkmalen stammen:

- Jedes Objekt erhält eine fortlaufende Nummer, die per Konstruktor-Parameter vergeben und in einer **int**-Instanzvariablen abgelegt wird.
- Der Konstruktor macht eine Kontrollausgabe mit der Objektnummer.
- Es wird eine Instanzmethode **finalize()** definiert, die das Ableben eines Objektes durch Konsolenausgabe seiner Nummer protokolliert (siehe Hinweise in Abschnitt 4.4.3). Ist eine Methode dieses Namens vorhanden, wird sie vom Garbage Collector vor dem Entfernen eines Objektes ausgeführt.

Untersuchen Sie auch, welchen Effekt der explizite Aufruf des Garbage Collectors am Ende des Testprogramms hat. Starten Sie den Müllsammler mit der **System**-Klassenmethode **gc()**:

```
System.gc();
```

3) Erstellen Sie eine Variante des Programms zu Aufgabe 2, die ohne Konstruktor-Parameter auskommt.

4) Erstellen Sie eine Klasse mit einer statischen Methode zur Berechnung der Fakultät über einen rekursiven Algorithmus. Erstellen Sie eine Test-Klasse, welche die rekursive Fakultätsmethode benutzt.

5 Elementare Klassen

In diesem Abschnitt wird gewissermaßen die objektorientierte Fortsetzung der elementaren Sprachelemente aus Abschnitt 3 präsentiert. Es werden wichtige Bausteine für Java-Programme (z.B. Felder, Zeichenketten) behandelt, die als *Klassen* realisiert sind. Sie gehören entweder zur Sprache Java selbst, oder zum **Java-API** (**A**pplication **P**rogramming **I**nterface), einer mächtigen Bibliothek mit Klassen für praktisch alle Standardaufgaben der Programmentwicklung (z.B. Stringverarbeitung, Dateizugriffe, Netzverbindungen), die integraler Bestandteil des Java-SDK ist. Die in Abschnitt 5 vorgestellten Klassen stehen *immer* zur Verfügung, so dass es zur akademischen Frage wird, ob man sie der *Programmiersprache* Java oder der Java-*Bibliothek* zuordnet.
Weil die API-Klassen in so genannten *Paketen* organisiert sind, folgt erst im Zusammenhang mit dieser wichtigen Software-Strukturierungsmöglichkeit eine systematische Beschreibung der Java-Klassenbibliothek. U.a. wird dort geklärt, welche API-Klassen stets verfügbar sind, und welche importiert werden müssen.

5.1 Arrays (Felder)

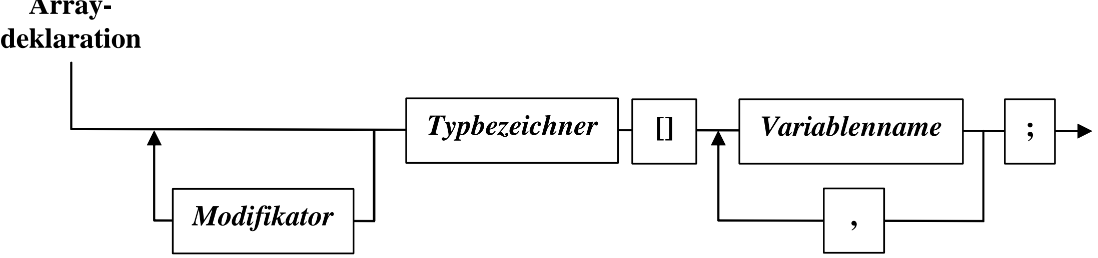
Ein Feld bzw. Array enthält eine feste Anzahl von Elementen des selben Datentyps, die gemeinsam verarbeitet (z.B. als Aktualparameter), aber auch einzeln über einen Index angesprochen werden können.
Hier ist als Beispiel ein eindimensionales Feld namens `uni` mit 5 Elementen vom Typ **int** zu sehen:

uni[0]	uni[1]	uni[2]	uni[3]	uni[4]
1950	1991	1997	2057	2005

Beim Zugriff auf ein einzelnes Element gibt man nach dem Arraynamen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung mit der 0 beginnt.
Weil der Index auch über eine *Variable* geliefert werden kann, bietet ein Array z.B. im Zusammenhang mit einer Wiederholungsanweisung erhebliche Vorteile gegenüber einer Anzahl von selbständigen Variablen.
Wir befassen uns zunächst mit *eindimensionalen* Arrays, behandeln später aber auch den mehrdimensionalen Fall.

5.1.1 Arrays deklarieren

Arrays sind in Java stets *Objekte*, so dass diese eigentlich recht elementaren Programmbestandteile erst jetzt behandelt werden. Eine Array-Variable ist in Java vom Referenztyp und wird folgendermaßen deklariert:



Im Vergleich zu der bisher bekannten Variablendeklaration (ohne Initialisierung) ist hinter dem Typbezeichner zusätzlich ein Paar mit eckigen Klammern anzugeben. Dieses darf alternativ auch hinter dem Variablennamen stehen. In unserem Beispiel könnte die Array-Variable `uni` also z.B. folgendermaßen deklariert werden:

```
int[] uni;
```

Bei der Deklaration entsteht nur eine Referenzvariable, nicht jedoch der Array selbst. Daher ist auch keine Array-Größe anzugeben.

5.1.2 Arrays erzeugen

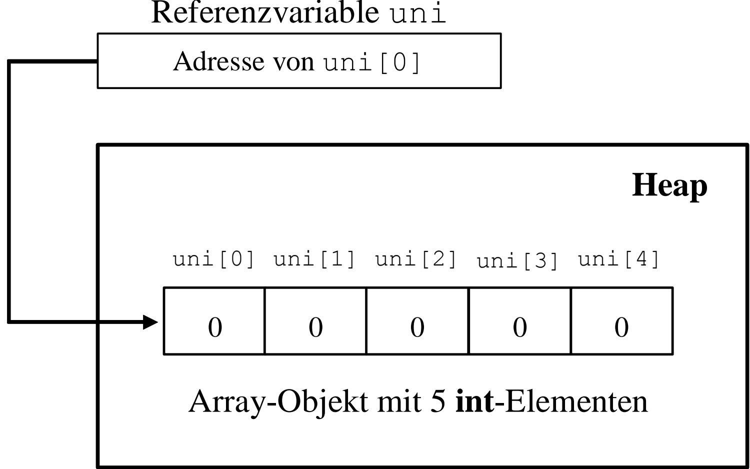
Mit Hilfe des **new**-Operators erzeugt man ein Array-Objekt mit bestimmter Größe auf dem Heap und schreibt seine Adresse in die Referenzvariable, z.B.:

```
uni = new int[max+1];
```

Im **new**-Operanden muss hinter dem Datentypen zwischen eckigen Klammern die Anzahl der Elemente festgelegt werden, was über einen beliebigen Ausdruck mit ganzzahligem Wert (Typ **byte**, **short**, **int, long** oder **char**) geschehen kann.
In Java kann man also die Länge eines Arrays zur Laufzeit festlegen, z.B. in Abhängigkeit von einer Benutzereingabe. Existiert ein Array-Objekt erst einmal, kann die Anzahl seiner Elemente allerdings nicht mehr geändert werden. Wer noch dynamischere Datenstrukturen benötigt, kann z.B. die API-Klasse **Vector** benutzen (siehe Abschnitt 5.3.1) oder selbst eine so genannte *verkettete Liste* aufbauen.
Die Deklaration einer Array-Referenzvariablen und die Erstellung des Array-Objektes kann man natürlich auch in einer Anweisung erledigen, z.B.:

```
int[] uni = new int[5];
```

Mit der Verweisvariablen `uni` und dem referenzierten Array-Objekt auf dem Heap haben wir insgesamt folgende Situation:



Weil es sich bei den Array-Elementen um Instanzvariablen eines *Objektes* handelt, erfolgt eine automatische Initialisierung nach den Regeln von Abschnitt 4.1.3. Die **int**-Elemente im letzten Beispiel werden also mit 0 initialisiert.

Aus der Objekt-Natur eines Arrays folgt unmittelbar, dass er vom Garbage Collector entsorgt wird, wenn keine Referenz mehr vorliegt. Um eine Referenzvariable aktiv von einem Array zu „entkoppeln“, kann man ihr z.B. den Wert **null** (Zeiger auf nichts) oder aber eine alternative Referenz zuweisen. Gemäß ihrer Deklaration kann eine Array-Referenzvariable auf einen beliebigen Array mit Elementen vom vereinbarten Typ zeigen.
Es ist auch ohne weiteres möglich, dass mehrere Referenzvariablen auf dasselbe Array-Objekt zeigen, z.B.:

Quellcode	Ausgabe

`class ArrayDetails {`
`  public static void main(String[] args){`
`    int[] x = new int[3], y;`
`    x[0] = 1; x[1] = 2; x[2] = 3;`
`    y = x; //y zeigt nun auf das selbe Array-Objekt wie x`
`    y[0] = 99;`
`    System.out.println(x[0]);`
`  }`
`}`
	`99`

5.1.3 Arrays benutzen

Der Zugriff auf die Elemente des Array-Objektes geschieht über die Referenzvariable, an die zwischen eckigen Klammern ein passender Index angehängt wird. Als Index ist ein beliebiger Ausdruck mit ganzzahligem Wert erlaubt (Typ **byte**, **short**, **int, long** oder **char**), wobei natürlich die Feldgrenzen zu beachten sind.
In folgender Schleife wird pro Durchgang ein zufällig gewähltes `uni`-Element (s.u.) inkrementiert:

```
for (i = 1; i <= drl; i++)
   uni[zzg.nextInt(5)]++;
```

Den Indexwert liefert hier ein Methodenaufruf mit Rückgabetyp **int**.

Wie in vielen anderen Programmiersprachen hat auch in Java das erste von *n* Feldelementen die Nummer 0 und folglich das letzte die Nummer *n* - 1. Damit existiert nach

```
int[] uni = new int[5];
```

kein Element `uni[5]`. Hier können sich leicht Fehler einschleichen, die das Laufzeitsystem veranlassen, eine Ausnahme zu melden und die verantwortliche Methode zu beenden, z.B.:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at UniRand.main(UniRand.java:14)
```

Man kann sich in Java generell darauf lassen, dass jede Überschreitung von Feldgrenzen verhindert wird, so dass es nicht zur unbeabsichtigten Verletzung anderer Speicherbereiche kommt. In anderen Programmiersprachen (z.B. C/C++) fehlt eine garantierte Feldgrenzenüberwachung. Hier führt ein entsprechender Programmierfehler im günstigen Fall zu einem Absturz, weil das Betriebssystem eine Speicherschutzverletzung feststellt. In ungünstigen Fällen kann es aber auch zu einem kaum nachvollziehbaren Fehlverhalten kommen.

Warum das erste Array-Element die Nummer 0 besitzt, wird durch folgende Überlegungen verständlich:

- Die Referenzvariable `uni` zeigt auf das *erste* Element des Array-Objektes.
- Der Operator **[]** wird auch als *Offset-Operator* bezeichnet. Bei einem Offset von 0 bewegt man sich *nicht* vom Anfang des Array-Objektes weg, spricht also das erste Element an.

Einem Programm muss natürlich die Länge aller Arrays bekannt sein. Damit bei Arrays mit problemabhängig (z.B. durch Benutzerentscheidung) festgelegter Länge keine lästige Routinearbeit entsteht, speichert Java die Längeninformation eines Array-Objekts automatisch in einer finalisierten Instanzvariablen namens **length**:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Prog {
  public static void main(String[] args) {
    System.out.print("Laenge des Vektors: ");
    int[] wecktor = new int[Simput.gint()];
    System.out.println();

    for(int i = 0; i < wecktor.length; i++) {
      System.out.print("Wert von Element "+i+": ");
      wecktor[i] = Simput.gint();
    }
    System.out.println();

    for(int i = 0; i < wecktor.length; i++)
      System.out.println(wecktor[i]);
  }
}
```

</td><td>

```
Laenge des Vektors: 3

Wert von Element 0: 7
Wert von Element 1: 13
Wert von Element 2: 4711

7
13
4711
```

</td></tr>
</table>

Auch beim Entwurf von *Methoden* mit Array-Parametern ist es von Vorteil, dass die Länge eines übergebenen Arrays ohne entsprechenden Extra-Parameter in der Methode bekannt ist.

5.1.4 Beispiel: Beurteilung des Java-Pseudozufallszahlengenerators

Als Zwischenbilanz kann man feststellen: Im Vergleich zu 5 einzelnen **int**-Variablen haben wir im aktuellen Beispiel mit dem Array einen deutlich verringerten Aufwand bei der Deklaration. Insbesondere beim Einsatz in einer Schleifenkonstruktion erweist sich die Ansprache der einzelnen Elemente über einen Index als überaus praktisch.

Die bisher wiedergegebenen Anweisungen lassen sich leicht zu einem Programm erweitern, das die Verteilungsqualität des Java-**Pseudozufallszahlengenerators** überprüft. Ein solcher Generator produziert Folgen von Zahlen mit einem bestimmten Verteilungsverhalten. Obwohl eine Serie perfekt von ihrem Startwert abhängt, kann sie in der Regel echte Zufallszahlen ersetzen. Manchmal ist es sogar von Vorteil, eine Serie über ihren festen Startwert reproduzieren zu können. Meist verwendet man aber variable Startwerte, z.B. abgeleitet aus einer Zeitangabe. Der Einfachheit halber redet man oft von *Zufallszahlen* und lässt den *Pseudo*-Zusatz weg.
Nach der folgenden Anweisung zeigt die Referenzvariable `zzg` auf ein Objekt der Klasse **Random** aus dem API-Paket **java.util**:

```
java.util.Random zzg = new java.util.Random();
```

Durch Verwendung des parameterfreien **Random**-Konstruktors entscheidet man sich für die Anzahl der Sekunden seit dem 1.1.1970, 00.00 Uhr, als Starwert für den Pseudozufall. Lieferant dieses Wertes ist die statische Methode **currentTimeMillis()** der Klasse **System** im API-Paket **java.lang**.[1]
Statt obiger Anweisung könnte man etwas umständlicher auch schreiben:

```
java.util.Random zzg = new java.util.Random(System.currentTimeMillis());
```

[1] Diese Klasse leistet uns bekanntlich über die **println()**-Methode ihrer Klassenvariablen **out** bei der Konsolenausgabe schon geraume Zeit wertvolle Dienste. Ihre Methode **currentTimeMillis()** ist u.a. bei der Messung von Zeitdifferenzen sehr nützlich.

Das angekündigte Programm zieht 10000 Zufallszahlen und überprüft deren Verteilung:

```
class UniRand {
   public static void main(String[] args) {
      final int drl = 10000;
      int i;

      int[] uni = new int[5];
      java.util.Random zzg = new java.util.Random();

      for (i = 1; i <= drl; i++)
         uni[zzg.nextInt(5)]++;

      System.out.println("Absolute Haeufigkeiten:");
      for (i = 0; i < 5; i++)
         System.out.print(uni[i] + " ");

      System.out.println("\n\nRelative Haeufigkeiten:");
      for (i = 0; i < 5; i++)
         System.out.print((double)uni[i]/drl + " ");
   }
}
```

In folgender Schleife werden mit der **Random**-Methode **nextInt()** Zufallszahlen aus der Menge {0, 1, 2, 3, 4} gezogen, die dann als Array-Index dienen. Folglich wird bei jedem Schleifendurchgang ein zufällig gewähltes `uni`-Element inkrementiert:

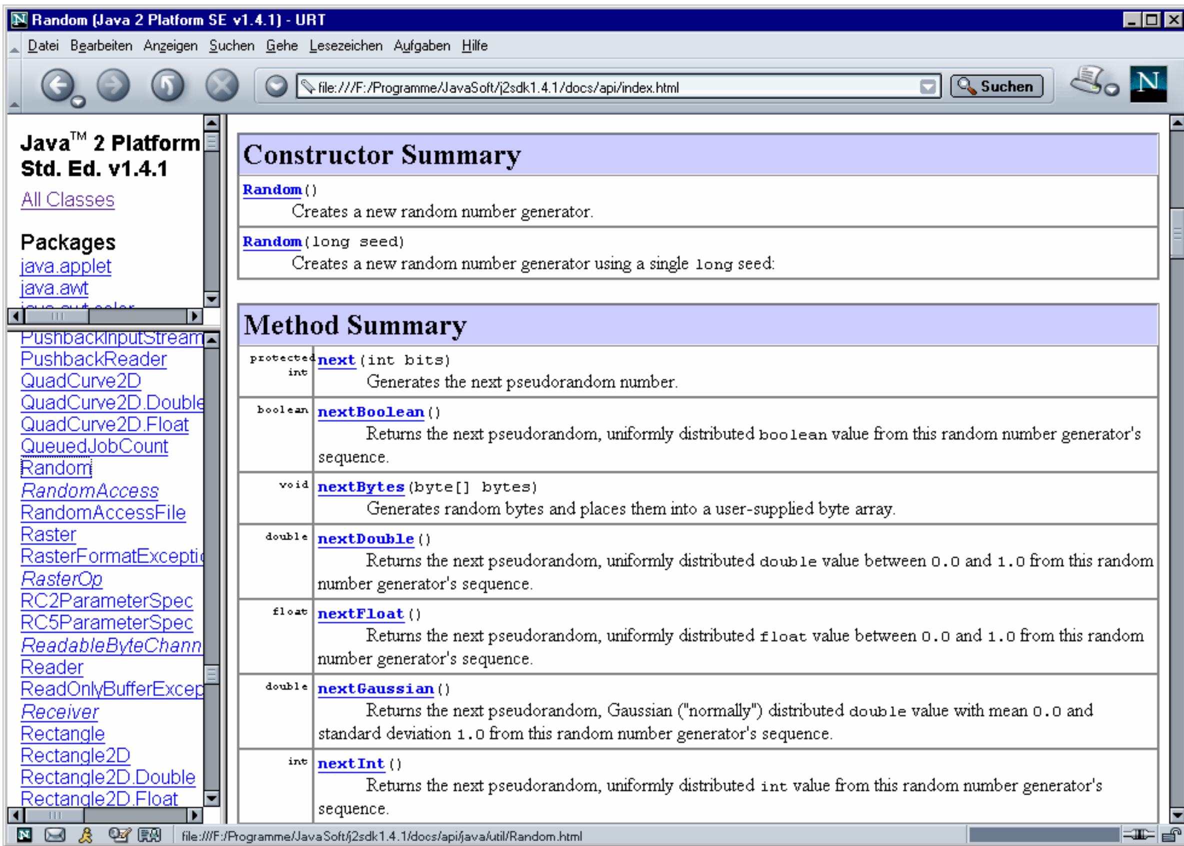
```
for (i = 1; i <= drl; i++)
   uni[zzg.nextInt(5)]++;
```

Ein guter Pseudozufallszahlengenerator liefert eine annähernde Gleichverteilung über der Trägermenge. Das **Random**-Objekt wird diesem Anspruch durchaus gerecht, wie das folgende Ergebnis-Beispiel zeigt:

```
Absolute Haeufigkeiten:
1950 1991 1997 2057 2005

Relative Haeufigkeiten:
0.195 0.1991 0.1997 0.2057 0.2005
```

Über die im Beispielprogramm verwendeten Java-Klassen **java.util.Random** und **java.lang.System** können Sie sich mit der JDK-Dokumentation informieren:

- Öffnen Sie die HTML-Startseite der Online-Dokumentation über den Eintrag **Java 2 SDK Documentation** in der Java-2-SDK - Programmgruppe.
- Klicken Sie auf den Link **Java 2 Platform API Specification**.
- Klicken Sie im linken oberen Frame auf **All Classes**.
- Klicken Sie im linken unteren Frame auf den Klassennamen **Random**.
  Anschließend erscheinen im rechten Frame detaillierte Informationen über die Klasse **Random**:



Statt explizit ein **Random**-Objekt zu erzeugen und mit der Produktion von Pseudozufallszahlen zu beauftragen, kann man auch die statische Methode **random()** aus der Klasse **Math** benutzen, die gleichverteilte **double**-Werte aus dem Intervall [0, 1) liefert, z.B.:

```
uni[(int) (Math.random()*5)]++;
```

Im Hintergrund erzeugt die Methode beim ihrem ersten Aufruf ein **Random**-Objekt über den parameterfreien Konstruktor:

```
new java.util.Random()
```

5.1.5 Initialisierungslisten

Bei Arrays mit wenigen Elementen ist die Möglichkeit von Interesse, beim Deklarieren der Referenzvariablen eine Initialisierungsliste anzugeben und das Array-Objekt dabei implizit (ohne Verwendung des **new**-Operators) zu erzeugen, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class ArrayDetails {
   public static void main(String[] args) {
      int[] wecktor = {1, 2, 3};
      System.out.println(wecktor[2]);
   }
}</pre></td><td>3</td></tr>
</table>

Die Deklarations- und Initialisierungsanweisung

```
int[] wecktor = {1, 2, 3};
```

ist äquivalent zu:

```
int[] wecktor = new int[3];
wecktor[0] = 1;
wecktor[1] = 2;
wecktor[2] = 3;
```

5.1.6 Objekte als Array-Elemente

Für die Elemente eines Arrays sind natürlich auch Referenztypen erlaubt. In folgendem Beispiel wird ein Array mit `Bruch`-Objekten erzeugt:

Quellcode	Ausgabe

<pre>class ArrayDetails {
   public static void main(String[] args) {
      Bruch b1 = new Bruch(1, 2, "b1 = ");
      Bruch b2 = new Bruch(5, 6, "b2 = ");
      Bruch[] bruvek = {b1, b2};
      bruvek[1].zeige();
   }
}</pre>
	<pre> 5
	 b2 = -----
	        6</pre>

Im nächsten Abschnitt lernen wir einen wichtigen Spezialfall von Arrays mit Referenztyp-Elementen kennen. Dort zeigen die Elementvariablen wiederum auf Arrays, so dass mehrdimensionale Felder entstehen.

5.1.7 Mehrdimensionale Felder

Speziell in der Matrixalgebra, aber auch auf vielen anderen Anwendungsbereichen, werden *mehrdimensionale* Felder benötigt. Ein zweidimensionales Feld wird in Java als *Array of Arrays* realisiert, z.B.:

Quellcode	Ausgabe

<pre>class ArrayDetails {
   public static void main(String[] args) {
      int[][] matrix = new int[4][3];
 	
      System.out.println("matrix.length   = "+
         matrix.length);
      System.out.println("matrix[0].length = "+
         matrix[0].length+"\n");
 	
      for(int i=0; i < matrix.length; i++) {
         for(int j=0; j < matrix[i].length; j++) {
          matrix[i][j] = (i+1)*(j+1);
          System.out.print(" "+ matrix[i][j]);
         }
         System.out.println();
      }
   }
}</pre>
	<pre>matrix.length = 4
	matrix[0].length = 3
	
	 1 2 3
	 2 4 6
	 3 6 9
	 4 8 12</pre>

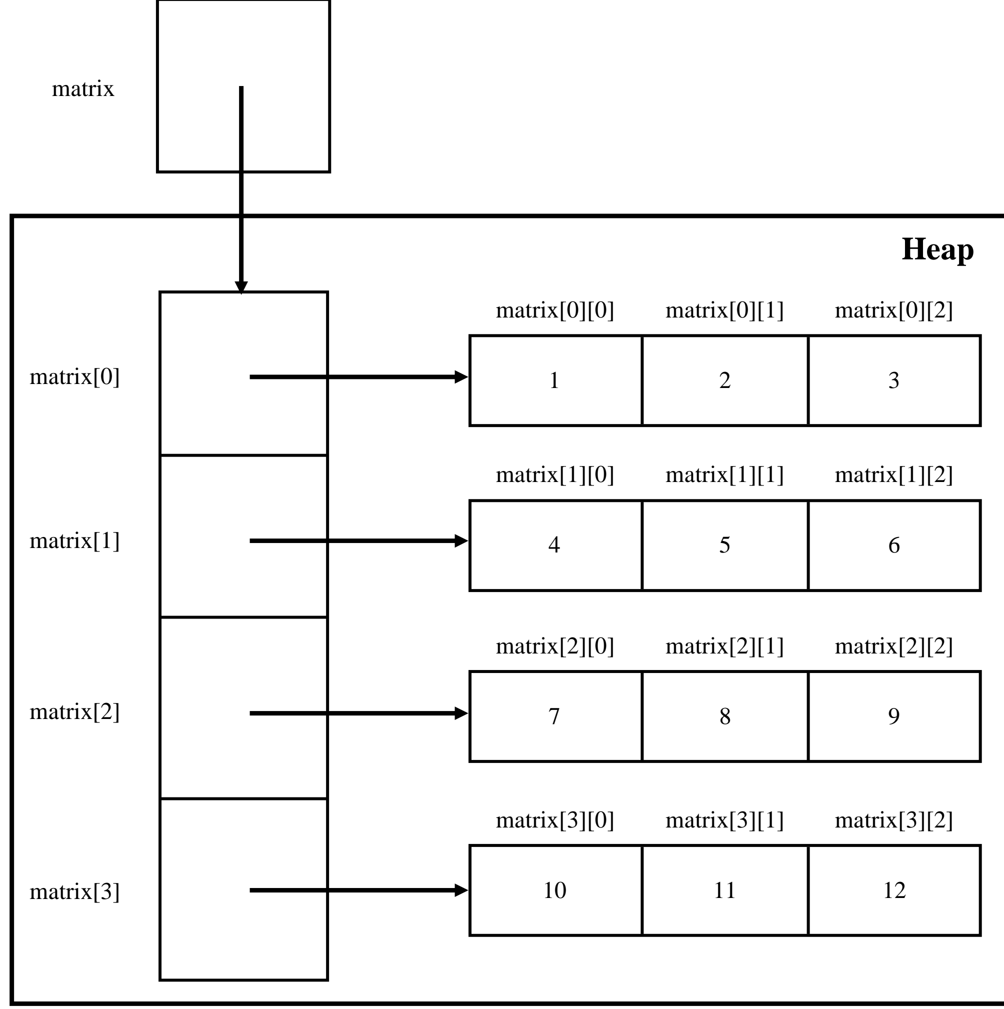
Dieses Verfahren lässt sich beliebig fortsetzen, um Arrays mit höherer Dimensionalität zu erzeugen.

Im Beispiel wird ein Array-Objekt namens `matrix` mit 4 Elementen `matrix[0]` bis `matrix[3]` erzeugt, bei denen es sich um Arrays mit 3 **int**-Elementen handelt.
Wir haben eine zweidimensionale Matrix zur Verfügung, auf deren Zellen man per Doppelindizierung zugreifen kann, wobei sich die Syntax leicht von der mathematischen Schreibweise unterscheidet.

Man kann aber auch mit einfacher Indizierung eine komplette Zeile ansprechen, was in obigem Programm z.B. geschieht, um die Länge der eindimensionalen Zeilen-Arrays zu ermitteln:

```
matrix[i].length
```

In der folgenden Abbildung, die sich an Mössenböck (2001, S. 104) orientiert, wird mit einem gerichteten Pfeil die Relation „Referenzvariable zeigt auf Array-Objekt“ ausgedrückt:



Auch im mehrdimensionalen Fall können Initialisierungslisten eingesetzt werden, z.B.:

Quellcode	Ausgabe

`class ArrayDetails {`
`    public static void main(String[] args) {`
`        int[][] matrix = {{1}, {1,2}, {1, 2, 3}};`
 	
`        for(int i=0; i < matrix.length; i++) {`
`            for(int j=0; j < matrix[i].length; j++)`
`                System.out.print(matrix[i][j]+" ");`
`            System.out.println();`
`            }`
`    }`
`}`
	1
	1 2
	1 2 3

Im letzten Beispielprogramm wird auch die in Java gegebene Flexibilität genutzt, eine Matrix mit unterschiedlich langen Zeilen anzulegen. So kann man sich etwa bei einer symmetrischen Matrix platzsparend darauf beschränken, die untere Dreiecksmatrix abzuspeichern.

Während in C++ beim *dynamischen Erzeugen* eines mehrdimensionalen Feldes per **new**-Operator alle Dimensionen mit Ausnahme der ersten eine feste Feldgrenze besitzen müssen, können in Java *alle* Feldgrenzen dynamisch gesetzt werden, z.B.:

<table>
<tr><th>Teillösung für eine dynamische mehrdimensionale Matrix in C++</th><th>Dynamische mehrdimensionale Matrix in Java</th></tr>
<tr><td><pre>#include <iostream>
using namespace std;

void main() {
    int z;
    cout << "Zeilen: ";
    cin >> z;
    int (*pti)[10] = new int[z][10];
}</pre></td><td><pre>class DynArray {
   public static void main(String[] args) {
       int z, s;
       System.out.print("Zeilen: ");
       z = Simput.gint();
       System.out.print("Spalten: ");
       s = Simput.gint();
       int[][] mat = new int[z][s];
   }
}</pre></td></tr>
</table>

5.1.8 Übungsaufgaben zu Abschnitt 5.1

1) Erstellen Sie ein Java-Programm, das 6 Lottozahlen (von 1 bis 49) zieht und sortiert ausgibt. Zum Sortieren können Sie z.B. das (sehr einfache) **Auswahlverfahren** (Selection Sort) benutzen:

- Für den Ausgangsvektor mit den Elementen 0, .., *k-1* wird das Minimum gesucht und an den linken Rand befördert. Dann wird der Vektor mit den Elementen 1 bis *k-1* analog behandelt, usw.
- Bei jeder Teilaufgabe muss man das kleinste Element eines Vektors finden, was auf folgende Weise geschehen kann:
    - Man geht davon aus, das Element am linken Rand sei das kleinste (genauer: *ein* Minimum).
    - Es wird sukzessive mit seinen rechten Nachbarn verglichen. Ist das Element an der Position *i* kleiner, so tauscht es mit dem „Linksaußen“ seinen Platz.
    - Nun steht am linken Rand ein Element, das die anderen Elemente mit Positionen kleiner oder gleich *i* nicht übertrifft. Es wird nun sukzessive mit den Elementen an den Positionen ab *i*+1 verglichen.
    - Nachdem auch das Element an der letzten Position mit dem Element am linken Rand verglichen worden ist, steht mit Sicherheit am linken Rand ein Element, zu dem sich kein kleineres findet.

2) Erstellen Sie ein Programm zur Primzahlensuche mit dem **Sieb des Eratosthenes**. Dieser Algorithmus reduziert sukzessive eine Menge von Primzahl-Kandidaten, die initial alle natürlichen Zahlen bis zu einer Obergrenze *K* enthält, also {1, 2, 3, ..., *K*}.

- Im ersten Schritt werden alle echten Vielfachen der Basiszahl 2 (also 4, 6, ...) aus der Kandidatenmenge gestrichen.
- Dann geschieht iterativ folgendes:
    - Als neue Basis *b* wird die kleinste Zahl gewählt, welche folgende Bedingungen erfüllt:

        b ist größer als die vorherige Basiszahl.
        b ist im bisherigen Verlauf nicht gestrichen worden.
    - Die echten Vielfachen der neuen Basis (also 2·*b*, 3·*b*, ...) werden aus der Kandidatenmenge gestrichen.

- Das Verfahren kann enden, wenn für eine neue Basis b gilt:

  $$b > \sqrt{K}$$

  Es kann keine Streichkandidaten mehr geben, denn: Ist eine natürliche Zahl $n \leq K$ *keine* Primzahl, dann gibt es natürliche Zahlen $x, y \in \{2, 3, ...\}$ mit

  $$n = x \cdot y, x \leq \sqrt{K} \text{ oder } y \leq \sqrt{K}$$

  Wegen $\sqrt{K} < b$ ist n bereits gestrichen worden. Die Kandidatenmenge enthält also nur noch Primzahlen.

Sollen z.B. alle Primzahlen kleiner oder gleich 18 bestimmt werden, so startet man mit folgender Kandidatenmenge:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Im ersten Schritt werden die echten Vielfachen der Basis 2 gestrichen:

1	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~

Als neue Basis wird die Zahl 3 gewählt (> 2, nicht gestrichen). Ihre echten Vielfachen werden gestrichen:

1	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~

Als neue Basis wird die Zahl 5 gewählt (> 3, nicht gestrichen). Allerdings ist 5 größer als $\sqrt{18}$ (≈ 4,24) und der Algorithmus daher bereits beendet. Die Primzahlen kleiner oder gleich 18 lauten also:

1, 2, 3, 5, 7, 11, 13 und 17

3) Erstellen Sie eine Klasse für zweidimensionale Matrizen mit Elementen vom Typ **float**. Implementieren Sie eine Methode zum Transponieren einer Matrix und vielleicht noch andere Methoden für klassische Aufgaben der Matrixalgebra.

5.2 Zeichenketten

Java bietet für den Umgang mit Zeichenketten zwei Klassen an:

- **String**
  String-Objekte können nach dem Erzeugen nicht mehr geändert werden. Diese „Read Only“ - Klasse ist für den *lesenden* Zugriff auf Zeichenketten optimiert.
- **StringBuffer**
  Für variable Zeichenketten sollte unbedingt die Klasse **StringBuffer** verwendet werden, weil deren Objekte nach dem Erzeugen beliebig verändert werden können.

5.2.1 Die Klasse String für konstante Zeichenkette

5.2.1.1 Implizites und explizites Erzeugen von String-Objekten

In der folgenden Deklarations- und Initialisierungsanweisung

```
String s1 = "abcde";
```

wird:

- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt auf dem Heap erzeugt,
- die Adresse des (neuen) Heap-Objektes in der Referenzvariablen abgelegt

Soviel objektorientierten Hintergrund sieht man der angenehm einfachen Anweisung auf den ersten Blick nicht an. In Java sind jedoch auch Zeichenketten*literale* als **String**-Objekte realisiert, so dass z.B.

```
"abcde"
```

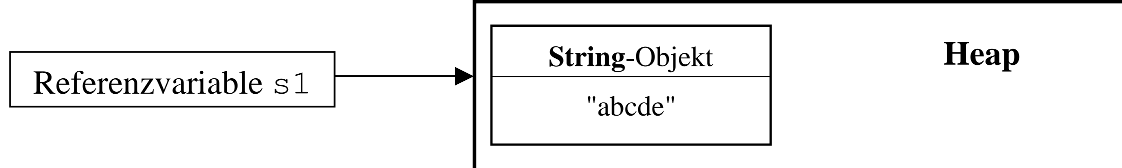
einen Ausdruck darstellt, der als Wert einen Verweis auf ein **String**-Objekt auf dem Heap liefert.

Weil in der Initialisierung

```
String s1 = "abcde";
```

kein **new**-Operator auftaucht, spricht man auch vom **impliziten** Erzeugen eines **String**-Objektes.

Wir haben im Hauptspeicher folgende Situation:



5.2.1.2 Der interne String-Pool

Erfolgt eine String-Initialisierung über einen *konstanten* Ausdruck, kommt der so genannte **interne String-Pool** ins Spiel: Existiert dort bereits ein Objekt mit demselben Inhalt, wird dessen Adresse verwendet. Anderenfalls wird ein neues Objekt im String-Pool angelegt.
Damit ist für Variablen mit Referenzen auf **String**-Pool-Objekte garantiert: Zwei Variablen stehen genau dann für die selbe Zeichenfolge, wenn sie den selben Referenzwert haben.

Kommt bei der Initialisierung ein *variabler* Ausdruck zum Einsatz, wird auf jeden Fall ein neues Objekt erzeugt, z.B.:

```
String de = "de";
String s3 = "abc"+de;
```

Dies geschieht auch bei expliziter Verwendung des **new**-Operators, z.B.:

```
String s4 = new String("abcde");
```

Beim Vergleich von **String**-Variablen per Identitätsoperator haben obige Ausführungen wichtige Konsequenzen, wie das folgende Programm zeigt:

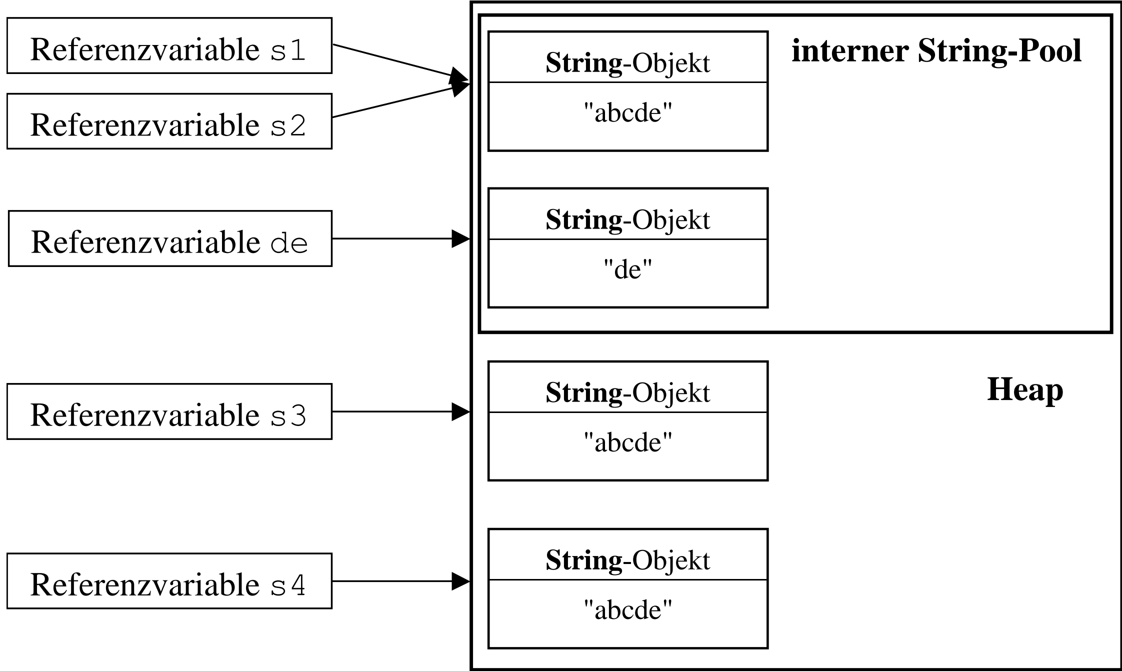
Quellcode	Ausgabe

<pre>class StringTest {
    public static void main(String[] args) {
        String s1 = "abcde";
        String s2 = "abc"+"de";
        String de = "de";
        String s3 = "abc"+de;
        String s4 = new String("abcde");
        System.out.print("(s1 == s2) = "+(s1==s2)+"\n"+
                         "(s1 == s3) = "+(s1==s3)+"\n"+
                         "(s1 == s4) = "+(s1==s4)+"\n"+
                         "(s3 == s4) = "+(s3==s4));
    }
}</pre>
	<pre>(s1 == s2) = true
	(s1 == s3) = false
	(s1 == s4) = false
	(s3 == s4) = false</pre>

Das merkwürdige[1] Verhalten des Programms hat folgende Ursachen:

- Wendet man den Identitätsoperator auf zwei **String**-Referenzvariablen an, werden die in den Variablen gespeicherten *Adressen* verglichen, keinesfalls die Inhalte der referenzierten **String**-Objekte.
- Erfolgt eine **String**-Initialisierung über einen *konstanten* Ausdruck *ohne* **new**-Operator, so entsteht nur dann ein neues **String**-Objekt, wenn noch kein inhaltsgleiches existiert. Andernfalls wird die Adresse der vorhandenen Instanz verwendet.
  Bei einer **String**-Initialisierung über einen variablen Ausdruck oder per **new**-Operator entsteht auf jeden Fall ein neues Objekt.

Im Beispielprogramm werden vier **String**-Objekte mit folgenden Referenzen erzeugt:



5.2.1.3 *String als Read Only - Klasse*

Nachdem ein **String**-Objekt auf dem Heap erzeugt worden ist, kann es nicht mehr geändert werden. Eventuell werden Sie diese Behauptung in Zweifel ziehen und ein Gegenbeispiel der folgenden Art vorbringen:

---

[1] „Merkwürdig" bedeutet hier, dass eine Aufnahme in das Langzeitgedächtnis empfohlen wird.

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class StringTest {
   public static void main(String[] args) {
      String testr = "abc";
      System.out.println("testr = " + testr);
      testr = testr + "def";
      System.out.println("testr = " + testr);
   }
}</pre></td><td><pre>testr = abc
testr = abcdef</pre></td></tr>
</table>

Was sich hier geändert hat, ist der Inhalt der Referenzvariablen `testr`: Die Referenz auf das **String**-Objekt mit dem Text „abc" wurde ersetzt durch die Referenz auf ein neues **String**-Objekt mit dem Text „abcdef". Das alte Objekt ist noch vorhanden, aber nicht mehr referenziert. Sobald das Laufzeitsystem Speicher benötigt, wird das alte Objekt vom Garbage Collector eliminiert.

5.2.1.4 Methoden für String-Objekte

Von den ca. 50 Methoden der Klasse der **String** werden in diesem Abschnitt nur die wichtigsten angesprochen. Für spezielle Anwendungen lohnt sich also ein Blick in die Dokumentation zum Java-SDK.

5.2.1.4.1 Verketten von Strings

Zum Verketten von Strings kann in Java der + - Operator verwendet werden, wobei beliebige Datentypen bei Bedarf automatisch in Strings konvertiert werden. In folgendem Beispiel wird mit Klammern dafür gesorgt, dass Java die + - Operatoren jeweils sinnvoll interpretiert (Verketten von Strings bzw. Addieren von Zahlen):

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class StringTest {
      public static void main(String[] args) {
            System.out.println("4 + 3 = " + (4 + 3));
      }
}</pre></td><td><pre>4 + 3 = 7</pre></td></tr>
</table>

Es ist übrigens eine Besonderheit, dass **String**-Objekte mit dem + - Operator verarbeitet werden können. Bei anderen Java-Klassen ist das aus C++ bekannte Überladen von Operatoren *nicht* erlaubt.

5.2.1.4.2 Vergleichen von Strings

Für den Test auf identischen **Inhalt** kann die **String**-Methode **equals(String** *vergl***)** verwendet werden, um den oben erläuterten Tücken beim Vergleich von **String**-Referenzvariablen per Identitätsoperator aus dem Weg zu gehen.
In folgendem Programm werden zwei **String**-Objekte zunächst nach ihren Speicheradressen verglichen, dann nach dem Inhalt:

Quellcode	Ausgabe

`class StringTest {`
`    public static void main(String[] args) {`
`        String s1 = "abc";`
`        String s2 = new String("abc");`
`        System.out.println(s1==s2);`
`        **System.out.println(s1.equals(s2));**`
`    }`
`}`
	false
	true

Leider ist die **equals()**-Methode nicht sehr performant, so dass bei einer aufwändigeren Vergleichstätigkeit der oben beschriebene **String**-Pool genutzt werden sollte. Mit Hilfe der **String**-Methode **intern()** können beliebige Strings „internalisiert" werden.
Wenn ein **String**-Objekt die **intern()**-Botschaft erhält, hängt das weitere Geschehen davon ab, ob im internen **String**-Pool bereits ein inhaltsgleiches **String**-Objekt existiert:

- Falls *ja*, wird dessen Adresse als Rückgabewert geliefert.
- Falls *nein*, wird im internen Pool ein inhaltsgleiches **String**-Objekt angelegt und dessen Adresse als Rückgabewert geliefert.

In folgendem Beispielprogramm wird dafür gesorgt, dass die Elemente des **String**-Vektors `s` auf Instanzen im internen **String**-Pool zeigen, so dass Inhaltsvergleiche über die Referenzwerte möglich sind. Beim Auswerten des Ausdrucks:

```
"abc" + (char)(zzg.nextInt(95)+32) + (char)(zzg.nextInt(95)+32)
```

entsteht auf dem Heap ein **String**-Objekt, dessen Inhalt mit „abc" beginnt und mit zwei zufällig gewählten Zeichen endet. Die von diesem Objekt ausgeführte **intern()**-Methode liefert die Adresse eines inhaltsgleichen (nötigenfalls neu erzeugten) Objekts aus dem internen **String**-Pool, so dass mit der Anweisung

```
for (int i = 0; i < anz; i++)
  s[i] = ("abc"+(char) (zzg.nextInt(95)+32)
               +(char) (zzg.nextInt(95)+32)).intern();
```

alle `s`–Elemente mit „internen Referenzen" versorgt werden.
Nach diesem Vorbemerkungen sollte der Quellcode problemlos lesbar sein:

Quellcode:

```
class StrIntern {
   public static void main(String[] args) {
      final int anz = 100000;
      String s[] = new String[anz];
      java.util.Random zzg =
         new java.util.Random(System.currentTimeMillis());

      for (int i = 0; i < anz; i++)
         s[i] = ("abc"+(char) (zzg.nextInt(95)+32)
                  +(char) (zzg.nextInt(95)+32)).intern();

      long vorher = System.currentTimeMillis();
      for (int i = 0; i < anz; i++) {
         if (s[i].equals("abcaa"))
            System.out.println("Treffer, i = "+i);
      }
      long diff = System.currentTimeMillis() - vorher;
      System.out.println("Zeit fuer equals - Vergl.: "+diff+"\n");

      vorher = System.currentTimeMillis();
      for (int i = 0; i < anz; i++)
         if (s[i] == "abcaa")
            System.out.println("Treffer, i = "+i);
      diff = System.currentTimeMillis() - vorher;
      System.out.println("Zeit fuer == - Vergl.: " + diff);
   }
}
```

Ausgabe:

```
Treffer, i = 668
Treffer, i = 7861
Treffer, i = 30122
Treffer, i = 57858
Treffer, i = 77039
Treffer, i = 81682
Zeit fuer equals - Vergl.: 100

Treffer, i = 668
Treffer, i = 7861
Treffer, i = 30122
Treffer, i = 57858
Treffer, i = 77039
Treffer, i = 81682
Zeit fuer == - Vergl.: 10
```

Es werden jeweils 100000 Vergleiche per Referenz bzw. per **equals()**-Aufruf durchgeführt, wobei sich enorme Laufzeitunterschiede ergeben (in Millisekunden gemessen auf einem PC mit 400-MHz-CPU).

Zum Testen auf **lexikographische Priorität** (z.B. beim Sortieren) kann die **String**-Methode **compareTo()** dienen:

Quellcode	Ausgabe

`class StringTest {`
`    public static void main(String[] args) {`
`        String a = "Müller, Anja", b = "Müller, Kurt",`
`                c = "Müller, Anja";`
`        System.out.println("< : " + a.compareTo(b));`
`        System.out.println("= : " + a.compareTo(c));`
`        System.out.println("> : " + b.compareTo(a));`
`    }`
`}`
	`< : -10`
	`= : 0`
	`> : 10`

compareTo() liefert folgende Ergebnisse zurück:

		compareTo()-Ergebnis
String, dessen **compareTo()**-Methode aufgerufen wird, ist im Vergleich zum Argument:	kleiner	negative Zahl
	gleich	0
	größer	positive Zahl

5.2.1.4.3 Länge einer Zeichenkette

Während bei Array-Objekten die Anzahl der Elemente in der *Instanzvariablen* **length** zu finden ist (vgl. Abschnitt 5.1), wird die aktuelle Länge einer Zeichenkette über die *Instanzmethode* **length()** ermittelt:

Quellcode	Ausgabe

`class StringTest {`
`    public static void main(String[] args) {`
`        char[] vek = {'a', 'b', 'c'};`
`        String str = "abc";`
`        System.out.println(vek.length);`
`        System.out.println(str.length());`
`    }`
`}`
	`3`
	`3`

5.2.1.4.4 Zeichen extrahieren, suchen oder ersetzen

Im Abschnitt über bedingte Anweisungen haben wir erstmals mit dem Array von **String**-Elementen gearbeitet, über den Java-Programme Zugang zu den beim Start übergebenen Kommandozeilen-Parametern haben. Dem Programm zur Persönlichkeitsdiagnose sollte beim Start als erster Kommandozeilenparameter ein Buchstabe (für die Lieblingsfarbe) übergeben werden. Dieser landet im Array-Element `args[0]` als erstes Zeichen und kann mit der String-Methode **charAt(int** *pos***)** dort ausgelesen werden:

Quellcode	Ausgabe

<pre>class StringTest {
   public static void main(String[] args) {
      if (args.length > 0) {
         System.out.println(args[0].substring(0,args[0].length()));
         System.out.println(args[0].indexOf("t"));
         System.out.println(args[0].indexOf("x"));
         System.out.println(args[0].startsWith("r"));
         System.out.println(args[0].charAt(0));
      }
   }
}</pre>
	<pre>rot
	2
	-1
	true
	r</pre>

Mit der Methode **substring(int** *start***, int** *ende***)** lassen sich alle Zeichen zwischen *start* (inklusive) und *ende* (exklusive) extrahieren. Wie man am Beispiel sieht, harmoniert die „exklusive" Ende-Regel (aufgrund der nullbasierten Indexierung) gut mit dem Verhalten der **length()**-Methode.

Mit der Methode **indexOf(String** *gesucht***)** kann man einen **String** nach einer anderen Zeichenkette oder einem **char**-Ausdruck durchsuchen. Als Rückgabewert erhält man ...

- nach erfolgreicher Suche: die Startposition der ersten Trefferstelle
- nach vergeblicher Suche: -1

Mit der Methode **startsWith**(**String** *start*) lässt sich feststellen, ob ein **String** mit einer bestimmten Zeichenfolge beginnt.

Weil ein **String** kein Array ist, kann auf die einzelnen Zeichen *nicht* per Offset-Operator ([]) zugegriffen werden. Mit der **String**-Methode **charAt()** steht aber ein gleichwertiger Ersatz zur Verfügung, wobei die Nummerierung der Zeichen wiederum bei 0 beginnt.

Wenn auf jeden Fall mit dem Offset-Operator gearbeitet werden soll, kann aus einem **String** über die Methode **toCharArray**() ein neuer **char**-Array mit identischem Inhalt erzeugt werden, z.B.:

Quellcode	Ausgabe

<pre>class Prog {
   public static void main(String[] args) {
      String s = "abc";
      char[] c = s.toCharArray();
      for (int i = 0; i < c.length; i++)
         System.out.println(c[i]);
   }
}}</pre>
	<pre>a
	b
	c</pre>

Mit der Methode **replace**(**char** *oldChar*, **char** *newChar*) erhält man einen neuen **String**, der aus dem angesprochenen Original durch Ersetzen eines alten Zeichens durch ein neues Zeichen hervorgeht, z.B.:

```
String s2 = s1.replace('C','c');
```

5.2.1.4.5 Groß-/Kleinschreibung normieren

Mit den Methoden **toUpperCase()** bzw. **toLowerCase()** kann man alle Zeichen eines Strings auf Groß- bzw. Kleinschreibung umstellen, was vor Vergleichen oft sinnvoll ist, z.B.:

Quellcode	Ausgabe

`class StringTest {`
`  public static void main(String[] args) {`
`   String a = "Otto", b = "otto";`
`   System.out.println(a.toUpperCase().equals(b.toUpperCase()));`
`  }`
`}`
	`true`

Im Aufruf der **equals()**-Methode stoßen wir auf eine stattliche Anzahl von Punktoperatoren, so dass vielleicht eine kurze Erklärung angemessen ist:

- Der Methodenaufruf `a.toUppercase()` erzeugt ein neues **String**-Objekt und liefert eine zugehörige Referenz.
- Diese Referenz ermöglicht es, dem neuen Objekt Botschaften zu übermitteln, was unmittelbar zum Aufruf der Methode **equals()** genutzt wird.

5.2.2 Die Klasse StringBuffer für veränderliche Zeichenketten

Für häufig zu ändernde Zeichenketten sollte man statt der Klasse **String** unbedingt die Klasse **StringBuffer** verwenden, weil hier beim Ändern einer Zeichenkette die (zeitaufwändige) Erzeugung eines neuen Objektes entfällt.

Ein **StringBuffer** kann nicht implizit erzeugt werden, jedoch stehen bequeme Konstruktoren zur Verfügung, z.B.:

- **StringBuffer**(**String** *str*)
  Beispiel: `StringBuffer sb = new StringBuffer("abc");`
- **StringBuffer**()
  Beispiel: `StringBuffer sb = new StringBuffer();`

In folgendem Programm wird eine Zeichenkette 20000-mal verlängert, zunächst mit Hilfe der **String**-Klasse, dann mit Hilfe der **StringBuffer**-Klasse:

```
class StringTest {
   public static void main(String[] args) {
      String s = "*";
      long vorher = System.currentTimeMillis();
      for (int i = 0; i < 20000; i++)
         s = s + "*";
      long diff = System.currentTimeMillis() - vorher;
      System.out.println("Zeit fuer String-Manipulation: " + diff);

      s = "*";
      vorher = System.currentTimeMillis();
      StringBuffer t = new StringBuffer(s);
      for (int i = 0; i < 20000; i++)
         t.append("*");
      s = t.toString();
      diff = System.currentTimeMillis() - vorher;
      System.out.println("Zeit fuer StringBuffer-Manipulation: " + diff);
   }
}
```

Die (in Millisekunden gemessenen) Laufzeiten unterscheiden sich erheblich:[1]

[1] Gemessen mit der Java Runtime Environment 1.4.1 auf einem Pentium 4 – System mit 2,26 GHz.

```
Zeit fuer String-Manipulation:       3828
Zeit fuer StringBuffer-Manipulation: 0
```

Ein **StringBuffer**-Objekt beherrscht u.a. die folgenden Methoden:

StringBuffer-Methode	Erläuterung
length()	liefert die Anzahl der Zeichen
append()	Der **StringBuffer** wird um die Stringrepräsentation des Argumentes verlängert, z.B.: `sb.append("*");` Es sind **append()**-Überladungen für zahlreiche Datentypen vorhanden.
insert()	Die Stringrepräsentation des Argumentes, das von nahezu beliebigem Typ sein kann, wird an einer bestimmten Stelle eingefügt, z.B.: `sb.insert(4, 3.14);`
delete()	Die Zeichen von einer Startposition (einschließlich) bis zu einer Endposition (ausschließlich) werden gelöscht, in folgendem Beispiel also gerade 2 Zeichen, falls der **StringBuffer** mindestens 3 Zeichen enthält: `sb.delete(1,3);`
replace()	Ein Bereich des **StringBuffers** wird durch den Argument-**String** ersetzt, z.B.: `sb.replace(1,3, "xy");`
toString()	Es wird ein **String**-Objekt mit dem Inhalt des **StringBuffers** erzeugt. Dies ist z.B. erforderlich, um zwei **StringBuffer**-Objekte mit Hilfe der **String**-Methode **equals()** vergleichen zu können: `sb1.toString().equals(sb2.toString())`

5.2.3 Übungsaufgaben zu Abschnitt 5.2

1) Erstellen Sie ein Programm zum Berechnen einer persönlichen Glückszahl (zwischen 1 und 100), indem Sie:

- Vor- und Nachnamen als Kommandozeilenparameter einlesen,
- den Anfangsbuchstaben des Vornamens sowie den letzten Buchstaben des Nachnamens ermitteln (beide in Großschreibung),
- die Nummern der beiden Buchstaben im Unicode-Zeichensatz bestimmen,
- die beiden Buchstaben-Nummern addieren und die Summe als Startwert für den Pseudozufallszahlengenerator verwenden.

Beenden Sie Ihr Programm mit einer Fehlermeldung, wenn weniger als 2 Kommandozeilenparameter übergeben wurden.

Tipp: Um ein Programm spontan zu beenden, kann die Methode **exit()** der Klasse **System** verwendet werden.

2) Die Klassen **String** und **StringBuffer** besitzen beide eine Methode namens **equals()**, doch bestehen gravierende Verhaltensunterschiede:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class EqualVergl {
  public static void main(String[] args) {
    StringBuffer sb1 = new StringBuffer("abc");
    StringBuffer sb2 = new StringBuffer("abc");
    System.out.println("sb1 = sb2 = "+sb1);
    System.out.println("StringBuffer-Vergl.: "+
      sb1.equals(sb2));

    String s1 = sb1.toString();
    String s2 = sb1.toString();
    System.out.println("\ns1 = s2 = "+s1);
    System.out.println("String-Vergl.:   "+
      s1.equals(s2));
  }
}</pre></td><td><pre>sb1 = sb2 = abc
StringBuffer-Vergl.: false

s1 = s2 = abc
String-Vergl.:   true</pre></td></tr>
</table>

Ermitteln Sie mit Hilfe der SDK-Dokumentation die Ursache für das unterschiedliche Verhalten.

3) Erstellen Sie eine Klasse `StringUtil` mit einer statischen Methode `wrapln()`, die einen **String** auf die Konsole schreibt und dabei einen korrekten Zeilenumbruch vornimmt. Anwender Ihrer Methode sollen die gewünschte Zeilenbreite vorgeben können und auch die Trennzeichen festlegen dürfen, aber nicht müssen (Methoden überladen!). Am Anfang einer neuen Zeile sollen außerdem keine Leerzeichen stehen.
In folgendem Programm wird die Verwendung der Methode demonstriert:

```
class StringUtilTest {
    public static void main(String[] args) {
        String s = "Dieser Satz passt nicht in eine Schmal-Zeile, "+
                   "die nur wenige Spalten umfasst.";
        StringUtil.wrapln(s, " ", 40);
        StringUtil.wrapln(s, 40);
        StringUtil.wrapln(s);
    }
}
```

Der zweite Methodenaufruf sollte folgende Ausgabe erzeugen:

```
Dieser Satz passt nicht in eine Schmal-
Zeile, die nur wenige Spalten umfasst.
```

Das Zerlegen eines Strings in einzelne *Tokens* können Sie einem Objekt der Klasse **StringTokenizer** aus dem Paket **java.util** überlassen. In folgendem Programm wird demonstriert, wie ein **StringTokenizer** arbeitet:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class TokenizerTest {
  public static void main(String[] args) {
    String s = "Dies ist der Satz, der zerlegt werden soll.";
    java.util.StringTokenizer stob =
      new java.util.StringTokenizer(s, " ", true);
    while (stob.hasMoreTokens())
      System.out.println(stob.nextToken());
}</pre></td><td><pre>Dies

ist

der

Satz,

der

zerlegt

werden

soll.</pre></td></tr>
</table>

Was als Token interpretiert werden soll, legt man über den letzten Parameter im Konstruktor der Klasse **java.util.StringTokenizer** fest (**boolean returnDelims**):

- **false**: Ein Token ist die maximale Sequenz von Zeichen, die keine Trennzeichen sind. Die Trennzeichen dienen nur zum Separieren der Tokens und werden ansonsten *nicht* zurück geliefert.
- **true**: Auch die Trennzeichen werden als Tokens behandelt.

5.3 Verpackungs-Klassen für primitive Datentypen

In Java existiert zu jedem primitiven Datentyp eine Wrapper-Klasse, in deren Objekte jeweils ein Wert des primitiven Typs verpackt werden kann (*to wrap* heißt *einpacken*):

Primitiver Datentyp	Wrapper-Klasse
byte	Byte
short	Short
int	Integer
long	Long
double	Double
float	Float
boolean	Boolean
char	Character
void	Void

Diese Verpackung ist z.B. dann sinnvoll, wenn eine Methode genutzt werden soll, die nur für Objekte verfügbar ist. Außerdem stellen die Wrapper-Klassen nützliche Konvertierungsmethoden und Konstanten bereit (jeweils klassenbezogen).

In der Regel verfügen die Wrapper-Klassen über zwei Konstruktoren mit jeweils einem Parameter, der vom primitiven „Basistyp" oder vom Typ **String** sein darf, z.B. bei **Integer**:

- **Integer**(**int** *value*)
  Beispiel: `Integer iw = new Integer(4711);`
- **Integer**(**String** *str*)
  Beispiel: `Integer iw = new Integer(args[0]);`

5.3.1 Einweg-Verpackungen

Das folgende Beispielprogramm berechnet die Summe der numerisch interpretierbaren Kommandozeilenparameter, hebt aber auf jeden Fall alle Parameter in einem **Vector**-Objekt auf:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
class Wrapper {
   public static void main(String[] args) {
      Double zahl;
      java.util.Vector v =
         new java.util.Vector(args.length);
      double summe = 0.0;
      int fehler = 0;

      for(int i=0; i < args.length; i++) {
         try {
            zahl = Double.valueOf(args[i]);
            v.addElement(zahl);
            summe += zahl.doubleValue();
         } catch(Exception e) {
            v.addElement(args[i]);
            fehler++;
         }
      }

      System.out.println("Ihre Eingaben:");
      for(int i=0; i < args.length; i++)
         System.out.println(" " + v.elementAt(i));
      System.out.println("\nSumme: " + summe +
                         "\nFehler: "+fehler);
   }
}
```

</td><td>

```
Ihre Eingaben:
 2.5
 3.99
 teggsd
 5.0
 noz
 13.78
 fuppes

Summe: 25.27
Fehler: 3
```

</td></tr>
</table>

Mit der Klasse **Vector** aus dem Paket **java.util** bietet Java eine sehr flexible Datenstruktur:

- Ein **Vector** kann beliebige Objekte als Elemente aufnehmen (aber eben keine Werte primitiver Typen).
- Die Größe des Vektors wird automatisch an den Bedarf angepasst.

Das Beispielprogramm versucht im Rahmen einer **try-catch** - Konstruktion, die später im Abschnitt über Ausnahmebehandlung ausführlich besprochen wird, für jeden Kommandozeilenparameter eine numerische Interpretation mit der **Double**-Konvertierungsmethode **valueOf()** (s.u.).
Jeder Kommandozeilenparameter wird mit der Methode **addElement(Object** *obj***)** als neues Element in den **Vector** aufgenommen:

- Bei gelungener Konvertierung als **Double**-Object,
- ansonsten als **String**-Objekt.

Weil die **Vector**-Methode **addElement(Object** *obj***)** nur Referenzvariablen als Argument akzeptiert, könnte ein **double**-Wert *nicht* in den Vektor aufgenommen werden.

Beachten Sie bitte, dass die Anweisung

```
zahl = Double.valueOf(args[i]);
```

bei jeder Ausführung ein neues **Double**-Objekt produziert, was in obigem Beispielprogramm durchaus erwünscht ist. Für den Einsatz in numerischen Ausdrücken sind die Wrapper-Klassen jedoch wenig geeignet, weil sie im Einweg- bzw. read-only – Sinn funktionieren: Der beim Erzeugen für ein Wrapper-Objekt festgelegte Wert kann nicht mehr geändert werden. Für diesen Zweck sind schlicht keine Methoden vorhanden. Sie sollen daher in einer Übungsaufgabe eine eigene Hüllenklasse ohne Einweg-Technik entwerfen, die numerischen Werten ebenfalls die Aufnahme in Container-Objekte (z.B. aus der Klasse **Vector**) eröffnet.

5.3.2 Konvertierungs-Methoden

Numerische Wrapper-Klassen stellen Methoden zum Konvertieren von Strings in Zahlen zur Verfügung.
Mit der Klassenmethode **valueOf(String** *str***)** erhält man ein neues Wrapper-Objekt mit der numerischen Interpretation des Strings als Wert, falls die Konvertierung gelingt. In folgendem Beispiel wird aus dem ersten Kommandozeilen-Parameter ein **Double**-Objekt erzeugt:

```
Double zahl = Double.valueOf(args[0]);
```

Statt mit **valueOf(String** *str***)** ein **Double**-Objekt zu erstellen, kann man auch mit **parseDouble(String** *str***)** einen Wert des primitiven Datentyps **double** gewinnen, z.B.:

```
double zahl = Double.parseDouble(args[0]);
```

Auch die Klasse **String** bietet eine Methode **valueOf()**, in diesem Fall aber zur Wandlung primitiver oder sonstiger Datentypen in Strings, z.B.:

```
String s = String.valueOf(summe);
```

In numerischen Ausdrücken sind Wrapper-Objekte im Vergleich zu Variablen des eingepackten primitiven Typs nicht sehr praktisch:

- Man kann ihren Wert nach dem Erzeugen des Objektes nicht mehr ändern.
- Zum Auslesen des Wertes ist ein Methodenaufruf erforderlich, z.B.:

```
summe += zahl.doubleValue();
```

5.3.3 Konstanten mit Grenzwerten

In den numerischen Wrapper-Klassen sind finalisierte und statische Instanzvariablen für diverse Grenzwerte definiert, z.B. in der Klasse **Double**:

Konstante	Inhalt
MAX_VALUE	Größter positiver (endlicher) Wert des Datentyps **double**
MIN_VALUE	Kleinster positiver Wert des Datentyps **double**
NaN	Not-a-Number - Ersatzwert für den Datentyp **double**
POSITIVE_INFINITY	Positiv-Unendlich - Ersatzwert für den Datentyp **double**
NEGATIVE_INFINITY	Negativ-Unendlich - Ersatzwert für den Datentyp **double**

Beispiel:

Quellcode:

```
class WrapperDemo {
   public static void main(String[] args) {
      System.out.println("Max. double-Zahl:\n"+
                          Double.MAX_VALUE);
   }
}
```

Ausgabe:

```
Max. double-Zahl:
1.7976931348623157E308
```

5.3.4 Character-Methoden zur Zeichen-Klassifikation

Die Wrapper-Klasse **Character** bietet einige statische Methoden zur Klassifikation von Unicode-Zeichen, die bei der Verarbeitung von Textdaten sehr nützlich sein können:

Methode	Erläuterung
isDigit(**char** *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen eine Ziffer ist, sonst **false**.
isLetter(**char** *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen ein Buchstabe ist, sonst **false**.
isLetterOrDigit(**char** *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen ein Buchstabe oder eine Ziffer ist, sonst **false**.
isWhitespace(**char** *ch*)	Die Methode liefert den Wert **true** zurück, wenn ein Trennzeichen übergeben wurde, sonst **false**. Zu den Trennzeichen gehören u.a.: • Leerzeichen (\u0020) • Tabulatorzeichen (\u0009) • Wagenrücklauf (\u000D) • Zeilenvorschub (\u000A)
isLowerCase(**char** *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen ein Kleinbuchstabe ist, sonst **false**.
isUpperCase(**char** *ch*)	Die Methode liefert den Wert **true** zurück, wenn das übergebene Zeichen ein Großbuchstabe ist, sonst **false**.

5.3.5 Übungsaufgaben zu Abschnitt 5.3

1) Ermitteln Sie den kleinsten möglichen Wert des Datentyps **byte**.

2) Ermitteln Sie die maximale natürliche Zahl *k*, für die in Java unter Verwendung des Funktionswertedatentyps **double** die Fakultät *k*! bestimmt werden kann.

3) Entwerfen Sie eine Hüllenklasse, welche die Aufnahme von **int**-Werten in Container-Klassen wie **Vector** ermöglicht, ohne (wie **Integer**, vgl. Abschnitt 5.3.1) die Werte der Objekte nach der Erzeugung zu fixieren.

4) Erweitern Sie die im Abschnitt 5.2.3 vorgeschlagene Klasse `StringUtil` um eine statische Methode namens `countChars()`, die für einen **String**–Parameter die Häufigkeiten der enthaltenen Zeichen protokolliert. In folgendem Programm wird die Verwendung der Methode demonstriert:

Quellcode

```
class StringUtilTest {
   public static void main(String[] args) {
      StringUtil.countChars("Am Anfang eines Astes "+
         "befindet sich meist ein Baum");
   }
}
```

Ausgabe

```
A: 3
B: 1
a: 2
b: 1
c: 1
d: 1
e: 7
f: 2
g: 1
h: 1
i: 5
m: 3
n: 5
s: 5
t: 3
u: 1
```

6 Pakete

In Java gehört jede Klasse zu einem *Paket* (engl.: *package*) und muss i. A. über ein durch Punkt getrenntes Paar aus Paketnamen und Klassennamen angesprochen werden, wie Sie es schon bei etlichen Beispielen kennen gelernt haben. Obwohl man etliche Klassen auch ohne vorangestellten Paketnamen ansprechen kann, gehören diese doch stets zu einem Paket:

- Alle Klassen in einem Dateiverzeichnis, die keinem Paket zugeordnet wurden (siehe unten), bilden ein **anonymes Paket**. Diese Situation war bei all unserem bisherigen Anwendungen regelmäßig gegeben und aufgrund der geringen Komplexität auch angemessen.
- Das Paket **java.lang** mit besonders fundamentalen Klassen (z.B. **String**, **System**, **Math**) wird bei jeder Anwendung automatisch importiert (siehe unten), so dass seine Klassen direkt angesprochen werden können.

Vielfach werden Java-Pakete auch als *Klassenbibliotheken* bezeichnet. Neben Klassen können sie auch *Schnittstellen* (*Interfaces*) enthalten, mit denen wir uns erst später beschäftigen werden.
Pakete erfüllen in Java viele wichtige Aufgaben, die vor allem in größeren Projekten relevant sind:

- **Große Projekte strukturieren**
  Wenn sehr viele Klassen vorhanden sind, kann man mit Paketen Ordnung schaffen. In der Regel befinden sich alle **class**-Dateien eines Paketes in einem Dateiverzeichnis, dessen Name mit dem Paketnamen übereinstimmt.
  Es ist auch ein *hierarchischer* Aufbau über *Unterpakete* möglich, wobei die Paketstruktur einem Dateiverzeichnisbaum entspricht. Im Namen eines konkreten (Unter-)Paketes folgen dann die Verzeichnisnamen aus dem zugehörigen Pfad durch Punkte getrennt aufeinander, z.B.:

  java.util.zip

  Vor allem bei der Weitergabe von Programmen ist es nützlich, eine komplette Paketstruktur in eine **Java-Archivdatei** (mit Extension **.jar**) zu verpacken (siehe unten).
  Sowohl der hierarchische Paketaufbau wie auch der Einsatz einer Java-Archivdatei werden gleich am Beispiel der Java-API-Klassenbibliothek veranschaulicht.
- **Namenskonflikte vermeiden**
  Jedes Paket bildet einen eigenen Namensraum. Identische Bezeichner stellen also kein Problem dar, solange sie sich in verschiedenen Paketen befinden.
- **Zugriffskontrolle steuern**
  Per Voreinstellung haben die Klassen eines Paketes wechselseitig uneingeschränkte Zugriffsrechte, während Klassen aus anderen Paketen nur auf **public**-Elemente zugreifen dürfen (s.u.).

6.1 Das API der Java 2 Standard Edition

Zur Java-Plattform gehören zahlreiche Pakete, die Klassen für wichtige Aufgaben der Programmentwicklung (z.B. Stringverarbeitung, Netzverbindungen, Datenbankzugriff) enthalten. Die Zusammenfassung dieser Pakete wird oft als **Java-API** (**A**pplication **P**rogramming **I**nterface) bezeichnet. Allerdings kann man nicht von *dem* Java-API sprechen, denn neben der kostenlos verfügbaren **Java 2 Standard Edition (J2SE)**, auf die wir uns beschränken, bietet die Firma SUN noch weitere, teilweise kostenpflichtige, Java-APIs an, z.B.:

- Java 2 Enterprise Edition (J2EE)
- Java 2 Micro Edition (J2ME)

Nähere Informationen finden Sie z.B. auf der Webseite http://java.sun.com/products/.

In der SDK-Dokumentation zur Java 2 Standard Edition (J2SE) sind deren Pakete umfassend dokumentiert:

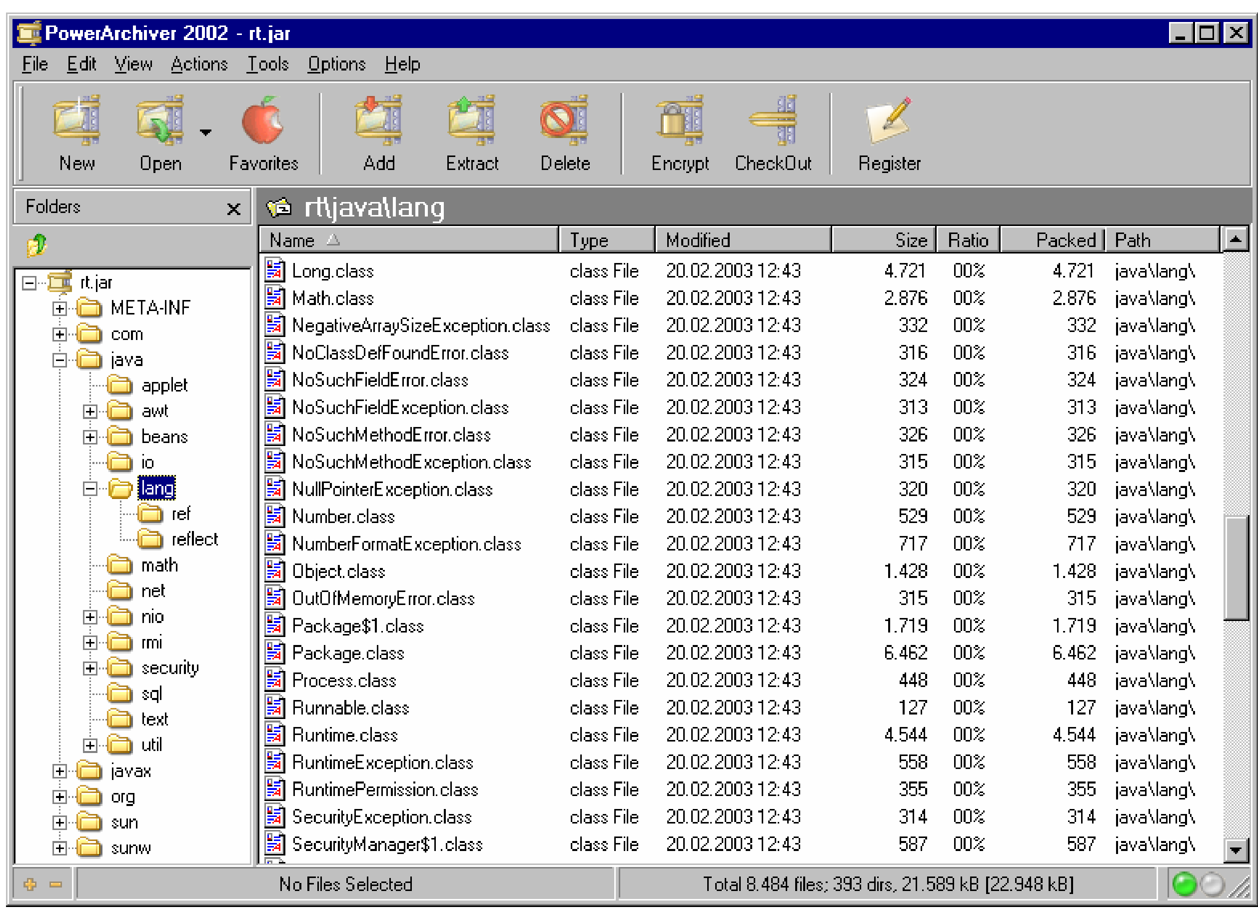
- Klicken Sie nach dem Start auf den Link **Java 2 Platform API Specification**.
  Im rechten Rahmen der Seite erscheint eine Liste mit allen Paketen im Standard-API.
- Im linken oberen Rahmen kann ein Paket (oder die Option **All Classes**) gewählt werden, und im linken unteren Rahmen erscheinen alle Klassen des ausgewählten Paketes.
  Neben den in normalem Schriftschnitt notierten Klassen erscheinen in kursiver Schrift die Schnittstellen (Interfaces), mit denen wir uns später beschäftigen werden.
  Nach dem Anklicken einer Klasse wird diese im Hauptrahmen ausführlich erläutert (z.B. mit einer Beschreibung der öffentlichen Variablen und Methoden).

Sie haben vermutlich schon mehrfach von diesem Informationsangebot Gebrauch gemacht.

Die zum Java-API gehörigen Bytecode-Dateien sind auf mehrere Java-Archivdateien (*.jar) verteilt, die sich bei der Version 1.4 der J2SE im Verzeichnis

...\jre\lib

befinden. Den größten Brocken bildet die Datei **rt.jar**. Hier befindet sich z.B. das Paket **java**, dessen Unterpaket **lang** die Bytecode-Datei zur Klasse **Math** enthält, deren Klassenmethoden wir schon mehrfach benutzt haben:[1]



Es folgen kurze Beschreibungen der wichtigsten Pakete im API der Java 2 Standard Edition (Version 1.4):

[1] Zur Anzeige der Datei **rt.jar** wird hier das Programm **PowerArchiver** verwendet, das zur Verwaltung von ZIP-Archiven dient.

- **java.applet**
  Das Paket enthält Klassen zum Programmieren von Applets.
- **java.awt**
  Das AWT-Paket (**A**bstract **W**indowing **T**oolkit) enthält Klassen zum Programmieren von graphischen Benutzerschnittstellen.
- **java.beans**
  Das Paket enthält Klassen zum Programmieren von Java-Komponenten (**beans** genannt), welche in der Java-Welt eine ähnliche Rolle spielen sollen wie die OCX- bzw. ActiveX-Komponenten in der Windows-Welt.
- **java.io**
  Das Paket enthält Ein- und Ausgabeklassen.
- **java.lang**
  Das Paket enthält fundamentale Klassen, die in jedem Programm ohne **import**-Anweisung (siehe unten) direkt angesprochen werden können.
- **java.math**
  Das Paket enthält die Klassen **BigDecimal** und **BigInteger** für Berechnungen mit beliebiger Genauigkeit. Das Paket **java.math** darf nicht verwechselt werden mit der *Klasse* **Math** im Paket **java.lang**.
- **java.net**, **javax.net**
  Die Pakete enthalten Klassen für Netzwerk-Applikationen.
- **java.nio**
  Dieses Paket (New I/O) bringt Verbesserungen bei Netzwerk- und Dateizugriffen, bei der Buffer-Verwaltung, sowie bei der Unterstützung von Zeichensätzen und regulären Ausdrücken.
- **java.rmi**, **javax.rmi**
  Mit Hilfe der der RMI-Pakete (**R**emote **M**ethod **I**nvocation) können Objekte genutzt werden, die auf fremden Rechnern existieren.
- **java.security**, **javax.security**
  Die Klassen dieser Pakete unterstützen moderne Sicherheits-Techniken (z.B. Zertifikate, Signaturen, Authentifizierung via Kerberos).
- **java.sql**, **javax.sql**
  Die Klassen dieses Paketes unterstützen den Datenbankzugriff via SQL.
- **java.text**
  Hier geht es um die Formatierung von Textausgaben und die Internationalisierung von Programmen. Wir haben mit **DecimalFormat** bereits eine Klasse aus diesem Paket kennen gelernt.
- **java.util**
  Dieses Paket enthält neben Kollektions-Klassen (z.B. **Vector**) wichtige Hilfsmittel wie den Pseudozufallszahlengenerator **Random**, den wir schon mehrfach benutzt haben.
- **javax.accessibility**
  Dieses Paket unterstützt behindertengerechte Ein- und Ausgabe.
- **javax.crypto**
  Dieses Paket bietet Verschlüsselungs-Techniken.
- **javax.imageio**
  Dieses Paket behandelt Aufgaben beim Lesen und Schreiben von Bilddateien in verschiedenen Formaten.
- **javax.naming**
  Dieses Paket enthält Klassen für den Zugriff auf Directory-Dienste (z.B. LDAP).
- **javax.print**
  Dieses Paket unterstützt den Java Print Service.

- **javax.sound**
  Dieses Paket unterstützt die Verarbeitung von Audio-Daten.
- **javax.swing**
  Im **swing**-Paket sind GUI-Klassen enthalten, die sich im Unterschied zu den **awt**-Klassen kaum auf die GUI-Komponenten des jeweiligen Betriebssystems stützen, sondern eigene Steuerelemente realisieren, was eine höhere Flexibilität und Portabilität mit sich bringt. Wir werden uns noch ausführlich mit den AWT- und Swing-Klassen beschäftigen.
- **javax.transaction**
  Mit den Klassen dieses Paketes wird eine transaktionsorientierte Programmierung unterstützt. Für die im Rahmen einer Transaktion ausgeführten Anweisungen ist sichergestellt, dass sie entweder vollständig oder gar nicht ausgeführt werden, was z.B. bei Datenbankzugriffen zur Sicherung der Datenintegrität von großer Bedeutung ist.
- **javax.xml**, **org.xml**
  Dieses Paket enthält Klassen zur Verarbeitung von XML-Dokumenten.
- **org.ietf**
  Dieses Paket unterstützt Sicherheitsmechanismen wie Authentifikation.
- **org.omg**
  Dieses Paket enthält Klassen für die CORBA-Unterstützung (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture).
- **org.w3c**
  Dieses Paket unterstützt das Document Object Model (DOM).

Neben den Paketen im Kern-API bietet SUN (siehe http://java.sun.com/products/OV_stdExt.html) auch noch etliche **Optional Packages** (früher als *Standard Extensions* bezeichnet) an, die kostenlos verfügbar sind, z.B.:

- Java Media Framework (JMF)
- Java 3D
- JavaMail

6.2 Pakete erstellen

Vor der *Verwendung* von Paketen behandeln wir deren *Produktion*, weil dabei die technischen Details gut veranschaulicht werden können.

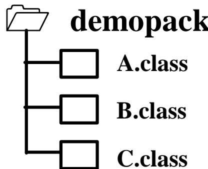
6.2.1 package-Anweisung und Paketordner

Wir erstellen zunächst ein einfaches Paket namens `demopack` mit den Klassen `A`, `B` und `C`. An den Anfang jeder einzubeziehenden Quellcodedatei setzt man die **package**-Anweisung mit dem Paketnamen, der üblicherweise *komplett klein* geschrieben wird, z.B.:

```
// Inhalt der Datei A.java
package demopack;

public class A {
    private static int anzahl;
    private int objnr;
    public A() {
        objnr = ++anzahl;
    }
    public void prinr() {
        System.out.println("Klasse A, Objekt Nr. " + objnr);
    }
}
```

Sind in einer Quellcodedatei *mehrere* Klassendefinitionen vorhanden, was in Java eher unüblich ist, so werden *alle* Klassen dem Paket zugeordnet. Vor einer **package**-Anweisung dürfen höchstens Kommentar- oder Leerzeilen stehen.
Die Klassen eines Paketes können von Klassen aus fremden Paketen nur dann verwendet werden, wenn sie als **public** definiert sind. Pro Quellcodedatei ist nur eine **public**-Klasse erlaubt. Zusätzlich müssen auch Methoden und Variablen explizit per **public**-Modifikator für fremde Pakete freigegeben werden. Steht z.B. kein **public**-Konstruktor zur Verfügung, können fremde Pakete eine Klasse zwar „sehen", aber keine Objekte dieses Typs erzeugen. Mit den Zugriffsrechten für Klassen, Methoden und Variablen werden wir uns in Abschnitt 6.4 ausführlich beschäftigen.

Die beim Übersetzen der zu einem Paket gehörigen Quellcodedateien entstehenden **class**-Dateien gehören in ein gemeinsames Dateiverzeichnis, dessen Name mit dem Paketnamen identisch ist. Für die Verwendung eines Paketes sind ausschließlich die **class**-Dateien erforderlich. In unserem Beispiel legen wir also ein Verzeichnis `demopack` an und sorgen dafür, dass die Bytecode-Dateien **A.class**, **B.class** und **C.class** dort landen:

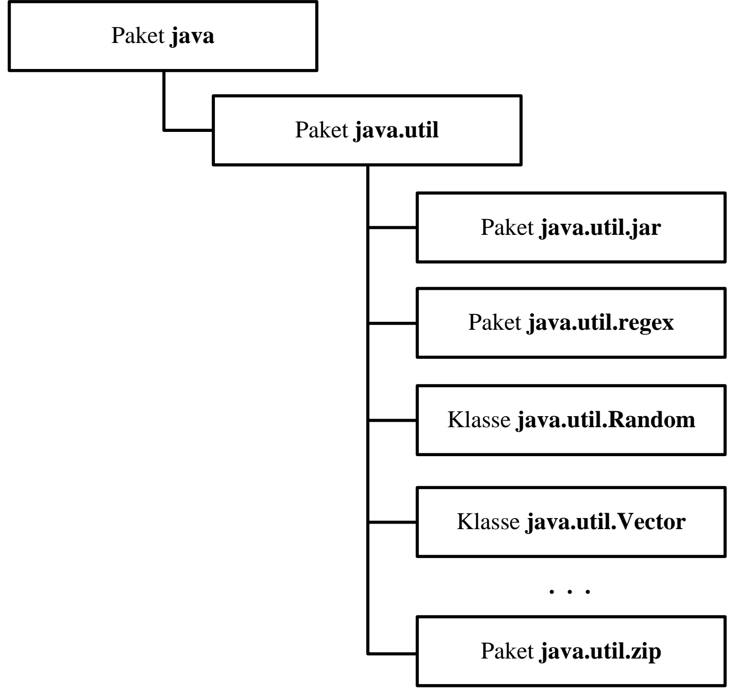


In Abhängigkeit von der Java-Entwicklungsumgebung geschieht das Erstellen des Paketordners und das Einsortieren der Bytecode-Dateien eventuell auch automatisch (siehe unten).
Für die Weitergabe von Programmen ist es oft sinnvoll, den Paketordner mit dem SDK-Werkzeug **jar.exe** in eine Java-Archivdatei zu verpacken (siehe Abschnitt 6.5).

Ohne **package**-Definition am Beginn einer Quellcode-Datei werden die resultierenden Klassen, die bekanntlich jeweils in einer eigenen Bytecode-Datei (mit Extension **.class**) landen, einem *anonymen* Paket zugeordnet. Alle Klassen in einem gemeinsamen Verzeichnis gehören dabei zum selben anonymen Paket.

6.2.2 Unterpakete

Ein Paket kann hierarchisch in **Unterpakete** eingeteilt werden, was bei den Java-API-Paketen in der Regel geschehen ist, z.B.:



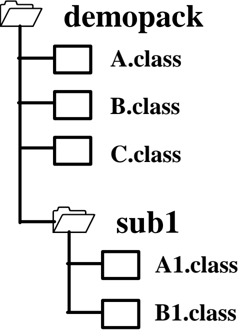
Auf jeder Stufe der Pakethierarchie können sowohl Klassen als auch Unterpakete enthalten sein. So enthält z.B. das Paket **java.util** u.a.

- die Klassen **Random**, **Vector**, ...
- die Unterpakete **jar**, **regex**, **zip**, ...

Soll eine Klasse einem Unterpaket zugeordnet werden, muss in der **package**-Anweisung am Anfang der Quellcodedatei der gesamte Paketpfad angegeben werden, wobei die Namensbestandteile jeweils durch einen Punkt getrennt werden.
Dies ist z.B. der Quellcode der Klasse `A1`, die in das Unterpaket `sub1` des `demopack`-Paketes eingeordnet wird:

```
package demopack.sub1;

public class A1 {
    private static int anzahl;
    private int objnr;
    public A1() {
        objnr = ++anzahl;
    }
    public void prinr() {
        System.out.println("Klasse A1, Objekt Nr. " + objnr);
    }
}
```

Die **class**-Dateien einer Pakethierarchie legt man zunächst in einem analog aufgebauten Dateiverzeichnisbaum ab, wobei jedem (Unter)paket ein (Unter)verzeichnis entspricht.
In unserem Beispiel müssen die **class**-Dateien also folgendermaßen angeordnet sein:

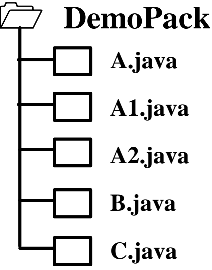


Mit dem SDK-Werkzeug **jar.exe** (siehe Abschnitt 6.5) lassen sich auch ganze Paket- bzw. Verzeichnisbäume in eine einzige eine Java-Archivdatei verpacken (z.B. **rt.jar** beim Java-API).

6.2.3 Paketunterstützung in der JCreator-Entwicklungsumgebung

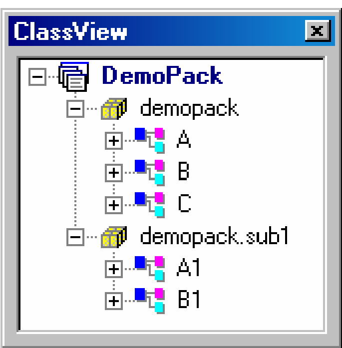
Mit unserer Entwicklungsumgebung JCreator 2.x kann das Paket `demopack` folgendermaßen erstellt werden:

- Neues Projekt beginnen, z.B. mit Projektverzeichnis `DemoPack`
- Zu jeder Klasse wird im Projektverzeichnis eine Quelldatei angelegt, wobei in der ersten Zeile eine **package**-Anweisung die Zugehörigkeit zu einem Paket oder Unterpaket festlegt (siehe oben). Im Beispiel entsteht also ein Projektverzeichnis mit folgenden Quellcode-Dateien:



- Nach einem erfolgreichen Compiler-Lauf entsteht im Projektordner der oben angegebene `demopack`-Dateiverzeichnisbaum mit den **class**-Dateien des Paketes.

Der JCreator bietet mit dem **PackageView** eine gute Hilfe, um auch bei hierarchisch strukturierten Paketen den Überblick zu behalten. Dieses Fenster kann mit dem Symbolschalter oder dem Menübefehl **View > Toolbars > PackageView** eingeschaltet werden und zeigt dann die zu einem Projekt gehörenden Pakete, z.B.:



6.3 Pakete verwenden

6.3.1 Verfügbarkeit der Dateien

Damit ein Paket genutzt werden kann, muss es sich in einem Ordner befinden, der vom Compiler bzw. Interpreter bei Bedarf nach Klassen durchsucht wird. Dafür kann man auf unterschiedliche Weise sorgen (vgl. Abschnitt 2.1.4):

- Paket im aktuellen Verzeichnis ablegen
  Dies ist natürlich keine sinnvolle Option, wenn ein Paket in mehreren Projekten eingesetzt werden soll.
- Paket in der **CLASSPATH**-Umgebungsvariablen berücksichtigen
- **classpath**-Option bei Aufruf des Compilers bzw. Interpreters angeben
  Dies geschieht im Rahmen einer korrekt konfigurierten Entwicklungsumgebung (z.B. beim JCreator) in der Regel automatisch.

Der Ordner mit den Java-API-Paketen wird grundsätzlich durchsucht, falls die Installation nicht beschädigt ist.

6.3.2 Namensregeln

Für den konkreten Zugriff auf die Klassen eines Paketes bietet Java folgende Möglichkeiten:

- **Verwendung des vollqualifizierten Klassennamens**
  Dem Klassennamen ist der durch Punkt abgetrennte Paketnamen voran zu stellen. Bei einem hierarchischen Paketaufbau ist der gesamte Pfad anzugeben, wobei die Unterpaketnamen wiederum durch Punkte zu trennen sind.
  Wir haben bereits mehrfach den Pseudozufallszahlengenerator der Klasse **Random** im Paket **java.util** auf diese Weise angesprochen, z.B.:

```
java.util.Random zzg = new java.util.Random(System.currentTimeMillis());
```

- **Import einer benötigten Klasse oder eines kompletten Paketes**
  Um die lästige Angabe von Paketnamen zu vermeiden, kann man einzelne Klassen und/oder Pakete *importieren*. Anschließend sind alle importierten Klassen direkt (ohne Paket-Präfix) ansprechbar.
  Die zuständigen **import**-Anweisungen sind an den Anfang einer Quelltextdatei zu setzen, ggf. aber *hinter* eine **package**-Deklaration.
  In folgendem Programm wird die Klasse **Random** aus dem API-Paket **java.util** importiert und verwendet:

```
import java.util.Random;
class Prog {
  public static void main(String[] args) {
    Random zzg = new Random(System.currentTimeMillis());
    System.out.println(zzg.nextInt(101));
  }
}
```

  Um *alle* Klassen aus dem Paket **java.util** zu importieren, schreibt man:

```
import java.util.*;
```

  Beachten Sie bitte, dass *Unterpakete* durch den Joker-Stern *nicht* einbezogen werden. Für sie ist bei Bedarf eine separate **import**-Anweisung fällig.
  Weil durch die Verwendung des Jokerzeichens *keine* Rechenzeit- oder Speicherressourcen verschwendet werden, ist dieses bequeme Vorgehen im Allgemeinen sinnvoll, falls keine Namenskollisionen (durch identische Klassennamen in verschiedenen Paketen) auftreten.
  Für das wichtige API-Paket **java.lang** übernimmt der Compiler den Import, so dass seine Klassen stets direkt angesprochen werden können.

In folgendem Programm wird das Paket `demopack` samt Unterpaket `sub1` importiert:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>
import demopack.*;
import demopack.sub1.*;

class PackDemo {
    public static void main(String[] ars) {
        A a = new A(), aa = new A();
        a.prinr();
        aa.prinr();
        B b = new B();
        b.prinr();
        C c = new C();
        c.prinr();
        A1 a1 = new A1();
        a1.prinr();
        B1 b1 = new B1();
        b1.prinr();
    }
}
</pre></td><td><pre>
Klasse A, Objekt Nr. 1
Klasse A, Objekt Nr. 2
Klasse B, Objekt Nr. 1
Klasse C, Objekt Nr. 1
Klasse A1, Objekt Nr. 1
Klasse B1, Objekt Nr. 1
</pre></td></tr>
</table>

Achtung: Der Compiler macht Probleme, wenn sich neben einem Paketordner (im selben Verzeichnis) die Quellcodedatei zu einer Klasse des Paketes befindet. In diesem Fall klappt das Importieren des kompletten Paketes *nicht*, während sich einzelne Klassen sehr wohl importieren lassen. Befindet sich z.B. neben dem Ordner **demopack** die Datei **A.java**, dann kommt es beim Compilieren des Programms:

```
import demopack.*;
class UTest {
  public static void main(String[] args) {
    A a = new A();
    a.prinr();
  }
}
```

zur Fehlermeldung:

```
U:\Eigene Dateien\Java\UTest\UTest.java:4: cannot access A
bad class file: U:\Eigene Dateien\Java\UTest\A.java
file does not contain class A
Please remove or make sure it appears in the correct subdirectory of the
classpath.
                A a = new A();
                ^
```

Das folgende Programm wird hingegen ohne Beanstandung übersetzt:

```
import demopack.A;
class UTest {
  public static void main(String[] args) {
    A a = new A();
    a.prinr();
  }
}
```

Entfernt man die Datei **A.java** aus dem Verzeichnis mit dem Paketordner **demopack**, dann klappt auch die Import-Anweisung mit Jokerzeichen.

6.4 Zugriffsmodifikatoren

Nach der Beschäftigung mit Paketen kann endlich präzise erläutert werden, wie in Java die für objektorientierte Softwareentwicklung außerordentlich wichtigen Zugriffsrechte auf Klassen, Instanzvariablen und Methoden festgelegt werden.

6.4.1 Zugriffsschutz für Klassen

Die Klassen eines Paketes sind für Klassen aus fremden Paketen nur dann sichtbar, wenn bei der Definition der Zugriffsmodifikator **public** angegeben wird, z.B.:

```
package demopack;
public class A {
    .    .    .
    .    .    .
}
```

Wird im `demopack`-Paket die Klasse `A` ohne **public**-Zugriffsmodifikator definiert, scheitert das Compilieren des in Abschnitt 6.3.2 vorgestellten Programms `PackDemo` mit folgender Meldung:

```
U:\Eigene Dateien\Java\Anwendungen\PackDemo> javac PackDemo.java
PackDemo.java:6: demopack.A is not public in demopack;
cannot be accessed from outside package
                A a = new A(), aa = new A();
                ^
```

Pro Quellcodedatei darf nur *eine* Klasse als **public** deklariert werden. Eventuell vorhandene zusätzliche Klassen sind also nur paketintern zu verwenden.
Diese Regel stellt allerdings keine Einschränkung dar, da man in der Regel ohnehin für jede Klasse eine eigene Quellcodedatei verwendet (mit dem Namen der Klasse plus angehängter Erweiterung **.java**).

Bei aufmerksamer Lektüre der (z.B. im Internet) zahlreich vorhandenen Java-Beschreibungen stellt man fest, dass keine einheitliche Auffassung darüber besteht, ob bei **ausführbaren Klassen** neben der statischen Methode **main()** auch die Klasse selbst als **public** definiert werden sollte, z.B.:

```
public class Hallo {
  public static void main(String[] args) {
    System.out.println("Hallo Allerseits!");
  }
}
```

Diese Praxis erscheint durchaus plausibel und systematisch, wird jedoch vom Java-Compiler bzw. – Interpreter *nicht* gefordert und stellt daher eine vermeidbare Mühe dar. Bei der Wahl einer Regel für dieses Manuskript habe ich mich am Verhalten der Java-Urheber orientiert: Gosling et at. (2000) lassen bei ausführbaren Klassen den Modifikator **public** systematisch weg.

6.4.2 Zugriffsschutz für Variablen und Methoden

Bei der Deklaration bzw. Definition von Variablen bzw. Methoden (objekt- oder klassenbezogen) können die Modifikatoren **private**, **protected** und **public** angegeben werden, um die Zugriffsrechte festzulegen. In der folgenden Tabelle wird die Wirkung bei einer Klasse beschrieben, die selbst als **public** definiert ist:

Modifikator	Der Zugriff ist erlaubt für ...			
	die eigene Klasse	**abgeleitete Klassen**	**Klassen im Paket**	**sonstige Klassen**
ohne	ja	nein	ja	nein
private	ja	nein	nein	nein
protected	ja	nur geerbte Elem.	ja	nein
public	ja	ja	ja	ja

Mit abgeleiteten Klassen und dem nur hier relevanten Zugriffsmodifikator **protected** werden wir uns bald beschäftigen.

Der vom Compiler bereit gestellte Standard-Konstruktor hat den Zugriffsschutz der Klasse.

Wird im `demopack`-Beispiel die Klasse `A` mit **public**-Zugriffsmodifikator versehen, ihre `prinr()`-Methode jedoch nicht, dann scheitert das Compilieren des Programms `PackDemo` mit folgender Meldung:

```
U:\Eigene Dateien\Java\Anwendungen\PackDemo> javac PackDemo.java
PackDemo.java:7: prinr() is not public in demopack.A;
cannot be accessed from outside package
                a.prinr();
                 ^
```

Für die *voreingestellte* Schutzstufe (nur Klassen aus dem eigenen Paket dürfen zugreifen) werden gelegentlich die Bezeichnungen *package* oder *friendly* verwendet.

6.5 Java-Archivdateien

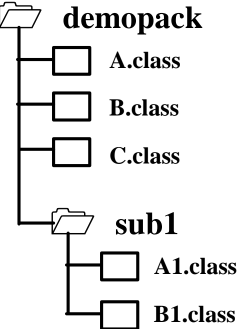
Wenn zu einem Programm zahlreiche **class**-Dateien und zusätzliche Hilfsdateien (z.B. mit Multimedia-Inhalten) gehören, dann sollten diese in einer Java-Archivdatei (Namenserweiterung **.jar**) zusammengefasst werden.
Einige Eigenschaften von Java-Archivateien:

- Übersichtlichkeit/Bequemlichkeit
  Im Vergleich zu zahlreichen Einzeldateien ist ein Archiv für den Anwender deutlich bequemer. Ein per Archiv ausgeliefertes Programm kann sogar direkt über die Archivdatei gestartet werden.
- Eine Archivdatei kann analog zu einem Verzeichnis in den Suchpfad für **class**-Dateien aufgenommen werden, z.B.:
  ```
  CLASSPATH = .;c:\Programme\DemoPack\demopack.jar
  ```
- Verkürzte Zugriffs- bzw. Übertragungszeiten
  Eine einzelne (am besten unfragmentierte) Archivdatei reduziert im Vergleich zu zahlreichen einzelnen Dateien die Zugriffszeiten beim Laden von Klassen und Ressourcen.
  Bei Java-Applets wird Übertragungszeit gespart, weil nicht für viele einzelne Dateien jeweils eine GET-Transaktion stattfinden muss.
- Kompression
  Java-Archivdateien können komprimiert werden, was für Applets durch den beschleunigten Internet-Transport sinnvoll ist, bei lokal installierten Anwendungen jedoch wegen der erforderlichen Dekomprimierung eher nachteilig. Die Archivdateien mit den Java-API-Klassen (z.B. **rt.jar**) sind daher *nicht* komprimiert.
  Weil Java-Archive das von PKWARE definierte ZIP-Dateiformat besitzen, können sie von diversen (De-)Komprimierungsprogrammen geöffnet werden. Das Erzeugen von Java-Archiven sollte man aber dem speziell für diesen Zweck entworfenen SDK-Werkzeug **jar.exe** (siehe unten) überlassen.

- Sicherheit
  Bei *signierten* JAR-Dateien kann sich der Anwender Gewissheit über den Urheber verschaffen und der Software entsprechende Rechte einräumen.

6.5.1 Archivdateien mit jar erstellen

Zum Erstellen und Verändern von Java-Archivdateien dient das Werkzeug **jar.exe** aus dem Java-SDK. Wir verwenden es dazu, aus dem `demopack`-Paket (siehe Abschnitt 6.2)



eine Archivdatei zu erzeugen. Dazu öffnen wir ein Konsolenfenster und wechseln zum Verzeichnis mit dem `demopack`-Ordner.
Dann wird mit folgendem **jar**-Aufruf das Archiv `demopack.jar` mit der gesamten Paket-Hierarchie erstellt:[1]

```
jar cf0 demopack.jar demopack
```

Darin bedeuten:

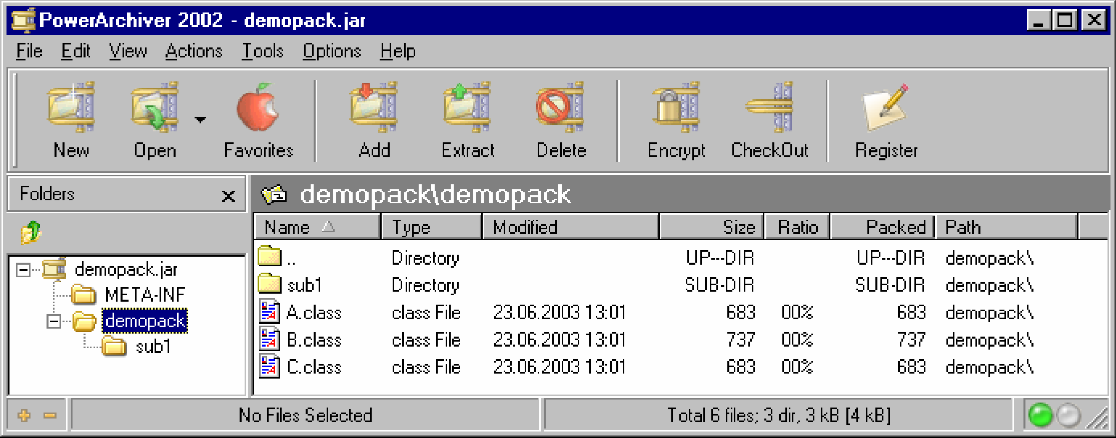
- 1. Parameter: Optionen
  Die Optionen bestehen aus jeweils einem Buchstaben und müssen unmittelbar hintereinander geschrieben werden:
  - **c**
    Mit einem **c** (für: create) wird das Erstellen eines Archivs angefordert.
  - **f**
    Mit **f** (für: file) wird die Ausgabe in eine Datei geleitet, wobei der Dateiname als zweiter Kommandozeilenparameter anzugeben ist.
  - **0**
    Mit der Ziffer **0** wird die ZIP-Kompression abgeschaltet.
- 2. Parameter: Archivdatei
  Der Archivdateiname muss einschließlich Extension (üblicherweise **.jar**) angegeben werden.
- 3. Parameter: zu archivierende Dateien und Ordner
  Bei einem Ordner wird rekursiv der gesamte Verzeichnis-Ast einbezogen. Dabei werden nicht nur Bytecode-, sondern z.B. auch Multimediadateien berücksichtigt.
  Selbstverständlich kann eine Archivdatei auch mehrere Pakete bzw. Ordner aufnehmen, was z.B. die ca. 22 MB große Datei **rt.jar** demonstriert, die praktisch alle Java-API-Pakete enthält.

Weitere Informationen über das Archivierungswerkzeug finden Sie z.B. in der SDK-Dokumentation unter *Java Archive (JAR) Files*.

---

[1] Sollte der Aufruf nicht klappen, befindet sich vermutlich das SDK-Unterverzeichnis **bin** (z.B. C:\Programme\j2sdk1.4.1\bin) nicht im Suchpfad für ausführbare Programme.

Aus obigem **jar**-Aufruf resultiert die folgende Java-Archivdatei (hier angezeigt vom Programm **PowerArchiver** zur Verwaltung von ZIP-Archiven):



6.5.2 Archivdateien verwenden

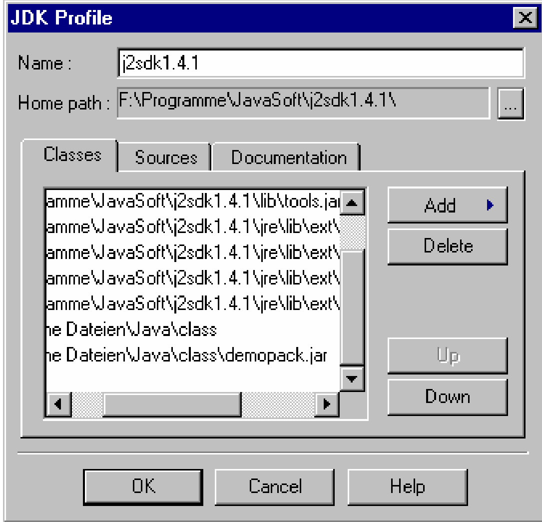
Um ein Archiv mit seinen Paketklassen nutzen zu können, muss es in den Klassen-Suchpfad des Compilers bzw. Interpreters aufgenommen werden.
Befindet z.B. die Archivdatei `demopack.jar` im aktuellen Verzeichnis zusammen mit der Quellcodedatei der Klasse `PackDemo`, die das Paket `demopack` importiert, dann kann das Übersetzen und Ausführen dieser Klasse mit folgenden Aufrufen der SDK-Werkzeuge **javac** und **java** durchgeführt werden:

```
javac -classpath demopack.jar;. PackDemo.java
java -classpath demopack.jar;. PackDemo
```

Über weitere Möglichkeiten, eine Archivdatei in den Klassen-Suchpfad aufzunehmen, wurde oben schon informiert (siehe Abschnitte 2.1.4 und 6.3.1).

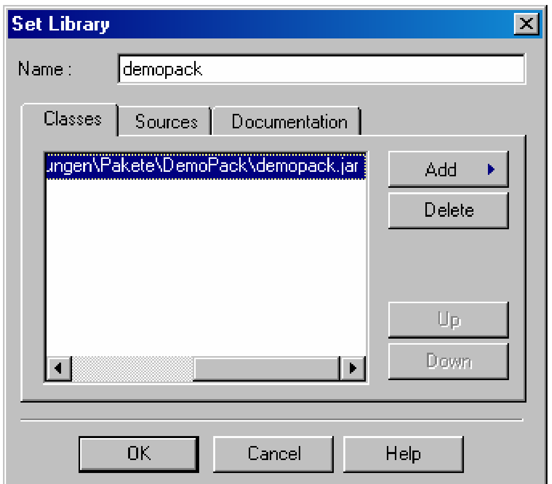
Wer mit dem JCreator 2.x arbeitet, kann eine Archivdatei folgendermaßen für *alle* Projekt verfügbar machen, die ein bestimmtes JDK-Profil verwenden:

- Dialogbox **Project Settings** öffnen mit **Project > Project Settings**
- Ein **JDK-Profile** markieren und mit **Edit** die Dialogbox **JDK-Profile** öffnen.
- Über **Add > Add Archive** eine Archivdatei wählen, z.B.:



So kann man vorgehen, wenn eine Archivdatei zunächst nur im *aktuellen* Projekt benötigt wird:

- Dialogbox **Project Settings** öffnen mit **Project > Project Settings**
- Registerkarte **Required Libraries** wählen und mit **New** die Dialogbox **Set Library** öffnen.
- Mit **Add > Add Archive** eine Archivdatei wählen und einem JCreator-internen Namen vergeben, z.B.:



6.5.3 Ausführbare JAR-Dateien

Um eine als Applikation ausführbare JAR-Datei zu erhalten, kann man folgendermaßen vorgehen:

- Man nimmt eine ausführbare Klasse in das Archiv auf, die bekanntlich eine Methode **main()** mit folgender Signatur besitzen muss:
  ```
  public static void main(String[] args)
  ```
- Diese Klasse wird im **Manifest** des Archivs als Hauptklasse eingetragen.
  Im Manifest eines Archivs wird sein Inhalt beschrieben. Es befindet sich in der Textdatei MANIFEST.MF, die das **jar**-Werkzeug im Ordner META-INF eines Archivs anlegt. Um dort z.B. die Hauptklasse `PackDemo` auszuzeichnen, legt man eine Textdatei an, die folgende Zeile und eine anschließende Zeilenschaltung enthält:
  ```
  Main-Class: PackDemo
  ```
  Im **jar**-Aufruf zum Erstellen des Archivs wird über die Option **m** eine Datei mit Manifest-Informationen angekündigt, z.B. mit dem Namen `PDApp.txt`:
  ```
  jar cmf0 PDApp.txt PDApp.jar PackDemo.class demopack
  ```
  Beachten Sie bitte, dass die Namen der Manifest- und der Archivdatei in derselben Reihenfolge wie die zugehörigen Optionen auftauchen müssen.

Nun kann die Applikation über die Archivdatei gestartet werden:

```
java -jar PDApp.jar
```

Dabei muss auf dem Zielrechner natürlich die Java-Laufzeitumgebung installiert sein.

Wenn mit dem Betriebssystem noch ein **jar**-Dateityp passend vereinbart wird, klappt der Start sogar per Mausklick auf die Archivdatei.

6.6 Übungsaufgaben zu Abschnitt 6

1) Legen Sie das seit Abschnitt 6.2 als Beispiel verwendete Paket `demopack` an. Es sind folgende Klassen zu erstellen:

- im Paket `demopack`: `A`, `B`, `C`

- im Unterpaket `sub1`: `A1`, `B1`

Erstellen Sie aus dem Paket eine Archivdatei, und verwenden Sie diese beim Übersetzen und Ausführen der im Abschnitt 6.3 wiedergegebenen Klasse `PackDemo`.

2) Wir haben früher in Mehrklassen-Anwendungen nie den Zugriffsmodifikator **public** eingesetzt, konnten aber trotzdem (z.B. in den „Hauptprogrammen“) Objekte anderer Klassen erzeugen und mit der Ausführung von Methoden beauftragen. Wieso war die **public**-Deklaration von Klassen und Methoden nicht erforderlich?

3) Erstellen Sie ein ausführbares Archiv mit unserem Währungskonvertierungsprogramm (DM in Euro). Einzupacken sind die Dateien:

- **DM2Euro.class**
  Im Ordner

  …\BspUeb\Elementare Sprachelemente\DM2EuroS

  finden Sie eine Variante, die dank Schleifenkonstruktion nicht für jeden Betrag neu gestartet werden muss.
- **Simput.class**
  `DM2Euro` benutzt eine Klassenmethode von `Simput`. Befinden sich beide **class**-Dateien in einem gemeinsamen Archiv, dann gehören die Klassen zum selben anonymen Paket, und die `Simput`–Methoden stehen in `DM2Euro` zur Verfügung.
  Sie finden die Datei **Simput.class** im Ordner:

  …\BspUeb\Simput

Außerdem wird noch eine Manifest-Datei benötigt, welche die auszuführende Klasse angibt (siehe Abschnitt 6.5.3).

Achtung: Wenn Sie im **jar**-Kommando eine Quelle mit Pfad angeben, dann wird sie im Archiv einem entsprechenden Verzeichnis zugeordnet, und die Klassen werden später vom Laufzeitsystem eventuell nicht gefunden.
So wird mit

```
jar cmf0 Manifest.txt DM2Euro.jar DM2Euro.class D:\Simput\Simput.class
```

zwar erfolgreich ein Archiv produziert, doch der anschließende Aufruf

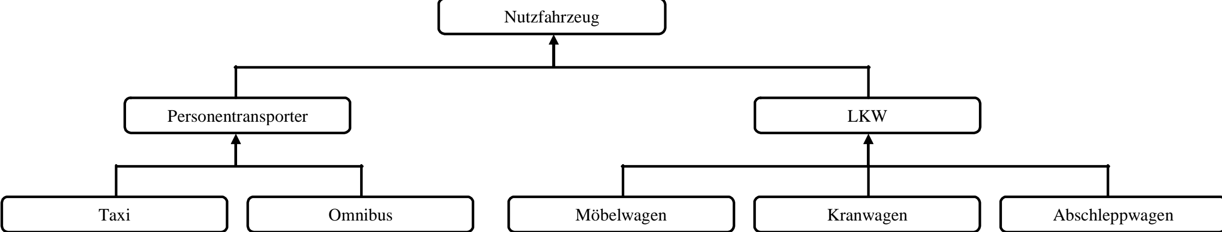
```
java -jar Dm2Euro.jar
```

führt zur Fehlermeldung

```
DM->Euro - Konvertierung

DM-Betrag oder 0 zum Beenden: Exception in thread "main" ja-
va.lang.NoClassDefFoundError: Simput
        at DM2Euro.main(DM2Euro.java:8)
```

7 Vererbung und Polymorphie

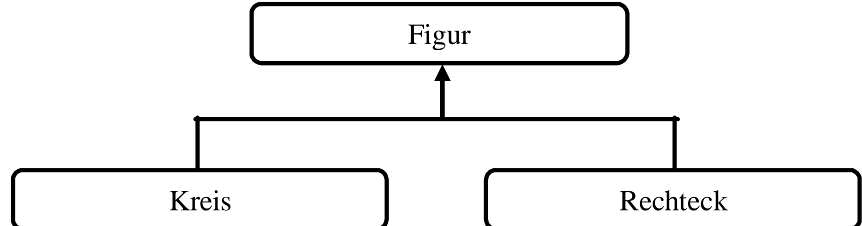
Bei einer objektorientierten Analyse einer Aufgabenstellung versucht man, alle beteiligten Objekte zu identifizieren und definiert für sie jeweils eine Klasse, die durch Eigenschaften (Instanz- und Klassenvariablen) und Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet ist. Beim Modellieren eines Gegenstandbereiches durch Java-Klassen sollten unbedingt auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet werden.
Eine Firma für Transportaufgaben aller Art mag die Nutzfahrzeuge in ihrem Fuhrpark folgendermaßen klassifizieren:



Einige Eigenschaften sind für alle Nutzfahrzeuge relevant (z.B. Anschaffungspreis, momentane Position, maximale Geschwindigkeit), andere betreffen nur spezielle Klassen (z.B. Anzahl der Fahrgäste, maximale Anhängelast, Hebekraft des Krans). Ebenso sind einige Handlungsmöglichkeiten bei allen Nutzfahrzeuge vorhanden (z.B. eigene Position melden), während andere speziellen Fahrzeugen vorbehalten sind (z.B. Fahrgäste aufnehmen, Klaviere transportieren).
Ein Programm zur Einsatzplanung und Verwaltung des Fuhrparks sollte diese Klassenhierarchie abbilden.

Übungsbeispiel

Unserer Übungsbeispiele bewegen sich natürlich in einem bescheideneren Rahmen. Zunächst betrachten wir eine einfache Hierarchie mit Klassen für geometrische Figuren:



Gerade beim Entwurf von Klassenhierarchien ist der Einsatz von expliziten Paketen eine sehr sinnvolle Option, auf die wir bei unseren Beispielen aber meist der Kürze halber verzichten wollen.

Klassenhierarchien deklarieren

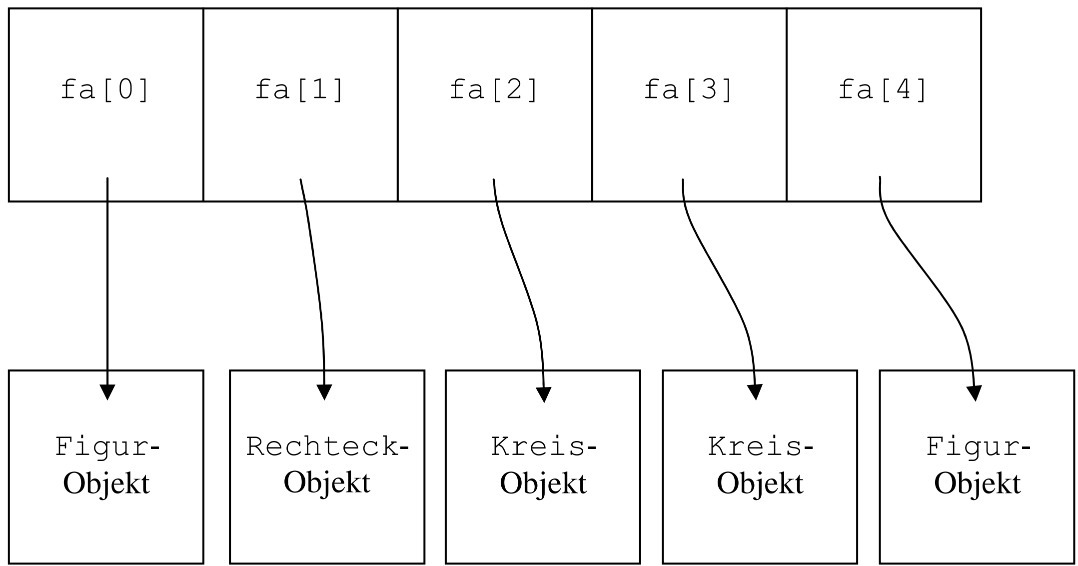
In objektorientierten Programmiersprachen wie Java ist es weder sinnvoll noch erforderlich, jede Klasse einer Hierarchie komplett neu zu deklarieren. Statt dessen geht man von der allgemeinsten Klasse aus leitet durch Spezialisierung neue Klassen ab, nach Bedarf in beliebig vielen Stufen.
Eine abgeleitete Klasse erbt alle Variablen und Methoden ihrer **Basis**- oder **Superklasse** und kann nach Bedarf Anpassungen bzw. Erweiterungen zur Lösung spezieller Aufgaben vornehmen:

- zusätzliche Variablen deklarieren
- zusätzliche Methoden definieren
- geerbte Methoden überschreiben, d.h. unter Beibehaltung des Namens umgestalten
- geerbte Variablen überdecken, z.B. unter Beibehaltung des Namens den Datentyp ändern

Polymorphie

Die Vererbungstechnologie der OOP erlaubt nicht nur die bequeme Deklaration von Klassenhierarchien, sondern bietet auch eine spezielle Flexibilität: Über Variablen vom Basisklassentyp lassen sich auch Objekte aus einer abgeleiteten Klasse verwalten. Zwar können dabei nur Methoden aufgerufen werden, die schon in der Basisklasse bekannt sind, doch kommen ggf. die überschreibenden Realisationen der *abgeleiteten* Klasse zum Einsatz. Es ist also nicht ungewöhnlich, dass z.B. ein Array vom Basisklassentyp Objekte aus verschiedenen (abgeleiteten) Klassen aufnimmt, die sich beim Aufruf „derselben“ Methode unterschiedlich verhalten.
Im Rahmen eines Grafik-Programms kommt vielleicht ein Array mit dem Elementtyp `Figur` zum Einsatz, dessen Elemente auf Objekte aus der Basisklasse oder aus einer abgeleiteten Klasse wie `Kreis` oder `Rechteck` zeigen:

Array `fa` mit Elemententyp `Figur`



Befragt man ein Objekt z.B. über die Methode `verdecktPunkt()`, ob es eine bestimmte Position überlagert, dann ermittelt es die Antwort je nach Gestalt auf unterschiedliche Weise.
Die Fähigkeit, mit Variablen vom Basisklassentyp auch Objekte aus abgeleiteten Klassen zu verwalten, bezeichnet man als *Polymorphie*. Gegen die unvermeidlichen Gewöhnungsprobleme beim Umgang mit diesem Konzept hilft nur praktische Erfahrung.

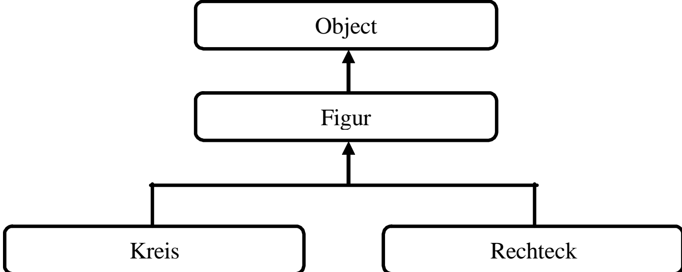
Wiederverwendbarkeit objektorientierter Software

Mit ihrem Vererbungsmechanismus bietet die objektorientierte Programmierung ideale Voraussetzungen dafür, vorhandene Software auf rationelle Weise zur Lösung neuer Aufgaben zu verwenden. Dabei können allmählich umfangreiche und dabei doch robuste und wartungsfreundliche Softwaresysteme entstehen. Spätere Verbesserungen bei einer Superklasse kommen allen (direkt oder indirekt) abgeleiteten Klassen zu Gute.
Die noch weit verbreitete Praxis, vorhanden Code per *Cut&Paste* in neuen Projekten zu verwenden, hat gegenüber einer sorgfältig geplanten Klassenhierarchie offensichtliche Nachteile.
Natürlich kann auch Java nicht garantieren, dass jede Klassenhierarchie exzellent entworfen ist und bis in alle Ewigkeit von einer stetig wachsenden Programmierer-Gemeinde eingesetzt wird.

Auch die *Polymorphie* verbessert die Chancen für produktives Software-Recycling. Dank dieser OOP-Technik kann ein Programm auch Objekte aus solchen Klassen verarbeiten, die zum Zeitpunkt seiner Übersetzung noch gar nicht existierten.

Hierarchie der Java-Klassen

In Java wird die Klassifikation insofern auf die Spitze getrieben, als *alle* Klassen (sowohl die im Java-API mitgelieferten als auch die vom Programmierer definierten) von der Klasse **Object** aus dem Paket **java.lang** abstammen. Wird (wie bei unseren bisherigen Beispielen) in der Definition einer Klasse keine Basisklasse angegeben, dann stammt sie auf direktem Wege von **Object** ab, anderenfalls indirekt.
Die oben dargestellte Klassenhierarchie zum Figuren-Übungsbeispiel muss also folgendermaßen vervollständigt werden:



7.1 Definition einer abgeleiteten Klasse

In folgendem Programm wird zunächst die Basisklasse `Figur` definiert, die Instanzvariablen für die X- und die Y-Position der linken oberen Ecke sowie einen initialisierenden Konstruktor besitzt:

```
class Figur {
   int xpos=100, ypos=100;
   Figur(int xpos_, int ypos_) {
      xpos = xpos_;
      ypos = ypos_;
      System.out.println("Figur-Konstruktor");
   }
   Figur() {}
}
```

Die Klasse `Kreis` wird mit Hilfe des Schlüsselwortes **extends** als Spezialisierung der Klasse `Figur` definiert. Sie erbt die beiden Positionsvariablen und ergänzt eine zusätzliche Instanzvariable für den Radius:

```
class Kreis extends Figur {
   int radius = 75;
   Kreis(int xpos_, int ypos_, int rad_) {
      super(xpos_, ypos_);
      radius = rad_;
      System.out.println("Kreis-Konstruktor");
   }
   Kreis() {}
}
```

Es wird ein expliziter `Kreis`-Konstrukor definiert, der über das Schlüsselwort **super** den Konstruktor der übergeordneten Klasse aufruft, statt dessen Arbeit selbst zu erledigen. Das Schlüsselwort hat übrigens den oben eingeführten Begriff *Superklasse* motiviert. In Abschnitt 7.3 werden wir uns mit einigen Regeln für **super**-Konstruktoren beschäftigen.
In folgenden Testprogramm wird ein Objekt aus der Basisklasse und ein Objekt aus der abgeleiteten Klasse erzeugt:

Quellcode	Ausgabe

`class Test {`
`  public static void main(String[] args) {`
`    Figur fig = new Figur(50, 50);`
`    Kreis krs = new Kreis(10, 10, 5);`
`  }`
`}`
	`Figur-Konstruktor`
	`Figur-Konstruktor`
	`Kreis-Konstruktor`

7.2 Der Zugriffsmodifikator protected

In folgender Variante des Beispielprogramms wird der Effekt des Zugriffsmodifikators **protected** demonstriert. Zunächst wird dafür gesorgt, dass die Klassen `Figur` und `Kreis` zu verschiedenen Paketen gehören, weil *innerhalb* eines Paketes die abgeleiteten Klassen grundsätzlich die selben Zugriffsrechte haben wie beliebige andere Klassen.

```
package fipack;

public class Figur {
    protected int xpos=100, ypos=100;
    Figur(int xpos_, int ypos_) {
        xpos = xpos_;
        ypos = ypos_;
        System.out.println("Figur-Konstruktor");
    }
    Figur() {}
}
```

Weil die Basisklasse `Figur` die Instanzvariablen `xpos` und `ypos` als **protected** deklariert, können ihre Werte in der `Kreis`-Methode `setzePos()` verändert werden, obwohl `Kreis` nicht zum selben Paket gehört:

```
import fipack.*;

class Kreis extends Figur {
    int radius = 75;
    Kreis(int xpos_, int ypos_, int rad_) {
        super(xpos_, ypos_);
        radius = rad_;
        System.out.println("Kreis-Konstruktor");
    }
    Kreis() {}
    void setzePos(int xpos_, int ypos_) {
        xpos = xpos_;
        ypos = ypos_;
    }
}
```

Es ist zu beachten, dass hier eine *geerbte Instanzvariable* von `Kreis`-Objekten verändert wird. Auf die `xpos`-Eigenschaft von `Figur`-Objekten hat auch die `Kreis`-Klasse keinen Zugriff. Wird z.B. im `Kreis`-Konstruktor ein `Figur`-Objekt erstellt, dann ist *kein* Zugriff auf dessen `xpos` erlaubt. Die folgenden Zeilen

```
Figur f = new Figur(20, 20);
f.xpos = 33;
```

führen zur Fehlermeldung

```
Kreis.java:12: xpos has protected access in fipack.Figur
   f.xpos = 33;
       ^
```

Während Objekte aus abgeleiteten Klassen ihre geerbten **protected**-Elemente direkt ansprechen können, haben fremde Klassen auf Elemente mit dieser Schutzstufe grundsätzlich keinen Zugriff:

```
import fipack.*;

class Test {
    public static void main(String[] ars) {
                  Kreis  k1 = new Kreis(50, 50, 10);
//verboten:       k1.xpos = 77;
                  k1.setzePos(77, 99);              //so klappts
    }
}
```

7.3 super-Konstruktoren und Initialisierungs-Sequenzen

Abgeleitete Klassen erben die Basisklassen-Konstruktoren *nicht*, können sie aber in eigenen Konstruktoren über das Schlüsselwort **super** benutzen. Dadurch wird z.B. es möglich, geerbte Instanzvariablen zu initialisieren, die in der Basisklasse als **private** deklariert wurden.
Wird in einem Konstruktor einer abgeleiteten Klasse kein Basisklassen-Konstruktors aufgerufen, dann ruft der Compiler implizit den parameterfreien Konstruktor der Basisklasse auf. Fehlt ein solcher, weil der Programmierer einen eigenen, parametrisierten Konstruktor erstellt und nicht durch einen expliziten parameterfreien Konstruktor ergänzt hat, dann protestiert der Compiler, z.B.:

```
Kreis.java:7: cannot resolve symbol
symbol  : constructor Figur  ()
location: class fipack.Figur
  Kreis(int xpos, int ypos, int rad_) {
                                      ^
```

Es gibt zwei offensichtliche Möglichkeiten, das Problem zu lösen:

- Im Konstruktor der abgeleiteten Klasse über das Schlüsselwort **super** einen parametrisierten Basisklassen-Konstruktor aufrufen. Dieser Aufruf muss am Anfang des Konstruktors stehen.
- In der Basisklasse einen parameterfreien Konstruktor definieren.

Der parameterfreie Basisklassen-Konstruktor wird übrigens auch vom Standard-Konstruktor der abgeleiteten Klasse aufgerufen.

Beim Erzeugen eines Unterklassen-Objektes laufen damit folgende Initialisierungs-Maßnahmen ab (vgl. Gosling et al. 2000, Abschnitt 12.5):

- Alle Elementvariablen werden mit den typspezifischen Nullwerten initialisiert.
- Der Unterklassen-Konstruktor beginnt seine Tätigkeit mit dem (impliziten oder expliziten) Aufruf eines Basisklassen-Konstruktors. Falls in der Vererbungshierarchie der Urahn **Object** noch nicht erreicht ist, wird am Anfang des Basisklassen-Konstruktors wiederum ein Konstruktor des nächst höheren Vorfahren aufgerufen, bis diese Rekursion schließlich mit dem Aufruf eines **Object**-Konstruktors endet.
  Auf jeder Hierarchieebene (beginnend bei **Object**) laufen zwei Teilschritte ab:
    - Die Instanzvariablen der Klasse werden initialisiert (gemäß Deklaration).
    - Der Rumpf des Konstruktors wird ausgeführt.

Betrachten wir als Beispiel das Geschehen beim Erzeugen eines `Kreis`-Objektes mit dem Konstruktor-Aufruf `Kreis(150, 200, 50)`:

- Die Instanzvariablen von `Kreis`-Objekten (auch die geerbten) werden angelegt und mit Nullen initialisiert.
- Aktionen für die Klasse **Object**:
    - Die Instanzvariablen der Klasse **Object** werden initialisiert.
      Derzeit sind mir zwar keine **Object**-Instanzvariablen bekannt, doch ist die Existenz von gekapselten Exemplaren durchaus möglich.
    - Der Rumpf des **Object**-Konstruktors wird ausgeführt.
      Weil der `Figur`-Konstruktor keinen expliziten Basisklassen-Konstruktor-Aufruf vornimmt, kommt der parameterfreie **Object**-Konstruktor zum Einsatz, der (bei Java 2, Version 1.4) ziemlich passiv bleibt:
      `Object() {}`
- Aktionen für die Klasse `Figur`:
    - Die Instanzvariablen `xpos` und `ypos` erhalten den Initialisierungswert laut Deklaration, was allerdings keine Änderung bewirkt.
    - Der Rumpf des Konstruktor-Aufrufs `Figur(150, 200)` wird ausgeführt, wobei `xpos` und `ypos` die Werte 150 bzw. 200 erhalten.
- Aktionen für die Klasse `Kreis`:
    - Die Instanzvariable `radius` erhält den Initialisierungswert 75 aus der Deklaration.
    - Der Rumpf des Konstruktor-Aufrufs `Kreis(150, 200, 50)` wird ausgeführt, wobei `radius` den Wert 50 erhält.

7.4 Überschreiben und Überdecken

7.4.1 Methoden überschreiben

Eine Basisklassen-Methode darf in einer Unterklasse durch eine Methode mit gleichem Namen und gleicher Parameterliste überschrieben werden, die dann für die speziellere Unterklassen-Situation ein angepasstes Verhalten realisiert.
Es liegt übrigens *keine* Überschreibung vor, wenn in der Unterklasse eine Methode mit gleichem Namen, aber abweichender Parameterliste deklariert wird. In diesem Fall sind die beiden Signaturen verschieden, und es handelt sich um eine Über*ladung*.
Unsere `Figur`-Basisklasse wird um eine Methode `wo()` erweitert, welche die Position der linken oberen Ecke ausgibt:

```
public class Figur {
    protected int xpos=100, ypos=100;
    Figur(int xpos_, int ypos_) {
        xpos = xpos_;
        ypos = ypos_;
    }
    Figur() {}
    void wo() {
        System.out.println("\nOben Links: (" + xpos + ", " + ypos + ") ");
    }
}
```

In der `Kreis`-Klasse kann eine bessere Ortsangaben-Methode realisiert werden, weil hier auch die rechte untere Ecke definiert ist[1]:

[1] Falls Sie sich über die Berechnungsvorschrift für die Y-Koordinate der rechten unteren `Kreis`-Ecke wundern: In der Computergrafik ist die Position (0, 0) in der *oberen* linken Ecke des Bildschirms bzw. des aktuellen Fensters angesiedelt. Die X-Koordinaten wachsen (wie aus der Mathematik gewohnt) nach rechts, während die Y-Koordinaten nach

```
class Kreis extends Figur {
    int radius = 75;
    Kreis(int xpos_, int ypos_, int rad_) {
        super(xpos_, ypos_);
        radius = rad_;
    }
    void wo() {
        super.wo();
        System.out.println("Unten Rechts: (" + (xpos+2*radius) +
                        ", " + (ypos+2*radius) + ")");
    }
    Kreis() {}
}
```

In der überschreibenden Methode kann man sich oft durch Rückgriff auf die überschriebene Methode die Arbeit erleichtern, wobei wieder das Schlüsselwort **super** zum Einsatz kommt.
Das folgende Programm schickt an eine `Figur` und an einen `Kreis` jeweils die Nachricht `wo()`, und beide zeigen ihr artgerechtes Verhalten:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Test {
  public static void main(String[] ars) {
    Figur f = new Figur(10, 20);
    f.wo();
    Kreis k = new Kreis(50, 50, 10);
    k.wo();
  }
}</pre></td><td><pre>Oben Links: (10, 20)

Oben Links: (50, 50)
Unten Rechts: (70, 30)</pre></td></tr>
</table>

Obwohl die beiden `wo()`-Methodenaufrufe unterschiedliches Verhalten zeigen, liegt *keine* Polymorphie vor, weil die Referenzvariablen `f` und `k` von verschiedenem Typ sind. Polymorphie setzt voraus, dass Referenzvariablen vom *selben* deklarierten Typ auf Objekte aus verschiedenen Klassen zeigen. Wenn dann über die typgleichen Referenzvariablen durch Aufruf einer (in den spezialisierten Klassen überschriebenen) Basisklassenmethode ein jeweils angepasstes Verhalten ausgelöst wird, spricht man von Polymorphie (siehe Abschnitt 7.5).

Auch bei den vom Urahn **Object** geerbten Methoden kommt ein Überschreiben in Frage. Die **Object**-Methode **toString()** liefert neben dem Klassennamen eine kodierte Objekt-Identität. Sie wird z.B. von der **String**-Methode **println()** automatisch genutzt, um die Zeichenketten-Repräsentation zu einer Objektreferenz zu ermitteln:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Test {
    static void main(String[] args) {
        Test tst1 = new Test();
        Test tst2 = new Test();
        System.out.println(tst1);
        System.out.println(tst2);
    }
}</pre></td><td><pre>Test@73d6a5
Test@111f71</pre></td></tr>
</table>

In der folgenden Klasse `Mint` wird die **Object**-Methode **toString()** überschrieben:

```
class Mint {
    int val;
    Mint(int val_) {
        val = val_;
```

unten wachsen. Wir wollen uns im Hinblick auf die in absehbarer Zukunft anstehende Programmierung grafischer Benutzeroberflächen schon jetzt daran gewöhnen.

```
    }
    public String toString() {
      return String.valueOf(val);
    }
}
```

Im Demonstrationsprogramm zeigt die **toString()**-Methode der Klasse `Mint` ein wenig spektakuläres, aber doch recht sinnvolles Verhalten:

Quellcode	Ausgabe

`class Test {`
`  public static void main(String[] args) {`
`    Mint zahl = new Mint(4711);`
`    System.out.println(zahl);`
`  }`
`}`
	4711

7.4.2 Finalisierte Methoden und Klassen

Gelegentlich möchte man das Überschreiben einer Methode *verhindern*, damit sich der Nutzer einer Unterklasse darauf verlassen kann, dass dort die geerbte Methode *nicht* überschrieben worden ist. Dient etwa die Methode `passwd()` einer Klasse `Acl` zum Abfragen eines Passwortes, will ihr Programmierer eventuell verhindern, dass `passwd()` in einer von `Acl` abstammenden Klasse `Bcl` überschrieben wird. Damit kann dem Nutzer der Klasse `Bcl` die ursprüngliche Funktionalität von `passwd()` garantiert werden.
Um das Überscheiben einer Methode zu verhindern, leitet man ihre Definition mit dem Modifikator **final** ein. Unsere `Figur`-Klasse könnte eine Methode `oleck()` zur Ausgabe der oberen linken Ecke erhalten, die von den spezialisierten Klassen nicht geändert werden muss und daher als **final** (endgültig) deklariert werden kann:

```
final public void oleck() {
      System.out.print("\nOben Links: (" + xpos + ", " + ypos + ") ");
}
```

Neben der beschriebenen Anwendungssicherheit bringt das Finalisieren einer Methode noch einen Performanz-Vorteil: Während bei nicht-finalisierten Methoden *das Laufzeitsystem* feststellen muss, welche Variante in Abhängigkeit von der Klassenzugehörigkeit des betroffenen Objektes tatsächlich ausgeführt werden soll (Polymorphie!), steht eine **final**-Methode schon beim Compilieren fest.

Die eben für das Finalisieren von *Methoden* genannten Sicherheitsüberlegungen können auch zum Entschluss führen, eine komplette Klasse mit dem Schlüsselwort **final** als *endgültig* zu deklarieren, so dass sie zwar verwendet, aber nicht mehr beerbt werden kann.

Finalisierte Klassen eignen sich auch zum Aufbewahren von Konstanten, z.B.:

```
package maipack;
final public class Konst {
   public final static int MAX = 333;
   public final static int MORITZ = 500;
   private Konst(){}
}
```

Um das unsinnige Erzeugen von Objekten zu einer Konstantenklasse zu verhindern, sollte man den Konstruktor als **private** definieren.
Das folgende Programm benutzt die eben definierte Klasse `Konst` aus dem Paket `maipack`:

Quellcode	Ausgabe
`import maipack.Konst;`	500

```
class Test {
    public static void main(String[] ars) {
        System.out.println(Konst.MORITZ);
    }
}
```

7.4.3 Elementvariablen überdecken

Wird in der abgeleiteten Klasse `Sohn` für eine Instanzvariable ein Name verwendet, der bereits eine Variable der beerbten Klasse `Vater` bezeichnet, dann wird die Basis-Variable überdeckt. Sie ist jedoch weiterhin vorhanden und kommt in folgenden Situationen zum Einsatz:

- Vom `Vater` geerbte Methoden greifen weiterhin auf die `Vater`-Variable zu, während die zusätzlichen Methoden der `Sohn`-Klasse auf die `Sohn`-Variable zugreifen.
- Analog zu einer überschriebenen Methode kann die überdeckte Variable mit Hilfe des Schlüsselwortes **super** weiterhin benutzt werden.

Im folgenden Beispielprogramm führt ein `Sohn`-Objekt eine `Vater`- und eine `Sohn`-Methode aus, um die beschriebenen Zugriffsvarianten zu demonstrieren:

Quellcode

```
// Datei Vater.java
class Vater {
 String x = "Vast";
 void vm() {
  System.out.println("x in Vater-Methode:    "+x);
 }
}
// Datei Sohn.java
class Sohn extends Vater {
 int x = 333;
 void sm() {
  System.out.println("x in Sohn-Methode:     "+x);
  System.out.println("super-x in Sohn-Meth.: "+
                        super.x);
 }
}
// Datei Test.java
class Test {
 public static void main(String[] args) {
  Sohn so = new Sohn();
  so.vm();
  so.sm();
 }
}
```

Ausgabe

```
x in Vater-Methode:    Vast
x in Sohn-Methode:     333
super-x in Sohn-Meth.: Vast
```

7.5 Polymorphie

Einer Basisklassen-Referenzvariablen kann jederzeit eine Unterklassen-Referenz zugewiesen werden. Dies ermöglicht es, Basisklassen-Referenzvariablen zur flexiblen Verwaltung von Objekten aus beliebigen Unterklassen zu verwenden.
Zur Demonstration definieren wir zunächst eine Klasse namens `Vater`:

```
class Vater {
    int iv = 1;

    void hallo() {
        System.out.println("hallo-Methode des Vaters");
    }
}
```

Daraus leiten wir Klasse namens `Sohn` ab, welche die väterliche `hallo()`-Methode überschreibt und zusätzliche Elemente ergänzt:

- die Instanzvariable `is`
- einen parameterfreien Konstruktor, der die geerbte Instanzvariable `iv` auf den Wert 2 setzt.
- die Methode `nurso()`

```
class Sohn extends Vater {
    int is = 3;

    Sohn() {iv = 2;}

    void hallo() {
        System.out.println("hallo-Methode des Sohnes");
    }

    void nurso() {
        System.out.println("nurso-Methode des Sohnes");
    }
}
```

Ein Array vom Typ `Vater` kann Referenzen auf Väter und Söhne aufnehmen, wie das folgende Beispielprogramm zeigt:

```
class Test {
  public static void main(String[] args) {
    Vater[] varray = new Vater[3];
    varray[0] = new Vater();
    varray[1] = new Sohn();
    System.out.println("iv-Wert von varray[1]: "
      + varray[1].iv);
    varray[0].hallo();
    varray[1].hallo();

    System.out.print("\nWollen Sie zum Abschluss noch einen"+
     " Vater oder einen Sohn erleben?"+
     "\nWaehlen Sie durch Abschicken von \"v\" oder \"s\": ");
    if (Character.toUpperCase(Simput.gchar()) == 'V')
      varray[2] = new Vater();
    else
      varray[2] = new Sohn();
    System.out.println();
    varray[2].hallo();
  }
}
```

Beim Ausführen der `hallo()`-Methode, stellt das Laufzeitsystem die tatsächliche Klassenzugehörigkeit fest und wählt die passende Methode aus (späte bzw. dynamische Bindung):

```
iv-Wert von varray[1]: 2
hallo-Methode des Vaters
hallo-Methode des Sohnes

Wollen Sie zum Abschluss noch einen Vater oder einen Sohn erleben?
Waehlen Sie durch Abschicken von "v" oder "s": s

hallo-Methode des Sohnes
```

Hier liegt *Polymorphie* vor, weil die beiden folgenden Kriterien erfüllt sind:

- Derselbe Methodenaufruf wird von den einzelnen Objekten unterschiedlich ausgeführt.
- Die Objekte werden über Referenzen desselben Basistyps angesprochen, und es wird erst zur Laufzeit festgestellt, welcher Klasse ein Objekt tatsächlich gehört.

Zum „Beweis", dass der Compiler die Klassenzugehörigkeit nicht kennen muss, darf im Beispielprogramm die Klasse des Array-Elementes `varray[2]` vom Benutzer festgelegt werden.

Über eine `Vater`-Referenzvariable, die auf ein `Sohn`-Objekt zeigt, sind Erweiterungen der `Sohn`-klasse (zusätzliche Elementvariablen und Methoden) nicht unmittelbar zugänglich. Wenn (auf eigene Verantwortung des Programmierers) eine Basisklassen-Referenz als Unterklassen-Referenz behandelt werden soll, um eine unterklassenspezifische Methode oder Variable anzusprechen, dann muss der **cast**-Operator verwendet werden, z.B.:

```
System.out.println(((Sohn) varray[1]).is);
```

Im Zweifelsfall sollte man sich über den **instanceof**-Operator vergewissern, ob das referenzierte Objekt tatsächlich zur vermuteten Klasse gehört.

```
if (varray[1] instanceof Sohn)
   ((Sohn) varray[1]).nurso();
```

7.6 Abstrakte Methoden und Klassen

Um die eben beschriebene gemeinsame Verwaltung von Objekten aus diversen Unterklassen über einen Array vom Basisklassentyp realisieren und dabei Polymorphie nutzen zu können, müssen die betroffenen Methoden in der Basisklasse vorhanden sein. Wenn es für die Basisklasse zu einer Methode keine sinnvolle Implementierung gibt, kann man die Methode dort als **abstract** deklarieren und auf eine Implementierung verzichten, z.B.:

```
abstract class Vater {
  abstract void hallo();
}
```

Enthält eine Klasse mindestens eine abstrakte Methode, dann handelt es sich um eine **abstrakte Klasse**, und die Klassendefinition muss mit dem Modifikator **abstract** eingeleitet werden.

Aus einer abstrakten Klasse kann man zwar keine Objekte erzeugen, aber andere Klassen ableiten. Implementiert eine abgeleitete Klasse die abstrakten Methoden, lassen sich Objekte daraus herstellen; anderenfalls ist sie ebenfalls abstrakt.
Im Beispiel werden aus dem abstrakten Vater zwei „konkrete" Klassen abgeleitet:

```
// Datei Tochter.java
class Tochter extends Vater {
  void hallo() {
    System.out.println("hallo-Methode der Tochter");
  }
}
```

```
// Datei Sohn.java
class Sohn extends Vater {
  void hallo() {
    System.out.println("hallo-Methode des Sohnes");
  }
}
```

Aus einer abstrakten Klasse lassen sich zwar keine Objekte erzeugen, doch kann sie als Datentyp verwendet werden. Referenzen dieses Typs sind ja auch unverzichtbar, wenn Objekte diverser Unterklassen gemeinsam verwaltet werden sollen:

Quellcode	Ausgabe

`class Test {`
`  public static void main(String[] args) {`
`    Vater[] varray = new Vater[2];`
`    varray[0] = new Tochter();`
`    varray[1] = new Sohn();`
`    varray[0].hallo();`
`    varray[1].hallo();`
`  }`
`}`
	`hallo-Methode der Tochter`
	`hallo-Methode des Sohnes`

7.7 Übungsaufgaben zu Abschnitt 7

1) Erweitern Sie das Beispiel mit den geometrischen Figuren (siehe Abschnitt 7.1) um eine aus `Figur` abgeleitete Klasse `Rechteck` mit zusätzlichen Instanzvariablen für Breite und Höhe und einer `wo()`-Methode, welche die geerbte `Figur`-Version überschreibt. Erstellen Sie ein Testprogramm, das polymorphe Methodenaufrufe demonstriert.

2) Warum kann der folgende Quellcode nicht übersetzt werden?

```
class Vater {
   int iv;
   Vater(int iv_) {iv = iv_;}
   void hallo() {System.out.println("hallo-Methode des Vaters");}
}

class Sohn extends Vater {
   int is = 3;
   void hallo() {System.out.println("hallo-Methode des Sohnes");}
}
```

3) Im folgenden Beispiel wird die Klasse `Kreis` aus der Klasse `Figur` abgeleitet:

```
// Datei Figur.java
package fipack;
public class Figur {
   int xpos, ypos;
}
// Datei Kreis.java
import fipack.*;
class Kreis extends Figur {
   int radius;
   void setzePos(int xpos_, int ypos_) {
      xpos = xpos_;
      ypos = ypos_;
   }
}
```

Trotzdem erlaubt der Compiler den `Kreis`-Objekten keinen direkten Zugriff auf ihre geerbten Instanzvariablen `xpos` und `ypos` (in der Methode `setzePos()`). Wie ist das Problem zu erklären und zu lösen?

8 Ausnahme-Behandlung

Zu Störungen im Programmablauf können neben Designfehlern (z.B. versuchter Feldzugriff mit falschem Indexwert) auch besondere Umstände außerhalb des Programms führen (z.B. Speichermangel, unterbrochene Netzverbindung).
Java unterstützt das Design von robusten Programmen, die auf besondere Umstände intelligent reagieren, durch eine ausgefeilte Technik zur Ausnahmebehandlung.

8.1 Unbehandelte Ausnahmen

Verhindert ein Problem die normale Ausführung einer Methode, dann wird eine so genannte **Ausnahme** (engl.: **exception**) „ausgelöst“ bzw. „geworfen“. Wir wollen etwas näher betrachten, was sich dabei abspielt:
Eine Ausnahme ist in Java ein Objekt aus der Klasse **Exception** oder aus einer problemspezifischen Unterklasse, das Informationen über den aufgetretenen Fehler enthält und über Instanzmethoden zugänglich macht. Tritt ein Fehler auf, dann wird ein **Exception**-Objekt erzeugt, und das Laufzeitsystem sucht nach einer Routine der aufrufenden Methode, die sich um das Problem kümmern kann. Wird eine solche Routine gefunden, erhält sie eine Referenz auf das Ausnahme-Objekt und kann dessen Methoden nutzen, um Informationen mit Relevanz für das weitere Vorgehen zu gewinnen. Existiert keine zum Ausnahme-Objekt passende Behandlungsroutine, untersucht das Laufzeitsystem entlang der Aufrufgeschichte die jeweils nächst höhere Methode. Findet sich letztlich auch in der **main**()-Methode keine Behandlungsroutine, dann erzeugt das Laufzeitsystem eine Fehlermeldung und beendet das Programm.

Die Initiative beim Auslösen einer Ausnahme kann ausgehen

- vom Laufzeitsystem
  Es löst etwa bei arithmetischen Fehlern (z.B. Division durch 0) eine **RuntimeException** aus.
- vom Programm, wozu auch die verwendeten Bibliotheks-Klassen gehören
  In jeder Methode kann mit der **throw**-Anweisung (siehe unten) eine Ausnahme erzeugt werden. Dies stellt eine effiziente Technik dar, den Aufrufer über ein Problem zu informieren und mit relevanten Informationen zu versorgen.

Wir werden uns anhand der verschiedenen Versionen eines Beispielprogramms damit beschäftigen,

- was bei unbehandelten Ausnahmen geschieht,
- wie man Ausnahmen abfängt,
- wie man selbst Ausnahmen wirft.

Das folgende Programm soll die Fakultät zu einer Zahl ausrechnen, die beim Start als Kommandozeilenparameter übergeben wird. Dabei beschränkt sich die **main**()-Methode auf die eigentliche Fakultätsberechnung und überlässt die Konvertierung und Validierung des übergebenen Strings der Methode `kon2int()`. Diese wiederum stützt sich bei der Konvertierung auf die Methode **Integer.parseInt**():

```
class Fakul {
   static int kon2int(String instr) {
      int arg = Integer.parseInt(instr);
      if (arg < 0  || arg > 170) {
         System.out.println("Unzulaessiges Argument: "+arg);
         System.exit(1);
         return 0;
      }
      else
         return arg;
   }

   public static void main(String[] args) {
      int argument = -1;
      if (args.length > 0)
         argument = kon2int(args[0]);
      else {
         System.out.println("Kein Argument angegeben");
         System.exit(1);
      }

      double fakul = 1.0;
      for (int i = 1; i <= argument; i++)
           fakul = fakul * i;
      System.out.println("Fakultaet: " + fakul);
   }
}
```

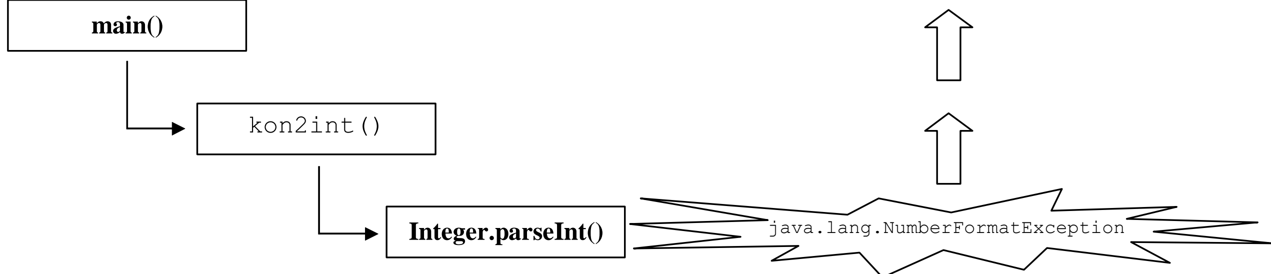
Das Programm ist durchaus bemüht, kritische Situationen zu vermeiden:

- **main()** überprüft, ob `args[0]` tatsächlich vorhanden ist, bevor dieser String beim Aufruf der Methode `kon2int()` als Parameter verwendet wird. Damit wird verhindert, dass es zu einer **ArrayIndexOutOfBoundsException** kommt, wenn der Benutzer das Programm ohne Kommandozeilen-Argument startet.
- `kon2int()` überprüft, ob die ermittelte **int**-Zahl im zulässigen Wertebereich liegt. Damit werden unsinnige Ergebnisse verhindert (z.B. Fakultät(-4) = 1).

Beide Methoden reagieren auf Probleme mit dem sofortigen Abbruch das Programms. Dazu rufen sie die Methode **System.exit()** auf, der als Aktualparameter ein Exitcode übergeben wird. Diese Vorgehensweise kann als klassische Technik zur Ausnahmebehandlung betrachtet werden, die bei groben Problemen nach wie vor in Frage kommt.
Der an **System.exit()** übergebene Aktualparameter wird dem Betriebssystem mitgeteilt und steht unter Windows in der Umgebungsvariablen ERRORLEVEL zur Verfügung, z.B.:

```
>java Except
Kein Argument angegeben

>echo %ERRORLEVEL%
1
```

Trotz der präventiven Bemühungen ist das Programm leicht aus dem Tritt zu bringen, indem man es mit einer nicht konvertierbaren Zeichnfolge füttert (z.B. „Vier“). Die zunächst betroffene Methode **Integer.parseInt()** wirft daraufhin eine **NumberFormatException**. Diese wird vom Laufzeitsystem entlang der Aufrufreihenfolge an `kon2int()` und dann an **main()** gemeldet:



Weil beide Methoden keine Behandlungsroutine bereit halten, endet das Programm mit folgender Fehlermeldung:

```
Exception in thread "main" java.lang.NumberFormatException: Vier
        at java.lang.Integer.parseInt(Integer.java:426)
        at java.lang.Integer.parseInt(Integer.java:476)
        at test.kon2int(Test.java:3)
        at test.main(Test.java:17)
```

8.2 Ausnahmen abfangen

8.2.1 Die try-Anweisung

In Java wird die Behandlung von Ausnahmen über die **try**-Anweisung unterstützt:

try {
    Überwachter Block mit Anweisungen für den normalen Programmablauf
}
catch (*Ausnahme-Klasse Parametername***) {**
    Anweisungen für die Behandlung von Ausnahmen der ersten Klasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (*Ausnahme-Klasse Parametername***) {**
    Anweisungen für die Behandlung von Ausnahmen der zweiten Klasse
}

. . .

// Optionaler Block mit Abschlussarbeiten.
finally {
    Anweisungen, die unabhängig vom Auftreten einer Ausnahme ausgeführt werden
}

Die Anweisungen für den ungestörten Ablauf setzt man in den **try**-Block. Treten bei der Ausführung dieses geschützten bzw. überwachten Blocks *keine* Fehler auf, wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**–Block ausgeführt wird.

Wird im **try**-Block eine Ausnahme ausgelöst, wird seine Ausführung abgebrochen und das Laufzeitsystem sucht nach einer **catch**-Klausel, welche eine Ausnahme aus dieser Klasse behandeln kann.
Jede **catch**-Klausel verfügt in formaler Analogie zu einer Methode in ihrem Kopfbereich über eine einelementige Parameterliste mit einem formalen Referenzparameter. Das Laufzeitsystem sucht nach einer **catch**-Klausel, deren Formalparameter vom Typ der zu behandelnden Ausnahme ist, und führt dann den zugehörigen Anweisungsblock aus.
catch-Blöcke werden oft auch als *Exeption-Handler* bezeichnet.

In der folgenden Variante der Methode `kon2int()` wird die von **Integer.parseInt()** ausgelöste Exception abgefangen:

```
static int kon2int(String instr) {
   int arg;
   try {
      arg = Integer.parseInt(instr);
   }
   catch (NumberFormatException e) {
      System.out.println("Fehler beim Konvertieren");
      System.exit(1);
      return 0;
   }

   if (arg < 0  || arg > 170) {
      System.out.println("Unzulaessiges Argument: "+arg);
      System.exit(1);
      return 0;
   }
   else
      return arg;
}
```

Beim Programmstart mit Kommandozeilenparameter „Vier" kommt es nun zu keinem Laufzeitfehler, sondern das Programm beendet seine Arbeit mit einer Fehlermeldung:

```
Fehler beim Konvertieren
```

Am Ende der **try**-Anweisung können in einem **finally**-Block solche Anweisungen stehen, die auf jeden Fall ausgeführt werden sollen, ob Ausnahmen aufgetreten sind oder nicht. Dies ist z.B. der ideale Platz, um die im **try**-Block geöffneten Dateien wieder zu schließen (siehe unten).
In anderen Programmiersprachen mit **try**-Anweisung fehlt ein **finally**-Block (z.B. in C++), so dass die Abschlussarbeiten eventuell mehrmals formuliert werden müssen.

Nach Ausführung eines **catch**-Blocks wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**–Block ausgeführt wird. Gleich folgen ausführlichere Erläuterungen zum Programmablauf bei verschiedenen Ausnahme-Konstellationen.

8.2.2 Programmablauf bei der Ausnahmebehandlung

Findet das Laufzeitsystem für eine Ausnahme in der aktuellen Methode keinen **catch**-Block, dann sucht es entlang der Aufrufsequenz weiter. Dies macht es leicht, die Behandlung einer Ausnahme der bestgerüsteten Methode zu überlassen.
In folgendem Beispiel dürfen Sie allerdings eine optimierte Einsatzplanung *nicht* erwarten. Es soll demonstrieren, welche Programmabläufe sich bei Ausnahmen ergeben können, die auf verschiedenen Stufen einer Aufrufhierarchie behandelt werden. Um das Beispiel einfach zu halten, wird auf Nützlichkeit und Praxisnähe verzichtet.
Das Programm nimmt via Kommandozeile ein Argument entgegen, interpretiert es numerisch und ermittelt den Rest aus der Division der Zahl 10 durch das Argument:

```
class Except {
   static int calc(String instr) {
      int erg = 0;
      try {
         System.out.println("try-Block von calc()");
         erg = 10 % Integer.parseInt(instr);
      }
      catch (NumberFormatException e) {
         System.out.println("NumberFormatException-Handler in calc()");
```

```
    }
    finally {
      System.out.println("finally-Block von calc()");
    }

    System.out.println("Nach try-Anweisung in calc()");
    return erg;
  }

  public static void main(String[] args) {
    try {
      System.out.println("try-Block von main()");
      System.out.println("10 % "+args[0]+" = "+calc(args[0]));
    }
    catch (ArithmeticException e) {
      System.out.println("ArithmeticException-Handler in main()");
    }
    finally {
      System.out.println("finally-Block von main()");
    }

    System.out.println("Nach try-Anweisung in main()");
  }
}
```

Die Methode **main()** lässt die eigentliche Arbeit von der Methode `calc()` erledigen und bettet den Aufruf in eine **try**-Anweisung mit **catch**-Block für die **ArithmeticException** ein.
`calc()` benutzt die Klassenmethode **Integer.parseInt()** sowie den Modulo-Operator in einem **try**-Block, wobei nur die (potentiell von **Integer.parseInt()** zu erwartende) **NumberFormatException** abgefangen wird.

Wir betrachten einige Ereignisse mit ihren Konsequenzen für den Programmablauf:

a) Normaler Ablauf
b) Exception in `calc()`, die dort auch behandelt wird
c) Exception in `calc()`, die in **main()** behandelt wird
d) Exception in **main()**, die nirgends behandelt wird

a) Normaler Ablauf

Beim Programmablauf *ohne* Ausnahmen (hier mit Kommandozeilen-Argument „8") werden die **try**- und die **finally**-Blöcke ausgeführt:

```
try-Block von main()
try-Block von calc()
finally-Block von calc()
Nach try-Anweisung in calc()
10 % 8 = 2
finally-Block von main()
Nach try-Anweisung in main()
```

b) Exception in `calc()`, die dort auch behandelt wird

Wird beim Ausführen der Anweisung

```
erg = 10 % Integer.parseInt(instr);
```

eine **NumberFormatException** an `calc()` gemeldet (z.B. wegen Kommandozeilen-Argument „acht" von **parseInt()** geworfen), kommt der zugehörige **catch**-Block zum Einsatz. Dann folgen:

- **finally**-Block in `calc()`
- restliche Anweisungen in `calc()`

An **main()** wird keine Ausnahme gemeldet, also werden nacheinander ausgeführt:

- **try**-Block
- **finally**-Block
- restliche Anweisungen

```
try-Block von main()
try-Block von calc()
NumberFormatException-Handler in calc()
finally-Block von calc()
Nach try-Anweisung in calc()
10 % acht = 0
finally-Block von main()
Nach try-Anweisung in main()
```

Zu der wenig überzeugenden Ausgabe

```
10 % acht = 0
```

kommt es, weil die **NumberFormatException** in `calc()` nicht *sinnvoll* behandelt wird. Wir werden bald elegante Möglichkeiten kennen lernen, derartige Pannen zu vermeiden. Das aktuelle Beispiel soll ausschließlich dazu dienen, Programmabläufe bei der Ausnahmebehandlung zu demonstrieren.

c) Exception in `calc()`, die in main() behandelt wird

Wird eine **ArithmeticException** an `calc()` gemeldet (z.B. wegen Kommandozeilen-Argument „0“), findet sich in der Methode kein passender Handler. Bevor die Methode verlassen wird, um entlang der Aufrufsequenz nach einem geeigneten Handler zu suchen, wird noch ihr **finally**-Block ausgeführt.
Im Aufrufer **main()** findet sich ein **ArithmeticException**–Handler, der nun zum Einsatz kommt. Dann geht es weiter mit dem zugehörigen **finally**-Block. Schließlich wird das Programm hinter der **try-**Anweisung der Methode **main()** fortgesetzt.

```
try-Block von main()
try-Block von calc()
finally-Block von calc()
ArithmeticException-Handler in main()
finally-Block von main()
Nach try-Anweisung in main()
```

d) Exception in main(), die nirgends behandelt wird

Übergibt der Benutzer gar kein Kommandozeilen-Argument, tritt in **main()** bei Zugriff auf `args[0]` eine **ArrayIndexOutOfBoundsException** auf (vom Laufzeitsystem geworfen). Weil sich kein zuständiger Handler findet, wird das Programm vom Laufzeitsystem beendet:

```
try-Block von main()
finally-Block von main()
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at Except.main(Except.java:22)
```

In einer komplexen Methode ist es oft sinnvoll, **try**-Anweisungen zu schachteln, wobei sowohl innerhalb eines **try**- als auch innerhalb eines **catch**-Blocks wiederum eine komplette **try**-Anweisung stehen darf. Daraus ergeben sich weitere Ablaufvarianten für eine flexible Ausnahmebehandlung.

8.2.3 Vergleich mit der traditionellen Ausnahmebehandlung

Eine konventionelle Ausnahmebehandlung stützt sich wesentlich auf die *Rückgabewerte*, mit denen Funktionen den Aufrufer über das Ergebnis ihrer Bemühungen informieren. Meist wird ein ganzzahliger *Returncode* verwendet, wobei die 0 einen erfolgreichen Ablauf meldet, während andere Zahlen für einen bestimmten Fehlertyp stehen. Nun ist im Prinzip nach jedem Funktionsaufruf der Returncode auszuwerten, so dass eine harmlose Sequenz von drei Aufrufen:

```
f1();
f2();
f3();
```

nach Ergänzen der Fehlerbehandlung zu einer länglichen und recht unübersichtlichen Konstruktion wird (nach Mössenböck 2001, S. 236):

```
returncode = f1();
if (returncode == 0) {
      returncode = f2();
      if (returncode == 0) {
            returncode = f3();
            if (returncode == 0) {
              .  .  .
            }
            else {
                  Ausnahmebehandlung für diverse f3()-Fehler}
      }
      else {
            Ausnahmebehandlung für diverse f2()-Fehler}
}
else {
      Ausnahmebehandlung für diverse f1()-Fehler}
```

Ein gut gesetzter Rückgabewert nutzt natürlich nichts, wenn sich der Aufrufer nicht darum kümmert. Ebenso unbeachtet kann eine *Statusvariable* bleiben, die oft alternativ zum Returncode als Kommunikationsmittel genutzt wird.

Eine weitere klassische Technik der Ausnahmebehandlung, die auch von den Beispielprogrammen in den Abschnitten 8.1 und 8.2.1 verwendet wird, ist das Beenden des kompletten Programms mit einem Exitcode, für den sich vielleicht das Betriebssystem oder ein anderes Programm interessiert.

Neben der ungesicherten Beachtung eines Returncodes ist am klassischen Verfahren aus praktischer Sicht auch zu bemängeln, dass eine Fehlerinformation aufwändig entlang der Aufrufersequenz nach oben gemeldet werden muss, wenn sie nicht an Ort und Stelle behandelt werden kann/soll.

Gegenüber der konventionellen Ausnahmebehandlung hat die in Java realisierte Technik u.a. folgende Vorteile:

- Garantierte Beachtung von Ausnahmen
  Im Unterschied zu Returncodes oder Fehlerstatusvariablen können Ausnahmen nicht ignoriert werden. Reagiert ein Programm nicht darauf, wird es vom Laufzeitsystem beendet.
- Zwang zur Behandlung von Ausnahmen
  Methoden, die Ausnahmen aus bestimmten Klassen werfen können, dürfen nur im Rahmen einer **try**-Anweisung mit geeignetem **catch**-Block ausgeführt werden. Für welche Ausnahmeklassen Behandlungszwang besteht, erfahren Sie in Abschnitt 8.4.
- Automatische Weitermeldung bis zur bestgerüsteten Methode
  Oft ist der unmittelbare Aufrufer nicht gut gerüstet zur Behandlung einer Ausnahme, z.B. nach dem vergeblichen Öffnen einer Datei. Dann soll eine „höhere“ Methode über das weitere Vorgehen entscheiden.

- Relativ geringer Aufwand
  Eine sorgfältige Ausnahmebehandlung kann sehr aufwändig werden und den Programmumfang erheblich erweitern. Daher ist es vorteilhaft, wenn *alle* Anweisungen in einem (beliebig langen **try**-Block) einer gemeinsamen Ausnahmebehandlung unterworfen werden können.
- Bessere Lesbarkeit des Quellcodes
  Mit Hilfe einer **try-catch-finally** - Konstruktion erreicht man eine Trennung zwischen den Anweisungen für den normalen Programmablauf und den diversen Ausnahme-Behandlungen, so dass der Quellcode übersichtlich bleibt.
- Bessere Fehlerinformationen für den Aufrufer
  Über ein **Exception**-Objekt kann der Aufrufer beliebig genau über einen aufgetretenen Fehler informiert werden, was bei einem klassischen, meist ganzzahligen, Rückgabewert nicht der Fall ist.

Es soll nicht verschwiegen werden, dass ein *Zwang* zur Ausnahmebehandlung lästig werden kann, weshalb die Java-Designer z.B. in den Methoden der Klassen **PrintStream** und **PrintWriter** (vgl. Abschnitt 10) keine **IOException** werfen und statt dessen ein Fehlersignal setzen, das über die Methode **checkError**() abgefragt werden kann.

8.2.4 Diagnostische Ausgaben

Statt im **catch**-Block eine *eigene* Fehlermeldung zu formulieren, kann man auch die **toString()**-Methode des übergebenen Ausnahme-Objektes aufrufen, was hier implizit im Rahmen eines **println()**-Aufrufs geschieht:

```
System.out.println(e);
```

Das Ergebnis enthält den Namen der Ausnahme und eventuell eine situationsspezifische Information, falls eine solche beim Erstellen des Ausnahme-Objektes via Konstruktor erzeugt wurde,z.B.:

```
java.lang.NumberFormatException: For input string: "Vier"
```

Wer nur die situationsspezifische Fehlerinformation, aber nicht den Namen der Exception-Klasse sehen möchte, verwendet die Methode **getMessage()**, z.B.:

```
System.out.println(e.getMessage());
```

In unserem Beispiel erscheint nur noch die Ausgabe:

```
For input string: "Vier"
```

Eine weitere nützliche Information, die ein Exeption-Objekte parat hat, ist die **Aufrufersequenz** (**stack trace**) von der ursprünglich betroffenen Methode bis zur **main()**-Methode, die von der virtuellen Maschine gestartet worden war. Mit dem Methodenaufruf

```
e.printStackTrace(System.out);
```

erhält man in unserem Beispiel:

```
java.lang.NumberFormatException: For input string: "Vier"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:426)
        at java.lang.Integer.parseInt(Integer.java:476)
        at Except.kon2int(Except.java:5)
        at Except.main(Except.java:27)
```

Die Ausnahme wird von der Methode **java.lang.Integer.parseInt(String s, int radix)** (Quellcodezeile 426 in der Datei **Integer.java**) geworfen und erhält im Konstruktor die nicht konvertierbare Zeichenfolge als Wert des **String**-Parameters **message**:

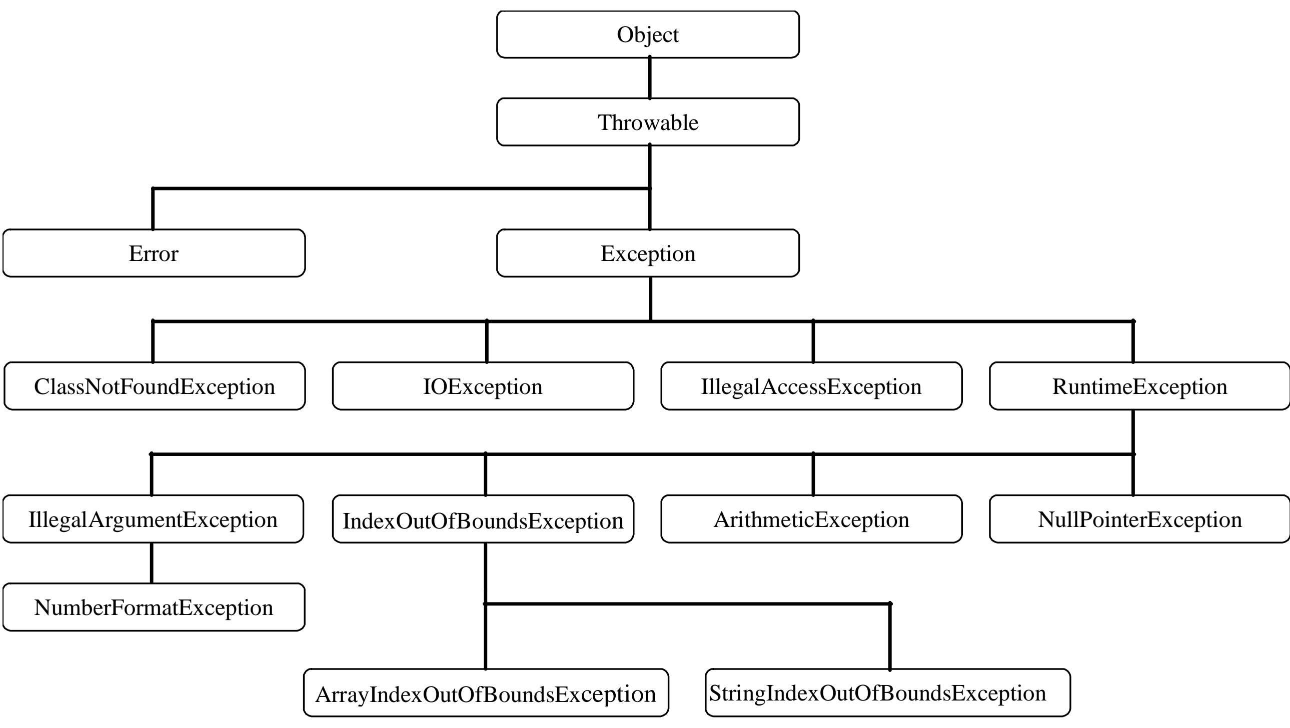
```
.
.
if (digit < 0) {
    throw new NumberFormatException(s);
} else {
    result = -digit;
}
.
.
```

Gerufen wurde **Integer.parseInt(String s, int radix)** von der Methode **Integer.parseInt(String s)**, die zwar den selben Namen besitzt, aber eine andere Signatur (Quellcodezeile 476 in der Datei **Integer.java**).
Weitere Stationen waren:

- `kon2int()` (Quellcodezeile 5 in der Datei **Except.java**)
- **main()** (Quellcodezeile 27 in der Datei **Except.java**)

8.3 Ausnahme-Klassen in Java

Java kennt zahlreiche Ausnahmeklassen, die mit ihren Vererbungsbeziehungen eine Klassenhierarchie bilden, aus der die folgende Abbildung einen kleinen Ausschnitt zeigt:



In einem **catch**-Block können auch *mehrere* Ausnahmen durch Wahl einer entsprechend breiten Ausnahme-Klasse abgefangen werden. In obigem Beispielprogramm zur Fakultätsberechnung könnte etwa statt der speziellen **NumberFormatException** auch die maximal breite Ausnahme-Klasse **Exception** angegeben werden:

```
static int kon2int(String instr) {
  int arg;
  try {
    arg = Integer.parseInt(instr);
  }
  catch (Exception e) {
    System.out.println("Fehler beim Konvertieren");
    System.exit(1);
    return 0;
  }

  .  .  .
  .  .  .

}
```

Sind mehrere **catch**-Blöcke vorhanden, dann werden diese beim Auftreten einer Ausnahme sequentiell von oben nach unten auf Zuständigkeit untersucht. Folglich ist darauf zu achten, dass speziellere Ausnahmeklassen *vor* allgemeineren stehen.

Wie die obige Klassenhierarchie zeigt, gilt neben der **Exception** auch der **Error** als **Throwable**. Allerdings geht es hier um kapitale Pannen, die auf jeden Fall einen regulären Programmablauf verhindern. Daher ist ein Abfangen von **Errors** nicht sinnvoll und nicht vorgesehen.
Kann der Interpreter z.B. eine für den Programmablauf benötigte Klasse nicht finden, wird ein **NoClassDefFoundError** gemeldet:

```
U:\Eigene Dateien\Java\Anwendungen\PackDemo>java PackDemo
Exception in thread "main" java.lang.NoClassDefFoundError: demopack/A
        at PackDemo.main(packdemo.java:7)
```

8.4 Obligatorische und freiwillige Ausnahmebehandlung

Bei Ausnahmen aus der Klasse **RuntimeException** und aus daraus abgeleiteten Klassen (siehe obige Klassenhierarchie) ist es dem Programmierer *freigestellt*, ob er sie abfangen möchte (**unchecked exceptions,** ungeprüfte Ausnahmen).
Alle übrigen Ausnahmen (z.B. **IOException**) *müssen* hingegen abgefangen werden (**checked exceptions**, geprüfte Ausnahmen).
In folgendem Programm soll mit der **read()**-Methode aus der Klasse **InputStream**, zu der das Standardeingabe-Objekt **System.in** gehört, ein Zeichen von der Tastatur gelesen werden:

```
class ChEx {
   public static void main(String[] args) {
      int key = 0;
      System.out.print("Beliebige Taste + Return: ");
      key = System.in.read();
      System.out.println(key);
   }
}
```

Weil **read()** potentiell eine **IOException** auslöst (siehe Online-Dokumentation), protestiert der Compiler:

```
ChEx.java:5: unreported exception java.io.IOException; must be
caught or declared to be thrown
        key = System.in.read();
                          ^
```

Da wir mittlerweile die **try**-Anweisung beherrschen, ist das Problem leicht zu lösen:

```
class ChEx {
    public static void main(String[] args) {
        int key = 0;
        System.out.print("Beliebige Taste + Return: ");
        try {
            key = System.in.read();
        } catch(java.io.IOException e) {
            System.out.println(e);
        }
        System.out.println(key);
    }
}
```

Gleich wird eine alternative Technik zum Umgang mit obligatorischen Ausnahmen vorgestellt, die bequemer und in vielen Fällen trotzdem ausreichend ist.

8.5 Ausnahmen auslösen (throw) und deklarieren (throws)

Unsere eigenen Methoden müssen sich nicht auf das Abfangen von Ausnahmen beschränken, die vom Laufzeitsystem oder von Bibliotheksmethoden stammen, sondern sie können sich auch als „Werfer“ betätigen, um bei misslungenen Aufrufen den Absender mit Hilfe der flexiblen Exception-Technologie zu informieren.
In folgender Variante unseres Beispielprogramms zur Fakultätsberechnung wird in der Methode `kon2int()` ein Ausnahme-Objekt aus der Klasse **IllegalArgumentException** (im Paket **java.lang**) erzeugt, wenn der Aktualparameter entweder nicht interpretierbar war, oder aber die erfolgreiche Interpretation ein unzulässiges Fakultäts-Argument ergeben hat:

```
static int kon2int(String instr) {
   int arg;
   try {
      arg = Integer.parseInt(instr);
      if (arg < 0  || arg > 170)
         throw new IllegalArgumentException ("Unzulaessiges Argument: "+arg);
      else
         return arg;
   }
   catch (NumberFormatException e) {
      throw new IllegalArgumentException ("Fehler beim Konvertieren");
   }
}
```

Zum Auslösen einer Ausnahme dient die **throw**-Anweisung. Sie enthält nach dem Schlüsselwort **throw** eine Referenz auf ein Ausnahme-Objekt. Wie im Beispiel benutzt man oft den **new**-Operator mit nachfolgendem Konstruktor, um das Ausnahme-Objekt zu erzeugen und die Referenz zu liefern.
Die meisten Ausnahme-Klassen besitzen *zwei* Konstruktoren:

- einen parameterfreien Konstruktor
- einen Konstruktor mit einem **String**-Parameter für eine Fehlermeldung (nähere Beschreibung der Ausnahme)

Wer eine fremde Methode benutzt, sollte wissen, welche Ausnahmen diese Methode auslöst und dann *nicht* lokal behandelt, so dass die aufrufende Methode eventuell zur Ausnahmebehandlung aufgefordert wird. Mit der **throws**-Klausel im Methodenkopf kann man darüber informieren, z.B.:

```
static int kon2int(String instr) throws IllegalArgumentException
```

Durch Kommata getrennt können in der **throws**-Klausel mehrere Ausnahme-Klassen angemeldet werden.

Bei **unchecked exceptions** (**RuntimeException** und Unterklassen) ist es dem Programmierer *freigestellt*, ob er die in einer Methode ausgelösten, aber nicht behandelten, Exceptions deklarieren möchte. Für alle übrigen Ausnahmen (z.B. **IOException**) gilt: Wenn eine Methode eine solche Ausnahme auslöst, muss sie diese entweder selbst abfangen oder aber in der **throws**-Liste ihres Methodenkopfs auflisten.

Dass eine Methode die selbst geworfenen Ausnahmen auch wieder auffängt, ist nicht unbedingt der Standardfall, aber in vielen Situationen eine praktische Möglichkeit, von verschiedenen potentiellen Schadstellen aus zur selben Ausnahmebehandlung zu verzweigen. Diese Technik wird im folgendem Beispiel ohne störenden Inhalt demonstriert:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td>

```
import java.io.*;
class Demo {
    static void nix() {
        try {
            // Anweisungen
            if (false)
             throw new IOException("A");
            // Anweisungen
            if (false)
             throw new IOException("B");
            // Anweisungen
            if (true)
             throw new IOException("C");
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }
    public static void main(String args[]) {
        nix();
    }
}
```

</td><td>

```
java.io.IOException: C
```

</td></tr>
</table>

In Abschnitt 8.4 haben Sie erfahren, dass man beim Aufruf einer Methode, die potentiell *obligatorische* Ausnahmen wirft, „präventive Maßnahmen“ ergreifen *muss*. In der Regel ist es empfehlenswert, die kritischen Aufrufe in einem geschützten Block vorzunehmen. Es ist aber auch erlaubt, über das Schlüsselwort **throws** die Verantwortung auf die nächst höhere Ebenen der Aufrufhierarchie abzuschieben. Im Beispielprogramm aus Abschnitt 8.4 kann sich die Methode **main()**, welche den potentiellen **IOException**-Absender **read()** ruft, der Pflicht zur Ausnahmebehandlung auf folgende Weise entledigen:

```
class ChEx {
    public static void main(String[] args) throws java.io.IOException {
        int key = 0;
        System.out.print("Beliebige Taste + Return: ");
        key = System.in.read();
        System.out.println(key);
    }
}
```

Man kann mit **throws** also nicht nur selbst erzeugte Ausnahmen anmelden, sondern auch Ausnahmen weiterleiten, die von aufgerufenen Methoden stammen.

8.6 Ausnahmen definieren

Mit Hilfe von Ausnahme-Objekten kann eine Methode beim Auftreten von Fehlern den aufrufenden Programmteil ausführlicher und präziser über Ursachen und Begleitumstände informieren, als dies mit klassischen Mitteln der Ausnahmebehandlung (z.B. Returncode, globale Statusvariablen) möglich wäre. Dabei muss man sich keinesfalls auf die im Java-API vorhandenen Ausnahme-Klassen beschränken, sondern kann eigene Ausnahmen definieren, z.B.:

```
public class BadFakulArgException extends Exception {
   protected int error, value;
   protected String instr;
   public BadFakulArgException(String desc, String instr_, int error_, int value_) {
      super(desc);
      instr = instr_;
      error = error_;
      value = value_;
   }

   public String getInstr() {return instr;}
   public int getError() {return error;}
   public int getValue() {return value;}
}
```

Hier wird die Klasse `BadFakulArgException` unter Verwendung des Schlüsselwortes **extends** aus der Ausnahme-Klasse **Exception** abgeleitet.
Durch Verwendung der handgestrickten Ausnahmeklasse `BadFakulArgException` kann unsere Methode `kon2int()` beim Auftreten von irregulären Argumenten neben einem beschreibenden String noch weitere Informationen an aufrufende Methoden übergeben:

- in `instr` die zu konvertierende Zeichenfolge
- in `error` einen numerischen Indikator für die Fehlerart
- in `value` das Konversionsergebnis (falls vorhanden, sonst -1)

Außerdem haben wir durch die Entscheidung, `BadFakulArgException` aus **Exception** abzuleiten, eine *checked exception* erzeugt, die im `kon2int()`-Methodenkopf deklariert werden *muss*:

```
static int kon2int(String instr) throws BadFakulArgException {
    int arg;
    try {
        arg = Integer.parseInt(instr);
        if (arg < 0  || arg > 170)
            throw new BadFakulArgException("Unzulaessiges Argument", instr, 2, arg);
        else
            return arg;
    }
    catch (NumberFormatException e) {
        throw new BadFakulArgException("Fehler beim Konvertieren", instr, 1, -1);
    }
}
```

Ebenso ist die **main()**-Methode gezwungen, beim `kon2int()`-Aufruf die `BadFakulArgException` abzufangen:

```
public static void main(String[] args) {
    int argument = -1;
    if (args.length == 0) {
        System.out.println("Kein Argument angegeben");
        System.exit(1);
    }

    try {
        argument = kon2int(args[0]);
        double fakul = 1.0;
        for (int i = 1; i <= argument; i += 1)
                fakul = fakul * i;
        System.out.println("Fakultaet: " + fakul);
    }
```

```
    catch (BadFakulArgException e) {
        System.out.println("Wg. Fehler " + e.getError() + " (" +
                        e.getMessage() + ") keine Berechnung moeglich.");
        switch (e.getError()) {
            case 1 :    System.out.println("String: \""+e.getInstr()+"\"");
                        break;
            case 2 :    System.out.println("Wert: "+e.getValue());
                        break;
        }
    }
}
```

Um eine *unchecked exception* zu erzeugen, wählt man eine Basisklasse aus der **RuntimeException**–Hierarchie.

8.7 Übungsaufgaben zu Abschnitt 8

1) Erstellen Sie ausnahmsweise ein Programm, das eine **NullPointerException** auslöst.
Hinweis: Man kann eine Referenzvariable „unbrauchbar" machen, indem man ihr den Wert **null** zuweist. Sofern keine weiteren Referenzen auf das Objekt zeigen, wird es bei Gelegenheit vom Müllsammler (garbage collector) entsorgt.

2) Beim Rechnen mit Gleitkommazahlen produziert Java in kritischen Situationen üblicherweise keine Ausnahmen, sondern operiert mit speziellen Werten wie **Double.POSITIVE_INFINITY** oder **Double.NaN**. Dieses Verhalten ist sicher oft nützlich, kann aber eventuell die Fehlersuche erschweren, wenn mit den speziellen Funktionswerten weiter gerechnet wird, und erst am Ende eines längeren Rechenweges das Ergebnis **NaN** auftaucht.
In folgendem Beispiel wird eine Methode zur Berechnung des dualen Logarithmus[1] verwendet, welche auf **java.lang.Math.log()** zurück greift und daher bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert.

Quellcode	Ausgabe

`class DuaLog {`
`    static double duaLog(double arg) {`
`        return Math.log(arg) / Math.log(2);`
`    }`
 	
`    public static void main(String[] args) {`
`        double a = duaLog(8);`
`        double b = duaLog(-1);`
`        System.out.println(a*b);`
`    }`
`}`
	`NaN`

Erstellen Sie eine Version, die bei ungeeigneten Argumenten eine **IllegalArgumentException** wirft.

---

[1] Für eine positive Zahl *a* ist ihr Logarithmus zur Basis *b* (> 0) definiert durch:

$$\log_b(a) := \frac{\log(a)}{\log(b)}$$

Dabei steht log() für den natürlichen Logarithmus zur Basis e (Eulersche Zahl).

9 Interfaces

Eventuell haben Sie sich schon einmal gefragt, was mit den *Implemented Interfaces* gemeint ist, die in der SDK-Dokumentation zu zahlreichen API-Klassen an prominenter Stelle angegeben werden, z.B. bei der Wrapper-Klasse **java.lang.Double**:

java.lang

Class Double

```
java.lang.Object
  |
  +--java.lang.Number
        |
        +--java.lang.Double
```

All Implemented Interfaces:

Comparable, Serializable

In diesem konkreten Fall wird über die Klasse ausgesagt:

- **Comparable**
  Weil die Klasse **java.lang.Double** das Interface (die Schnittstelle) **java.lang.Comparable** implementiert, können die Objekte in einem **Double**-Array mit der (statischen) Methode **java.util.Arrays.sort()** bequem sortiert werden, z.B.:
  ```
  Double[] da = new Double[13];
     .    .    .
  java.util.Arrays.sort(da);
  ```
- **Serializable**
  Weil die Klasse **Double** auch das Interface **java.io.Serializable** implementiert, können **Double**-Objekte mit Hilfe von Byte-Strömen auf bequeme Weise gespeichert und eingelesen werden. Diese (bei komplexeren Klassen beeindruckende) Option werden wir bald im Abschnitt 10 über Ein- und Ausgabe kennen lernen.

Durch ein Interface werden Verhaltenskompetenzen von Objekten definiert, wobei im Kern meist eine Liste von Methoden-Signaturen steht, also eine Schnittstelle für die Interaktion mit anderen Objekten. Interfaces signalisieren „Zusatzqualifikationen“ von Klassen und können als Referenzdatentypen verwendet werden.

Implementiert eine Klasse ein Interface, …

- muss sie alle damit verbundenen Verpflichtungen erfüllen, z.B. Methoden mit den im Interface geforderten Signaturen anbieten.
- können bestimmte Methoden mit Objekten dieser Klasse ausgeführt werden. So setzt z.B. **java.util.Arrays.sort()** das Interface **Comparable** voraus.
- werden Variablen vom Typ dieser Klasse vom Compiler akzeptiert, wo der Interface-Datentyp vorgeschrieben ist.

Im aktuellen Abschnitt 9 werden die Begriffe *Interface* und *Schnittstelle* synonym benutzt

9.1 Interfaces definieren

Wenngleich das *Implementieren* einer Schnittstelle im Programmieralltag eher relevant wird als das *Definieren*, fangen wir mit Letzterem an, weil dabei Struktur und Funktion einer Schnittstelle deutlich werden.

Eine Schnittstelle ist ein in Java erstmals verwendetes OOP-Sprachelement, das wir zunächst als Klasse mit ausschließlich abstrakten Methoden begreifen können. Damit sind keine *Objekte* von diesem Datentyp denkbar (so wenig wie bei abstrakten Klassen), aber *Referenzvariablen* sind erlaubt und als Abstrakationsmittel sehr nützlich. Sie dürfen auf Objekte aus *allen* Klassen zeigen, welche die Schnittstelle implementieren.
Betrachten wir als Beispiel die Definition der Schnittstelle **java.lang.Comparable**:[1]

```
public interface Comparable {
    public int compareTo(Object o);
}
```

Im Schnittstellenkopf ist neben dem Schlüsselwort **interface** und dem Namen noch der Modifikator **public** mit der üblichen Bedeutung erlaubt. Im Schnittstellenrumpf werden abstrakte Methoden aufgeführt, deren Rumpf durch ein Semikolon ersetzt ist.
Nun wird deutlich, was mit *Schnittstelle* bzw. *interface* gemeint ist: Hier wird festgeschrieben, dass Objekte eines Datentyps bestimmte Methodenaufrufe beherrschen müssen, wobei die Implementierung offen bleibt.

Einige allgemeine Regeln für Schnittstellen-Definitionen:

Modifikator **public**	Wird **public** *nicht* angegeben, ist die Schnittstelle nur innerhalb ihres Paketes verwendbar.
Modifikator **abstract**	Weil Schnittstellen grundsätzlich **abstract** sind, wird der Modifikator *nicht* angegeben.
Schlüsselwort **interface**	Das obligatorische Schlüsselwort dient zur Unterscheidung zwischen Klassen- und Schnittstellen-Deklarationen.
Schnittstellenname	Wie bei Klassennamen sollte man den ersten Buchstaben groß schreiben.
extends Interfac2, Interfac3	Die von Klassen bekannte Vererbung über das Schlüsselwort **extends** wird auch bei Interfaces unterstützt. Dabei gilt: • Während bei Java-*Klassen* die (z.B. von C++ bekannte) **Mehrfachvererbung** *nicht* unterstützt wird, ist sie bei Java-*Schnittstellen* möglich (und oft auch sehr sinnvoll). • Die Hierarchie der Java-Interfaces ist unabhängig von der Klassen-Hierarchie.
Methodendeklaration	In einer Schnittstelle sind alle Methoden grundsätzlich **public** und **abstract**. Die beiden Schlüsselwörter können also weggelassen werden. Ein dem Schlüsselwort **public** widersprechender Zugriffsmodifikator ist verboten. Es sind auch Schnittstellen ohne Methoden erlaubt. Diese eignen sich zum Aufbewahren von Konstanten.

Konstanten	Neben Methoden sind auch Variablen erlaubt, wobei diese implizit als **public**, **final** und **static** deklariert sind.
	Beispiel:
	`public interface Farben {`
	`    int ROT = 1, GRUEN = 2, BLAU = 3;`
	`}`

Bekennt sich eine Klasse zu einer Schnittstelle, dann ...

[1] Sie finden diese Definition in der Datei **Comparable.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der SDK-Installation auf Ihre Festplatte befördert werden.

- muss sie alle Methoden der Schnittstelle implementieren (oder sich selbst als **abstract** deklarieren),
- kann sie auf alle Konstanten der Schnittstellen-Deklaration ohne Angabe des Schnittstellennamens zugreifen.

Die `Demo`-Schnittstelle in folgendem Beispiel verlangt das Implementieren einer `say1()`-Methode und enthält eine **int**-Konstante namens `ONE`:

```
// Datei Demo.java
interface Demo {
    int ONE = 1;
    int say1();
}
// Datei ImplDemo.java
class ImplDemo implements Demo {
    public int say1() {return ONE;}
}

// Datei Test.java
class Test {
    public static void main(String[] args) {
        ImplDemo tobj = new ImplDemo();
        System.out.println("ImplDemo-Objekt: " + tobj.say1());
        System.out.println("Test-Klasse:     " + Demo.ONE);
    }
}
```

Man kann die Konstanten einer Schnittstelle auch ohne Implementierung ansprechen, indem man den Schnittstellennamen mit Hilfe des Punktoperators vor den Variablennamen setzt. Damit eignen sich Schnittstellen auch als Konstantenklassen, wobei man auf jede Methodendeklaration verzichtet.

Es sind auch Schnittstellen erlaubt, die weder Methoden noch Konstanten enthalten. Ein besonders wichtiges Beispiel ist die in Abschnitt 10 über Datenströme zu behandelnde Schnittstelle **java.io.Serializable**:[1]

```
public interface Serializable {
}
```

Durch das Implementieren dieser Schnittstelle teilt eine Klasse mit, dass sie gegen das Serialisieren ihrer Objekte nichts einzuwenden hat.

9.2 Interfaces implementieren

Soll für die Objekte einer Klasse angezeigt werden, dass sie auch den Datentyp einer bestimmten Schnittstelle erfüllen, muss diese im Kopf der Klassendefinition nach dem Schlüsselwort **implements** aufgeführt werden. Als Beispiel dient eine Klasse namens `Figur`, die nur begrenzte Ähnlichkeit mit namensgleichen früheren Beispiel-Klassen besitzt. Sie implementiert das Interface **Comparable**, damit `Figur`-Arrays bequem sortiert werden können:

```
class Figur implements Comparable {
    int xpos, ypos;
    String name = "unbenannt";
    Figur(String name_, int xpos_, int ypos_) {
        name = name_; xpos = xpos_; ypos = ypos_;
    }
```

[1] Sie finden diese Deklaration in der Datei **Serializable.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der SDK-Installation auf Ihre Festplatte befördert werden.

```
    public int compareTo(Object obj) {
        Figur jene = (Figur) obj;
        if (xpos < jene.xpos)
            return -1;
        else if (xpos == jene.xpos)
            return 0;
        else
            return 1;
    }
}
```

Alle Methoden einer im Kopf angemeldeten Schnittstelle müssen im Rumpf der Klassendefinition implementiert werden, sofern keine abstrakte Klasse entstehen soll. Nach der in Abschnitt 9.1 wiedergegebenen **Comparable**-Deklaration ist also im letzten Beispiel eine Methode mit der folgenden Signatur erforderlich:

```
public int compareTo(Object o);
```

In semantischer Hinsicht soll sie eine `Figur` beauftragen, sich mit dem per Aktualparameter bestimmten Artgenossen zu vergleichen. Bei obiger Realisation werden Figuren nach der X-Koordinate ihrer linken oberen Ecke verglichen:

- Liegt die angesprochene Figur links vom Vergleichspartner, dann wird –1 zurück gemeldet.
- Haben beide Figuren in der linken oberen Ecke dieselbe X-Koordinate, lautet die Antwort 0.
- Ansonsten wird eine 1 gemeldet.

Damit wird eine *Anordnung* der `Figur`-Objekte definiert und einem erfolgreichen Sortieren (z.B. per **java.util.Arrays.sort()**) steht nichts mehr im Wege.

Damit über den (per **Comparable**-Definition vorgeschriebenen) Referenzparameter vom Typ **Object** die speziellen Eigenschaften und Attribute von `Figur`-Objekten angesprochen werden können, ist eine explizite Typanpassung erforderlich. Im Beispiel wird eine `Figur`-Referenzvariable deklariert, damit diese Typanpassung nicht bei jedem Zugriff fällig ist:

```
Figur jene = (Figur) obj;
```

Ist das an die `Figur`-Implementation von **compareTo()** übergebene Vergleichsobjekt keine `Figur`, denn führt die explizite Typanpassung zu einer **java.lang.ClassCastException**.

Weil die Methoden einer Schnittstelle grundsätzlich **public** sind, muss diese Schutzstufe auch für die *implementierenden* Methoden gelten, wozu in deren Deklaration der Zugriffsmodifikator **public** explizit anzugeben ist. Anderenfalls äußert sich der Compiler so:

```
U:\Eigene Dateien\Java\Fimplement\Figur.java:8: compareTo(java.lang.Object) in
Figur cannot implement compareTo(java.lang.Object) in java.lang.Comparable; at-
tempting to assign weaker access privileges; was public
      int compareTo(Object obj) {
          ^
```

Während eine Klasse nur *eine* Super- bzw. Basisklasse besitzt, kann sie beliebig viele Schnittstellen implementieren, so dass ihre Objekte entsprechend viele Datentypen erfüllen. Wie wir inzwischen wissen, wird der Klasse aber nichts geschenkt (mal abgesehen von den Konstanten der Schnittstellen), sondern sie schließt Verträge mit dem Compiler ab und muss die vertragsgemäßen Leistungen erbringen.

Auch Schnittstellen ändern nichts daran, dass für Java-*Klassen* eine Mehrfach*vererbung* ausgeschlossen ist. Beim Definieren der Programmiersprache wurde diese Möglichkeit wegen einiger

Risiken bewusst *nicht* aus C++ übernommen. Allerdings erlauben Schnittstellen in vielen Fällen eine Ersatzlösung, denn:

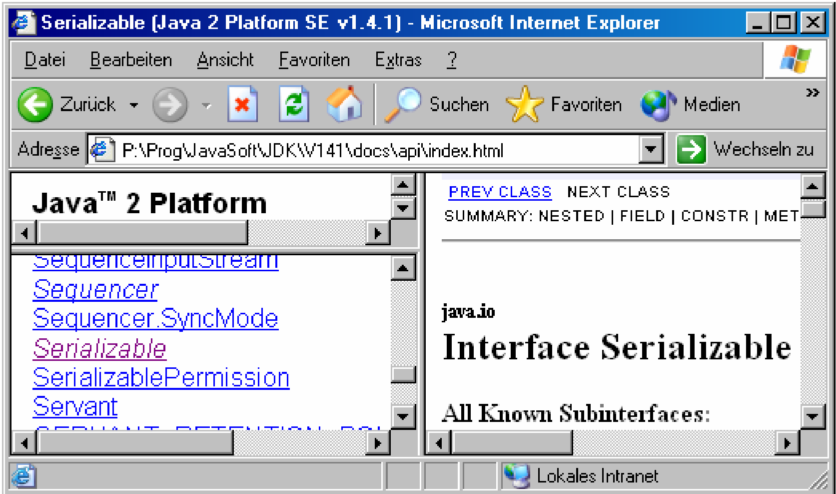
- Eine Klasse darf beliebig viele Schnittstellen implementieren.
- Bei Schnittstellen ist Mehrfachvererbung erlaubt.

Im Zusammenhang mit dem Thema *Vererbung* ist noch von Bedeutung, dass eine Unterklasse die Schnittstellen ihrer Basisklasse erbt. Wird z.B. die Klasse `Kreis` von der oben vorgestellten Klasse `Figur` abgeleitet, so übernimmt sie auch die Schnittstelle **Comparable**, und die statische **sort()**-Methode der Klasse **Arrays** kann auf Felder mit `Kreis`-Elementen angewendet werden:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>import java.util.*;
class Test {
   public static void main(String[] ars) {
      Kreis[] k = new Kreis[3];
      k[0] = new Kreis("A", 250, 50, 10);
      k[1] = new Kreis("B", 150, 50, 20);
      k[2] = new Kreis("C", 50, 50, 30);
      System.out.println(k[0].name + " "
         + k[1].name + " " + k[2].name);
      Arrays.sort(k);
      System.out.println(k[0].name + " "
         + k[1].name + " " + k[2].name);
   }
}</pre></td><td><pre>A B C
C B A</pre></td></tr>
</table>

Auch die Schnittstellen bilden eine Hierarchie, doch ist diese unabhängig von der Klassenhierarchie. Daher kann eine Schnittstelle von beliebigen Klassen implementiert werden.

Über die Interfaces im Java-API informiert die SDK-Dokumentation, wobei sie im Navigations-Frame an den kursiv gesetzten Namen zu erkennen sind, z.B.:



9.3 Interfaces als Referenz-Datentypen verwenden

Mit der Definition einer Schnittstelle wird ein neuer Referenzdatentyp vereinbart, der anschließend in Variablendeklarationen und Parameterlisten einsetzbar ist. Eine Referenzvariable des neuen Typs kann auf Objekte jeder Klasse zeigen, welche die Schnittstelle implementiert, z.B.:

Quellcode

```
interface DemInt {
   int sagWas();
}

class ImplDemo1 implements DemInt {
   public int sagWas() {return 1;}
}

class ImplDemo2 implements DemInt {
   public int sagWas() {return 2;}
}

class Test {
   public static void main(String[] args) {
      DemInt tobj1 = new ImplDemo1(),
             tobj2 = new ImplDemo2();
      System.out.println("ImplDemo1-Objekt: " +
                         tobj1.sagWas());
      System.out.println("ImplDemo2-Objekt: " +
                         tobj2.sagWas());
   }
}
```

Ausgabe

```
ImplDemo1-Objekt: 1
ImplDemo2-Objekt: 2
```

Damit wird es z.B. möglich, Objekte aus beliebigen Klassen in einem Array gemeinsam zu verwalten, sofern alle Klassen dasselbe Interface implementieren. Zwar lässt sich derselbe Zweck auch mit **Objekt**-Referenzen erreichen, doch leidet unter so viel Liberalität die Typsicherheit.

9.4 Übungsaufgaben zu Abschnitt 9

1) Indem eine Klasse das Interface **Cloneable** implementiert, erlaubt sie das *Klonen* ihrer Objekte, wobei ein neues Objekt mit identischen Werten für alle Instanzvariablen entsteht.
Zwar besitzt das Interface **Cloneable** keine Methoden, doch sollte eine implementierende Klasse die als **protected** definierte **Object**-Methode **clone()** durch eine **public**-Methode überschreiben, wobei die von **clone()** potentiell zu erwartende **CloneNotSupportedException** abgefangen oder deklariert werden muss.

Probieren Sie das Objekt-Klonen mit folgender Klasse aus:

```
class Klaas implements Cloneable {
   private static int anzahl;
   private int nr;

   Klaas() {
      nr = ++anzahl;
   }

   void hallo() {
      System.out.println("Ich bin Klaas Nr. " + nr);
   }

   public Object clone() throws CloneNotSupportedException {
      return super.clone();
   }
}
```

Per `Klaas`-Konstruktor erhält eigentlich jedes Objekt eine eindeutige Nummer. Ein geklontes Objekt sollte jedoch denselben `nr`-Wert erhalten wie das Original.

2) Das folgende Programm enthält einige Trockenübungen zur Verwendung von **Interface**-Datentypen und außerdem einen Fehler, den sie entdecken und beseitigen sollen:

```
interface Leer {}

class ImplDemo implements Leer {
  public int sagWas() {return 1;}
}

class Test {
  public static void main(String[] args) {
    Leer tobj = new ImplDemo();
    System.out.println(tobj.sagWas());
  }
}
```

10 Ein-/Ausgabe über Datenströme

In diesem Abschnitt behandeln wir elementare Verfahren zum sequentiellen Datenaustausch zwischen einem Java-Programm und externen Datenquellen bzw. –senken:

- Primitive Werte, Zeichen oder ganze Objekte in Dateien schreiben bzw. von dort lesen
- Primitive Werte oder Zeichen auf der Konsole ausgeben bzw. von dort entgegen nehmen

Damit werden u.a folgende Themen *nicht* angesprochen:

- Alternative Datenquellen bzw. -senken (z.B. Netzwerkverbindungen, Pipes)
- Wahlfreier Dateizugriff

Die Java-Klassen zur Datenein- und -ausgabe befinden sich im Paket **java.io**, das folglich von jedem betroffenen Programm importiert werden sollte, z.B.:

```
import java.io.*;
```

Im Bemühen um eine umfassende und systematische Behandlung der diversen Ein-/Ausgabeproblemstellungen hat sich eine stattliche und auf den ersten Blick recht unübersichtliche Anzahl von **io**-Klassen ergeben. Die folgende Einschätzung von Eckel (2002, TIJ314.htm) ist als Warnung und vorsorglicher Trost gedacht:

> As a result there are a fair number of classes to learn before you understand enough of Java's I/O picture that you can use it properly. In addition, it's rather important to understand the evolution history of the I/O library, even if your first reaction is "don't bother me with history, just show me how to use it!" The problem is that without the historical perspective you will rapidly become confused with some of the classes and when you should and shouldn't use them.

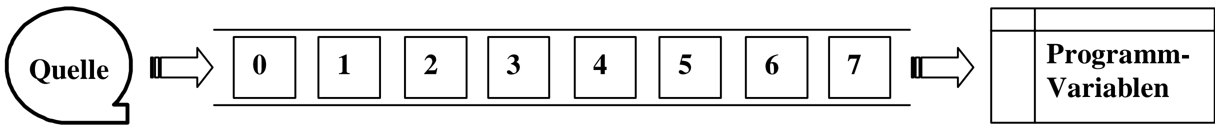
Wie schon im bisherigen Verlauf des Kurses kann auch im aktuellen Abschnitt nur ein erster Eindruck, bestenfalls ein Überblick vermittelt werden. Bei konkreten Aufgaben können Sie z.B. die SDK-Dokumentation konsultieren, die umfassende Informationen über die IO-Klassen bietet, z.B. vollständige Listen aller Methoden.

10.1 Grundprinzipien

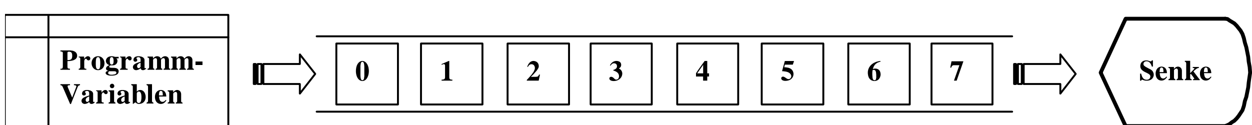
10.1.1 Datenströme

In Java wird die sequentielle Datenein- und –ausgabe über so genannte **Streams** abgewickelt:

Ein Programm **liest** Daten aus einem **Eingabestrom**, der aus einer Datenquelle (z.B. Datei, Eingabegerät, Netzwerkverbindung) gespeist wird:



Ein Programm **schreibt** Daten in einen **Ausgabestrom**, der die Werte von Programmvariablen zu einer Datensenke befördert (z.B. Datei, Ausgabegerät, Netzverbindung):



Ein- bzw. Ausgabeströme werden in Java-Programmen durch Objekte aus geeigneten Klassen des Paketes **java.io** repräsentiert, wobei die Auswahl u.a. von der angeschlossenen Datenquelle bzw. –senke sowie vom Typ der zu transportierenden Daten abhängt.

Datenquellen bzw. –senken können sich innerhalb und außerhalb eines Java-Programms befinden:

- *interne* Quellen bzw. Senken:
  - **byte**-Arrays
  - **char**-Arrays
  - **Strings**
  - Pipes
    Sie dienen der Kommunikation zwischen Threads (siehe unten).
- *externe* Quellen bzw. Senken:
  - Dateien
  - Geräte
  - Netzverbindungen

Ziel des Java-Datenstromkonzeptes ist es, Ein- und Ausgaben möglichst unabhängig von den Besonderheiten konkreter Datenquellen und –senken formulieren zu können.

Nach so vielen allgemeinen bzw. abstrakten Bemerkungen wird es Zeit für ein konkretes Beispiel. Das folgende Programm schreibt zunächst einen **byte**-Array in eine Datei und liest die Daten anschließend wieder zurück:

Quellcode

```
import java.io.*;

class StreamDemo {
  public static void main(String[] ars) {
    String name = "demo.txt";
    byte[] arr = {0,1,2,3,4,5,6,7};

    // byte-Array in den Ausgabestrom schreiben
    try {
      FileOutputStream fos = new FileOutputStream(name);
      fos.write(arr);
      fos.close();
    } catch (Exception e) {  //Ausnahmebehandlung
    }

    // Eingabestrom öffnen und byteweise in arr einlesen
    try {
      FileInputStream fis = new FileInputStream(name);
      fis.read(arr);
      fis.close();
    } catch (Exception e) {  //Ausnahmebehandlung
    }

    for (int i = 0; i < arr.length; i++)
      System.out.println(arr[i]);
  }
}
```

Ausgabe

```
0
1
2
3
4
5
6
7
```

Es benutzt dazu einen Aus- und einen Eingabestrom, die durch Objekte aus geeigneten Klassen des Paketes **java.io** realisiert werden (**FileOutputStream** und **FileInputStream**).

10.1.2 Taxonomie der Stromverarbeitungsklassen

Das Paket **java.io** enthält vier abstrakte Basisklassen, von denen alle für uns relevanten Stromverarbeitungsklassen abstammen:

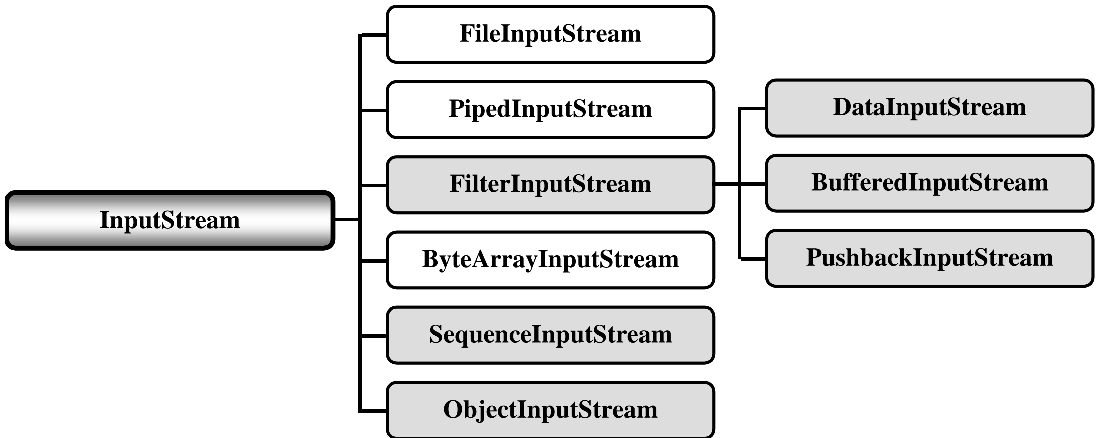
- **InputStream** und **OutputStream**
  Die Klassen aus den zugehörigen Hierarchien verarbeiten Ströme mit Bytes[1] als Elementen, wobei jedoch keine Beschränkung auf den Datentyp **byte** besteht. Andere Datentypen werden bei der Ausgabe in entsprechende **byte**-Sequenzen zerlegt und bei der Eingabe aus diesen zusammen gesetzt.
- **Reader** und **Writer**
  Die Klassen aus den zugehörigen Hierarchien verarbeiten **Zeichen**- bzw. **Character-Ströme** mit **Unicode-Zeichen** als Elementen.

Während die byteorientierten Klassen schon zur ersten Java-Generation gehörten, wurden die zeichenorientierten Klassen erst mit der Version 1.1 eingeführt, um Probleme mit der Internationalisierung von Java-Programmen zu lösen. Wo alte und neue Lösungen zur Verarbeitung von Textdaten konkurrieren, sollten die zeichenorientierten Klassen den Vorzug erhalten.

Bei den Abkömmlingen der vier abstrakten Basisklassen sind nach der Funktion zu unterscheiden:

- **Ein- bzw. Ausgabeklassen**
  Sie haben **direktem Kontakt zu Datenquellen bzw. –senken**.
- **Eingabe- bzw. Ausgabetransformationsklassen**
  Sie dienen zum **Transformieren bzw. Filtern** von Eingabe- bzw. Ausgabeströmen.

Den Hierarchien zu den vier Basisklassen werden später eigene Abschnitte gewidmet. Vorab werfen wir schon mal einen Blick auf die **InputStream**–Familie. In der folgenden Abbildung sind die Eingabeklassen mit einen weißen, und die Eingabetransformationsklassen mit einem grauen Hintergrund dargestellt:



Die abgeleiteten Klassen vervollständigen das Protokoll von **InputStream**, d.h. sie implementieren die dort teilweise abstrakten Methoden. Folglich bieten sie über *das selbe Protokoll* Zugang zu verschiedenen Datenquellen.

10.1.3 Zum guten Schluss

Allen Java-**IO**-Stromklassen gemeinsam ist die Methode **close()**, welche einen Strom schließt und alle damit assoziierten Ressourcen frei gibt. Ein explizites **close()** ist in manchen Fällen überflüssig,

[1] Wenn in Abschnitt 10 der Namensteil *Byte* auftaucht, ist keine Java-Wrapper-Klasse gemeint, sondern eine 8 Bit umfassende Informationseinheit der Datenverarbeitung.

weil die vom garbage collector oder beim Beenden des Programms ausgeführte Methode **finalize()** einen **close()**-Aufruf enthält (z.B. bei der Klasse **FileInputStream**). Es gibt jedoch viele Gründe, **close()** explizit aufzurufen:

- Viele Stromklassen überschreiben die von **java.lang.Object** geerbte **finalize()**-Methode *nicht*. Weil das Erbstück einen leeren Anweisungsblock besitzt, findet insbesondere *kein* automatisches Schließen statt.
- In den Java-Versionen 1.0 und 1.1 kann man sich nicht darauf verlassen, dass **finalize()** auf jeden Fall ausgeführt wird.
- Viele Ausgabestrom-Klassen setzen Puffer ein, die unbedingt vor dem Entfernen der Objekte geleert werden müssen, z.B. durch ein explizites **close()**.
- Viele Methoden zur Dateiverwaltung (z.B. das Umbenennen) sind bei geöffneten Dateien nicht anwendbar.
- Nicht mehr benötigte Ströme sollten so früh wie möglich geschlossen werden, um die assoziierten Ressourcen für andere Prozesse frei zu geben.

Um allen Problemen aus dem Weg zu gehen, schließt man am besten jeden Strom so früh wie möglich.

10.1.4 Aufbau und Verwendung der Transformationsklassen

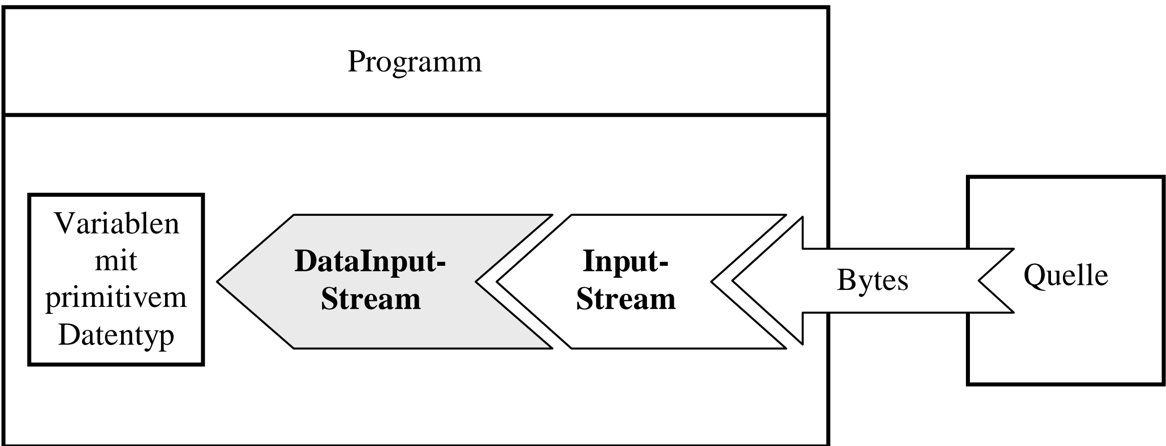
Eine Transformationsklasse baut auf einer Ein- bzw. Ausgabeklasse auf und stellt Methoden für eine erweiterte Funktionalität zur Verfügung. Wie diese Zusammenarbeit organisiert wird, betrachten wir am Beispiel der byteorientierten Eingabe-Transformationsklasse **DataInputStream**.

Diese Klasse besitzt …

- eine Instanzvariable vom Typ **InputStream**
- in ihrem Konstruktor einen Parameter vom Typ **InputStream**, dessen Aktualwert der **InputStream**-Instanzvariablen zugewiesen wird

Folglich kann beim Erstellen eines **DataInputStream**-Objektes die Referenz auf ein Objekt aus einer beliebigen Klasse, die von **InputStream** abstammt, übergeben werden.

Die Transformationsleistung eines **DataInputStream**–Objektes besteht darin, Werte primitiver Datentypen aus einer **byte**-Sequenz passender Länge zusammen zu setzen. Der Filter nimmt also Bytes entgegen und liefert z.B. **int**-Werte (bestehend aus 4 Bytes) aus. Aus dem Byte-Strom wird ein Strom mit Elementen vom gewünschten primitiven Typ:



Wird für ein **DataInputStream**–Objekt die **close()**-Methode aufgerufen, dann leitet es diese Botschaft an das verbundene **InputStream**-Objekt weiter.

Im folgenden Beispielprogramm kooperiert ein **DataInputStream** mit der byteorientierten Eingabeklasse **FileInputStream**, um **int**-Werte aus einer Datei zu lesen.
Zuvor werden diese **int**-Werte in die selbe Datei *geschrieben*, wobei das Gespann aus der Ausgabe-Transformationsklasse **DataOutputStream** und der Ausgabeklasse **FileOutputStream** zum Einsatz kommt. Hier zerlegt der Filter die **int**-Werte in einzelne Bytes und schiebt sie in den Ausgabestrom.

Quellcode

```
import java.io.*;

class DataInOutputStream {
  public static void main(String[] egal) {
    String name = "demo.txt";
    int[] arr = {1024,2048,4096,8192};
    int i;

    // int-Array in den Ausgabestrom schreiben
    try {
      FileOutputStream fos = new FileOutputStream(name);
      DataOutputStream dos = new DataOutputStream(fos);
      for (i=0; i<arr.length; i++)
        dos.writeInt(arr[i]);
      dos.close();
    } catch (Exception e) {//Ausnahmebehandlung
    }

    // Eingabestrom öffnen und int-Werte einlesen
    try {
      FileInputStream fis = new FileInputStream(name);
      DataInputStream dis = new DataInputStream(fis);
      for (i=0; i<arr.length; i++)
        arr[i] = dis.readInt();
      dis.close();
    } catch (Exception e) {//Ausnahmebehandlung
    }

    for (i=0; i < arr.length; i++)
      System.out.println(arr[i]);
  }
}
```

Ausgabe

```
1024
2048
4096
8192
```

Am Beispiel **DataInputStream** sollen noch einmal wichtige Merkmale einer Transformations- bzw. Filterklasse zusammengefasst werden:

- Die Klasse **DataInputStream** besitzt eine Instanzvariable vom Typ **InputStream**, über die der Kontakt zu einer Datenquelle hergestellt wird. Diese wird im Konstruktor initialisiert.
- Die **DataInputStream**-Eingabemethoden beauftragen den eingebundenen **InputStream** mit dem Beschaffen von Bytes in hinreichender Menge. Dann werden Bytes aus dem Eingabestrom entnommen und zu Werten eines primitiven Datentyps zusammen gesetzt.
- **DataInputStream**-Objekte können mit jedem **InputStream**-Objekt kooperieren. Bei Lesen von primitiven Datenwerten aus einer Datei kann man die Eingabeklasse **FileInputStream** verwenden.
- Ein Aufruf der **DataInputStream**-Methode **close()** wird an das verbundene **InputStream**-Objekt durchgereicht.

10.2 Verwaltung von Dateien und Verzeichnissen

Der Umgang mit Dateien und Verzeichnissen (z.B. Erstellen, auf Existenz prüfen, Löschen, Attribute lesen und setzen) wird in Java durch die Klasse **File** aus dem Paket **java.io** unterstützt. Viele wichtige Methoden dieser Klasse werden im weiteren Verlauf dieses Abschnitts anhand von Codefragmenten vorgestellt, die aus folgendem Programm entnommen wurden:

```
import java.io.*;

class FileDemo {
  public static void main(String[] egal) {
    byte[] arr = {1, 2, 3};
  . . .
  . . .
  }
}
```

10.2.1 Verzeichnis anlegen

Zunächst legen wir im aktuellen Verzeichnis den Unterordner **AusDir** an:

```
String dname = "AusDir";
File dir = new File(dname);
if (dir.exists())
  if (!dir.isDirectory()) {
    System.out.println(dname+" existiert, ist aber kein Verzeichnis.");
    System.exit(1);
  }
else
  if (dir.mkdir())
    System.out.println("Verzeichnis "+dname+" erstellt");
  else {
    System.out.println("Verzeichnis "+dname+" konnte nicht erstellt werden.");
    System.exit(1);
  }
```

Im **File**-Konstruktor kann ein absoluter (z.B. **U:/Java/FileDemo/AusDir**) oder relativer Pfad (siehe Beispiel) angegeben werden. Weil der Rückwärts-Trennstrich in Java eine Escape-Sequenz einleitet, muss unter Windows zwischen Verzeichnisnamen entweder der gewöhnliche Trennstrich oder ein verdoppelter Rückwärts-Trennstrich gesetzt werden (z.B. **U:\\Java\\FileDemo\\AusDir**).

Mit der **File**-Methode **exists()** lässt sich die Existenz eines Ordners oder einer Datei überprüfen. Ihr boolscher Rückgabewert ist genau dann **true**, wenn die Suche erfolgreich war.
Ob es sich bei einem Verzeichniseintrag um ein Unterverzeichnis handelt, stellt man mit der Methode **isDirectory()** fest.

Um ein neues Verzeichnis anzulegen, verwendet man die Methode **mkdir()**. Sollen dabei ggf. auch erforderliche Zwischenstufen automatisch angelegt werden, ist die Methode **mkdirs()** zu verwenden.

10.2.2 Dateien explizit erstellen

Zwar wird z.B. beim Erzeugen eines **FileOutputStream**-Objekts eine verknüpfte Datei bei Bedarf automatisch erstellt, doch ergeben sich auch Anlässe, eine Datei explizit anzulegen, wozu die Methode **File.createNewFile()** bereit steht:

```
String name = dname+"/Ausgabe.txt";
File f = new File(name);
if (!f.exists()) {
   try {
      f.createNewFile();
      System.out.println("Datei "+name+" erstellt");
   } catch (Exception e) {
      System.out.println("Fehler beim Erstellen der Datei "+name);
      System.exit(1);
   }
}
```

10.2.3 Informationen über Dateien und Ordner

Neben **isDirectory**() kennen **File**-Objekte noch weitere Informations-Methoden, z.B.:

- **String getAbsolutePath**()
  Absoluten Pfadnamen feststellen
- **long length**()
  Dateigröße in Bytes feststellen
- **boolean canWrite**()
  Prüft, ob das Programm schreibend zugreifen darf

Im Beispiel werden die Anfragen an ein **File**-Objekt gerichtet, das eine Datei repräsentiert:

```
System.out.println("\nEigenschaften der Datei "+name);
System.out.println("  Vollst. Pfad:       " + f.getAbsolutePath());
System.out.println("  Groesse in Bytes:   " + f.length());
System.out.println("  Schreiben moeglich: " + f.canWrite()+"\n");
```

Ausgabe:

```
Eigenschaften der Datei AusDir/Ausgabe.txt
  Vollst. Pfad:       U:\Java\FileDemo\AusDir\Ausgabe.txt
  Groesse in Bytes:   3
  Schreiben moeglich: true
```

10.2.4 Verzeichnisinhalte auflisten

Im folgenden Codefragment wird ein namenloses **File**-Objekt mit der Botschaft **listFiles()** beauftragt, für jeden Eintrag im aktuellen Verzeichnis ein Element im **File**-Array `names` anzulegen:

```
File names[] = new File(".").listFiles();
System.out.println("Dateien im akt. Verzeichnis:");
for (int i=0; i < names.length; i++)
   System.out.println(" "+names[i].getName());

names = new File(".").listFiles(new FileFilter("java"));
System.out.println("\nDateien im akt. Verzeichnis mit Extension .java:");
for (int i=0; i < names.length; i++)
   System.out.println(" "+names[i].getName());
```

Anschließend werden die Datei- oder Verzeichnisnamen mit Hilfe der **File**-Methode **getName()** ausgegeben:

```
Dateien im akt. Verzeichnis:
 FileDemo.class
 FileDemo.java
 FileDemo.jcp
 FileDemo.jcw
 src_filedemo.txt
 FileFilter.class
```

```
 FileFilter.java
 AusDir

Dateien im akt. Verzeichnis mit Extension .java:
 FileDemo.java
 FileFilter.java
```

Eine alternative **listFiles**()-Überladung liefert sogar eine *gefilterte* Liste mit **File**-Verzeichniseinträgen, benötigt dazu aber ein Objekt aus einer Klasse, die das Interface **java.io.FilenameFilter** implementiert. Im Beispiel wird dazu die Klasse `FileFilter` definiert:

```
class FileFilter implements FilenameFilter {
  private String ext;

  public FileFilter(String ext_) {
     ext = ext_;
   }

  public boolean accept(File dir, String name) {
     return name.toLowerCase().endsWith("."+ext);
  }
}
```

Um den **FilenameFilter**–Interface-Vertrag zu erfüllen, muss `FileFilter` die Methode **accept()** überschreiben.

10.2.5 Umbenennen

Mit der **File**-Methode **renameTo()** lässt sich eine Datei oder ein Verzeichnis umbenennen, wobei als Parameter ein **File**-Objekt mit dem neuen Namen zu übergeben ist:

```
File fn = new File(dname+"/Rausgabe.txt");
if (f.renameTo(fn))
   System.out.println("\nDatei "+f.getName()+" umbenannt in "+fn.getName());
else {
   System.out.println("Fehler beim Umbenennen der Datei "+f.getName());
   fn = f;
}
```

Anschließend ist die Datei bzw. der Ordner unter dem *neuen* Namen anzusprechen.
Beim Umbenennen wie beim anschließend zu beschreibenden Löschen einer Datei darf diese mit keinem offenen Ein- oder Ausgabestrom-Objekt verbunden sein.

10.2.6 Löschen

Mit der **File**-Methode **delete()** löscht man eine Datei bzw. einen Ordner:

```
if (fn.delete())
   System.out.println("Datei "+fn.getName()+" geloescht");
else {
   System.out.println("Fehler beim Loeschen der Datei "+fn.getName());
   System.exit(1);
}

if (dir.delete())
   System.out.println("Verzeichnis "+dir.getName()+" geloescht");
else {
   System.out.println("Fehler beim Loeschen des Ordners "+dir.getName());
   System.exit(1);
}
```

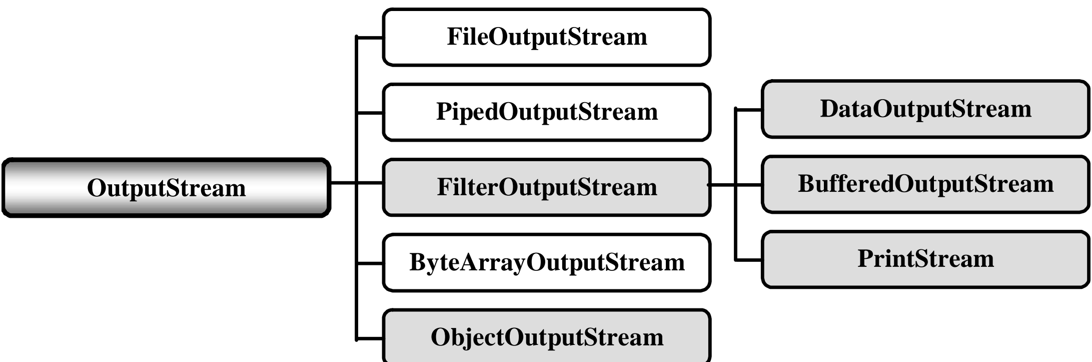
10.3 Klassen zur Verarbeitung von Byte-Strömen

In Java 1.0 stammten *alle* Ein-/Ausgabeklassen von **InputStream** oder **OutputStream** ab. Während sich die Lösungen zur Ein- und Ausgabe von primitiven Datenwerten und Objekten bewährten, machte die Behandlung von Unicode-Zeichen vor allem beim Internationalisieren von Java-Software Probleme. Mit Java 1.1 wurden daher zur Verarbeitung von Textdaten die neuen Basisklassen **Reader** und **Writer** mit ihren Klassenhierarchien eingeführt. Für andere Ein-/Ausgabeprobleme sind aber nach wie vor die byteorientierten Klassen adäquat. An einigen Stellen (z.B. bei der Standardausgabe) haben außerdem die alten Lösungen zur Zeichenverarbeitung überlebt.

10.3.1 Die OutputStream-Hierarchie

10.3.1.1 Überblick

In der folgenden Abbildung sehen Sie den für uns relevanten Teile der Klassenhierarchie zur Basisklasse **OututStream**, wobei die Ausgabeklassen (in direktem Kontakt mit Datensenken) mit einem weißen Hintergrund und die Ausgabe*transformations*klassen mit einem grauen Hintergrund dargestellt sind:

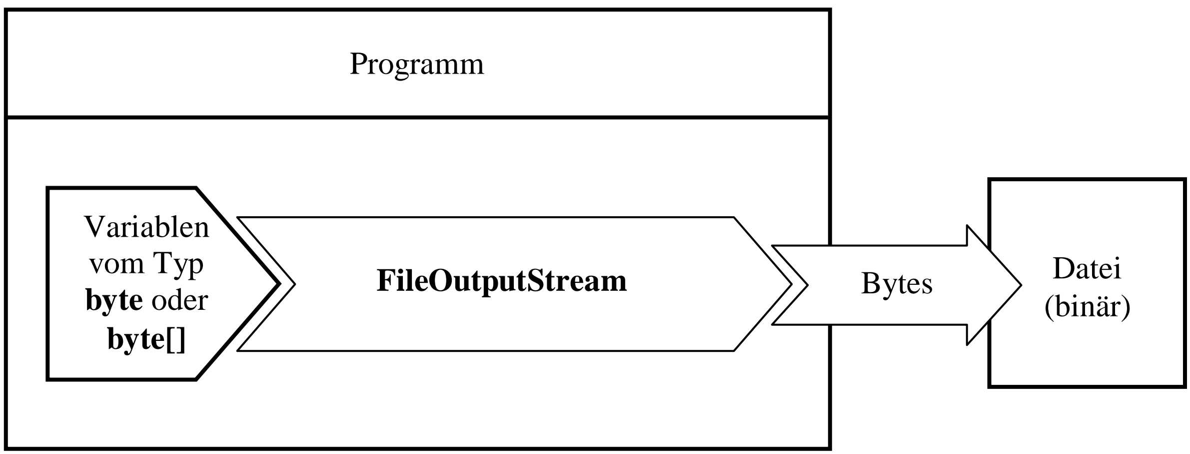


Mit der Transformationsklasse **ObjektOutputStream** können komplette Objekte in einen byteorientierten Ausgabestrom geschrieben werden. Sie wird zusammen mit ihrem Gegenstück **ObjektInputStream** in Abschnitt 10.5 behandelt.
Die folgenden **OutputStream**-Unterklassen werden in diesem Manuskript *nicht* näher beschrieben:

- **PipedOutputStream**
  Objekte dieser Klasse schreiben Bytes in eine Pipe, die zur Kommunikation zwischen Threads dient.
- **ByteArrayOutputStream**
  Objekte dieser Klasse schreiben Bytes in einen **byte**-Array, also in eine interne Datensenke.

10.3.1.2 FileOutputStream

Ein **FileOutputStream**-Objekt ist mit einer Datei verbunden und befördert auftragsgemäß die Inhalte von **byte**–Variablen oder –Arrays dorthin:



Im **FileOutputStream**-Konstruktor wird die anzusprechende Datei über ein **File**-Objekt (siehe Abschnitt 10.2) oder über einen **String** spezifiziert.
Existiert eine Datei bereits, wird sie von den **FileOutputStream**-Methoden gnadenlos überschrieben. Sollen die geplanten Ausgaben an den vorhandenen Inhalt der Ausgabedatei *angehängt* werden, ist im Konstruktor ein zweiter Parameter mit dem Wert **true** anzugeben.

Constructor Summary
`FileOutputStream(File file)` Creates a file output stream to write to the file represented by the specified `File` object.
`FileOutputStream(File file, boolean append)` Creates a file output stream to write to the file represented by the specified `File` object.
`FileOutputStream(String name)` Creates an output file stream to write to the file with the specified name.
`FileOutputStream(String name, boolean append)` Creates an output file stream to write to the file with the specified `name`.

Obwohl wir die Klasse **FileOutputStream** in den Abschnitten 10.1.1 und 10.1.4 bereits eingesetzt haben, folgt noch ein kleines Beispielprogramm zum *Kopieren* von Dateien.
Dabei wird mit der **FileInputStream**–Methode **read()** aus der Quelldatei jeweils ein Byte gelesen, das anschließend von der **FileOutputStream**–Methode **write()** in die Zieldatei befördert wird:

```
import java.io.*;
class FileCopy {
    public static void main(String[] egal) {
        int ab = 0;
        try {
            FileInputStream  fis = new FileInputStream("quelle.txt");
            FileOutputStream fos = new FileOutputStream("ziel.txt");
            int buffer = fis.read();
            while (buffer != -1) {
             ab++;
             fos.write(buffer);
             buffer = fis.read();
            }
            fis.close();
            fos.close();
        } catch (IOException e) {
            System.out.println(e);
            System.exit(1);
        }
        System.out.println("Es wurden "+ab+" Bytes kopiert.");
    }
}
```

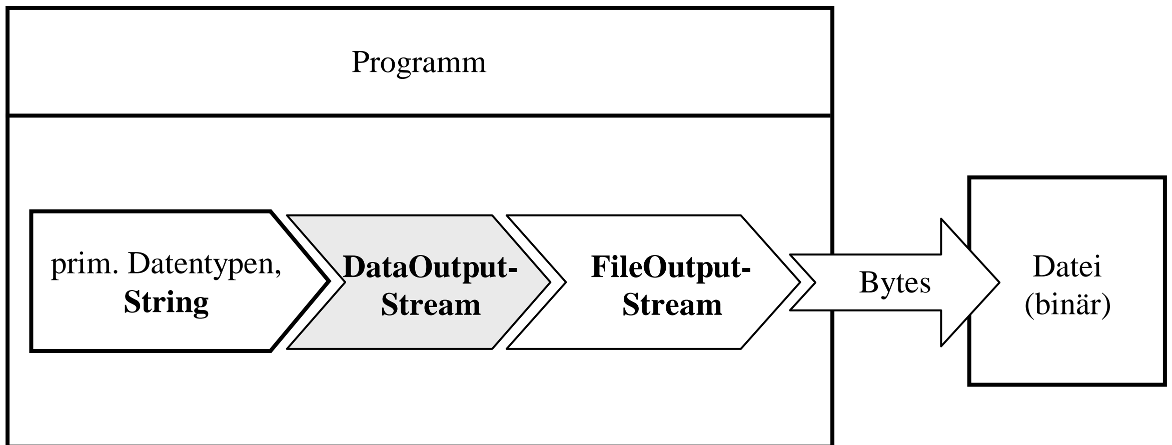
Während die Ausgabedatei nötigenfalls neu angelegt wird, führt eine fehlende Eingabedatei zur folgenden **IOException**:

```
java.io.FileNotFoundException: quelle.txt (Das System kann die
angegebene Datei nicht finden)
```

Beim Rückgabewert der Methode **FileInputStream.read()** bzw. beim Parameter der Methode **FileOutputStream.write()** hätte man eigentlich den Datentyp **byte** (statt **int**) erwartet. Es wird aber tatsächlich nur ein Byte gelesen bzw. geschrieben (vgl. API-Dokumentation).

10.3.1.3 DataOutputStream

Mit einem Objekt aus der Transformationsklasse **DataOutputStream** lassen sich die Werte primitiver Datentypen über einen **OutputStream** so in eine Datensenke befördern, dass sie mit dem komplementären Gespann aus einem **InputStream** und einem **DataInputStream** wieder zurück geholt werden können.



Neben primitiven Datentypen kann ein **DataOutputStream** auch **String**-Werte ausgeben, wobei u.a. die Platz sparende UTF8-Kodierung zur Verfügung steht (siehe Abschnitt 10.4.1.2). Nähere Erläuterungen zu den folgenden Schreibmethoden finden Sie in der SDK-Dokumentation:

void	**write**(byte[] b, int off, int len)
void	**write**(int b)
void	**writeBoolean**(boolean v)
void	**writeByte**(int v)
void	**writeBytes**(String s)
void	**writeChar**(int v)
void	**writeChars**(String s)
void	**writeDouble**(double v)
void	**writeFloat**(float v)
void	**writeInt**(int v)
void	**writeLong**(long v)
void	**writeShort**(int v)
void	**writeUTF**(String str)

Obwohl schon in Abschnitt 10.1.4 ein **DataOutputStream** zu sehen war, gönnen wir uns noch ein weiteres Beispiel. Es schreibt Werte vom Typ **int**, **double** und **String** in eine Datei:

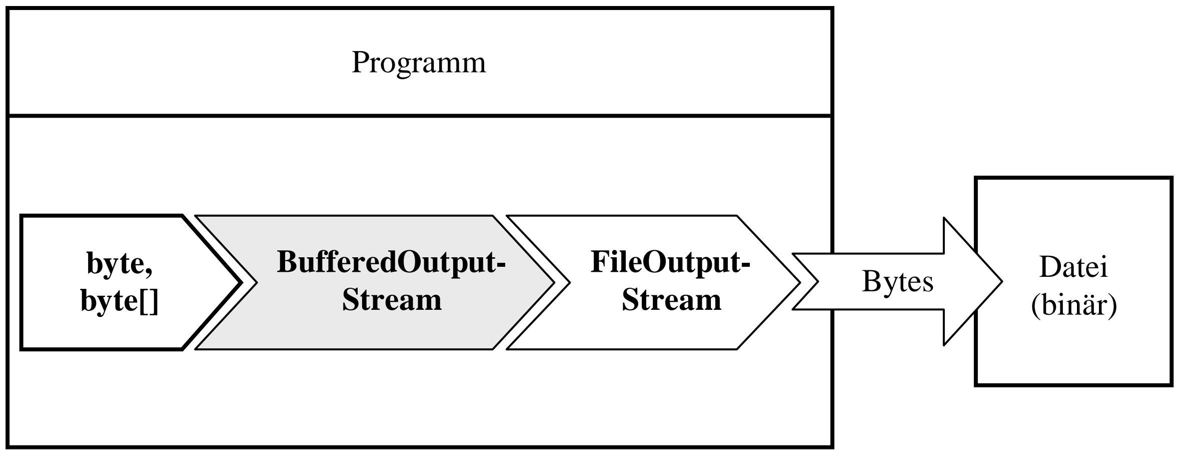
```
import java.io.*;
class DataOutputStreamDemo {
    public static void main(String[] egal) {
        DataOutputStream dos;
        DataInputStream dis;

        // int, double und String schreiben
        try {
            dos = new DataOutputStream(
                    new FileOutputStream("demo.dat"));
            dos.writeInt(4711);
            dos.writeDouble(Math.PI);
            dos.writeUTF("DataOutputStream-Demo");
            dos.close();
        } catch (Exception e) {//Ausnahmebehandlung
        }

        // int, double und String lesen
        try {
            dis = new DataInputStream(
                    new FileInputStream("demo.dat"));
            System.out.println("readInt()-Ergebnis:\t"+dis.readInt()+
                                  "\nreadDouble()-Ergebnis:\t"+dis.readDouble()+
                                  "\nreadUTF()-Ergebnis:\t"+dis.readUTF());
            dis.close();
        } catch (Exception e) {//Ausnahmebehandlung
        }
    }
}
```

Ein **DataInputStream**-Objekt holt die Werte zurück:

```
readInt()-Ergebnis:      4711
readDouble()-Ergebnis:   3.141592653589793
readUTF()-Ergebnis:      DataOutputStream-Demo
```

10.3.1.4 BufferedOutputStream

Zur Beschleunigung von Ein- oder Ausgaben setzt man oft Transformationsklassen ein, die durch Paketierung und Zwischenspeichern die Anzahl der (meist langsamen) Zugriffe auf Datenquellen oder –senken reduzieren. Diese kooperieren mit Ein- bzw. Ausgabeklassen, die direkten Kontakt mit Datenquellen bzw. –senken haben.
Ein **BufferedOutputStream**–Objekt nimmt **byte**-Variablen oder –Arrays entgegen und leitet diese in geeigneten Portionen an ein **OutputStream**-basiertes Ausgabeobjekt weiter (z.B. an ein **File-OutputStream**-Objekt):



Im **BufferedOutputStream**–Konstruktor ist obligatorisch ein **OutputStream**-Objekt zu übergeben (vgl. Abschnitt 10.1.4). Optional kann die voreingestellte Puffergröße von 512 Bytes geändert werden. Über diese und viele weitere Details informiert die SDK-Dokumentation:

Constructor Summary
`BufferedOutputStream(OutputStream out)` Creates a new buffered output stream to write data to the specified underlying output stream with a default 512-byte buffer size.
`BufferedOutputStream(OutputStream out, int size)` Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size.

Das folgende Beispielprogramm schreibt 50 KB Daten in eine Datei, zunächst ungepuffert, dann unter Verwendung eines **BufferedOutputStreams**:

```
import java.io.*;
class BufferedOutputStreamDemo {
  public static void main(String[] ars) {
    long time1 = System.currentTimeMillis();
    try {
      FileOutputStream fos = new FileOutputStream("noBuffer.dat");
      for (int i = 1; i <= 51200; i++)
          fos.write(i);
      fos.close();
    } catch (Exception e) {  }
    long time2 = System.currentTimeMillis();
    System.out.println("Zeit fuer die ungepufferte Ausgabe: " + (time2-time1));

    try {
      FileOutputStream fos = new FileOutputStream("Buffer.dat");
      BufferedOutputStream bos = new BufferedOutputStream(fos, 8192);
      for (int i = 1; i <= 51200; i++)
          bos.write(i);
      bos.close();
    } catch (Exception e) {  }
    long time3 = System.currentTimeMillis();
    System.out.println("Zeit fuer die gepufferte Ausgabe:   " + (time3-time2));
  }
}
```

Durch den Einsatz des **BufferedOutputStream**-Objektes (mit Puffergröße 8192) Puffers kann der Zeitaufwand beim Schreiben erheblich reduziert werden (Angaben in Millisekunden):

```
Zeit fuer die ungepufferte Ausgabe: 351
Zeit fuer die gepufferte Ausgabe:   60
```

Wichtig: Ein **BufferedOutputStream** muss unbedingt vor seinem Ableben (z.B. am Ende des Programms) per **flush()** entleert werden, weil sonst die zwischengelagerten Bytes verfallen. Dies kann auch über die **BufferedOutputStream**–Methode **close()** geschehen, die **flush()** aufruft und anschließend den zugrunde liegenden **OutputStream** schließt.
Das obige Programm schreibt exakt 51200 Bytes in die Ausgabedateien. Entfernt man die Anweisung

```
bos.close();
```

dann werden nur 49152 (= 6 · 8192) Bytes in die Ausgabedatei **Buffer.dat** geschrieben!

Wegen des erheblichen Performanzvorteils sollten Ein-/Ausgabe - Puffer reichlich eingesetzt werden, was durch geschickte Klassendefinitionen mit geringfügigem Aufwand möglich ist.

10.3.1.5 PrintStream

Die Transformationsklasse **PrintStream** dient dazu, Werte beliebigen Typs in einer für Menschen lesbaren Form auszugeben, z.B. auf der Konsole.
Wir haben schon einige praktische Erfahrung mit dieser Klasse gesammelt, weil der per **System.out** ansprechbare Standard-Ausgabestrom ein **PrintStream**-Objekt ist. Dies gilt auch für den Standard-*Fehler*ausgabestrom, der über die Klassenvariable **System.err** ansprechbar ist. Wir haben nie direkt in die Standard-Fehlerausgabe geschrieben, doch werden im Java-API häufig Ausnahmeobjekte beauftragt, ihre Methode **printStackTrace()** auszuführen, die eine Ausgabe der Aufrufsequenz an **System.err** bewirkt.

Ein **PrintStream**-Objekt kann mit Hilfe seiner vielfach überladenen Methoden **print()** und **println()** Werte mit beliebigem Datentyp ausgeben, z.B.:

Quellcode

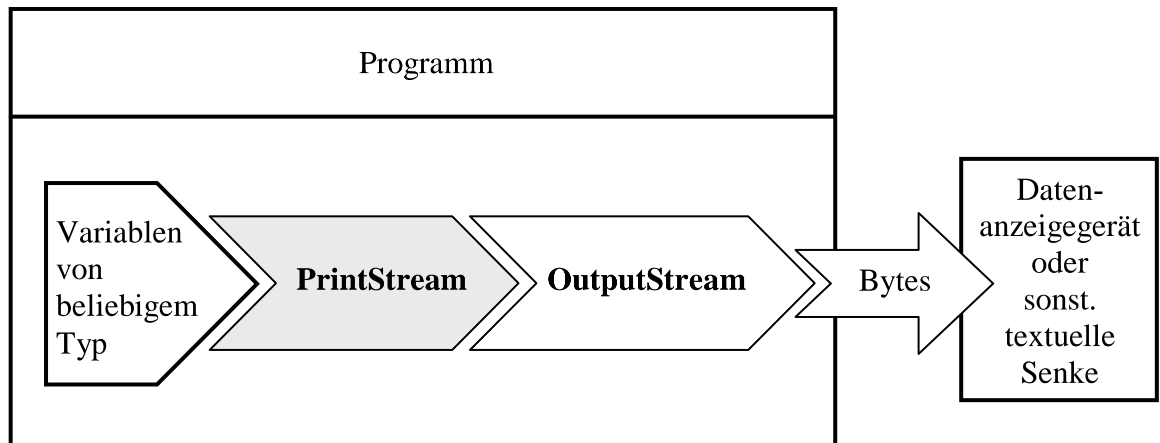
```
class PrintStreamDemo {
 public static void main(String[] args) {
    PrintStreamDemo wob = new PrintStreamDemo();
    System.out.println("Ein PrintStream kann Variablen\n"
       +"bel. Typs verarbeiten.\n"
       +"Z.B. die double-Zahl\n"+"   "+Math.PI
       +"\noder auch das Objekt\n"+"   "+wob);
 }
}
```

Ausgabe

```
Ein PrintStream kann Variablen
bel. Typs verarbeiten.
Z.B. die double-Zahl
   3.141592653589793
oder auch das Objekt
   PrintStreamDemo@cac268
```

Zur Bequemlichkeit trägt außerdem bei, dass im Unterschied zu anderen **OutputStream**-Unterklassen von den **PrintStream**-Methoden *keine* **IOException** geworfen wird. Statt dessen wird ein, von uns bisher nicht beachtetes Fehlersignal gesetzt, das mit **checkError**() abgefragt werden kann. Es wäre in der Tat recht umständlich, jeden Aufruf der Methode **System.out.println()** in einen geschützten **try**-Block zu setzen.

Hinter das per **System.out** ansprechbare **PrintStream**-Transformationsobjekt ist noch ein **FileOutputStream-Objekt** geschaltet, das mit der Konsole verbunden ist.
Generell kann man die **PrintStream**-Arbeitsweise ungefähr folgendermaßen darstellen, wobei an Stelle der abstrakten Klasse **OutputStream** eine konkrete Unterklasse stehen muss:



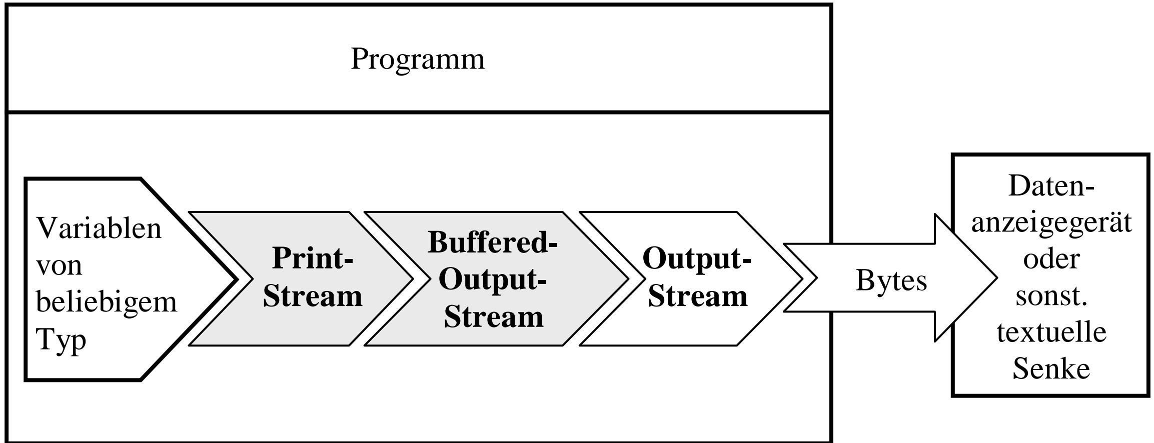
Im nächsten Beispiel ist zu sehen, wie mit Hilfe der Transformationsklasse **PrintStream** Werte primitiver Datentypen in eine *Textdatei* geschrieben werden (über einen **FileOutputStream**):

```
import java.io.*;

class PrinSt {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream("ps.txt");
        } catch (FileNotFoundException e) {//Ausnahmebehandlung
        }

        PrintStream ps = new PrintStream(fos);
        ps.println(64798 + " " + Math.PI);
        ps.close();
    }
}
```

In der Ausgabedatei **ps.txt** landen die Zeichenkettenrepräsentationen des **int**- und des **double**-Wertes:

```
64798 3.141592653589793
```

Wenn ein **PrintStream**-Objekt zwecks Geschwindigkeitsoptimierung in einen **BufferedOutputStream** schreibt, sind zwei Transformationsobjekte nacheinander geschaltet:



In dieser Situation müssen Sie einen **PrintStream**-Konstruktor mit **autoFlush**-Parameter wählen und diesen auf **true** setzen, wenn (z.B. in einer interaktiven Anwendung) der Puffer in folgenden Situationen automatisch geleert werden soll:

- nach dem Schreiben eines **byte**-Arrays
- nach Ausgabe eines Newline-Zeichens (\n)
- nach Ausführen einer **println()**-Methode

Ein gutes Beispiel für die Kombination aus **PrintStream** und **BufferedOutputStream** ist der per **System.out** ansprechbare Standard-Ausgabestrom, der analog zu folgendem Codefragment initialisiert wird:

```
FileOutputStream fdout =
    new FileOutputStream(FileDescriptor.out);
BufferedOutputStream bos =
    new BufferedOutputStream(fdout, 128);
PrintStream ps =
    new PrintStream(bos, true);

System.setOut(ps);
```

Mit der statischen Variablen **out** der Klasse **FileDescriptor** wird der Bezug zur Konsole hergestellt. Im **PrintStream**-Konstruktor wird für den **autoFlush**-Parameter der Wert **true** verwendet.

Über die System-Methode **setOut()** kann ein selbst entworfener Strom als Standardausgabe in Betrieb genommen werden. Im Beispiel ist die Standardausgabe allerdings „originalgetreu“ nachgebaut worden.

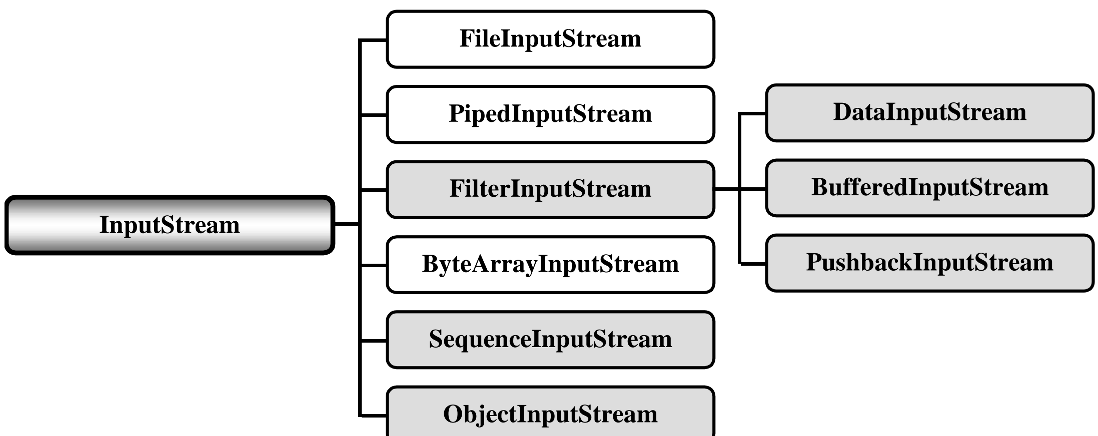
Bei der Ausgabe von *Textdaten* übersetzen **PrintStream**-Objekte den Java-internen Unicode unter Verwendung der plattformspezifischen Standardkodierung in Bytes. Welche Konsequenzen sich daraus ergeben können, zeigt die unglückliche Ausgabe von Umlauten im Java-Konsolenfenster unter Windows, z.B.:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
  public static void main(String[] args) {
    System.out.println("Ganz schön übel!");
  }
}</pre></td><td><pre>Ganz sch÷n ³bel!</pre></td></tr>
</table>

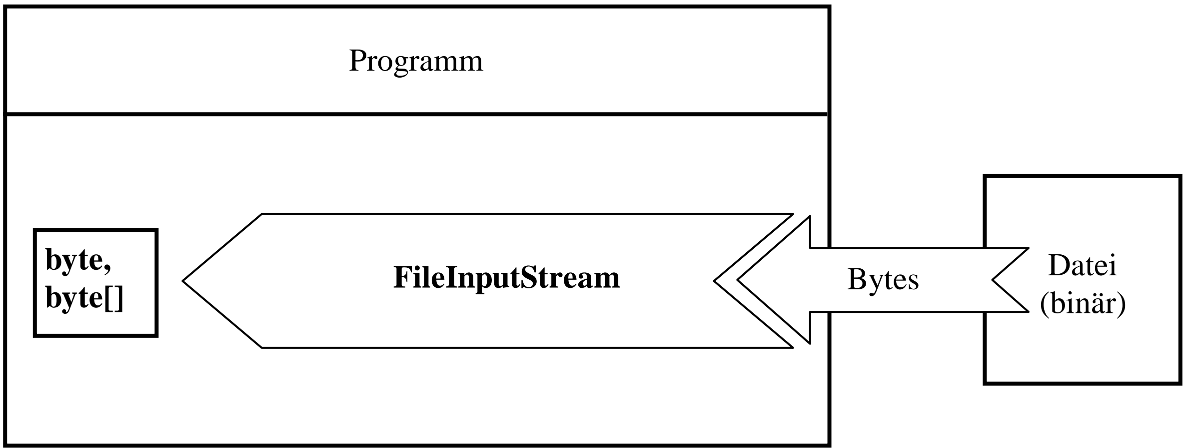
Daher ist zur Textausgabe oft die modernere und flexiblere Klasse **PrintWriter** vorzuziehen (siehe Abschnitt 10.4.1).

10.3.2 Die InputStream-Hierarchie

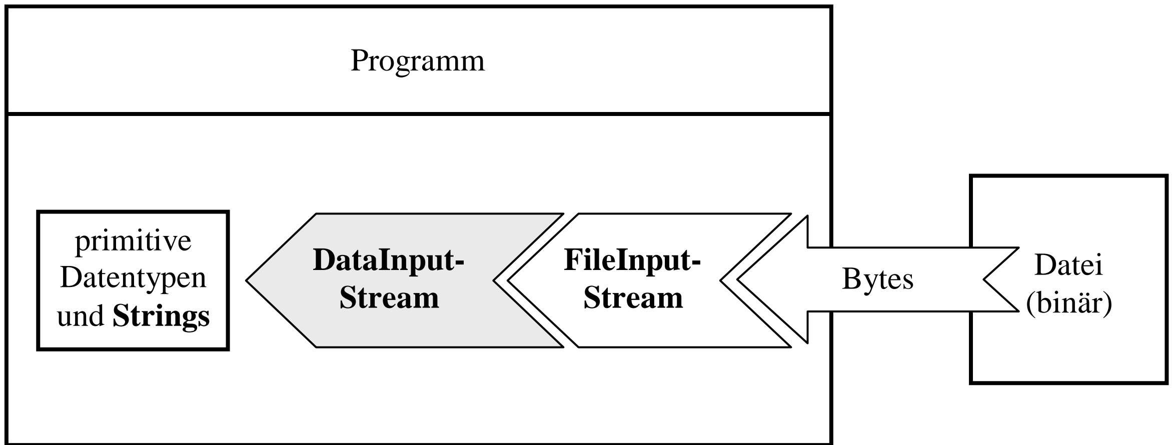
Bei den Klassen der **InputStream**-Hierarchie können wir uns im Wesentlichen mit Verweisen auf analoge Klassen der **OutputStream**-Hierarchie begnügen. Um Ihnen das Blättern zu ersparen, wird die schon in Abschnitt 10.1.2 gezeigte Abbildung zur **InputStream**-Hierarchie wiederholt (Eingabeklassen mit weißem Hintergrund und Eingabe*transformations*klassen mit grauem Hintergrund):



Wie man mit einem **FileInputStream**–Objekt Bytes aus einer Datei liest, wurde schon im Abschnitt 10.3.1.2 über den **FileOutputStream** gezeigt.



Die Transformationsklasse **DataInputStream** liest primitive Datentypen sowie **Strings** aus einem Bytestrom und ist uns zusammen mit ihrem Gegenstück **DataOutputStream** schon in Abschnitt 10.3.1.3 begegnet.



Mit der Transformationsklasse **ObjectInputStream** werden wir uns in Abschnitt 10.5 näher beschäftigen. Sie ermöglicht das Einlesen kompletter Objekte aus einem byteorientierten Eingabestrom.

Die restlichen Klassen sollen nur kurz vorgestellt werden:

- **PipedInputStream**
  Objekte dieser Klasse lesen Bytes aus einer Pipe, die zur Kommunikation zwischen Threads dient.
- **ByteArrayInputStream**
  Objekte dieser Klasse lesen Bytes aus einem **byte**-Array.
- **BufferedInputStream**
  Analog zum **BufferedOutputStream** (siehe Abschnitt 10.3.1) wird ein Zwischenspeicher eingesetzt, um das Lesen aus einem Eingabestrom zu beschleunigen.
- **PushbackInputStream**
  Diese eher selten benötigte Klasse bietet Methoden, um aus einem Eingabestrom entnommenen Bytes wieder zurück zu stellen.
- **SequenceInputStream**
  Mit Hilfe dieser Transformationsklasse kann man mehrere Objekte aus **InputStream**-Unterklassen hintereinander koppeln und als einen einzigen Strom behandeln. Ist das Ende eines Eingabestroms erreicht, wird er geschlossen, und der nächste Strom in der Sequenz wird geöffnet.

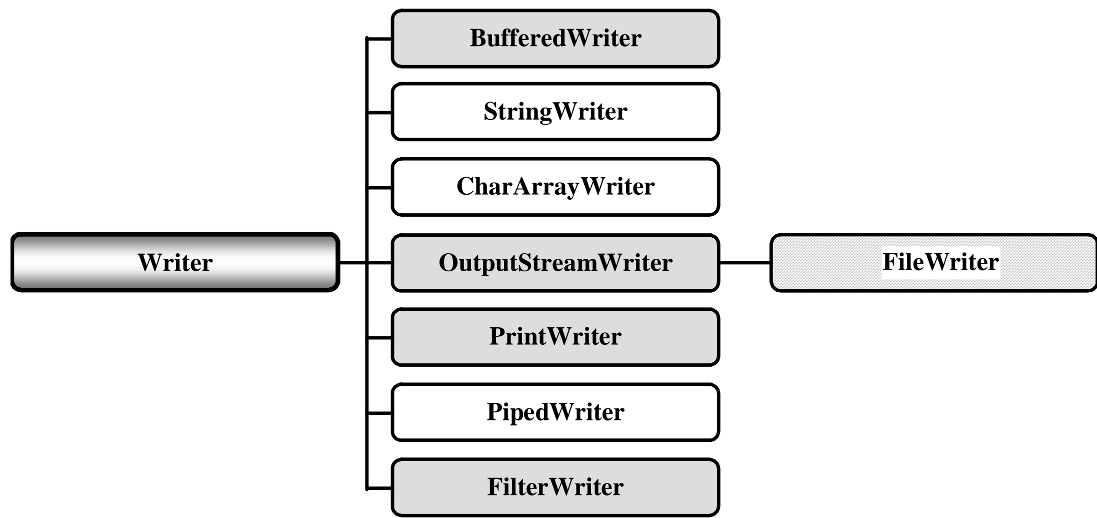
10.4 Klassen zur Verarbeitung von Character-Strömen

In diesem Abschnitt werden die mit Java 1.1 eingeführten Klassen zur Verarbeitung von Character-Strömen behandelt, die von den beiden abstrakten Basisklassen **Writer** bzw. **Reader** abstammen. Sie sind in der Regel gegenüber den älteren Byte-Strom-Klassen (z.B. wegen der unproblematischen Internationalisierung) zu bevorzugen, wenn konkurrierende Funktionalitäten vorliegen.

10.4.1 Die Writer-Hierarchie

10.4.1.1 Überblick

Alle IO-Klassen zur Verarbeitung von zeichenorientierten *Ausgabe*strömen stammen von **Writer** ab:



In der Abbildung sind Ausgabeklassen (in direktem Kontakt mit einer Datensenke) mit weißem Hintergrund dargestellt, Ausgabe*transformations*klassen mit grauem Hintergrund. Weil die Klasse **FileWriter** eine später noch zu besprechende Besonderheit aufweist, ist sie mit schraffiertem Hintergrund gezeichnet.

10.4.1.2 Brückenklassen und Encodings, OutputStreamWriter

Die Klasse **OutputStreamWriter** übersetzt als **Brückenklasse** die von Java intern verwendeten 16-bittigen Unicode-Zeichen in 8-bittige Bytes, wobei unterschiedliche Kodierungsschemata (engl. *encodings*) benutzt werden können. Ihre Besonderheit im Unterschied zu gewöhnlichen Transformationsklassen besteht darin, dass sie einen Character-Strom in einen Byte-Strom überführt.
Die folgenden Kodierungsschemata (bezeichnet nach den Vorschriften der IANA (**I**nternet **A**ssigned **N**umbers **A**uthority)) werden von *allen* Java-Implementationen unterstützt:

<table>
<tr><th>IANA-Bezeichnung</th><th>Beschreibung</th></tr>
<tr><td>US-ASCII</td><td>7-bit-ASCII-Code
Bei den Unicode-Zeichen \u0000 bis \u007F wird das niederwertige Byte ausgegeben, ansonsten ein Fragezeichen (0x3F).</td></tr>
<tr><td>ISO-8859-1</td><td>Erweiterter ASCII-Code (Latin-1)
Bei den Unicode-Zeichen \u0000 bis \u00FF wird das niederwertige Byte ausgegeben, ansonsten ein Fragezeichen (0x3F).</td></tr>
<tr><td>UTF-8</td><td>Bei diesem Schema werden die Unicode-Zeichen durch eine variable Anzahl von Bytes kodiert. So können alle Unicode-Zeichen ausgegeben werden, ohne die platzverschwenderische Anhäufung von Null-Bytes bei den ASCII-Zeichen in Kauf nehmen zu müssen:
<table>
<tr><th colspan="2">Unicode-Zeichen</th><th rowspan="2">Anzahl Bytes</th></tr>
<tr><th>von</th><th>bis</th></tr>
<tr><td>\u0000</td><td>\u0000</td><td>2</td></tr>
<tr><td>\u0001</td><td>\u007F</td><td>1</td></tr>
<tr><td>\u0080</td><td>\u07FF</td><td>2</td></tr>
<tr><td>\u0800</td><td>\uFFFF</td><td>3</td></tr>
</table></td></tr>
<tr><td>UTF-16BE</td><td>Für alle Unicode-Zeichen werden 16 Bit in Big-Endian Reihenfolge ausgegeben: Das höherwertige Byte zuerst.
In Java ist diese Reihenfolge generell voreingestellt (auch bei anderen Datentypen).
Beim großen griechischen Delta (\u0394) wird ausgegeben: 03 94</td></tr>
</table>

IANA-Bezeichnung	Beschreibung
UTF-16LE	Für alle Unicode-Zeichen werden 16 Bit in **Little-Endian** Reihenfolge ausgegeben: Das niederwertige Byte zuerst. Bei großen griechischen Delta (\u0394) wird ausgegeben: 94 03 Die Little-Endian - Reihenfolge ist auf der Intel-Plattform eigentlich Standard, was Java aber per Voreinstellung korrigiert.

Unter MS-Windows ist folgendes Kodierungsschema voreingestellt:

IANA-Bezeichnung	Beschreibung
Cp1252	Windows Latin-1 (ANSI) Im Unterschied zu ISO-8859-1 werden die Cp1252-Zeichen von 0x80 bis 0x9F (in ISO-8859-1 reserviert für <control>) mit „höheren“ Unicode-Zeichen assoziiert. So wird z.B. sinnvollerweise das Eurozeichen (Unicode: \u20AC) abgebildet auf seine ANSI-Kodierung (0x80), so dass es von Windows-Textverarbeitungsprogrammen korrekt dargestellt wird.

Auf der Webseite

http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html

werden noch zahlreiche weitere Kodierungsschemata aufgelistet.

Komplette Übersetzungstabellen zu wichtigen encodings finden sich auf der Webseite:

http://www.ingrid.org/java/i18n/encoding/mapping.html

Im folgenden Programm werden die gerade tabellierten Kodierungsschemata nacheinander dazu verwendet, einen kurzen Text mit dem Umlaut „ä“ (\u00E4) in eine Datei zu schreiben:

```
import java.io.*;

class OSW {
    public static void main(String[] args) throws IOException {
         String[] encoding = {"US-ASCII", "ISO-8859-1", "Cp1252", "UTF-8",
                              "UTF-16BE", "UTF-16LE"};
         FileOutputStream fos = new FileOutputStream("test.txt");
         OutputStreamWriter osw = null;
         PrintWriter pw = null;

         for (int i = 0; i < 6; i++) {
             osw = new OutputStreamWriter(fos, encoding[i]);
            pw = new PrintWriter(osw, true); // 2. Parameter: autoFlush
            pw.println(encoding[i] + " ae = ä");
         }
        pw.close();
    }
}
```

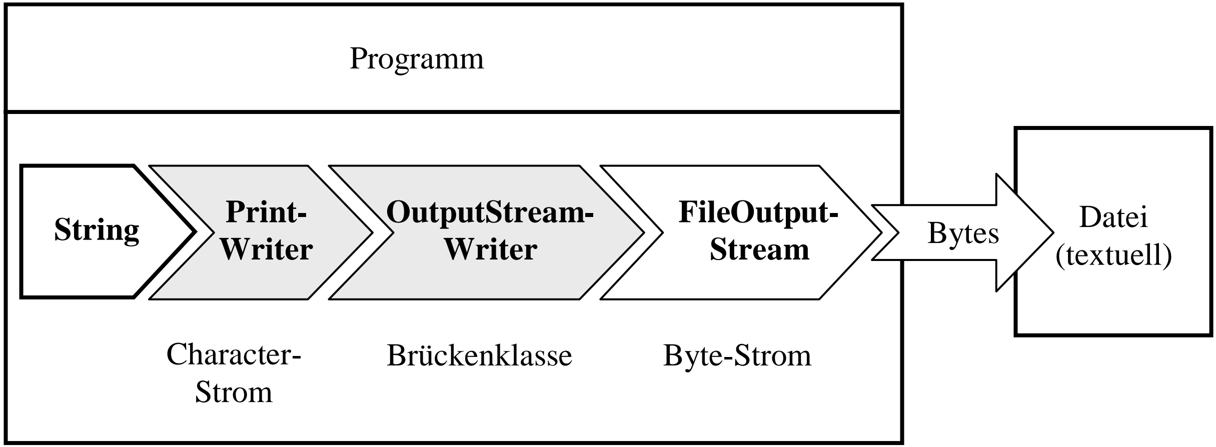
In einem Hex-Editor sieht die Ergebnisdatei folgendermaßen aus:

```
test.txt
00000000 5553 2D41 5343 4949 2061 6520 3D20 3F0D US-ASCII ae = ?.
00000010 0A49 534F 2D38 3835 392D 3120 6165 203D .ISO-8859-1 ae =
00000020 20E4 0D0A 4370 3132 3532 2061 6520 3D20  ...Cp1252 ae =
00000030 E40D 0A55 5446 2D38 2061 6520 3D20 C3A4 ...UTF-8 ae = ..
00000040 0D0A 0055 0054 0046 002D 0031 0036 0042 ...U.T.F.-.1.6.B
00000050 0045 0020 0061 0065 0020 003D 0020 00E4 .E. .a.e. .=. ..
00000060 000D 000A 5500 5400 4600 2D00 3100 3600 ....U.T.F.-.1.6.
00000070 4C00 4500 2000 6100 6500 2000 3D00 2000 L.E. .a.e. .=. .
00000080 E400 0D00 0A00                          ......
```

Für das Unicode-Zeichen \u00E4 wird jeweils ausgegeben:

Kodierungsschema	Byte(s) in der Ausgabe
US-ASCII	3F
ISO-8859-1	E4
Cp1252	E4
UTF-8	C3 A4
UTF-16BE	00 E4
UTF-16LE	E4 00

Das Beispielprogramm arbeitet mit folgender Datenstrom-Architektur:



In der Abbildung ist passend zum Beispielprogramm nur der Datentyp **String** als **PrintWriter**-Eingabe zu sehen. Allerdings bietet diese Klasse wie der „Vorgänger“ **PrintStream** Ausgabemethoden für praktisch alle Datentypen.

10.4.1.3 Implizite und explizite Pufferung, BufferedWriter

OutputStreamWriter–Objekte (und damit auch **FileWriter**-Objekte) sammeln die per Unicode-Wandlung entstandenen Bytes zunächst in einem internen Puffer. Daher muss auf jeden Fall vor dem Ableben eines solchen **Writers** (z.B. beim Programmende) der Puffer geleert werden. Dazu stehen mehrere Möglichkeiten bereit:

- direkter Aufruf der Methode **flush()**
- Aufruf der Methode **close()**, die wiederum **flush()** aufruft
- **autoflush** eines vorgeschalteten **PrintWriters** (siehe unten) per Konstruktor-Parameter aktivieren
  Im Unterschied zur Klasse **PrintStream** findet das automatische Entleeren des Puffers nur beim Aufruf einer **println()**-Methode statt.

Für die *explizite* Pufferung von *Zeichen* (statt Bytes) steht in der **Writer**-Hierarchie die Transformationsklasse **BufferedWriter** zur Verfügung, wobei die voreingestellte Puffergröße von 8192 Zeichen per Konstruktor-Parameter verändert werden kann.
Abweichend vom Aufbau der **OutputStream**-Hierarchie ist **BuffererdWriter** allerdings *nicht* von **FilterWriter** abgeleitet.
In folgendem Beispiel wird ein **BufferedWriter** zwischen einen **PrintWriter** und einen **FileWriter** gesetzt:

```
import java.io.*;

class BufferedWriterDemo {
  public static void main(String[] args) throws IOException {
    long time1 = System.currentTimeMillis();
       PrintWriter pw =  new PrintWriter(
                         new BufferedWriter(
                         new FileWriter("test.txt"), 16384));
    for (int i = 1; i <= 10000; i++)
      pw.println("Zeile " + i);
    pw.close();
    System.out.println("Benoetigte Zeit: "+(System.currentTimeMillis()-time1));

  }
}
```

Aufgrund der eingangs erwähnten impliziten Byte-Pufferung durch die **OutputStreamWriter**-Unterklasse **FileWriter** fällt der Gewinn durch den **BufferedWriter**-Einsatz nicht so gewaltig aus: Der Zeitaufwand sinkt von 94 auf 78 Millisekunden.
Weil die systematische Behandlung der Klassen **PrintWriter** und **FileWriter** noch aussteht, wird ein Diagramm zum letzten Beispiel erst am Ende von Abschnitt 10.4.1.6 präsentiert.

BufferedWriter–Objekte kommen gelegentlich auch ohne Zutun des Programmierers zum Einsatz, weil sie von anderen Klassen intern zur Pufferung verwendet werden, z.B. bei der Kopplung eines **PrintWriters** an einen **OutputStream**. In diesem Fall nimmt der **PrintWriter**-Konstruktor insgeheim einen **BufferedWriter** in Betrieb:[1]

```
public PrintWriter(OutputStream out, boolean autoFlush) {
      this(new BufferedWriter(new OutputStreamWriter(out)), autoFlush);
}
```

Damit arbeitet insgesamt folgendes Gespann:

Abschließend sollen zwei wichtige Empfehlungen für den Umgang mit **Writer**-Subklassen nochmals erwähnt werden:

- Es ist auf jeden Fall eine abschließende Pufferentleerung erforderlich.
- Bei Bedarf kann durch Einsatz eines **BufferedWriters** die Performanz gesteigert werden.

10.4.1.4 PrintWriter

Die Transformationsklasse **PrintWriter** besitzt diverse **print()**- bzw. **println()**-Überladungen, um primitive Datentypen sowie Objekte in textueller (für Menschen lesbarer Form) auszugeben (z.B. in eine Datei oder auf die Konsole). Sie wurde mit Java 1.1 als Nachfolger bzw. Ergänzung der älteren Klasse **PrintStream** eingeführt, die aber zumindest im Standard-Ausgabestrom **System.out** und im Standard-Fehlerausgabestrom **System.err** weiter lebt (vgl. Abschnitt 10.3.1.5).

Als angekoppelte Ausgabe-Basisklasse kommen **OutputStream** oder **Writer** in Frage:

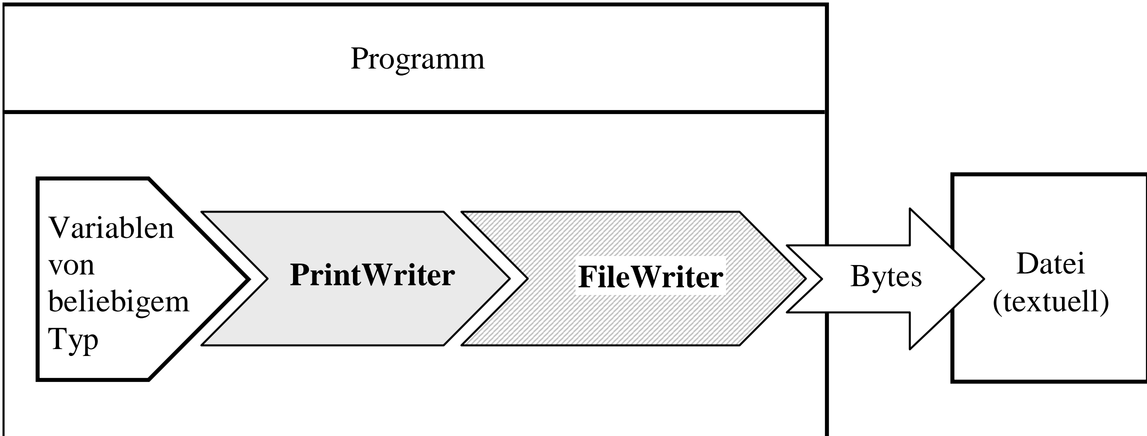
[1] Sie finden diese Definition in der Datei **PrintWriter.java**, die wiederum im Archiv **src.zip** mit den API-Quelltexten steckt. Das Quelltextarchiv kann bei der SDK-Installation auf Ihre Festplatte befördert werden.

Constructor Summary

PrintWriter(OutputStream out) Create a new PrintWriter, without automatic line flushing, from an existing OutputStream.
PrintWriter(OutputStream out, boolean autoFlush) Create a new PrintWriter from an existing OutputStream.
PrintWriter(Writer out) Create a new PrintWriter, without automatic line flushing.
PrintWriter(Writer out, boolean autoFlush) Create a new PrintWriter.

Wird ein **PrintWriter** auf einen Byte-Strom aufgesetzt (z.B. **FileOutputStream**), dann kommt automatisch ein übersetzender **OutputStreamWriter** mit voreingestelltem Kodierungsschema zum Einsatz. Alternativ kann auch ein expliziter **OutputStreamWriter** einschaltet werden (siehe Abschnitt 10.4.1.2).

Meist verwendet man eine von **Writer** abstammende Ausgabeklasse, z.B. **FileWriter**:



Vermutlich erstaunt Sie an dieser Abbildung, dass ein Character-Strom *Bytes* an eine Senke ausliefert, wo doch laut Abschnitt 10.4.1.2 ein Objekt aus der speziellen Transformationsklasse **OutputStreamWriter** benötigt wird, um einen Character-Strom in einen Byte-Strom zu überführen. Des Rätsels Lösung steckt in der speziellen Konstruktion der (von **OutputStreamWriter** abgeleiteten) Klasse **FileWriter**, die in Abschnitt 10.4.1.6 beschrieben wird.

Beim Einsatz der Klasse **PrintWriter** ist zu beachten, dass stets ein *interner Puffer* im Spiel ist (vgl. Abschnitt 10.4.1.3). Am Ende eines Programms muss also unbedingt (z.B. per **close**()-Aufruf) der Zwischenspeicher entleert werden:

```
import java.io.*;
class PrintWriterDemo {
  public static void main(String[] egal) throws IOException {
    FileOutputStream fos = new FileOutputStream("pw.txt");
    PrintWriter pw = new PrintWriter(fos);
    for (int i = 1; i < 3000; i++)
      pw.println(i);
    pw.close();
  }
}
```

Soll ein **PrintWriter**–Objekt nach jedem **println**()-Methodenaufruf seinen Puffer automatisch ausgeben (z.B. bei einer *interaktiven* Anwendung), dann muss im Konstruktor der **autoFlush**-Parameter auf **true** gesetzt werden (Beispiel: siehe Abschnitt 10.4.1.5). Wo kein **autoFlush** erforderlich ist, sollte aus Performanzgründen darauf verzichtet werden.

Wie die **PrintStream**-Methoden werfen auch die **PrintWriter**-Methoden *keine* **IOException**, sondern setzen ein Fehlersignal, das mit **checkError**() abgefragt werden kann.

Abweichend vom Aufbau der **OutputStream**-Hierarchie ist **PrintWriter** *nicht* von **FilterWriter** abgeleitet.

10.4.1.5 Umlaute in Java-Konsolenanwendungen unter Windows

Mit Hilfe der Brückenklasse **OutputStreamWriter** lässt sich endlich ein kleines Problem lösen, das uns während des ganzen Kurses begleitet hat:

- Windows arbeitet in Konsolenfenstern DOS-konform mit dem ASCII-Zeichensatz, in grafikorientierten Anwendungen hingegen mit dem ANSI-Zeichensatz.
- Die virtuelle Java-Maschine arbeitet unter Windows grundsätzlich mit dem ANSI-Zeichensatz. Infolgedessen werden Umlaute in Java-Konsolenanwendungen unter Windows falsch dargestellt.

Ein **OutputStreamWriter** -Objekt mit dem Kodierungsschema Cp850 („IBM-ASCII“, „code page 850“) ermöglicht die korrekte Darstellung der Umlaute:

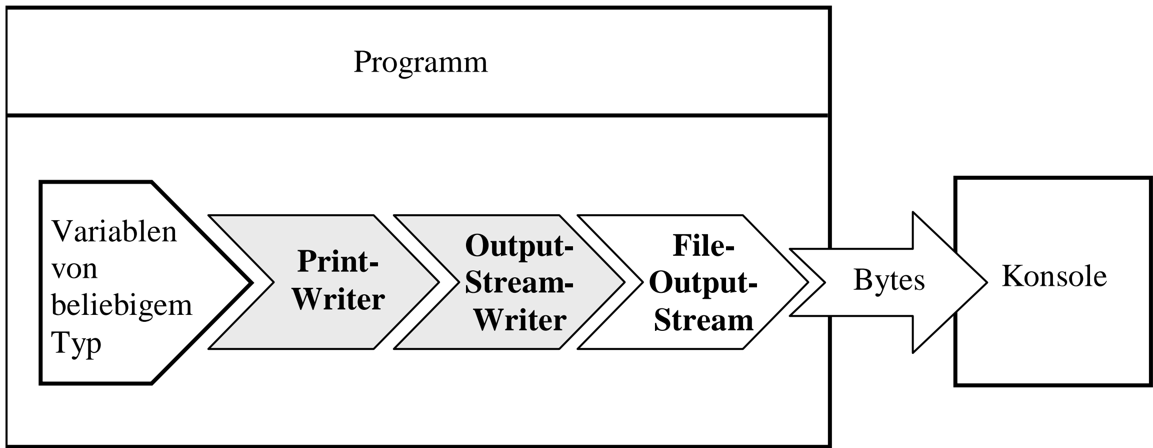
Quellcode:

```
import java.io.*;
class Prog {
 public static void main(String[] ars) throws IOException {
    FileOutputStream fos =
       new FileOutputStream(FileDescriptor.out);
    OutputStreamWriter osw = null;
    if (System.getProperty("os.arch").equals("x86"))
       osw = new OutputStreamWriter(fos, "Cp850");
    else
       osw = new OutputStreamWriter(fos);

    PrintWriter pw = new PrintWriter(osw, true);
    pw.println("War der Ärger nötig? (j/n)");
    char antwort = Simput.gchar();
    pw.println("Ihre Meinung: "+antwort);
 }
}
```

Ausgabe:

```
War der Ärger nötig? (j/n)
n
Ihre Meinung: n
```

Mit der statischen Variablen **out** der Klasse **FileDescriptor** wird der Bezug zur Konsole hergestellt. Um die Plattformunabhängigkeit nicht einzuschränken, wird das spezielle Kodierungsschema nur dann verwendet, wenn der **System**-Methodenaufruf **getProperty("os.arch")** als Betriebssystem-Architektur „x86“ zurück meldet.
Es liegt in jedem Fall die folgende Datenstrom-Architektur vor:



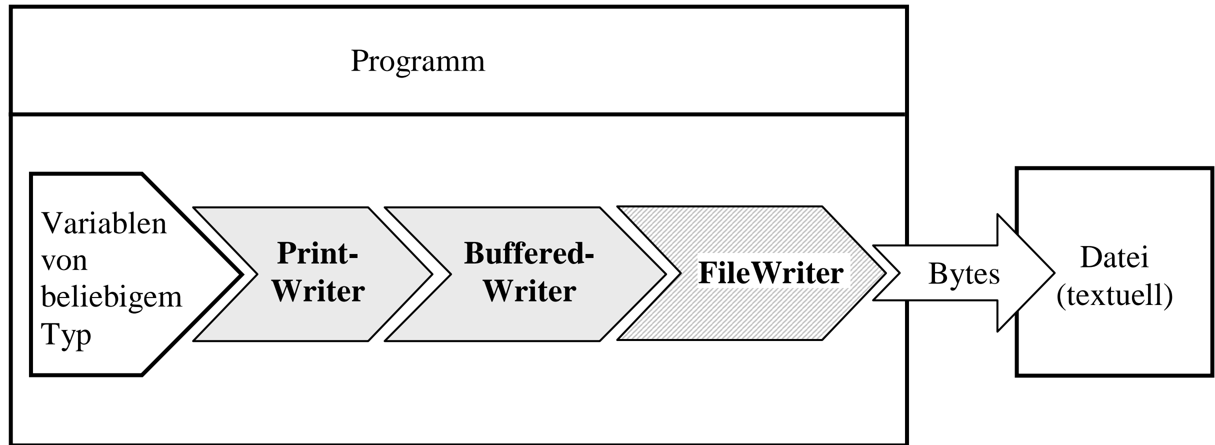
10.4.1.6 FileWriter

Die **PrintWriter**-Klasse wird häufig zur textuellen Ausgabe von primitiven Datenwerten oder Zeichenketten in eine *Datei* verwendet und dabei bevorzugt zusammen mit der Klasse **FileWriter** eingesetzt (siehe Abbildung in Abschnitt 10.4.1.4).
In obiger Darstellung der **Writer**-Hierarchie erhielt die Klasse **FileWriter** einen schraffierten Hintergrund, weil sie sich benimmt wie die (weiß hinterlegten) Ausgabeklassen, obwohl sie keinen direkten Kontakt mit einer Datei hat, sondern intern zu diesem Zweck ein **FileOutputStream**-Objekt erzeugt, also eigentlich eine (grau zu hinterlegende) Transformationsklasse ist.
Wichtiger als die akademische Bemerkung zur korrekten Klassifikation der Klasse **FileWriter** sind ihre bequemen Konstruktoren, die das Öffnen einer per **String** oder **File**-Objekt bestimmten Datei zum Überschreiben oder Anhängen von Daten erlauben:

Constructor Summary
`FileWriter(File file)` Constructs a FileWriter object given a File object.
`FileWriter(File file, boolean append)` Constructs a FileWriter object given a File object.
`FileWriter(String fileName)` Constructs a FileWriter object given a file name.
`FileWriter(String fileName, boolean append)` Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

Wie es bei einer **OutputStreamWriter**-Unterklasse zu erwarten ist, wandelt ein **FileWriter**–Objekt die übergebenen Zeichen in Bytes. Dabei kommt das voreingestellte Kodierungsschema der Plattform zum Einsatz (bei MS-Windows: *Cp1252* alias *Windows Latin 1*).
Wer mit dem voreingestellten Kodierungsschema nicht zufrieden ist, kann eine andere **OutputStreamWriter**–Unterklasse benutzen (siehe Abschnitt 10.4.1.2). Hier zeigt sich die Flexibilität der **Writer**-Hierarchie gegenüber der älteren **OutputStream**-Hierarchie, deren **PrintStream**-Klasse auf die Standardkodierung festgelegt ist.

Ein Beispiel für den **FileWriter**-Einsatz zusammen mit einem **PrintWriter** haben Sie schon in Abschnitt 10.4.1.4 kennen gelernt. In Abschnitt 10.4.1.3 war sogar das bei hohen Performanz-Ansprüchen empfehlenswerte Gespann aus **PrintWriter**, **BufferedWriter** und **FileWriter** zu sehen:



10.4.1.7 Sonstige Writer-Subklassen

Bei den folgenden **Writer**-Unterklassen beschränken wir uns auf kurze Hinweise:

- **StringWriter** und **CharArrayWriter**
  StringWriter-Objekte schreiben ihre Ausgabe in einen dynamisch wachsenden **StringBuffer**. Im folgenden Beispiel werden die Eingaben von einem **PrintWriter** geliefert:

Quellcode	Ausgabe

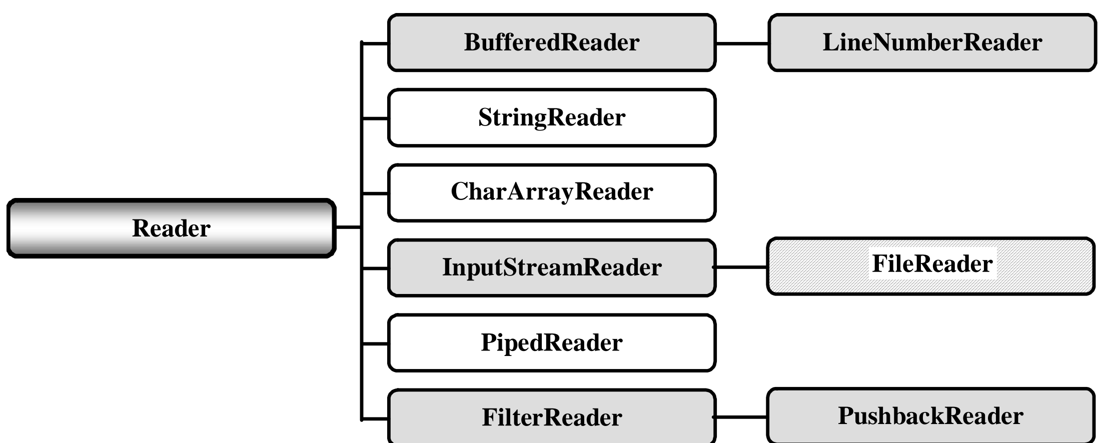
<pre>import java.io.*;
class StringWriterDemo {
   public static void main(String[] args) {
     StringWriter sw = new StringWriter();
     PrintWriter pw = new PrintWriter(sw);
     for (int i = 1; i <= 5; i++)
        pw.println("Zeile " + i);
     System.out.println(sw.toString());
   }
}</pre>
	<pre>Zeile 1
	Zeile 2
	Zeile 3
	Zeile 4
	Zeile 5</pre>

  CharArrayWriter-Objekte verhalten sich im Wesentlichen genauso.
- **PipedWriter**
  Diese Klasse ist das zeichenorientierte Analogon zum **PipedOutputStream**.
- **FilterWriter**
  Diese abstrakte Basisklasse bietet sich dazu an, eigene Transformationsklassen für zeichenorientierte Ausgabeströme abzuleiten.

10.4.2 Die Reader-Hierarchie

10.4.2.1 Überblick

Alle IO-Klassen zur Verarbeitung von zeichenorientierten *Eingabe*strömen stammen von **Reader** ab:



In der Abbildung sind Eingabeklassen (in direktem Kontakt mit einer Datenquelle) mit weißem Hintergrund dargestellt, Eingabe*transformations*klassen mit grauem Hintergrund. Weil die Klasse **FileReader** eine später noch zu besprechende Besonderheit aufweist, ist sie mit schraffiertem Hintergrund gezeichnet.

Bei den meisten **Reader**-Unterklassen beschränken wir uns auf kurze Hinweise:

- **BufferedReader**
  Diese Klasse dient zur gepufferten Eingabe und ist vor allem wegen ihrer **readLine()**-Methode zum Einlesen einer kompletten Zeile recht beliebt.
  Die voreingestellte Puffergröße von 8192 Zeichen kann per Konstruktor geändert werden. Nach meinen Erfahrungen ist allerdings durch einen **BufferedReader** kaum ein Zeitgewinn zu erzielen (ca. 10%). Eine mögliche Erklärung für den mäßigen Effekt ist vielleicht die von der Klasse **InputStreamReader** automatisch vorgenommene Byte-Strom-Pufferung. Eventuell sind auch andere Optimierungen durch das Java-Laufzeit- oder das Betriebssystem beteiligt.
- **LineNumberReader**
  Dieser gepufferte Zeichen-Eingabestrom erweitert seine Basisklasse um Methoden zur Verwaltung der Zeilennummern.
- **StringReader** und **CharArrayReader**
  StringReader bzw. **CharArrayReader** lesen aus einem **String** bzw. **char**-Array.
- **InputStreamReader**
  Diese Brückenklasse ist das Gegenstück zu Klasse **OutputStreamReader**, wandelt also Bytes in Unicode-Zeichen unter Verwendung eines einstellbaren Kodierungsschemas (vgl. Abschnitt 10.4.1.2).
  Analog zu dem in Abschnitt 10.4.1.3 beschriebenen Verhalten der Klasse **OutputStreamReader** findet zur Beschleunigung der Konvertierung automatisch eine Pufferung des Byte-Stroms statt.
- **PipedReader**
  Objekte dieser Klasse lesen Zeichen aus einer Pipe, die zur Kommunikation zwischen Threads dient.
- **FilterReader**
  Diese abstrakte Basisklasse bietet sich dazu an, eigene Transformationsklassen für zeichenbasierte Eingabeströme abzuleiten.
- **PushbackReader**
  Diese eher selten benötigte Klasse bietet Methoden, um aus einem Eingabestrom entnommenen Zeichen wieder zurück zu stellen.

10.4.2.2 FileReader

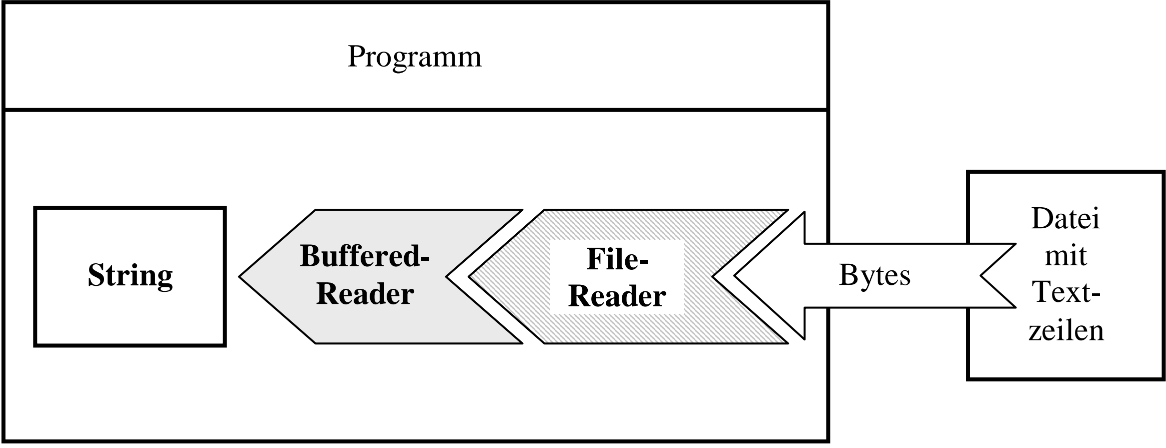
Die Klasse **FileReader** ist das Gegenstück zur Klasse **FileWriter**. Sie kann wie eine Eingabeklasse verwendet werden, obwohl sie keinen direkten Kontakt zu einer Datei hat, sondern intern zu diesem Zweck ein **FileInputStream**-Objekt erzeugt, also eigentlich eine Transformationsklasse ist.
Wie es bei einer **InputStreamReader**-Unterklasse zu erwarten ist, wandelt ein **FileReader**–Objekt die eingelesenen Bytes in Unicode-Zeichen um. Dabei kommt das voreingestellte Kodierungsschema der Plattform zum Einsatz (bei MS-Windows: Cp1252 alias Windows Latin 1). Wer mit dem voreingestellten Kodierungsschema nicht zufrieden ist, kann eine andere **InputStreamReader**–Unterklasse benutzen (vgl. Abschnitt 10.4.1.2).

In der Regel setzt man vor den **FileReader** noch einen **BufferedReader**, der zwar nur einen mäßigen Zeitgewinn bringt (siehe oben), aber die bei Dateien mit Zeilenstruktur sehr nützliche Methode **readLine()** beisteuert, z.B.:

Quellcode	Ausgabe

<pre>import java.io.*;
class FR {
  public static void main(String[] args) throws IOException {
    String[] starry = new String[100];
    BufferedReader br = new BufferedReader(
                         new FileReader("test.txt"));
    for (int i = 0; i < 100; i++)
      starry[i] = br.readLine();
    System.out.println(starry[99]);
    br.close();
  }
}</pre>
	Zeile 100

Das Beispielprogramm arbeitet mit folgender Datenstrom-Architektur:



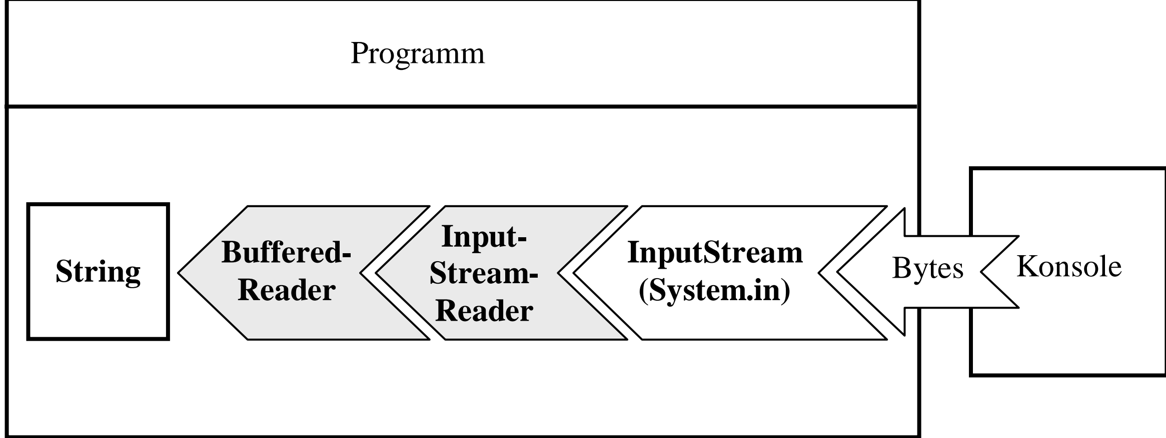
10.4.2.3 *Eingaben an der Konsole*

Nun sind Sie im Stande, die von der Klasse `Simput` zur Verfügung gestellten Methoden zur Eingabe primitiver Datentypen via Tastatur komplett zu verstehen (und zu kritisieren). Als Beispiel wird die Methode `gint()` wiedergegeben:

```
import java.io.*;
public class Simput {
    static public boolean status;
    static public String  errdes = "";
    .   .   .
    .   .   .
    static public int gint() {
        int eingabe = 0;
        BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));
        try {
            eingabe = Integer.parseInt(in.readLine().trim());
            status = true;
            errdes = "";
        } catch(Exception e) {
            status = false;
            errdes = "Eingabe konnte nicht konvertiert werden.";
            System.out.println("Falsche Eingabe!\n");
        }
        return eingabe;
    }
    .   .   .
    .   .   .
}
```

Die Konsole ist mit dem per **System.in** ansprechbaren Objekt aus der byteorientierten Klasse **InputStream** verbunden.[1] Mit einem **InputStreamReader** werden die hereinströmenden Bytes in einen Zeichenstrom gewandelt, aus dem über einen **BufferedReader** gelesen wird, bevorzugt mit der bequemen Methode **readLine()**.

Den Weg von der Tastatur bis zum **String**, der dann von der statischen Methode **Integer.parseInt()** weiterverarbeitet wird, kann man sich ungefähr so vorstellen:



10.5 Objektserialisierung

Nach vielen Mühen und lästigen Details kommt nun als Belohnung für die Ausdauer eine angenehm einfache Lösung für ein wichtiges Problem. Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern und laden. Erfreulicherweise können Objekte tatsächlich genau so wie primitive Datentypen in einen Bytestrom geschrieben bzw. von dort gelesen werden. Die zuständigen Klassen gehören zur **OutputStream**- bzw. **InputStream**-Hierarchie und hätten schon früher behandelt werden können. Für ein attraktives und wichtiges Spezialthema ist aber auch die Platzierung am Ende der IO-Behandlung (sozusagen als Krönung) nicht unangemessen.

Die Übersetzung eines Objektes (mit all seinen Instanzvariablen) in einen Bytestrom bezeichnet man recht treffend als *Objektserialisierung*. Einzige Voraussetzung für die Serialisierbarkeit einer Klasse: Sie muss die Schnittstelle **Serializable** implementieren, wobei diese Schnittstelle allerdings keinerlei Methoden deklariert.

Für das Schreiben von Objekten ist die byteorientierte Ausgabeklasse **ObjectOutputStream** zuständig, für das Lesen die byteorientierte Eingabeklasse **ObjectInputStream**.

In folgendem Beispielprogramm wird ein Objekt der serialisierbaren Klasse `Kunde` mit der **ObjectOutputStream-**Methode **writeObject()** in eine Datei befördert und anschließend mit der **ObjectInputStream**–Methode **readObject()** von dort zurück geholt. Außerdem wird demonstriert, dass die beiden Klassen auch Methoden zum Schreiben bzw. Lesen von primitiven Datentypen besitzen (z.B. **writeInt()** bzw. **readInt()**):

[1] Eine Besonderheit (ohne große Relevanz für unser gegenwärtiges Thema) besteht darin, dass aus der abstrakten (!) Klasse **InputStream** eigentlich keine Objekte erzeugt werden können.

```
// Datei Kunde.java
import java.io.*;

class Kunde implements Serializable {
   private String vorname;
   private String name;
   private transient int stimmung;
   private int nkaeufe;
   private double aussen;
   public Kunde(String vorname_, String name_, int stimmung_,
                int nkaeufe_, double aussen_) {
      vorname = vorname_;
      name = name_;
      stimmung = stimmung_;
      nkaeufe = nkaeufe_;
      aussen = aussen_;
   }
   public void prot() {
      System.out.println("Name: " + vorname + " " + name);
      System.out.println("Stimmung: " + stimmung);
      System.out.println("Anz.Einkaeufe: " + nkaeufe +
                         " Aussenstaende: " + aussen+ "\n");
   }
}

// Datei Serialisierung.java
import java.io.*;

class Serialisierung {
   public static void main(String[] args) throws Exception {
      Kunde demo = new Kunde("Fritz", "Orth", 1, 13, 426.89);
      System.out.println("Zu sichern:\n");
      demo.prot();
      int anzahl;
      ObjectOutputStream oos = new ObjectOutputStream(
                               new FileOutputStream("test.bin"));
      oos.writeObject(demo);
      oos.writeInt(1);
      oos.close();
      ObjectInputStream ois = new ObjectInputStream(
                              new FileInputStream("test.bin"));
      Kunde unbekannt = (Kunde) ois.readObject();
      anzahl = ois.readInt();
      System.out.println(anzahl + " Fall eingelesen:\n");
      unbekannt.prot();
   }
}
```

Beim Schreiben eines Objektes wird auch seine Klasse festgehalten. Beim Lesen eines Objektes wird zunächst die zugehörige Klasse festgestellt und in die virtuelle Maschine geladen (falls noch nicht vorhanden). Dann wird das Objekt auf dem Heap angelegt, und die Instanzvariablen erhalten die rekonstruierten Werte.
Als **transient** deklarierte Instanzvariablen werden von **writeObject()** und von **readObject()** übergangen. Damit eignet sich dieser Modifikator z.B. für ...

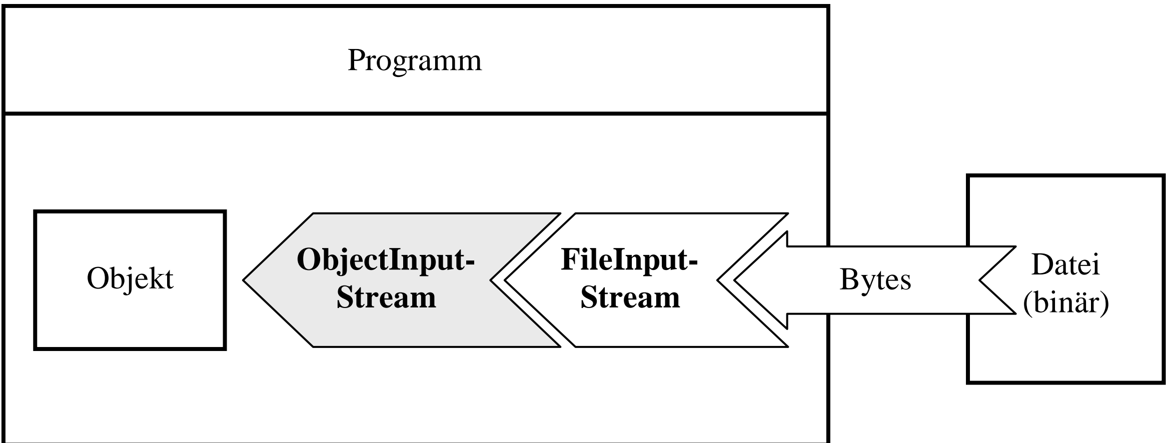
- Variablen, die aus Sicherheitsgründen nicht in einer Datei gespeichert werden sollen,
- Variablen, deren temporärer Charakter ein Speichern überflüssig macht.

Im Beispielprogramm gibt es keine sonderlich sinnvolle Anwendung für transiente Instanzvariablen. Die Wirkung des Modifikators ist trotzdem in der Ausgabe des Programms gut zu erkennen:

```
Zu sichern:

Name: Fritz Orth
Stimmung: 1
Anz.Einkaeufe: 13 Aussenstaende: 426.89

1 Fall eingelesen:

Name: Fritz Orth
Stimmung: 0
Anz.Einkaeufe: 13 Aussenstaende: 426.89
```

Die Instanzvariable `stimmung` des eingelesenen Objektes besitzt den **int**-Initialwert 0, während die übrigen Elementvariable über beide Serialisierungsschritte hinweg ihre Werte behalten haben.

In der folgenden Abbildung wird die Rekonstruktion eines Objektes skizziert:



10.6 Empfehlungen zur Verwendung der IO-Klassen

Weil die Java-IO-Behandlung durch die Vielzahl beteiligter Klassen auf Anfänger etwas unübersichtlich wirkt, folgt in diesem Abschnitt eine rezeptartige Beschreibung wichtiger Spezialfälle.

10.6.1 Textdaten in sequentielle Dateien schreiben

Um Textdaten (mit Datentyp **String** oder **char**) in eine sequentiell organisierte Datei zu schreiben, verwendet man in der Regel einen **FileWriter** (siehe Abschnitt 10.4.1.6), dessen Konstruktor die Ausgabedatei öffnet, in Kombination mit einem **PrintWriter** (siehe Abschnitt 10.4.1.4), der bequeme Ausgabemethoden bietet (z.B. **println**()).

Beispiel:

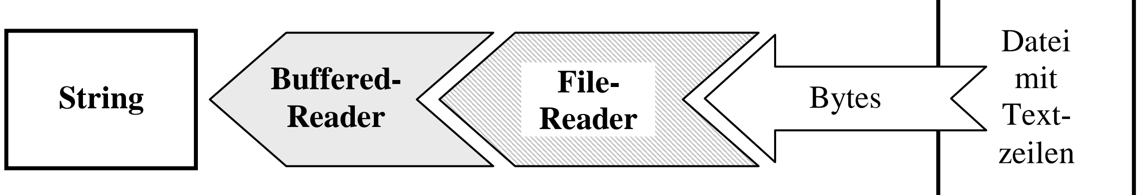
```
PrintWriter pw =  new PrintWriter(
                  new FileWriter("test.txt"));
pw.println("Diese Zeile landet in der Datei test.txt.");
```

Wer mit dem voreingestellten Kodierungsschema nicht zufrieden ist, kann statt **FileWriter** die Klasse **OutputStreamWriter** benutzen (siehe Abschnitt 10.4.1.2).

Neben **char**- oder **String**-Inhalten, die Java-intern als Unicode-Zeichen abgelegt sind, können dank **PrintWriter** auch andere Datentypen geschrieben werden, wobei in eine Zeichenfolgen-Repräsentation gewandelt wird.

10.6.2 Textdaten aus sequentiellen Dateien lesen

Um Textdaten aus einer sequentiell organisierte Datei zu lesen, verwendet man in der Regel ein Objekt aus der Klasse **FileReader** (siehe Abschnitt 10.4.2.2).
In der Regel schaltet man hinter den **FileReader** noch einen **BufferedReader**, der die Anzahl der Dateizugriffe reduziert und vor allem die nützliche Methode **readLine()** bietet.



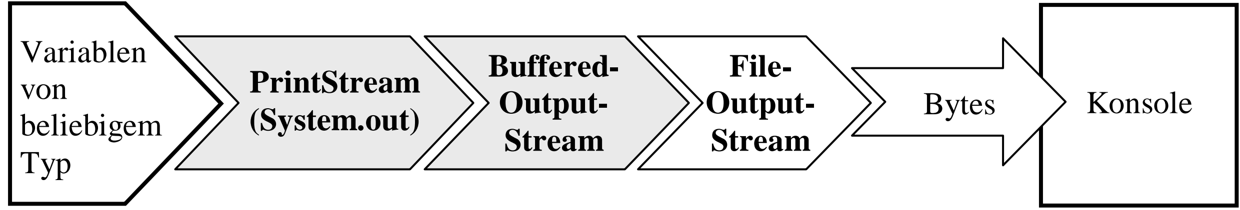
Beispiel:

```
BufferedReader br =  new BufferedReader(
                     new FileReader("test.txt"));
String s = br.readLine();
```

Wer mit dem voreingestellten Kodierungsschema nicht zufrieden ist, kann statt **FileReader** die Klasse **InputStreamReader** benutzen (siehe Abschnitt 10.4.1.2).

10.6.3 Ausgabe auf die Konsole

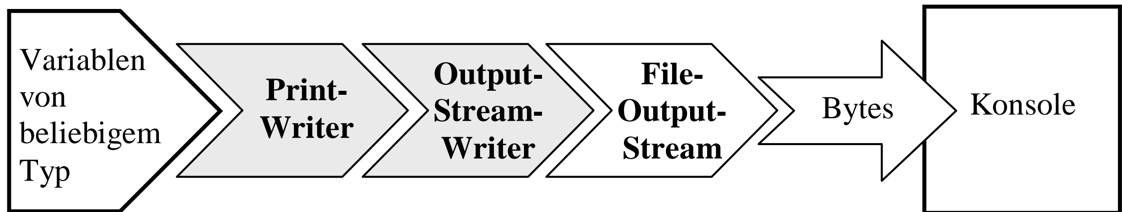
In Abschnitt 10.3.1.5 wird beschrieben, wie die Inhalte von Variablen beliebigen Typs mit Hilfe der **PrintStream**-Methoden **print()** und **println()** zum automatisch initialisierten Standard-Ausgabestrom **System.out** gelangen:



Beispiel:

```
System.out.println("Hallo");
```

Wenn es Problem mit dem automatisch gewählten Kodierungsschema gibt, ist statt dessen ein **PrintWriter** mit passendem **OutputStreamWriter** zu verwenden:



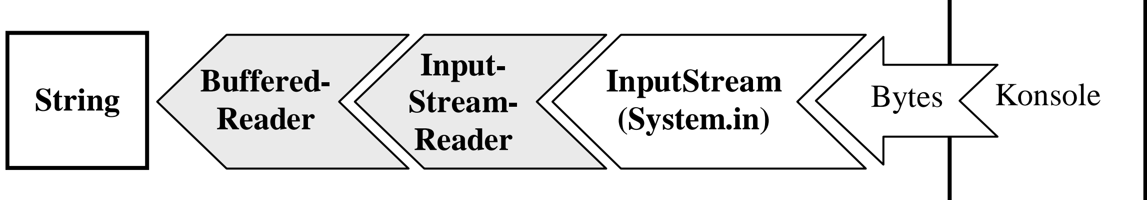
Weil der **OutputStreamWriter** die durch Wandlung von Unicode-Zeichen entstehenden Bytes automatisch puffert (vgl. Abschnitt 10.4.1.3), sollte beim **PrintWriter** die **autoFlush**-Option eingeschaltet werden.

Beispiel:

```
FileOutputStream fos = new FileOutputStream(FileDescriptor.out);
PrintWriter pw = new PrintWriter(
            new OutputStreamWriter(fos, "Cp850"), true);
pw.println("Hallo Wörld");
```

10.6.4 Tastatureingaben

In Abschnitt 10.4.2.3 wird beschrieben, wie man Tastatureingaben mit Hilfe der Transformationsklassen **InputStreamReader** und **BuffererdReader** vom **InputStream**-Objekt **System.in** entgegen nimmt.

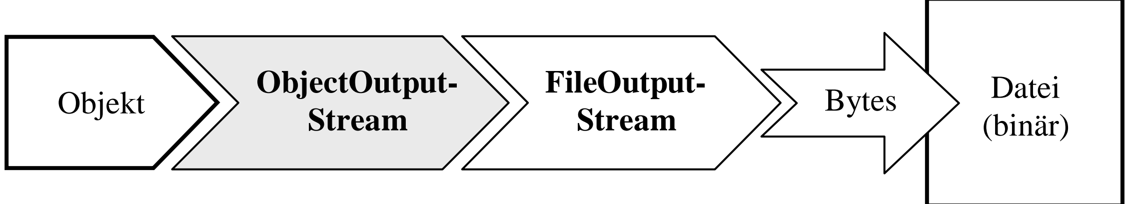


Beispiel:

```
BufferedReader in = new BufferedReader(
                    new InputStreamReader(System.in));
System.out.print("Ihr Alter: ");
String s = in.readLine();
int alter = Integer.parseInt(s);
```

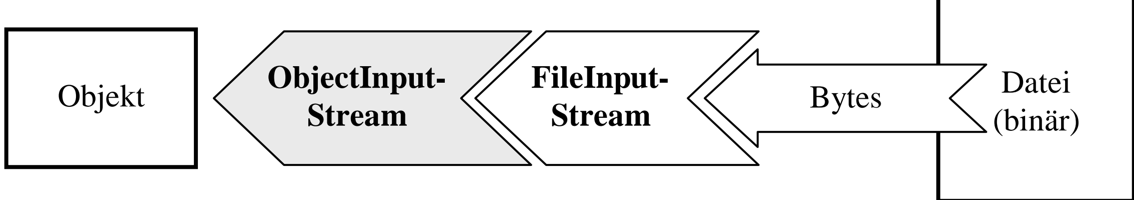
10.6.5 Objekte schreiben und lesen

Um Objekte in eine Datei zu schreiben oder aus einer zu Datei lesen, verwendet man die in Abschnitt 10.5 vorgestellten Klassen **ObjectOutputStream** und **ObjectInputStream**:



Beispiel:

```
ObjectOutputStream oos = new ObjectOutputStream(
                         new FileOutputStream("test.bin"));
oos.writeObject(demobj);
```

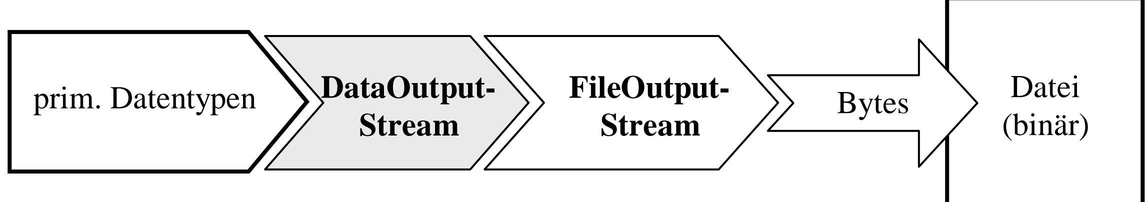


Beispiel:

```
ObjectInputStream ois = new ObjectInputStream(
                        new FileInputStream("test.bin"));
Kunde unbekannt = (Kunde) ois.readObject();
```

10.6.6 Primitive Datentypen binär in eine sequentielle Datei schreiben

Um primitive Datentypen (z.B. **int**, **double**) binär in eine sequentiell organisierte Datei zu schreiben, verwendet man ein Filterobjekt aus der Klasse **DataOutputStream** in Kombination mit einem Ausgabeobjekt aus der Klasse **FileOutputStream**:

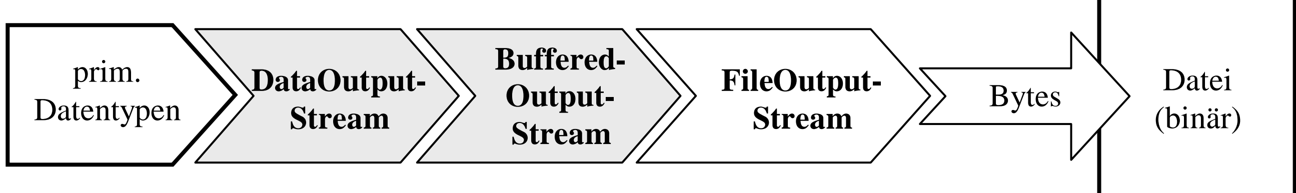


Beispiel:

```
dos = new DataOutputStream(new FileOutputStream("demo.dat"));
dos.writeInt(4711);
dos.writeDouble(Math.PI);
```

Beim primitiven Datentyp **byte** ist kein **DataOutputStream**-Filterobjekt erforderlich, weil bereits die **OutputStream**-Methoden eine hinreichende Funktionalität bieten.

Soll die Ausgabe gepuffert erfolgen, um die Anzahl der Dateizugriffe gering zu halten, dann muss ein Filterobjekt aus der Klasse **BufferedOutputStream** eingesetzt werden:



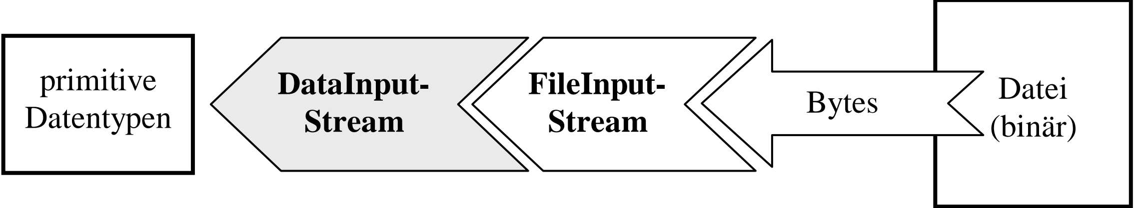
Hier wird ein Puffer mit 16384 Bytes Kapazität verwendet:

```
dos = new DataOutputStream(
      new BufferedOutputStream(
      new FileOutputStream("demo.dat"), 16384));
```

Beim Einsatz eines Puffers muss auf jeden Fall vor dem Programmende die **flush()**-Methode des **DataOutputStream**-Objektes aufgerufen werden, wobei sich der Aufruf automatisch fortpflanzt. Die **flush()**-Methode muss nicht unbedingt explizit aufgerufen werden, weil der ohnehin generell empfohlene **close()**-Aufruf dies automatisch erledigt.

10.6.7 Primitive Datentypen aus einer sequentiellen Binärdatei lesen

Um primitive Datentypen (z.B. **int**, **double**) aus einer sequentiell organisierten Datei zu lesen, verwendet man ein Filterobjekt aus der Klasse **DataInputStream** in Kombination mit einem Eingabeobjekt aus der Klasse **FileInputStream**:

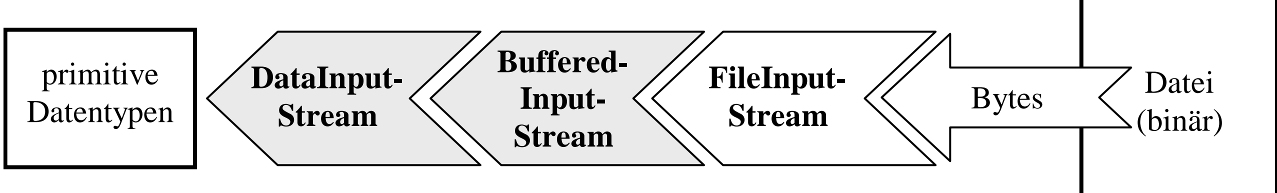


Beispiel:

```
dis = new DataInputStream(new FileInputStream("demo.dat"));
i = dis.readInt();
d = dis.readDouble();
```

Beim primitiven Datentyp **byte** ist kein **DataInputStream**-Filterobjekt erforderlich, weil bereits die **InputStream**-Methoden eine hinreichende Funktionalität bieten.

Soll die Eingabe gepuffert erfolgen, um die Anzahl der Dateizugriffe gering zu halten, dann muss ein Filterobjekt aus der Klasse **BufferedInputStream** eingesetzt werden:



Hier wird ein Puffer mit 16384 Bytes Kapazität verwendet:

```
dis = new DataInputStream(
      new BufferedInputStream(
      new FileInputStream("demo.dat"), 16384));
```

10.7 Missbilligte Methoden

Im Laufe der Evolution des Java-I/O-Systems mussten einige vormals gebräuchliche Methoden als *deprecated* (dt.: *veraltet*, *unerwünscht*, *missbilligt*) eingestuft werden. Sie sind aus Kompatibilitätsgründen noch vorhanden, doch wird von ihrer Verwendung abgeraten. Ein Beispiel ist die Methode **readLine()** der Klasse **DataInputStream**, die in Java 1.0 verwendet wurde, um Zeichenketten von der Tastatur einlesen, z.B.:

Quellcode

```
import java.io.*;

class DataInput{
  public static void main(String[] args)
      throws IOException {
    String wort;
    DataInputStream dis = new DataInputStream(System.in);
    System.out.print("Wort eingeben: ");
    wort = dis.readLine();
    System.out.println("Sie gaben ein: " + wort);
    dis.close();
  }
}
```

Ausgabe

```
Wort eingeben: egal
Sie gaben ein: egal
```

Als byteorientierte Eingabetransformationsklasse kann **DataInputStream** mit beliebigen **InputStream**-Abkömmlingen kooperieren. In Java ist die Konsoleneingabe als Objekt aus der Klasse **InputStream** realisiert, das über die statische Instanzvariable **System.in** angesprochen werden kann.
Außerdem besitzt die Klasse **DataInputStream** eine Methode **readLine()**, und man kann erwarten, mit einem auf **System.in** aufgesetzten **DataInputStream**-Objekt Zeichenketten von der Tastatur einlesen zu können.
Der Beispieldialog klappt auch erwartungsgemäß, aber die folgende Compiler-Warnung erinnert uns daran, dass man Zeichenketten ja eigentlich *nicht* mit byteorientierten Strömen einlesen soll:

```
Note: U:\Eigene Dateien\Java\BspUeb\IO\DataInputStream\Falsche Verwen-
dung\Deprecated\DataInput.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.
```

In der Online-Dokumentation zur **DataInputStream**-Methode **readLine()** erhalten wir genauere Informationen und Empfehlungen:

> **Deprecated.** *This method does not properly convert bytes to characters. As of JDK 1.1, the preferred way to read lines of text is via the* `BufferedReader.readLine()` *method. Programs that use the* `DataInputStream` *class to read lines can be converted to use the* `BufferedReader` *class by replacing code of the form:*
>
> ```
>           DataInputStream d = new DataInputStream(in);
> ```
>
> *with:*
>
> ```
>           BufferedReader d
>                = new BufferedReader(new InputStreamReader(in));
> ```

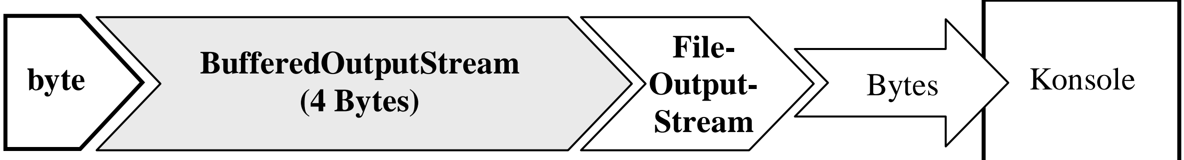
Diese Kritik richtet sich nur gegen die Methode **DataInputStream.readln()**, keinesfalls gegen die immer noch wichtige Klasse **DataInputStream**, mit der wir oben erfolgreich Werte primitiver Datentypen aus einer Binärdatei (!) gelesen haben.

10.8 Übungsaufgaben zu Abschnitt 10

1) Erstellen Sie ein Programm zur Demonstration der Ausgabepufferung. Um mitverfolgen zu können, wie beim Überlaufen des Puffers Daten weitergeleitet werden, sollte als Senke die Konsole zum Einsatz kommen.
Wie Sie aus dem Abschnitt 10.3.1.5 wissen, ist der per **System.out** ansprechbare **PrintStream** bei aktivierter **autoFlush**-Option hinter einen **BufferedOutputStream** mit 128 Bytes Puffergröße geschaltet, was insgesamt keine guten Beobachtungsmöglichkeiten bietet.
Als Alternative mit besseren Forschungsmöglichkeiten wird daher folgende Ausgabestromkonstruktion vorgeschlagen:

```
FileOutputStream fdout =
     new FileOutputStream(FileDescriptor.out);
BufferedOutputStream bos =
     new BufferedOutputStream(fdout, 4);
```

Über die statische Variable **out** der Klasse **FileDescriptor** wird der Bezug zur Konsole hergestellt. Dorthin schreibt der **FileOutputStream** `fdout`, an den der **BufferedOutputStream** `bos` mit der untypisch kleinen Puffergröße von 4 Bytes gekoppelt ist:



An Stelle der bequemen **PrintStream**-Ausgabemethoden (z.B. **println()**) kommen wir mit der **BufferedOutputStream**-Methode **write()** aus, wenn die auszugebenden Bytes so gewählt werden, dass eine interpretierbare Bildschirmausgabe entsteht. Dies ist z.B. bei folgenden Aufruf der Fall:

```
bos.write(i+47);
```

Bei `i` = 1 wird das niederwertigste Byte der **int**-Zahl 48 (= 0x30) in den Ausgabestrom geschoben. Dieses ist aber in jedem 8-Bit-Zeichensatz die Kodierung der Null, so dass diese Ziffer auf der Konsole erscheint. Bei `i` = 2 erscheint dementsprechend eine Eins usw.
Jetzt müssen Sie nur noch per „Zeitlupe“ dafür sorgen, dass man das Füllen und Entleeren des Puffers mitverfolgen kann, z.B.:

```
  start = System.currentTimeMillis();
  for (byte i = 1; i <= 10; i++) {
     time = start + i*1000;
     while (System.currentTimeMillis() < time);
     bos.write(i+47);
     System.out.print('\u0007');
  }
```

Im Lösungsvorschlag wird zusätzlich per Ton die Ankunft eines Bytes im Puffer gemeldet.
Sollten sich bei Programmende noch Bytes im Puffer befinden, müssen diese per **flush()** oder **close()** vor dem Untergang bewahrt werden.

2) Als byteorientierte Eingabetransformationsklasse kann **DataInputStream** in Kooperation mit beliebigen **InputStream**-Abkömmlingen die Werte primitiver Datentypen lesen. In Java zeigt die Referenzvariable **System.in** auf ein Objekt aus der Klasse **InputStream**. Nun kann man hoffen, mit einem **DataInputStream**-Objekt, das auf **System.in** aufsetzt, Werte primitiver Datentypen von der Tastatur entgegen nehmen zu können. In folgendem Programm wird versucht, beim Benutzer einen **short**-Wert (Ganzzahl mit 2 Bytes) zu erfragen:

Quellcode

```
import java.io.*;

class DataInput{
  public static void main(String[] args)
       throws IOException {
     short zahl;
     DataInputStream dis = new DataInputStream(System.in);
     System.out.print("Zahl eingeben: ");
     zahl = dis.readShort();
     System.out.println("OK: " + zahl);
  }
}
```

Ausgabe

```
Zahl eingeben: 0
OK: 12301
```

Ein Blick auf den Beispieldialog zeigt, dass offenbar eine Transformation stattfindet, deren Ergebnis jedoch wenig brauchbar ist.
Wie ist das Ergebnis zu erklären?
Welche Transformationsklassen sollten statt **DataInputStream** eingesetzt werden?
Wozu sollte man die Klasse **DataInputStream** verwenden?

3) Erweitern Sie bitte das **DataOutputStream**-Demonstrationsprogramm in Abschnitt 10.3.1.3 um eine Aus- bzw. Eingabepufferung.
4) Wie kann man bei folgendem Programm den Quellcode vereinfachen und dabei auch noch die Laufzeit ungefähr halbieren?

```
import java.io.*;

class PrintWriterDemo {
  public static void main(String[] egal) throws IOException {
     long time = System.currentTimeMillis();
     FileOutputStream fos = new FileOutputStream("pw.txt");
     PrintWriter pw = new PrintWriter(fos, true);
     for (int i = 1; i < 30000; i++) {
       pw.println(i);
     }
     pw.close();
     System.out.println("Zeit: " + (System.currentTimeMillis()-time));
  }
}
```

5) Zum Schluss noch eine Aufgabe für besonders aufmerksame Leser: Die beiden folgenden Programme unterscheiden sich darin, dass ein **PrintWriter** bzw. ein **PrintStream** zur Ausgabe von Zeichenfolgen auf der Konsole eingesetzt wird. In beiden Fällen ist die **autoFlush**-Option eingeschaltet, und es kommt die in beiden Klassen vorhandene **print()**-Methode zum Einsatz. Wie ist das unterschiedliche Verhalten zu erklären?

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>import java.io.*;
class Prog {
  public static void main(String[] args)
    throws IOException {
    PrintStream px = new PrintStream(
      new FileOutputStream(FileDescriptor.out), true);
    for (int i = 0; i < 10; i++)
      px.print("Zeile "+i+"\n");
  }
}</pre></td><td><pre>Zeile 0
Zeile 1
Zeile 2
Zeile 3
Zeile 4
Zeile 5
Zeile 6
Zeile 7
Zeile 8
Zeile 9</pre></td></tr>
</table>

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>import java.io.*;
class Prog {
  public static void main(String[] args)
    throws IOException {
    PrintWriter px = new PrintWriter(
      new FileOutputStream(FileDescriptor.out), true);
    for (int i = 0; i < 10; i++)
      px.print("Zeile "+i+"\n");
  }
}</pre></td><td></td></tr>
</table>

11 GUI-Programmierung mit Swing/JFC

Mit den Eigenschaften und Vorteilen einer graphischen Benutzeroberfläche (engl.: **G**raphical **U**ser **I**nterface) sind Sie sicher sehr vertraut. Eine GUI-Anwendung präsentiert dem Benutzer besitzt ein oder mehrere Fenster, die neben Bereichen zur Bearbeitung von programmspezifischen Dokumenten in der Regel mehrere Bedienelemente zur Benutzerinteraktion besitzen (z.B. Menüzeile, Befehlsschalter, Kontrollkästchen, Textfelder, Auswahllisten). Die von einer Plattform (in unserem Fall also von Java) zur Verfügung gestellten Bedienelemente bezeichnet man oft als *Komponenten*, *controls*, *Steuerelemente* oder *widgets* (Wortkombination aus *window* und *gadgets*). Weil die Steuerelemente intuitiv und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern Sie den Umgang mit moderner Software erheblich.

Im Vergleich zu Konsolenprogrammen geht es nicht nur anschaulicher und intuitiver, sondern vor allem auch „ereignisreicher" zu: Ein Konsolenprogramm entscheidet selbst darüber, welche Anweisung als nächstes ausgeführt wird, wenngleich das Programm auf Umgebungsbedingungen, Benutzerwünsche oder eingelesene Daten flexibel reagieren kann. Um seine Aufgaben zu erledigen, verwendet ein Konsolenprogramm diverse Dienste des Laufzeitsystems, z.B. beim Öffnen einer Datei. Für den Ablauf einer GUI-Applikation mit graphischer Benutzeroberfläche ist ein **ereignisorientiertes Paradigma** wesentlich, wobei das Laufzeitsystem als Vermittler von Ereignissen in erheblichem Maße den Ablauf mitbestimmt, indem es Methoden des Programms aufruft. Ausgelöst werden die Ereignisse in der Regel vom Benutzer, der mit der Hilfe von GUI-Komponenten seine Wünsche artikuliert.

Anmerkung zum Fenster-Design mit graphischen Entwicklungswerkzeugen

Wir werden in diesem Kurs generell die GUI-Gestaltung *manuell* vornehmen und dabei die technischen Grundlagen kennen lernen. Im späteren Programmieralltag sollten Sie zur Steigerung der Produktivität eine Entwicklungsumgebung mit graphischem Fenster-Designer verwenden (z.B. NetBeans von Sun, JBuilder von Borland). M.E. ist es erst dann sinnvoll, den zu einem Fenster gehörenden Java-Code automatisch erstellen zu lassen, wenn man die Assistenten-Produkte verstehen und nötigenfalls modifizieren kann.

11.1 Java Foundation Classes

Im Standard-Lieferumfang des Java-SDK sind leistungsfähige Klassen zur GUI-Programmierung enthalten, die gemeinsam als **J**ava **F**oundation **C**lasses (JFC) bezeichnet werden (siehe z.B. http://java.sun.com/products/jfc/index.html). Die JFC-Software verbessert und erweitert die ältere, als *Abstract Window Toolkit* (AWT) bezeichnete, Java-GUI-Technologie, die aber nicht überflüssig geworden ist und daher in folgender Liste mit JFC-Bestandteilen noch erscheint:

- **A**bstract **W**indowing **T**oolkit (AWT, enthalten im Paket **java.awt)**
  Das bereits in Java 1.0 vorhandene AWT ist heute teilweise überholt, stellt aber immer noch wichtige Basisklassen für die aktuellen GUI-Komponenten. Grundidee beim AWT-Entwurf war die möglichst weitgehende Verwendung von Steuerelementen des *Wirtsbetriebssystems*. Aus der notwendigen Beschränkung auf den kleinsten gemeinsamen Nenner aller zu unterstützenden Plattformen resultierte ein beschränkter Funktionsumfang. Die Kopplung jedes Java-Bedienelementes an eine Entsprechung des Betriebsystems förderte auch nicht unbedingt die Ausführungsgeschwindigkeit, vielleicht der Grund dafür, die AWT-Komponenten als „schwergewichtig" zu bezeichnen.
  In diesem Manuskript werden aus dem AWT nur noch die nach wie vor relevanten Basisklassen berücksichtigt. Wer (z.B. aus Kompatibilitätsgründen) die AWT-Steuerelemente verwenden möchte, kann sich z.B. in Kröckertskothen (2001, Kap. 13) informieren.

- **Swing** (enthalten im Paket **javax.swing**)
  Mit Java 1.1 wurden die komplett in Java realisierten „leichtgewichtigen" Komponenten eingeführt, die heute in der Regel zu bevorzugen sind, falls man nicht auf Java-1.0-Kompatibilität festgelegt ist. Während die *Top-Level-Fenster* nach wie vor schwergewichtig sind und als Schnittstelle zum Betriebsystem dienen, werden die Steuerelemente komplett von Java verwaltet, was einige gravierende Vorteile bringt:
  - Weil die Beschränkung auf den kleinsten gemeinsamen Nenner entfällt, stehen **mehr Komponenten** zur Verfügung.
  - Java-Anwendungen können auf allen Betriebsystemen ein **einheitliches Erscheinungsbild** bietet, müssen es aber nicht:
  - Für die Swing-Komponenten kann der Programmierer ein **Look & Feel** wählen (verfügbar: Java, Windows, Motif), während die AWT-Komponenten stets auf das GUI-Design des Betriebssystems festgelegt sind.

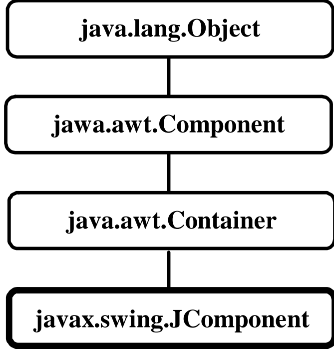
  Beim Zeichnen seiner Komponenten greift Java natürlich auf die primitiven Grafikroutinen des Betriebssystems zurück.
- **Java 2D** (enthalten im Paket **java.awt**)
  Mit dem Java2D-API lassen stehen leistungsfähige Klassen für ambitionierte 2D-Grafikanwendungen bereit.
- **Drag-&-Drop** (enthalten im Paket **java.awt.dnd**)
  Drag-&-Drop – Methoden werden Java-intern und auch bei der Kooperation mit anderen Anwendungen unterstützt.
- **Accessibility** (enthalten im Paket **javax.accessibility**)
  Über die Integration von zusätzlichen Ein- oder Ausgabegeräten in die Benutzerschnittstelle können Java-Programme leicht an spezielle Anforderungen angepasst werden (z.B. über Bildschirm-Leseprogramme oder Braille-Ausgaben für sehbehinderte Menschen).

11.2 Elementare Klassen im Paket javax.swing

Dieser Abschnitt soll einen Überblick zu den im Kurs eingesetzten Teilen des Swing-Paketes vermitteln.

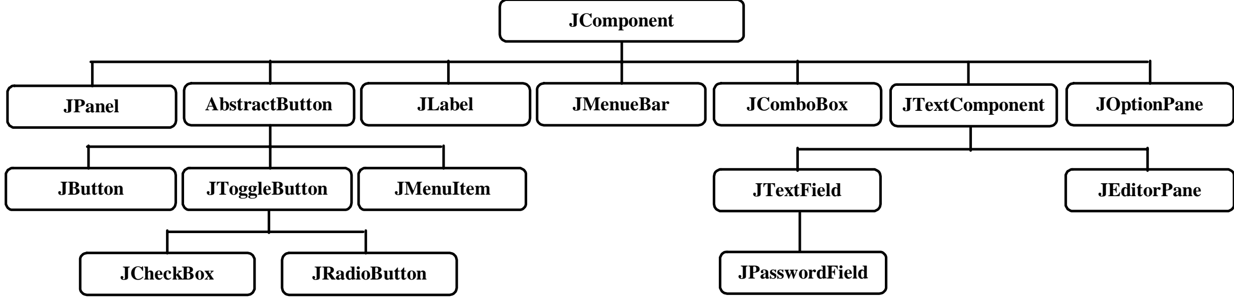
11.2.1 Komponenten und Container

Die fenster-internen Swing-Komponenten stammen meist von der Klasse **javax.swing.JComponent** ab, die wiederum zahlreiche Handlungskompetenzen und Eigenschaften über folgende Ahnenreihe erwirbt:

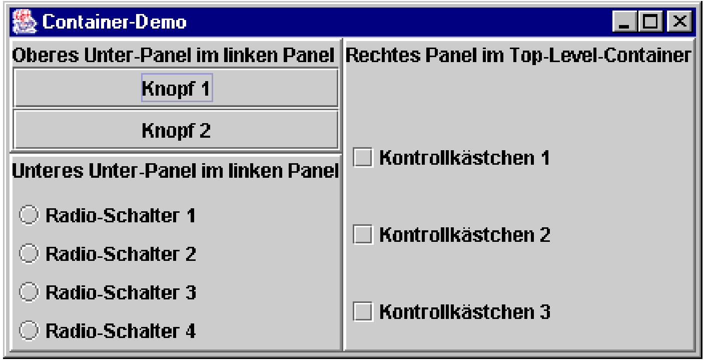


Komponenten sind also auch Objekte, allerdings mit optischer Präsenz auf dem Bildschirm und einigen zusätzlichen Kompetenzen: Sie interagieren mit dem Benutzer, wobei *Ereignisse* entstehen, die anderen Objekten zwecks Bearbeitung gemeldet werden (siehe unten).

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse für die in diesem Kurs behandelten **JComponent**-Abkömmlinge:[1]



Wie an den Namen unschwer zu erkennen ist, stehen die meisten Klassen für Steuerelemente, die aus GUI-Systemen wohlbekannt sind (Befehlsschalter, Label etc.).
In der Sammlung befindet sich mit **JPanel** aber auch ein *Container*. Diese Komponente entfaltet keine nennenswerte Interaktion mit dem Benutzer, sondern dient zur Aufnahme und damit zur Gruppierung von anderen Swing-Komponenten. Im Sinne einer flexiblen GUI-Gestaltung bietet Java die Möglichkeit, in einem Container neben „atomaren" Komponenten (z.B. **JButton**, **JLabel**) auch untergeordnete Container (in beliebiger Schachtelungstiefe) unterzubringen. Im folgenden Beispiel befinden sich innerhalb eines **JFrame**-Top-Level-Containers (siehe unten) insgesamt 4 **JPanel**-Container, um Komponenten aus den Klassen **JLabel**, **JButton**, **JCheckBox** und **JRadioButton** anzuordnen:



Die Frage, welcher von den beiden Begriffen *Komponente* und *Container* in Javas GUI-Technologie dem anderen vorgeordnet ist, kann aufgrund des Klassenstammbaums nicht perfekt geklärt werden:

- **java.awt.Container** stammt von **java.awt.Component** ab
- **javax.swing.JComponent** stammt von **java.awt.Container** ab

Relevant sind aber letztlich nicht die (teilweise historisch bedingten) Namen, sondern die Methoden und Eigenschaften, die eine Klasse von ihren Vorfahren übernimmt. Tatsächlich erben alle **JComponent**-Unterklassen von **java.awt.Container** die Methode **add()** zum „Einfüllen" von Komponenten in einen Container, so dass man z.B. ein Kontrollkästchen in einen Befehlsschalter einfügen kann:

[1] Die Klasse **JTextComponent** stammt aus dem Paket **javax.swing.text**, alle anderen Klassen stammen aus dem Paket **javax.swing**.

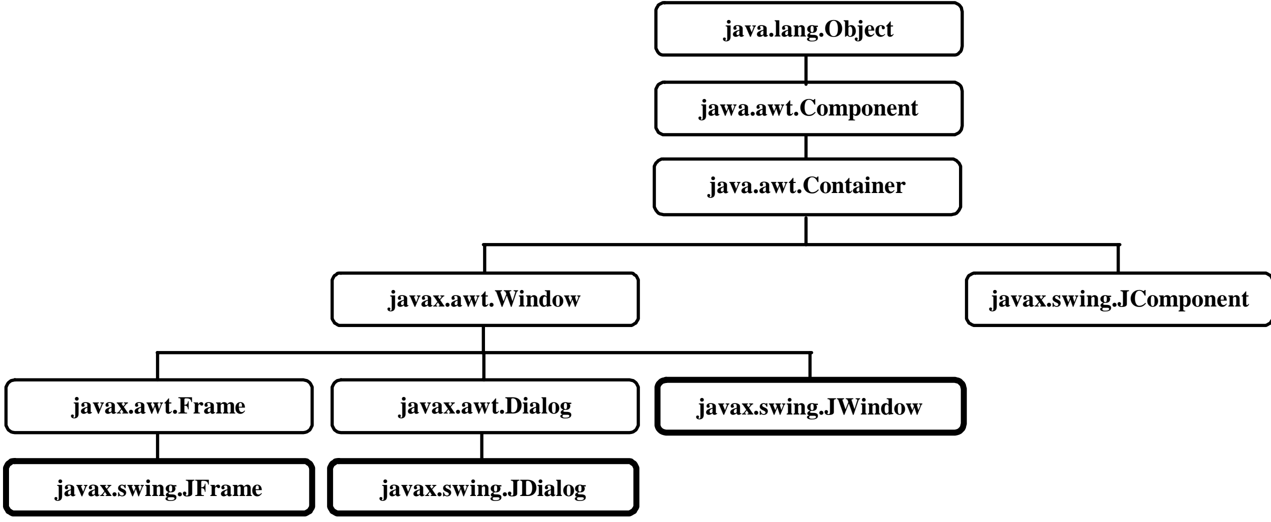
Allerdings fördern derartige Konstruktionen nicht unbedingt die Bedienbarkeit eines Programms. Im nächsten Abschnitt geht es um den Haupt-Container einer Swing-Anwendung:

11.2.2 Top-Level Container

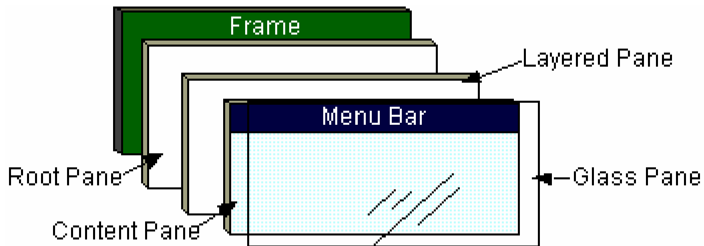
Jedes Programm mit Swing-GUI benötigt mindestens einen **Top-Level-Container**, der als Fenster zu der vom Betriebssystem verwalteten Benutzeroberfläche dient und die leichtgewichtigen Swing-Komponenten aufnimmt. In Abschnitt 11 werden alle Beispielprogramme als Top-Level-Container eine Komponente vom Typ **JFrame** benutzen, die ein Rahmenfenster mit folgender Ausstattung realisiert:

- Rahmen
- Titelzeile
- Bedienelemente zum Schließen, Minimieren und Maximieren des Fensters
- Systemmenü (über die Java-Tasse am linken Rand der Titelzeile erreichbar)

JFrame und die anderen schwergewichtigen Swing-Komponenten (siehe unten) stammen nicht von **JComponent** ab, sondern haben einen etwas anderen Stammbaum:



JFrame-Container sind mehrschichtig aufgebaut, wie die folgende (aus dem Java-Tutorial entnommene) Abbildung zeigt:



Wir werden uns auf die **Content Pane** (Inhaltsschicht) beschränken, die wir in unseren Programmen über eine von der **JFrame**-Methode **getContentPane()** gelieferte Referenz ansprechen können (siehe unten). Andere Schichten werden z.B. zum Realisieren der Drag-&-Drop - Funktionalität benötigt.

In Swing-Anwendungen sind noch andere Top-Level-Container möglich, die aber seltener benutzt werden:

- **JDialog**
  Mit dieser Klasse werden Dialogfenster realisiert, denen im Vergleich zum Rahmenfenster die Bedienelemente zum Maximieren und zum Minimieren fehlen. Sie können zwar auch als selbständige Fenster einer Anwendung arbeiten, werden aber meist als Kind eines Rahmenfensters erzeugt, so das sie beim Schließen des Besitzers automatisch verschwinden. In der kindlichen Rolle kann ein **JDialog**-Objekt *modal* arbeiten, so dass während seines Auftritts das übergeordnete Fenster blockiert ist.
- **JWindow**
  Bei den mit dieser Klasse erzeugten Fenster bestehen im Vergleich zu den Rahmenfenstern folgende Einschränkungen:
  - kein Rahmen
  - keine Titelzeile, also auch keine Bedienelemente zum Minimieren, Maximieren und Schließen sowie kein Systemmenü
  - Der Benutzer kann die Größe und die Position des Fensters nicht ändern.

Bei den später zu behandelnden Applets, die im Rahmen eines Web-Browser-Fensters ausgeführt werden, kann die Klasse **JApplet**-Objekt einen Swing-basierten Top-Level-Container realisieren.

11.3 Beispiel für eine Swing-Anwendung

In folgendem Swing-Programm kommen zwei Label (mit einem Text bzw. einem Bild als Inhalt) sowie ein Befehlsschalter zum Einsatz:

Den Quellcode werden wir im weiteren Verlauf des Manuskriptes vollständig besprechen:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class SwApp extends JFrame {
    private String labelPrefix = "Anzahl der Klicks: ";
```

```
    private JLabel label = new JLabel(labelPrefix + "0    ");
    private int numClicks = 0;
    final private byte maxicon = 3;
    private ImageIcon icon[] = new ImageIcon[maxicon];
    private JLabel labicon;
    private byte iconr = 0;

    public SwApp() {
        super("Swing-Demo");
        Container cont = getContentPane();

        JButton button = new JButton("Ich mag Swing!");
        button.setMnemonic(KeyEvent.VK_S);

        JPanel panli = new JPanel();
        panli.setBorder(BorderFactory.createEmptyBorder(30, 30, 30, 30));
        panli.setLayout(new GridLayout(0, 1));
        panli.add(button);
        panli.add(label);
        cont.add(panli, BorderLayout.CENTER);

        icon[0] = new ImageIcon("duke.gif");
        icon[1] = new ImageIcon("fight.gif");
        icon[2] = new ImageIcon("snooze.gif");
        labicon = new JLabel(icon[0]);
        cont.add(labicon, BorderLayout.EAST);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
                if (iconr < maxicon-1)
                   iconr++;
                else
                   iconr = 0;
                labicon.setIcon(icon[iconr]);
            }
        });

        addWindowListener(new Ex());

        setSize(300, 150);
        show();
    }

    public static void main(String[] args) {
        SwApp swapp = new SwApp();
    }

    private class Ex extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
           if (JOptionPane.showConfirmDialog(e.getWindow(),
                  "Wollen Sie nach "+ numClicks +
                  " Klicks wirklich schon aufhören?",
                  titel, JOptionPane.YES_NO_OPTION) == 0)
              System.exit(0);
    }
}
```

Zu Beginn werden die Pakete **javax.swing**, **java.awt** und **java.awt.event** importiert, um benötigte Klassen bequem ansprechen zu können.

Kernstück des Programms ist die von **JFrame** abgeleitete Klasse `SwApp`. Von dieser Rahmenfenster-Sorte wird in der **main()**-Methode *ein* Objekt erzeugt:

```
public static void main(String[] args) {
    SwApp swapp = new SwApp();
}
```

Weil das Rahmenfenster mit seinen Steuerelementen ein vergleichsweise komplexes Objekt darstellt, hat der `SwApp`-Konstruktor einige Arbeit:

- Das Fenster erhält seinen Titel durch expliziten Aufruf des Basisklassen-Konstruktors:
  ```
  super(titel);
  ```
- Um bequem auf die Inhaltsschicht (content pane) des Rahmenfensters zugreifen zu können, wird mit der **JFrame**-Methode **getContentPane()** eine lokale Referenzvariable auf diesen **Container** angelegt:
  ```
  Container cont = getContentPane();
  ```
- Wie der Konstruktor die Komponenten des Rahmenfensters erzeugt, konfiguriert und positioniert sowie die Ereignisbehandlung vorbereitet, wird gleich im Detail erklärt.
- Am Ende seiner Tätigkeit legt der Konstruktor noch eine initiale Größe für das Fenster fest und holt es dann auf den Bildschirm:
  ```
  setSize(300, 150);
  show();
  ```
  An Stelle des **show**-Methodenaufrufs findet man oft die folgende äquivalnete Alternative:
  ```
  setVisible(true);
  ```

Das `SwApp`-Objekt braucht keine weiteren Anweisungen (z.B. durch die Methode **main()**), um in Kooperation mit dem Benutzer und dem Laufzeitsystem für einen „spannenden und abwechslungsreichen“ Programmablauf zu sorgen. Es aktualisiert bei einem Mausklick auf den Schalter oder nach der Tastenkombination **<Alt>+<S>** die beiden Label, lässt sich verschieben, in der Größe ändern, minimieren, maximieren und beenden.

11.4 Swing-Komponenten, Teil 1

In diesem Manuskript werden (in zwei Portionen) elementare Swing-Komponenten vorgestellt. Eine optische Präsentation *aller* Swing-Komponenten finden Sie im Java-Tutorial (SUN Inc. 2002) über:

Creating a GUI with JFC/Swing > Getting Started with Swing >
About the JFC and Swing > A Visual Index to the Swing Components

11.4.1 Label

Wie man ein Label als Objekt aus der Klasse **JLabel** deklariert und über den **JLabel**-Konstruktor erzeugt, dürfte nach Inspektion des ersten Beispiels schon hinreichend klar sein:

```
private JLabel label;
   .    .    .
label = new JLabel(labelPrefix + "0    ");
```

Hier wird ein initial anzuzeigender Text vereinbart.

Das zweite Label unseres Beispielprogramms dient zur Anzeige von GIF-Dateien, die von **ImageIcon**-Objekten repräsentiert werden:

```
private ImageIcon icon[] = new ImageIcon[maxicon];
icon[0] = new ImageIcon("duke.gif");
icon[1] = new ImageIcon("fight.gif");
```

```
icon[2] = new ImageIcon("snooze.gif");
```

Die **ImageIcon**-Objekte passen ganz gut in den Abschnitt über GUI-Design, doch soll der begrifflichen Klarheit halber darauf hingewiesen werden, dass es sich *nicht* um Komponenten handelt, weil sie keinerlei Ereignisse auslösen können. Neben dem GIF-Format (*Graphics Interchange Format*) wird auch das JPEG-Format (*Joint Photographic Experts Group*) unterstützt.
Beim Erzeugen des zweiten Label-Objektes wird ein **ImageIcon** als initiale Anzeige festgelegt:

```
private JLabel labicon;
   .    .    .
labicon = new JLabel(icon[0]);
```

Man kann diverse Swing-Komponenten mit **ImageIcon**-Objekten verschönern, wobei die Auswahl per Konstruktor (wie bei **JLabel**) oder per **setIcon()**-Methode (wie bei **JButton**) erfolgt.

11.4.2 Befehlsschalter

Auch die Syntax zum Deklarieren bzw. Erzeugen eines Befehlsschalters bietet keinerlei Überraschungen:

```
private JButton button;
   .    .    .
JButton button = new JButton("Ich mag Swing!");
```

Mit der **JButton**-Methode **setMnemonic()** kann eine **<Alt>**-Tastenkombination als Äquivalent zum Mausklick auf den Schalter festgelegt werden, z.B. **<Alt><S>**:

```
button.setMnemonic(KeyEvent.VK_S);
```

Den **int**-wertigen Parameter der Methode **setMnemonic()** legt man am besten über die in der Klasse **KeyEvent** definierten VK-Konstanten (**V**irtual **K**ey) fest.

11.4.3 Zubehör für Swing-Komponenten

Es sind zahlreiche Möglichkeiten verfügbar, das optische Erscheinungsbild und die Bedienbarkeit von Swing-Komponenten zu verbessern.

Mit der **JComponent**-Methode **setToolTipText()** kann man Tool-Tipps zu einzelnen Steuerelementen definieren, z.B.:

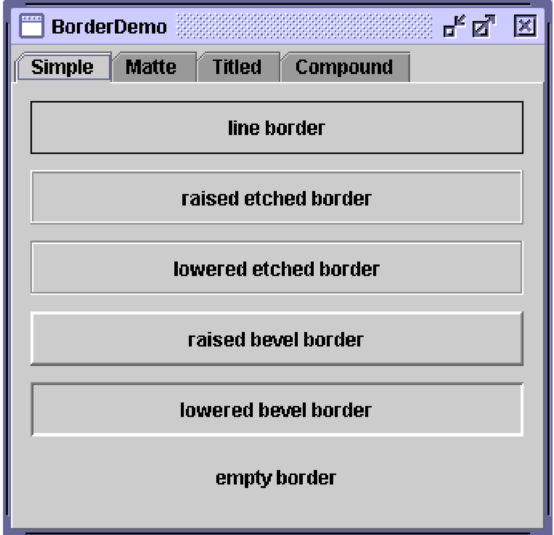
```
button.setToolTipText("Befehlsschalter");
```

Diese erscheinen in einem PopUp-Fenster, wenn der Mauszeiger über einer betroffenen Komponente verharrt:

Außerdem lässt sich über die Methode **setBorder()** ein Rahmen festlegen, wobei die Klasse **BorderFactory** mit ihren statischen Methoden diverse Modelle herstellen kann. Im Einführungsbeispiel verschaffen wir dem **JPanel**-Container `panli` etwas „Luft“:

```
panli.setBorder(BorderFactory.createEmptyBorder(30, 30, 30, 30));
```

Den Quellcode zu der folgenden Rahmen-Musterschau:



finden Sie auf der Webseite:

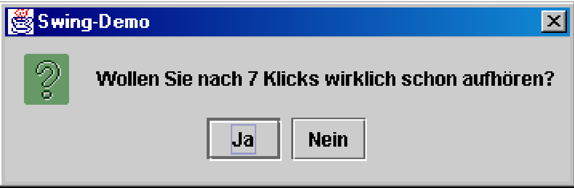
http://java.sun.com/docs/books/tutorial/uiswing/misc/example-1dot4/BorderDemo.java

11.4.4 Standarddialoge

Mit statischen Methoden der Klasse **JOptionPane** ist es sehr einfach, Standarddialoge zu erzeugen, um Nachrichten auszugeben oder Informationen abzufragen. Im Beispielprogramm wird mit der Methode **showConfirmDialog()** vom Benutzer eine Bestätigung erbeten:

```
if (JOptionPane.showConfirmDialog(e.getWindow(),
                 "Wollen Sie nach "+ numClicks +
                 " Klicks wirklich schon aufhören?",
                 titel, JOptionPane.YES_NO_OPTION) == 0)
  System.exit(0);
```

Passend zur gewählten **JOptionPane.YES_NO_OPTION** erscheint eine Dialogbox mit Fragezeichen-Dekoration und geeigneten Schaltflächen:



Als Rückgabewert liefert die Methode:

- -1 Nach einem Schließen der Dialogbox per Systemmenü etc.
- 0 Nach einem Klick auf **Ja**
- 1 Nach einem Klick auf **Nein**

Die oben verwendete **showConfirmDialog()**-Überladung kennt folgende Parameter:

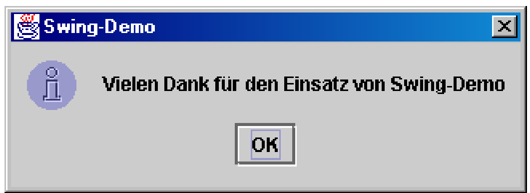
Typ	Name	Erläuterung
Component	parentComponent	Die Angabe einer Elternkomponente (an Stelle der Alternative **null**) hat z.B. zur Folge, dass die Dialogbox in der Nähe ihrer Elternkomponente erscheint. Im Beispiel wird das elterliche Rahmenfenster über die **WindowEvent**-Methode **getWindow()** ermittelt.
Object	message	die auszugebende Nachricht bzw. Frage

Typ	Name	Erläuterung
String	title	der Text für die Fensterzeile der Dialogbox
int	optionType	Legt fest, welche Schaltflächen am unteren Rand der Dialogbox erscheinen sollen. Mögliche Werte: • YES_NO_OPTION • YES_NO_CANCEL_OPTION • OK_CANCEL_OPTION

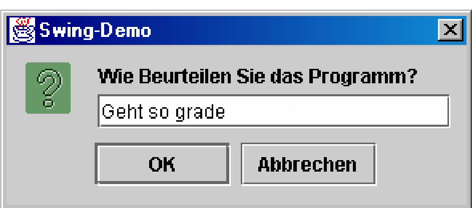
Die von **showConfirmDialog()** erzeugte Dialogbox verhält sich *modal*, blockiert also im geöffneten Zustand die Elterkomponente. Insbesondere endet ein **showConfirmDialog()**-Aufruf erst dann, wenn die zugehörige Dialogbox geschlossen wird.

Analog zu **showConfirmDialog()** können noch weitere **JOptionPane**-Methoden eingesetzt werden, die ebenfalls *modale* Dialoge präsentieren, z.B.:

- **showMessageDialog()**
  Informiert den Benutzer, z.B.:



- **showInputDialog()**
  Fordert zu einer Eingabe auf, z.B.:



Obwohl **JOptionPane** von **JComponent** abstammt (wie z.B. auch **JLabel** und **JButton**), erzeugt man in der Regel keine **JOptionPane**-Objekte, um sie in einen Container zu stecken, sondern bringt über **JOptionPane**-Klassenmethoden selbständige Dialogfenster auf den Bildschirm.

11.5 Die Layout-Manager der Container

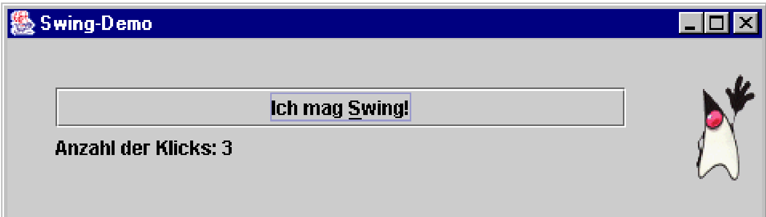
Eine **JPanel**-Komponente kann als Behälter für Swing-Komponenten dienen und dabei selbst in einem übergeordneten Container untergebracht werden, z.B. im Top-Level-Container des **JFrame**-Fensters. Im unserem Beispiel wird ein **JPanel**-Behälter namens `panli` verwendet:

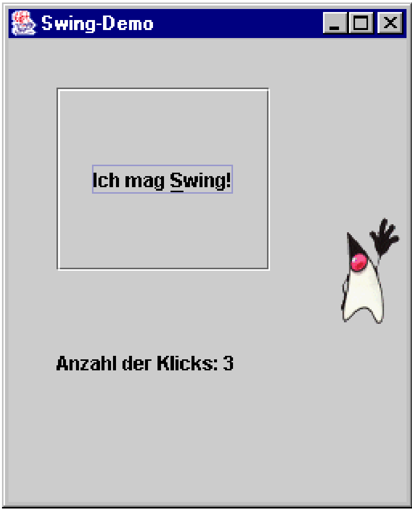
```
private JPanel panli;
   .    .    .
JPanel panli = new JPanel();
```

Um eine Komponente in einen Container aufzunehmen, benutzt man eine der zahlreichen **add()**-Überladungen, z.B.:

```
panli.add(button);
panli.add(label);
```

Die Entscheidung für eine **add()**-Variante hängt auch vom **Layout-Manager** ab, der für eine sinnvolle Anordnung der in einem Container enthaltenen Komponenten sorgt und diese Anordnung bei Größenänderungen sogar dynamisch angepasst, z.B.:





Dank Layout-Manager können sich Java-Programmierer auf die Logik ihrer Anwendung konzentrieren und müssen sich nicht um Positionen und Größen der einzelnen Steuerelemente kümmern. Durch Verschachteln von Containern, für die jeweils ein spezieller Layout-Manager engagiert werden kann, sollte sich fast jede Design-Idee verwirklichen lassen.

Selbstverständlich ist es aber auch möglich, alle Layout-Details pixel-genau festzulegen, und mit dem folgenden Aufruf der **JFrame**-Methode **setResizable()** kann man eine Änderungen der Fenstergröße durch den Benutzer verhindern:

```
setResizable(false);
```

11.5.1 BorderLayout

In unserem Beispielprogramm bleiben bei nahezu beliebigen Veränderungen des Anwendungsfensters (siehe oben) die folgenden räumlichen Relationen erhalten:

- Das Label mit dem Klickzähler steht senkecht unter dem Befehlsschalter.
- Das Bild erscheint stets rechts neben den beiden anderen Komponenten.

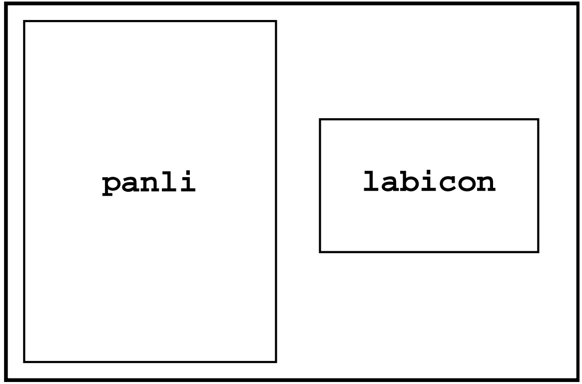
Eine wesentliche Voraussetzung für dieses Verhalten sind die beiden beteiligten **Container**.
Das Anwendungsfenster (Klasse `SwApp`, abgeleitet aus **JFrame**) ist ein (schwergewichtiger) Top-Level-Container und enthält (auf seiner content pane):

- den untergeordneten (leichtgewichtigen) Container `panli` aus der Klasse **JPanel**
- das Label `labicon`

Die beiden leichtgewichtigen Komponenten sind über die **add()**-Methode der **JFrame**–Inhaltsschicht (ein **Container**-Objekt, über die Referenzvariable `cont` ansprechbar) zu ihrem Standort gekommen:

```
cont.add(panli, BorderLayout.CENTER);
cont.add(labicon, BorderLayout.EAST);
```

Wir eben beschrieben, halten die beiden Komponenten `panli` und `labicon` folgende räumliche Anordnung ein:



Dafür sorgt der **Layout-Manager** der **JFrame**-Komponente `SwApp` (genauer: der zugehörigen contant pane). Weil wir die Voreinstellung nicht geändert haben, ist der `SwApp`-Layout-Manger ein Objekt aus der Klasse **BorderLayout**. Dies wird deutlich in den Positionierungsparametern der oben wiedergegebenen **add()**-Aufrufe:

- `panli` soll das Zentrum des `SwApp`-Containers besetzen, das sich mangels West-Komponente am linken Rand befindet.
- `labicon` hält sich am östlichen `SwApp`-Rand auf.

Welche Plätze ein **BorderLayout**-Manager insgesamt für einen Container verwalten kann, zeigt folgendes Beispielprogramm, das an jeder möglichen Position einen Befehlsschalter enthält:

Für die Abstände zwischen den Komponenten (horizontal und vertikal jeweils 5 Pixel) wurde durch folgenden Layout-Manager gesorgt:

```
private BorderLayout layout = new BorderLayout(5,5);
```

Welche Gestaltungsmöglichkeiten ein **BorderLayout** durch sein Verhalten bei unbesetzten Positionen und bei Veränderungen der Container-Fläche bietet, sollten Sie durch Probieren herausfinden.

11.5.2 GridLayout

Der `panli`-Container enthält seinerseits zwei Komponenten:

- `button`
- `label`

Für `panli` wurde mit der **JPanel**-Methode **setLayout()**, geerbt von der Klasse **java.awt.Container**, ein Layout-Manager aus der Klasse **GridLayout** engagiert, der im Allgemeinen eine ($z \times s$)-Komponentenmatrix verwalten kann, und im Beispiel dafür sorgt, dass alle `panli`-Komponenten bei linksbündiger Ausrichtung übereinander stehen. Dazu wird im Konstruktor *eine* Spalte und (über den Aktualparameter Null) eine unbestimmte Anzahl von Zeilen angekündigt:

```
panli.setLayout(new GridLayout(0, 1));
```

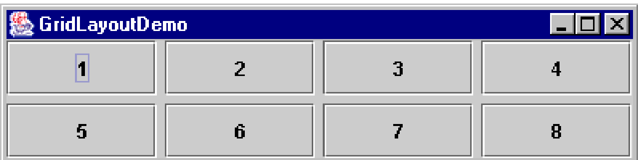
Ist ein solcher Layout-Manager einmal im Dienst, verteilt er die Komponenten automatisch auf die freien Plätze, so dass die **add()**-Methode ohne Platzanweisung auskommt:

```
panli.add(button);
panli.add(label);
```

In folgendem Beispiel wurde dem **JFrame**-Rahmenfenster über den Aufruf

```
cont.setLayout(new GridLayout(2,4,5,5));
```

ein (2 × 4)-**GridLayout** mit Zwischenabständen von jeweils 5 Pixeln verpasst, um 8 phantasielos beschriftete Schalter unterzubringen:



Der verfügbare Platz wird von einem **GridLayout** gleichmäßig auf alle Komponenten verteilt.

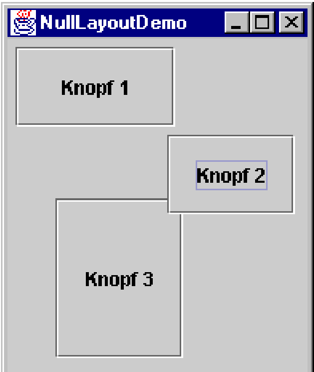
11.5.3 FlowLayout

Recht verbreitet ist das **FlowLayout**, das bei vielen Containern (z.B. **JPanel**) als Voreinstellung dient (bei Verzicht auf die explizite Spezifikation eines Layout-Managers per Konstruktor oder **set-Layout**()). Es ordnet die Komponenten nebeneinander an, bis ein „Zeilenumbruch" erforderlich wird:

An Stelle der horizontalen Zentrierung ist alternativ auch eine links- bzw. rechtsbündige Anordnung möglich, z.B.:

11.5.4 Freies Layout

Wer noch mehr Freiheit benötigt, kann auf einen Layout-Manager verzichten und alle Positionen bzw. Größen individuell festlegen, so dass z.B. sogar überlappende Komponenten möglich werden:



Ein freies Layout kann man folgendermaßen realisieren:

- Den voreingestellten Layout-Manager abschalten mit **setLayout(null)**

- Positionen und Größen der Komponenten mit **setBounds()** festlegen

Dies wird in folgendem Programmfragment demonstriert:

```
    .   .   .
    .   .   .
class NullLayoutDemo extends JFrame {
    private JButton k1, k2, k3;
    private Container cont = getContentPane();

    public NullLayoutDemo() {
        super("NullLayoutDemo");

        cont.setLayout(null);

        k1 = new JButton("Knopf 1");
        k1.setBounds(5,5,100,50);
        cont.add(k1);
        k2 = new JButton("Knopf 2");
        k2.setBounds(100,60,80,50);
        cont.add(k2);
        k3 = new JButton("Knopf 3");
        k3.setBounds(30,100,80,100);
        cont.add(k3);
    }
    .   .   .
    .   .   .
}
```

11.6 Ereignisbehandlung

Die Besonderheit einer Komponente im Unterschied zum gewöhnlichen Objekt besteht neben ihrem optischen Auftritt in der Fähigkeit, **Ereignisse** (z.B. Mausklicks) zu erkennen und an ein anderes Objekt weiterzuleiten, das dann durch Ausführen einer passenden Methode reagieren kann.

11.6.1 Das Delegationsmodell

Bei der seit Java 1.1 benutzten Ereignisbehandlung nach dem **Delegationsmodell** sind folgende Objekte beteiligt:

- **Ereignisquelle**
  Dies ist eine Komponente, die Ereignisse feststellen und weiterleiten kann. Im **Swing**-Demoprogramm, das wir seit Abschnitt 11.3 besprechen, kann der Befehlsschalter `button` diese Rolle spielen.
  Auslöser ist letztlich der Benutzer, von dessen Mausklick die Quellkomponente durch Vermittlung des Betriebssystems und des Java-Laufzeitsystems erfährt. Aus der Sicht des Programms ist es jedoch sinnvoll, den Befehlsschalter als Ereignisquelle aufzufassen.
  Neben dem Befehlsschalter enthält das Beispielprogramm noch eine zweite potentielle Ereignisquelle: das Rahmenfenster. Ein mögliches Ereignis ist hier z.B. der Mausklick auf das Symbol zum Schließen des Fensters.
  Eine Komponente kann von Ereignissen aus unterschiedlichen Klassen betroffen werden (z.B. **MouseEvent**, **ActionEvent**, **KeyEvent**).
- **Ereignisobjekt**
  Zu jedem Ereignistyp gehört in Java eine Ereignisklasse, über deren Methoden der anschließend vorzustellende Ereignisempfänger nähere Informationen über das Ereignis abfragen kann. Z.B. kann er nötigenfalls mit **getSource()** die Ereignisquelle in Erfahrung bringen, um

angemessen zu reagieren. Bei einem Mausereignis (siehe unten) lässt sich über **getX()** und **getY()** der Ort des Geschehens feststellen.

- **Ereignisempfänger**
  Ein Ereignis wird in der Regel *nicht* „direkt an der Quelle" behandelt. Statt dessen wird die Bearbeitung an ein anderes Objekt, den Ereignisempfänger (engl.: *event listener*) , *delegiert*. Zu jeder Ereignisklasse gehört ein *Interface*, das Kriterien für potentielle Ereignisempfänger vorschreibt. Eine Ereignisbearbeitung kann nur an Objekte einer Klasse delegiert werden, die das zugehörige Ereignis-Interface implementiert.
  Ereignisempfänger können in der Regel *verschiede* Ereignisse mit speziellen Methoden behandeln, die man auch *Event-Handler* nennt (siehe unten).

Diese Architektur mag auf den ersten Blick unnötig komplex erscheinen, hat aber z.B. dann ihre Vorteile, wenn in einem Programm mehrere Komponenten als Quelle für das selbe Ereignis fungieren. In diesem Fall wären getrennt agierende Ereignisbehandlungsmethoden unökonomisch und eventuell sogar unsicher.
Außerdem fördert eine Trennung von Benutzeroberfläche und Ereignisbehandlung die Wiederverwendbarkeit von Software.

11.6.2 Ereignistypen und Ereignisklassen

Potentielle Empfänger für ein **WindowEvent** müssen das Interface **WindowListener** implementieren, zu dem etliche Methoden gehören, wie die API-Dokumentation zeigt:

Method Summary	
void	**windowActivated**(WindowEvent e) Invoked when the window is set to be the user's active window, which means the window (or one of its subcomponents) will receive keyboard events.
void	**windowClosed**(WindowEvent e) Invoked when a window has been closed as the result of calling dispose on the window.
void	**windowClosing**(WindowEvent e) Invoked when the user attempts to close the window from the window's system menu.
void	**windowDeactivated**(WindowEvent e) Invoked when a window is no longer the user's active window, which means that keyboard events will no longer be delivered to the window or its subcomponents.
void	**windowDeiconified**(WindowEvent e) Invoked when a window is changed from a minimized to a normal state.
void	**windowIconified**(WindowEvent e) Invoked when a window is changed from a normal to a minimized state.
void	**windowOpened**(WindowEvent e) Invoked the first time a window is made visible.

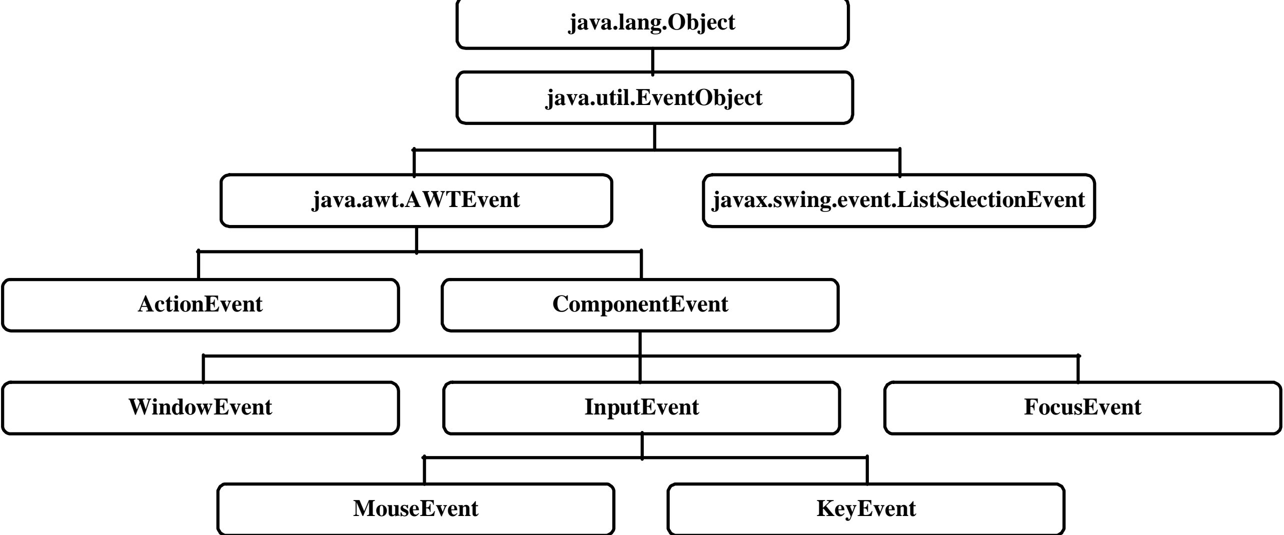
Aus der Anzahl und den Namen der Methoden wird klar, dass die Ereignisklasse **WindowEvent** bei *mehreren* Ereignissen zum Einsatz kommt, z.B. beim Schließen und Verkleinern eines Fensters. Die Java-Designer haben sich beim API-Design generell für eine überschaubare Anzahl von Ereignisklassen entschieden, die jeweils für mehrere, zusammen gehörige Ereignistypen zuständig sind.

Beim Erzeugen eines Ereignisobjektes wird der exakte Typ in der Instanzvariablen **id** festgehalten. Aus dieser Instanzvariablen erfährt die vom Ereignis betroffene Komponente (Ereignisquelle):

- Die Ereignisklasse
  Damit ist klar, an welchen Empfänger das Ereignis weitergeleitet werden muss.

- Die beim Empfänger aufzurufende Ereignisbehandlungsmethode

In der folgenden Abbildung sehen Sie die Abstammungsverhältnisse für die im Kurs behandelten Ereignisklassen:



Klassen ohne Paketangabe gehören zum Paket **java.awt.event**.

11.6.3 Ereignisempfänger registrieren

Besitzt ein Objekt die nötigen Voraussetzungen, kann es bei der Ereignisquelle durch einen ereignisklassenspezifischen Methodenaufruf als Empfänger **registriert** werden. Im **Swing**-Demoprogramm trägt die Ereignisquelle `SwApp` (eine **JFrame**-Komponente) für die Ereignisklasse **WindowEvent** mit der Methode **addWindowListener()** ein neu erzeugtes Objekt aus der Klasse `Ex` als Empfänger ein:

```
addWindowListener(new Ex());
```

Damit die Klasse `Ex` den Anforderungen an einen **WindowEvent**-Empfänger genügt, hat sie das Interface **WindowListener** (siehe oben) zu implementieren (direkt oder indirekt).

Zum Registrieren eines Ereignisempfängers sind jeweils spezielle Methoden zuständig, wodurch festgelegt ist, welche Ereignisklassen eine Komponente weiterleiten kann. So kann z.B. das Ereignis **ListSelectionEvent** (Benutzer wechselt in einer Liste den gewählten Eintrag) zwar von **javax.swing.JList**, nicht aber von **javax.swing.JButton**, in Umlauf gebracht werden, weil nur die zuerst genannte Komponente über die erforderliche Methode **addListSelectionListener()** verfügt.

In der folgenden Tabelle ist für einige Ereignisklassen festgehalten:

- zugrunde liegendes Benutzerverhalten
- vom Ereignisempfänger zu implementierendes Interface
  Hier ist festgelegt, welche Handlungskompetenzen (Methoden) eine Listener-Klasse besitzen muss.
- zuständige Empfänger-Registrierungsmethode

Außerdem sind die im nächsten Abschnitt zu beschreibenden *Adapterklassen* angegeben, die für viele Ereignisklassen zur Vereinfachung der Empfänger-Definition verfügbar sind:

Ereignisklasse	Mögliche Auslöser	Empfänger-Interface, zugeh. Adapter, Registrierungsmethode
ActionEvent	Benutzer klickt auf einen Befehlsschalter, drückt **<Enter>** in einem Textfeld oder wählt einen Menüeintrag.	ActionListener addActionListener()
ComponentEvent	Eine Komponente wird sichtbar.	ComponentListener ComponentAdapter addComponentListener()
WindowEvent	Benutzer schließt das Anwendungsfenster.	WindowListener WindowAdapter addWindowListener()
MouseEvent[23]	Benutzer klickt, während sich die Maus über einer Komponente befindet.	MouseListener MouseAdapter addMouseListener()
MouseEvent[1]	Benutzer bewegt die Maus über einer Komponente.	MouseMotionListener MouseMotionAdapter addMouseMotionListener()
KeyEvent	Benutzer drückt eine Taste.	KeyListener KeyAdapter addKeyListener()
FocusEvent	Eine Komponente erhält den Eingabefokus.	FocusListener FocusAdapter addFocusListener()
ListSelectionEvent	Benutzer wechselt in Liste den gewählten Eintrag.	ListSelectionListener addListSelectionListener()

Bei einer Ereignisquelle können zu einer Ereignisklasse auch *mehrere* Ereignisempfänger registriert werden, so dass Ereignisobjekte ggf. an mehrere Empfänger weitergeleitet werden.

11.6.4 Adapterklassen

In unserem Beispiel soll der **WindowEvent**-Empfänger zur Ereignisquelle `SwApp` nur aktiv werden, wenn ein Benutzer das Fenster *schließen* und damit die Anwendung beenden möchte. In diesem Fall wird ein **WindowEvent** mit bestimmter **Event-ID** erzeugt und eigentlich nur die **WindowListener**–Methode **windowClosing()** benötigt. Um in solchen Fällen überflüssigen Programmieraufwand zu vermeiden, hat man zu vielen Ereignis-Interfaces jeweils noch so genannten **Adapterklassen** eingeführt. Diese implementieren das zugehörige Interface mit *leeren* Methoden. Leitet man aus einer Adapterklasse ab, ist also das fragliche Interface erfüllt, und man muss nur die wirklich benötigten Methoden durch Überschreiben funktionstüchtig machen.
In unserem Beispiel wird die Ereignisempfänger-Klasse `Ex` aus der zum Interface **WindowListener** gehörigen Adapterklasse **WindowAdapter** abgeleitet:[24]

---

[23] Ein **MouseEvent** wird ggf. an einen registrierten **MouseListener** *und* an einen registrierten **MouseMotionListener** weitergeleitet.

[24] Dass bei der Klasse `Ex` der Zugriffsmodifikator **private** erlaubt ist, hängt mit einer Besonderheit zusammen, die in Abschnitt 11.6.6.1 besprochen wird.

```
private class Ex extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    if (JOptionPane.showConfirmDialog(e.getWindow(),
                "Wollen Sie nach "+ numClicks +
                " Klicks wirklich schon aufhören?",
                titel, JOptionPane.YES_NO_OPTION) == 0)
      System.exit(0);
  }
}
```

Der Event-Handler **windowClosing()** fragt nach, ob der Benutzer allen Ernstes aufhören möchte (vgl. Abschnitt 11.4.4). Erst nach einer Bestätigung dieser Absicht, wird das Programm beendet.

11.6.5 Schließen von Fenstern und GUI-Programmen

Man muss nicht unbedingt einen eigenen Ereignisempfänger definieren, wenn beim Schließen eines **JFrame**-Fensters lediglich die Anwendung beendet werden soll. Seit Java 1.3 kann man mit folgender **JFrame**-Methode für diese Standard-Behandlung sorgen:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Allerdings ist es nicht immer sinnvoll, ein Programm ohne weitere Maßnahmen zu beenden (z.B. ohne Prüfung auf ungesicherte Dokumente).

Ein **JFrame**-Fenster lässt sich auch dann per Fenstertitelzeilen-Symbol oder Systemmenü schließen, wenn man weder einen **WindowListener** registriert, noch die **setDefaultCloseOperation()**-Methode benutzt hat. Allerdings verschwindet dabei nur das *Fenster* vom Bildschirm, während die Anwendung nichts von diesem Ereignis erfährt und weiter läuft. War die Anwendung aus einem Konsolenfenster über das Werkzeug **java.exe** gestartet worden, erhält also der Benutzer *keine* neue Eingabeaufforderung. Dazu muss er die immer noch aktive Anwendung erst beenden, z.B. mit der Tastenkombination **<Strg>+<C>**.

Mit Hilfe eines **WindowEvent**-Handlers kann das Schließen eines **JFrame**-Fensters *nicht* verhindert werden, wir gewinnen vielmehr die Möglichkeit, auf dieses Ereignis zu reagieren.

Um einer **JFrame**-Komponente zu verbieten, auf Benutzerwunsch hin von der Bildfläche zu verschwinden, sendet man ihr folgende Botschaft zu:

```
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
```

Der gerade begonnene Exkurs zur Frage der **Terminierung von GUI-Anwendungen** soll noch etwas fortgeführt werden. Im Beispielprogramm beschränkt sich die **main()**-Methode darauf, ein Objekt aus der Klasse `SwApp` zu erzeugen, und endet folglich mit der Rückkehr des Konstruktor-Aufrufs:

```
public static void main(String[] args) {
    SwApp swapp = new SwApp();
}
```

Während unsere *Konsolen*-Programme nach dem Verlassen der **main()**-Methode beendet waren, geht es beim Swing-Demoprogramm zu diesem Zeitpunkt erst richtig los. Wie lange der Spaß andauert, hängt vom Verhalten des Benutzers und von den Ereignis-Behandlungsmethoden ab.

Um dieses, mit unseren bisherigen Vorstellungen unvereinbare, Verhalten erklären zu können, müssen wir das Konzept der **Threads** einbeziehen, das in Java eine große Rolle spielt und später noch ausführlicher zur Sprache kommt. Ein Programm (ein Prozess) kann in mehrere *nebenläufige Ausführungsfäden* zerlegt werden, was speziell bei Java-Programmen mit GUI automatisch geschieht.

Nachdem Sie ein Java-Programm aus einer Konsole gestartet haben, können Sie unter Windows mit der Tastenkombination **<Strg>+<Unterbr>** eine Liste seiner aktiven Threads anfordern.

Ein Java-Programm (ob mit oder ohne GUI) endet genau dann, wenn eine der folgenden Bedingungen eintritt:

- Alle Benutzer-Threads sind abgeschlossen.
  Neben den Benutzer-Threads kennt Java noch so genannte *Deamon*-Threads, die ein Programm *nicht* am Leben erhalten können.
- Ein Thread ruft die Methode **System.exit()** oder die Methode **Runtime.exit()** auf, und der **Security Manager**[25] hat nichts dagegen.

Während ein Konsolenprogramm nur *einen* Benutzer-Thread besitzt (namens **main**), tauchen bei GUI-Programmen *zusätzliche* Benutzer-Threads auf, z.B. **AWT-EventQueue-0**.
Sobald in der **main()**-Methode unseres Beispielprogramms das Anwendungsfenster (ein **JFrame**-Abkömmling) erzeugt wird, starten die GUI-Threads. Während anschließend mit der Methode **main()** auch der Thread **main** endet, leben die GUI-Threads weiter.

Dort befindet sich auch eine Referenz auf das Anwendungsfenster, so dass der Garbage Collector beim Verlassen der Methode **main()** mit ihrer lokalen `SwApp`-Referenzvariablen *keinen* Anlass hat, das Anwendungsfenster zu beseitigen. Die lokale `SwApp`-Referenzvariable ist sogar überflüssig, wie die folgende Variante der **main()**-Methode zeigt:

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

Um ein GUI-Programm per Anweisung zu beenden, muss die Methode **System.exit()** aufgerufen werden, was aber z.B. eine **JFrame**-Komponente aufgrund des oben vorgestellten Methodenaufrufs

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

automatisch erledigen kann.

11.6.6 Optionen zur Definition von Ereignisempfängern

In diesem Abschnitt lernen Sie u.a. zwei Prinzipien zur Klassendefinition kennen, die nicht nur bei der Ereignisbehandlung eingesetzt werden können: innere und anonyme Klassen.

11.6.6.1 Innere Klassen als Ereignisempfänger

Wer das vollständige Quellprogramm aufmerksam liest, wird feststellen, dass die `Ex`–Klassendefinition *innerhalb* der `SwApp`-Klassendefinition steht, was `Ex` zur **inneren Klasse** macht. Auf diese Weise könnten die `SwApp`-Instanzvariablen in den `Ex`-Methoden angesprochen werden, was im Beispiel durch den Zugriff auf `numClicks` demonstriert wird.
Trotz des nicht ganz überzeugenden Beispiels können Sie sich bestimmt vorstellen, welchen Nutzen innere Klassen gerade beim Definieren eines Ereignisempfängers haben.

Einige Eigenschaften von inneren Klassen:

[25] Diesen zentralen Bestandteil der Java-Sicherheitsarchitektur können wir aus Zeitgründen nicht behandeln.

- In den Methoden der inneren Klasse kann man auf die Variablen und Methoden der äußeren Klasse zugreifen.
- Der Compiler erzeugt auch für die innere Klasse eine eigene **class**-Datei, in deren Namen die Bezeichner für die innere und die umgebende Klasse eingehen, so dass im Beispiel der Name **SwApp$Ex.class** resultiert.
- Innere Klassen dürfen geschachtelt werden.
- Während für die bisher gewohnten (äußeren) Klassen nur der Zugriffsmodifikator **private** erlaubt ist, können bei inneren Klassen dieselben Zugriffsmodifikatoren verwenden werden wie bei Instanzvariablen bzw. –methoden:

Zugriffsmodifikator	Die innere Klasse ist sichtbar für ...
ohne	andere Klassen im selben Paket
private	nur für die Rahmenklasse
protected	aus der Rahmenklasse abgeleitete Klassen
public	beliebige Klassen

11.6.6.2 Anonyme Klassen als Ereignisempfänger

Nun haben wir unser Beispielprogramm fast vollständig durchleuchtet mit Ausnahme der zentralen Ereignisbehandlung:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        numClicks++;
        label.setText(labelPrefix + numClicks);
        if (iconr < maxicon-1)
           iconr++;
        else
           iconr = 0;
        labicon.setIcon(icon[iconr]);
    }
});
```

Hier registriert die Ereignisquelle `button` für das **ActionEvent** einen Empfänger, wobei nicht nur das Objekt dynamisch erzeugt, sondern die gesamte Klasse ad hoc definiert wird.
Wie bei den inneren Klassen haben wir auch bei einer solchen **anonymen Klasse** die Möglichkeit, in den Ereignis-Behandlungsmethoden auf Instanzvariablen der „Rahmenklasse" zuzugreifen, was in diesem Fall außerordentlich hilfreich ist.

Einige Eigenschaften von anonymen Klassen:

- Definition und Instantiierung finden in einem **new**-Operanden statt, wobei an Stelle eines Konstruktors zu einer (konkreten) Klasse das von der anonymen Klasse implementierte Interface oder die erweiterte Basisklasse auftritt. Dann folgt ein Klassendefinitions-Block, der im Beispiel nur die überschriebene Methode **actionPerformed()** enthält.
- Es kann nur eine einzige Instanz erzeugt werden.
  Werden mehrere Instanzen benötigt, ist die innere Klasse (siehe oben) vorzuziehen.
- Es sind keine Konstruktoren, keine statischen Methoden und keine statischen Variablen erlaubt.
- Das eine Objekt der anonymen Klasse ist auf das Protokoll der Basisklasse bzw. der implementierten Schnittstelle beschränkt, so dass die Definition zusätzlicher **public**-Methoden sinnlos ist.
- Der Compiler erzeugt auch für die anonyme Klasse eine eigene **class**-Datei, in deren Namen der Bezeichner für die umgebende Klasse eingeht, so dass im Beispiel der Name **SwApp$1.class** resultiert.

11.6.6.3 Do-It-Yourself – Ereignisbehandlung

Zur Behandlung der von Instanz-Komponenten oder vom Rahmenfenster verschickten Ereignisse muss eine von **JFrame** abstammende Klasse nicht unbedingt *Fremd*klassen beauftragen, sondern kann den Job auch selbst übernehmen, sofern sie die erforderlichen Ereignis-Interfaces erfüllt. Dies wird gleich an einem Beispielprogramm zum Umgang mit Maus-Ereignissen demonstriert.
In der Regel ist allerdings wegen der einfacheren Code-Wiederverwendung eine Trennung von GUI und Ereignisbehandlung durch Definition spezialisierter Klassen zu bevorzugen.

11.6.7 Tastatur- und Mausereignisse

11.6.7.1 KeyEvent

Im folgendem Beispiel wird der Umgang mit dem **KeyEvent** sowie dem zugehörigen Interface **KeyListener** bzw. der Klasse **KeyAdapter** demonstriert. Es kommt eine anonyme Klasse zum Einsatz, die unter Angabe der erweiterten Basisklasse definiert wird:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class KeyListenerDemo extends JFrame {
    private JLabel keyCode, uChar;

    public KeyListenerDemo() {
        super("KeyListener-Demo");

        keyCode = new JLabel("KeyCode:");
        uChar = new JLabel("Unicode-Zeichen:");
        getContentPane().add(keyCode, BorderLayout.NORTH);
        getContentPane().add(uChar, BorderLayout.SOUTH);

        addKeyListener(new KeyAdapter() {
           public void keyPressed(KeyEvent e) {
              keyCode.setText("KeyCode: "+String.valueOf(e.getKeyCode()));
           }
           public void keyTyped(KeyEvent e) {
              uChar.setText("Unicode-Zeichen: "+String.valueOf(e.getKeyChar()));
           }
           public void keyReleased(KeyEvent e) {
              keyCode.setText("KeyCode: ");
              uChar.setText("Unicode-Zeichen: ");
           }
        });

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(200, 70);
        show();
    }

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

Das registrierte **KeyListener**-Objekt kann die bei aktivem **JFrame**-Rahmenfenster auftretenden Tastendrücke verarbeiten, wobei es sich auf das Protokollieren beschränkt:

Bei aktivierter Zielkomponente löst ein Tastendruck das (teilweise plattformabhängige) low-level-Ereignis **KEY_PRESSED** aus und bewirkt einen Aufruf der **KeyListener**-Methode **keyPressed()**, die ein **KeyEvent**-Objekt als Parameter erhält. Analog kommt das KEY_RELEASED-Ereignis zustande.
Eine Taste(nkombination) führt hingegen nur dann zu einem Ereignis vom Typ **KEY_TYPED** und damit zu einem Aufruf der **KeyListener**-Methode **keyTyped()**, wenn ihr ein Unicode-Zeichen entspricht.

11.6.7.2 MouseEvent

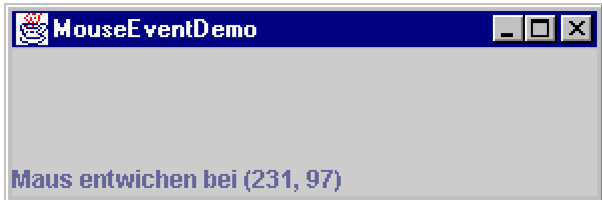
Im folgendem Beispiel wird der Umgang mit dem **MouseEvent** sowie dem zugehörigen Interface **MouseListener** vorgeführt. Außerdem wird gezeigt (wie in Abschnitt 11.6.6.3 angekündigt), dass eine von **JFrame** abstammende Klasse nicht unbedingt *Fremd*klassen mit der Ereignisbehandlung beauftragen muss, sondern den Job auch selbst übernehmen kann, sofern sie die erforderlichen Ereignis-Interfaces erfüllt:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MouseEventDemo extends JFrame implements MouseListener, MouseMotionListener {
   private JLabel statusZeile = new JLabel();

   public MouseEventDemo() {
      super("MouseEventDemo");
      getContentPane().add(statusZeile, BorderLayout.SOUTH);

      addMouseListener(this);
      addMouseMotionListener(this);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 100);
      show();
   }
   public void mouseClicked(MouseEvent e) {
      statusZeile.setText("Mausklick bei ("+e.getX()+", "+e.getY()+")");
   }
   public void mousePressed(MouseEvent e) {
      statusZeile.setText("Maustaste gedrückt bei ("+e.getX()+", "+e.getY()+")");
   }
   public void mouseReleased(MouseEvent e) {
      statusZeile.setText("Maustaste losgelassen bei ("+e.getX()+", "+e.getY()+")");
   }
   public void mouseEntered(MouseEvent e) {
      statusZeile.setText("Maus eingedrungen bei ("+e.getX()+", "+e.getY()+")");
   }
   public void mouseExited(MouseEvent e) {
      statusZeile.setText("Maus entwichen bei ("+e.getX()+", "+e.getY()+")");
   }
   public void mouseDragged(MouseEvent e) {
      statusZeile.setText("Maus gezogen bei ("+e.getX()+", "+e.getY()+")");
   }
   public void mouseMoved(MouseEvent e) {
      statusZeile.setText("Maus bewegt bei ("+e.getX()+", "+e.getY()+")");
   }

   public static void main(String[] arg) {
      new MouseEventDemo();
   }
}
```

Beim Registrieren der Ereignisempfänger gibt ein Objekt der Klasse `MouseEventDemo` als Empfänger mit dem Schlüsselwort **this** sich selbst an:

```
addMouseListener(this);
addMouseMotionListener(this);
```

Mit dem Programm lassen sich diverse Maus-Ereignisse beobachten, z.B.:



11.7 Swing-Komponenten, Teil 2

In diesem Abschnitt werden einige weitere Swing-Komponenten vorgestellt. Eine Darstellung *aller* Komponenten verbietet sich aus Platzgründen und ist auch nicht erforderlich, weil Sie bei Bedarf in der API-Dokumentation und im Java-Tutorial (SUN 2002) alle benötigten Informationen finden.

11.7.1 Einzeilige Text-Komponenten

Im Rahmen der **JOptionPane**-Klassenmethode **showInputDialog()** konnten Sie schon eine einzeilige Textkomponente besichtigen. In einem **JFrame**-Fenster kann dieses elementare Bedienelement mit Hilfe einer **JTextField**-Komponente implementiert werden, z.B.:

Sobald der Benutzer die **Enter**-Taste drückt, während die **JTextField**-Komponente den Eingabe-Fokus hat, wird ein **ActionEvent** ausgelöst. Im Beispiel präsentiert der Event-Handler daraufhin eine Message-Box mit dem erfassten Text, den er über die **JTextField**-Methode **getText()** ermittelt hat:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JTextFieldDemo extends JFrame {
   private JTextField name;
   private JLabel label = new JLabel("Wie ist Ihr Name? ");
   private final static String titel = "JTextFieldDemo";

   public JTextFieldDemo() {
      super(titel);
      Container cont = getContentPane();
      cont.setLayout(new FlowLayout());

      name = new JTextField(40);
      name.addActionListener(
         new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               JOptionPane.showMessageDialog(((JComponent) e.getSource()).getParent(),
                "Sie heißen "+name.getText(), titel, JOptionPane.INFORMATION_MESSAGE);
            }
```

```
         });

      cont.add(label);
      cont.add(name);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(500, 90);
      setVisible(true);
   }

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

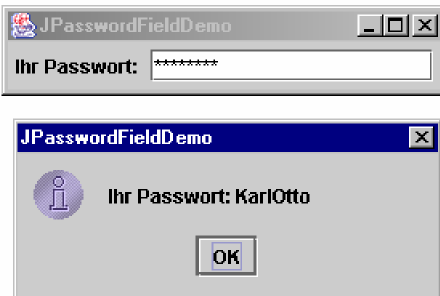
Statt der voreingestellten linksbündigen Ausrichtung der Textfeld-Inhalte kann mit der **JTextField**-Methode **setHorizontalAlignment()** auch eine zentrierte oder rechtsbündige Ausrichtung gewählt werden, z.B.:

```
textFeld.setHorizontalAlignment(JTextField.RIGHT);
```

Rechtsbündige Textfelder sind z.B. bei der Erfassung von Zahlen zu bevorzugen.

Mit **setEditable(false)** wird für eine **JTextField**-Komponente festgelegt, dass sie vom Benutzer nicht geändert werden darf. Sie wird dann benutzt wie eine **JLabel**-Komponente, unterscheidet sich von dieser jedoch durch das Design.

Zum Erfassen von Passwörtern steht in Java die Komponente **JPasswordField** bereit, die im Unterschied zu **JTextField** für jedes eingegebene Zeichen ein Sternchen anzeigt, z.B.:



Das erfasste Passwort kann mit der **JPasswordField**-Methode **getPassword()** als **char**-Array extrahiert werden, was im Event-Handler des Beispielprogramms zur Weiterverwendung in einem **String**-Konstruktor geschieht:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JPasswordFieldDemo extends JFrame {
   private JPasswordField pw;
   private JLabel label = new JLabel("Ihr Passwort: ");
   private final static String titel = "JPasswordFieldDemo";

   public JPasswordFieldDemo() {
      super(titel);
      Container cont = getContentPane();
      cont.setLayout(new FlowLayout());

      pw = new JPasswordField(16);
      pw.addActionListener(
         new ActionListener() {
               public void actionPerformed(ActionEvent e) {
                  JOptionPane.showMessageDialog(((JComponent) e.getSource()).getParent(),
                   "Ihr Passwort: " + new String(pw.getPassword()), titel,
                   JOptionPane.INFORMATION_MESSAGE);
               }
```

```
        });

      cont.add(label);
      cont.add(pw);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      pack();
      setVisible(true);
   }

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

Ansonsten bringt das letzte Beispielprogramm noch einen Nachtrag zur **JFrame**-Rahmenfensterkomponente: Statt wie in den bisherigen Beispielen die initiale Fenstergröße mit **setSize()** festzulegen, wird das Fenster über die (von **java.awt.Window** geerbte) Methode **pack()** aufgefordert, es möge seine Größe passend zu den enthaltenen Komponenten einrichten.

11.7.2 Kontrollkästchen und Optionsfelder

In diesem Abschnitt werden zwei Umschalter vorgestellt:

- Für **Kontrollkästchen** steht die Swing-Komponente **JCheckBox** zur Verfügung.
- Für ein **Optionsfeld** verwendet man Komponenten vom Typ **JRadioButton**.

In folgendem Programm kann für den Text einer **JLabel**-Komponente über zwei Kontrollkästchen der Schriftschnitt und über ein Optionsfeld die Schriftart gewählt werden:

Die beiden Kontrollkästchen (`bold` und `italic` genannt) werden aus Layout-Gründen in einem eigenen **JPanel**-Container untergebracht, der seinerseits am linken Rand des **JFrame**-Fensters sitzt:

```
   private JCheckBox bold, italic;
      . . .
   bold = new JCheckBox("Fett");
   italic = new JCheckBox("Kursiv");
   cbPanel.add(bold);
   cbPanel.add(italic);

   CheckHandler cbHandler = new CheckHandler();
   bold.addItemListener(cbHandler);
   italic.addItemListener(cbHandler);
```

Als **ActionEvent**-Empfänger wird für beide Kontrollkästchen dasselbe Objekt aus der internen Klasse `CheckHandler` verwendet, welche das Interface **ItemListener** implementiert. Dieses Interface beschränkt sich auf die Methode **itemStateChanged()**, die bei jedem Statuswechsel einer **JCheckBox** aufgerufen wird:

```
class CheckHandler implements ItemListener {
  public void itemStateChanged(ItemEvent e) {
    Font oldFont = bsp.getFont(), newFont = null;
    int oldStyle = oldFont.getStyle(), newStyle = 0;

    if (e.getSource() == bold) {
      if (e.getStateChange() == ItemEvent.SELECTED) {
        newStyle = oldStyle + Font.BOLD;
      } else {
        newStyle = oldStyle - Font.BOLD;
      }
    } else if (e.getSource() == italic) {
      if (e.getStateChange() == ItemEvent.SELECTED) {
        newStyle = oldStyle + Font.ITALIC;
      } else {
        newStyle = oldStyle - Font.ITALIC;
      }
    }

    newFont = oldFont.deriveFont(newStyle, 16);
    bsp.setFont(newFont);
  }
}
```

Der **ActionEvent**-Handler stellt mit **getSource()** die Ereignisquelle fest und passt sein Verhalten an. Er verwendet dabei einige Methoden zum Umgang mit Schriftarten, die in einem Java-Programm als Objekte der Klasse **Font** vertreten sind:

- Die **Component**-Methode **getFont()** stellt die Schriftart einer Komponente fest.
- Die **Font**-Methode **getStyle()** ermittelt den Stil (Schnitt) einer Schriftart, wobei **PLAIN**, **BOLD**, **ITALIC** und **BOLD+ITALIC** in Frage kommen.
- Durch die **Font**-Methode **deriveFont()** wird ein **Font**-Objekt durch eine Variante mit neuem Stil und neuer Größe ersetzt.
- Mit der **JComponent**-Methode **setFont()** wird die Schriftart einer Komponente geändert.

Ob das Quell-Kontrollkästchen ein- oder ausgeschaltet wurde, ermittelt der Event-Handler mit der **ItemEvent**-Methode **getStateChange()**.

Auch die drei Optionsschalter haben einen gemeinsamen **ItemListener**. Zudem sorgt ein Objekt aus der Klasse **ButtonGroup** dafür, dass stets nur ein Gruppenmitglied den Status **true** besitzt:

```
private JRadioButton arial, tiro, courier;
. . .
courier = new JRadioButton("Courier", false);
tiro = new JRadioButton("Times Roman", true);
arial = new JRadioButton("Arial", false);
rbPanel.add(courier);
rbPanel.add(tiro);
rbPanel.add(arial);

rbGroup = new ButtonGroup();
  rbGroup.add(courier);
rbGroup.add(tiro);
rbGroup.add(arial);

RadioHandler rbHandler = new RadioHandler();
courier.addItemListener(rbHandler);
tiro.addItemListener(rbHandler);
arial.addItemListener(rbHandler);
```

Auch im **ItemEvent**-Handler der Optionsschalter kommen die oben vorgestellten Schriftartenverwaltungs-Methoden zum Einsatz:

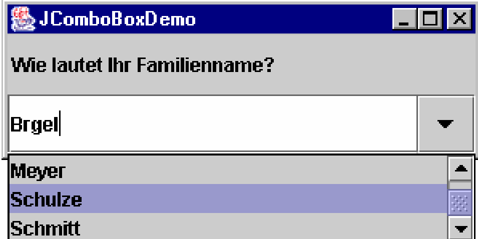
```
    class RadioHandler implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
        int style = bsp.getFont().getStyle();
        Font oldFont = null, newFont;

        if (e.getSource() == courier) {
          oldFont = courierFont;
        } else if (e.getSource() == tiro) {
          oldFont = tiroFont;
        } else if (e.getSource() == arial) {
          oldFont = arialFont;
        }

        newFont = oldFont.deriveFont(style, 16);
        bsp.setFont(newFont);
      }
    }
```

11.7.3 Kombinationsfelder

Die **JComboBox**-Komponente bietet eine Kombination aus einem einzeiligen Textfeld und einer Liste, von der normalerweise nur ein Element sichtbar ist. Um eine Wahl zu treffen, hat der Benutzer zwei Möglichkeiten:

- den Namen der gewünschten Option eintragen und mit **Enter** quittieren
- die versteckte Liste aufklappen und die gewünschte Option markieren

In folgendem Programm wird die Frage nach dem Familiennamen durch eine Liste mit den häufigsten Namen erleichtert:



Per Voreinstellung funktioniert die **JComboBox** als reine DropDown-Liste, erlaubt also kein Editieren der Listeneinträge. Dies wird im Beispiel mit dem Methodenaufruf **setEditable(true)** geändert:

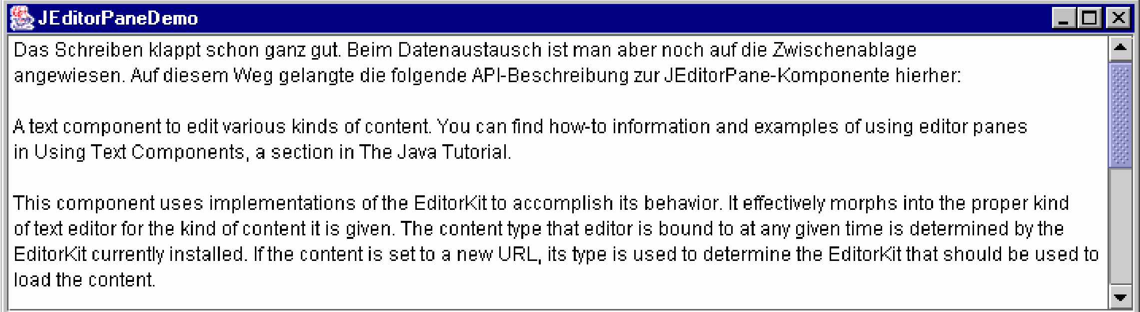
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class JComboBoxDemo extends JFrame {
 private JComboBox familienName;
 private JLabel prompt;
 private final String[] auswahl = {"Müller", "Meyer", "Schulze", "Schmitt"};
 private final static String titel = "JComboBoxDemo";

 public JComboBoxDemo() {
  super(titel);
  Container cont = getContentPane();
  cont.setLayout(new GridLayout(0,1));

  prompt = new JLabel(" Wie lautet Ihr Familienname?");
  familienName = new JComboBox(auswahl);
  familienName.setMaximumRowCount(3);
  familienName.setEditable(true);
```

```
   familienName.addItemListener(
    new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
       if(e.getStateChange() == ItemEvent.SELECTED) {
        JComboBox cb = (JComboBox) e.getSource();
        JOptionPane.showMessageDialog(cb.getParent(),
          "Sie heißen "+cb.getSelectedItem(), titel, JOptionPane.INFORMATION_MESSAGE);
       }
      }
     });

   cont.add(prompt);
   cont.add(familienName);

   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setSize(300, 90);
   setVisible(true);
  }

  public static void main(String[] args) {
   JComboBoxDemo prog = new JComboBoxDemo();
  }
}
```

Ein **ItemEvent**-Handler wird bei jeder Selektion *und* bei jeder Deselektion aufgerufen. Im Beispiel wird mit Hilfe der **ItemEvent**-Methode **getStateChange()** dafür gesorgt, dass die Messagebox nur erscheint, wenn eine neue Wahl vorgenommen wurde.

11.7.4 Ein (fast) kompletter Editor als Swing-Komponente

Die Swing-Komponente **JTextField** hat noch eine große Schwester namens **JEditorPane**, die einen recht brauchbaren Texteditor implementiert:



Für die Rohversion des Editors, die immerhin z.B. schon den Datenaustausch via Zwischenablage beherrscht, muss man sehr wenig Aufwand betreiben:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Editor extends JFrame{
   private JScrollPane roll;
   private JEditorPane text;

   public Editor() {
      super("JEditorPaneDemo");
      text = new JEditorPane();
      roll = new JScrollPane(text);
      getContentPane().add(roll, BorderLayout.CENTER);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 200);
      setVisible(true);
   }
```

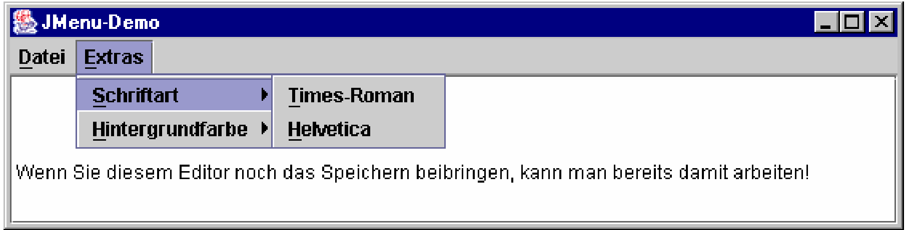
```
  public static void main(String[] arg) {
    new Editor();
  }
}
```

Zu erwähnen ist noch, dass die **JEditorPane**-Komponente Rollbalken erhält, indem sie in eine in eine **JScrollPane**-Komponente eingepackt wird.

Im nächsten Abschnitt werden wir den Editor um ein Menü mit wichtigen Funktionen erweitern.

11.7.5 Menüs

Um das Potential der **JEditorPane**-Komponente besser auszuschöpfen, erweitern wir das in Abschnitt 11.7.4 begonnene Programm um ein Menü:



Als **Menüzeile** verwenden wir einen Spezial-Container der Klasse **JMenuBar**, der im Norden der **ContentPane** unseres **JFrame**-Fensters untergebracht wird:

```
private JMenuBar menueZeile;
   .  .  .
menueZeile = new JMenuBar();
getContentPane().add(menueZeile, BorderLayout.NORTH);
```

Die Menüs werden als Komponenten der Klasse **JMenu** erstellt, bei Bedarf mit einem Tastenkürzel versehen und dann auf der Menüzeile untergebracht, wobei man sich auf den voreingestellten Layout-Manager dieses Spezial-Containers verlassen kann, z.B.:

```
private JMenu fileMenue, extrasMenue, fontMenue, farbMenue;
   .  .  .
extrasMenue = new JMenu("Extras");
extrasMenue.setMnemonic('E');
menueZeile.add(extrasMenue);
```

Beim Erzeugen der **JMenu**-Komponenten wird noch nicht zwischen Menüs und Untermenüs unterschieden. Eine **JMenu**-Komponente kann über ihre **add**()-Methode andere **JMenu**-Komponenten als Untermenüs aufnehmen, z.B.:

```
fontMenue = new JMenu("Schriftart");
fontMenue.setMnemonic('S');
extrasMenue.add(fontMenue);
```

Menüeinträge, die keine Untermenüs darstellen, werden über Komponenten der Klasse **JMenuItem** realisiert. Diese Klasse stammt von **AbstractButton** ab und löst **ActionEvents** aus.
Jede **JMenu**-Komponente (ob Haupt- oder Untermenü) kann mit ihrer **add**()-Methode **Menüitems** aufnehmen, z.B.:

```
JMenuItem timesItem;
   .  .  .
timesItem = new JMenuItem("Times-Roman");
timesItem.setMnemonic('T');
fontMenue.add(timesItem);
```

Für jedes Menüitem wird ein **ActionListener** registriert, z.B.:

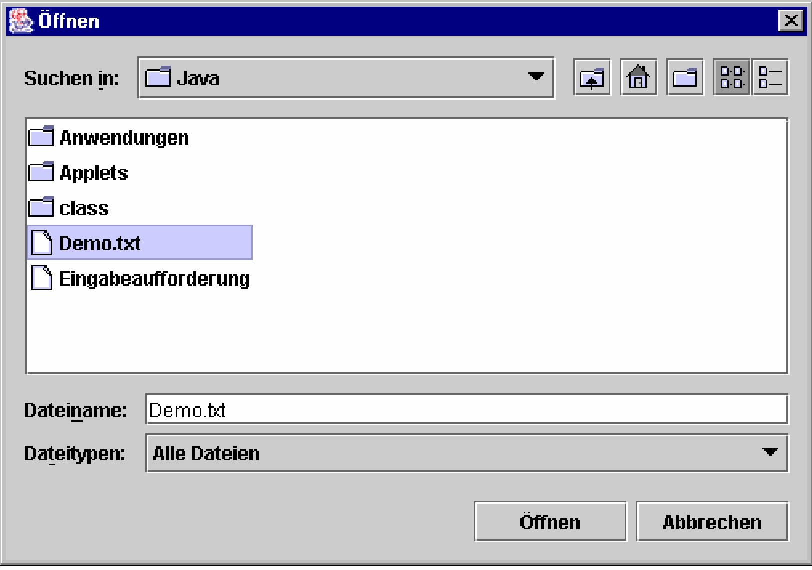
```
openItem.addActionListener(new
   ActionListener() {
      public void actionPerformed(ActionEvent e) {
         openFile();
         if (datei != null) readText();
      }
});
```

11.7.6 Dateiauswahldialog

Im eben wiedergegebenen **ActionEvent**-Handler zum Menüitem **Datei > Öffnen** unseres Editor-Programms wird eine anwendungsinterne Methode `openFile()` aufgerufen, die für den Standarddialog zum Öffnen einer Datei eine Komponente der Klasse **JFileChooser** einspannt:

```
private void openFile() {
   JFileChooser fc = new JFileChooser();
   int returnCode = fc.showOpenDialog(this);
   if (returnCode == JFileChooser.APPROVE_OPTION)
      datei = fc.getSelectedFile();
   else
      datei = null;
}
```

Das Ergebnis kann sich sehen lassen:

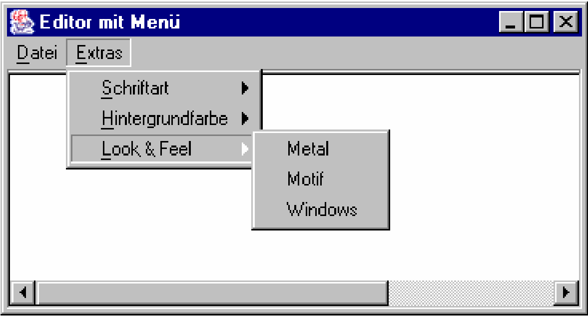


11.8 Look & Feel umschalten

Wer den in Abschnitt 11.7.6 gezeigten Dateiauswahldialog lieber im aktuellen Windows-Design erleben möchte, der kann die Swing-Option nutzen, das Look & Feel einer Anwendung zwischen folgenden Alternativen umzuschalten:

- **Metal** (Java-Standard)
- **Motif** (ein verbreiteter X-11 – Window-Manager unter UNIX)
- **Windows**

Bevor man ein solches Look & Feel - Menü präsentieren kann, ist etwas Arbeit angesagt.



Neben dem `lookAndFeel`-Menü und den Menü-Items (`metalItem`, `motifItem` und `windowsItem`) wird ein Array mit Elementen der Klasse **LookAndFeelInfo** deklariert, die in der Klasse **UIManager** geschachtelt ist, so dass wir im Kurs erstmals zwischen zwei Klassennamen den Punktoperator verwenden:

```
private JMenu lookAndFeel;
private JMenuItem metalItem, motifItem, windowsItem;
private UIManager.LookAndFeelInfo lafs[];
```

Den Wert für die Array-Referenzvariable `lafs` liefert die **UIManager**-Methode **getInstalledLookAndFeels()**:

```
lafs = UIManager.getInstalledLookAndFeels();
```

Als Ereignisempfänger für die Look & Feel – Menüitems wird folgende interne Klasse definiert:

```
class UIChooser implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == metalItem) {
      selUI(0);
    } else if (e.getSource() == motifItem) {
      selUI(1);
    } else if (e.getSource() == windowsItem){
      selUI(2);
    }
  }
}
```

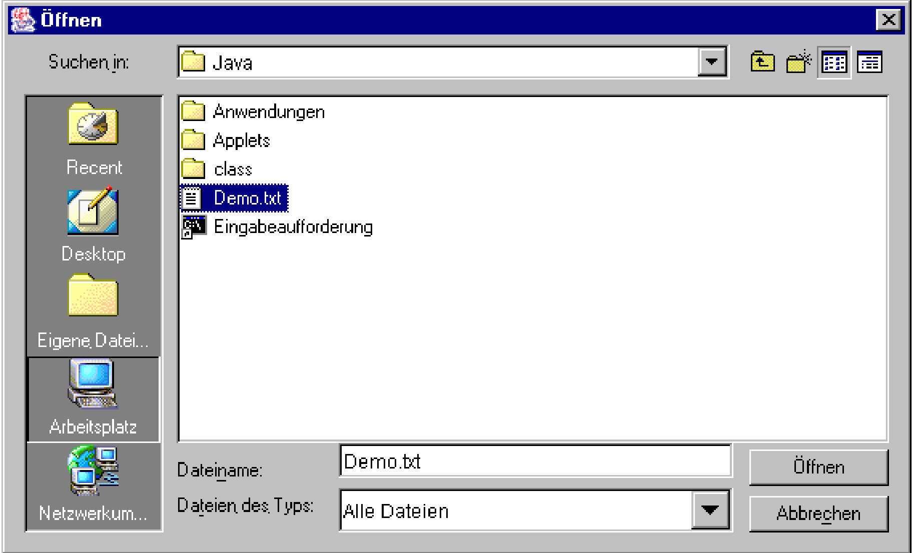
Im Konstruktor der Anwendungsklasse wird ein Objekt aus dieser Klasse erzeugt, das die Ereignisbehandlung für alle Look & Feel – Menüitems übernimmt:

```
UIChooser ui = new UIChooser();

lookAndFeel = new JMenu("Look & Feel");
lookAndFeel.setMnemonic('L');
extrasMenue.add(lookAndFeel);
metalItem = new JMenuItem("Metal");
metalItem.addActionListener(ui);
motifItem = new JMenuItem("Motif");
motifItem.addActionListener(ui);
windowsItem = new JMenuItem("Windows");
windowsItem.addActionListener(ui);
lookAndFeel.add(metalItem);
lookAndFeel.add(motifItem);
lookAndFeel.add(windowsItem);
```

Nachzutragen ist noch die im **ActionEvent**-Handler benutzte Methode `selUI()`:

```
private void selUI(int ui) {
   try {
     UIManager.setLookAndFeel(lafs[ui].getClassName());
     SwingUtilities.updateComponentTreeUI(this);
   } catch (Exception e) {}
   text.setBackground(Color.WHITE);
}
```

Nach diesen Mühen haben die Anwender des Editors u.a. die Möglichkeit, das Look & Feel auf **Windows** umstellen, um dann z.B. den folgenden Dateiauswahldialog zu benutzen:



11.9 Übungsaufgaben zu Abschnitt 11

1) Ermitteln Sie mit Hilfe eines kurzen Programms die Kennungen zu folgenden Ereignissen:

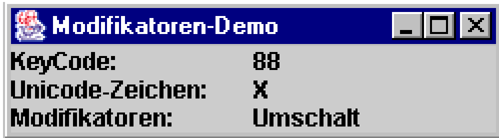
- Eine Schaltfläche wurde betätigt.
- Über einer Komponente wurde eine Maustaste gedrückt.
- Die über einer Komponente gedrückte Maustaste wurde losgelassen.
- Eine Taste wurde gedrückt bzw. losgelassen.
- Ein Fenster wurde aktiviert.

Sie werden feststellen, dass alle 2 bzw. 3 Maustasten dieselbe Ereigniskennung liefern. Mit Hilfe der **MouseEvent**-Methode **getButton()** kann man die Tasten aber doch unterscheiden.

2) Erstellen Sie ein Tastatur-Demonstrationsprogramm, das für jede gedrückte Taste(nkombination) ausgibt:

- KeyCode der zuletzt gedrückten Taste
- Unicode-Zeichen (falls definiert)
- gedrückte Modifikator-Tasten (Umschalt, Steuerung, Alt)

So ähnlich sollte Ihr Programm z.B. auf die Tastenkombination **<Umschalt>+<x>** reagieren:



Verwenden Sie bitte zur Anordnung der Komponenten ein **GridLayout** mit 3 Zeilen und 2 Spalten.

3) Zu einer anonymen Klasse lässt sich nur *eine* Instanz erzeugen. Es ist aber durchaus möglich, ein solches Objekt als **ActionListener** für mehrere Befehlsschalter zu verwenden, indem der zuerst versorgte Schalter mit **getActionListeners()** nach dem zuständigen Ereignisempfänger befragt und die erhaltene Referenz anschließend wieder verwendet wird.
Wenngleich die beschriebene Konstellation keine nennenswerten Vorteile bietet, wird doch die Routine beim Umgang mit Ereignissen und anonymen Klassen durch die Anfertigung eines Beispielprogramms (z.B. mit zwei Befehlsschaltern) gefördert.

4) Schreiben Sie einen Euro-DM-Konverter, der in etwa folgende Benutzeroberfläche bietet:

Für den Lösungsvorschlag wurden einige im Manuskript nicht behandelte Techniken eingesetzt:

- Für den mit **Konvertieren** beschrifteten Befehlsschalter wurde mit folgendem Methodenaufruf (gerichtet an die **RootPane**-Schicht des **JFrame**-Fensters) festgelegt, dass sich die **Enter**-Taste an ihn richten soll:
  ```
  getRootPane().setDefaultButton(konvert);
  ```
- Die beiden **JPanel**-Komponenten, die den linken bzw. rechten Teil des Fensters besetzen, sind aus ästhetischen Gründen jeweils mit einem Rahmen versehen worden, z.B.:
  ```
  panLinks.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
  ```
  Das in der Methode **setBorder()** benötigte **Border**-Objekt wird von der Klassenmethode **BorderFactory.createEmptyBorder()** erzeugt.
- Außerdem wurden Änderungen in der Eingabefokus-Verwaltung vorgenommen, die im Screenshot nicht sichtbar, aber für den Benutzer recht nützlich sind:
  - Beim Start erhält das Eingabefeld den Fokus über den Methodenaufruf:
    ```
    eingabeFeld.requestFocus();
    ```
  - Die seltener benötigten Komponenten werden aus der Fokussequenz herausgenommen, z.B.:
    ```
    euro2dm.setFocusable(false);
    ```
    Leider ist die Methode **setFocusable()** erst ab der SDK-Version 1.4 verfügbar.
  - Der mit **Konvertieren** beschrifteten Befehlsschalter gibt den Fokus spontan weiter (an das Eingabefeld):
    ```
    konvert.transferFocus();
    ```

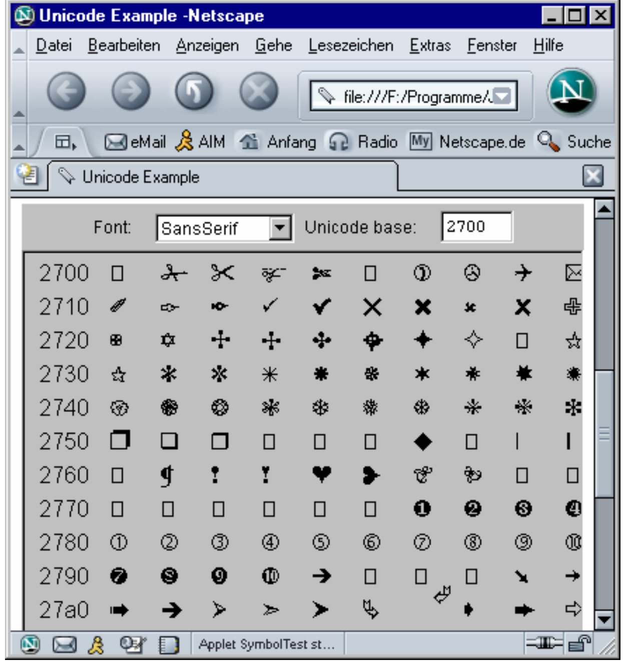
Weitere Hinweise zum Lösungsvorschlag:

- Die initial mit **Euro** beschriftete **JTextField**-Komponente soll nur zur Ausgabe dienen. Daher wurde via **setEditable(false)** festgelegt, dass sie vom Benutzer nicht geändert werden kann.

- Bei dem horizontalen Pfeil in den Beschriftungen des Optionenfeldes handelt es sich um das Unicode-Zeichen mit der Nummer 0x27A0. Über eine Escape-Sequenz lassen sich beliebige Unicode-Zeichen in einem Java-Programm verwenden, z.B.:

```
dm2euro = new JRadioButton("DM "+'\u27a0'+" Euro", true);
```

  Als Unicode-Browser kann das mit dem Java-SDK ausgelieferte Applet **SymbolTest** wertvolle Dienste leisten:



  Ausgehend vom SDK-Wurzelverzeichnis ist es folgendermaßen zu finden:

  ...\demo\applets\SymbolTest

5) Erweitern Sie das Editor-Beispielprogramm in Abschnitt 11.7 um die Möglichkeit, den bearbeiteten Text zu sichern.

12 Applets

Java war lange vor allem dazu gedacht, das WWW mit kleine Prográmmchen (Applets) aufzuwerten, die von einem Webserver schnell via Internet zum lokalen Rechner transportiert und dort im Browser-Fenster ausgeführt werden können. Auf diese Weise lassen sich HTML-Seiten attraktiv (z.B. multimedial), dynamisch und interaktiv gestalten. Die Anwender können einen prinzipiell unbegrenzten Fundus an Software nutzen, ohne diese lokal installieren zu müssen.
Wie ein interaktives, optisch attraktives Java-Applet sich im Browser-Fenster darstellt, war schon in der Einleitung zu sehen. Wer auch die akustischen Qualitäten des **TicTacToe**-Beispiels genießen möchte, muss das zum Java 2 SDK gehörige Applet auf einem multimedia-tauglichen Rechner ausführen.
Die ursprüngliche Bevorzugung von Applets gegenüber den Java-Programmen kommt z.B. darin zum Ausdruck, dass einige Java-Versionen lang die Sound-Ausgabe ausschließlich in Applets möglich war.
Mittlerweile hat sich die Lage jedoch deutlich gewandelt:

- Es gibt einige alternative, teilweise einfacher zu realisierende, Möglichkeiten zur dynamischen und interaktiven Gestaltung von Webseiten (z.B. Javaskript, Macromedia Flash, animierte Gifs, ActiveX). Nur bei besonders anspruchsvollen Web-Anwendungen (z.B. Internet-Banking, komplexe Visualisierungen) hat sich Java seine Vorrangstellung bewahrt.
- Java hat sich Java inzwischen zu einer vollwertigen, weitgehend universell einsetzbaren Programmiersprache entwickelt, die wegen ihrer modernen Konzeption hohes Ansehen geniest und eine zunehmende Verbreitung erlebt. M.E haben Java-Applikationen, die wir im bisherigen Kursverlauf ausschließlich kennen gelernt haben, mittlerweile eine größerer Bedeutung als Applets. Außerdem werden Java-Lösungen auf dem Webserver (**J**ava **S**erver **P**ages, Servlets) immer populärer.

Damit haben Applets zwar ihre ursprüngliche Bedeutung verloren, doch bieten sie nach wie interessante Möglichkeiten für viele Szenarien und stellen damit einen Zusatznutzen der Programmiersprache Java dar.

Durch die Browser-Einbettung ergeben sich zwar einige Besonderheiten bei Applets, doch bleiben wir bei der selben Programmiersprache, können also die Syntaxregeln und den größten Teil der Java-Klassen auch für Applets verwenden. Über Möglichkeiten, eine Java-Aplikation mit geringem Aufwand in ein Java-Applet umzuwandeln, informiert z.B. Krüger (2002, Abschnitt 40.3).

Damit ein Browser Java-Applets ausführen kann, benötigt er eine virtuelle Java-Maschine, die auf möglichst aktuellem Stand sein sollte. Leider haben manche Browser mit der Java-Entwicklung *nicht* Schritt gehalten, so dass man sich bei der Applet-Entwicklung für *beliebige* Internet-Nutzer mit unbekanntem Browser auf den Stand von Java 1.1 beschränken und z.B. beim GUI-Design auf die Swing-Komponenten verzichten muss.
Günstiger ist die Situation bei *Intranet*-Anwendungen, weil dort die Ausrüstung der Browser eher unter Kontrolle ist. Weiter Hinweise und Lösungsvorschläge zur Browser-Problematik finden Sie in Abschnitt 12.5.
Während bei Java-*Programmen* die Swing-Komponenten eindeutig zu bevorzugen sind, hängt also bei Applets die Entscheidung von den konkreten Umständen ab. Für den aktuellen Abschnitt des Kurses habe ich mich dafür entschieden, der Kompatibilität halber mit *AWT*-Komponenten zu arbeiten, wobei die eigentlichen Lerninhalte zum Thema Applets von der GUI-Frage weitgehend unberührt sind.
Allerdings garantiert der Verzicht auf Swing-Komponenten noch keine Kompatibilität mit älteren virtuellen Maschinen. Im Prinzip muss dazu auch ein Compiler mit passender Version verwendet werden. Z.B. kann die in Netscape 4.7 eingebaute VM mit Java-Version 1.1.5 die in diesem Ab-

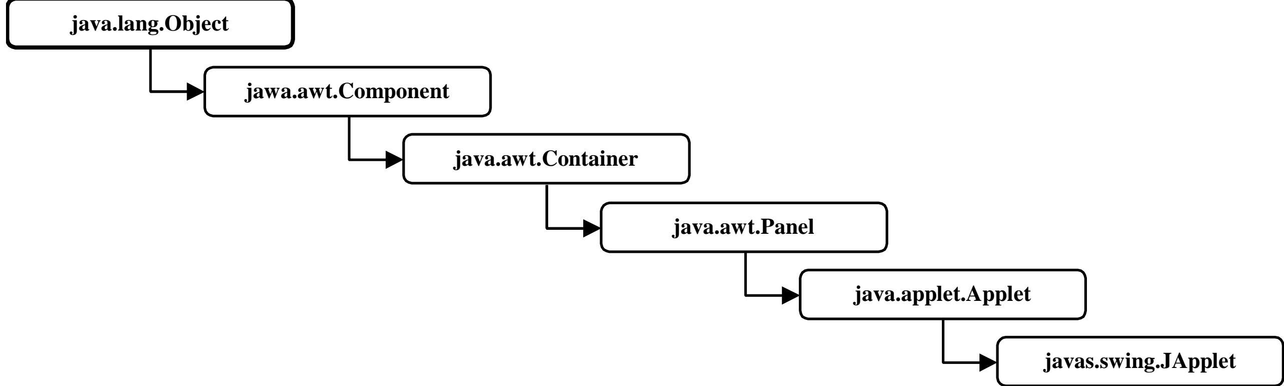
schnitt vorgestellten Applets in der Regel ausführen, wenn sie mit dem SDK 1.3 erstellt werden, während die Produkte der SDK-Version 1.4 eher nicht funktionieren.

Nach den langen Vorbemerkungen und Einordnungsversuchen wollen wir nun den Aufbau und die Verhaltensweisen eines Applets näher betrachten.

12.1 Stammbaum der Applet-Basisklasse

Wie eine Java-*Anwendung* besteht auch ein Java-*Applet* aus mindestens einer Klassendefinition, wobei die beim Applet-Start aktive Hauptklasse stets aus **java.applet.Applet** (bei Beschränkung auf AWT-Komponenten) oder aus **javax.swing.JApplet** (bei Verwendung von Swing-Komponenten) abgeleitet werden muss.

Um zu wissen, welche Methoden und Variablen die Klasse **(J)Applet** von ihren Vorfahren erbt, lohnt sich ein Blick auf den Stammbaum:



Insbesondere ist ein Applet also eine *Komponente* und folglich eine potentielle Ereignisquelle. Während eine Java-Anwendung auch „ereignislos“ und textorientiert entworfen werden kann, ist ein Applet stets grafik- und ereignisorientiert.
Die in der Klasse **(J)Applet** im Vergleich zu einer Java-*Anwendung* definierten Zusatzkompetenzen beziehen sich vor allem auf die Kooperation mit dem Browser, in dessen Kontext ein Applet abläuft.
Wichtige Kooperations-Methoden werden gleich mit Hilfe des folgenden Beispiels erläutert, das gemäß obiger Verabredung auf der Klasse **java.applet.Applet** basiert:

```
import java.awt.*;
import java.applet.*;

public class Life extends Applet {
  private String iniStatus;
  private int startCount;
  private int stopCount;
  private int paintCount;

  public void paint(Graphics g) {
    paintCount++;
    g.drawString("Applet-Status:",0, 10);
    g.drawString("-----------------",0, 20);
    g.drawString(iniStatus,0, 40);
    g.drawString("start()-Aufrufe = " + startCount, 0, 60);
    g.drawString("stop()-Aufrufe = " + stopCount, 0, 80);
    g.drawString("paint()-Aufrufe = " + paintCount, 0, 100);
  }
```

```
   public void init() {
      setBackground(Color.WHITE);
      iniStatus = "Neu initialisiert";
   }

   public void start() {
      startCount++;
   }

   public void stop() {
      stopCount++;
      iniStatus = "Altlast";
   }

   public void destroy() {
      AudioClip clip = getAudioClip(getCodeBase(),"Kuckuck.au");
      clip.play();
      long time = System.currentTimeMillis();
      while (System.currentTimeMillis() - time < 600);
   }
}
```

Weil hier kritische Ereignisse im Leben eines Applets sichtbar gemacht werden sollen, hat dieses Beispiel den Namen `Life` erhalten.
Während die Startklasse einer *Anwendung* (zumindest bis zur Java-Version 1.4) ohne den Modifikator **public** auskommt, kann die Startklasse eines *Applets* nicht darauf verzichten.

12.2 Applet-Start via Browser oder Appletviewer

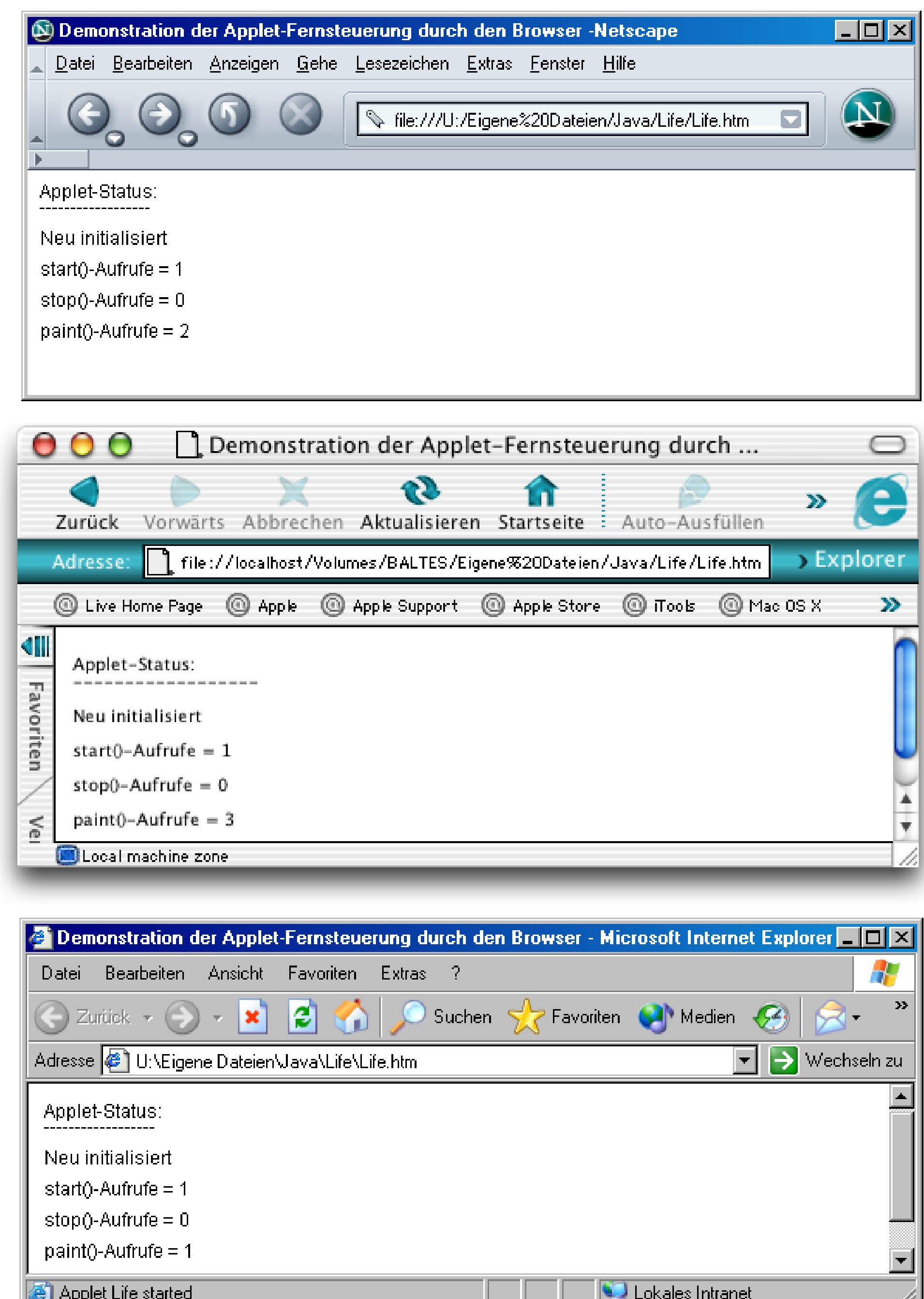
Wesentliche Eigenart eines Applets ist seine Einbettung in eine HTML-Seite, wobei im Normalfall ein so genanntes **applet**-Tag verwendet wird, das im Life-Beispiel z.B. folgendermaßen aussehen kann:

```
<html>
<head>
<title>Demonstration der Applet-Fernsteuerung durch den Browser</title>
</head>
<body>
<applet code="Life.class" width=300 height=120>
</applet>
</body>
</html>
```

Bei den Attributen des Applet-Tags beschränkt sich das Beispiel auf die wichtigsten Angaben:

- **code**
  Name der Bytecodedatei zur Hauptklasse
- **width**, **height**
  Die Breite und Höhe der Appletfläche in Pixeln

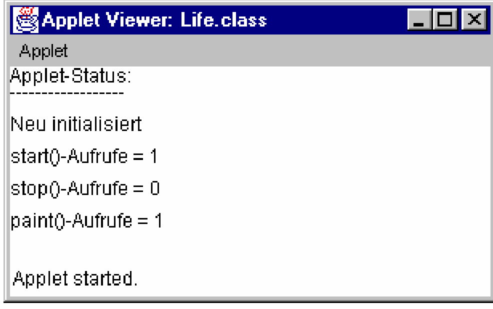
Um ein Applet in einem Browser auszuführen, öffnet man das zugehörige HTML-Dokument, z.B. per URL-Eingabe, über das **Datei**-Menü oder schlicht per Drag-&-Drop, z.B.:



Die Rolle des Browsers beim Starten und Ausführen eines Java-Applets kann auch vom Jva-SDK-Hilfsprogramm **appletviewer** übernommen werden. Es ist in der Entwicklungsphase sogar zu bevorzugen, weil die übliche Cache-Strategie der Browser das Testen der aktuellen Applet-Version oft erschwert.
Bei Verwendung der **JCreator** –Entwicklungsumgebung startet der Appletviewer automatisch nach dem Befehl **Execute Project** (Tastenbefehl **F5**), wenn es sich beim aktuellen Projekt um ein Applet handelt.
Man kann den Appletviewer aber auch in einem Konsolenfenster starten und dabei den Namen der HTML-Datei als Kommandozeilenparameter übergeben, z.B.:

```
U:\Eigene Dateien\Java\Applets\Life>appletviewer Life.htm
```

Das im vorigen Abschnitt wiedergegebene Applet `Life` sieht im Fenster des Appletviewers unmittelbar nach dem Start folgendermaßen aus:



Über weitere Attribute des Applet-Tags und sonstige Optionen beim Einbinden von Applets auf HTML-Seiten informiert z.B. Münz (2001).

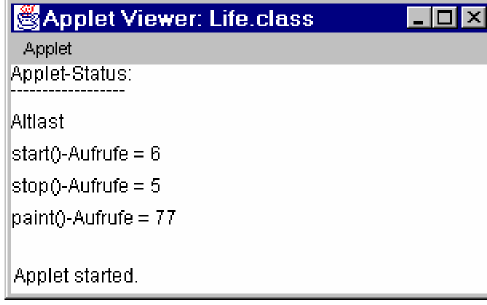
12.3 Methoden für kritische Lebensereignisse

Wer das Applet in Abschnitt 12.1 vor dem Hintergrund unserer bisherigen Erfahrungen mit Java-Anwendungen näher betrachtet, wird bald die **main()**-Methode vermissen, über die eine Java-*Anwendung* vom Interpreter in Gang setzt wird. Bei einem Applet läuft die Initialphase deutlich anders ab:

- Der Browser (oder Appletviewer) erzeugt automatisch ein Objekt der im Applet-Tag angegebenen Klasse, die aus diesem Grund unbedingt als **public** deklariert werden muss.
- Üblicherweise benutzt man *keinen* Konstruktor der Klasse, um das beim Applet-Start erzeugte Objekt zu initialisieren. Dazu verwendet man die **Applet**-Methode **init()**, die vom Browser beim Laden der Klasse automatisch ausgeführt wird. In unserem Beispiel wird hier nur der Wert einer **String**-Variablen verändert, um den automatischen **init()**-Aufruf sichtbar zu machen.
- Nach **init()** führt der Browser (oder Appletviewer) die **start()**-Methode des Applets aus. Während die **init()**-Methode eines Applet-Objektes nur *einmal* aufgerufen wird, führt der Browser die **start()**-Methode auch beim Reaktivieren eines zwischenzeitlich gestoppten Applet-Objektes aus.
- Schließlich wird die Methode **paint()** des Applet-Objektes aufgerufen, mit der die Grafikelemente gezeichnet werden (vgl. Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**). Dies geschieht im weiteren Leben des Objektes immer dann, wenn das Fenster aus irgend einem Grund neu gezeichnet werden muss, z.B. nach der Rückkehr aus dem ikonisierten Zustand.

In der **init()**-Methode unseres Beispiels wird mit der Applet-Methode **setBackground()** die Hintergrundfarbe der Applet-Fläche festgelegt. Außerdem wird in einer Stringvariablen ein „Tätigkeitsnachweis“ hinterlassen.
Die Methoden **start()** und **paint()** erhöhen jeweils eine Zählvariable, welche die Anzahl ihrer bisherigen Aufrufe festhält.
In **paint()** schreibt das via Referenzparameter übergebene **Graphics**-Objekt[1] mit seiner Methode **drawString()** alle Statusinformationen auf die Applet-Fläche, so dass wir über den bisherigen Lebensweg des Applets informiert sind, z.B.:

[1] Streng genommen kann man einem ***Graphics****-Objekt* nicht reden, weil **Graphics** eine abstrakte Klasse ist. Allerdings kennen wie die konkrete, aus **Graphics** abgeleitete, Klasse des Parameter-Objektes *nicht*, so dass wir uns der Einfachheit halber die ungenaue Redeweise leisten.



In Abschnitt 13.1 werden wir uns näher mit der Graphikausgabe beschäftigen.
Damit der Browser (oder Appletviewer) die genannten Methoden aufrufen kann müssen sie alle als **public** definiert werden. Sollten Sie anderes planen, wird schon der Compiler protestieren: Die Methoden sind in den Basisklassen als **public** definiert, und beim Überschreiben dürfen generell keine Zugriffsrechte eingeschränkt werden.
Die geerbten Varianten der Methoden sind übrigens unterschiedlich aktiv:

- Die **(J)Applet**-Methoden **init()** und **start()** tun *nichts*.
- Die **Container**-Methode **paint()** reicht die Aufforderung zum Zeichnen nötigenfalls an Kindfenster weiter, was in unserem Beispiel nicht erforderlich ist.

Nicht nur in der Initialphase gibt der Browser dem Applet vor, was zu tun ist:

- Der Browser kann die **stop()**-Methode eines Applets ausführen, um es vorübergehend zu deaktivieren. Dabei bleiben alle Ressourcen des Applets erhalten, so dass es später fortgesetzt werden kann, wobei die **start()**-Methode erneut ausgeführt wird. Wie gleich in einer Tabelle zu sehen ist, machen die Browser der Firmen Microsoft und Netscape allerdings wenig Gebrauch von der **stop()**-Methode.
- Ist ein Applet-Objekt am Ende seiner Existenz angelangt, führt der Browser seine **destroy()**-Methode aus, wobei alle Ressourcen freigegeben werden. Dies geschieht z.B. dann, wenn der Browser selbst beendet wird.

Wie in der folgenden Tabelle für vier spezielle Situationen gezeigt wird, verfahren die gängigen Browser nicht ganz einheitlich mit Applets:

	Wechsel zu anderer HTML-Seite und Rückkehr	Änderung der Browser-Fenstergröße	Ikonis. des Browsers und Rückkehr zur Normalgröße	HTML-Seite neu laden
Appletviewer im Java SDK 1.4.1	*entfällt*	paint()	stop() start(), paint()	destroy() init(), start(), paint()
Internet Expl. 6 Win mit Java-Plugin 1.4.1	destroy() init(), start(), paint()	*keine* Botschaft: Applet-Ausgabe wird undefiniert	paint()	destroy() init(), start(), paint()
Internet Expl. 5.2 Mac mit JVM 1.3	destroy() init(), start(), paint()	paint()	paint()	destroy() init(), start(), paint()
Netscape 4.7 Win mit integrierter JVM	stop() start(), paint()	paint()	paint()	stop() start(), paint()
Netscape 7.1 Win mit Java-Plugin 1.4.1	destroy() init(), start(), paint()	paint()	paint()	destroy() init(), start(), paint()

12.4 Sonstige Methoden für die Applet-Browser-Kooperation

Anschließend werden noch einige Methoden für die Interaktion zwischen Applet und Browser vorgestellt. Wer sich für die Interaktion zwischen mehreren, simultan aktiven, Applets interessiert, kann sich z.B. bei Krüger (2002, Abschnitt 40.2) informieren.

12.4.1 Parameter übernehmen

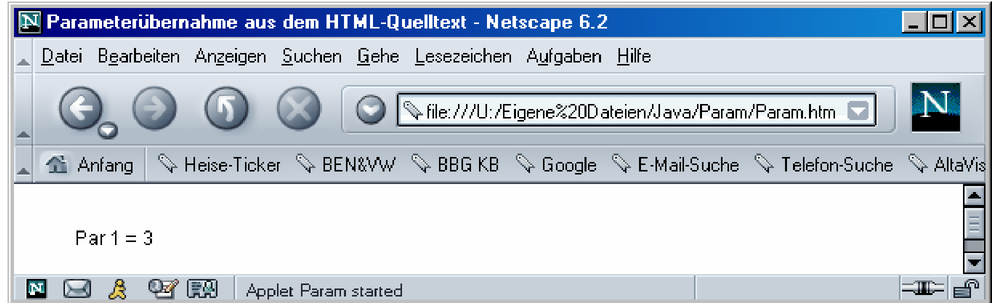
Ein Applet-Programmierer kann seinem Kunden (dem Web-Designer) eine Möglichkeit schaffen, das Verhalten des Applets über Parameter zu steuern. Zur Vereinbarung von konkreten Parameterausprägungen im HTML-Code dienen **param**-Tags, z.B.:

```
<html>
<head>
<title>Parameterübernahme aus dem HTML-Quelltext</title>
</head>
<body>
<applet code="Param.class" width=200 height=50>
<param name = "Par1" value = "3">
</applet>
</body>
</html>
```

Ein Applet wird seine Steuerparameter in der Regel schon in der **init()**-Methode ermitteln, z.B.:

```
import java.awt.*;
import java.applet.*;

public class Param extends Applet {
   private String par1;
   private int pari1;

   public void init() {
      setBackground(Color.WHITE);
      par1 = getParameter("Par1");
      try {pari1 = Integer.parseInt(par1);}
      catch (Exception e) {//Ausnahmen behandeln
      };
   }

   public void paint(Graphics g) {
      g.drawString("Par 1 = " + pari1,30, 30);
   }
}
```

Die zuständige Methode **getParameter()** liefert nur Strings, so dass eventuell mit den entsprechenden Methoden der Wrapper-Klassen noch eine Konvertierung vorzunehmen ist.
Im Beispiel erhalten wir folgendes Ergebnis:

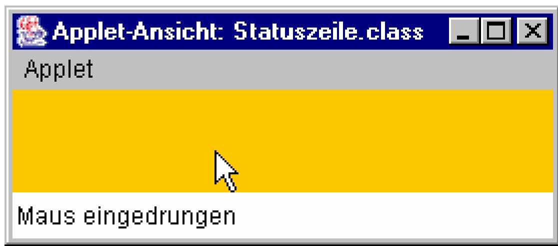


12.4.2 Browser-Statuszeile ändern

Über seine Methode **showStatus()** hat ein Applet Zugriff auf die Browser-Statuszeile. In folgendem Beispiel werden dort Besuche der Maus dokumentiert:

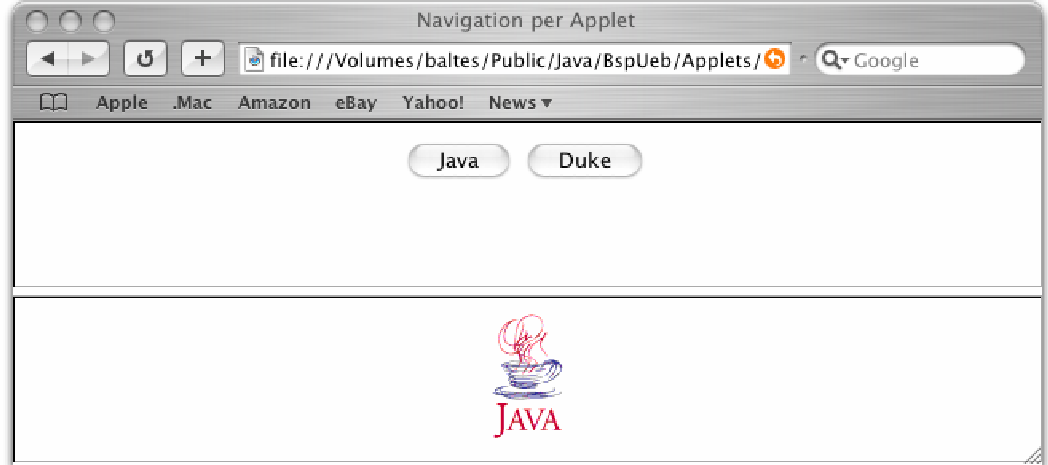
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Statuszeile extends Applet {
  public void init() {
    setBackground(Color.ORANGE);
    addMouseListener(new MausDetector());
  }
class MausDetector extends MouseAdapter {
  public void mouseEntered(MouseEvent e) {
    showStatus("Maus eingedrungen");
  }
  public void mouseExited(MouseEvent e) {
    showStatus("Maus entwischt");
  }
}
}
```

Wie das Beispiel zeigt, ist die in Abschnitt 11.6 vorgestellte Ereignisbehandlung auch bei AWT-basierten Applets verwendbar:



12.4.3 Andere Webseiten öffnen

Über die Möglichkeit, aus einem Applet heraus HTML-Seiten zu öffnen, kann man z.B. ein Frameset mit Navigationszone oder auch eine Imagemap erstellen. Wir wollen uns mit dem ersten Thema beschäftigen und das folgende Frameset realisieren. Als Browser kommt diesmal Safari 1.0 unter MacOS 10.2 zum Einsatz:



Im oberen Frame („`navigation`“ genannt) wird die Datei **Navigation.htm** geöffnet, die das Applet **Appligator.class** aufruft. Im unteren Frame („`content`“ genannt) wird initial die Datei **Content1.htm** (mit der animierten Java-Tasse) geöffnet.

Das gesamte Frameset ist in der Datei **NaviDemo.htm** enthalten:

```
<html>
<head>
<title>Navigation per Applet</title>
</head>
<frameset rows="*,*">
  <frame name="navigation" src="Navigation.htm">
  <frame name="content" src="Content1.htm">
  <noframes>
  <body>
  <p>Diese Seite verwendet Frames. Frames werden von Ihrem Browser aber nicht
  unterstützt.</p>
  </body>
  </noframes>
</frameset>
</html>
```

In der Datei **Navigation.htm** wird mit dem **center**-Tag für einen zentrierten Auftritt des Applets gesorgt:

```
<html>
<body>
<center>
<applet code="Appligator.class" width=200 height=30>
</applet>
</center>
</body>
</html>
```

In der `Appligator`-Klassendefinition werden gemäß obiger Verabredung die Befehlsschalter über AWT-Komponenten realisiert. An Stelle der Swing-Klasse **JButton** kommt die AWT-Klasse **Button** zum Einsatz, was aber keine weiteren syntaktischen Konsequenzen hat.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.*;

public class Appligator  extends Applet  implements ActionListener {
   private Button duke = new Button("Duke");
   private Button java = new Button("Java");

   public void init() {
      setBackground(Color.WHITE);
      add(java);
      add(duke);
      java.addActionListener(this);
      duke.addActionListener(this);
   }
```

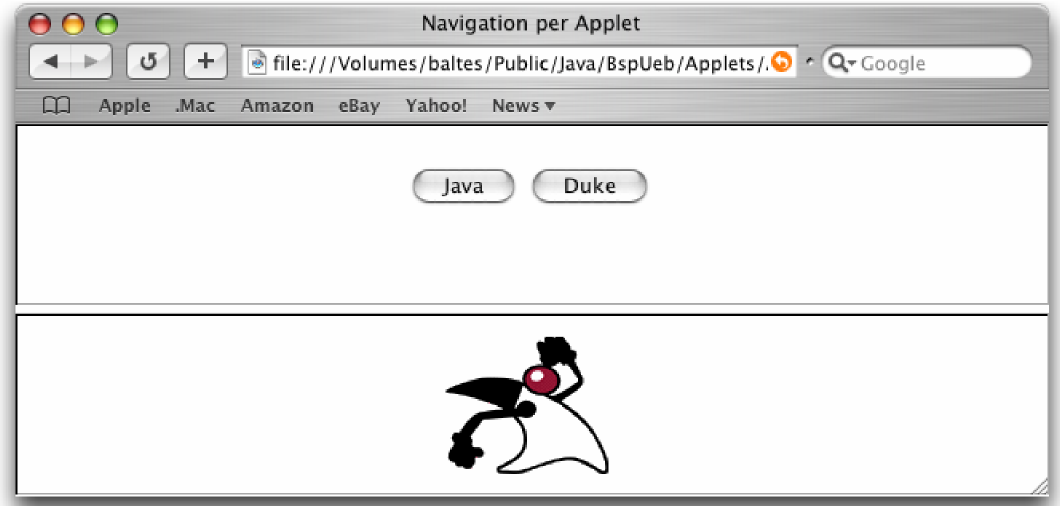
```
  public void actionPerformed(ActionEvent e)
  {
    URL neus;
    try {
      if (e.getSource() == java)
        neus = new URL(getCodeBase(), "Content1.htm");
      else
        neus = new URL(getCodeBase(), "Content2.htm");
      getAppletContext().showDocument(neus, "content");
    }
    catch (Exception ex) {}
   }
}
```

In der **init**()-Methode wird zunächst mit **setBackground**() die Hintergrundfarbe des Applets eingestellt.
Schon in Abschnitt 11.2.2 wurde das Applet als Top-Level-Container eingeordnet. Daher kann es mit seiner **add**()-Methode die beiden **Button**-Komponenten aufnehmen. Im Unterschied zur **JFrame**-Komponente wird hier keine spezielle *Schicht* (z.B. Content Pane) angesprochen.

Außerdem ist die Klasse `Appligator` als Ereignisempfänger für die Befehlsschalter gerüstet, weil sie das Interface **ActionListener** implementiert. In der Methode **actionPerformed**() wird je nach betätigtem Schalter ein URL-Objekt aus der passenden HTML-Datei erzeugt (`Content1.htm` oder `Content2.htm`), welche über die Methode **getCodeBase**() im Pfad des Applets lokalisiert wird.

Mit **getAppletContext**() wird ein Objekt erzeugt, welches den Browser (oder Appletviewer) vertritt. An dieses Objekt richtet sich dann die Botschaft **showDocument**(), die als Parameter den URL der anzuzeigenden HTML-Seite sowie den Zielframe enthält.
In der API-Dokumentation zu **AppletContext.showDocument**() sind einige alternative Ziele angegeben. Z.B. erhält man mit der Zielangabe **_blank** eine neues Toplevel-Browserfenster.

Abschließend soll auch noch der Zustand nach einem Mausklick auf den Schalter **Duke** gezeigt werden:



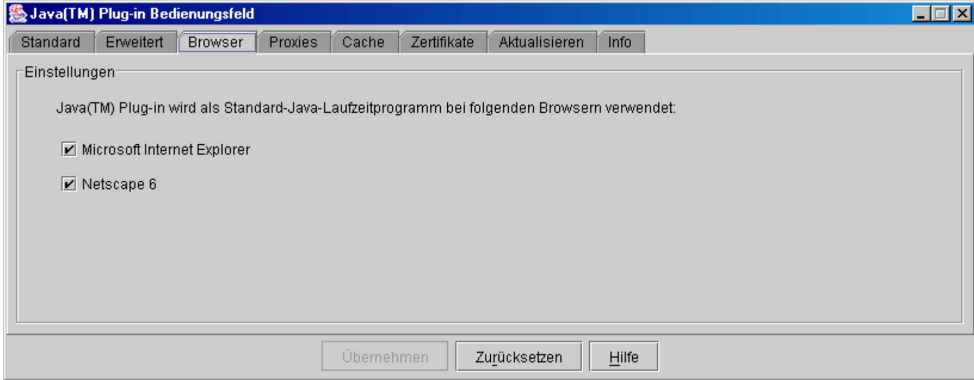
12.5 Das Java-Browser-Plugin

Wer das Swing-GUI bevorzugt, leitet die Hauptklasse seines Applets von **javax.swing.JApplet** ab. Allerdings wird Swing von vielen Browsern noch nicht unterstützt. So arbeitet der Netscape Navigator 4.78 noch mit der Java-Version 1.1.5, und der Internet Explorer unterstützt (mit Microsofts JVM) sogar manche JRE-1.1-Eigenschaften nicht.

Mit dem von Sun kostenlos angebotenen Java-Plugin können allerdings praktisch alle Browser Swing-tauglich gemacht werden. Es ersetzt die im Browser eingebaute virtuelle Maschine durch den jeweils aktuellen Stand der Java-Entwicklung, wobei natürlich neben Swing/JFC auch andere Java-Bestandteile profitieren.

Auf vielen Rechner ist das Java-Plugin schon unbemerkt installiert worden, z.B. zusammen mit der Java Runtime Environment (JRE) ab Version 1.2 oder mit dem entsprechenden Java SDK. Die genannten Sun-Produkte erweitern bei ihrer Installation die vorgefundenen Browser automatisch um das Java-Plugin.

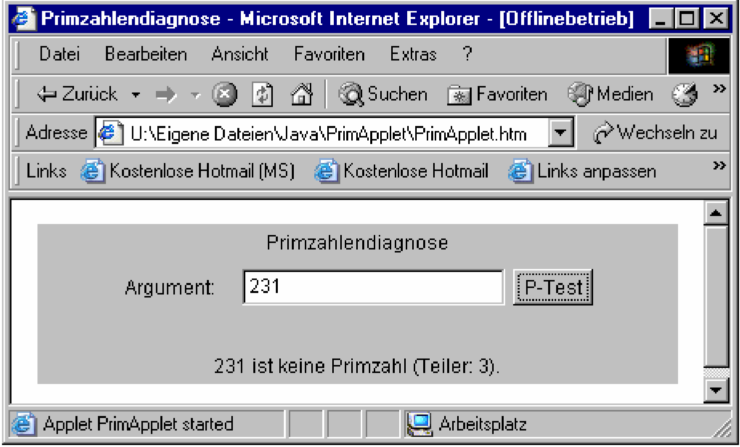
Unter MS-Windows wird mit dem Java-Plugin auch ein Systemsteuerungs-Konfigurationsprogramm installiert. Hier kann man z.B. einstellen, ob das Plugin als Java-Standard-Laufzeitumgebung für die unterstützten Browsern verwendet werden soll:



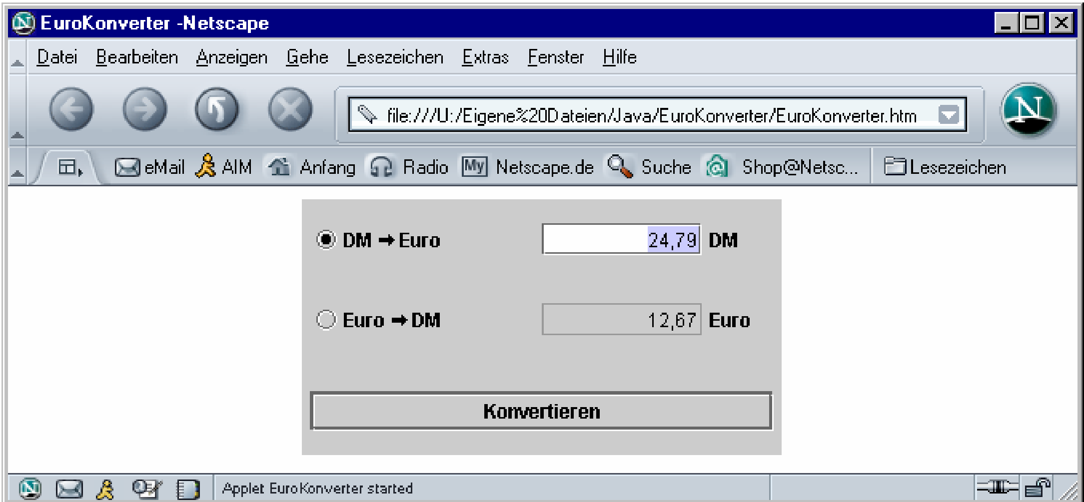
Ein aktives Java-Plugin macht sich im Statusbereich der Windows-Taskleiste (neben der Uhr) durch ein Java-Symbol bemerkbar.

12.6 Übungsaufgaben zu Abschnitt 12

1) Erstellen Sie ein Applet zur Primzahlendiagnose, z.B. mit folgender Benutzeroberfläche:



2) Erstellen Sie eine Applet-Variante zum Euro-DM-Konverter, den Sie als Übungsaufgabe zu Abschnitt 11 entwickelt haben.
In diesem Vorschlag wird in Abweichung von unserer bisherigen Praxis ein *Swing*-GUI verwendet:



13 Multimedia

Dieser Abschnitt enthält elementare Hinweise zu zweidimensionaler Grafik und Sound, die für Applets und Java-Anwendungen in gleicher Weise gelten.

13.1 Grafik

Die bereits behandelten GUI-Komponenten bieten etliche Möglichkeiten zur individuelle Gestaltung der Benutzeroberfläche (z.B. durch Texte in diversen Schriftarten, Bilder oder speziell gestaltete Rahmen). Allerdings ist oft eine freie Grafikausgabe unumgänglich, die von Java mit zahlreichen Methoden zum Zeichnen elementarer Gebilde (z.B. Linien, Ellipsen, Rechtecke, Texte) unterstützt wird.
Weil die meisten von **jawa.awt.Component** abstammenden Klassen bemalt werden können, bieten sich dem kreativen Programmierer verschiedene Möglichkeiten, z.B.:

- **Bemalen einer leeren Zeichenfläche**
  In einem Swing-GUI eignet sich vor allem die Komponente **JPanel** als Zeichenfläche für das freie Gestalten. Zwar kann man auch Top-Level-Container wie **JFrame** bemalen, doch insbesondere bei einer Kombination von Swing-Bedienelementen mit freien Grafiken sollten letztere auf **JPanel**-Komponenten untergebracht werden.
- **Eigene Bedienelemente entwerfen**
  Von einer Standardkomponente wie z.B. **JButton** ausgehend kann man eine eigene Bedienelement-Klasse mit dem gewünschten Design ableiten. Wenngleich die Zeichnungsfläche stets rechteckig ist, lassen sich über intelligente Ereignismethoden auch anders geformte Bedienelemente realisieren, z.B. eine achteckige STOPP-Schalfläche.

In Abschnitt 13.1 wird in erster Linie das Swing-API berücksichtigt, das auch bei der Grafikausgabe einige Verbesserungen im Vergleich zum älteren AWT enthält (z.B. die automatische Doppelpufferung gegen das Flackern bei der Bildschirmausgabe). Die (sehr einfach gehaltenen) Applet-Beispiele beschränken sich aber wie in Abschnitt 12 aus Kompatibilitätsgründen auf das AWT-GUI.

Für besonders anspruchsvolle Grafikausgaben empfiehlt sich das in diesem Kurs aus Zeitgründen nicht behandelte **Java 2D API** (siehe z.B. SUN Inc. 2003).

13.1.1 Die Klasse Graphics

Im Abschnitt 12 haben wir in der **paint()**-Methode einiger Applets ohne große Erläuterungen die Methode **drawString()** verwendet, um Text als *Grafikelement* auf die Appletfläche zu bringen, z.B.:

```
g.drawString("Par 1 = " + pari1, 30, 30);
```

Die Zeichenbefehle haben wir an ein per Referenzparameter zur Verfügung gestelltes Objekt geschickt, das die abstrakte Klasse **java.lang.Graphics** erfüllt.
Zu welcher *konkreten* Klasse das (automatisch erzeugte) Objekt gehört, kann man über die **Object**-Methode **getClass()** ermitteln:

```
g.drawString(g.getClass().toString(), 50,50);
```

Bei Java 1.4.1 stellt man unter MS-Windows die folgende **Graphics**-Subklasse fest:

sun.java2d.SunGraphics2D

Auf einem Macintosh mit Java 1.3 ist anzutreffen:

com.apple.mrj.internal.awt.graphics.PenGraphics

Während die abstrakte Klasse **Graphics** eine plattformunabhängige Grafik-Schnittstelle definiert, stellt z.B. die konkrete Klasse **com.apple.mrj.internal.awt.graphics.PenGraphics** deren plattformspezifische Implementation auf dem Macintosh dar.
Wir haben uns schon in Abschnitt 12.3 entschieden, der Einfachheit halber die nicht ganz korrekte Bezeichnung **Graphics**-Objekt zu verwenden, um von der Plattform unabhängig zu bleiben und die umständlichen Namen der konkreten Klassen zu vermeiden.

Ein **Graphics**-Objekt bietet nicht nur ca. 50 Ausgabemethoden für diverse graphische Elemente wie Linien, Rechtecke, Ovale, Polygone etc. (siehe API-Referenz), sondern stellt auch einen so genannten **Grafikkontext** mit allen für eine Grafikausgabe relevanten Informationen zur Verfügung. U.a. kennt das **Graphics**-Objekt von der Komponente, deren Oberfläche zu zeichnen ist:

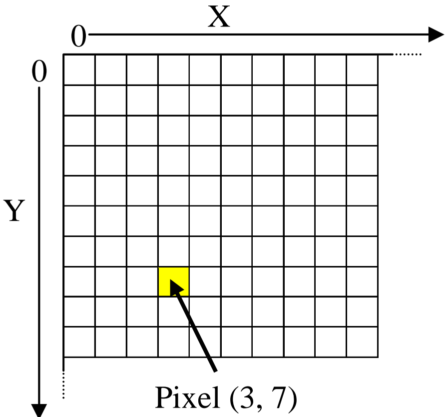
- Bildschirmposition und Größe des zu zeichnenden Rechtecks
- Vorder- und Hintergrundfarbe
- Schriftart

In Applets und GUI-Anwendungen gehört zu jeder sichtbaren Komponente, also zu jedem sichtbaren Objekt aus einer von **jawa.awt.Component** abstammenden Klasse ein Grafikkontext.[1]

Um auf die Oberfläche eine Komponente zeichnen zu können, beschafft man sich mit der **Component**-Methode **getGraphics()** eine Referenz zum Grafikkontext der Komponente. Bei Verwendung der **Component**-Methode **paint()**, die bei Applets und GUI-Anwendungen z.B. dann aufgerufen wird, wenn die Komponenten-Oberfläche ganz oder teilweise neu zu zeichnen ist, wird automatisch ein **Graphics**-Objekt mitgeliefert.

13.1.2 Das Koordinatensystem der Zeichenfläche

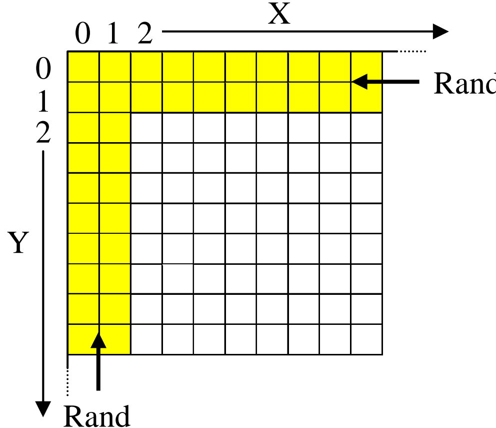
Die Grafikausgabe basiert auf einem Koordinatensystem mit dem Ursprung (0,0) in der linken oberen Ecke der betroffenen Komponente, wobei die X-Werte von links nach rechts, und die die Y-Werte von oben nach unten wachsen:



Weil die Positionsangaben in Pixeln gemacht werden, ist die tatsächliche Größe eines Grafikelementes von Größe und Auflösung des Bildschirms abhängen.

Beim Bemalen einer Swing-Komponente ist der ggf. vorhandene *Rand* zu berücksichtigen. So wird z.B. durch einen 2 Pixel breiten Rand der Punkt (2,2) zur linken oberen Ecke der verfügbaren Zeichenfläche, und bei Breite und Höhe gehen jeweils 4 Pixel verloren:

[1] Weil bei den *AWT*-Komponenten die Wirtsplattform wesentlich beteiligt ist, gelten hier Einschränkungen: Ein Bemalen der Oberfläche ist möglich bei den Top-Level-Containern **Frame** und **Applet** sowie bei der Komponente **Canvas**.



Über die aktuelle Größe der Zeichenfläche sowie der Ränder kann man sich z.B. mit den folgenden **JComponent**-Methoden informieren (siehe Beispiel in Abschnitt 13.1.3.3):

- **getHeight()**
- **getWidth()**
- **getInsets()**

13.1.3 Organisation der Grafikausgabe

Jede Grafikausgabe findet auf der Oberfläche einer Komponente statt, wobei sich dort auch ohne individuelle Grafikbefehle des Programmierers mehr oder weniger aufwändige „Gemälde" befinden: auf einer **JPanel**-Komponente immerhin eine Hintergrundfarbe, auf einem Befehlsschalter außerdem z.B. eine Beschriftung.
Ein GUI-Fenster muss aus verschiedenen Anlässen neu gezeichnet bzw. aktualisiert werden, z.B.:

- Der erste Auftritt eines Fensters oder die Rückkehr aus dem minimierten Zustand
- Ein zuvor überdeckter Teil des Fensters wird sichtbar.
- Änderungen im Programm erfordern eine Aktualisierung der Anzeige.

Im Zusammenhang mit der GUI-Programmierung haben wir uns nicht damit beschäftigt, wie das Laufzeitsystem die Aktualisierung sicher stellt. Sobald sich ein Programm nicht auf Standardkomponenten beschränkt, sondern „freie" Zeichnungen vornimmt, wird jedoch ein Blick hinter die Kulissen interessant.
Die Aktualisierung beginnt mit der „umfassendsten", änderungsbedürftigen Komponente und wird jeweils mit den enthaltenen Komponenten fortgesetzt.
Weil die GUI-Ausgabemethoden im selben Thread (Ausführungsfaden, siehe unten) ablaufen wie die Ereignisbehandlungsmethoden gilt:

- Während eine Ereignisbehandlungsmethode abläuft, ist keine GUI-Ausgabe möglich.
- Während einer GUI-Ausgabe kann keine Ereignisbehandlungsmethode ausgeführt werden.

13.1.3.1 System-initiierte Aktualisierung

Einer Komponente wird vom Laufzeitsystem automatisch eine **paint()**-Botschaft zugestellt, sobald ihre Oberfläche neu zu zeichnen ist, z.B. nach der Rückkehr eines Fensters aus der Windows-Taskleiste oder aus dem Macintosh-Dock. Alle von **java.awt.Component** abgeleiteten Klassen verfügen über eine **paint()**-Methode mit folgender Signatur:

```
public void paint(Graphics g)
```

Per **Graphics**-Parameter wird der zum Zeichnen erforderliche Grafikkontext übergeben.

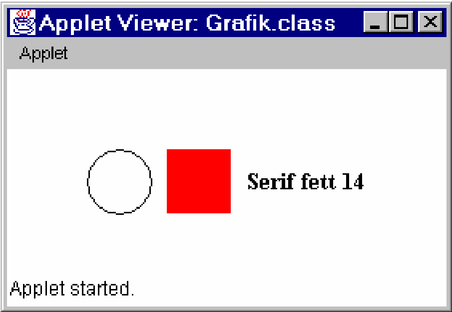
Bei **paint()** handelt es sich um eine typische **Callback**-Methode, die vom Programm bereit gehalten und vom Laufzeitsystem aufgerufen wird. Während ein Programm bzw. Applet in der Regel die **paint()**–Methoden der Komponenten nicht direkt aufruft, kann es dem Laufzeitsystem aber nahe legen, dies bei nächster Gelegenheit zu tun (siehe unten).

Um die Bemalung einer Komponente zu beeinflussen, definiert man eine eigene Klasse mit geeigneter Basisklasse und überschreibt die Methode **paint()**. Mit dieser Vorgehensweise haben Sie im Zusammenhang mit den Applets schon Erfahrungen gesammelt, an die wir nun in einem Beispiel mit elementaren Grafikausgaben anknüpfen:

```
import java.awt.*;
import java.applet.*;

public class PaintA extends Applet {
  public void init() {
    setBackground(Color.WHITE);
    setFont(new Font("Serif", Font.BOLD, 14));
  }
  public void paint(Graphics g) {
    g.drawOval(50, 50, 40, 40);
    g.drawString("Serif fett 14", 150, 75);
    g.setColor(Color.RED);
    g.fillRect(100, 50, 40, 40);
  }
}
```

Hier wird die Klasse der individuell zu bemalenden Komponente aus **Applet** abgeleitet. In der **paint()**-Überschreibung kommen die **Graphics**-Methoden **drawOval()**, **drawString()**, **setColor()** und **fillRect()** zum Einsatz. Mit **drawOval()** wird z.B. an der Bildschirmposition (50, 50) ein Oval mit einer Höhe und Breite von 40 Einheiten gezeichnet, also ein Kreis mit diesem Durchmesser. Weil die Hintergrundfarbe und die Schriftart des Applets unverändert bleiben, können sie schon in der **init()**-Methode mit **setBackground()** bzw. **setFont()** festgelegt werden.

Sobald das Applet seine Fläche neu zeichnen muss, ruft das Laufzeitsystem seine **paint()**-Methode auf, wo das übergebene **Graphics**-Objekt mit den erforderlichen Ausgaben beauftragt wird:



Im Beispielprogramm sind zwei wichtige Spezialthemen der Grafikausgabe tangiert, die leider nicht im angemessenen Umfang behandelt werden können:

- **Farben**
  Mit der **Graphics**-Methode **setColor()** kann die aktuelle Zeichenfarbe gesetzt werden, z.B.:
  ```
  g.setColor(Color.RED);
  ```
  Bis zur API-Version 1.3 waren abweichend von der üblichen Bezeichnungsweise *klein* geschrieben Farbkonstanten zu verwenden (z.B. **Color.red**). Seit der Version 1.4 sind beide Schreibweisen erlaubt.

- **Schriftarten**
  Mit der Methode **setFont()** kann man Schriftart und –auszeichnung für spätere Grafik-Textausgaben festlegen, wobei ein **Font**-Objekt übergeben werden muss, z.B.:
  ```
  g.setFont(new Font("Serif", Font.BOLD, 14));
  ```
  Eine Methode **setFont()** steht sowohl in der Klasse **java.awt.Component** als auch in der Klasse **java.awt.Graphics** zur Verfügung.
  Mit **Font**-Objekten haben wir übrigens schon in Abschnitt 11.7.2 gearbeitet.

Wird das Applet-Beispiel in eine Anwendung mit Swing-GUI umgewandelt, kann die Callback-Methode **paint()** unverändert bleiben:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class PaintS extends JFrame {
  public PaintS() {
    super("Demo zu Grafik im Swing-Anwendungen");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBackground(Color.WHITE);
    setFont(new Font("Serif", Font.BOLD, 14));
    setSize(275, 125);
    setVisible(true);
  }

  public void paint(Graphics g) {
    g.drawOval(50, 50, 40, 40);
    g.drawString("Serif fett 14", 150, 75);
    g.setColor(Color.RED);
    g.fillRect(100, 50, 40, 40);
  }

  public static void main(String[] args) {
    PaintS graf = new PaintS();
  }
}
```

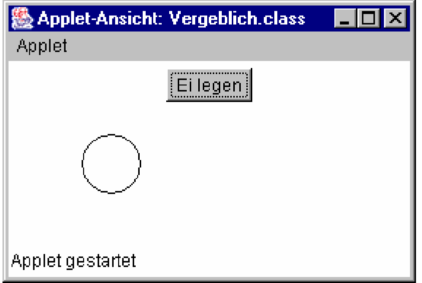
Hier wird die Klasse der individuell zu bemalenden Komponente aus **JFrame** abgeleitet.

Wenn freie Grafiken gemeinsam mit Swing-Bedienelementen in einem Fenster auftreten, sollten erstere auf einer Komponente mit **JPanel**-Abstammung untergebracht werden.

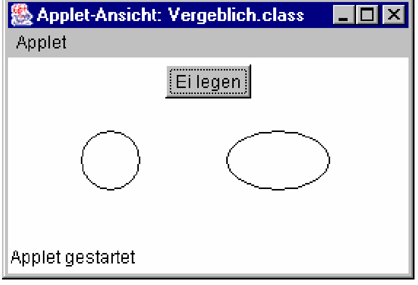
In folgendem Applet soll die Rolle der Callback-Methode **paint()** noch einmal demonstriert werden. Um in der Ereignisbehandlungsmethode **actionPerformed()** zu einem Befehlsschalter auf die Fläche des Applets ein Ei zeichnen zu können, wird in der **init()**-Methode eine Referenz auf den zugehörigen Grafikkontext mit **getGraphics()** ermittelt und in der Instanzvariablen `mg` gespeichert:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Vergeblich extends Applet implements ActionListener {
   private Graphics mg;
   private Button ei = new Button("Ei legen");

   public void init() {
      setBackground(Color.WHITE);
      mg = getGraphics();
      ei.addActionListener(this);
      add(ei);
   }

   public void paint(Graphics g) {
      g.drawOval(50, 50, 40, 40);
   }

   public void actionPerformed(ActionEvent e) {
      mg.drawOval(150, 50, 70, 40);
   }
}
```

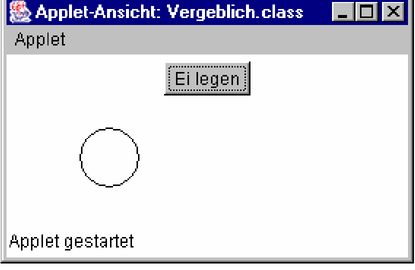
Ein Mausklick auf den Schalter hat zunächst den gewünschten Effekt, doch ist die Freude nur von kurzer Dauer:



Nach dem Start



Unmittelbar nach einem Mausklick auf den Schalter



Nach einer Größenänderung des Fensters

Was dauerhaft auf einer Komponente sichtbar sein soll, muss mit deren **paint**()-Methode gemalt werden. Gleich erfahren Sie, wie ein Programm bzw. Applet das Aktualisieren der Grafikanzeige veranlassen kann, ohne die Methode **paint()** direkt aufzurufen.

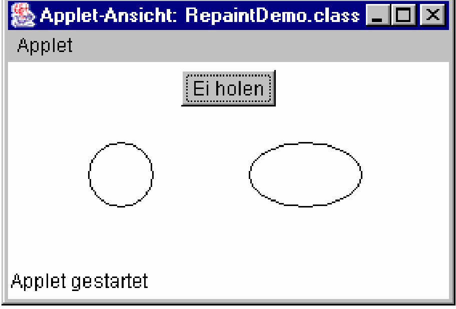
13.1.3.2 Programm-initiierte Aktualisierung

Ein Programm oder Applet kann jederzeit die Aktualisierung der Grafikanzeige veranlassen, indem es die **Component**-Methode **repaint()** für die betroffene Komponente aufruft. In der folgenden Variante des Hühner-Applets verändert die Befehlsschalter-Ereignisroutine eine boolsche Instanzvariable der Applet-Komponente und ruft dann deren **repaint**()-Methode auf:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class RepaintDemo extends Applet implements ActionListener {

   private boolean eida = false;
   private Button ei = new Button("Ei legen");
```

```
   public void init() {
      setBackground(Color.WHITE);
      ei.addActionListener(this);
      add(ei);
   }

   public void paint(Graphics g) {
      g.drawOval(50, 50, 40, 40);
      if (eida)
         g.drawOval(150, 50, 70, 40);
   }

   public void actionPerformed(ActionEvent e) {
      eida = ! eida;
      if (eida)
         ei.setLabel("Ei holen");
      else
         ei.setLabel("Ei legen");
      repaint();
   }
}
```

Diese Konstruktion zeigt ein sinnvolles Verhalten, z.B.:

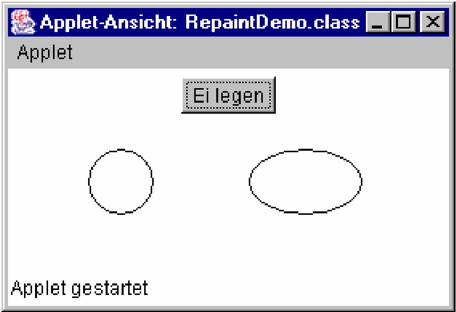


repaint()-Anforderungen gelangen in eine Warteschlange und werden (baldmöglichst) vom selben Thread (Ausführungsfaden, siehe unten) bearbeitet, der auch für die Ereignisbehandlungsmethoden zuständig ist.

Zwar führt jeder **repaint**()-Aufruf letztlich zu einem **paint**()-Aufruf, doch entfaltet **repaint**() zuvor einige, teilweise implementationsabhängige, Tätigkeiten, so dass ein direkter Aufruf der **paint**()-Methode *nicht* sinnvoll ist. Wer z.B. abenteuerlustig in obigem Applet den **repaint**()-Aufruf durch

```
paint(getGraphics());
```

ersetzt, wird beim Abholen von Eiern frustriert:



Bei einem **Applet** hat eine **repaint**()-Anforderung wie bei allen schwergewichtigen Komponenten folgende Konsequenzen:

- **repaint()** ruft die **Component**-Methode **update()** auf.
- **update()** bemalt die Komponentenfläche mit der Hintergrundfarbe, löscht also den bisherigen Inhalt.
- **update()** ruft **paint()** auf.

Beim direkten **paint()**-Aufruf wird also u.a. das automatische Löschen des bisherigen Inhalts durch die **Component**-Methode **update()** übergangen.
Auch wenn das Erhalten der alten Ausgaben ausnahmsweise einmal erwünscht ist, sollte man trotzdem *nicht* die Methode **paint()** direkt aufrufen, sondern statt dessen die Methode **update()** überschreiben.

Im Swing-API ist die Grafikaktualisierung in vielen Details anders gelöst (siehe Abschnitt 13.1.3.3); z.B. wird die **Component**-Methode **update()** für Swing-Komponenten nicht aufgerufen. Es gilt jedoch für beide APIs:

- Um die Anzeige einer Komponente zu aktualisieren, ruft man deren **repaint()**-Methode auf.
- Keinesfalls sollte man die **paint()**-Methode direkt aufrufen.

An Stelle der parameterfreien **repaint()**-Methode ist zur Beschleunigung der Grafikausgabe oft eine Überladung mit Angabe des tatsächlich aktualisierungsbedürftigen Rechtecks zu bevorzugen:

void repaint(int *x*, **int** *y*, **int** *width*, **int** *height*)

13.1.3.3 Details zur Grafikausgabe in Swing

Durch die mit Swing eingeführten Neuerungen (z.B. Komponenten-Umrahmungen, wählbares *Look & Feel*, automatische Doppelpufferung gegen das Flackern bei der Bildschirmausgabe) wurden einige Erweiterungen der Grafikausgabe erforderlich (siehe Fowler 2003).
Die **paint()**-Implementation der Klasse **javax.swing.JComponent** ruft nacheinander folgende Methoden auf:

- **paintComponent()**
- **paintBorder()**
  Zeichnet den Rahmen der Komponente.
- **paintChildren()**
  Zeichnet die enthaltenen Komponenten.

Bei den meisten Swing-Komponenten wird das *Look&Feel* (das Erscheinungsbild und das Interaktionsverhalten) durch ein separates Objekt aus der Klasse **ComponentUI** (*UI delegate*) implementiert, auf das die Instanzvariable **ui** zeigt. In diesem Fall hat ein **paintComponent()**–Aufruf folgende Konsequenzen:

- **paintComponent()** ruft **ui.update()** auf.
- Sofern die Komponente *nicht* transparent ist, die Eigenschaft **opaque** also den voreingestellten Wert **true** besitzt, füllt **ui.update()** die Fläche der Komponente mit ihrer aktuellen Hintergrundfarbe.
- **ui.update()** ruft **ui.paint()**, und diese Methode zeichnet den Inhalt der Komponente.

Sofern die bei Swing-Komponenten voreingestellte **Doppelpufferung** nicht abgeschaltet wurde, erhalten **paintComponent()**, **paintBorder()** und **paintChildren()** einen Offscreen-Graphikkontext. Erst die fertig gezeichnete Fläche gelangt auf den Bildschirm, so dass kein lästiges Flackern auftreten kann.

Für individuelle Grafikausgaben in Java-Anwendungen mit Swing-GUI sollte (von Trivialfällen abgesehen) eine eigene Klasse aus **JPanel** abgeleitet werden:

- Bei Grafikausgaben auf der Oberfläche eines Top-Level-Containers kann es zu Überlappungen mit dort enthaltenen Komponenten kommen.
- Zwar unterscheidet sich die Klasse **JPanel** in der Java-Version 1.4 nur noch unwesentlich von ihrer Basisklasse **JComponent**, jedoch wird sie als Zeichnungsgrundlage in der Regel weiterhin bevorzugt.

Im folgenden Beispiel bietet die von **JPanel** abstammende Klasse `PaintPanel` eine Methode `wechsle()` an, um einen Kreis, ein Quadrat und einen Text unabhängig voneinander erscheinen bzw. verschwinden lassen:

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

class PaintPanel extends JPanel {
   private boolean kreisDa = false, quadratDa = false, textDa = false;

   public PaintPanel() {
      setFont(new Font("Serif", Font.BOLD, 14));
      setBorder(BorderFactory.createEtchedBorder());
   }

   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      int x = (getWidth()-174)/2;
      int y = (getHeight()-40)/2;
      if (kreisDa)
         g.drawOval(x, y, 40, 40);
      if (quadratDa) {
         g.setColor(Color.RED);
         g.fillRect(x+50, y, 40, 40);
      }
      if (textDa) {
         g.setColor(Color.BLACK);
         g.drawString("Serif fett 14", x+100, y+25);
      }
      Insets ins = getInsets();
      g.setColor(Color.YELLOW);
      g.fillRect(ins.left, ins.top, 10, 10);
      g.fillRect(ins.left, getHeight()-ins.bottom-10, 10, 10);
      g.fillRect(getWidth()-ins.right-10, ins.top, 10, 10);
      g.fillRect(getWidth()-ins.right-10, getHeight()-ins.bottom-10, 10, 10);
   }

   public void wechsle(int figur) {
      switch (figur) {
         case 1: kreisDa = !kreisDa; break;
         case 2: quadratDa = !quadratDa; break;
         case 3: textDa = !textDa; break;
      }
      repaint();
   }
}
```

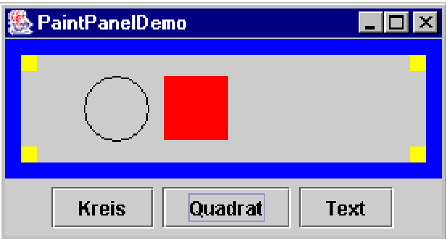
Bei der Definition einer auf **JComponent** zurück gehenden eigenen Klasse mit individueller Grafikausgabe sollte man statt **paint()** grundsätzlich die Methode **paintComponent()** überschrieben. Dann bleibt die oben beschriebene, wesentlich von der **JComponent**-Methode **paint()** realisierte Logik der Swing-Grafikausgabe erhalten (z.B. Doppelpufferung, Zeichnen von eventuell enthaltenen Kind-Komponenten).
Wer die Randzone einer Komponente individualisieren möchte, kann zusätzlich **paintBorder()** überschreiben.

Am Anfang der **paintComponent**()-Überschreibung sollte in der Regel mit

```
super.paintComponent(g);
```

die Basisklassenvariante aufgerufen werden, deren Verhalten oben skizziert wurde. Damit treffen die eigenen Grafikausgaben stets auf einen perfekt vorbereiteten Hintergrund.

Im folgenden Programm



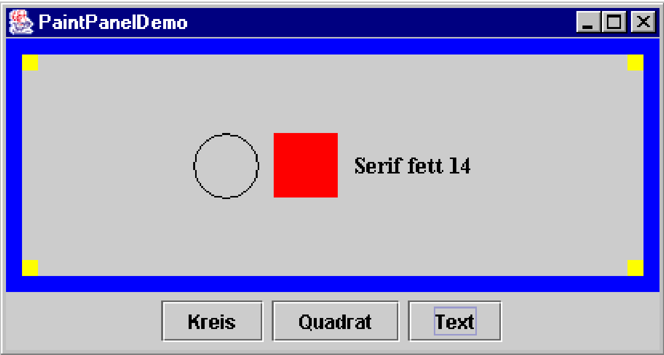
wird eine `PaintPanel`-Komponente eingesetzt und über die **actionPerformed**()-Methode zu drei Befehlsschaltern mit `wechsle()`-Botschaften versorgt:

```
public void actionPerformed(ActionEvent e) {
   if (e.getSource() == kreis) {
      tafel.wechsle(1);
   }
   else if (e.getSource() == quadrat) {
      tafel.wechsle(2);
   }
   else {
      tafel.wechsle(3);
   }
}
```

Wir fassen noch einmal zusammen, wie eine von **JComponent** abstammende eigene Komponente auf den Bildschirm gebracht wird:

- **super.paintComponent(g)** erstellt den Hintergrund.
- Es folgen die individuellen Grafikausgaben in **paintComponent**().
- **paintBorder**() zeichnet ggf. einen Rand.
- Falls vorhanden, werden via **paintChildren**() die enthaltenen Komponenten gezeichnet.

Damit die schaltbaren Grafikelemente stets zentriert erscheinen, wird in der `PaintPanel`-Methode **paintComponent**() die aktuelle Größe der Zeichenfläche mit den **JComponent**-Methoden **getWidth**() und **getHeight**() ermittelt.



Mit **getInsets**() beschafft sich **paintComponent**() ein **Insets**-Objekt, dessen Instanzvariablen **top**, **bottom**, **left** und **right** über die Breite der Ränder informieren. Damit können die vier gelben Quadrate so positioniert werden, dass sie bei beliebiger Wahl der Umrandung in den Ecken erscheinen.

13.2 Sound

Waren bis JDK 1.2 Javas Sound-Fähigkeiten auf Applets und auf das etwas spartanische Sun-Audioformat AU (Mono, 8 Bit) beschränkt, werden nun in Applets *und* Anwendungen viele gängige Audioformate unterstützt (AU, AIFF, WAV, MIDI, RMF).
In folgendem Beispiel-Applet wird eine Musikbox realisiert, die derzeit 3 Stücke vorspielen kann, aber beliebig ausbaufähig ist:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Sound extends Applet implements ActionListener {
   private Button start = new Button("Start");
   private Button wechseln = new Button("Wechseln");
   private Button stop = new Button("Stop");
   private Label label = new Label("");
   private final static int maxclip = 3;
   private int actclip = 0;
   private boolean music;
   private AudioClip clip[] = new AudioClip[maxclip];
   private String[] song = new String[maxclip];

   public void init() {
      setBackground(Color.WHITE);
      setLayout(new BorderLayout());
      start.addActionListener(this);
      wechseln.addActionListener(this);
      stop.addActionListener(this);
      add(start, BorderLayout.WEST);
      add(wechseln, BorderLayout.CENTER);
      add(stop, BorderLayout.EAST);
      add(label, BorderLayout.SOUTH);
      java.net.URL cb = getCodeBase();
      clip[0] = getAudioClip(cb,"chirp1.au"); song[0] = "Vogel-Geschwister (.au)";
      clip[1] = getAudioClip(cb,"Kuckuck.au"); song[1] = "Kuckuck (.au)";
      clip[2] = getAudioClip(cb,"trippygaia1.mid"); song[2] = "Trippygaia (mid)";
      label.setText(song[0]);
   }
   public void stop() {
      clip[actclip].stop();
   }
   public void start() {
      if (music)
         clip[actclip].loop();
   }
   public void actionPerformed(ActionEvent e) {
      if (e.getSource() == stop) {
         clip[actclip].stop();
         music = false;
      }
      else {
         if (e.getSource() == wechseln) {
              clip[actclip].stop();
            if (actclip < maxclip-1)
               actclip++;
             else
               actclip = 0;
            label.setText(song[actclip]);
         } else  // Source == start
            music = true;
         if (music)
            clip[actclip].loop();
      }
   }
}
```

Die **Applet**-Methode **getAudioClip()** erstellt aus einer Audiodatei ein Objekt aus einer Klasse[1], die das Interface **AudioClip** erfüllt. Im Beispiel werden die Audiodateien mit Hilfe eines URL-Objektes im Pfad des Applets lokalisiert:

```
java.net.URL cb = getCodeBase();
clip[0] = getAudioClip(cb,"chirp1.au");
```

Jedes **AudioClip**-Objekt repräsentiert ein Musikstück und bietet u.a. die folgenden Methoden:

- **play()**
  Einmal spielen
- **loop()**
  Endlos spielen
- **stop()**
  Wiedergabe beenden

Das Applet stellt die Beschallung ein, sobald der Browser die Methode **stop()** aufruft, und beginnt beim Aufruf der Methode **start()** wieder mit dem Abspielen des aktuellen Titels.
Ein Überschreiben der **Applet**-Methode **paint()** ist nicht erforderlich, weil sich auf der Oberfläche des Applets nur Komponenten befinden. Diese zeichnen sich selbst, sobald sie von der geerbten **paint()**-Methode dazu aufgefordert werden.

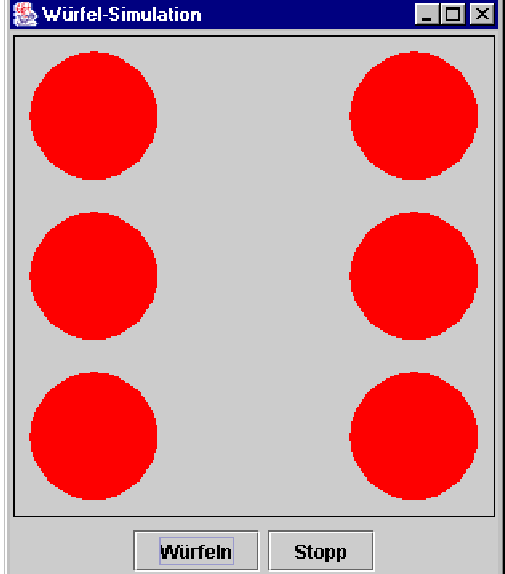
Die Bedienungsoberfläche hat noch kein disko-taugliches Design, sieht aber zumindest auf einem Macintosh schon recht passabel aus:

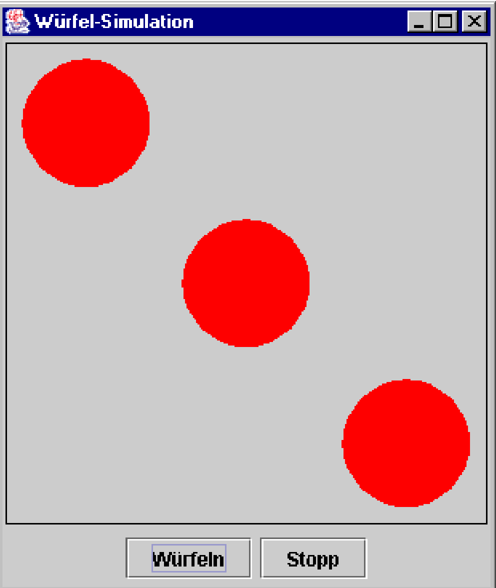
Beim Einsatz des Applets in einem Browser ist zu beachten, dass eine virtuelle Maschine mit der minimalen Java-Version 1.2 benötigt wird (vgl. Abschnitt 12.5).

[1] Es handelt sich um die Klasse **sun.applet.AppletAudioClip**, was man aber als Programmierer normalerweise nicht wissen muss.

13.3 Übungsaufgaben zu Abschnitt 13

1) Erstellen Sie ein Würfel-Simulationsprogramm nach folgendem Muster:





2) Erstellen Sie ein Applet, das zum Zählen von Ereignissen beliebiger Art über einen Schalter sowie ein Label mit aktueller Anzeige des Zählerstandes verfügt. Dem zuständigen Sachbearbeiter soll seine monotone Tätigkeit durch eine nach Belieben ein- und ausschaltbare Hintergrundmusik erleichtert werden, so dass sich ungefähr folgende Benutzeroberfläche ergibt:

14 Threads

Wir sind längst daran gewöhnt, dass moderne Betriebssysteme mehrere Programme (Prozesse) parallel betreiben können, sodass z.B. ein längerer Ausdruck keine Zwangspause zur Folge hat. Während der Druckertreiber die Ausgabeseiten aufbaut, kann z.B. ein Java-Programm entwickelt oder im Internet recherchiert werden. Sofern nur *ein* Prozessor vorhanden ist, der den einzelnen Programmen bzw. Prozessen reihum vom Betriebssystem zur Verfügung gestellt wird, reduziert sich zwar die Ausführungsgeschwindigkeit jedes Programms im Vergleich zum Solobetrieb, doch ist in den meisten Anwendungen ein flüssiges Arbeiten möglich.
Als Ergänzung zum gerade beschriebenen **Multitasking**, das ohne Zutun der Programmierer vom Betriebssystem bewerkstelligt wird, ist es oft sinnvoll oder gar unumgänglich, auch *innerhalb* einer Anwendung nebenläufige *Ausführungsfäden* zu realisieren, wobei man hier vom **Multithreading** spricht. Bei einem aktuellen Internet-Browser muss man z.B. nicht untätig den quälend langsamen Aufbau einer Seite abwarten, sondern kann in einem anderen Browser-Fenster Suchbegriffe eingeben etc.
Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die *Threads* eines Programms im selben Adressraum ab, so dass sie gelegentlich auch als *leichtgewichtige Prozesse* bezeichnet werden. Sie haben z.B. einen gemeinsamen Heap, wohingegen jeder Thread als selbständiger Kontrollfluss bzw. Ausführungsfaden aber einen eigenen Stack benötigt.

Vor der Einführung von Java gehörte die Unterstützung von Threads zur Guru-HighTech-Programmierung, weil man die Single-Thread-Architektur von Programmiersprachen wie C/C++ erst durch direkte Betriebssystemaufrufe erweitern konnte. In Java ist die Multithread-Unterstützung in die Sprache (genauer: in das API) eingebaut und von jedem Programmierer ohne große Probleme zu nutzen.

14.1 Threads erzeugen

Ein Thread ist in Java als Objekt der gleichnamigen Klasse bzw. einer Unterklasse realisiert. Im ersten Beispiel werden die Klassen `ProThread` und `KonThread` aus der Klasse **Thread** abgeleitet. Sie sollen einen Produzenten und einen Konsumenten modellieren, die gemeinsam auf einen Lagerbestand einwirken, der von einem Objekt der Klasse `Lager` gehütet wird:

```
public class Lager {
  private int bilanz;
  private int anz;
  private static final int MANZ = 20;
  private static final int STARTKAP = 100;

  public boolean offen() {
    if (anz < MANZ)
      return true;
    else {
      System.out.println("\nLieber "+Thread.currentThread().getName()+
        ", es ist Feierabend!");
      return false;
    }
  }

  private String formZeit() {
    java.sql.Time t = new java.sql.Time(System.currentTimeMillis());
    return t.toString();
  }
```

```
   public void ergaenze(int add) {
      bilanz += add;
      anz++;
      System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
         " ergaenzt\t"+add+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
   }

   public void liefere(int sub) {
      bilanz -= sub;
      anz++;
      System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
         " entnimmt\t"+sub+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
   }

   public Lager(int start) {
      bilanz = start;
      ProThread pt = new ProThread(this);
      KonThread kt = new KonThread(this);
      pt.start();
      kt.start();
   }
   public static void main(String[] args) {
      new Lager(STARTKAP);
      System.out.println("Der Laden ist offen (Bestand = "+STARTKAP+")\n");
   }
}
```

Die **main()**-Methode beschränkt sich im Wesentlichen darauf, ein `Lager`-Objekt zu erzeugen. Im `Lager`-Konstruktor wird ein `ProThread`- und ein `KonThread`-Objekt generiert:

```
ProThread pt = new ProThread(this);
KonThread kt = new KonThread(this);
```

Weil beide Threads mit dem `Lager`-Objekt kooperieren sollen, erhalten sie als Konstruktor-Parameter eine entsprechende Referenz über das Schlüsselwort **this**.

Anschließend werden die beiden Threads vom Zustand **new** durch Aufruf ihrer **start()**-Methode in der Zustand **ready** gebracht:

```
pt.start();
kt.start();
```

Von der **start()**-Methode eines Threads wird seine **run()**-Methode aufgerufen, welche die im Thread auszuführenden Anweisungen enthält. Eine aus **Thread** abgeleitete Klasse muss also vor allem die **run()**-Methode überschreiben, z.B.:

```
public class ProThread extends Thread {
   private Lager pl;

   public ProThread(Lager pl_) {
      super("Produzent");
      pl = pl_;
   }
   public void run() {
      while (pl.offen()) {
         pl.ergaenze((int) (Math.random()*100));
         try {
            Thread.sleep((int) (1000 + Math.random()*3000));
         } catch(InterruptedException ie) {
            System.err.println(ie.toString());}
      }
   }
}
```

Im Beispiel enthält die **run()**-Methode eine **while**-Schleife, die bis zum Eintreten einer Terminierungsbedingung läuft.
Ein Thread im Zustand **ready** wartet auf die Zuteilung der (bzw. einer) CPU durch das Laufzeitsystem und erreicht dann den Zustand **running**. Die VJM arbeitet **preemtiv**, d.h. sie kann den Threads die Rechenerlaubnis jederzeit entziehen. Damit wechseln die Threads in Abhängigkeit von den Vergaberichtlinien des Laufzeitsystems zwischen den Zuständen **ready** und **running**. (siehe Abschnitt 14.6.1).
Sobald seine **run()**-Methode abgearbeitet ist, endet ein Thread. Er befindet sich dann im Zustand **dead** und kann *nicht* erneut gestartet werden.

Im Beispiel ergänzt der `ProThread` innerhalb einer **while**-Schleife das Lager um eine zufallsbestimmte Menge. Er spricht über die per Konstruktor beschaffte Referenz das `Lager`-Objekt an und ruft dessen `ergaenze()`-Methode auf:

```
pl.ergaenze((int) (Math.random()*100));
```

Anschließend legt er sich durch Aufruf der statischen **Thread**-Methode **sleep()** ein (wiederum zufallsabhängiges) Weilchen zur Ruhe:

```
Thread.sleep((int) (1000 + Math.random()*3000));
```

Durch Ausführen dieser Methode wechselt der Thread vom Zustand **running** zum Zustand **sleeping**.
Weil die Methode **sleep()** potentiell die obligatorische Ausnahme **InterruptedException** wirft, wenn ein anderer Thread oder der Benutzer den Schlafenden per Unterbrechungs-Aufforderung aufweckt, ist eine Ausnahmebehandlung erforderlich. Im Beispiel wird die Ausnahme auf der Standardfehlerausgabe protokolliert. Schläft ein Thread, während der Benutzer das Programm bzw. Applet abbricht, erscheint dort z.B.:

```
java.lang.InterruptedException: sleep interrupted
```

Die in Abschnitt 14.5 näher zu beschreibende **Thread**-Methode **interrupt()** wird oft dazu eingesetzt, einen per **sleep()** in den Schlaf oder per **wait()** (siehe Abschnitt 14.2.2) in den Wartezustand geschickten Thread über das beschriebene Exception-Verfahren wieder aufzuwecken. In der Ausnahmebehandlung muss der Programmierer dann geeignet reagieren, was bei unseren Übungsprogrammen aber nicht erforderlich ist.

Zum `ProThread`-Konstruktor ist noch anzumerken, dass im Aufruf des Superklassen-Konstruktors ein Thread-Name festgelegt wird.

Der Konsumenten-Thread ist weitgehend analog definiert:

```
public class KonThread extends Thread {
  private Lager pl;

  public KonThread(Lager pl_) {
    super("Konsument");
    pl = pl_;
  }
  public void run() {
    while (pl.offen()) {
      pl.liefere((int) (5 + Math.random()*100));
      try {
          Thread.sleep((int) (1000 + Math.random()*3000));
      } catch(InterruptedException ie){
          System.err.println(ie.toString());}
    }
  }
}
```

In beiden **run()**-Methoden wird vor jedem Schleifendurchgang geprüft, ob das Lager noch offen ist. Nach Dienstschluss (im Beispiel: nach 20 Arbeitsgängen) enden beide **run()**-Methoden und damit auch die Threads.
Der automatisch zur Startklasse kreierte Thread **main** ist zu diesem Zeitpunkt ebenfalls bereits Geschichte, weil er mit der `Lager`-Methode **main()** seine Tätigkeit einstellt, so dass die gesamte Anwendung endet.[1]

In den beiden Ausführungsfäden `Produzent` bzw. `Konsument`, die von einem `ProThread` bzw. einem `KonThread`-Objekt begründet werden, kommt das `Lager`-Objekt wesentlich zum Einsatz:

- In seiner Methode `offen()` entscheidet es auf Anfrage, ob weitere Veränderungen des Lagers möglich sind.
- Die Methoden `ergaenze()` und `liefere()` erhöhen oder reduzieren den Lagerbestand, aktualisieren die Anzahl der Lagerveränderungen und protokollieren jede Maßnahme.
  Dazu besorgen Sie sich mit der stationären **Thread**-Methode **currentThread()** eine Referenz auf den aktuell ausgeführten Thread und stellen per **getName()** dessen Namen fest.
- Mit Hilfe der privaten `Lager`-Methode `formZeit()` erhält das Ereignis-Protokoll bequem formatierte Zeitangaben.

In einem typischen Ablaufprotokoll des Programms zeigen sich einige Ungereimtheiten, verursacht durch das unkoordinierte Agieren der beiden Threads:

```
Der Laden ist offen (Bestand = 100)

Nr. 2:  Konsument entnimmt      7       um 00:30:55 Uhr. Stand: 121
Nr. 1:  Produzent ergaenzt      28      um 00:30:55 Uhr. Stand: 121
Nr. 3:  Produzent ergaenzt      97      um 00:30:58 Uhr. Stand: 218
Nr. 4:  Konsument entnimmt      99      um 00:30:59 Uhr. Stand: 119
Nr. 5:  Produzent ergaenzt      37      um 00:31:00 Uhr. Stand: 156
Nr. 6:  Konsument entnimmt      74      um 00:31:02 Uhr. Stand: 82
Nr. 7:  Produzent ergaenzt      29      um 00:31:03 Uhr. Stand: 111
Nr. 8:  Konsument entnimmt      93      um 00:31:04 Uhr. Stand: 18
Nr. 9:  Produzent ergaenzt      5       um 00:31:06 Uhr. Stand: 23
Nr. 10: Konsument entnimmt      55      um 00:31:06 Uhr. Stand: -32
Nr. 11: Produzent ergaenzt      38      um 00:31:08 Uhr. Stand: 6
Nr. 12: Konsument entnimmt      66      um 00:31:08 Uhr. Stand: -60
Nr. 13: Produzent ergaenzt      66      um 00:31:11 Uhr. Stand: 6
Nr. 14: Konsument entnimmt      43      um 00:31:11 Uhr. Stand: -37
Nr. 15: Produzent ergaenzt      21      um 00:31:13 Uhr. Stand: -16
Nr. 16: Konsument entnimmt      20      um 00:31:14 Uhr. Stand: -36
Nr. 17: Produzent ergaenzt      67      um 00:31:14 Uhr. Stand: 31
Nr. 18: Konsument entnimmt      70      um 00:31:15 Uhr. Stand: -39
Nr. 19: Konsument entnimmt      99      um 00:31:17 Uhr. Stand: -138
Nr. 20: Produzent ergaenzt      12      um 00:31:17 Uhr. Stand: -126

Lieber Konsument, es ist Feierabend!

Lieber Produzent, es ist Feierabend!
```

[1] Nachdem Sie ein Java-Programm aus einer Konsole gestartet haben, können Sie unter Windows mit der Tastenkombination **<Strg>+<Unterbr>** eine Liste seiner aktiven Threads anfordern.

U.a. fällt negativ auf:

- Im ersten Protokolleintrag wird berichtet, dass vom Startwert 100 ausgehend eine Entnahme von 7 Einheiten zu einem Bestand von 121 Einheiten führt. Der zweite Eintrag lässt vermuten, dass die Ergänzung von 28 Einheiten ohne Effekt auf den Lagerbestand bleibt.
- Zwischenzeitlich wird der Bestand mehrmals negativ, was in einem realen Lager nicht passieren kann.

14.2 Threads synchronisieren

14.2.1 Monitore

Am Anfang des eben wiedergegebenen Ablaufprotokolls stehen zwei „wirre" Einträge, die folgendermaßen zu erklären sind:

- Der (zuerst gestartete) Produzenten-Thread hat die Methode `ergaenze()` in Angriff genommen und die Anweisung

  ```
  bilanz += add;
  ```

  ausgeführt, was zum Zwischenergebnis `bilanz = 128` führte.
- Dann musste der Produzent seine Arbeit unterbrechen, weil der Konsumenten-Thread vom Laufzeitsystem aktiviert, d.h. vom Zustand **ready** in den Zustand **running** befördert wurde.
- Mit seiner Anforderung von 7 Einheiten hat der Konsument in der Methode `liefere()` die Lagerbilanz auf 121 gebracht. Im Unterschied zum Produzenten wurde der Konsument bei seinem Lagerzugriff *nicht* unterbrochen, so dass nach

  ```
  bilanz -= sub;
  ```

  auch noch die Protokollierungs-Anweisung ausgeführt wurde. Allerdings ist der Stand von 121 nicht nachvollziehbar, weil die vorherige Ergänzung nicht protokolliert worden war.
- Nach dem nächsten Thread-Wechsel macht der Produzent mit der Protokollausgabe weiter, wobei aber der *aktuelle* `bilanz`-Wert (unter Berücksichtigung der zwischenzeitlichen Konsumenten-Aktivität) erscheint.

Offenbar muss verhindert werden, dass zwei Threads simultan auf das Lager zugreifen. Genau dies ist mit dem von Java unterstützten **Monitor**-Konzept leicht zu realisieren.
Zu einem *Monitor* kann jedes Objekt werden, sobald eine seiner Methoden den Modifikator **synchronized** erhält. Sobald ein Thread eine als **synchronized** deklarierte Methode betritt, wird er zum Besitzer dieses Monitors. Man kann sich auch vorstellen, dass er den (einzigen) Schlüssel zu den synchronisierten Bereichen des Monitors an sich nimmt. In der englischen Literatur wird der Vorgang als *obtaining the lock* beschrieben. Jedem anderen Thread wird der Zutritt zum Monitor verwehrt und er muss warten.
Sobald der Monitor-Besitzer die **synchronized**-Methode beendet, kann ein wartender Thread den Monitor übernehmen und seine Arbeit fortsetzen.
In unserem Beispiel sollten die `Lager`-Methoden `offen()`, `ergaenze()` und `liefere()` als **synchronized** deklariert werden, z.B.:

```
public synchronized void ergaenze(int add) {
  bilanz += add;
  anz++;
  System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
    " ergaenzt\t"+add+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
}
```

Nun unterbleiben die wirren Protokolleinträge, doch die Ausflüge in negative Lagerzustände finden nach wie vor statt:

```
Der Laden ist offen (Bestand = 100)

Nr. 1:  Produzent ergaenzt      36      um 01:44:52 Uhr. Stand: 136
Nr. 2:  Konsument entnimmt      68      um 01:44:52 Uhr. Stand: 68
Nr. 3:  Produzent ergaenzt      23      um 01:44:53 Uhr. Stand: 91
Nr. 4:  Konsument entnimmt      81      um 01:44:54 Uhr. Stand: 10
Nr. 5:  Produzent ergaenzt      8       um 01:44:55 Uhr. Stand: 18
Nr. 6:  Konsument entnimmt      29      um 01:44:56 Uhr. Stand: -11
Nr. 7:  Produzent ergaenzt      90      um 01:44:57 Uhr. Stand: 79
Nr. 8:  Konsument entnimmt      68      um 01:44:58 Uhr. Stand: 11
Nr. 9:  Produzent ergaenzt      38      um 01:45:00 Uhr. Stand: 49
Nr. 10: Konsument entnimmt      14      um 01:45:01 Uhr. Stand: 35
Nr. 11: Produzent ergaenzt      34      um 01:45:02 Uhr. Stand: 69
Nr. 12: Konsument entnimmt      33      um 01:45:04 Uhr. Stand: 36
Nr. 13: Produzent ergaenzt      30      um 01:45:05 Uhr. Stand: 66
Nr. 14: Konsument entnimmt      27      um 01:45:07 Uhr. Stand: 39
Nr. 15: Produzent ergaenzt      63      um 01:45:08 Uhr. Stand: 102
Nr. 16: Produzent ergaenzt      18      um 01:45:09 Uhr. Stand: 120
Nr. 17: Konsument entnimmt      18      um 01:45:10 Uhr. Stand: 102
Nr. 18: Konsument entnimmt      93      um 01:45:12 Uhr. Stand: 9
Nr. 19: Produzent ergaenzt      62      um 01:45:12 Uhr. Stand: 71
Nr. 20: Konsument entnimmt      100     um 01:45:13 Uhr. Stand: -29

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

Neben dem **synchronized**-Modifikator für Methoden bietet Java auch den synchronisierten *Block*, wobei statt einer kompletten Methode nur eine einzelne Blockanweisung in den synchronisierten Bereich aufgenommen und ein beliebiges Objekt als Monitor angegeben wird. Diese Möglichkeit wird in Abschnitt 14.4 vorgestellt, wo die Synchronisations-Techniken zum Unterbrechen von Threads eingesetzt werden.

14.2.2 Koordination per wait() und notify()

Mit Hilfe der **Object**-Methoden **wait()** und **notify()** können negative Lagerbestände in unserem Produzenten-Lager-Konsumenten-Beispiel verhindert werden:
Trifft eine Konsumenten-Anfrage auf einen unzureichenden Lagerbestand, dann wird der Thread mit der Methode **wait()** in den Zustand **waiting** versetzt. Die Methode **wait()** ist bereits in der Urahnklasse **Object** definiert und kann nur in einem Monitor, also im synchronisierten Bereich, aufgerufen werden, z.B.:

```
public synchronized void liefere(int sub) {
  while (bilanz < sub)
    try {
      System.out.println(Thread.currentThread().getName()+
        " muss warten: Keine "+sub+" Einheiten vorhanden.");
      wait();
    } catch (InterruptedException ie) {
        System.err.println(ie.toString());
    }

  bilanz -= sub;
  anz++;
  System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
    " entnimmt\t"+sub+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
}
```

Mit den **Object**-Methoden **notify()** bzw. **notifyAll()** kann ein Thread aus dem Zustand **waiting** in den Zustand **ready** versetzt werden:

- **notify()** versetzt den Thread mit der längsten Wartezeit in den Zustand **ready**.
- **notifyAll()** versetzt alle wartenden Threads in den Zustand **ready**.

Wie **wait()** können auch **notify()** und **notifyAll()** nur in einem synchronisierten Bereich aufgerufen werden, z.B.:

```
public synchronized void ergaenze(int add) {
  bilanz += add;
  anz++;
  System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
    " ergaenzt\t"+add+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
  notify();
}
```

Nun produziert das Beispielprogramm nur noch realistische Lagerprotokolle, z.B.:

```
Der Laden ist offen (Bestand = 100)

Nr. 1:  Konsument entnimmt      46      um 10:28:39 Uhr. Stand: 54
Nr. 2:  Produzent ergaenzt      54      um 10:28:39 Uhr. Stand: 108
Nr. 3:  Konsument entnimmt      57      um 10:28:41 Uhr. Stand: 51
Konsument muss warten: Keine 97 Einheiten vorhanden.
Nr. 4:  Produzent ergaenzt      12      um 10:28:43 Uhr. Stand: 63
Konsument muss warten: Keine 97 Einheiten vorhanden.
Nr. 5:  Produzent ergaenzt      77      um 10:28:46 Uhr. Stand: 140
Nr. 6:  Konsument entnimmt      97      um 10:28:46 Uhr. Stand: 43
Nr. 7:  Produzent ergaenzt      10      um 10:28:49 Uhr. Stand: 53
Nr. 8:  Konsument entnimmt      19      um 10:28:50 Uhr. Stand: 34
Nr. 9:  Produzent ergaenzt      53      um 10:28:53 Uhr. Stand: 87
Nr. 10: Konsument entnimmt      72      um 10:28:53 Uhr. Stand: 15
Nr. 11: Produzent ergaenzt      58      um 10:28:55 Uhr. Stand: 73
Nr. 12: Konsument entnimmt      27      um 10:28:55 Uhr. Stand: 46
Konsument muss warten: Keine 47 Einheiten vorhanden.
Nr. 13: Produzent ergaenzt      104     um 10:28:57 Uhr. Stand: 150
Nr. 14: Konsument entnimmt      47      um 10:28:57 Uhr. Stand: 103
Nr. 15: Produzent ergaenzt      40      um 10:29:01 Uhr. Stand: 143
Nr. 16: Konsument entnimmt      34      um 10:29:01 Uhr. Stand: 109
Nr. 17: Konsument entnimmt      17      um 10:29:04 Uhr. Stand: 92
Nr. 18: Produzent ergaenzt      65      um 10:29:05 Uhr. Stand: 157
Nr. 19: Produzent ergaenzt      31      um 10:29:07 Uhr. Stand: 188
Nr. 20: Konsument entnimmt      76      um 10:29:07 Uhr. Stand: 112

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

14.3 Das Interface Runnable

In unseren bisherigen Beispielen waren die Baupläne der als Thread ausgeführten Objekte aus der Klasse **Thread** abgeleitet. Oft benötigt man aber eine Thread-fähige Klasse, die einen anderen Stammbaum besitzt, z.B. aus **(J)Applet** abgeleitet ist. In einer solchen Situation, die in anderen Programmiersprachen möglicherweise durch Mehrfachvererbung gelöst wird, kommt in Java ein Interface zum Einsatz.
In konkreten Fall heißt es **Runnable** und verlangt von einer implementierenden Klasse lediglich eine **run()**-Methode mit dem vom Thread auszuführende Code. Beim *Erzeugen* eines Threads wird die Klasse **Thread** aber doch benötigt, z.B.:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Laufschrift extends Applet implements Runnable {
   private final String TXT = "Wo laufen sie denn? Wo laufen sie denn hin?";
   private final int PAUSE  = 10;
   private final int SPRUNG = 2;
   private final int HOEHE  = 10;

   Thread laufband;
   private int x, y,     // akt. Schreibpos.
               tb;       // Textbreite

   private final String LABELPREFIX = "Anzahl der Klicks: ";
   private Label lab;
   private int numClicks = 0;
   private Button butt;

   public void init() {
      setBackground(Color.YELLOW);
      lab = new Label(LABELPREFIX + "0     ");
      butt = new Button("Klicker");
      y =  getSize().height-HOEHE;
      tb = getFontMetrics(getFont()).stringWidth(TXT);
      add(butt);
      add(lab);
      butt.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            numClicks++;
            lab.setText(LABELPREFIX + numClicks);
         }
      });
      laufband = new Thread(this, "Laufband");
      laufband.start();
   }

   public void paint(Graphics g){
      g.drawString(TXT, x, y);
   }

   public void run() {
      while (true) {
         try {
            Thread.sleep(PAUSE);
         } catch(Exception e) {
            return;
         }
         if (x + tb < 0)
            x = getSize().width;
         else
            x -= SPRUNG;
         repaint();
      }
   }

}
```

Man erzeugt ein **Thread**-Objekt per Konstruktor mit **Runnable**-Parameter:

```
Thread(Runnable target)
Thread(Runnable target, String name)
```

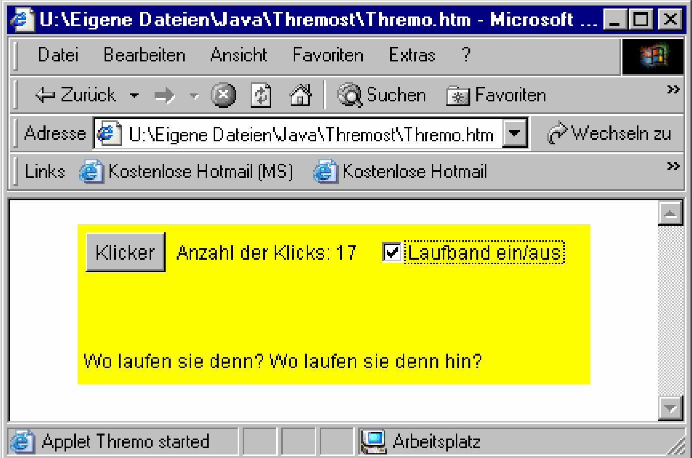
und übergibt als Aktualparameter ein Objekt, dessen **run()**-Methode im Thread ausgeführt werden soll. Optional kann man zusätzlich den Namen des neuen Threads festlegen.

In obigem Beispiel beschäftigt sich ein Thread damit, einen Text kontinuierlich über die Applet-Fläche laufen zu lassen. Gleichzeitig interagiert der **main**-Thread mit dem Benutzer, der z.B. vorbeifahrende Autos per Mausklick zählen kann:

Das Applet ist noch nicht ganz ausgereift; z.B. sollte durch Doppelpufferung das deutliche Flickern des Lauftextes reduziert werden. Wie dies durch Verwendung von Swing-Komponenten sehr bequem geschehen kann, haben Sie im Abschnitt über die zweidimensionale Grafikausgabe erfahren.

14.4 Threads unterbrechen

Zum Unterbrechen und Reaktivieren eines Threads sollten an Stelle der **Thread**-Methoden **suspend()** und **resume()**, die als missbilligt (deprecated) eingestuft sind, die **Object**-Methoden **wait()** und **notify()** eingesetzt werden, die wir schon zur Synchronisation von Threads verwendet haben.
In folgender Variante des Applets aus Abschnitt 14.3 wird ein Kontrollkästchen angeboten, um den Laufschrift-Thread ein- oder auszuschalten:



In der boolschen Instanzvariablen `lban` wird der aktuelle Schaltzustand des Kontrollkästchens gespeichert. Wird in der **run()**-Methode des Threads der `lban`-Wert **false** angetroffen, dann begibt sich der Thread freiwillig in den Wartezustand. Wie Sie bereits wissen, darf die Methode **wait()** nur in einem synchronisierten Bereich aufgerufen werden. Eine **run()**-Methode als **synchronized** zu deklarieren, ist zumindest dann wenig sinnvoll, wenn mehrere Thread-Instanzen mit dieser Methode arbeiten sollen.

Als Alternative zum **synchronized**-Modifikator für Methoden steht in Java auch der **synchronisierte Block** zur Verfügung, wobei statt einer kompletten Methode nur eine einzelne Blockanweisung in den synchronisierten Bereich aufgenommen wird, z.B.:

```
public void run() {
  while (true) {
    try {
      Thread.sleep(PAUSE);
      synchronized(this) {
        while (!lban)
          wait();
      }
    } catch(InterruptedException ie) {
      System.err.println(ie.toString());
    }
    if (x + tb < 0)
      x = getSize().width;
    else
      x -= SPRUNG;
    repaint();
  }
}
```

Nach dem Schlüsselwort **synchronized** wird in runden Klammern ein beliebiges Objekt als Monitor (Lock-Objekt) für die Synchronisation angegeben. Im Beispiel übernimmt das Applet-Objekt diese Rolle selbst.
Die **notify**()-Botschaft zum Reaktivieren des wartenden Threads wird im Beispiel von der Ereignisbehandlungsmethode **itemStateChanged()** zum Kontrollkästchen abgeschickt:

```
public synchronized void itemStateChanged(ItemEvent e) {
  lban = !lban;
  if (lban)
    notify();
  else
    repaint();
}
```

Dieses **notify()** ist an dasjenige Lock-Objekt zu richten, auf das der Thread wartet. Dies ist gewährleistet, wenn die **notify**()-Methode in einem synchronisierten Bereich dieses Lock-Objektes aufgerufen wird. In unserem Fall ist das Applet-Objekt der verantwortliche Monitor. Weil das selbe Objekt auch als Ereignisempfänger für das Kontrollkästchen tätig ist, sorgen wir mit einem **synchronized**-Modifikator für die Ereignisbehandlungsmethode für den korrekten **notify**()-Adressaten.
Wäre als Ereignisempfänger für das Kontrollkästchen z.B. ein Objekt einer inneren Klasse tätig, bliebe der **notify**()–Aufruf in der obigen **itemStateChanged**()-Implementation ohne Effekt.

14.5 Threads stoppen

Zum Stoppen eines Threads sollte man nicht mehr die unerwünschte **Thread**-Methode **stop()** verwenden, sondern ein behutsames, kommunikatives Prozedere nutzen:

- Mit der **Thread**-Methode **interrupt**() wird signalisiert, dass ein Thread seine Tätigkeit einstellen soll. Der betroffene Thread wird nicht abgebrochen, sondern sein Interrupt-Signal wird auf den Wert **true** gesetzt.
- Ein gut erzogener Thread, der mit einem Interrupt-Signal rechnen muss, prüft in seiner **run**()-Methode regelmäßig durch Aufruf der **Thread**-Methode **isInterrupted**(), ob er sein Wirken einstellen soll. Falls ja, verlässt er einfach die **run**()-Methode und erreicht damit den Zustand **dead**.

Als Beispiel soll eine etwas primitive Variante des Produzenten-Lager-Konsumenten-Beispiels dienen, wobei der Lagerverwalter den Konsumenten-Thread zum Terminieren auffordert, sobald dieser mit einem Wunsch über den Lagerbestand hinaus geht:

```
public void liefere(int sub) {
   if (bilanz - sub < 0)
      Thread.currentThread().interrupt();
   else {
      bilanz -= sub;
      anz++;
      System.out.println("Nr. "+anz+":\t"+Thread.currentThread().getName()+
         " entnimmt\t"+sub+"\tum "+formZeit()+" Uhr. Stand: "+bilanz);
   }
}
```

Trifft der Konsumenten-Thread auf ein Interrupt-Signal, stellt er (wenn auch etwas mürrisch) seine Tätigkeit ein:

```
public void run() {
   while (pl.offen()) {
      if (isInterrupted()) {
         System.out.println("Als Kunde muss ich mir so etwas "+
            "nicht gefallen lassen!");
         return;
      }
      pl.liefere((int) (5 + Math.random()*100));
      int limit = (int) (1000 + Math.random()*3000);
      long time = System.currentTimeMillis();
      while(System.currentTimeMillis() - time < limit);
   }
}
```

In dieser **run()**-Methode wird bewusst ein Aufruf von **Thread.sleep()** vermieden, weil ein **interrupt()**-Aufruf in der Schlafenszeit zu einer **InterruptedException** führt. Mit dieser Konstellation werden wir uns gleich beschäftigen.

In der folgenden Lager-Sequenz muss der Konsument frühzeitig aussteigen:

```
Der Laden ist offen (Bestand = 100)

Nr. 1:  Produzent ergaenzt      78      um 01:52:12 Uhr. Stand: 178
Nr. 2:  Konsument entnimmt      88      um 01:52:12 Uhr. Stand: 90
Nr. 3:  Konsument entnimmt      11      um 01:52:13 Uhr. Stand: 79
Nr. 4:  Produzent ergaenzt      43      um 01:52:15 Uhr. Stand: 122
Nr. 5:  Konsument entnimmt      47      um 01:52:16 Uhr. Stand: 75
Nr. 6:  Produzent ergaenzt      18      um 01:52:18 Uhr. Stand: 93
Als Kunde muss ich mir so etwas nicht gefallen lassen!
Nr. 7:  Produzent ergaenzt      55      um 01:52:21 Uhr. Stand: 148
Nr. 8:  Produzent ergaenzt      13      um 01:52:24 Uhr. Stand: 161
Nr. 9:  Produzent ergaenzt      75      um 01:52:26 Uhr. Stand: 236
Nr. 10: Produzent ergaenzt      6       um 01:52:28 Uhr. Stand: 242
Nr. 11: Produzent ergaenzt      17      um 01:52:31 Uhr. Stand: 259
Nr. 12: Produzent ergaenzt      36      um 01:52:32 Uhr. Stand: 295
Nr. 13: Produzent ergaenzt      48      um 01:52:35 Uhr. Stand: 343
Nr. 14: Produzent ergaenzt      31      um 01:52:39 Uhr. Stand: 374
Nr. 15: Produzent ergaenzt      35      um 01:52:41 Uhr. Stand: 409
Nr. 16: Produzent ergaenzt      97      um 01:52:45 Uhr. Stand: 506
Nr. 17: Produzent ergaenzt      99      um 01:52:46 Uhr. Stand: 605
Nr. 18: Produzent ergaenzt      17      um 01:52:48 Uhr. Stand: 622
Nr. 19: Produzent ergaenzt      37      um 01:52:50 Uhr. Stand: 659
Nr. 20: Produzent ergaenzt      64      um 01:52:53 Uhr. Stand: 723

Lieber Produzent, es ist Feierabend!
```

Angewandt auf einen per **wait()** in den Wartezustand versetzten oder per **sleep()** ruhig gestellten Thread hat **interrupt()** folgende Effekte:[1]

- Der Thread wird sofort in den Zustand **ready** versetzt, auch wenn die **sleep()**-Zeit noch nicht abgelaufen ist.
- Es wird eine **InterruptedException** geworfen, die vom **wait()**- bzw. **sleep()**-Aufrufer abgefangen werden muss.
- Das Interrupt-Signal wird ggf. *aufgehoben* (auf **false** gesetzt).

Es kann daher sinnvoll sein, **interrupt()** in der **catch**-Klausel der **InterruptedException**-Behandlung erneut aufzurufen, um das Interrupt-Signal wieder auf **true** zu setzen, damit die **run()**-Methode bei nächster Gelegenheit passend reagiert, z.B.:

```
public void run() {
  while (pl.offen()) {
    if (isInterrupted()) {
      System.out.println("Als Kunde muss ich mir so etwas "+
        "nicht gefallen lassen!");
      return;
    }
    pl.liefere((int) (5 + Math.random()*100));
    try {
      Thread.sleep((int) (1000 + Math.random()*3000));
    } catch(InterruptedException ie) {
        interrupt();
    }
  }
}
```

In unserem Beispiel findet die **InterruptedException**-Behandlung allerdings in der **run()**-Methode statt, so dass es sich anbietet, durch eine **return**-Anweisung in der **catch**-Klausel die **run()**-Methode und damit den Thread sofort zu beenden:

```
public void run() {
  . . .
    try {
      Thread.sleep((int) (1000 + Math.random()*3000));
    } catch(InterruptedException ie) {
      return;
    }
  }
}
```

14.6 Thread-Lebensläufe

In diesem Abschnitt wird zunächst die Vergabe von Arbeitsberechtigungen für konkurrierende Threads behandelt. Dann fassen wir unsere Kenntnisse über die verschiedenen Zustände eines Threads und über Anlässe für Zustandswechsel zusammen.

14.6.1 Scheduling und Prioritäten

Den Bestandteil der virtuellen Maschine, der die verfügbare Rechenzeit auf die arbeitswilligen und -fähigen Threads (Zustand **ready**) verteilt, bezeichnet man als **Scheduler**.

[1] Bei einem Thread, der auf einen *Monitor* wartet, verhält sich **interrupt()** wie zu Beginn des Abschnitts beschrieben, setzt also das Interrupt-Signal des Threads.

Er orientiert sich u.a. an den **Prioritäten** der Threads, die in Java Werte von 1 bis 10 annehmen können:

int-Konstante in der Klasse **Thread**	Wert
Thread.MAX_PRIORITY	10
Thread.NORM_PRIORITY	5
Thread.MIN_PRIORITY	1

Es hängt allerdings zum Teil von der Plattform ab, wie viele Prioritätsstufen wirklich unterschieden werden.
Der in einer Java-Anwendung automatisch gestartete Thread **main** hat z.B. die Priorität 5, was man unter Windows in einer Java-Konsolenanwendung über die Tastenkombination **<Strg>+<Pause>** in Erfahrung bringen kann, z.B.:

```
"main" prio=5 tid=0x00034D28 nid=0x1464 runnable [7f000..7fc3c]
```

Ein Thread (z.B. **main**) überträgt seine aktuelle Priorität auf die bei seiner Ausführung gestarteten Threads, z.B.:

```
"Konsument" prio=5 tid=0x00A33DF0 nid=0x220 waiting on condition [ac9f000..ac9fd88]
   .   .   .
"Produzent" prio=5 tid=0x00A33C98 nid=0x136c waiting on condition [ac5f000..ac5fd88]
```

Mit den Thread-Methoden **getPriority()** bzw. **setPriority()** lässt sich die Priorität eines Threads feststellen bzw. ändern.

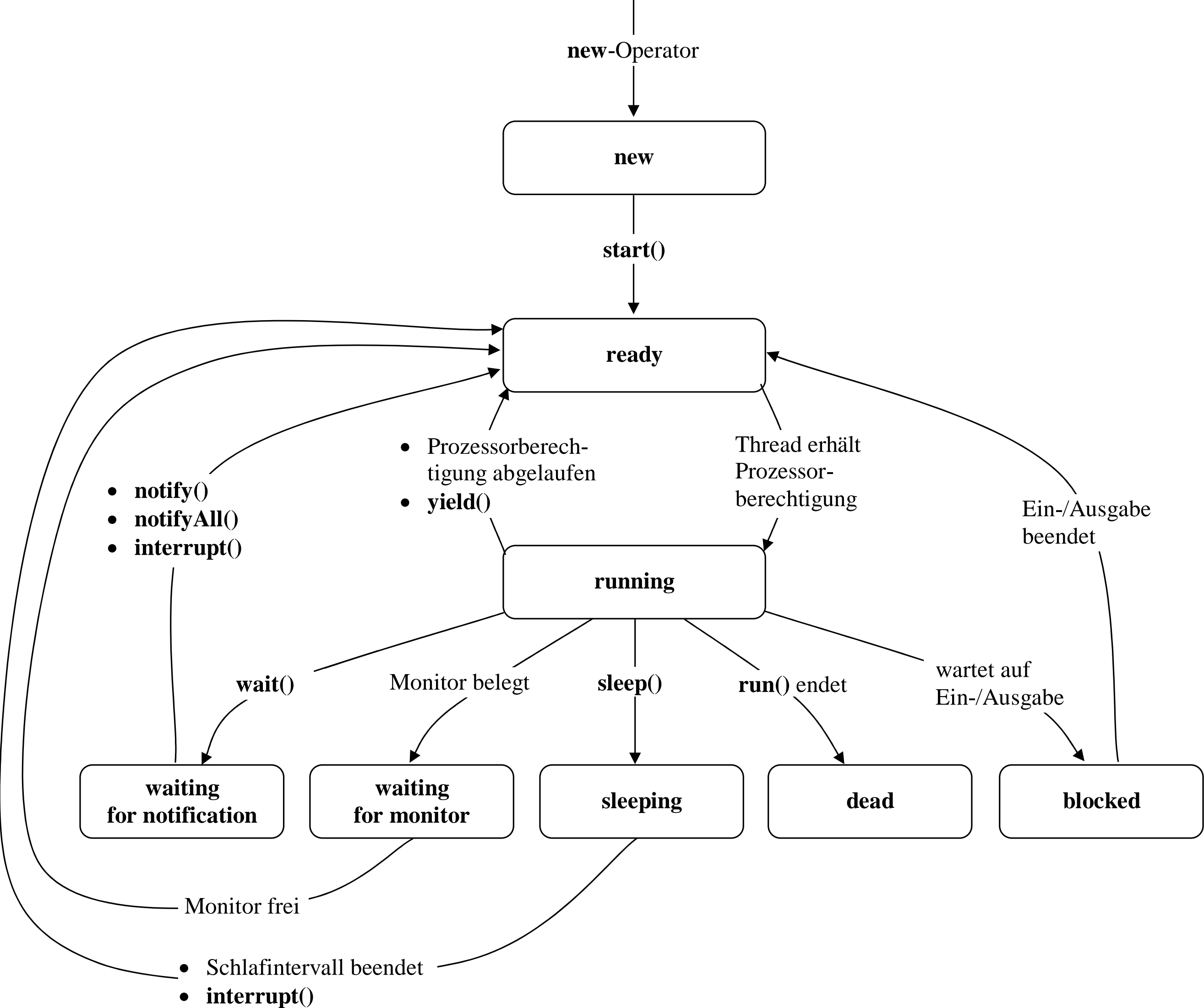
In der Spezifikation für die virtuelle Java-Maschine wird das Verhalten des Schedulers bei der Rechenzeit-Vergabe an die Threads nicht sehr präzise beschrieben. Er muss lediglich sicherstellen, dass die einem Thread zugeteilte Rechenzeit mit der Priorität ansteigt.
In der Regel läuft der Thread mit der höchsten Priorität, jedoch kann der Scheduler Ausnahmen von dieser Regel machen, z.B. um das *Verhungern* (engl. *starvation*) eines anderen Threads zu verhindern, der permanent auf Konkurrenten mit höherer Priorität trifft. Daher darf keinesfalls der korrekte Ablauf eines Pogramm davon abhängig sein, dass sich die Rechenzeit-Vergabe an Threads in einem strengen Sinn an den Prioritäten orientiert.
Leider gibt es im Verhalten des Schedulers bei Threads *gleicher* Priorität relevante Unterschiede zwischen den JVMs, so dass diesbezüglich *keine vollständige* Plattformunabhängigkeit besteht. Auf einigen Plattformen (z.B. unter MS-Windows) benutzt die JVM das von modernen Betriebsystemen zur *Prozess*verwaltung verwendete **preemtive Zeitscheibenverfahren** für die Thread-Verwaltung:

- Die Threads gleicher Priorität werden reihum (*Round-Robin*) jeweils für eine festgelegte Zeitspanne ausgeführt.
- Ist die Zeitscheibe eines Threads verbraucht, wird er vom Scheduler in den Zustand **ready** versetzt, und der Nachfolger erhält Zugang zum Prozessor.

Auf anderen Plattformen (z.B. unter Solaris) benutzt die JVM ein rein prioritätenbasiertes Multithreading, so dass ein Thread in der Regel nur von Konkurrenten mit höherer Priorität verdrängt wird.
Freundliche Threads können aber auch ohne Round-Robin-Unterstützung der JVM den gleichrangigen Kollegen eine Chance geben, indem sie bei passender Gelegenheit die Thread-Methode **yield()** aufrufen, um den Prozessor abzugeben und sich wieder in die Warteschlage der rechenwilligen Threads einzureihen. Mit Hilfe dieser Methode können also die Plattformunterschiede in der Thread-Verwaltung kompensiert werden.

14.6.2 Zustände von Threads

In der folgenden Abbildung nach Deitel & Deitel (1999, S. 738) werden wichtige Thread-Zustände und Anlässe für Zustandsübergänge dargestellt:



14.7 Sonstige Thread-Themen

14.7.1 Daemon-Threads

Neben den in Abschnitt 14 behandelten Benutzer-Threads kennt Java noch so genante Daemon-Threads, die im Hintergrund zur Unterstützung anderer Threads tätig sind und dabei nur aktiv werden, wenn ungenutzte Rechenzeit vorhanden ist. Mit dem Garbage Collector haben Sie ein typisches Exemplar dieser Gattung bereits kennen gelernt.
Mit der Thread-Methode **setDeamon()** lässt sich auch ein Benutzer-Thread dämonisieren, was allerdings vor dem Aufruf seiner **start()**-Methode geschehen muss.

Um das Terminieren von Daemon-Threads braucht man sich in der Regel nicht zu kümmern, denn ein Java-Programm oder -Applet endet, sobald ausschließlich Daemon-Threads vorhanden sind.

14.7.2 Deadlocks

Wer sich beim Einsatz von Monitoren zur Thread-Synchronisation ungeschickt anstellt, kann so genannte Deadlocks produzieren, wobei sich Threads gegenseitig blockieren.
Im folgenden Beispiel begeben sich `Thread1` und `Thread2` jeweils in einen Monitor, der durch das Synchronisieren von Klassenmethoden entsteht, so dass man der jeweiligen Klasse (`DeadLock1` bzw. `DeadLock2`) die Rolle des Aufpassers zuschreiben kann:

```
class DeadLock1 {
   synchronized static void m() {
      System.out.println(Thread.currentThread().getName()+" in DeadLock1.m");
      try {
         Thread.sleep(100);
      } catch (InterruptedException e) {}
      System.out.println(Thread.currentThread().getName()+
         " wartet darauf, dass DeadLock2.m frei wird ...");
      DeadLock2.m();
   }

   public static void main(String[] args) {
      Thread1 t1 = new Thread1();
      Thread2 t2 = new Thread2();
      t1.start();
      t2.start();
   }
}

class DeadLock2 {
   synchronized static void m() {
      System.out.println(Thread.currentThread().getName()+" in DeadLock2.m");
      try {
         Thread.sleep(100);
      } catch (InterruptedException e) {}
      System.out.println(Thread.currentThread().getName()+
         " wartet darauf, dass DeadLock1.m frei wird ...");
      DeadLock1.m();
   }
}

class Thread1 extends Thread {
   Thread1() {super("Thread1");}
   public void run() {
      DeadLock1.m();
   }
}

class Thread2 extends Thread {
   Thread2() {super("Thread2");}
   public void run() {
      DeadLock2.m();
   }
}
```

Nach einem kurzen Schläfchen versuchen die beiden Threads, eine Methode aus dem jeweils anderen (blockierten) Monitor aufzurufen. Weil kein Thread seinen Monitor frei gibt, bevor er den anderen betreten darf, kommt es zur Blockade, und das Programm hängt fest:

```
C:\WINDOWS\System32\cmd.exe - java DeadLock1

U:\Eigene Dateien\Java\Deadlock>java DeadLock1
Thread1 in DeadLock1.m
Thread2 in DeadLock2.m
Thread1 wartet darauf, dass DeadLock2.m frei wird ...
Thread2 wartet darauf, dass DeadLock1.m frei wird ...
```

14.7.3 Threads und Swing

Das Swing-API wurde (im Unterschied zum AWT-API) aus Performanzgründen für einen *Single-Thread* - Betrieb konzipiert. Greifen trotzdem mehrere Threads simultan auf eine Swing-Komponente zu, kann es zu irregulären Verhalten (z.B. bei der Bildschirmaktualisierung) kommen. Wegen der fehlenden Thread-Sicherheit sollte man den exklusiven Zugriff auf die Swing-Komponenten unbedingt dem bei GUI–Anwendungen bzw. –Applets automatisch vorhandenen **Ereignisverteilungs-Thread** (Event Dispatch - Thread) überlassen. Er informiert Komponenten über Ereignisse und fordert sie ggf. zur Aktualisierung ihrer Oberfläche auf. Weil alle Ereignisbehandlungsmethoden in diesem Thread ablaufen, sollte sich deren Zeitaufwand in Grenzen halten, weil sonst die Benutzeroberfläche zäh reagiert.
Beachten Sie also bei Verwendung des Swing-APIs unbedingt die folgende Single-Thread-Regel: Änderungen mit Auswirkung auf die Darstellung von bereits realisierten Swing-Komponenten dürfen nur im Rahmen des Event Dispatch - Threads vorgenommen werden, also in Ereignisbehandlungsmethoden. Als *realisiert* gilt eine Swing-Komponente dann, wenn für das zugehörige Top-Level-Fenster eine Methoden **setVisible()**, **show()** oder **pack()** aufgerufen worden ist.

Allerdings bietet das Swing-API etliche Möglichkeiten, Änderungen von Komponenten außerhalb des Ereignisbehandlungs-Threads zu veranlassen, ohne die obige Regel zu verletzen, z.B.:

- **repaint()**, **revalidate()**
  Diese **JComponent**-Methoden platzieren eine Aktualisierungs-Anforderung in die Ereigniswarteschlange.
- **invokeLater()**, **invokeAndWait()**
  Mit diesen statischen **SwingUtilities**-Methoden kann die **run()**-Methode eines per Parameter übergebenen **Runnable**-Objektes asynchron im Ereignisverteilungs-Thread ausgeführt werden.
- **Timer**-Objekte
  Objekte der Klasse **javax.swing.Timer** eignen sich für Aufgaben, die nach Ablauf einer bestimmten Zeitspanne oder regelmäßig ausgeführt werden sollen, z.B.:

  ```
  final int delay = 1000;
  ActionListener timerListener = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
          //.die regelmäßig auszuführenden Aktionen
      }
  };
  new Timer(delay, timerListener).start();
  ```

  Die vom **Timer**-Objekt über **action events** aktivierten Methoden dürfen nicht zu aufwändig sein, weil sie im Ereignisbehandlungs-Thread ablaufen und das GUI behindern können.

14.8 Zusammenfassung

Ein Thread ist ein selbständiger Ausführungsfaden (Kontrollfluss) innerhalb eines Java-Programms bzw. –Applets. Alle Threads eines Programms laufen im selben Adressraum. Sie teilen sich den Heap (mit den Objekten des Programms), haben jedoch jeweils einen eigenen Stack.
Bei der Vergabe von Rechenzeit an die Threads berücksichtigt der Scheduler der JVM deren Prioritäten und wendet meist für Threads gleicher Priorität ein Zeitscheibenverfahren an.

Zum Erzeugen eines Threads bietet Java zwei Optionen:

- Eine eigene Klasse aus **java.lang.Thread** ableiten und ein Objekt dieser Klasse erzeugen
- In einer eigenen Klasse das Interface **java.lang.Runnable** implementieren und ein Objekt dieser Klasse an einen **Thread**-Konstruktor übergeben

Über *Monitore* kann man verhindern, dass ein Code-Bereich von mehreren Threads parallel durchlaufen wird:

- Sobald ein Thread eine als **synchronized** definierte Instanzmethode für ein Objekt aufruft, ignoriert dieses Objekt jeden Aufruf einer **synchronized**-Instanzmethode durch einen anderen Thread, bis der erste Thread den Monitor verlassen hat.
- Sobald ein Thread eine als **synchronized** definierte statische Methode einer Klasse aufruft, sind alle statischen **synchronized**-Methoden der Klasse für alle anderen Threads gesperrt, bis der erste Thread den Monitor verlassen hat.
- Sobald ein Thread einen **synchronized**-Block betreten hat, der durch ein Lock-Objekt geschützt wird, sind *alle* von diesem Objekt geschützten **synchronized**-Blöcke für andere Threads gesperrt, bis der erste Thread das Lock-Objekt frei gibt.
- Ein Lock-Objekt kann gleichzeitig synchronisierte Instanzmethoden und synchronisierte Blöcken überwachen.

Wichtige Methoden der Klasse **java.lang.Thread**:

Methode	Beschreibung
currrentThread()	liefert eine Referenz auf das aktuelle Thread-Objekt
getName()	liefert den Namen des Threads
getPriority()	ermittelt die Priorität des Threads
interrupt()	Ein per **wait()** deaktivierter oder bei **sleep()** ruhig gestellter Thread wird durch eine **InterruptedException** aufgeweckt. Für andere Threads wird das Interrupt-Signal auf **true** gesetzt.
isAlive()	testet, ob der Thread lebt, d.h. gestartet wurde und noch nicht beendet ist
run()	enthält den vom Thread auszuführenden Code
setPriority()	setzt die Priorität des Threads
sleep()	lässt den Thread eine bestimmte Zeit schlafen
start()	initiiert den Thread und ruft seine **run()**-Methode auf
yield()	gibt Rechenzeit an Threads gleicher Priorität ab (nur relevant, wenn die JVM kein Zeitscheiben-Scheduling unterstützt)

Wichtige Methoden der Klasse **java.lang.Object**:

Methode	Beschreibung
notify()	Von den auf einen Monitor wartenden Threads wird derjenige mit der längsten Wartezeit in den Zustand **ready** versetzt.
notifyAll()	Alle auf einen Monitor wartenden Threads werden in den Zustand **ready** versetzt.
wait()	Der aktuelle Thread wird in den Wartezustand versetzt.

14.9 Übungsaufgaben zu Abschnitt 14

1) Das folgende Programm startet einen Thread, lässt ihn 10 Sekunden lang gewähren und versucht dann, den Thread wieder zu beenden:

```
class Prog {
  public static void main(String[] args) throws InterruptedException {
    Schnarcher st = new Schnarcher();
    st.start();
    System.out.println("Thread gestarted.");
    Thread.sleep(10000);
    while(st.isAlive()) {
      st.interrupt();
      System.out.println("\nThread beendet!?");
      Thread.sleep(1000);
    }
  }
}
```

Außerdem wird demonstriert, dass man auch den Thread **main** per **sleep()** in den vorübergehenden Ruhezustand schicken kann. Um die **try-catch**-Konstruktion zu vermeiden, wird die von **sleep()** potentiell zu erwartende **InterruptedException** in **main()**-Methodenkopf per **throws**-Klausel deklariert.

Der Thread prüft in seiner **run()**-Methode zunächst, ob das Interrupt-Signal gesetzt ist, und beendet sich ggf. per **return**. Falls keine Einwände gegen seine weitere Tätigkeit bestehen, schreibt der Thread nach einer kurzen Wartezeit ein Sternchen auf die Konsole:

```
class Schnarcher extends Thread {
  public void run() {
    while (true) {
      if(isInterrupted())
        return;
      try {
        sleep(100);
      } catch(InterruptedException ie) {}
      System.out.print("*");
    }
  }
}
```

Wie die Ausgabe eines Programmlaufs zeigt, ignoriert der Thread das erste Interrupt-Signal, reagiert aber auf das zweite:

```
Thread gestarted.
*********************************************************************************
********************
Thread beendet!?
*********
Thread beendet!?
*
```

Verändert man die 100 ms lange Thread-interne Kunstpause 101 oder 99 ms, dann bleiben sogar sämtliche Interrupt-Signale wirkungslos:

```
Thread gestarted.
***********************************************************************
Thread beendet!?
*******
Thread beendet!?
*******
Thread beendet!?
*******
Thread beendet!?
*******
    .    .    .
```

Wie ist das Verhalten zu erklären, und wie sorgt man für eine zuverlässiges Beenden des Threads?

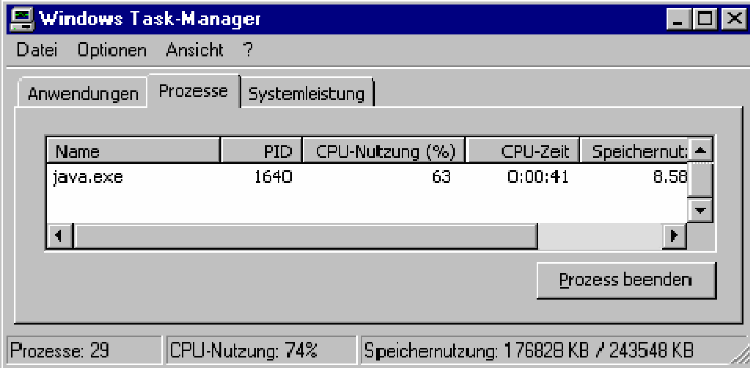
2) Erstellen sie eine Anwendung oder ein Applet, um den Effekt der Prioritäten auf das Verhalten zweier Threads beobachten zu können:

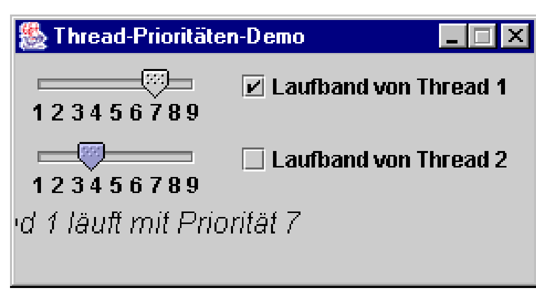
- Der Benutzer soll die Möglichkeit haben, die Prioritäten der beiden Threads zu verändern.
- Die Ausführungsgeschwindigkeiten der Threads sollen als „Rennen“ zu beobachten sein.
- Vielleicht bauen Sie noch Bedienelement ein, um die Threads einzeln unterbrechen und re-aktivieren zu können (per **wait()** und **notify()**).

Bei diesem Lösungsvorschlag handelt es sich um eine Anwendung mit Swing-GUI:

Beide Threads zeichnen eine Laufschrift im unteren Bereich eines **JFrame**-Fensters. Zur Einstellung der Prioritäten sind Schieberegler vorhanden, realisiert mit **JSlider**-Komponenten (siehe API-Dokumentation).

Unter Windows NT/2000/XP kann man mit dem **Task Manager** (**<Strg>+<Alt>+<Entf>**, **<T>**) beobachten, wie sich die CPU-Nutzung der Virtuellen Maschine in Abhängigkeit von den beiden Thread-Aktivitäten bzw. -Prioritäten ändert, z.B.





Lässt man mehrere Threads mit hoher Priorität laufen, reagiert die graphische Benutzeroberfläche eventuell sehr zäh.

15 Literatur

Balzert, H. (1999). *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Heidelberg: Spektrum

Deitel, H. M. & Deitel, P. J. (1999). *Java: how to program* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

Echtle, K. & Goedicke, M. (2000). *Lehrbuch der Programmierung mit Java*. Heidelberg: dpunkt.

Eckel, B. (2002). *Thinking in Java* (2nd ed.). New Jersey: Prentice Hall. Online-Dokument: http://www.mindview.net/Books/TIJ/

Erlenkötter, H. (2001). *Java. Programmieren von Anfang an*. Reinbek: Rowohlt.

Fowler, A. (2003). Painting in AWT and Swing. Online-Dokument: http://java.sun.com/products/jfc/tsc/articles/painting/index.html

Gosling, J., Joy, B., Steele, G. L. & Bracha, G. (2000). The Java Language Specification. Online-Dokument: http://web2.java.sun.com/docs/books/jls/

Kröckertskothen, T. (2001). *Java 2. Grundlagen und Einführung*. Hannover: RRZN.

Krüger, G. (2002). *Goto Java* (2. Auflage). Online-Dokument: http://www.javabuch.de/download.html

Middendorf, S. & Singer, R. (1999). *Java. Programmierhandbuch und Referenz für die Java-2-Plattform*. Heidelberg: dpunkt.

Mössenböck, H. (2001). *Sprechen Sie Java. Eine Einführung in das systematische Programmieren*. Heidelberg: dpunkt.

Münz, S. (2001). SELFHTML 8.0. Online-Dokument: http://selfaktuell.teamone.de/extras/download.shtml

Goll, J., Weiß, C. & Rothländer, P. (2000). *Java als erste Programmiersprache*. Stuttgart: Teubner.

RRZN (1999). *Grundlagen der Programmierung mit Beispielen in C++ und Java*. Hannover.

SUN Inc. (2003). *The Java Tutorial*. Online-Dokument: http://java.sun.com/docs/books/tutorial/

Ullenboom, C. (2003). Java ist auch eine Insel (2. Aufl.). Galileo Computing. Online-Dokument: http://www.galileocomputing.de/katalog/openbook/gp/GPP-opji/

16 Anhang

16.1 Operatorentabelle

In der folgenden Tabelle sind alle im Kurs behandelten Operatoren in absteigender Priorität (von oben nach unten) mit Auswertungsrichtung aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch fette horizontale Linien begrenzt. Sie verfügen dann stets auch über dieselbe Auswertungsrichtung.

Operator	**Bedeutung**	**Richtung**
[] () .	Feldindex Methodenaufruf Komponentenzugriff	L → R
! ++, -- - (*type*) new	Negation Prä- oder Postinkrement bzw. -dekrement Vorzeichenumkehr Typumwandlung Objekterzeugung	L ← R
*, / %	Punktrechnung Modulo	L → R
+, - +	Strichrechnung Stringverkettung	L → R
<<, >>	Links- bzw. Rechts-Shift	L → R
>, <, >=, <= instanceof	Vergleichsoperatoren Typprüfung	L → R
==, !=	Gleichheit, Ungleichheit	L → R
& &	Bitweises UND Logisches UND (mit unbedingter Auswertung)	L → R

Operator	Bedeutung	Richtung
^	Exklusives logisches ODER	L → R
| |	Bitweises ODER Logisches ODER (mit unbedingter Auswertung)	L → R
&&	Logisches UND (mit bedingter Auswertung)	L → R
||	Logisches ODER (mit bedingter Auswertung)	L → R
? :	Konditionaloperator	L ← R
= +=, -=, *=, /=, %=	Wertzuweisung Wertzuweisung mit Aktualisierung	L ← R

16.2 Lösungsvorschläge zu den Übungsaufgaben

Abschnitt 2 (Werkzeuge zum Entwickeln von Java-Programmen)

Aufgabe 2

Syntaxfehler:

- Die schließende Klammer zum Rumpf der Klassendefinition fehlt.
- Die Zeichenfolge im **println**()-Aufruf muss mit dem "-Zeichen abgeschlossen werden.

Semantikfehler:

- Der Methodenname „mein" ist falsch geschrieben.
- Die Methode **main()** muss als **public** definiert werden.

Aufgabe 3

- falsch
- richtig
- richtig
- falsch

Abschnitt 3 (Elementare Sprachelemente)

Abschnitt 3.1 (Einstieg)

Aufgabe 1

Der 1. Aufruf klappt. Der *Name* des **main()**-Parameters ist beliebig.
Der 2. Aufruf scheitert: **static** vergessen.
Der 3. Aufruf scheitert: falscher Rückgabetyp **int**.
Der 4. Aufruf klappt: Der Offsetoperator ist kommutativ.

Aufgabe 2

Unzulässig sind:

- 4you
  Bezeichner müssen mit einem Buchstaben beginnen.
- else
  Schlüsselwörter wie **else** sind als Bezeichner verboten.

Aufgabe 3

Der erste Plus-Operator im **println**()-Parameter konvertiert sein rechtes Argument in eine Zeichenfolge, weil sein linkes Argument eine Zeichenfolge ist. Durch Klammerung muss dafür gesorgt werden, dass der zweite Plus-Operator zuerst ausgeführt wird und daher seine beiden Argumente addiert, bevor sein Ergebnis zum Argument für den ersten Plus-Operator wird.

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
   public static void main(String[] args) {
      System.out.println("3.3 + 2 = " + (3.3 + 2));
   }
}</pre></td><td><pre>3.3 + 2 = 5.3</pre></td></tr>
</table>

Abschnitt 3.2 (Variablen und primitive Datentypen)

Aufgabe 1

char gehört zu den integralen (ganzzahligen) Datentypen. Zeichen werden über ihre Nummer im aktuellen Zeichensatz gespeichert, das Zeichen *c* offenbar durch die Zahl 99 (im Dezimalsystem).

In der folgenden Anweisung wird der **char**-Variablen `z` die Unicode-Escapesequenz für das Zeichen *c* zugewiesen:

```
char z = '\u0063';
```

Der dezimalen Zahl 99 entspricht die hexadezimale Zahl 0x63 (= 6 · 16 + 3).

Aufgabe 2

Durch das Suffix **`l`** im Literal `7l` wird ein **long**-Literal gekennzeichnet.

Aufgabe 3

Die Variable `i` ist nur im innersten Block gültig.

Aufgabe 4

```
class Prog {
   public static void main(String[] args) {
      System.out.println("Dies ist ein Java-Zeichenkettenliteral:\n   \"Hallo\"");
   }
}
```

Aufgabe 5

```
class Prog {
  public static void main(String[] args) {
    float PI = 3.141593f;
    double radius = 2.0;
    System.out.println("Der Flaecheninhalt betraegt: " + PI * radius * radius);
  }
}
```

Abschnitt 3.3 (Einfache Eingabe bei Konsolenanwendungen)

`gint()` liefert bei einer ungültigen Eingabe den Wert 0, wobei aber eine Warnung auf dem Bildschirm erscheint und außerdem die Klassenvariable[1] `Simput.status` auf **false** gesetzt wird, z.B.:

Quellcode	Ausgabe

`class Prog {`
`  public static void main(String[] args) {`
`    System.out.print("Eine ganze Zahl bitte: ");`
`    int i = Simput.gint();`
`    System.out.println("i = " + i +`
`      ", Simput.status = " + Simput.status);`
`  }`
`}`
	`Eine ganze Zahl bitte: haha`
	`Falsche Eingabe!`
	
	`i = 0, Simput.status = false`

In folgendem Programm werden `gchar()` und `gdouble()` verwendet:

```
class Prog {
  public static void main(String[] args) {
    System.out.print("Setzen Sie bitte ein Zeichen: ");
    char c = Simput.gchar();
    System.out.println("c = "+ c);
    System.out.print("\nNun bitte eine gebrochene Zahl (mit Dezimalpunkt!): ");

    double d = Simput.gdouble();
    System.out.println("d = "+ d);
  }
}
```

Abschnitt 3.4 (Operatoren und Ausdrücke)

Aufgabe 1

Ausdruck	Typ	Wert	Anmerkung
`6/4*2.0`	**double**	2.0	
`(int)6/4.0*3`	**double**	4.5	Der Typumwandlungsoperator hat die höchste Priorität und wirkt daher auf die 6.

`3*5+8/3%4*5`	**int**	25	Abarbeitung mit Zwischenergebnissen:
			`15 + 8/3%4*5`
			`15 + 2%4*5`
			`15 + 2*5`
			`15 + 10`

[1] Dieser Begriff wird später ausführlich behandelt.

Aufgabe 2

`erg1` erhält den Wert 0, denn:

- `(i++ == j ? 7 : 8)` hat den Wert 8, weil $2 \neq 3$ ist.
- `8 % 2` ergibt 0.

Auch `erg2` erhält den Wert 0, denn der Präinkrementoperator trifft auf die bereits vom Postinkrementoperator in der vorangehenden Zeile auf den Wert 3 erhöhte Variable `i` und setzt diese auf den Wert 4, so dass die Bedingung im Konditionaloperator erneut den Wert **false** hat.

Aufgabe 3

- `la1` = **false**
  ^ wird zuerst ausgewertet.
- `la2` = **true**
  Klammern erzwingen die alternative Auswertungsreihenfolge.
- `la3` = **false**
  012 ist oktal und hat den dezimalen Wert 10.
- `la4` = **false**

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\Exp

Aufgabe 5

Diese Scherzfrage hat Ihnen wohl kein Kopfzerbrechen bereitet. Die betragsmäßig kleinste ganze Zahl 0 kann natürlich problemlos gespeichert werden.

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2Euro

Abschnitt 3.5 (Anweisungen)

Aufgabe 1

Weil die **else**-Klausel der *zweiten* (nach oben nächstgelegenen) **if**-Anweisung zugeordnet wird, ergibt sich folgender Gewinnplan:

`losNr`	Gewinn
durch 13 teilbar	Nichts
nicht durch 13, aber durch 7 teilbar	1 €
weder durch 13, noch durch 7 teilbar	100 €

Aufgabe 2

Im logischen Ausdruck der **if**-Anweisung findet an Stelle eines Vergleichs eine *Zuweisung* statt.

Aufgabe 3

In der **switch**-Anweisung wird es versäumt, per **break** den Durchfall zu verhindern.

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\PrimitivOB

Aufgabe 5

Das Semikolon am Ende der Zeile

```
while (i < 100);
```

muss vom Compiler als die zur **while**-Schleife gehörige (leere) Anweisung interpretiert werden, so dass eine *Endlosschleife* vorliegt.
Die restlichen Zeilen werden als selbständiger Anweisungsblock gedeutet, aber nie ausgeführt. Obwohl ein Programm mit dieser Konstruktion vollkommen nutzlos ist, verbraucht es reichlich CPU-Zeit und bringt den Prozessor zum Glühen.

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2EuroS

Aufgabe 7

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\FloP

Bei $i = 52$ rechnet Java letztmals richtig.

Aufgabe 8

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

...\BspUeb\Elementare Sprachelemente
...\BspUeb\Elementare Sprachelemente

Abschnitt 4 (Klassen und Objekte)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\R2Vec

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\Life\a (Nummerierung per Konstruktor-Parameter)

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\Life\b (Nummerierung über eine statische Variable)

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\Life\FakulRek

Abschnitt 5 (Elementare Klassen)

Abschnitt 5.1 (Arrays)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Lotto

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Eratosthenes

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Marry

Abschnitt 5.2(Zeichenketten)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\PerZuf

Aufgabe 2

Die Klasse **StringBuffer** hat die von **java.lang.Object** geerbte **equals()**-Methode *nicht* überschrieben, so dass *Referenzen* verglichen werden. In der Klasse String ist **equals()** jedoch so überschrieben worden, dass die referenzierten Zeichenfolgen verglichen werden.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\StringUtil

Abschnitt 5.3 (Verpackungs-Klassen für primitive Datentypen)

Aufgabe 1

Lösungsvorschlag:

<table>
<tr><th>Quellcode</th><th>Ausgabe</th></tr>
<tr><td><pre>class Prog {
    public static void main(String[] args) {
        System.out.println("Min. byte-Zahl: \n " + Byte.MIN_VALUE);
    }
}</pre></td><td><pre>Min. byte-Zahl:
 -128</pre></td></tr>
</table>

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\MaxFakul

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\Mint

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Klassen\StringUtil

Abschnitt 6 (Pakete)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\DemoPack

Aufgabe 2

Lösungsvorschlag: Die beteiligten Klassen befanden sich stets im selben Verzeichnis und waren keinem Paket explizit zugeordnet, so dass sie sich in einem gemeinsamen (anonymen) Paket befanden. Klassen eines Paketes haben per Voreinstellung wechselseitig volle Rechte.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Pakete\JAR

Zum Einpacken eignet sich das Kommando:

```
jar cmf0 Manifest.txt DM2Euro.jar DM2Euro.class Simput.class
```

Mit Hilfe der fertigen Archivdatei lässt sich die Klasse `DM2Euro` folgendermaßen starten:

```
java -jar DM2Euro.jar
```

Abschnitt 7 (Vererbung und Polymorphie)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Vererbung und Polymorphie\Figuren

Aufgabe 2

In der `Vater`-Klasse fehlt ein parameterfreier Konstruktor.

Aufgabe 3

In der Klasse Figur haben `xpos` und `ypos` den voreingestellten Zugriffsschutz (**package**). Weil `Kreis` nicht zum selben Paket gehört, hat diese Klasse keinen direkten Zugriff. Soll dieser Zugriff möglich sein, müssen `xpos` und `ypos` in der `Figur`–Definition die Schutzstufe **protected** (oder **public**) erhalten.

Abschnitt 8 (Ausnahme-Behandlung)

Aufgabe 1

Lösungsvorschlag:

```
class Except {
  public static void main(String[] args) {
    int[] vek = new int[5];
    vek = null;
    vek[0] = 1;
  }
}
```

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Exceptions\DuaLog

Abschnitt 9 (Interfaces)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Interface\Klonen

Aufgabe 2

Über die `Leer`-Referenzvariable stehen nur Methoden zur Verfügung, die in der `Leer`-Schnittstellendeklaration enthalten sind, also gar keine. Um über eine `Leer`-Referenz die `ImplDemo`-Methode `sagWas()` nutzen zu können, ist eine explizite Konversion erforderlich:

```
class Test {
  public static void main(String[] args) {
    Leer tobj = new ImplDemo();
    System.out.println(((ImplDemo)tobj).sagWas());
  }
}
```

Abschnitt 10 (Ein-/Ausgabe über Datenströme)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\IO\BufferedOutputStream\auf die Konsole

Aufgabe 2

Beim Ausführen der Methode **readShort()** fordert das **DataInputStream**–Objekt den angekoppelten **InputStream** auf, zwei Bytes zu beschaffen. Schickt der Benutzer z.B. eine eingetippte „0" per **<Enter>**-Taste ab, dann gelangen folgende Bytes in den **InputStream**:

- 00110000 (0x30, 8-Bit-Code des Zeichens „0")
- 00001101 (0x0D, 8-Bit-Code für Wagenrücklauf)

Anschließend entnimmt **readShort()** dem **InputStream** die beiden Bytes und interpretiert sie als vorzeichenbehaftete 16-Bit-Ganzzahl, was im Beispiel den dezimalen Wert 12301 ergibt.
Während der Filter korrekt arbeitet, ist die Eingabe passender Bytes via Tastatur (also **System.in**) offenbar äußerst umständlich bis unmöglich. Offenbar ist die Transformationsklasse **DataInputStream** nicht dafür eignet (und auch nicht dafür gedacht), **short**-Werte über das **InputStream**–Objekt **System.in** einzulesen. Sie wurde dafür konstruiert, die über einem **DataOutputStream** geschriebenen Bytes in Werte des ursprünglichen Datentyps zurück zu verwandeln.
Von der Tastatur sollte man über ein Gespann aus **BufferedReader** (bietet bequeme Methoden wie **readLine()**) und **InputStreamReader** (wandelt Bytes in Zeichen) *Zeichen* entgegen nehmen. Abschließend kann man versuchen, mit Hilfe der Konvertierungsmethoden der Wrapper-Klassen aus den Zeichen primitive Datenwerte zu gewinnen.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\IO\DataOutputStream\Mit Puffer

Aufgabe 4

Im **PrintWriter**-Konstruktor ist die **autoFlush**-Option eingeschaltet, was bei einer Dateiausgabe in der Regel keinen Nutzen bringt. So hat jeder **println()**-Aufruf einen zeitaufwändigen Dateizugriff zur Folge, und die vom **PrintWriter** automatisch vorgenommene Pufferung wird außer Kraft gesetzt. Für das Programm ist folgender Konstruktor besser geeignet:

```
PrintWriter(fos)
```

Aufgabe 5

Bei einem **PrintStream** wird u.a. durch das Newline-Zeichen ein AutoFlush ausgelöst, beim **PrintWriter** geschieht dies jedoch nur beim Aufruf der **println()**-Methode, welche die plattformspezifische Zeilenschaltung benutzt.

Abschnitt 11 (GUI-Programmierung mit Swing/JFC)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\EreignisID

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\KeyListener

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\AnonClass

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\EuroKonverter

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Swing\Editor\Mit Sichern

Abschnitt 12 (Applets)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Applets\PrimApplet

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Applets\EuroKonverter

Abschnitt 13 (Multimedia)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multimedia\Grafik\Wuerfel

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multimedia\Sound\Hintergrundmusik

Abschnitt 14 (Threads)

Aufgabe 1

a) Es gehen Interrupt-Signale verloren, weil sie dem Thread in der Schlafphase zugestellt werden, wobei eine **InterruptedException** ausgelöst wird, die das Interrupt-Signal wieder auf **false** setzt. Im Beispiel geht das Interrupt-Signal offenbar genau dann *nicht* verloren, wenn die **sleep()**-Zeit im `Schnarcher` ein Teiler der **sleep()**-Zeit in der **while**-Schleife des **main**-Threads ist. Dies war jedoch nur auf einem bestimmten Rechner zu beobachten (CPU: K6-3 mit 400 MHz)! Ein schlecht programmierter Thread-Einsatz kann also für sehr verwirrende Fehlermuster sorgen.

b) Man erreicht ein zuverlässiges Verhalten des `Schnarcher`-Threads, indem man im **InterruptedException**-Handler entweder erneut **interrupt()** aufruft, um das verlorenen Interrupt-Signal zu ersetzen, oder per **return** die **run()**-Methode und damit den Thread beendet.

Aufgabe 2

Den Quellcode zu dem in der Aufgabenstellung vorgeführten Swing-Programm finden Sie im Verzeichnis:

...\BspUeb\Threads\Prior