Dietmar Abts

Grundkurs JAVA

Java 2 Plattform

Eine Einführung in das objektorientierte Programmieren mit Beispielen und Übungsaufgaben

Mit Online-Service zum Buch

2. Auflage

Ausbildung und Studium

Dietmar Abts

Grundkurs JAVA

Ausbildung und Studium

Die Bücher der Reihe „Ausbildung und Studium" bieten praxisorientierte Einführungen für die Aus- und Weiterbildung sowie Bausteine für ein erfolgreiches berufsqualifizierendes Studium.

Unter anderem sind erschienen:

Turbo Pascal Wegweiser für Ausbildung und Studium
von Ekkehard Kaier

Delphi Essentials
von Ekkehard Kaier

Programmieren mit Fortran 90
von Hans-Peter Bäumer

Wirtschaftsmathematik mit dem Computer
von Hans Benker

Datenbank-Engineering
von Alfred Moos und Gerhard Daues

Visual Basic Essentials
von Ekkehard Kaier

Excel für Betriebswirte
von Robert Horvat
und Kambiz Koochaki

Grundkurs Wirtschaftsinformatik
von Dietmar Abts und Wilhelm Mülder

Praktische Systemprogrammierung
von Helmut Weber

Ingenieurmathematik mit Computeralgebra-Systemen
von Hans Benker

Excel für Techniker und Ingenieure
von Hans-Jürgen Holland
und Uwe Bernhardt

Relationales und objektrelationales SQL
von Wolf-Michael Kähler

Kostenstellenrechnung mit SAP® R/3®
von Franz Klenger und Ellen Falk Kalms

Theorie und Praxis relationaler Datenbanken
von René Steiner

OBERON
von B. Marincek, J.L. Marais
und E. Zeller

Studienführer Wirtschaftsinformatik
von Peter Mertens, Peter Chamoni,
Dieter Ehrenberg, Joachim Griese,
Lutz J. Heinrich und Karl Kurbel (Hrsg.)

Einführung in UNIX
von Werner Brecht

Objektorientierte Programmierung in JAVA
von Otto Rauh

PL/I für Workstations
von Eberhard Sturm

Effektiv Programmieren in C und C++
von Dietmar Herrmann

Module, Klassen, Verträge
von Karlheinz Hug

Aufbaukurs Wirtschaftsinformatik
von Dietmar Abts und Wilhelm Mülder

Online-Publishing für Studenten und Wissenschaftler
von Michael Beißwenger

Visual Basic für technische Anwendungen
von Jürgen Radel

Grundkurs Algorithmen und Datenstrukturen
von Andreas Solymosi
und Ulrich Grude

Grundkurs JAVA
von Dietmar Abts

Vieweg

Dietmar Abts

Grundkurs JAVA

Eine Einführung in das objekt-
orientierte Programmieren mit Beispielen
und Übungsaufgaben

vieweg

Die Deutsche Bibliothek – CIP-Einheitsaufnahme
Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

1. Auflage Oktober 2000
ISBN 978-3-528-15711-1 ISBN 978-3-663-10208-3 (eBook)
DOI 10.1007/978-3-663-10208-3
Alle Rechte vorbehalten
© Springer Fachmedien Wiesbaden, 2000
Ursprünglich erschienen bei Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig/Wiesbaden, 2000.

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlags unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

www.vieweg.de

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Höchste inhaltliche und technische Qualität unserer Produkte ist unser Ziel. Bei der Produktion und Auslieferung unserer Bücher wollen wir die Umwelt schonen: Dieses Buch istauf säurefreiem und chlorfrei gebleichtem Papier gedruckt. Die Einschweißfolie besteht aus Polyäthylen und damit aus organischen Grundstoffen, die weder bei der Herstellung noch bei der Verbrennung Schadstoffe freisetzen.

Konzeption und Layout des Umschlags: Ulrike Weigel, www.CorporateDesignGroup.de

Gedruckt auf säurefreiem Papier

Vorwort zur zweiten Auflage

Java ist eine noch junge Programmiersprache, die sich innerhalb kurzer Zeit etabliert hat. Sie unterscheidet sich in vielerlei Hinsicht von anderen Programmiersprachen. Dabei ist nicht allein ihre Objektorientierung das hauptsächliche Unterscheidungsmerkmal. Java wird heute als universelle Programmiersprache für Internet- und Intranet-Anwendungen sowohl auf der Client- als auch auf der Server-Seite eingesetzt. Ein weiteres wesentliches Merkmal von Java ist die Plattformunabhängigkeit. Dank des Konzepts der virtuellen Maschine sind in Java programmierte Anwendungen einschließlich ihrer grafischen Oberflächen ohne Portierung auf nahezu allen Rechnersystemen lauffähig. Allerdings hängt es vom Grad der Java-Unterstützung des eingesetzten Web-Browsers ab, ob Applets, die mit der neuesten Java-Version entwickelt wurden, ohne Umstände lauffähig sind. Java hat von den Erfahrungen mit anderen Programmiersprachen wie Smalltalk, C und C++ profitiert. Wesentliche Konzepte wurden übernommen. Auf allzu komplexe und fehleranfällige Eigenschaften wurde bewusst verzichtet, um die Sprache einfach und robust halten zu können.

In der vorliegenden Auflage werden zur Entwicklung grafischer Oberflächen Swing-Komponenten auf der Basis von Java 2 eingesetzt. Sämtliche Beispielprogramme sind von AWT auf Swing umgestellt worden.

Zielsetzung des Buches

Dieser Grundkurs bietet eine strukturierte und anschauliche Einführung in grundlegende Aspekte der Java-Programmierung. Dabei werden kennzeichnende Eigenschaften der Objektorientierung, wie Klassendefinition, Vererbung und Polymorphismus, ausführlich dargestellt. Das Buch kann nicht die gesamte Java-Klassenbibliothek, die sehr umfangreich ist, vorstellen. Nur die für das Grundverständnis wichtigen Klassen und Methoden werden diskutiert. Eine vollständige Beschreibung aller Klassen findet man in einschlägigen Referenzhandbüchern und in der Online-Dokumentation zum Java 2 Software Development Kit (siehe Literatur- und Quellenverzeichnis). Obwohl dieser Kurs keine Programmiererfahrung voraussetzt, erleichtern

Kenntnisse in einer Programmiersprache wie z.B. Pascal, C oder C++ den Einstieg.

Inhalt und Gliederung

Die einzelnen Kapitel bauen aufeinander auf.

- *Kapitel 1* gibt eine Übersicht über die Konzeption von Java. Hier wird bereits ein erstes Beispielprogramm vorgestellt.
- *Kapitel 2* beschäftigt sich mit den nicht objektorientierten Grundlagen der Sprache, wie z.B. Datentypen, Operatoren und Kontrollstrukturen.
- *Kapitel 3* führt die objektorientierten Konzepte ein, deren Verständnis grundlegend für alles Weitere ist.
- *Kapitel 4* widmet sich der Behandlung von Fehlern, die während der Programmausführung auftreten können.
- *Kapitel 5* stellt einige nützliche Klassen mit ihren Methoden vor.
- *Kapitel 6* enthält die für den Zugriff auf Dateien wichtigen Klassen und Methoden.
- *Kapitel 7* bietet eine Einführung in die Programmierung mehrerer gleichzeitig laufender Anweisungsfolgen (Threads) innerhalb eines Programms.
- *Kapitel 8* befasst sich mit der Entwicklung von grafischen Oberflächen sowie der Behandlung von Ereignissen, wie z.B. die Auswahl eines Menüpunkts oder das Drücken eines Buttons.
- *Kapitel 9* behandelt Applets, die in Web-Seiten eingebunden sind und unter der Kontrolle des Web-Browsers laufen. Hier werden auch Techniken zur Entwicklung von Animationen vorgestellt.
- *Kapitel 10* enthält einen Ausblick auf weitere Anwendungsmöglichkeiten: Zugriff auf eine relationale Datenbank mit Hilfe von SQL und die Programmierung von Client-Server-Anwendungen auf der Basis von TCP/IP und der Socket-Schnittstelle. Abschließend wird eine Datenbankanwendung entwickelt, die über ein HTML-Formular im Browser bedient werden kann.

Programm- und Aufgabensammlung

Zahlreiche Beispielprogramme helfen bei der Umsetzung der Konzepte in lauffähige Anwendungen. Jedes Kapitel (mit Ausnahme von Kapitel 1) enthält am Ende Aufgaben, die den behandelten Stoff einüben und vertiefen. Alle Programme wurden mit dem von Sun bereitgestellten Software Development Kit Java 2 SDK v 1.3 unter Windows 98 getestet. Der Quellcode zu den Beispielprogrammen und zu den Lösungen der Aufgaben steht – zusammengefasst in einer ZIP-Archivdatei – im Internet zum Herunterladen zur Verfügung. Die Web-Adresse ist dem Literatur- und Quellenverzeichnis zu entnehmen. Das Archiv enthält zudem eine HTML-Startseite, die die Navigation durch die Programmsammlung erleichtert.

Kommentare und Verbesserungsvorschläge richten Sie bitte als E-Mail an den Autor: `abts@fh-niederrhein.de`

Ratingen, im Oktober 2000 Dietmar Abts

Inhaltsverzeichnis

1 Entwicklung und Konzeption von Java 1

2 Grundlagen 11
- 2.1 Kommentare und Bezeichner 11
- 2.2 Variablen und Datentypen 13
- 2.3 Ausdrücke und Operatoren 17
- 2.4 Kontrollstrukturen 25
- 2.5 Aufgaben 33

3 Objektorientierte Programmierung 35
- 3.1 Klassen und Objekte 35
- 3.2 Methoden 39
- 3.3 Konstruktoren 42
- 3.4 Statische Attribute und Methoden 44
- 3.5 Vererbung 46
- 3.6 Abstrakte Klassen 50
- 3.7 Modifizierer von Klassen, Attributen und Methoden 54
- 3.8 Interfaces 56
- 3.9 Innere Klassen 59
- 3.10 Pakete 62
- 3.11 Aufgaben 65

4 Ausnahmebehandlung 67
- 4.1 Ausnahmeklassen 67
- 4.2 Auslösung und Weitergabe von Ausnahmen 69
- 4.3 Abfangen von Ausnahmen 70
- 4.4 Aufgaben 74

5 Ausgewählte Klassen 77
- 5.1 Die Klassen String und StringBuffer 77
- 5.2 Arrays 83
- 5.3 Die Klasse StringTokenizer 90
- 5.4 Hüllklassen 92
- 5.5 Mathematische Funktionen 97
- 5.6 Listen 101
- 5.7 Hashtabellen 109

5.8 Die Klasse System ... 115
5.9 Klassendeskriptoren ... 117
5.10 Zeitangaben und -berechnungen ... 120
5.11 Aufgaben ... 125

6 Dateiverarbeitung ... 129
6.1 Übersicht über Datenströme ... 130
6.2 Die Klasse File ... 136
6.3 Sequentielle Dateiverarbeitung ... 140
6.4 Random-Access-Dateien ... 152
6.5 Aufgaben ... 156

7 Threads ... 159
7.1 Das Erzeugen von Threads ... 160
7.2 Synchronisation von Threads ... 166
7.3 Das Unterbrechen von Threads ... 175
7.4 Aufgaben ... 177

8 Grafische Benutzungsoberflächen ... 179
8.1 Übersicht ... 180
8.2 Fenster und Grafik ... 182
8.3 GUI-Komponenten und Ereignisbehandlung ... 191
8.3.1 Das Event-Modell ... 191
8.3.2 Buttons ... 198
8.3.3 Labels ... 204
8.3.4 Scrollbalken und spezielle Container ... 206
8.3.5 Textkomponenten ... 214
8.3.6 Auswahlkomponenten ... 222
8.4 Layout-Manager ... 227
8.5 Menüs und Symbolleisten ... 234
8.6 Dialogfenster ... 240
8.7 Tabellen ... 250
8.8 Mausaktionen ... 255
8.9 Aufgaben ... 260

9 Applets ... 263
9.1 Der Lebenszyklus eines Applets ... 263
9.2 Die Appletumgebung ... 267
9.3 Hybridanwendungen ... 275
9.4 Wiedergabe von Bild- und Tondaten ... 277
9.5 Animation ... 281

9.6 Aufgaben 283

10 Datenbank- und Netzanwendungen 285
10.1 Datenbankzugriffe mit JDBC 285
10.2 Netzwerkprogrammierung 301
10.3 Datenbankabfragen über HTTP 310
10.4 Aufgaben 324

Literatur- und Quellenverzeichnis 327

Sachwortverzeichnis 329

1 Entwicklung und Konzeption von Java

Die Anfänge

Die Entstehungsgeschichte der Programmiersprache *Java* begann im Jahre 1991. Eine Gruppe um Patrick Naughton und James Gosling bei *Sun Microsystems* arbeitete an einem System zur Entwicklung von plattformunabhängiger Software für Geräte der Konsumelektronik. Teil des Systems war der Interpreter *Oak*. Die Vermarktung scheiterte jedoch am geringen Interesse der Anwender. Erst der Durchbruch des *World Wide Web* (WWW) eröffnete dem Entwicklerteam eine neue Zielrichtung. Oak wurde in Java umbenannt.

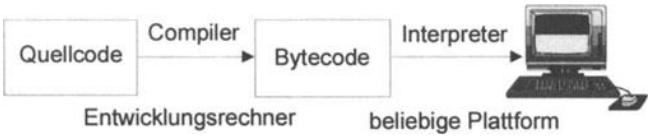
Im Frühjahr 1995 wurde unter dem Namen *HotJava* ein in der Sprache Java geschriebener Web-Browser vorgestellt, der in der Lage war, kleine Java-Programme (so genannte *Applets*) aus dem Internet zu laden und innerhalb des Browsers auszuführen. 1996 wurde die erste Version der Entwicklungsumgebung *Java Development Kit* (JDK) freigegeben. Seitdem steht die Entwicklung von Java nicht still. Die Sprache wurde um zahlreiche Schnittstellen erweitert und die Entwicklungsumgebung funktional ausgebaut.

Das Konzept

Plattformunabhängigkeit und *Internet-Fähigkeit* sind die beiden wichtigsten Eigenschaften von Java, zusammengefasst in dem Slogan: *Write once – run anywhere*. Das vom *Java-Compiler* aus dem Quellcode erzeugte Programm, der so genannte *Bytecode*, ist unabhängig von der Rechnerarchitektur und läuft auf jedem Rechner, auf dem eine spezielle Software, die *Java Virtual Machine* (JVM), existiert. Diese JVM ist heute für jedes gängige Betriebssystem verfügbar. Die virtuelle Maschine JVM stellt eine Schicht zwischen dem Bytecode und der zu Grunde liegenden Rechnerplattform dar. Ein Interpreter übersetzt die Bytecode-Befehle in plattformspezifische Prozessorbefehle. Die JVM kann die Ausführung der Java-Programme überwachen und verhindern, dass Befehle

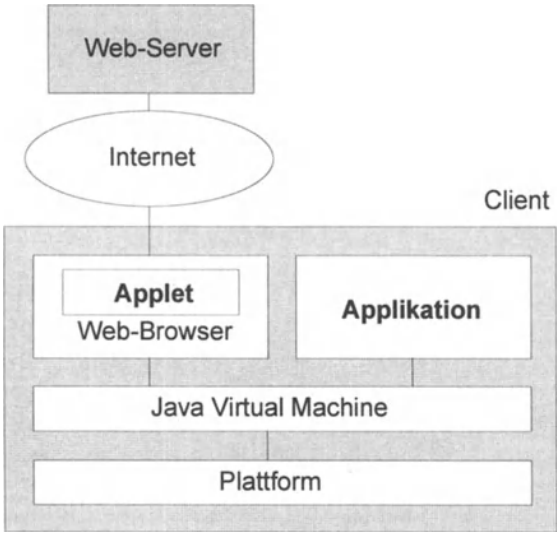
ausgeführt werden, die die Sicherheit des Systems gefährden. Zur Leistungssteigerung von wiederholt auszuführenden Programmteilen kann ein *Java-Just-In-Time-Compiler* (JIT) eingesetzt werden, der zur Laufzeit den Bytecode vor der Programmausführung in Prozessorbefehle übersetzt.

Bild 1.1: Übersetzung und Ausführung



Ein Java-Programm kann als eigenständig lauffähiges Programm (*Applikation*) oder als *Applet* erstellt werden.

Bild 1.2: Applikationen und Applets



Eine Applikation kann im Gegensatz zum Applet ohne die Kontrolle eines Browsers (oder eines ähnlichen Programms) ausgeführt werden. Applets werden in HTML-Seiten des WWW referenziert, durch einen Java-fähigen Browser vom Web-Server über das Netz geladen und unter der Kontrolle des Browsers innerhalb des Web-Dokuments ausgeführt. Applets sind im Unterschied zu Applikationen aus Sicherheitsgründen in den Möglichkeiten des Zugriffs auf Ressourcen des lokalen Rechners eingeschränkt.

Sicherheit

Das Java-Sicherheitsmodell sieht jedes Applet als potentiell gefährlich an. Deshalb laufen Applets in einer eigenen, vom System gesicherten Umgebung (*Sandbox*) ab. Die Sandbox kann vom Anwender konfiguriert werden, indem er Sicherheitsrichtlinien in Form von Regeln für den Zugriff auf lokale Ressourcen festlegt. So können z.B. in vertrauenswürdigen Intranet-Umgebungen Zugriffe auf Dateien der lokalen Festplatte erlaubt werden.

Um die Herkunft eines Applets eindeutig bestimmen und eine Manipulation während der Übertragung im Netz ausschließen zu können, kann ein Applet mit einer digitalen Signatur versehen werden (*signierte Applets*). Stellt das Überprüfungsverfahren fest, dass die Signatur stimmt, so ist das Applet in Ordnung.

Merkmale der Sprache

Java besitzt die folgenden Sprachmerkmale:

- Java ist eine *objektorientierte Sprache*. Sie unterstützt alle zentralen Aspekte der Objektorientierung wie Klassen, Datenkapselung, Vererbung und Polymorphismus. Wesentliche Konzepte wurden von *Smalltalk*, *C* und *C++* übernommen.
- Die Sprache ist bewusst *einfach* gehalten. Im Unterschied zu C++ gibt es in Java z.B. keine expliziten Zeiger, keine Header-Dateien und keine Mehrfachvererbung.
- Die in einem Programm benutzten Klassen können als Dateien *an unterschiedlichen Orten* liegen. Sie werden erst zur Laufzeit des Programms bei Bedarf geladen.

- Das Speichermanagement in Java erfolgt automatisch. Während der Laufzeit eines Programms kann der Speicherplatz für nicht mehr benötigte Objekte vom Laufzeitsystem freigegeben (*Garbage Collection*) werden.
- Zur Zeichendarstellung nutzt Java den internationalen *Unicode*-Standard (16-Bit-Codierung).
- Java ist eine *stark typisierte Sprache*. Bereits bei der Übersetzung in den Bytecode werden Datentypüberprüfungen ausgeführt und Typinkonsistenzen erkannt.
- In Java gibt es eine strukturierte Behandlung (*Exception-Handling*) von Laufzeit-Fehlern, die während der Abarbeitung eines Programms auftreten können. So wird z.B. während der Laufzeit die Einhaltung von Indexgrenzen beim Zugriff auf Arrays überwacht.
- Java unterstützt die zeitlich parallele Ausführung von Anweisungsfolgen (*Threads*) innerhalb von Programmen (*Multithreading*).

Wichtige APIs

Die Java-Klassenbibliotheken stellen eine Reihe nützlicher *APIs* (Application Programming Interfaces) in Form von Klassen und Interfaces für die Anwendungsprogrammierung zur Verfügung. Die folgende Liste enthält eine kurze Beschreibung wichtiger APIs.

- Die *Java Foundation Classes* (JFC) sind eine Sammlung von Klassen zur Implementierung von grafischen Benutzungsoberflächen und umfassen u.a. AWT, Java 2D, Swing-Komponenten und das Accessibility API.
- Das *Abstract Window Toolkit* (AWT) umfasst eine große Anzahl von Klassen zur Erstellung von grafischen Oberflächen auf unterschiedlichen Plattformen.
- Das *Java 2D API* unterstützt die Erzeugung von Grafiken und komplexen geometrischen Objekten.
- Mit *Swing-Komponenten* wird ein plattformunabhängiges Design von grafischen Oberflächen mit auf allen Rechnern einheitlichem *Look and Feel* erreicht. Swing enthält alle AWT-Komponenten und stellt zusätzlich neue Be-

dienungselemente (z.B. Tabellen, Baumansichten) zur Verfügung.

- Das *Accessibility API* ist eine Schnittstelle zur Einbindung verschiedener Ein- und Ausgabetechnologien wie z.B. Spracheingabe, Sprach- und Texterkennung, Textausgabe auf einem Blindensprache-Terminal.
- Das *Java Native Interface* (JNI) ermöglicht den Zugriff auf Systemressourcen der zu Grunde liegenden Plattform über deren Systembibliotheken und Betriebssystemfunktionen. Es können Funktionen aufgerufen werden, die mit anderen Programmiersprachen (z.B. C oder C++) implementiert sind. Der Einsatz von JNI führt damit allerdings zu plattformabhängigem Code.
- Das *Remote Method Invocation* Interface (RMI) ermöglicht den Aufruf von Java-Methoden, die auf einem anderen Rechner vorliegen. Damit können Programme erstellt werden, die über ein Netz miteinander kommunizieren (verteilte Anwendung).
- Auf der Basis von *CORBA* (Common Object Request Broker Architecture) können Java-Programme mit in anderen objektorientierten Sprachen programmierten Anwendungen zusammenarbeiten. Der Standard CORBA stellt die notwendigen Schnittstellen für die Anbindung unterschiedlicher Hard- und Software-Plattformen an ein verteiltes Gesamtsystem bereit. Mit *Java IDL* (Interface Definition Language) wird die Schnittstelle zwischen Java und CORBA plattformunabhängig beschrieben.
- So genannte *Servlets* stellen das serverseitige Pendant zu Applets dar. Servlets sind kleine Java-Programme, die auf dem Web-Server ablaufen. Sie nehmen Anfragen von einem Client entgegen, erzeugen ein Ergebnis und schicken es an den Client zurück.
- *Java Beans* ist die Realisierung einer Komponententechnologie für Java. Java-Programme können aus bereits existierenden Softwarekomponenten (Beans) zusammengesetzt werden. Entwicklungswerkzeuge können

diese Komponenten analysieren und in einem bestimmten Rahmen an die jeweiligen Anforderungen anpassen.

- Eine einheitliche Schnittstelle zwischen Java und relationalen Datenbanken ermöglicht den Datenzugriff aus Java-Programmen heraus. Das *JDBC API* (Java Database Connectivity) besitzt die Klassen und Methoden, um über die Standard-Datenbanksprache *SQL* auf Datenbanken verschiedener Hersteller zugreifen zu können. Erweiterungen, wie z.B. die direkte Einbettung der SQL-Befehle in Java (SQLJ), erleichtern den Datenzugriff.
- Das *Security API* erlaubt die Entwicklung bzw. Einbindung von Verschlüsselungsfunktionen in eine Java-Anwendung. Digitale Signaturen können erzeugt und überprüft werden.
- Das *Java 3D API* bietet eine Schnittstelle zu Grafiksprachen und macht 3D-Animation möglich.
- Das *Java Advanced Imaging API* ermöglicht die Entwicklung von effizienter Bildverarbeitungssoftware für im Netz verteilte Bilddaten.

Die Java-2-Familie

Sun hat Java 2 in vier Bereiche eingeteilt:

- *Java 2 Standard Edition* (J2SE) vorwiegend für Client-Anwendungen auf PCs und Workstations
- *Java 2 Enterprise Edition* (J2EE) mit wichtigen APIs für Unternehmensanwendungen auf der Basis von Client-Server-Architekturen
- *Java 2 Micro Edition* (J2ME) für kleine Geräte wie Handhelds und für so genannte Embedded-Systeme
- *Smartcard Edition* für den Einsatz auf Chipkarten

Entwicklungsumgebungen

Das von Sun Microsystems kostenlos zur Verfügung gestellte *Software Development Kit* (SDK) der Standard Edition enthält die Bibliotheken, API-Beschreibungen und Programme, die man zum Übersetzen, Ausführen und Testen von Java-Programmen braucht. Das SDK bietet allerdings keine komfortable grafische Oberfläche. Die Programme werden auf Kommandozeilenebene aufgerufen.

Von verschiedenen Herstellern werden integrierte Entwicklungsumgebungen zur Unterstützung von komplexen Projekten mit teamfähigen Werkzeugen angeboten, z.B. *Visual Age for Java* von IBM, *JBuilder* von Inprise/Borland und *Visual Café* von Symantec.

Ein erstes Beispiel

Das erste Beispielprogramm, dessen Quellcode in der Datei `ErstesBeispiel.java` abgespeichert ist, enthält die Definition einer Klasse mit dem Namen `ErstesBeispiel`:

```
public class ErstesBeispiel { ... }
```

Im Innern der geschweiften Klammern sind eine Variable, die eine ganze Zahl (`int` steht für Integer) speichern kann und den Namen `nummer` trägt, und eine Reihe von Funktionen, in denen die eigentlichen Programmanweisungen stehen, festgelegt.

Programm 1.1

```
// Datei: ErstesBeispiel.java

public class ErstesBeispiel {
   private int nummer;

   public ErstesBeispiel() {
      nummer = 0;
   }

   public int getNummer() {
      return nummer;
   }

   public void setNummer(int nr) {
      nummer = nr;
   }

   public static void main(String[] args) {
      ErstesBeispiel eb = new ErstesBeispiel();
      System.out.println(eb.getNummer());
      eb.setNummer(4711);
      System.out.println(eb.getNummer());
   }
}
```

Jede Java-Applikation startet mit einer Funktion mit Namen `main`. Im Beispiel erzeugt die erste Anweisung dieser Funktion mit Hilfe des Operators `new` und der Funktion `ErstesBeispiel` ein Objekt dieser Klasse. Diese konkrete Ausprägung der Klasse enthält Speicherplatz für eine ganze Zahl (hier mit `0` vorbelegt) und bietet die Funktionen `getNummer` und `setNummmer` als Leistung an. Mit Hilfe der Variablen `eb` kann auf dieses Objekt zugegriffen werden. Die Funktion `getNummer` gibt den Wert von `nummer` zurück. Die Funktion `setNummer` ermöglicht die Veränderung des Wertes von `nummer`, indem bei ihrem Aufruf als Parameter eine ganze Zahl übergeben wird.

Die zweite Anweisung der `main`-Funktion enthält den Aufruf der Funktion `System.out.println`. Sie gibt den mitgegebenen Parameter auf dem Bildschirm aus: hier den Rückgabewert der für das Objekt aufgerufenen Funktion `getNummer`, also `0`. Anschließend wird der Inhalt der Variablen `nummer` mit dem Wert `4711` überschrieben. Der neue Wert wird ausgegeben.

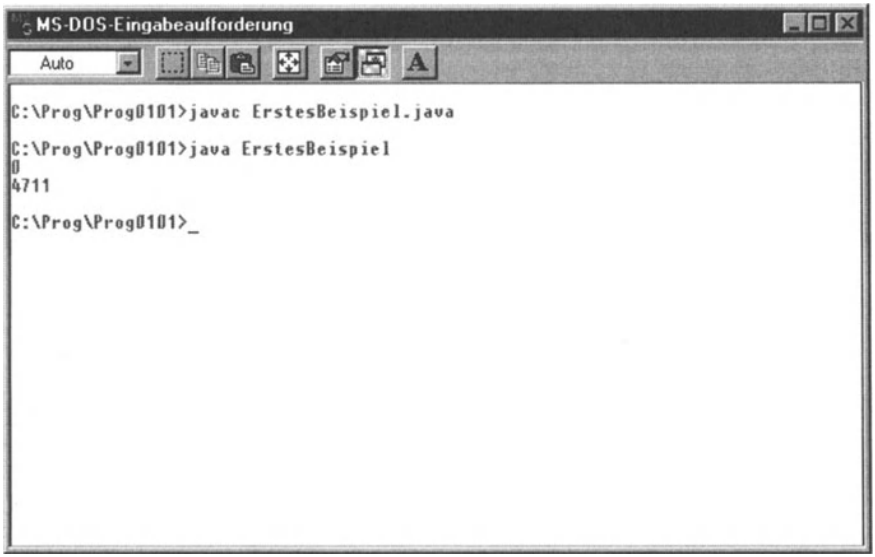
Für die Übersetzung und Ausführung werden die Werkzeuge der Entwicklungsumgebung SDK benutzt. Um das Beispielprogramm zu übersetzen, gibt man in der Kommandozeile (im DOS-Fenster) folgenden Befehl ein:

```
javac ErstesBeispiel.java
```

Nach erfolgreicher Übersetzung des Quellcodes existiert die Datei `ErstesBeispiel.class`, die den Bytecode enthält. Zur Ausführung wird der Java-Interpreter wie folgt aufgerufen:

```
java ErstesBeispiel
```

Bild 1.3:
Übersetzen und
Ausführen



2 Grundlagen

Dieses Kapitel beschreibt die elementaren, nicht objektorientierten Sprachelemente von Java. Insbesondere werden Datentypen, Variablen, Operatoren und Kontrollstrukturen behandelt. Mitunter ist es unerlässlich, objektorientierte Sprachelemente zu benutzen, um lauffähige Programme zu erhalten. Diese Elemente werden aber erst im Kapitel 3 ausführlich behandelt. Die Verständlichkeit des vorliegenden Kapitels wird hierdurch nicht beeinträchtigt.

2.1 Kommentare und Bezeichner

Kommentare

Kommentare im Quellcode werden vom Compiler ignoriert. Java kennt drei Arten von Kommentaren:

- *Einzeilige Kommentare*
  Diese beginnen mit den Zeichen `//` und enden am Ende der aktuellen Zeile. Beispiel:
  ```
  int z;  // zählt die gelesenen Zeilen
  ```
- *Mehrzeilige Kommentare*
  Diese beginnen mit `/*`, enden mit `*/` und können sich über mehrere Zeilen erstrecken. Beispiel:
  ```
  /* Diese Zeilen stellen
     einen mehrzeiligen
     Kommentar dar
  */
  ```
  Die Positionierung im Quelltext ist vollkommen frei, `/*` und `*/` müssen nicht am Zeilenanfang stehen.
- *Dokumentationskommentare*
  Diese beginnen mit `/**`, enden mit `*/` und können sich ebenfalls über mehrere Zeilen erstrecken. Sie werden vom SDK-Werkzeug `javadoc` zur automatischen Generierung von Programmdokumentation verwendet.

Bezeichner

Bezeichner sind Namen für Variablen, Klassen, Interfaces, Methoden (Funktionen) und Pakete. Sie können aus beliebig vielen Unicode-Buchstaben und Ziffern bestehen. Zusätzlich sind der Unterstrich _ und das Dollarzeichen $ erlaubt. Das erste Zeichen eines Namens darf keine Ziffer sein. Es wird zwischen Groß- und Kleinschreibung der Namen unterschieden. Die Bezeichner dürfen nicht mit den Schlüsselwörtern der Sprache und den Literalen `true`, `false` und `null` übereinstimmen.

Unicode ist ein standardisierter Zeichensatz, mit dem die Schriftzeichen aller gängigen Sprachen dargestellt werden können. Ein Unicode-Zeichen ist 16 Bit lang, was demnach 65.536 verschiedene Zeichen ermöglicht. Die ersten 128 Zeichen des Unicode-Zeichensatzes sind die 7-Bit-ASCII-Zeichen.

Schlüsselwörter

Die folgende Liste enthält die für Java reservierten *Schlüsselwörter*. Sie dürfen nicht als Bezeichner verwendet werden:

```
abstract   double      int        super
boolean    else        interface  switch
break      extends     long       synchronized
byte       final       native     this
case       finally     new        throw
catch      float       package    throws
char       for         private    transient
class      goto        protected  try
const      if          public     void
continue   implements  return     volatile
default    import      short      while
do         instanceof  static
```

Namens-konventionen

Namenskonventionen erhöhen die Lesbarkeit von Programmen. Folgende Regeln haben sich durchgesetzt:

- Namen werden mit gemischten Groß- und Kleinbuchstaben geschrieben, wobei Großbuchstaben zum Trennen von Wortstämmen dienen.
- *Variablennamen* beginnen mit einem Kleinbuchstaben, z.B. `args`, `meinKonto`. Namen von *Konstanten* (Variablen mit unveränderbarem Wert) bestehen aus Groß-

buchstaben. Einzelne Wörter werden durch _ getrennt, z.B. `KEY_FIRST`.

- *Methoden* werden nach Verben benannt, ihre Namen beginnen ebenfalls mit einem Kleinbuchstaben, z.B. `berechne`, `zeichneFigur`.
- *Klassen- und Interfacenamen* beginnen mit einem Großbuchstaben, z.B. `ErstesBeispiel`. Da Klassennamen als Teil des Namens der Datei auftauchen, die die entsprechende Klasse im Bytecode enthält, unterliegen diese auch den Regeln des jeweiligen Dateisystems.

2.2 Variablen und Datentypen

Daten werden in Programmen durch *Variablen* repräsentiert. Einer Variablen entspricht ein Speicherplatz im Hauptspeicher, in dem der aktuelle Wert der Variablen abgelegt ist. Variablen können nur ganz bestimmte Werte aufnehmen. Dies wird durch den *Datentyp* festgelegt.

Variablendeklaration

Die Deklaration einer Variablen erfolgt in der Form

```
Typname Variablenname;
```

Hierdurch wird Speicherplatz eingerichtet. Der Variablen kann gleich durch eine explizite *Initialisierung* ein Wert zugewiesen werden. Beispiel:

```
int nummer = 10;
```

Mehrere Variablen des gleichen Typs können in einer Liste, in der die Variablennamen dem Datentyp folgen und durch Kommata getrennt sind, deklariert werden. Beispiel:

```
int alter, groesse;
```

Java kennt acht so genannte *primitive Datentypen*:

Tabelle 2.1: Primitive Datentypen

Datentyp	Bit	Wertebereich
`boolean`		`false, true`
`char`	16	`0 ... 65.535`
`byte`	8	`-128 ... 127`
`short`	16	`-32.768 ... 32.767`
`int`	32	`-2.147.483.648 ...` `2.147.483.647`
`long`	64	`-9.223.372.036.854.775.808 ...` `9.223.372.036.854.775.807`
`float`	32	Absolutbetrag ca. $1,4 * 10^{-45} \ldots 3,4 * 10^{38}$ Genauigkeit ca. 7 Stellen
`double`	64	Absolutbetrag ca. $4,9 * 10^{-324} \ldots 1,8 * 10^{308}$ Genauigkeit ca. 15 Stellen

Konstante Werte bezeichnet man als *Literale*.

Wahrheitswerte

Der *logische Typ* `boolean` kennt zwei verschiedene Literale: `true` und `false`. Dieser Datentyp wird dort verwendet, wo ein logischer Operand erforderlich ist (z.B. bei Bedingungen in Fallunterscheidungen und Schleifen). Eine Umwandlung der Wahrheitswerte in ganzzahlige Werte ist nicht möglich.

Zeichen

Der *Zeichentyp* `char` dient dazu, einzelne Zeichen des Unicode-Zeichensatzes zu speichern. Literale werden in einfache Anführungszeichen eingeschlossen und als Unicode-Zeichen oder als so genannte *Escape-Sequenzen* angegeben:

Tabelle 2.2: Escape-Sequenzen

Escape-Sequenz	Bedeutung
`\b`	Backspace
`\t`	horizontaler Tabulator
`\n`	Zeilentrenner (Newline)

`\f`	Seitenumbruch (Formfeed)
`\r`	Wagenrücklauf (Carriage return)
`\"`	doppeltes Anführungszeichen "
`\'`	einfaches Anführungszeichen '
`\\`	Backslash \
`\x`*dd*	ASCII-codiertes Zeichen in Hexadezimalschreibweise
`\`*ddd*	ASCII-codiertes Zeichen in Oktalschreibweise
`\u`*dddd*	Unicode-Zeichen in Hexadezimalschreibweise (z.B. `\u0020` für das Leerzeichen)

Für die deutschen Umlaute und ß gelten die folgenden Unicode-Escapes:

Ä	`\u00c4`	ä	`\u00e4`	ß	`\u00df`
Ö	`\u00d6`	ö	`\u00f6`		
Ü	`\u00dc`	ü	`\u00fc`		

Ganze Zahlen

Die *ganzzahligen Typen* `byte`, `short`, `int`, `long` sind vorzeichenbehaftet. Literale können in Dezimal-, Oktal- oder Hexadezimalform geschrieben werden. Ein oktaler Wert beginnt mit dem Präfix `0`, ein hexadezimaler Wert mit dem Präfix `0x` oder `0X`. Gültige Ziffern sind bei dezimalen Literalen `0` bis `9`, bei oktalen Literalen `0` bis `7` und bei hexadezimalen Literalen `a` bis `f` und `A` bis `F`. Negative Zahlen werden durch Voranstellen des Minuszeichens `-` dargestellt. Ganzzahlige Literale sind vom Typ `int`, wenn nicht der Buchstabe `l` oder `L` angehängt ist. In diesem Fall sind sie vom Typ `long`.

Fließkommazahlen

Literale der *Fließkommatypen* `float` und `double` werden in Dezimalschreibweise notiert. Sie bestehen aus einem Vorkommateil, einem Dezimalpunkt, einem Nachkommateil, einem Exponenten und einem Suffix. Es muss mindestens der

Dezimalpunkt, der Exponent oder das Suffix vorhanden sein. Entweder der Vorkommateil oder der Nachkommateil darf wegfallen. Dem Vorkommateil und dem Exponenten kann ein Vorzeichen + oder - vorangestellt werden. Der Exponent, der durch `e` oder `E` eingeleitet wird, und das Suffix sind optional. Das Suffix `f` oder `F` kennzeichnet ein `float`-Literal, das Suffix `d` oder `D` ein `double`-Literal. Falls kein Suffix angegeben ist, handelt es sich um ein `double`-Literal.

Beispiele: `1.0` `.5` `-1.3e8` `4.578f`

Zeichenketten

Konstante *Zeichenketten* sind Folgen von Zeichenliteralen, die in doppelte Anführungszeichen eingeschlossen sind.

Beispiel: `"Das ist eine Zeichenkette"`

Einen zugehörigen primitiven Datentyp gibt es allerdings nicht, sondern eine Klasse `String`, über die Zeichenketten erzeugt werden können. Mit dem Operator + können Zeichenketten aneinander gehängt werden.

Im folgenden Beispiel wird eine Zeichenkette mit einem booleschen Wert bzw. einer ganzen Zahl durch + verknüpft. Dabei wird der boolesche Wert bzw. die Zahl automatisch in eine Zeichenkette umgewandelt.

Programm 2.1

```
// Datei: VarTest.java

public class VarTest {
   public static void main(String[] args) {
      boolean b = true;
      char c = 'x';
      int i = 4711;
      double d = 1.2345;

      System.out.println("b: " + b);
      System.out.println("c: " + c);
      System.out.println("i: " + i);
      System.out.println("d: " + d);
   }
}
```

Die Ausgabe des Programms ist:

```
b: true
c: x
i: 4711
d: 1.2345
```

2.3 Ausdrücke und Operatoren

Mit *Operatoren* können Zuweisungen und Berechnungen vorgenommen und Bedingungen formuliert und geprüft werden. Operatoren sind Bestandteile von Ausdrücken.

Ausdruck

Ein *Ausdruck* besteht aus Operatoren, Operanden, auf die ein Operator angewandt wird, und Klammern, die zusammen eine Auswertungsvorschrift beschreiben. Operanden können Variablen und Literale, aber auch Methodenaufrufe sein. Jeder Ausdruck hat einen Rückgabewert, dessen Typ sich aus den Typen der Operanden und der Art des Operators bestimmt. Vorrangregeln legen die Reihenfolge der Auswertung fest, wenn mehrere Operatoren im Ausdruck vorkommen. In den folgenden Tabellen, die die verschiedenen Operatoren aufführen, wird die Rangfolge durch Zahlen notiert. Priorität 1 kennzeichnet den höchsten Rang. Durch Setzen von runden Klammern lässt sich eine bestimmte Auswertungsreihenfolge erzwingen:

`2+3*4` hat den Wert `14`, `(2+3)*4` hat den Wert `20`.

Bei den Operatoren unterscheidet man arithmetische, relationale, logische, Bit-, Zuweisungs- und sonstige Operatoren.

Arithmetische Operatoren

Die *arithmetischen Operatoren* haben numerische Operanden und liefern einen numerischen Rückgabewert.

implizite Typumwandlung

Haben die Operanden unterschiedliche Datentypen, so wird automatisch eine *Typumwandlung* "nach oben" durchgeführt: der kleinere Typ der beiden Operanden wird in den Typ des

größeren umgewandelt. Der Ergebnistyp entspricht dem größeren der beiden Operanden.

Man beachte, dass bei Division von ganzen Zahlen der Nachkommateil abgeschnitten wird: `13/5` hat den Wert 2.

Der *Modulo-Operator* `%` berechnet den Rest `r = a % b` einer ganzzahligen Division von `a` durch `b` so, dass gilt:

```
a = [a/b] * b + r,
```

wobei `[...]` für das Ergebnis der ganzzahligen Division steht.

Beispiel: `13 % 5` hat den Wert 3
`13.8 % 5.6` hat den Wert `2.6`

Tabelle 2.3: Arithmetische Operatoren

Operator	Bezeichnung	Priorität
+	positives Vorzeichen	1
-	negatives Vorzeichen	1
++	Inkrement	1
--	Dekrement	1
*	Multiplikation	2
/	Division	2
%	Modulo	2
+	Addition	3
-	Subtraktion	3

Beim einstelligen *Inkrement-* und *Dekrementoperator*, der sich nur auf Variablen anwenden lässt, wird zwischen *Präfix-* und *Postfixform* unterschieden, je nachdem, ob der Operator vor oder hinter dem Operanden steht:

`++a` hat den Wert von `a+1`, `a` wird um 1 erhöht,
`--a` hat den Wert von `a-1`, `a` wird um 1 verringert,
`a++` hat den Wert von `a`, `a` wird um 1 erhöht,
`a--` hat den Wert von `a`, `a` wird um 1 verringert.

Programm 2.2

```
// Datei: ArithmOp.java

public class ArithmOp {
   public static void main(String[] args) {
      double z = 13.8, n = 5.6;
      int a = 5;

      System.out.println("z/n: " + z/n);
      System.out.println("z%n: " + z%n);
      System.out.print("++a: " + ++a + " , ");
      System.out.println("a: " + a);
      System.out.print("a++: " + a++ + " , ");
      System.out.println("a: " + a);
      System.out.print("--a: " + --a + " , ");
      System.out.println("a: " + a);
      System.out.print("a--: " + a-- + " , ");
      System.out.println("a: " + a);
   }
}
```

Die Ausgabe des Programms ist:

```
z/n: 2.4642857142857144
z%n: 2.6000000000000014
++a: 6 , a: 6
a++: 6 , a: 7
--a: 6 , a: 6
a--: 6 , a: 5
```

Relationale Operatoren

Relationale Operatoren vergleichen Ausdrücke mit numerischem Wert miteinander. Das Ergebnis ist vom Typ `boolean`.

Bei Fließkommawerten sollte die Prüfung auf exakte Gleichheit oder Ungleichheit vermieden werden, da es bei längeren Rechnungen zu Rundungsfehlern kommen kann und die erwartete Gleichheit oder Ungleichheit nicht zutrifft. Stattdessen sollte mit den Operatoren <= und >= gearbeitet werden, um die Übereinstimmung der Werte bis auf einen relativen Fehler zu prüfen.

Tabelle 2.4: Relationale Operatoren

Operator	Bezeichnung	Priorität
<	kleiner	5
<=	kleiner oder gleich	5
>	größer	5
>=	größer oder gleich	5
==	gleich	6
!=	ungleich	6

Logische Operatoren

Logische Operatoren verknüpfen Wahrheitswerte miteinander. Java stellt die Operationen UND, ODER, NICHT und das "exklusive ODER" zur Verfügung.

Tabelle 2.5: Logische Operatoren

Operator	Bezeichnung	Priorität
!	NICHT	1
&	UND mit vollständiger Auswertung	7
^	exklusives ODER	8
|	ODER mit vollständiger Auswertung	9
&&	UND mit kurzer Auswertung	10
||	ODER mit kurzer Auswertung	11

UND und ODER gibt es in zwei Varianten. Bei der so genannten kurzen Variante (*short circuit*) wird der zweite Operand nicht mehr ausgewertet, wenn das Ergebnis des Gesamtausdrucks schon feststeht. Z.B. ist A UND B falsch, wenn A falsch ist, unabhängig vom Wahrheitswert von B.

`!a` ergibt `false`, wenn `a` den Wert `true` hat, und `true`, wenn `a` den Wert `false` hat.

Die folgende Tabelle zeigt die Ergebnisse der übrigen Operatoren:

Tabelle 2.6: Verknüpfung von Wahrheitswerten

a	b	a & b a && b	a ^ b	a | b a || b
true	true	true	false	true
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

Programm 2.3

```
// Datei: LogOp.java

public class LogOp {
   public static void main(String[] args) {
      int a = 2, b = 3;

      System.out.println(a == 2 && b < 8);
      System.out.println(a != 2 || !(b < 2));
      System.out.println(a == 2 ^ b > 0);
   }
}
```

Die Ausgabe des Programms ist:

```
true
true
false
```

Bitoperatoren

Bitoperatoren arbeiten auf der Binärdarstellung der Operanden, also mit 8 (`byte`), 16 (`short`, `char`), 32 (`int`) oder 64 Bit (`long`).

Tabelle 2.7: Bitoperatoren

Operator	Bezeichnung	Priorität
~	Bitkomplement	1
<<	Linksschieben	4
>>	Rechtsschieben	4
>>>	Rechtsschieben mit Nachziehen von Nullen	4
&	bitweises UND	7
^	bitweises exklusives ODER	8
|	bitweises ODER	9

`~a` entsteht aus `a`, indem alle Bit von `a` invertiert werden, d.h. `0` geht in `1` und `1` in `0` über. Die folgende Tabelle zeigt die Ergebnisse der Verknüpfungen der jeweils korrespondierenden einzelnen Bit der beiden Operanden:

Tabelle 2.8: Bitweise Verknüpfungen

a	b	a & b	a ^ b	a | b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Bei den *Schiebeoperatoren* werden alle Bit des ersten Operanden um so viele Stellen nach links bzw. rechts geschoben, wie im zweiten Operanden angegeben ist. Beim *Linksschieben* werden von rechts Nullen nachgezogen. Beim *Rechtsschieben* werden von links Nullen nachgezogen, falls der erste Operand positiv ist. Ist er negativ, werden Einsen nachgezogen. Beim Operator >>> werden von links immer Nullen nachgezogen.

Zuweisungsoperatoren

Neben der einfachen *Zuweisung* können arithmetische und bitweise Operatoren mit der Zuweisung kombiniert werden. Bei der einfachen Zuweisung = wird der rechts stehende Ausdruck ausgewertet und der links stehenden Variablen zugewiesen. Dabei müssen die Datentypen beider Seiten kompatibel sein. Eine automatische Umwandlung der rechten Seite "nach oben" in den Typ der Variablen wird gegebenenfalls durchgeführt.

Tabelle 2.9: Zuweisungsoperatoren

Operator	Bezeichnung	Priorität
=	einfache Zuweisung	13
op=	kombinierte Zuweisung, dabei steht *op* für *, /, %, +, -, <<, >>, >>>, &, ^, |	13

a *op*= b entspricht a = a *op* b, wobei für *op* der anzuwendende Operator einzusetzen ist, z.B. a += b.

Programm 2.4

```
// Datei: ZuweisOp.java

public class ZuweisOp {
   public static void main(String[] args) {
      int a, b = 5, c = 9;

      a = b + c;
      System.out.println("a: " + a);
      a += b;
      System.out.println("a: " + a);
      a = b = 0;
      System.out.println("a: " + a + " , b: " + b);
   }
}
```

Die Ausgabe des Programms ist:

```
a: 14
a: 19
a: 0 , b: 0
```

Bedingungsoperator

Der *Bedingungsoperator* `?:` benötigt drei Operanden:

`Bedingung ? Ausdruck1 : Ausdruck2`

Bei `Bedingung` handelt es sich um einen logischen Ausdruck. Falls `Bedingung` wahr ist, ergibt sich der Wert von `Ausdruck1`, sonst der Wert von `Ausdruck2`. Der Bedingungsoperator hat die Priorität 12.

Beispiel:
Der Ausdruck `a < 0 ? -a : a` hat den Wert des Absolutbetrags von `a`, wenn `a` eine Zahl ist.
Der Ausdruck `a < b ? b : a` ergibt das Maximum der beiden Zahlen `a` und `b`.

Typumwandlung

Cast-Operator

Mit dem *Cast-Operator* wird eine explizite *Typumwandlung* vorgenommen. Der Ausdruck `(type)a` wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um. Der Cast-Operator darf nur auf der rechten Seite einer Zuweisung auftauchen. Er hat die Priorität 1.

Beispiel:
Um einer `int`-Variablen `b` einen `double`-Wert zuzuweisen, ist folgende Typumwandlung erforderlich: `b = (int)a;`. Hierbei wird der Nachkommateil ohne Rundung abgeschnitten. Bei der Umwandlung einer `short`-Variablen `a` in einen `byte`-Wert `b` durch `b = (byte)a;` wird das höherwertige Byte von `a` abgeschnitten. Hat z.B. `a` den Wert `257`, so hat `b` nach der Zuweisung den Wert `1`.

Das folgende Beispielprogramm zeigt, wie eine *reelle Division* ganzer Zahlen erzwungen werden kann:

Programm 2.5

```
// Datei: CastOp.java

public class CastOp {
   public static void main(String[] args) {
      double z;
      int x = 5, y = 3;
      z = x / y;
      System.out.println("z: " + z);
      z = (double)x / y;
      System.out.println("z: " + z);
   }
}
```

Die Ausgabe des Programms ist:

```
z: 1.0
z: 1.6666666666666667
```

2.4 Kontrollstrukturen

Anweisungen

Anweisungen stellen die kleinsten ausführbaren Einheiten eines Programms dar. Eine Anweisung kann eine Deklaration enthalten, einen Ausdruck (Zuweisung, Inkrement, Dekrement, Methodenaufruf, Erzeugung eines Objekts) auswerten oder den Ablauf des Programms steuern. Das Semikolon ; markiert das Ende einer Anweisung. Die *leere Anweisung* ; wird dort benutzt, wo syntaktisch eine Anweisung erforderlich ist, aber von der Programmlogik her nichts zu tun ist.

Block

Ein *Block* ist ein Paar von geschweiften Klammern, die Anweisungen enthalten. Ein Block gilt als eine einzelne Anweisung und kann überall dort verwendet werden, wo eine elementare Anweisung erlaubt ist. Damit können Blöcke ineinander geschachtelt werden. Variablen, die in einem Block deklariert werden, sind nur dort gültig und sichtbar.

Verzweigungen

Verzweigungen erlauben die bedingte Ausführung von Anweisungen.

if ... else

Die `if`-*Anweisung* tritt in zwei Varianten auf:

```
if (Ausdruck)
   Anweisung
```

oder

```
if (Ausdruck)
   Anweisung1
else
   Anweisung2
```

Ausdruck hat den Typ `boolean`. Im ersten Fall wird *Anweisung* nur ausgeführt, wenn *Ausdruck* den Wert `true` hat. Im zweiten Fall wird *Anweisung1* ausgeführt, wenn *Ausdruck* den Wert `true` hat, andernfalls wird *Anweisung2* ausgeführt. Ein `else` wird dem nächstmöglichen `if` zugeordnet. Das zeigt u.a. folgendes Beispielprogramm:

Programm 2.6

```
// Datei: IfTest.java

public class IfTest {
   public static void main(String[] args) {
      int a = 2, b, c = -1;

      if (a >= 0) {
         System.out.println("a ist groesser gleich 0");
         b = 1;
      }
      else {
         System.out.println("a ist kleiner 0");
         b = -1;
      }

      if (b == 1)
         if (c > 0)
            System.out.println("c ist groesser 0");
         else
            System.out.println("c ist kleiner gleich 0");
   }
}
```

Die Ausgabe des Programms ist:

```
a ist groesser gleich 0
c ist kleiner gleich 0
```

switch

Die `switch`-*Anweisung* führt je nach Wert des Ausdrucks, den sie wie ein Argument erhält, alternative Anweisungen aus.

```
switch (Ausdruck) {
   case Konstante:
      Anweisungen

   ...

   default:
      Anweisungen
}
```

Ausdruck muss vom Typ `char`, `byte`, `short` oder `int` und *Konstante* ein konstanter Ausdruck sein. Diese konstanten Ausdrücke müssen paarweise verschiedene Werte haben. In Abhängigkeit vom Wert von *Ausdruck* wird die Sprungmarke angesprungen, deren Konstante mit dem Wert des Ausdrucks übereinstimmt. Dann werden *alle* dahinter stehenden Anweisungen bis zum Ende der switch-Anweisung oder bis zum ersten `break` ausgeführt. Die Anweisung `break` führt zum sofortigen Verlassen der switch-Anweisung. Die optionale Marke `default` wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird. *Anweisungen* steht für keine, eine oder mehrere einzelne Anweisungen.

Programm 2.7

```
// Datei: SwitchTest.java

public class SwitchTest {
   public static void main(String[] args) {
      char wahl = 's';
```

```
        switch (wahl) {
           case 'n':
           case 'N':
              System.out.println("Option Neu");
              break;
           case 'o':
           case 'O':
              System.out.println("Option Oeffnen");
              break;
           case 's':
           case 'S':
              System.out.println("Option Speichern");
              break;
           case 'e':
           case 'E':
              System.out.println("Option Ende");
              break;
           default:
              System.out.println("Option ungueltig");
        }
     }
  }
```

Die Ausgabe des Programms ist:

```
Option Speichern
```

Schleifen

Schleifen führen Anweisungen wiederholt aus, solange bis eine Bedingung erfüllt ist.

while

Die `while`-*Schleife* ist eine *abweisende* Schleife, d.h. die Ausführungsbedingung wird jeweils vor Eintritt in die Schleife überprüft.

```
while (Ausdruck)
   Anweisung
```

Ausdruck muss vom Typ `boolean` sein. Hat *Ausdruck* den Wert `true`, wird *Anweisung* ausgeführt, andernfalls

wird mit der Anweisung fortgefahren, die dem Schleifenkörper folgt.

do ... while

Die do-*Schleife* ist eine *nicht abweisende* Schleife, d.h. die Anweisungen im Schleifenkörper werden mindestens einmal ausgeführt.

```
do
   Anweisung
while (Ausdruck);
```

Die Schleife wird beendet, wenn *Ausdruck*, der vom Typ `boolean` sein muss, den Wert `false` hat.

Programm 2.8

```
// Datei: WhileTest.java
import java.io.*; // (*)

public class WhileTest {
   public static void main(String[] args) throws // (*)
      IOException {
      BufferedReader ein = new BufferedReader(
         new InputStreamReader(System.in)); // (*)
      int zahl, i = 1, summe = 0;

      do {
         System.out.print("Bitte Zahl eingeben: ");
         zahl = Integer.valueOf(ein.readLine()).
                  intValue(); // (*)
      } while (zahl <= 0);

      ein.close(); // (*)

      while (i <= zahl) {
         summe += i;
         ++i;
      }

      System.out.println("Summe 1 bis " + zahl + ": " +
         summe);
   }
}
```

Das Programm liest eine Zahl von der Tastatur ein und prüft, ob sie größer 0 ist. Ansonsten wird die Eingabeaufforderung wiederholt. Anschließend werden die Zahlen von 1 bis zur eingegebenen Zahl aufaddiert. Das Ergebnis wird ausgegeben. Die mit `(*)` gekennzeichneten Konstrukte werden in späteren Kapiteln erklärt.

Beispiel:

```
Bitte Zahl eingeben: 0
Bitte Zahl eingeben: 100
Summe 1 bis 100: 5050
```

for

Die `for`-*Schleife* wiederholt eine Anweisung in Abhängigkeit von Kontrollausdrücken.

```
for (Init; Bedingung; Update)
  Anweisung
```

`Init` ist eine Liste von durch Kommata voneinander getrennten Anweisungen oder Variablendeklarationen. *`Init`* wird einmal vor dem Start der Schleife aufgerufen. Dieser Initialisierungsteil darf auch fehlen.

`Bedingung` ist ein Ausdruck vom Typ `boolean`. Die Bedingung wird zu Beginn jedes Schleifendurchgangs getestet. *`Anweisung`* wird nur ausgeführt, wenn *`Bedingung`* den Wert `true` hat. Fehlt dieser Ausdruck, so wird `true` als Bedingung angenommen.

`Update` ist eine Liste von durch Kommata voneinander getrennten Anweisungen. Sie kann auch leer sein. *`Update`* wird nach jedem Durchlauf der Schleife ausgewertet, bevor *`Bedingung`* das nächste Mal ausgewertet wird. In den meisten Fällen dient *`Update`* dazu, den Schleifenzähler zu verändern und damit die Laufbedingung zu beeinflussen.

Programm 2.9

```
// Datei: ForTest.java

public class ForTest {
   public static void main(String[] args) {
      int zahl = 100, summe = 0;

      for (int i = 1; i <= zahl; ++i)
         summe += i;

      System.out.println("Summe 1 bis " + zahl + ": " +
         summe);
   }
}
```

Die Ausgabe des Programms ist:

```
Summe 1 bis 100: 5050
```

Sprunganweisungen

Sprunganweisungen werden hauptsächlich verwendet, um Schleifendurchgänge vorzeitig zu beenden.

break

Die Anweisung `break` beendet eine `while`-, `do`-, `for`- oder `switch`-Anweisung, die die `break`-Anweisung unmittelbar umgibt. Um aus geschachtelten Schleifen herauszuspringen, gibt es eine Variante:

`break` *`Marke`*`;`

`Marke` steht für einen selbst gewählten Bezeichner. Diese Anweisung verzweigt an das Ende der Anweisung, an deren Anfang diese Marke steht. Eine *markierte Anweisung* hat die Form:

`Marke``:` *`Anweisung`*

`break`-Anweisungen mit Marke dürfen auch in beliebigen markierten Blöcken genutzt werden.

continue

Die Anweisung `continue` unterbricht den aktuellen Schleifendurchgang einer `while`-, `do`- oder `for`-Schleife und springt an die Wiederholungsbedingung der sie unmittelbar

umgebenden Schleife. Wie bei der `break`-Anweisung gibt es auch hier die Variante mit Marke:

```
continue Marke;
```

Programm 2.10

```
// Datei: SprungTest.java

public class SprungTest {
   public static void main(String[] args) {
      int i, j;

      Hauptschleife:
      for (i = 1; i <= 10; ++i)
         for (j = 1; j <= 10; ++j) {
            System.out.print(i * j + "\t");
            if (j == 5) {
               System.out.println();
               continue Hauptschleife;
            }
         }
   }
}
```

Die Ausgabe des Programms ist:

```
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
4       8       12      16      20
5       10      15      20      25
6       12      18      24      30
7       14      21      28      35
8       16      24      32      40
9       18      27      36      45
10      20      30      40      50
```

2.5 Aufgaben

1. Welche der folgenden Bezeichner sind ungültig?

   ```
   Hallo_Welt   $Test    _abc       2test
   #hallo       te?st    Girokonto  const
   ```

2. Warum führt der folgende Code zu einem Compiler-Fehler?

   ```
   int x = 0;
   long y = 1000;
   x = y;
   ```

3. Schreiben Sie ein Programm, das mit einem einzigen Aufruf von `System.out.print` eine Zeichenkette in mehreren Bildschirmzeilen ausgibt.

   Benutzen Sie für diese Aufgabe und auch für alle anderen Programmieraufgaben dieses Kapitels folgenden Rahmen:

   ```
   public class Klassenname {
    public static void main(String[] args) {
            ...
    }
   }
   ```

   Die Datei, die den Quellcode enthält, muss den Namen `Klassenname.java` haben. Für `Klassenname` ist ein selbst zu wählender Bezeichner einzusetzen, z.B. `Aufg0203` (`0203` steht für die 3. Aufgabe im 2. Kapitel).

4. Schreiben Sie ein Programm, das die Anzahl von Sekunden im Monat Januar berechnet.

5. Schreiben Sie ein Programm, das ermittelt, wie hoch ein Sparguthaben von 5.000 Geldeinheiten bei 7,5 % Verzinsung nach Ablauf eines Jahres ist.

6. Welche Werte haben die folgenden Ausdrücke und welche Werte haben die Variablen nach der Aus-

wertung, wenn `a` den Anfangswert `1` und `b` den Anfangswert 7 hat?

a) `--a` b) `a--` c) `a++ + b`

d) `b = ++a` e) `a = b++` f) `-(a--) - -(--b)`

7. Schreiben Sie ein Programm, das `n` ganze Zahlen von der Tastatur einliest und den Mittelwert in einer `while`-Schleife berechnet. Schauen Sie sich hierzu das Programm 2.8 an.

8. Schreiben Sie ein Programm, das eine bestimmte Anzahl von Sternchen (`*`) in Form eines Dreiecks auf dem Bildschirm ausgibt.

9. Schreiben Sie ein Programm, das den Buchwert in Höhe von 15.000 Geldeinheiten mit einem Abschreibungssatz von 40 % und einem Restwert von 100 Geldeinheiten geometrisch degressiv abschreibt.

10. Schreiben Sie ein Programm, das eine ganze Zahl vom Typ `int` in Binärdarstellung (32 Bit) ausgibt. Benutzen Sie hierzu die Bitoperatoren `&` und `<<`.

3 Objektorientierte Programmierung

Dieses Kapitel behandelt die objektorientierten Sprachelemente von Java. Im Vordergrund stehen die Begriffe *Klasse*, *Objekt*, *Attribut*, *Methode*, *Vererbung* und *Polymorphismus*. Die objektorientierte Programmierung löst sich von dem Prinzip der Trennung von Daten und Funktionen, wie sie in den traditionellen Programmiersprachen der dritten Generation üblich ist. Stattdessen werden Daten und Funktionen zu selbständigen Modellierungseinheiten, den Klassen, gekapselt.

3.1 Klassen und Objekte

Java-Programme bestehen aus Klassendefinitionen. Eine *Klasse* ist eine allgemeingültige Beschreibung von Dingen, die in verschiedenen Ausprägungen vorkommen können, aber alle eine gemeinsame Struktur und ein gemeinsames Verhalten haben. Sie ist ein Bauplan für die Erzeugung von einzelnen konkreten Ausprägungen. Diese Ausprägungen bezeichnet man als *Objekte* oder *Instanzen* der Klasse. Variablen, auch *Attribute* genannt, und *Methoden* (Funktionen) sind die Bestandteile einer Klasse. Attribute beschreiben den *Zustand* und Methoden das *Verhalten* eines Objekts.

Klasse

Der allgemeine Aufbau einer *Klasse* ist:

```
[Modifizierer] class Klassenname
   [extends Basisklasse]
   [implements Schnittstellen] {
   Attributdeklarationen
   Methodendeklarationen
}
```

Die in eckigen Klammern angegebenen Einheiten sind optional und werden an späterer Stelle erklärt. `Modifizierer` schränken die Verwendung der Klasse ein, `Basisklasse` ist die Klasse, von der die vorliegende Klasse bei Vererbung ab-

geleitet ist, `Schnittstellen` steht für die Namen der Interfaces, deren Eigenschaften von der Klasse realisiert werden.

Beispiel:

```
class Konto {
   int kontonummer;
   double saldo;

   int getKontonummer() {
      return kontonummer;
   }

   void setKontonummer(int nr) {
      kontonummer = nr;
   }

   double getSaldo() {
      return saldo;
   }

   void setSaldo(double betrag) {
      saldo = betrag;
   }

   void zahleEin(double betrag) {
      saldo += betrag;
   }

   void zahleAus(double betrag) {
      saldo -= betrag;
   }
}
```

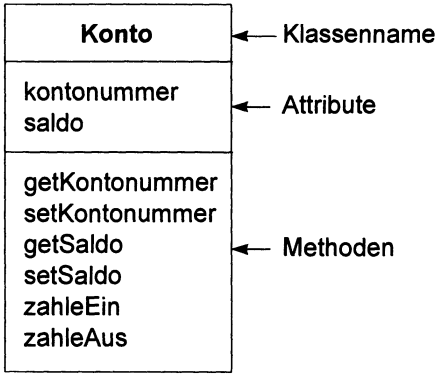
Die Klasse `Konto` enthält die Attribute `kontonummer` und `saldo` und vier Methoden, die die Attribute mit Werten versehen bzw. die aktuellen Attributwerte zurückliefern, sowie zwei Methoden, die den Saldo erhöhen bzw. vermindern.

Quelldatei

Eine Quelldatei sollte *nur eine* Klassendefinition enthalten, wobei die Datei dann den gleichen Namen wie die Klasse

(abgesehen von der Endung `.java`) trägt. Damit besteht eine eindeutige Zuordnung zwischen Klassen und Quelldateien, was der Übersichtlichkeit in einem Projekt dient. Sind mehrere Klassen in einer Datei definiert, so darf höchstens eine den Modifizierer `public` haben. Diese Klasse bestimmt dann den Namen der Datei.

Bild 3.1: Klasse Konto



Objekt

Um von einer Klasse ein *Objekt* zu erzeugen, wird eine Variable vom Typ der Klasse deklariert und anschließend ein mit Hilfe des Operators `new` neu angelegtes Objekt dieser Variablen zugewiesen.

Beispiel:

```
Konto meinKonto;
meinKonto = new Konto();
```

oder zusammengefasst:

```
Konto meinKonto = new Konto();
```

Der Operator `new` erzeugt mit Hilfe der speziellen Methode `Konto()` Speicherplatz für ein neues Objekt vom Typ `Konto`.

Referenzvariable

Die Variable `meinKonto` enthält nur einen Adressverweis auf diesen Speicherplatz. Solche Variablen werden *Referenzvariablen* genannt. Klassen sind also *Referenztypen*. Die

vordefinierte Konstante `null` bezeichnet eine leere Referenz. Referenzvariablen mit Wert `null` verweisen nirgendwohin.

Beispiel:

```
Konto meinKonto = new Konto();
Konto k;
k = meinKonto;
```

Die Variablen `meinKonto` und `k` verweisen beide auf dasselbe Objekt.

Für zwei Referenzvariablen `x` und `y` hat `x == y` den Wert `true`, wenn beide auf dasselbe Objekt verweisen.

Der Datentyp eines Attributs kann ein primitiver Datentyp oder ein Referenztyp sein. Nicht initialisierte Attribute erhalten bei der Objekterzeugung einen Standardwert: für Zahlen `0`, für Variablen vom Typ `boolean` `false` und für Referenzvariablen `null`.

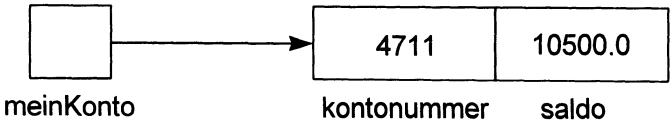
Um auf Attribute oder Methoden eines Objekts zugreifen zu können, wird die *Punktnotation* verwendet:
`Referenz.Attribut` bzw. *`Referenz.Methode`*

Programm 3.1

```
// Datei: KontoTest.java

public class KontoTest {
   public static void main(String[] args) {
      Konto meinKonto = new Konto();
      meinKonto.setKontonummer(4711);
      meinKonto.setSaldo(500.);
      meinKonto.zahleEin(10000.);
      System.out.println(meinKonto.getSaldo());
   }
}
```

Die Ausgabe des Programms ist: `10500.0`

Bild 3.2: Objektreferenz



3.2 Methoden

Operationen auf Objekten werden mit *Methoden* realisiert. Methoden können nur innerhalb von Klassen vereinbart werden. Diese Vereinbarung sieht wie folgt aus:

Methode

```
[Modifizierer] Typ Methodenname([Parameter])
   [throws Exception-Liste] {
   Anweisungen
}
```

Die in eckigen Klammern angegebenen Einheiten sind optional. `Modifizierer` und `throws`-Klausel werden an späterer Stelle erklärt.

Rückgabetyp

Eine Methode kann einen Wert zurückgeben. Der *Rückgabetyp* kann ein primitiver Datentyp oder ein Referenztyp sein. Eine Methode, die keinen Wert zurückgibt, muss den Typ `void` haben.

Werte werden mit Hilfe der Anweisung `return` zurückgegeben:

```
return Ausdruck;
```

Bei `void`-Methoden fehlt der Ausdruck. Die `return`-Anweisung beendet die Methode. Der *Rückgabewert* kann an der Aufrufstelle weiterverwendet werden. `Ausdruck` muss zuweisungskompatibel zum Typ der Methode sein.

Parameter

Die in runden Klammern angegebenen *Parameter* (Typ und Name) spezifizieren die Argumente, die der Methode beim Aufruf übergeben werden. Mehrere Parameter werden durch Kommata getrennt.

call by value

Für jeden Parameter wird innerhalb der Methode ein eigener Speicherplatz erzeugt, in den der Parameterwert kopiert wird *(call by value)*. Dadurch bleiben die Variablen, deren Werte beim Aufruf der Methode als Argumente übergeben werden, unverändert. Ist das Argument vom Referenztyp, verweist also auf ein Objekt, so kann die Methode dieses Objekt verändern, da die Referenzvariable und ihre Kopie auf dasselbe Objekt verweisen. Das folgende Beispiel veranschaulicht den Parametermechanismus.

Programm 3.2

```
// Datei: ParamTest.java

public class ParamTest {
   void m(int x, Konto k) {
      x = 4711;
      k.setKontonummer(4711);
   }

   public static void main(String[] args) {
      ParamTest p = new ParamTest();
      Konto kto = new Konto();
      int a = 1000;
      kto.setKontonummer(1000);
      System.out.println("vorher: " + a + " " +
         kto.getKontonummer());
      p.m(a, kto);
      System.out.println("nachher: " + a + " " +
         kto.getKontonummer());
   }
}
```

Die Ausgabe des Programms ist:

```
vorher: 1000 1000
nachher: 1000 4711
```

lokale Variable

Variablen, die innerhalb einer Methode oder innerhalb eines Blocks einer Methode deklariert werden, bezeichnet man als *lokale Variablen*. Sie werden angelegt, wenn die Ausführung der Methode bzw. des Blocks beginnt, und werden zerstört, wenn die Methode bzw. der Block verlassen wird.

Beginnt die Deklaration einer lokalen Variablen mit `final`, so handelt es sich um eine Konstante, die entweder sofort bei ihrer Deklaration mit einem Wert initialisiert wird oder der einmal, vor dem ersten Zugriff, ein Wert zugewiesen wird.

Variablen, die im Initialisierungsteil des Kopfs einer `for`-Anweisung deklariert werden, sind nur innerhalb der `for`-Anweisung gültig. Innerhalb eines Blocks kann auf die Variablen der umgebenden Blöcke bzw. der umgebenden Methode sowie auf die Attribute der umschließenden Klasse

zugegriffen werden. Es ist nicht erlaubt, eine bereits deklarierte lokale Variable in einem tiefer geschachtelten Block erneut mit dem gleichen Namen zu deklarieren. Allerdings darf ihr Name mit einem Attributnamen übereinstimmen. Deklarationen lokaler Variablen können mit anderen Anweisungen gemischt werden.

this

Innerhalb einer Methode bezeichnet `this` einen Wert, der das Objekt referenziert, für das die Methode aufgerufen wurde. `this` wird benutzt, um auf "verdeckte" Attribute der eigenen Klasse zuzugreifen, die eigene Instanz als Wert zurückzugeben oder sie als Argument beim Aufruf einer Methode zu verwenden.

Beispiel:

```
void setKontonummer(int kontonummer) {
   this.kontonummer = kontonummer;
}
```

In diesem Beispiel verdeckt die lokale Variable das Attribut `kontonummer` der Klasse `Konto`. Mit `this` wird das Attribut trotz Namensgleichheit zugänglich.

Signatur

Der Name einer Methode zusammen mit der Reihenfolge und den Typen der Parameter heißt *Signatur*.

Überladen

Es können mehrere Methoden mit gleichem Namen, aber unterschiedlichen Signaturen deklariert werden. Der Compiler entscheidet beim Methodenaufruf, welche der deklarierten Methoden aufgerufen wird. Diese Technik nennt man *Überladen* von Methoden. Überladene Methoden unterscheiden sich also in der Parameterliste. Sie sollten allerdings eine vergleichbare Funktionalität haben.

Beispiel:

```
int max(int a, int b) {
   return a < b ? b : a;
}

double max(double a, double b) {
   return a < b ? b : a;
}
```

```
int max(int a, int b, int c) {
   return max(max(a, b), c);
}
```

3.3 Konstruktoren

Konstruktoren sind spezielle Methoden, die bei der Erzeugung eines Objekts aufgerufen werden und Initialisierungen sowie besondere Aufgaben ausführen können.

Ein *Konstruktor* trägt den Namen der zugehörigen Klasse und hat keinen Rückgabetyp. Ansonsten wird er wie eine Methode deklariert und kann auch überladen werden.

Standard-konstruktor

Wenn für eine Klasse kein Konstruktor explizit deklariert ist, wird ein *Standardkonstruktor* ohne Parameter vom Compiler eingebaut. Ist ein Konstruktor mit oder ohne Parameter explizit deklariert, so erzeugt der Compiler keinen Standardkonstruktor mehr.

Beispiel: Für die Klasse `Konto` werden Konstruktoren deklariert.

```
Konto() { }

Konto(int kontonummer) {
   this.kontonummer = kontonummer;
}

Konto(int kontonummer, double saldo) {
   this.kontonummer = kontonummer;
   this.saldo = saldo;
}
```

Programm 3.3

```
// Datei: KonstrTest.java

public class KonstrTest {
   public static void main(String[] args) {
      Konto k1 = new Konto();
      Konto k2 = new Konto(4711);
      Konto k3 = new Konto(1234, 1000.);
```

```
        System.out.println(k1.getKontonummer());
        System.out.println(k1.getSaldo());
        System.out.println(k2.getKontonummer());
        System.out.println(k2.getSaldo());
        System.out.println(k3.getKontonummer());
        System.out.println(k3.getSaldo());
    }
}
```

Die Ausgabe des Programms ist:

```
0
0.0
4711
0.0
1234
1000.0
```

Konstruktoren können sich gegenseitig aufrufen, so dass vorhandener Programmcode wiederverwendet werden kann. Der Aufruf eines Konstruktors muss als erste Anweisung mit dem Namen `this` erfolgen.

Beispiel: Der letzte Konstruktor des vorigen Beispiels hätte auch so geschrieben werden können:

```
Konto(int kontonummer, double saldo) {
   this(kontonummer);
   this.saldo = saldo;
}
```

Garbage Collector

Der durch `new` für Objekte reservierte Speicherplatz wird vom *Garbage Collector* (Müllabfuhr) des Java-Laufzeitsystems automatisch freigegeben, wenn er nicht mehr gebraucht wird. Wenn z.B. die Methode, in der ein Objekt erzeugt wurde, endet oder die Referenzvariable auf den Wert `null` gesetzt wurde, kann das Objekt nicht mehr benutzt werden, da keine Referenz darauf verweist. Wann der Garbage Collector startet, ist nicht festgelegt. Er startet spätestens jedoch dann, wenn Speicherplatz zur Erzeugung neuer Objekte benötigt wird oder der Garbage Collector explizit mit `System.gc()` aufgerufen wird.

Die Methode `void finalize()` wird automatisch für ein nicht mehr benötigtes Objekt vom Garbage Collector aufgerufen, bevor der durch das Objekt belegte Speicherplatz freigegeben wird. `finalize` kann durch eine Klasse implementiert werden und spezielle Aufräumarbeiten erledigen.

3.4 Statische Attribute und Methoden

Es können Attribute und Methoden deklariert werden, deren Nutzung nicht an die Existenz von Objekten gebunden ist.

Klassenvariable

Ein Attribut einer Klasse, dessen Deklaration mit dem Schlüsselwort `static` versehen ist, nennt man *Klassenvariable*. Klassenvariablen werden beim Laden der Klasse erzeugt.

Es existiert nur ein Exemplar dieser Variablen, unabhängig von der Anzahl der Instanzen der Klasse, und ihre Lebensdauer erstreckt sich über das ganze Programm. Der Zugriff von außen kann über den Klassennamen in der Form: `Klassenname.Variablenname` erfolgen.

Instanzvariable

Nicht statische Attribute werden zur Unterscheidung auch *Instanzvariablen* genannt.

Mit Klassenvariablen können beispielsweise Instanzenzähler realisiert werden, wie das folgende Beispiel zeigt.

Programm 3.4

```
// Datei: ZaehlerTest.java

public class ZaehlerTest {
   static int zaehler = 0;

   ZaehlerTest() {
      ++zaehler;
   }

   int getZaehler() {
      return zaehler;
   }
```

```
    public static void main(String[] args) {
        ZaehlerTest z1 = new ZaehlerTest();
        ZaehlerTest z2 = new ZaehlerTest();

        System.out.println(z1.getZaehler());
    }
}
```

Die Ausgabe des Programms ist: 2

statische Initialisierung

Klassenvariablen können neben der üblichen Zuweisung von Werten bei der Deklaration durch einen Block, der mit dem Schlüsselwort `static` eingeleitet wird, initialisiert werden. Eine Klasse kann mehrere Blöcke dieser Art haben. Sie werden in der aufgeschriebenen Reihenfolge beim Laden der Klasse ausgeführt.

Beispiel:

```
static int a;
static int b;
static {
    a = 2;
    b = 5 * a;
}
```

Klassenmethode

Eine Methode, deren Deklaration mit dem Schlüsselwort `static` versehen ist, nennt man *Klassenmethode*. Klassenmethoden können von außen über den Klassennamen aufgerufen werden. Man braucht keine Instanz der Klasse, um sie aufrufen zu können.

Instanzmethode

Nicht statische Methoden werden zur Unterscheidung *Instanzmethoden* genannt.

Klassenmethoden können nicht auf Instanzvariablen zugreifen und keine Instanzmethoden aufrufen.

main

Durch Hinzufügen einer Methode

```
public static void main(...)
```

wird eine beliebige Klasse zu einer *Java-Applikation*. Diese Klassenmethode ist der Startpunkt der Anwendung. Sie wird vom Java-Laufzeitsystem aufgerufen. `main` ist als statische

Methode deklariert, da zum Startzeitpunkt noch kein Objekt existiert. Der Parameter der Methode `main` wird an späterer Stelle erklärt.

Programm 3.5

```
// Datei: MaxTest.java

public class MaxTest {
   static int max(int a, int b) {
      return a < b ? b : a;
   }

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

Die Ausgabe des Programms ist: 5

3.5 Vererbung

Neue Klassen können auf Basis bereits vorhandener Klassen definiert werden. *Vererbung* ist der Mechanismus, der dies ermöglicht. Um eine neue Klasse aus einer bestehenden abzuleiten, wird im Kopf der Klassenbeschreibung das Schlüsselwort `extends` zusammen mit dem Namen der Klasse, von der abgeleitet wird, verwendet.

Superklasse
Subklasse

Die neue Klasse wird als *Subklasse* oder *abgeleitete Klasse*, die andere als *Superklasse* oder *Basisklasse* bezeichnet. Die Subklasse übernimmt automatisch alle Attribute und Methoden ihrer Superklasse (mit Ausnahme der Konstruktoren und der mit `private` gekennzeichneten Elemente), so dass diese Eigenschaften in der Subklasse nicht implementiert werden müssen. Die Subklasse kann natürlich eigene Attribute und Methoden besitzen und sogar übernommene Methoden der Superklasse überschreiben.

Überschreiben

Beim *Überschreiben* einer Methode bleiben ihre Signatur und ihr Rückgabetyp unverändert. Die Anweisungen im Methodenrumpf werden geändert, um so auf die Besonderheiten

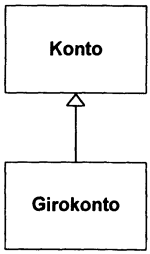
der Subklasse als Spezialisierung der Superklasse eingehen zu können.

Beispiel:

```
class Girokonto extends Konto {
   double limit;

   void setLimit(double limit) {
      this.limit = limit;
   }

   void zahleAus(double betrag) {
      if (betrag <= saldo + limit)
         saldo -= betrag;
   }
}
```

Hier erbt die Klasse `Girokonto` die Attribute und Methoden der Klasse `Konto`. Da aber beim Girokonto noch der Überziehungskredit geprüft werden muss, wird die Methode `zahleAus` neu implementiert. Beim Aufruf der Methode `zahleAus` für ein Objekt vom Typ `Girokonto` wird nun die überschriebene Methode verwendet.

Bild 3.3: Vererbungsbeziehung



super

Eine überschriebene Methode `m` der Subklasse verdeckt die ursprüngliche Methode `m` der Superklasse. Mit Hilfe des Schlüsselworts `super` kann die Superklassenmethode `m` aufgerufen werden: `super.m(...)`.

Konstruktoren

Konstruktoren werden nicht vererbt und können demzufolge nicht überschrieben werden. Innerhalb eines Konstruktors der Subklasse kann ein Konstruktor der Superklasse mittels `super(...)` als erste Anweisung aufgerufen werden.

Beispiel:

```
Girokonto(int kontonummer, double saldo, double limit) {
   super(kontonummer, saldo);
   this.limit = limit;
}
```

Fehlt der Aufruf von `super(...)` in einem Konstruktor der Subklasse, so setzt der Compiler automatisch den Aufruf `super()` ein. Fehlt der parameterlose Konstruktor der Superklasse, erzeugt der Compiler einen Fehler. Besitzt die Subklasse keinen expliziten Konstruktor, so erzeugt der Compiler automatisch einen parameterlosen Konstruktor, der lediglich den Aufruf des parameterlosen Konstruktors der Superklasse enthält.

Programm 3.6

```
// Datei: GirokontoTest.java

public class GirokontoTest {
   public static void main(String[] args) {
      Girokonto gk = new Girokonto(4711, 500., 2000.);

      System.out.println(gk.getSaldo());
      gk.zahleAus(3000.);
      System.out.println(gk.getSaldo());
      gk.setLimit(2500.);
      gk.zahleAus(3000.);
      System.out.println(gk.getSaldo());
   }
}
```

Die Ausgabe des Programms ist:

```
500.0
500.0
-2500.0
```

Object

Jede Klasse hat höchstens eine Superklasse (Einfachvererbung). Eine Klasse, die nicht explizit von einer anderen Klasse erbt, ist automatisch von der in Java definierten Klasse `Object` abgeleitet. `Object` selbst besitzt keine Superklasse und ist damit die Wurzel der Klassenhierarchie von Java.

Zuweisungskompatibilität

Eine Referenzvariable vom Typ einer Klasse kann jeder Variablen vom Typ einer in der Vererbungshierarchie übergeordneten Klasse zugewiesen werden (*Up-Cast*). Ein Objekt einer Subklasse ist insbesondere vom Typ der Superklasse.

Programm 3.7

```
// Datei: KompTest.java

public class KompTest {
   public static void main(String[] args) {
      Konto k;
      k = new Konto(4711, 5000.);
      k.zahleAus(1000.);                        // (1)
      System.out.println(k.getSaldo());

      k = new Girokonto(1020, 800., 2000.);
      k.zahleAus(3000.);                        // (2)
      System.out.println(k.getSaldo());

      ((Girokonto)k).setLimit(4000.);           // (3)
   }
}
```

Die Ausgabe des Programms ist:

```
4000.0
800.0
```

Programm 3.7 zeigt, dass die Variable `k` vom Typ `Konto` auf ein Objekt eines anderen Typs verweisen kann. Die Klasse `Konto` enthält die Methode `zahleAus`. In (1) wird `zahleAus` der Klasse `Konto` aufgerufen, in (2) wird jedoch die in der Subklasse `Girokonto` überschriebene Methode `zahleAus` aufgerufen. Es wird also stets die "richtige" Version der Methode verwendet.

(3) zeigt, dass eine explizite Typumwandlung (*Down-Cast*) erforderlich ist, um auf die Methode `setLimit` der Klasse `Girokonto` zugreifen zu können, da `setLimit` in der Klasse `Konto` nicht existiert.

instanceof

Der Operator `instanceof` kann verwendet werden, um festzustellen, zu welcher Klasse ein bestimmtes Objekt gehört:

`Referenzvariable` `instanceof` *`Klasse`*

Dieser Ausdruck hat den Wert `true`, wenn die Referenzvariable auf ein Objekt der Klasse *`Klasse`* oder auf ein Objekt einer in der Vererbungshierarchie ihr untergeordneten Klasse verweist.

Beispiel:

```
Girokonto k;
k = new Girokonto(1020, 800., 2000.);
```

Die Ausdrücke `(k instanceof Konto)` und `(k instanceof Girokonto)` haben beide den Wert `true`.

3.6 Abstrakte Klassen

Klassen, deren Beschreibung mit dem Schlüsselwort `abstract` beginnt, heißen *abstrakte Klassen*. Von solchen Klassen kann kein Objekt erzeugt werden. Sie dienen nur als Basisklasse bei der Vererbung.

In abstrakten Klassen dürfen auch so genannte *abstrakte Methoden* auftreten. Abstrakte Methoden besitzen keinen Anweisungsblock. Sie müssen in Subklassen überschrieben werden, andernfalls sind die abgeleiteten Klassen ebenfalls abstrakt und müssen mit dem Schlüsselwort `abstract` gekennzeichnet werden.

Den *Nutzen abstrakter Klassen* veranschaulicht das folgende Beispiel:

Programm 3.8

```
// Datei: FigurTest.java

public class FigurTest {
   public static void main(String[] args) {
      Figur[] f = new Figur[3];                    // (1)
      f[0] = new Kreis(10, 10, 5);
      f[1] = new Rechteck(0, 0, 20, 6);
      f[2] = new Kreis(100, 50, 12);

      for (int i = 0; i < 3; ++i) {                // (2)
         f[i].zeichne();
         System.out.println("Flaeche: " +
            f[i].getFlaeche());
      }
   }
}


// Datei: Figur.java

abstract class Figur {
   int x, y;

   Figur(int x, int y) {
      this.x = x;
      this.y = y;
   }

   abstract void zeichne();
   abstract double getFlaeche();
}


// Datei: Rechteck.java

class Rechteck extends Figur {
   int breite, hoehe;

   Rechteck(int x, int y, int breite, int hoehe) {
      super(x, y);
      this.breite = breite;
      this.hoehe = hoehe;
   }
```

```
   void zeichne() {
      System.out.println("Rechteck mit Bezugspunkt: ("
         + x + "," + y + "), Breite: " + breite
         + ", Hoehe: " + hoehe);
   }

   double getFlaeche() {
      return breite * hoehe;
   }
}


// Datei: Kreis.java

class Kreis extends Figur {
   int radius;

   Kreis(int x, int y, int radius) {
      super(x, y);
      this.radius = radius;
   }

   void zeichne() {
      System.out.println("Kreis mit Bezugspunkt: ("
         + x + "," + y + "), Radius: " + radius);
   }

   double getFlaeche() {
      return radius * radius * 3.14159;
   }
}
```

Die Ausgabe des Programms ist:

```
Kreis mit Bezugspunkt: (10,10), Radius: 5
Flaeche: 78.53975
Rechteck mit Bezugspunkt: (0,0), Breite: 20,
     Hoehe: 6
Flaeche: 120.0
Kreis mit Bezugspunkt: (100,50), Radius: 12
Flaeche: 452.38896
```

Die von der abstrakten Klasse `Figur` abgeleiteten Klassen `Rechteck` und `Kreis` überschreiben jeweils die abstrakten Methoden `zeichne` und `getFlaeche` durch eine konkrete Implementierung.

Anweisung (1) enthält die Erzeugung eines so genannten Arrays (eine Reihung von Elementen des gleichen Typs), das aus drei Elementen vom Typ `Figur` besteht. Der Zugriff auf die einzelnen Elemente erfolgt über einen numerischen Index. Arrays werden ausführlich in Kapitel 5.2 erklärt.

Der Aufruf von `zeichne` bzw. `getFlaeche` in der `for`-Schleife (2) führt immer die zum Typ des konkreten Objekts passende Methode aus.

Diese *einheitliche Behandlung* ist möglich, da eine Referenz vom Typ einer Subklasse einer Referenzvariablen vom Typ ihrer Superklasse zugewiesen werden kann und die hier in Frage kommenden Methoden bereits in der Superklasse vorkommen, wenn auch in abstrakter Form. Die Klasse `Figur` dient also als "Einstiegspunkt" für die Methodenaufrufe.

Polymorphismus

Die Eigenschaft, dass der formal gleiche Aufruf einer Methode unterschiedliche Reaktionen (je nach adressiertem Objekt) auslösen kann, wird als *Polymorphismus* (Vielgestaltigkeit) bezeichnet.

Polymorphismus setzt voraus, dass vor der Ausführung der Methode eine Verbindung zwischen dem Methodenaufruf und der zum konkreten Objekt passenden Methode hergestellt wird. Der Compiler muss Code generieren, um zur Laufzeit zu entscheiden, welche Version der überschriebenen Methoden aufgerufen werden soll (*dynamisches Binden*). Der Einsatz dieses Mechanismus vereinfacht die Programmentwicklung. So wirkt sich beispielsweise die Erweiterung des obigen Programmbeispiels um eine weitere Klasse (z.B. `Dreieck`) nicht auf den Schleifenkörper der `for`-Anweisung aus.

3.7 Modifizierer von Klassen, Attributen und Methoden

Mit Hilfe bestimmter Schlüsselwörter in Java, so genannten *Modifizierern*, können die Eigenschaften von Klassen, Attributen und Methoden verändert werden.

Die folgende Tabelle führt die Modifizierer auf und zeigt, bei welchen Programmelementen diese verwendet werden können.

Tabelle 3.1: Modifizierer im Überblick

Modifizierer	Klasse	Attribut	Methode	Konstruktor
`public`	x	x	x	x
`protected`		x	x	x
`private`		x	x	x
`static`		x	x	
`final`	x	x	x	
`abstract`	x		x	
`native`			x	
`synchronized`			x	
`transient`		x		

Die Modifizierer `abstract` und `static` wurden bereits erläutert, `synchronized` wird später im Zusammenhang mit Threads und `transient` im Zusammenhang mit der Serialisierung von Objekten behandelt..

Zugriffsrechte auf Klassen, Attribute und Methoden werden über die Modifizierer `public`, `protected` und `private` gesteuert. In diesem Zusammenhang ist der Begriff des *Pakets*, der später ausführlich behandelt wird, von Bedeutung. Ein Paket ist eine Sammlung von Klassen, die aus bestimmten Gründen zusammengefasst werden (z.B. weil sie alle zu einem bestimmten Anwendungsprojekt gehören). Jede Klasse gehört zu genau einem Paket.

public

Klassen, die mit `public` gekennzeichnet sind, können überall benutzt werden. In einer Quelldatei darf höchstens eine Klasse als `public` deklariert werden.

Auf Attribute, die als `public` vereinbart sind, kann von überall her lesend und schreibend zugegriffen werden.

Methoden und Konstruktoren, deren Deklaration mit `public` beginnt, können von allen Klassen benutzt werden.

protected

Auf Attribute, Methoden und Konstruktoren, die mit `protected` gekennzeichnet sind, können die eigene Klasse und alle anderen Klassen innerhalb desselben Pakets zugreifen. Derartige Attribute und Methoden werden an alle Subklassen weitervererbt und sind dann dort als geerbte Komponenten zugänglich.

Beim Überschreiben von Methoden dürfen die Zugriffsrechte nicht reduziert werden. Insbesondere kann also eine `public`-Methode nur `public` überschrieben werden. Eine `protected`-Methode kann `public` oder `protected` überschrieben werden.

private

Attribute, Methoden und Konstruktoren, die mit `private` gekennzeichnet sind, können nur in der sie deklarierenden Klasse benutzt werden. Attribute sollten immer als `private` deklariert werden und über Methoden zugänglich gemacht werden.

Will man die Objekterzeugung für eine Klasse verhindern, genügt es, nur einen einzigen parameterlosen Konstruktor mit dem Modifizierer `private` zu definieren.

Sind weder `public`, `protected` noch `private` angegeben, so können die Klassen, Attribute, Methoden und Konstruktoren in allen Klassen desselben Pakets benutzt werden.

final

Aus einer mit `final` gekennzeichneten Klasse sind keine Subklassen ableitbar.

Attribute mit dem Modifizierer `final` sind nicht veränderbar. Solche *Konstanten* müssen bei ihrer Deklaration initialisiert

werden oder es muss ihnen in jedem Konstruktor ein Wert zugewiesen werden.

Eine `final`-Methode darf in Subklassen nicht überschrieben werden.

native

Methoden, die mit `native` gekennzeichnet sind, werden wie abstrakte Methoden ohne Anweisungsblock deklariert. Die Implementierung erfolgt dann extern in einer anderen Programmiersprache (Java Native Interface).

3.8 Interfaces

Interfaces sind Schnittstellenbeschreibungen, die festlegen, was man mit der Schnittstelle machen kann, aber keine Funktionalität realisieren. Interfaces können nur Konstanten und abstrakte Methoden enthalten. Sie haben den folgenden Aufbau:

```
[public] interface Interfacename
   [extends Interface-Liste] {
   Konstanten
   abstrakte Methoden
}
```

Die in eckigen Klammern angegebenen Einheiten sind optional. Interfacenamen unterliegen den gleichen Namenskonventionen wie Klassennamen. Die Konstanten haben implizit die Modifizierer `public`, `static` und `final`. Die Methoden sind alle implizit `public` und `abstract`.

extends

Interfaces können im Gegensatz zu Klassen von mehreren anderen Interfaces abgeleitet werden. Damit können mehrere Schnittstellenbeschreibungen in einem Interface zusammengefasst werden. Die Interfacenamen werden nach `extends`, durch Kommata voneinander getrennt, aufgeführt.

implements

Klassen können ein oder mehrere Interfaces mit Hilfe des Schlüsselwortes `implements` (siehe Kapitel 3.1) implementieren. Mehrere Interfacenamen werden durch Kommata voneinander getrennt. Jede Klasse, die ein Interface implemen-

tiert, muss alle Methoden des Interface implementieren, ansonsten ist die Klasse abstrakt.

Interfaces erzwingen, dass verschiedene Klassen, die sonst keine Gemeinsamkeiten haben, die gleichen Methoden bereitstellen, ohne dass eine abstrakte Superklasse vereinbart werden muss. Sie fördern damit die Klarheit und Lesbarkeit des Programms.

Referenztyp

Ein Interface ist ein *Referenztyp*. Variablen können vom Typ eines Interface sein. Eine Klasse, die ein Interface implementiert, ist auch vom Typ des Interface.

Referenzen auf Interfaces können für polymorphe Methodenaufrufe genutzt werden (siehe Aufgabe 11).

Der Operator `instanceof` kann auch für Interfaces anstelle von Klassen genutzt werden.

Programm 3.9

```
// Datei: Bonus.java

public interface Bonus {
   double MIN_FUER_BONUS = 1000.;
   void addBonus(double betrag);
}


// Datei: Konto.java

public class Konto implements Bonus {
   private int kontonummer;
   private double saldo;

   public Konto() { }

   public Konto(int kontonummer) {
      this.kontonummer = kontonummer;
   }

   public Konto(int kontonummer, double saldo) {
      this.kontonummer = kontonummer;
      this.saldo = saldo;
   }
```

```
    public int getKontonummer() {
        return kontonummer;
    }

    public void setKontonummer(int nr) {
        kontonummer = nr;
    }

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double betrag) {
        saldo = betrag;
    }

    public void zahleEin(double betrag) {
        saldo += betrag;
    }

    public void zahleAus(double betrag) {
        saldo -= betrag;
    }

    public void addBonus(double betrag) {
        if (saldo >= MIN_FUER_BONUS)
            saldo += betrag;
    }
}

// Datei: InterfaceTest.java

public class InterfaceTest {
    public static void main(String[] args) {
        Konto k = new Konto(4711, 2000.);

        System.out.println(k.getSaldo());
        k.addBonus(20.);
        System.out.println(k.getSaldo());
    }
}
```

Die Ausgabe des Programms ist:

```
2000.0
2020.0
```

3.9 Innere Klassen

Innere Klassen sind Klassen, die *innerhalb* einer bestehenden Klasse vereinbart werden. Solche Klassen werden intensiv bei der Ereignisbehandlung im Rahmen der Entwicklung von grafischen Oberflächen verwendet. Wir behandeln hier zwei Varianten: *Instanzklassen* und *anonyme Klassen.*

Instanzklasse

Objekte von *Instanzklassen* können nur im Verbund mit einer Instanz der äußeren Klasse erzeugt werden. Sie haben vollen Zugriff auf alle Attribute und Methoden der sie umgebenden Klasse. Die Instanz der äußeren Klasse `A` kann durch `A.this` explizit referenziert werden.

Das folgende Beispielprogramm veranschaulicht die Verwendung einer Instanzklasse.

Programm 3.10

```
// Datei: InnerTest.java

public class InnerTest extends Quelle {
   public static void main(String[] args) {
      InnerTest t = new InnerTest();
      t.sendeEreignis(1);
      t.sendeEreignis(2);
   }

   public InnerTest() {
      registriereEmpfaenger(new MeinEmpfaenger());
   }

   class MeinEmpfaenger implements Empfaenger {
      public void reagiere(int e) {
         System.out.println("Ereignis " + e);
      }
   }
}
```

```
// Datei: Quelle.java

public class Quelle {
   private Empfaenger em;

   public void registriereEmpfaenger(Empfaenger em) {
      this.em = em;
   }

   public void sendeEreignis(int e) {
      if (em != null)
         em.reagiere(e);
   }
}

// Datei: Empfaenger.java

public interface Empfaenger {
   void reagiere(int e);
}
```

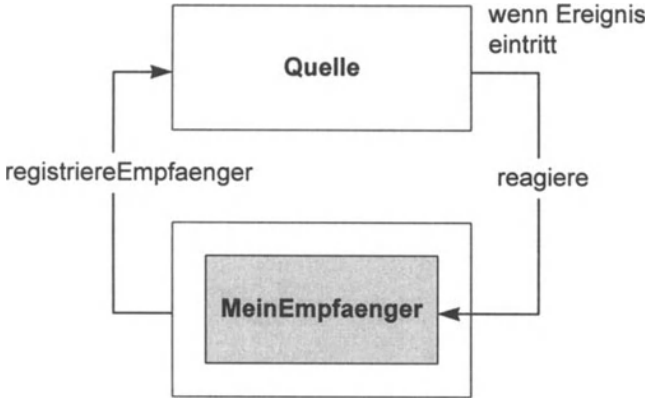
Die Ausgabe des Programms ist:

```
Ereignis 1
Ereignis 2
```

Erläuterungen zum Programm

Für ein Objekt der Klasse `Quelle` (die Ereignisquelle) können bestimmte Ereignisse (hier durch die Nummern 1 und 2 symbolisiert) ausgelöst werden. Die Reaktion auf diese Ereignisse erfolgt in einem Objekt der Klasse `MeinEmpfaenger` (dem Ereignisempfänger). Diese Klasse ist als Instanzklasse vereinbart und implementiert das Interface `Empfaenger`. Die Ereignisquelle sendet aber nur dann Ereignisse an den Ereignisempfänger, wenn sich dieser vorher bei der Ereignisquelle registriert hat. Die Registrierung erfolgt im Konstruktor der von `Quelle` abgeleiteten Klasse `InnerTest`. Die Methoden zur Registrierung und zum Senden von Ereignissen sind in der Klasse `Quelle` implementiert. Die Ereignisse werden in der Methode `main` ausgelöst.

Bild 3.4: Ereignisquelle und -empfänger



anonyme Klasse

Eine *anonyme Klasse* ist eine innere Klasse, die ohne Klassennamen in einer `new`-Anweisung eingesetzt wird. Sie wird aus einer anderen Klasse abgeleitet oder implementiert ein Interface, hat aber keine eigenen Konstruktoren.

```
new Konstruktor(...) { Klassenrumpf }
```

```
new Interface() { Klassenrumpf }
```

`Konstruktor` steht für einen Konstruktor der Superklasse, *`Interface`* für das im Klassenrumpf implementierte Interface. Der Rückgabewert ist vom Typ der Klasse des Konstruktors bzw. vom Typ des Interface.

Aus dem Klassenrumpf kann auf Attribute und Methoden der umgebenden Klasse sowie auf alle lokalen `final`-Variablen eines umgebenden Blocks bzw. Methodenrumpfs zugegriffen werden.

Anstelle der Instanzklasse im Programmbeispiel 3.10 kann eine anonyme Klasse verwendet werden.

Programm 3.11

```
// Datei: InnerTest.java

public class InnerTest extends Quelle {
   public static void main(String[] args) {
      InnerTest t = new InnerTest();
      t.sendeEreignis(1);
      t.sendeEreignis(2);
   }

   public InnerTest() {
      registriereEmpfaenger(new Empfaenger() {
         public void reagiere(int e) {
            System.out.println("Ereignis " + e);
         }
      } );
   }
}
```

3.10 Pakete

Eine größere Anwendung besteht aus einer Vielzahl von entwickelten Klassen und Interfaces, die aus Gründen der Zweckmäßigkeit (z.B. bessere Übersicht) zu größeren Einheiten, den *Paketen*, zusammengefasst werden.

Der *Name eines Pakets* besteht im Allgemeinen aus mehreren mit Punkt getrennten Teilen, z.B. `projekt1.gui`.

package

Um eine Klasse einem bestimmten Paket zuzuordnen, muss als erste Anweisung im Quellcode folgende Anweisung stehen:

`package` *`Paketname;`*

Der Paketname wird (beim SDK) auf das Dateiverzeichnis abgebildet, d.h. jedem Namensteil entspricht ein Unterverzeichnis.

Beispielsweise muss die `.class`-Datei zu einer Klasse mit der `package`-Klausel `package projekt1.gui;` im Verzeichnis `projekt1\gui` (bei Windows) liegen.

Um eine Klasse verwenden zu können, muss angegeben werden, in welchem Paket sie liegt. So kann die Klasse mit ihrem vollen Namen `Paket.Klasse` angesprochen werden.

import

Eine andere Möglichkeit ist, die gewünschte Klasse am Anfang der Quelldatei (hinter möglichen `package`-Anweisungen) bekannt zu machen:

```
import Paket.Klasse;
```

Im folgenden Quellcode kann dann die Klasse mit ihrem einfachen Namen angegeben werden, sofern es in allen anderen importierten Paketen keine Klasse mit dem gleichen Namen gibt.

Mit der folgenden Anweisung können alle Klassen eines Pakets importiert werden:

```
import Paket.*;
```

Eine Klasse wird vom Compiler im Paket erst dann gesucht, wenn sie im Programm benötigt wird.

Beispiel:

```
import projekt1.gui.Fenster;
```

`Fenster.class` ist im Verzeichnis `projekt1\gui` gespeichert. Wo dieses Verzeichnis im Dateisystem liegt, kann z.B. in der Umgebungsvariablen `CLASSPATH` angegeben werden:

```
set CLASSPATH=C:\java;C:\;.    (unter Windows)
```

Der Java-Compiler (wie auch der Java-Interpreter) sucht nach dem Unterverzeichnis `projekt1` in den Verzeichnissen `C:\java`, `C:\` und im aktuellen Verzeichnis.

Klassen, in denen die `package`-Anweisung fehlt, gehören automatisch zu einem Default-Paket. Sie können ohne explizite `import`-Anweisung gesucht werden. Das Default-Paket entspricht dem aktuellen Arbeitsverzeichnis.

Wichtige Pakete der Java-Entwicklungsumgebung sind z.B. `java.lang`, `java.io`, `java.util`, `java.awt` und `java.applet` und `javax.swing`.

`java.lang` enthält grundlegende Klassen und wird ohne explizite `import`-Anweisung automatisch importiert.

Programm 3.12

```
// Datei: KlasseA.java
package demo;

public class KlasseA {
   public void zeige() {
      System.out.println("Hallo");
   }
}

// Datei: PaketTest.java
import demo.*;

public class PaketTest {
   public static void main(String[] args) {
      KlasseA a = new KlasseA();
      a.zeige();
   }
}
```

Beispiel zur Übersetzung und Ausführung des Programms:

Das Verzeichnis `Test` enthalte das Unterverzeichnis `src` und das leere Unterverzeichnis `classes`. `KlasseA.java` soll in `src`, `PaketTest.java` in `Test` liegen. Das aktuelle Verzeichnis sei `Test`.

1. Übersetzung

   ```
   javac -d classes src\KlasseA.java
   ```

   `KlasseA.class` wird in `classes\demo` gespeichert.

   ```
   set CLASSPATH=classes
   javac PaketTest.java
   ```

   `PaketTest.class` wird im aktuellen Verzeichnis gespeichert.

   Alternativ kann auch eingegeben werden:

   ```
   javac -classpath classes PaketTest.java
   ```

2. Ausführung

```
set CLASSPATH=.;classes
java PaketTest
```

oder

```
java -cp .;classes PaketTest
```

Wird nach Schritt 1 eine Archivdatei `classes.zip` (oder `classes.jar`), die `demo\KlasseA.class` enthält, erzeugt und im aktuellen Verzeichnis gespeichert, so kann das Programm auch wie folgt aufgerufen werden:

```
java -cp .;classes.zip PaketTest
```

3.11 Aufgaben

1. Erläutern Sie das *Grundkonzept* der Objektorientierung.
2. Erstellen Sie eine Klasse `Abschreibung`, die den Anschaffungspreis, die Anzahl der Nutzungsjahre und den Abschreibungssatz enthält. Vereinbaren Sie zwei Konstruktoren und die beiden Abschreibungsmethoden: lineare und geometrisch-degressive Abschreibung (siehe auch Aufgabe 9 in Kapitel 2). Die Buchwerte für die einzelnen Jahre sollen am Bildschirm ausgegeben werden. Erstellen Sie auch eine Klasse mit der `main`-Methode zum Testen.
3. Wie kann eine Methode Daten von außen aufnehmen und nach außen weiterreichen?
4. Besitzt folgende Klasse einen *Standardkonstruktor*?

```
class A {
   int a;
   A(int i) {
      a = i;
   }
}
```

5. Erstellen Sie eine Klasse `Beleg`, deren Objekte automatisch eine bei der Zahl 10000 beginnende laufende Belegnummer erhalten.

6. Leiten Sie von der Klasse `Konto` (siehe Kapitel 3.1 und 3.3) eine neue Klasse ab, die die Auszahlung nur durchführt, wenn der Auszahlungsbetrag das Guthaben nicht überschreitet, und die eine neue Methode enthält, die Zinsen gutschreibt.

7. Was versteht man unter der *Signatur* einer Methode und worin besteht der Unterschied zwischen dem *Überladen* und dem *Überschreiben* einer Methode?

8. Ergänzen Sie das Programm 3.8 um die neue Figur "Rechtwinkliges Dreieck".

9. Erläutern Sie die Vorteile des *dynamischen Bindens* aus programmtechnischer Sicht.

10. Sind folgende Anweisungen korrekt? `K2` sei hier Subklasse von `K1`.

```
K1 p1 = new K1();
K2 p2 = new K2();
p1 = p2;
p2 = (K2)p1;
```

11. Entwickeln Sie eine Variante zu Programm 3.8, indem Sie statt einer abstrakten Basisklasse ein Interface benutzen, das die Klassen `Rechteck` und `Kreis` implementieren.

4 Ausnahmebehandlung

Während der Ausführung eines Programms können diverse Fehler bzw. *Ausnahmen* (*Exceptions*) auftreten, z.B. eine ganzzahlige Division durch 0, der Zugriff auf ein Arrayelement mit einem Index außerhalb der Grenzen des Arrays oder der Zugriff auf eine nicht vorhandene Datei.

In Java gibt es einen Mechanismus, solche Fehler in einer strukturierten Form zu behandeln. Dabei kann die Ausnahmebehandlung von dem Code der Methode, in der die Ausnahmebedingung auftritt, isoliert werden.

Ausnahme-Mechanismus

Das *Grundprinzip des Mechanismus* sieht wie folgt aus:

- Die Methode einer Klasse erkennt eine Ausnahmesituation, erzeugt ein Objekt einer Ausnahmeklasse und löst damit eine Ausnahme aus.
- Die Ausnahme kann entweder in derselben Methode abgefangen und behandelt werden, oder sie kann an die aufrufende Methode weitergereicht werden.
- Die Methode, an die die Ausnahme weitergereicht wurde, hat nun ihrerseits die Möglichkeit, sie entweder abzufangen und zu behandeln oder weiterzureichen.
- Wird die Ausnahme nur immer weitergereicht und in keiner Methode behandelt, bricht das Programm mit einer Fehlermeldung ab.

4.1 Ausnahmeklassen

Ausnahmen sind Objekte der Klasse `Throwable` oder ihrer Subklassen, die wie folgt gegliedert sind:

```
Throwable
. Error
. Exception
. . RunTimeException
```

Viele Pakete der Klassenbibliothek definieren ihre eigenen Ausnahmeklassen, die aus den aufgeführten Klassen abgeleitet sind.

Die Klasse `Error` und ihre Subklassen repräsentieren alle schwerwiegenden Fehler, die innerhalb des Laufzeitsystems auftreten und in der Regel von einem Anwendungsprogramm nicht behandelt werden können.

Exception

Die Klasse `Exception` und ihre Subklassen repräsentieren so genannte normale Fehler eines Programms. Dabei gibt es zwei Varianten: solche Ausnahmen, die vom Programm behandelt werden müssen (*kontrollierte Ausnahmen*) und solche, die vom Programm behandelt werden können, aber nicht müssen (*nicht kontrollierte Ausnahmen*). Die Klasse `RunTimeException` und ihre Subklassen repräsentieren die nicht kontrollierten Ausnahmen.

Konstruktoren der Klasse `Exception` sind:

```
Exception()
Exception(String meldung)
```

Die Klasse `String` repräsentiert Zeichenketten und wird im Kapitel 5.1 ausführlich behandelt.

Wichtige Methoden der Klasse `Throwable` sind:

`String getMessage()`
liefert die Fehlermeldung des Objekts.

`void printStackTrace()`
gibt Informationen zum Laufzeit-Stack aus.

Beispiel für eine selbst erstellte Ausnahmeklasse, die an späterer Stelle genutzt wird:

Beispiel

```
// Datei: RangAusnahme.java

public class RangAusnahme extends Exception {
   public RangAusnahme() { }

   public RangAusnahme(int rang) {
      super("Falscher Rang: " + rang);
   }
}
```

4.2 Auslösung und Weitergabe von Ausnahmen

Kontrollierte Ausnahmen müssen entweder abgefangen und behandelt oder an den Aufrufer weitergegeben werden (*catch or throw*). Der Compiler achtet auf die Einhaltung dieser Regel.

throw

Mit Hilfe der Anweisung

`throw` *`Ausnahmeobjekt;`*

wird eine Ausnahme ausgelöst. `throw` unterbricht das Programm an der aktuellen Stelle, um die Ausnahme zu behandeln oder die Weitergabe auszuführen.

throws

Wird die Ausnahme nicht in der sie auslösenden Methode behandelt, muss sie weitergereicht werden. Die Weitergabe geschieht mit Hilfe der `throws`-*Klausel* im Methodenkopf (siehe Kapitel 3.2):

`throws` *`Exception-Liste`*

`Exception-Liste` führt eine oder mehrere durch Kommata getrennte Ausnahmeklassen auf.

Programm 4.1

```
// Datei: Rang.java

public class Rang {
   private int persNummer;
   private int rang;

   public Rang(int nr, int r) throws RangAusnahme {
      if (r < 1 || r > 3)
```

```
            throw new RangAusnahme(r);
        persNummer = nr;
        rang = r;
    }

    public void zeige() {
        System.out.println("Personalnummer: " +
            persNummer + ", Rang: " + rang);
    }
}


// Datei: ThrowsTest.java

public class ThrowsTest {
    public static void main(String[] args)
        throws RangAusnahme {

        Rang r = new Rang(100456, 4);
        r.zeige();
    }
}
```

Die Ausgabe des Programms ist:

```
Exception in thread "main" RangAusnahme:
Falscher Rang: 4
      at Rang.<init>(Rang.java:9)
      at ThrowsTest.main(ThrowsTest.java:7)
```

Das Fehlen einer `throws`-Klausel in diesem Beispiel führt zu einer Fehlermeldung des Compilers.

4.3 Abfangen von Ausnahmen

try ... catch

Das Abfangen von Ausnahmen innerhalb einer Methode erfolgt mit Hilfe der Anweisung `try` :

```
try {
   Anweisungen
}
catch (Parameter) {
   Anweisungen
}
```

Der `try`-Block legt den Bereich fest, in dem die abzufangenden Ausnahmen auftreten können. *`Parameter`* besteht aus Typ (eine Subklasse von `Throwable`) und Bezeichner. Tritt in diesem Block eine Ausnahme auf, die zum Parameter der `catch`-Klausel passt, so fährt die Programmausführung mit der ersten Anweisung dieses `catch`-Blocks fort. Anweisungen im `try`-Block, die hinter der Anweisung liegen, die die Ausnahme verursacht hat, werden nicht mehr ausgeführt. Im `catch`-Block kann eine Fehlerbehebung oder eine Reaktion auf die Ausnahme codiert werden.

Es können mehrere `catch`-Blöcke codiert werden, die dann in der Reihenfolge ihres Auftretens abgearbeitet werden. Eine `catch`-Klausel fängt die Ausnahme ab, die zu ihrem Parameter zuweisungskompatibel ist. Hierauf folgende `catch`-Blöcke werden nicht durchlaufen. Ist kein passender `catch`-Block vorhanden, wird die aktuelle Methode beendet und die Ausnahme an die aufrufende Methode weitergereicht. Tritt im `try`-Block keine Ausnahme auf, wird mit der Anweisung hinter dem letzten `catch`-Block fortgesetzt.

finally

Hinter dem letzten `catch`-Block kann die `finally`-Klausel auftreten:

```
finally {
   Anweisungen
}
```

Der `finally`-Block enthält Anweisungen, die in jedem Fall ausgeführt werden sollen. Er wird durchlaufen,

- wenn der `try`-Block ohne Auftreten einer Ausnahme normal beendet wurde,
- wenn eine Ausnahme in einem `catch`-Block behandelt wurde,

- wenn eine aufgetretene Ausnahme in keinem `catch`-Block behandelt wurde,
- wenn der `try`-Block durch `break`, `continue` oder `return` verlassen wurde.

Die `try`-Anweisung kann auch ohne `catch`-Blöcke, aber dann mit einem `finally`-Block auftreten.

Programm 4.2

```
// Datei: TryTest.java

public class TryTest {
   public static void main(String[] args) {
      int persNummer = 100456;
      int rang = 4;
      Rang r = null;

      try {
         r = new Rang(persNummer, rang);
         System.out.println("Alles OK");
      }
      catch (RangAusnahme e) {
         System.out.println(e.getMessage());
         e.printStackTrace();
         try {
            r = new Rang(persNummer, 3);
         }
         catch (RangAusnahme a) { }
      }
      r.zeige();
   }
}
```

Die Ausgabe des Programms ist:

```
Falscher Rang: 4
RangAusnahme: Falscher Rang: 4
        at Rang.<init>(Rang.java:9)
        at TryTest.main(TryTest.java:10)
Personalnummer: 100456, Rang: 3
```

Hat die Variable `rang` den Wert 1, so ist die Ausgabe:

```
Alles OK
Personalnummer: 100456, Rang: 1
```

Programm 4.3

```
// Datei: FinallyTest.java
import java.io.*;

public class FinallyTest {
   public static void durchlaufe() throws Exception {
      int zahl;
      BufferedReader ein = new BufferedReader(
         new InputStreamReader(System.in)); // (*)
      Rang r;

      while (true) {
         System.out.print("Bitte Zahl eingeben: ");
         zahl = Integer.valueOf(ein.readLine()).
                  intValue(); // (*)
         try {
            if (zahl == 1)
               throw new RangAusnahme(0);
            if (zahl == 2)
               throw new Exception();
            if (zahl == 3)
               break;
         }
         catch (RangAusnahme e) {
            System.out.println(e.getMessage());
         }
         finally {
            System.out.println("finally durchlaufen");
         }
      }
      System.out.println("Ende der Schleife");
      ein.close(); // (*)
   }

   public static void main(String[] args)
      throws Exception {
      durchlaufe();
   }
}
```

Die mit (*) gekennzeichneten Konstrukte werden in späteren Kapiteln erklärt.

Ein-/Ausgabebeispiel:

```
Bitte Zahl eingeben: 0
finally durchlaufen

Bitte Zahl eingeben: 1
Falscher Rang: 0
finally durchlaufen

Bitte Zahl eingeben: 2
finally durchlaufen
Exception in thread "main"
java.lang.Exception
    at FinallyTest.durchlaufe(
         FinallyTest.java:19)
    at FinallyTest.main(FinallyTest.java:36)
```

Programmabbruch!

```
Bitte Zahl eingeben: 3
finally durchlaufen
Ende der Schleife
```

4.4 Aufgaben

1. Erläutern Sie den Mechanismus der *Ausnahmebehandlung*. Welche Schlüsselwörter sind mit den einzelnen Schritten verbunden?
2. Was besagt die *catch-or-throw-Regel*?
3. Fangen Sie die ganzzahlige Division durch 0 mit einer `try`-Anweisung ab (`ArithmeticException`).
4. In einer Schleife sollen über Tastatur ganze Zahlen eingegeben werden (siehe Programm 4.3). Eingabefehler, wie z.B. die Eingabe von Buchstaben, sollen dabei abgefangen werden (`NumberFormatException`). Bei

einer fehlerhaften Eingabe soll die Zahl erneut eingegeben werden.

5. Die ganzzahlige Variable `zufallszahl` kann in einer Schleife zufällige Werte annehmen. Es soll eine Ausnahme ausgelöst und abgefangen werden, wenn der Wert 0 ist. Zu diesem Zweck ist eine Ausnahmeklasse `IstNull` zu erstellen, die eine entsprechende Fehlermeldung speichert. Die Zufallszahl kann mittels `(int)(Math.random() * 100.)` erzeugt werden.

5 Ausgewählte Klassen

In diesem Kapitel werden einige wichtige Klassen aus den Java-Standardpaketen vorgestellt. Sie sind in jeder Java-Implementierung vorhanden und können in verschiedenen Anwendungssituationen nutzbringend eingesetzt werden.

5.1 Die Klassen String und StringBuffer

String

Die Klasse `String` repräsentiert Zeichenketten, bestehend aus Unicode-Zeichen. Objekte vom Typ `String` (im Folgenden kurz "Strings" genannt) sind nach der Initialisierung *nicht mehr veränderbar.* Alle Stringliterale (konstante Zeichenketten) werden als Objekte dieser Klasse implementiert. So zeigt z.B. die Referenzvariable `name` nach der Initialisierung durch `String name = "Schmitz"` auf das Objekt, das die konstante Zeichenkette `"Schmitz"` repräsentiert.

Konstruktoren:

```
String()
String(String s)
String(char[] c)
String(byte[] b)
String(byte[] b, int offset, int length)
```

Sie erzeugen eine leere Zeichenkette bzw. ein `String`-Objekt aus dem Inhalt des Parameters. Im letzten Fall werden `length` Bytes beginnend mit dem Index `offset` in eine Zeichenkette umgewandelt.

Im Folgenden werden die wichtigsten Methoden der Klasse `String` vorgestellt.

Länge

```
int length()
```

liefert die Länge der Zeichenkette.

Verketten

`String concat(String s)`
erzeugt einen neuen String, der den Inhalt von `s` an den des Strings, für den die Methode aufgerufen wurde, anhängt.

Strings können auch mit dem Operator + verkettet werden. Ein `String`-Objekt entsteht auch dann, wenn dieser Operator auf einen String und eine Variable oder ein Literal vom primitiven Datentyp angewandt wird.

Extrahieren

`char charAt(int i)`
liefert das Zeichen an der Position `i` der Zeichenkette. Das erste Zeichen hat die Position `0`.

`String substring(int begin)`
liefert einen String, der alle Zeichen des Strings, für den die Methode aufgerufen wurde, ab der Position `begin` enthält.

`String substring(int begin, int end)`
liefert einen String, der alle Zeichen ab der Position `begin` bis `end - 1` enthält. Die Länge ist also `end - begin`.

`String trim()`
schneidet alle zusammenhängenden Leer- und Steuerzeichen (das sind die Zeichen kleiner oder gleich `'\u0020'`) am Anfang und Ende der Zeichenkette ab.

Vergleichen

`boolean equals(Object obj)`
liefert `true`, wenn `obj` ein String ist und das aktuelle `String`-Objekt und `obj` die gleiche Zeichenkette repräsentieren.

`boolean equalsIgnoreCase(String s)`
wirkt wie `equals`, ignoriert aber eventuell vorhandene Unterschiede in der Groß- und Kleinschreibung.

`boolean startsWith(String s)`
liefert `true`, wenn die Zeichenkette im aktuellen String mit der Zeichenkette in `s` beginnt.

`boolean endsWith(String s)`
liefert `true`, wenn die Zeichenkette im aktuellen String mit der Zeichenkette in `s` endet.

```
boolean regionMatches(
  int i, String s, int j, int l)
```

liefert `true`, wenn der Teilstring des Strings, der ab der Position `i` beginnt und die Länge `l` hat, mit dem Teilstring von `s` übereinstimmt, der an der Position `j` beginnt und `l` Zeichen lang ist.

```
int compareTo(String s)
```

vergleicht die Zeichenkette mit der Zeichenkette in `s` lexikographisch und liefert `0`, wenn beide Zeichenketten gleich sind, einen negativen Wert, wenn die Zeichenkette kleiner ist als die in `s`, und einen positiven Wert, wenn die Zeichenkette größer ist als die in `s`.

Suchen

```
int indexOf(int c)
```

liefert die Position des ersten Auftretens des Zeichens `c` in der Zeichenkette, andernfalls wird `-1` zurückgegeben.

```
int indexOf(int c, int from)
```

wirkt wie die vorige Methode mit dem Unterschied, dass die Suche ab der Position `from` in der Zeichenkette beginnt.

```
int indexOf(String s)
```

liefert die Position des ersten Zeichens des ersten Auftretens der Zeichenkette `s` in dieser Zeichenkette, andernfalls wird `-1` zurückgegeben.

```
int indexOf(String s, int from)
```

wirkt wie die vorige Methode mit dem Unterschied, dass die Suche ab der Position `from` in der Zeichenkette beginnt.

Die folgenden vier Methoden suchen analog nach dem letzten Auftreten des Zeichens `c` bzw. der Zeichenkette `s`:

```
int lastIndexOf(int c)
int lastIndexOf(int c, int from)
int lastIndexOf(String s)
int lastIndexOf(String s, int from)
```

Ersetzen

```
String replace(char old, char new)
```

erzeugt einen String, in dem jedes Auftreten des Zeichens `old` in der aktuellen Zeichenkette durch das Zeichen `new` ersetzt ist.

`String toLowerCase()`
wandelt die Großbuchstaben der Zeichenkette in Kleinbuchstaben um und liefert diesen String dann zurück.

`String toUpperCase()`
wandelt die Kleinbuchstaben der Zeichenkette in Großbuchstaben um und liefert diesen String dann zurück.

Konvertieren

`static String valueOf(`*Typ* `x)`
wandelt x in einen String um. *Typ* steht für `boolean`, `char`, `int`, `long`, `float`, `double`, `char[]` und `Object`. Im letzten Fall wird `"null"` geliefert, falls das Objekt `null` ist, sonst wird `x.toString()` geliefert.

`byte[] getBytes()`
liefert ein Array von Bytes, das die Zeichen des Strings in der Standardcodierung enthält.

`String toString()`
gibt den String selbst zurück.

Diese Methode existiert auch in der Klasse `Object`. Mittels `toString` wird ein Objekt `obj` in eine für das Objekt charakteristische Zeichenkette umgewandelt. Z.B. führt der Aufruf von `System.out.println(obj)` zur Ausgabe von `obj.toString()`.

Das folgende Programm demonstriert einige der hier aufgeführten Methoden.

Programm 5.1

```
// Datei: StringTest.java

public class StringTest {
   public static void main(String[] args) {
      String nachname = "Schmitz";
      String vorname = new String("Hugo");
      String name;

      // Verketten
      name = nachname + ", " + vorname;
      name = name.concat(" Egon");
      System.out.println(name);
```

```
        // Laenge
        System.out.println(name.length());

        // Extrahieren
        System.out.println(name.substring(0, 13));

        // Vergleichen
        if (name.endsWith("Egon"))
           System.out.println("Name endet mit \"Egon\"");
        if (nachname.equals("Schmitz"))
           System.out.println("Nachname ist \"Schmitz\"");
        if (vorname.compareTo("Egon") > 0)
           System.out.println(
              "Vorname ist groesser \"Egon\"");

        // Suchen
        if (name.indexOf("Hugo") >= 0)
           System.out.println("Name enthaelt \"Hugo\"");

        // Ersetzen
        System.out.println(name.toLowerCase());

        // Konvertieren
        String s;
        s = String.valueOf(3.57);
        System.out.println(s.replace('.', ','));
    }
}
```

Die Ausgabe des Programms ist:

```
Schmitz, Hugo Egon
18
Schmitz, Hugo
Name endet mit "Egon"
Nachname ist "Schmitz"
Vorname ist groesser "Egon"
Name enthaelt "Hugo"
schmitz, hugo egon
3,57
```

StringBuffer

Objekte der Klasse `StringBuffer` enthalten veränderbare Zeichenketten. Bei Änderungen wird der benötigte Speicherplatz automatisch in der Größe angepasst.

Konstruktoren:

```
StringBuffer()
StringBuffer(int length)
StringBuffer(String s)
```

Sie erzeugen einen leeren Puffer, einen leeren Puffer der Größe `length` Zeichen bzw. einen Puffer mit dem Inhalt von `s`.

Methoden:

Länge

`int length()`
liefert die Länge der enthaltenen Zeichenkette.

Konvertierung in einen String

`String toString()`
wandelt die Zeichenkette im `StringBuffer`-Objekt in einen String.

`String substring(int begin)`
`String substring(int begin, int end)`
liefert einen String, der bei `begin` beginnt und die Zeichen bis zum Ende des Puffers bzw. bis `end - 1` enthält.

Einfügen und Löschen

`StringBuffer append(`*Typ* `x)`
hängt die Stringdarstellung von `x` an das Ende der Zeichenkette im aktuellen Objekt. *Typ* steht für `boolean`, `char`, `int`, `long`, `float`, `double`, `char[]`, `String` und `Object`.

`StringBuffer insert(int i, Typ x)`
fügt die Stringdarstellung von `x` an der Position `i` der Zeichenkette im aktuellen Objekt ein. *Typ* wie bei `append`.

`StringBuffer delete(int begin, int end)`
löscht den Teilstring ab der Position `begin` bis zur Position `end - 1`.

Die Methoden `append`, `insert` und `delete` verändern jeweils das Original und liefern es zusätzlich als Rückgabewert.

Zeichen lesen und ändern

`char charAt(int i)`
liefert das Zeichen an der Position `i`.

`void setCharAt(int i, char c)`
ersetzt das an der Position `i` stehende Zeichen durch `c`.

Programm 5.2

```
// Datei: StringBufferTest.java

public class StringBufferTest {
   public static void main(String[] args) {
      StringBuffer s = new StringBuffer("Ich");

      s.append("was soll es bedeuten ...");
      System.out.println(s);
      s.insert(3, " weiss nicht, ");
      System.out.println(s);

      int len = s.length();
      for (int i = 0; i < len; ++i) {
         if (s.charAt(i) == 'e')
            s.setCharAt(i, 'u');
         if (s.charAt(i) == 'E')
            s.setCharAt(i, 'U');
      }
      System.out.println(s);
   }
}
```

Die Ausgabe des Programms ist:

```
Ichwas soll es bedeuten ...
Ich weiss nicht, was soll es bedeuten ...
Ich wuiss nicht, was soll us buduutun ...
```

5.2 Arrays

Ein *Array* (Feld) ist eine Reihung von Elementen des gleichen Datentyps, die man unter einem gemeinsamen Namen ansprechen kann.

Deklaration

Die *Deklaration* eines Arrays erfolgt in der Form

```
Typ[] Arrayname;
```

Typ ist hierbei ein primitiver Datentyp, ein Klassen- oder ein Interfacename.

Erzeugung

Zur *Erzeugung* eines Arrays wird die new-Anweisung benutzt:

```
new Typ[Ausdruck]
```

Ausdruck legt die Größe des Arrays fest und muss einen ganzzahligen Wert liefern.

Beispiel:

```
int[] x;
x = new int[10];
```

Hierdurch wird ein Array x vom Typ `int` erzeugt, das `10` Zahlen aufnehmen kann.

Steht zum Zeitpunkt der Deklaration bereits fest, wie viele Elemente das Array aufnehmen soll, können Deklaration und Erzeugung zusammengefasst werden.

Beispiel:

```
int[] x = new int[10];
```

Initialisierung

Ein Array kann bei der Deklaration erzeugt und direkt mit Literalen initialisiert werden. Die Größe des Arrays ergibt sich aus der Anzahl der zugewiesenen Literale.

Beispiel:

```
int[] x = {1, 10, 4, 0};
```

x hat 4 Elemente.

Arrays sind Objekte

In Java wird ein Array wie ein normales Objekt behandelt. Die Array-Variable ist eine Referenzvariable, die auf dieses Objekt zeigt. Die Größe eines Arrays kann zur Laufzeit festgelegt werden. Sie kann dann aber nicht mehr verändert werden.

Länge

Die Anzahl der Elemente eines Arrays kann über das Attribut `length` abgefragt werden. Für das Array `x` im letzten Beispiel gilt: `x.length` hat den Wert `4`.

Zugriff auf Elemente

Die Elemente eines Arrays der Größe `n` werden von `0` bis `n-1` durchnummeriert. Der Zugriff auf ein Element erfolgt über seinen Index:

`Arrayname[Ausdruck]`

`Ausdruck` muss den Rückgabetyp `int` haben.

Die Einhaltung der Array-Grenzen wird vom Laufzeitsystem geprüft. Bei Überschreiten der Grenzen wird die Ausnahme `ArrayIndexOutOfBoundsException` ausgelöst.

Programm 5.3

```
// Datei: ArrayTest.java

public class ArrayTest {
   public static void main(String[] args) {
      int[] zufall = new int[10];

      for (int i = 0; i < zufall.length; ++i)
         zufall[i] = (int)(Math.random() * 100.);

      for (int i = 0; i < zufall.length; ++i)
         System.out.print(zufall[i] + " ");
      System.out.println();

      String[] tage =
         {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};

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

mehrdimensionale Arrays

Mehrdimensionale Arrays werden als geschachtelte Arrays angelegt. Referenzvariablen für solche Arrays werden erzeugt, indem mehrere Klammernpaare hintereinander angegeben

werden. Der Zugriff auf ein Element erfolgt durch Angabe aller erforderlichen Indizes.

Auch eine Initialisierung mit Literalen ist möglich. Werte einer Dimension werden durch geschweifte Klammern zusammengefasst.

Beispiel:

```
int[][] x = new int[2][3];
```

erzeugt eine 2x3-Matrix.

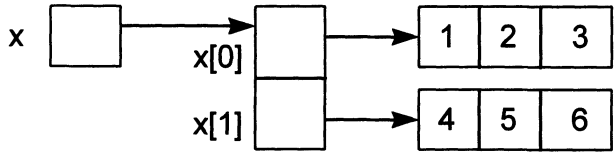
Diese Anweisung hat dieselbe Wirkung wie:

```
int[][] x;
x = new int[2][];
for (int i = 0; i < 2; i++)
   x[i] = new int[3];
```

Alternativ mit Initialisierung:

```
int[][] x = {{1, 2, 3}, {4, 5, 6}};
```

Bild 5.1: 2x3-Matrix



Bei der Initialisierung können für jedes Element einer Dimension unterschiedlich viele Elemente initialisiert werden. Das folgende Programm zeigt, wie "nicht rechteckige" Arrays erzeugt werden können.

Programm 5.4

```
// Datei: NichtRechteck.java

public class  NichtRechteck {
   public static void main(String[] args) {
      int[][] x = {{1}, {1,2}, {1,2,3}, {1,2,3,4}};
```

```
        for (int i = 0; i < x.length; ++i) {
            for (int j = 0; j < x[i].length; ++j)
                System.out.print(x[i][j]);
            System.out.println();
        }
    }
}
```

Die Ausgabe des Programms ist:

```
1
12
123
1234
```

Kommando-zeilen-Parameter

Beim Aufruf einer *Java-Applikation* kann man dem Programm in der Kommandozeile Parameter mitgeben:

`java` *`Programm param1 param2 ...`*

Diese *Kommandozeilen-Parameter* werden der Methode `main` übergeben:

```
public static void main(String[] args)
```

`args` ist ein Array vom Typ `String` und enthält die Parameter der Kommandozeile *`param1`*, *`param2`* usw. als Elemente.

Das folgende Programm gibt die Kommandozeilen-Parameter auf dem Bildschirm aus.

Programm 5.5

```
// Datei: Kommandozeile.java

public class  Kommandozeile {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; ++i)
            System.out.println((i+1) + ". Parameter: "
                + args[i]);
    }
}
```

Der Aufruf
`java Kommandozeile Dies ist ein "T e s t"`
liefert die Ausgabe:

```
1. Parameter: Dies
2. Parameter: ist
3. Parameter: ein
4. Parameter: T e s t
```

Die Klasse Arrays

Die Klasse `Arrays` aus dem Paket `java.util` bietet Methoden zum komfortablen Arbeiten mit Arrays.

`static void fill(`*Typ*`[] a, `*Typ*` val)`
weist allen Elementen in `a` den Wert `val` zu. Als *Typ* kann ein primitiver Datentyp oder `String` eingesetzt werden.

`static void fill(`*Typ*`[] a, int from, int to, `*Typ*` val)`
weist allen Elementen in `a` ab Index `from` bis zum Index `to - 1` den Wert `val` zu. Als *Typ* kann ein primitiver Datentyp oder `String` eingesetzt werden.

`static void sort(`*Typ*`[] a)`
sortiert die Elemente des Arrays `a` aufsteigend. *Typ* steht für einen der Datentypen `byte`, `char`, `short`, `int`, `long`, `float`, `double` oder `String`.

`static int binarySearch(`*Typ*`[] a, `*Typ*` key)`
durchsucht das Array `a` nach dem Wert `key` unter Verwendung des Verfahrens der binären Suche. Dazu müssen die Elemente in `a` aufsteigend sortiert sein. *Typ* steht für einen der Datentypen `byte`, `char`, `short`, `int`, `long`, `float`, `double` oder `String`. Wird der Wert `key` gefunden, so wird sein Index zurückgegeben. Andernfalls ist der Rückgabewert negativ und zwar `-i-1`, wobei `i` der Index des ersten größeren Werts bzw. `a.length` ist, wenn alle Elemente in `a` kleiner als `key` sind.

Bemerkung: Obige Methoden existieren auch für allgemeine Objekte als Elemente eines Arrays vom Typ `Object`. Zur Anwendung der Sortier- und der Suchmethode müssen alle Elemente das Interface `Comparable` mit der Methode

`int compareTo(Object obj)` implementieren und paarweise vergleichbar sein.

Programm 5.6

```
// Datei: ArraysTest.java
import java.util.*;

public class  ArraysTest {
   public static void main(String[] args) {
      int[] x = new int[8];
      double[] y = {5., 1.3, 3.75, 2., 10.};
      char[] z = {'c', 'a', 'f', 'u', 'm'};
      int a;

      Arrays.fill(x, 2);
      Arrays.fill(x, 4, 6, 3);
      for (int i = 0; i < x.length; ++i)
         System.out.print(x[i] + " ");
      System.out.println();

      Arrays.sort(y);
      for (int i = 0; i < y.length; ++i)
         System.out.print(y[i] + " ");
      System.out.println();

      a = Arrays.binarySearch(y, 3.75);
      System.out.println(a);

      a = Arrays.binarySearch(y, 3.74);
      System.out.println(a);

      Arrays.sort(z);
      for (int i = 0; i < z.length; ++i)
         System.out.print(z[i] + " ");

      System.out.println();

      a = Arrays.binarySearch(z, 'f');
      System.out.println(a);

      a = Arrays.binarySearch(z, 'z');
      System.out.println(a);
   }
}
```

Die Ausgabe des Programms ist:

```
2 2 2 2 3 3 2 2
1.3 2.0 3.75 5.0 10.0
2
-3
a c f m u
2
-6
```

5.3 Die Klasse StringTokenizer

Die Klasse `StringTokenizer` aus dem Paket `java.util` ist ein nützliches Werkzeug zur Bearbeitung von Texten. Ein Text in Form einer Zeichenkette wird als Mischung von Textteilen (*Tokens*), die aus zusammenhängenden Zeichen bestehen, und besonderen Zeichen (z.B. Leerzeichen, Interpunktionszeichen), die die Textteile voneinander trennen (*Trennzeichen*), angesehen.

Die Leistung der Klasse ist nun, den Text mittels spezieller Methoden in einzelne Tokens zu zerlegen. Dabei können die Zeichen, die als Trennzeichen dienen sollen, vorgegeben werden.

Beispiel:
Trennzeichen seien Leerzeichen, Komma und Punkt. Die Tokens der Zeichenkette `"Ich weiß nicht, was soll es bedeuten"` sind dann: `Ich`, `weiß`, `nicht`, `was`, `soll`, `es` und `bedeuten`.

Konstruktoren

```
StringTokenizer(String s)
StringTokenizer(String s, String delim)
StringTokenizer(String s, String delim,
  boolean returnTokens)
```

`s` ist die Zeichenkette, die zerlegt werden soll. `delim` enthält die Trennzeichen. Ist `delim` nicht angegeben, so wird `" \t\n\r\f"` benutzt. Hat `returnTokens` den Wert `true`, so werden auch die Trennzeichen als Tokens angesehen.

Methoden

`int countTokens()`
liefert die Anzahl der noch verbleibenden Tokens.

`boolean hasMoreTokens()`
liefert den Wert `true`, wenn noch mindestens ein weiteres Token vorhanden ist.

`String nextToken()`
liefert das nächste Token.

`String nextToken(String delim)`
liefert das nächste Token, wobei jetzt und für die nächsten Zugriffe die Zeichen in `delim` als Trennzeichen gelten.

Programm 5.7

```
// Datei: TextZerlegen.java
import java.util.*;
import java.io.*;

public class  TextZerlegen {
   public static void main(String[] args) throws
      IOException {
      BufferedReader ein = new BufferedReader(
         new InputStreamReader(System.in)); // (*)
      String zeile;
      StringTokenizer st;
      String delim = " /:.*;,{}[]()=!\t\"\\";

      while ((zeile = ein.readLine()) != null) { // (*)
         st = new StringTokenizer(zeile, delim);
         while (st.hasMoreTokens())
            System.out.println(st.nextToken());
      }
      ein.close(); // (*)
   }
}
```

Das Programm liest Zeilen ein und zerlegt diese in Tokens. Die mit `(*)` gekennzeichneten Konstrukte werden an späterer Stelle erklärt. Durch Aufruf des Programms in der Form

```
java TextZerlegen < TextZerlegen.java
```

wird die Datei `TextZerlegen.java` zeilenweise gelesen. Die Ausgabe beginnt mit:

```
Datei
TextZerlegen
java
import
java
util
import
java
io
public
class
...
```

5.4 Hüllklassen

Zu jedem primitiven Datentyp gibt es eine so genannte *Hüllklasse* (*Wrapper-Klasse*), deren Objekte einer Variablen dieses Datentyps entsprechen.

Hüllklassen bieten eine Reihe von Methoden für den Zugriff auf die "umhüllten" Variablen. So können z.B. Zeichenketten in Werte des entsprechenden primitiven Datentyps umgewandelt werden und umgekehrt.

Tabelle 5.1: Hüllklassen

Primitiver Datentyp	Hüllklasse
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Konstruktoren

Zur Erzeugung von "Hüllobjekten" kann dem Konstruktor ein Literal bzw. eine Variable vom primitiven Datentyp oder ein String mit entsprechendem Inhalt übergeben werden. Bei der Übergabe von inkompatiblen Strings wird im Fall der numerischen Hüllklassen die Ausnahme `NumberFormatException` ausgelöst.

```
Boolean(boolean val)
Boolean(String s)
```

```
Character(char val)
```

```
Byte(byte val)
Byte(String s)
```

```
Short(short val)
Short(String s)
```

```
Integer(int val)
Integer(String s)
```

```
Long(long val)
Long(String s)
```

```
Float(float val)
Float(double val)
Float(String s)
```

```
Double(double val)
Double(String s)
```

wichtige Methoden

Für alle Hüllklassen gibt es die Methoden `equals` und `toString`:

`boolean equals(Object obj)`
vergleicht das Hüllobjekt mit `obj` und gibt `true` zurück, falls `obj` vom Typ der Hüllklasse ist und denselben Wert hat.

`String toString()`
liefert den Wert des Hüllobjekts als Zeichenkette.

Die einzelnen Klassenmethoden
`static` *Typ* `valueOf(String s)`
erzeugen aus dem von `s` repräsentierten Wert ein Objekt der entsprechenden Hüllklasse und liefern es als Rückgabewert.

Für *Typ* kann `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float` und `Double` eingesetzt werden.

Für die Hüllklassen `Byte`, `Short`, `Integer` und `Long` gibt es noch die Variante:

`static Typ valueOf(String s, int base)`
`base` bezeichnet die Basis des Zahlensystems, das bei der Umwandlung zu Grunde gelegt wird.

Beispiel: `Integer.valueOf("101", 2)` erzeugt ein Hüllobjekt, das die ganze Zahl 5 repräsentiert.

Die folgenden Methoden liefern den durch das Objekt der jeweiligen Hüllklasse repräsentierten Wert:

```
boolean booleanValue()
char charValue()
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
```

Beispiel:

```
double x = Double.valueOf("123.75").doubleValue();
```

Hier wird aus dem String `"123.75"` zunächst ein Objekt vom Typ `Double` erzeugt und zurückgeliefert, dann der entsprechende `double`-Wert zugewiesen.

numerische Konstanten

In jeder numerischen Hüllklasse gibt es die Konstanten `MIN_VALUE` und `MAX_VALUE`, die den kleinsten bzw. größten Wert des Wertebereichs des entsprechenden Datentyps darstellen:

```
static final Typ MIN_VALUE
static final Typ MAX_VALUE
```

Für *Typ* kann `byte`, `short`, `integer`, `long`, `float` oder `double` eingesetzt werden.

Die Klassen `Float` und `Double` haben zusätzlich die Konstanten `NaN` (Not a Number), `NEGATIVE_INFINITY` und `POSITIVE_INFINITY`. `NaN` stellt einen undefinierten Wert dar, wie er bei der Division `0.0/0.0` entsteht. `NEGATIVE_INFINITY` bzw. `POSITIVE_INFINITY` entsteht bei der Division negativer bzw. positiver Zahlen durch `0`.

Programm 5.8

```
// Datei: HuellTest.java

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

   // Zeichenkette in Zahl umwandeln
      Integer intObj;
      String s = "10500";
      int x;
      intObj = Integer.valueOf(s);
      x = intObj.intValue();
      System.out.println("\"" + s + "\" -> " + x);

      // oder kuerzer
      x = Integer.valueOf(s).intValue();
      System.out.println("\"" + s + "\" -> " + x);

   // Zahl in Zeichenkette umwandeln
      double y = 123.76;
      Double doubleObj = new Double(y);
      s = doubleObj.toString();
      System.out.println(y + " -> \"" + s + "\"");

      // oder kuerzer
      s = String.valueOf(y);
      System.out.println(y + " -> \"" + s + "\"");

   // Konstanten
      System.out.println("byte Min: " + Byte.MIN_VALUE);
      System.out.println("byte Max: " + Byte.MAX_VALUE);
      System.out.println("short Min: " + Short.MIN_VALUE);
      System.out.println("short Max: " + Short.MAX_VALUE);
```

```
        System.out.println("int Min: " + Integer.MIN_VALUE);
        System.out.println("int Max: " + Integer.MAX_VALUE);
        System.out.println("long Min: " + Long.MIN_VALUE);
        System.out.println("long Max: " + Long.MAX_VALUE);
        System.out.println("float Min: " + Float.MIN_VALUE);
        System.out.println("float Max: " + Float.MAX_VALUE);
        System.out.println("double Min: " +
           Double.MIN_VALUE);
        System.out.println("double Max: " +
           Double.MAX_VALUE);

        System.out.println(0./0.);
        System.out.println(1./0.);
        System.out.println(-1./0.);
    }
}
```

Die Ausgabe des Programms ist:

```
"10500" -> 10500
"10500" -> 10500
123.76 -> "123.76"
123.76 -> "123.76"
byte Min: -128
byte Max: 127
short Min: -32768
short Max: 32767
int Min: -2147483648
int Max: 2147483647
long Min: -9223372036854775808
long Max: 9223372036854775807
float Min: 1.4E-45
float Max: 3.4028235E38
double Min: 4.9E-324
double Max: 1.7976931348623157E308
NaN
Infinity
-Infinity
```

5.5 Mathematische Funktionen

Die Klasse `Math` enthält die beiden `double`-Konstanten `E` (Eulersche Zahl e) und `PI` (Kreiskonstante π) sowie grundlegende *mathematische Funktionen*, die als Klassenmethoden realisiert sind. Bei unzulässigen Operationen wird die Ausnahme `ArithmeticException` ausgelöst.

Tabelle 5.2: Mathematische Funktionen

Methode	Erläuterung
Typ `abs(`*Typ* `x)`	Absolutbetrag von x. *Typ* steht für `int`, `long`, `float`, `double`.
Typ `min(`*Typ* `x,` *Typ* `y)`	Minimum von x und y. *Typ* steht für `int`, `long`, `float`, `double`.
Typ `max(`*Typ* `x,` *Typ* `y)`	Maximum von x und y. *Typ* steht für `int`, `long`, `float`, `double`.
`double ceil(double x)`	liefert die kleinste ganze Zahl größer oder gleich x
`double floor(double x)`	liefert die größte ganze Zahl kleiner oder gleich x
`int round(`*Typ* `x)`	liefert die ganze Zahl, die entsteht, wenn die Nachkommastellen von `x+0.5` abgeschnitten werden. Typ steht für `float`, `double`.
`double sqrt(double x)`	Quadratwurzel von x
`double pow(double x, double y)`	Potenz x^y
`double exp(double x)`	e^x
`double log(double x)`	Natürlicher Logarithmus von x
`double sin(double x)`	Sinus von x

`double cos(double x)`	Cosinus von x
`double tan(double x)`	Tangens von x
`double asin(double x)`	Arcussinus von x
`double acos(double x)`	Arcuscosinus von x
`double atan(double x)`	Arcustangens von x
`double random()`	erzeugt gleichverteilte Zufallszahlen >= 0 und < 1

Programm 5.9

```
// Datei: MathTest.java

public class  MathTest {
   public static void main(String[] args) {
      double x = 3.5, y = 2.;
      double abstand, radius = 2., flaeche;
      double anfangsBetrag = 5000., n = 10.;
      double zinssatz = 7.5, endBetrag;

      abstand = Math.sqrt(Math.pow(x,2) + Math.pow(y,2));
      System.out.println("Abstand: " + abstand);
      flaeche = Math.PI * Math.pow(radius,2);
      System.out.println("Flaeche: " + flaeche);
      endBetrag = anfangsBetrag *
         Math.pow(1. + zinssatz / 100., n);
      System.out.println("Endbetrag: " + endBetrag);
      System.out.println(Math.ceil(3.6));
      System.out.println(Math.floor(3.6));
      System.out.println(Math.round(3.6));
   }
}
```

Die Ausgabe des Programms ist:

```
Abstand: 4.031128874149275
Flaeche: 12.566370614359172
Endbetrag: 10305.157810823555
4.0
3.0
4
```

DecimalFormat

Die Klasse `DecimalFormat` aus `java.text` dient zur Darstellung und Umwandlung von Dezimalzahlen.

`DecimalFormat(String pattern)`
erzeugt ein Objekt mit dem durch `pattern` vorgegebenen Format.

Der String `pattern` kann sich aus folgenden Zeichen zusammensetzen:

- `#` Ziffer, führende Nullen werden nicht angezeigt
- `0` Ziffer, führende Nullen werden als 0 angezeigt
- `.` Dezimalpunkt
- `,` Symbol für Tausendergruppierung
- `%` Darstellung als Prozentzahl (nach Multiplikation mit 100)

Als Präfix und Suffix des darzustellenden Wertes können beliebige Zeichen auftreten.

`String format(double x)`
formatiert x gemäß der Vorgabe.

Beispiel:

```
DecimalFormat f = new DecimalFormat("###,##0.00");
String s = f.format(24522.4567);
```

s enthält die Zeichenkette `"24.522,46"`.

Zufallszahlen

Mit der Klasse `Random` aus dem Paket `java.util` können *Zufallszahlen* erzeugt werden.

Konstruktoren sind:

`Random()`
`Random(long seed)`
`seed` liefert die Startbedingung für die Erzeugung der Zufallszahlen. Zwei Objekte, die mit dem gleichen Wert `seed` erzeugt werden, generieren gleiche Folgen von Zufallszahlen.

Wird `seed` nicht übergeben, wird der Zufallszahlengenerator auf Basis der aktuellen Systemzeit initialisiert.

`void setSeed(long seed)`
ändert den Startwert des Zufallszahlengenerators.

Die folgenden Methoden liefern gleichverteilte Zufallszahlen vom Typ `int`, `long`, `float` und `double`:

```
int nextInt()
long nextLong()
float nextFloat()
double nextDouble()
```

In den beiden letzten Fällen sind die Zufallszahlen >= 0 und < 1.

`double nextGaussian()`
liefert eine normalverteilte Zufallszahl mit dem Mittelwert 0 und der Standardabweichung 1.

Programm 5.10

```
// Datei: RandomTest.java
import java.util.*;

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

      // Zufallszahlen zwischen 0 und 99

      Random rand = new Random(1);
      for (int i = 0; i < 10; ++i)
         System.out.print(
            Math.abs(rand.nextInt()) % 100 + " ");
      System.out.println();

      rand = new Random(2);
      for (int i = 0; i < 10; ++i)
         System.out.print(
            Math.abs(rand.nextInt()) % 100 + " ");
      System.out.println();
```

```
        rand = new Random(1);
        for (int i = 0; i < 10; ++i)
           System.out.print(
              (int)(rand.nextDouble() * 100.) + " ");
        System.out.println();

        rand = new Random();
        for (int i = 0; i < 10; ++i)
           System.out.print(
              (int)(rand.nextDouble() * 100.) + " ");
        System.out.println();
     }
}
```

Die Ausgabe des Programms ist:

```
25 76 95 26 8 8 69 83 39 0
79 44 16 35 78 95 84 38 1 59
73 41 20 33 96 0 96 93 94 93
87 18 26 90 34 20 17 16 85 35
```

Die letzte Reihe von Zufallszahlen ist nicht reproduzierbar.

5.6 Listen

Eine *Liste* ist eine Folge von Elementen, in der jedes Element, außer dem letzten, genau einen Nachfolger und jedes, außer dem ersten, genau einen Vorgänger hat. Grundlegende Listenoperationen sind der Aufbau der Liste, das Durchlaufen der Liste, das Einfügen und das Entfernen von Listenelementen.

In diesem Kapitel werden als Repräsentation einer Liste die Klassen `Vector` und `Stack` vorgestellt, ferner das Interface `Enumeration` und als Anwendungsbeispiel so genannte *verkettete Listen.*

Vector

Die Klasse `Vector` aus dem Paket `java.util` implementiert eine dynamisch wachsende Reihung von Elementen des Typs `Object` (im Folgenden kurz "Vektor" genannt), auf die über einen Index zugegriffen werden kann.

Jeder der folgenden *Konstruktoren* erzeugt ein Objekt ohne Elemente. Der interne Speicherplatz wird zunächst für eine bestimmte Anzahl von Elementen bereitgestellt. Die Anfangskapazität bzw. der Zuwachswert, um den im Bedarfsfall die Kapazität erhöht werden soll, kann vorgegeben werden, sonst werden Standardwerte angenommen.

```
Vector()
Vector(int kapazitaet)
Vector(int kapazitaet, int zuwachs)
```

Wichtige Methoden:

`int size()`
liefert die aktuelle Größe des Vektors.

`boolean isEmpty()`
liefert `true`, falls der Vektor kein Element enthält.

Einfügen

`void addElement(Object obj)`
hängt `obj` an das Ende des Vektors.

`void insertElementAt(Object obj, int i)`
fügt `obj` an der Position `i` in den Vektor ein. Das erste Element des Vektors hat die Position `0`. Die Elemente, die sich bisher an dieser bzw. einer dahinter liegenden Position befanden, werden um eine Position zum Ende des Vektors hin verschoben.

Entfernen

`void removeElementAt(int i)`
entfernt das Element an der Position `i`. Die folgenden Elemente werden um eine Position nach vorne verschoben.

`boolean removeElement(Object obj)`
entfernt das erste Element `obj` und liefert `true`. Die folgenden Elemente werden um eine Position nach vorne verschoben. Enthält der Vektor kein Element `obj`, wird `false` zurückgegeben..

`void removeAllElements()`
entfernt alle Elemente aus dem Vektor.

Ersetzen

`void setElementAt(Object obj, int i)`
ersetzt das Element an der Position `i` durch `obj`.

Zugriff auf Elemente

`Object firstElement()`
liefert das erste Element des Vektors.

`Object lastElement()`
liefert das letzte Element des Vektors.

`Object elementAt(int i)`
liefert das Element an der Position `i`.

Enumeration

Das Interface `Enumeration` wird verwendet, um alle Elemente einer Aufzählung nacheinander zu durchlaufen. Die Schnittstelle deklariert die beiden folgenden abstrakten Methoden:

`boolean hasMoreElements()`
liefert `true`, wenn noch weitere Elemente vorliegen.

`Object nextElement()`
liefert das nächste Element.

Die Klasse `Vector` stellt diese Methoden bereit:

`Enumeration elements()`
erzeugt ein Objekt vom Typ `Enumeration` für alle Elemente des Vektors.

Programm 5.11

```
// Datei: VectorTest.java
import java.util.*;

public class  VectorTest {
   public static void main(String[] args) {
      Vector v = new Vector();

      v.addElement("Die erste Zeichenkette");
      v.addElement("Die zweite Zeichenkette");
      v.addElement(new Integer(4711));
      v.insertElementAt("Die dritte Zeichenkette", 2);
      v.addElement(new Integer(4712));

      Enumeration enum = v.elements();
```

```
        while (enum.hasMoreElements()) {
            Object obj = enum.nextElement();
            System.out.println(obj);
        }
    }
}
```

Die Ausgabe des Programms ist:

```
Die erste Zeichenkette
Die zweite Zeichenkette
Die dritte Zeichenkette
4711
4712
```

Stack

Als *Stack* (Stapel) wird eine Liste bezeichnet, die nach dem LIFO-Prinzip (last in, first out) arbeitet. Elemente können in der umgekehrten Reihenfolge entnommen werden, wie sie eingetragen wurden.

Die Klasse `Stack` aus dem Paket `java.util` repräsentiert einen solchen Stapel. Sie ist von der Klasse `Vector` abgeleitet.

Der Konstruktor `Stack()` liefert einen leeren Stapel.

`boolean empty()`
liefert den Wert `true`, wenn der Stapel leer ist.

`Object push(Object obj)`
legt das Objekt `obj` auf den Stapel und liefert es als Rückgabewert.

`Object pop()`
liefert das oberste Element des Stapels und entfernt es.

`Object peek()`
liefert das oberste Element des Stapels, ohne es zu entfernen.

`int search(Object obj)`
sucht nach dem Objekt `obj` im Stapel und liefert die Position, an der es sich befindet, oder `-1`, wenn es nicht im Stapel enthalten ist. Das oberste Element hat die Position `1`.

Programm 5.12

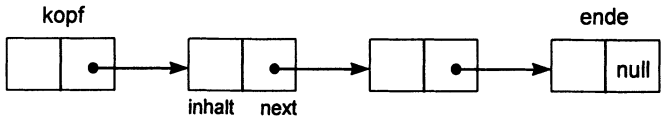
```
// Datei: StackTest.java
import java.util.*;

public class  StackTest {
   public static void main(String[] args) {
      Stack s = new Stack();
      int size;
      s.push(Integer.valueOf("1"));
      s.push(Integer.valueOf("2"));
      s.push(Integer.valueOf("3"));
      size = s.size();
      for (int i = 0; i < size; ++i)
         System.out.println(
            ((Integer)s.pop()).intValue());
   }
}
```

Die Ausgabe des Programms ist:

```
3
2
1
```

Das folgende Beispielprogramm stellt eine Technik vor, um beliebig viele Objekte vom Typ `Object` in einer so genannten *verketteten Liste* zu verwalten. Eine solche Liste besteht aus Listenelementen, die jeweils Daten sowie eine Referenz auf das nächste Element enthalten. Ein Listenkopf verweist auf das erste Element, die Referenz des letzten Elements der Liste ist `null`.

Bild 5.2: Verkettete Liste



Programm 5.13

```
// Datei: ListenElement.java

class ListenElement {
   Object inhalt;
   ListenElement next;

   ListenElement(Object inhalt) {
      this.inhalt = inhalt;
      this.next = null;
   }
}

// Datei: ListenEnum.java
import java.util.*;

class ListenEnum implements Enumeration {
   private ListenElement e;

   public ListenEnum(ListenElement e) {
      this.e = e;
   }

   public boolean hasMoreElements() {
      return e != null;
   }

   public Object nextElement() throws
      NoSuchElementException {
      if (e == null)
         throw new NoSuchElementException();
      Object obj = e.inhalt;
      e = e.next;
      return obj;
   }
}

// Datei: Liste.java
import java.util.*;

public class Liste {
   protected ListenElement kopf, ende, aktuell;
   protected int laenge;
```

```
public Liste() {
   kopf = ende = aktuell = null;
   laenge = 0;
}

public boolean istLeer() {
   return kopf == null;
}

public int getLaenge() {
   return laenge;
}

public Object erstesElement() throws
   NoSuchElementException {
   if (kopf == null)
      throw new NoSuchElementException();
   aktuell = kopf;
   return aktuell.inhalt;
}

public Object naechstesElement() throws
   NoSuchElementException {
   if (aktuell == null)
      throw new NoSuchElementException();
   if (aktuell == ende)
      throw new NoSuchElementException();
   aktuell = aktuell.next;
   return aktuell.inhalt;
}

public Object aktuellesElement() throws
   NoSuchElementException {
   if (aktuell == null)
      throw new NoSuchElementException();
   return aktuell.inhalt;
}

public void addElement(Object obj) {
   ListenElement neu = new ListenElement(obj);
   if (kopf == null)
      kopf = ende = aktuell = neu;
   else {
```

```
            ende.next = neu;
            ende = aktuell = neu;
         }
         laenge++;
      }

      public void entferneElement() throws
         NoSuchElementException {
         if (aktuell == null)
            throw new NoSuchElementException();
         if (aktuell == kopf) {
            kopf = kopf.next;
            aktuell = kopf;
         }
         else {
            ListenElement temp = kopf;
            while (temp.next != aktuell)
               temp = temp.next;
            temp.next = aktuell.next;
            aktuell = temp;
            if (aktuell.next == null)
               ende = aktuell;
         }
         laenge--;
      }

      public Enumeration elemente() {
         return new ListenEnum(kopf);
      }
   }

   // Datei: ListenTest.java
   import java.util.*;

   public class ListenTest {
      public static void main(String[] args) {
         Liste liste = new Liste();

         liste.addElement("Element 1");
         liste.addElement("Element 2");
         liste.addElement("Element 3");

         System.out.println("Aufzaehlung:");
```

```
        Enumeration enum = liste.elemente();

        while (enum.hasMoreElements())
           System.out.println(enum.nextElement());

        System.out.println("erstes Element: " +
           liste.erstesElement());
        System.out.println("naechstes Element: " +
           liste.naechstesElement());
        liste.entferneElement();
        System.out.println("geloescht");
        System.out.println("aktuelles Element: " +
           liste.aktuellesElement());

        System.out.println("Aufzaehlung:");
        enum = liste.elemente();
        while (enum.hasMoreElements())
           System.out.println(enum.nextElement());
     }
}
```

Die Ausgabe des Programms ist:

```
Aufzaehlung:
Element 1
Element 2
Element 3
erstes Element: Element 1
naechstes Element: Element 2
geloescht
aktuelles Element: Element 1
Aufzaehlung:
Element 1
Element 3
```

5.7 Hashtabellen

Hashtable

Instanzen der Klasse `Hashtable` aus dem Paket `java.util` ermöglichen die Speicherung von Datenpaaren aus einem Schlüssel und einem zugeordneten Wert sowie den effizienten Zugriff auf den Wert über den Schlüssel. Eine

solche Struktur wird üblicherweise als zweispaltige Tabelle dargestellt. Beispiel:

Schlüssel	Wert
Farbe	rot
Hoehe	100.0
Durchmesser	150.0
Gewicht	5.0

Die Klassen der Objekte, die in die Tabelle eingetragen werden sollen, müssen die Methode boolean `equals(Object obj)` implementieren. Als Schlüssel sind nur solche Objekte zugelassen, deren Klassen außerdem die Methode `int hashCode()` implementieren (z.B. `String`, die Hüllklassen). Diese Methode erzeugt einen ganzzahligen Wert, den so genannten *Hashwert*, der für die Speicherung in *Hashtabellen* gebraucht wird. Verschiedene Schlüssel dürfen den gleichen Hashwert liefern. Wenn zwei Schlüssel mittels der Methode `equals` als gleich festgestellt werden, müssen ihre Hashwerte gleich sein.

Ein Objekt der Klasse `Hashtable` kann mit Hilfe des Konstruktors `Hashtable()` angelegt werden.

`int size()`
liefert die Anzahl der Schlüssel in der Tabelle.

`boolean isEmpty()`
liefert `true`, wenn die Tabelle keine Einträge enthält.

Einfügen

`Object put(Object key, Object value)`
fügt das Paar (`key`, `value`) in die Tabelle ein. `key` und `value` dürfen nicht `null` sein. Falls `key` bisher noch nicht in der Tabelle eingetragen ist, liefert die Methode `null` zurück, ansonsten den alten zugeordneten Wert, der nun durch den neuen Wert ersetzt wird.

Löschen

`Object remove(Object key)`
entfernt den Eintrag zum Schlüssel `key` und gibt den zugeordneten Wert zurück. Ist `key` nicht in der Tabelle vorhanden, wird `null` zurückgegeben.

`void clear()`
entfernt alle Einträge aus der Tabelle.

Zugriff auf Elemente

`Object get(Object key)`
liefert den dem Schlüssel `key` zugeordneten Wert. Ist der Schlüssel nicht in der Tabelle eingetragen, wird `null` zurückgegeben.

`boolean contains(Object value)`
liefert `true`, falls `value` in der Tabelle als Wert vorkommt.

`boolean containsKey(Object key)`
liefert `true`, falls `key` in der Tabelle als Schlüssel vorkommt.

Enumeration

`Enumeration elements()`
liefert eine Aufzählung aller Werte der Tabelle (siehe auch Kapitel 5.6).

`Enumeration keys()`
liefert eine Aufzählung aller Schlüssel der Tabelle (siehe auch Kapitel 5.6).

Das folgende Programm demonstriert das Arbeiten mit Hashtabellen, wobei für die Tabelleneinträge eigene Klassen deklariert werden.

Programm 5.14

```
// Datei: HashTest.java
import java.util.*;

public class HashTest {
   public static void main(String[] args) {
      Hashtable h = new Hashtable();
      Key k1 = new Key(4711);
      Key k2 = new Key(1122);
      Key k3 = new Key(5000);
      Value v1 = new Value(100.5);
      Value v2 = new Value(99.99);
```

```
        h.put(k1, v1);
        h.put(k2, v2);
        h.put(k3, v1);

        Enumeration e = h.keys();
        while (e.hasMoreElements()) {
           Key k = (Key)e.nextElement();
           System.out.println(k.getVar() + " -> " +
              ((Value)h.get(k)).getVar());
        }
     }
  }

  class Key {
     private int var;

     Key(int x) {
        var = x;
     }

     public boolean equals(Object obj) {
        return obj instanceof Key &&
           var == ((Key)obj).var;
     }

     public int hashCode() {
        return var % 100;
     }

     int getVar() {
        return var;
     }
  }

  class Value {
     private double var;

     Value(double x) {
        var = x;
     }
```

```
    public boolean equals(Object obj) {
        return obj instanceof Value &&
            var == ((Value)obj).var;
    }
    double getVar() {
        return var;
    }
}
```

Die Ausgabe des Programms ist:

```
5000 -> 100.5
1122 -> 99.99
4711 -> 100.5
```

Properties

Die Klasse `Properties` aus dem Paket `java.util` ist eine Subklasse der Klasse `Hashtable`. Als Schlüssel und Werte sind nur Strings erlaubt. Es existieren Methoden zum Laden und Speichern aus bzw. in Dateien. Ein Objekt dieser Klasse wird als *Property-Liste* bezeichnet.

`Properties()`
`Properties(Properties defaults)`
erzeugen eine Property-Liste. Ist `defaults` angegeben, so wird in dem Fall, dass der Schlüssel in der Liste nicht gefunden wird, in der Liste `defaults` gesucht.

`String getProperty(String key)`
liefert den dem Schlüssel `key` zugeordneten Wert oder `null`, wenn der Schlüssel nicht gefunden wurde.

`String getProperty(String key, String defaultValue)`
liefert den dem Schlüssel `key` zugeordneten Wert oder den Wert `defaultValue`, wenn der Schlüssel nicht gefunden wurde.

`Enumeration propertyNames()`
liefert ein `Enumeration`-Objekt, mit dem alle Schlüssel der Liste aufgelistet werden können (siehe auch Kapitel 5.6).

`void store(OutputStream out, String header) throws IOException`
schreibt die Liste in den `OutputStream out`. Als Kommentar wird `header` in die Ausgabedatei geschrieben. `store` speichert nicht die Einträge aus der Default-Liste.

`void load(InputStream in) throws IOException`
lädt die Einträge aus dem `InputStream in`.

`store` und `load` erzeugt bzw. erwartet ein spezielles Format für den Dateiinhalt (siehe folgendes Beispiel).

`InputStream` und `OutputStream` werden später im Rahmen der Dateiverarbeitung erläutert.

Datei prop.txt

```
#Beispiel
#Mon Jul 24 17:04:16 GMT+02:00 2000
Farbe=rot
Gewicht=5.0
Durchmesser=150.0
Hoehe=100.0
```

Programm 5.15

```
// Datei: PropTest.java
import java.util.*;
import java.io.*;

public class PropTest {
   public static void main(String[] args) throws
      IOException {
      Properties p = new Properties();
      String wert;

      FileInputStream ein = new FileInputStream(
         "prop.txt"); // (*)
      p.load(ein);
      ein.close(); // (*)

      wert = p.getProperty("Hoehe");
      System.out.println("Hoehe=" + wert);

      p.put("Gewicht", String.valueOf(6.5));

      FileOutputStream aus = new FileOutputStream(
         "prop.txt"); // (*)
```

```
        p.store(aus, "Beispiel");
        aus.close(); // (*)
    }
}
```

Die mit `(*)` gekennzeichneten Konstrukte werden an späterer Stelle erklärt.

5.8 Die Klasse System

Die Klasse `System` enthält wichtige Methoden zur Kommunikation mit dem zu Grunde liegenden Betriebssystem. Es können keine Objekte vom Typ `System` erzeugt werden. Alle Methoden von `System` sind Klassenmethoden.

in, out, err

`System` enthält folgende Klassenvariablen:

```
public final static InputStream in
public final static PrintStream out
public final static PrintStream err
```

Sie sind die *Standarddatenströme* zur Eingabe, Ausgabe und Fehlerausgabe.

Die Klassen `InputStream` und `PrintStream` werden später im Rahmen der Dateiverarbeitung erläutert.

gc

`static void gc()`
ruft den *Garbage Collector* auf (siehe Kapitel 3.3).

arraycopy

`static void arraycopy(Object src, int srcPos, Object dst, int dstPos, int length)`
kopiert `length` Elemente aus dem Array `src` in das Array `dst`, jeweils ab der Position `srcPos` bzw. `dstPos`.

exit

`static void exit(int status)`
beendet das laufende Programm. `status` wird an den Aufrufer des Programms übergeben. Üblicherweise signalisiert 0 ein fehlerfreies Programmende.

currentTimeMillis

`static long currentTimeMillis()`
liefert die Anzahl Millisekunden, die seit dem 1.1.1970 um 00:00:00 Uhr GMT bis zum Aufruf der Methode vergangen sind.

System-Properties

`static Properties getProperties()`
liefert die definierten Systemeigenschaften (*System-Properties*) der Plattform als Objekt der Klasse `Properties`.

`static String getProperty(String key)`
liefert den Wert der Systemeigenschaft mit dem Namen `key` oder `null`, wenn keine Eigenschaft mit diesem Namen gefunden wurde.

```
static String getProperty(String key,
  String defaultValue)
```
liefert den Wert der Systemeigenschaft mit dem Namen `key` oder den Wert `defaultValue`, wenn keine Eigenschaft mit diesem Namen gefunden wurde.

Tabelle 5.3: Einige System-Properties

Property	Bedeutung
`file.separator`	Dateipfadtrennzeichen
`java.class.path`	aktueller Klassenpfad
`java.class.version`	Version der Klassenbibliothek
`java.home`	Installationsverzeichnis
`java.vendor`	Herstellername
`java.vendor.url`	URL des Herstellers
`java.version`	Java-Versionsnummer
`line.separator`	Zeilentrennzeichen
`os.arch`	Betriebssystemarchitektur
`os.name`	Betriebssystemname
`os.version`	Betriebssystemversion
`path.separator`	Trennzeichen in PATH-Angaben
`user.dir`	aktuelles Arbeitsverzeichnis
`user.home`	Home-Verzeichnis
`user.name`	Anmeldename

Das folgende Programm listet alle verfügbaren Systemeigenschaften auf.

Programm 5.16

```
// Datei: SystemTest.java
import java.util.*;

public class SystemTest {
   public static void main(String[] args) {
      Properties p = System.getProperties();

      Enumeration e = p.propertyNames();
      while (e.hasMoreElements()) {
         String s = (String)e.nextElement();
         System.out.println(s + "=" +
            System.getProperty(s));
      }
   }
}
```

5.9 Klassendeskriptoren

Class

Jede Klasse und jedes Interface lässt sich durch ein Objekt der Klasse `Class` beschreiben. Ein solches Objekt wird als *Klassendeskriptor* bezeichnet. Es existieren keine Konstruktoren. Objekte vom Typ `Class` entstehen automatisch, wenn Klassen geladen werden.

Klassendeskriptoren ermöglichen es, zur Laufzeit Informationen über Klassen zu beschaffen und Objekte von beliebigen Klassen zu erzeugen, deren Existenz z.B. zur Entwicklungszeit des Programms noch nicht bekannt war.

Die Methode `final Class getClass()` der Klasse `Object` liefert einen Klassendeskriptor für die Klasse, für deren Objekt die Methode aufgerufen wurde.

Beispiel:

```
String s = new String();
Class desk = s.getClass();
```

`desk` ist ein Klassendeskriptor für die Klasse `String`.

Klassenliterale

Für die primitiven Datentypen und das Schlüsselwort `void` gibt es Konstanten des Typs `Class`. Diese werden durch den Typnamen mit Suffix `.class` gebildet, z.B. `int.class`. Allgemein erhält man für eine beliebige Klasse `A` mit `A.class` eine Referenz auf den Klassendeskriptor von `A`.

einige Methoden

```
static Class forName(String name)
  throws ClassNotFoundException
```

liefert einen Klassendeskriptor für die Klasse bzw. das Interface mit dem Namen `name`. Der Name muss vollständig spezifiziert sein, also z.B. `java.lang.String`.

```
Object newInstance()
  throws InstantiationException,
  IllegalAccessException
```

erzeugt ein neues Objekt der Klasse, die durch den Klassendeskriptor repräsentiert wird, für den die Methode aufgerufen wurde. Diese Klasse muss einen parameterlosen Konstruktor besitzen.

Beispiel:

```
Class desk = Class.forName("java.lang.String");
Object obj = desk.newInstance();
```

```
String getName()
```

liefert den voll qualifizierten Namen der Klasse (des Interface), die (das) durch den Klassendeskriptor repräsentiert wird.

```
Class getSuperclass()
```

liefert einen Klassendeskriptor für die Superklasse der Klasse, für deren Klassendeskriptor die Methode aufgerufen wurde. Repräsentiert dieser Klassendeskriptor die Klasse `Object` oder ein Interface, wird `null` zurückgegeben.

```
boolean isInterface()
```

liefert `true`, wenn der Klassendeskriptor, für den die Methode aufgerufen wurde, ein Interface repräsentiert.

```
Class[] getInterfaces()
```

liefert ein Array von Klassendeskriptoren für Interfaces, die diejenige Klasse implementiert hat bzw. von denen dasjenige

Interface abgeleitet ist, für deren bzw. für dessen Klassendeskriptor die Methode aufgerufen wurde.

Das folgende Beispielprogramm wendet diese Methoden an, um Informationen über Klassen zu erhalten, deren Namen über Tastatur eingegeben werden.

Programm 5.17

```
// Datei: ClassTest.java
import java.io.*;

public class ClassTest {
   public static void main(String[] args)
      throws IOException, ClassNotFoundException,
      InstantiationException, IllegalAccessException {

      BufferedReader ein = new BufferedReader(
         new InputStreamReader(System.in)); // (*)
      String zeile;
      Class c1, c2;
      Class[] v;
      boolean isInterface;
      int len;

      while (true) {
         System.out.println(
            "Klassen-/Interfacename oder \'q\':");
         zeile = ein.readLine(); // (*)

         if (zeile.equals("q"))
            break;

         c1 = Class.forName(zeile);
         isInterface = c1.isInterface();
         System.out.print(
            isInterface ? "Interface: " : "Klasse: ");
         System.out.println(c1.getName());

         c2 = c1.getSuperclass();
         if (c2 != null)
            System.out.println("Superklasse: " +
               c2.getName());

         v = c1.getInterfaces();
```

```
            len = v.length;

            if (len > 0) {
               if (isInterface)
                  System.out.println("Superinterfaces:");
               else
                  System.out.println(
                     "implementierte Interfaces:");
               for (int i = 0; i < len; ++i)
                  System.out.println(v[i].getName());
            }
            System.out.println();
         }
         ein.close(); // (*)
      }
   }
```

Die mit `(*)` gekennzeichneten Konstrukte werden an späterer Stelle erklärt.

Ein-/Ausgabebeispiel:

```
Klassen-/Interfacename oder 'q':
java.util.Vector

Klasse: java.util.Vector
Superklasse: java.util.AbstractList
implementierte Interfaces:
java.util.List
java.lang.Cloneable
java.io.Serializable

Klassen-/Interfacename oder 'q':
q
```

5.10 Zeitangaben und -berechnungen

In diesem Kapitel werden die Klassen `Date`, `TimeZone` und `GregorianCalendar` aus dem Paket `java.util` benutzt, um Datum und Uhrzeit anzugeben bzw. zu berechnen.

Date

Ein Objekt der Klasse `Date` repräsentiert einen Zeitpunkt.

Konstruktoren:

`Date()`
erzeugt ein Objekt, das die aktuelle Systemzeit des Rechners repräsentiert.

`Date(long time)`
erzeugt ein Objekt, das den Zeitpunkt `time` Millisekunden nach dem 1.1.1970 um 00:00:00 Uhr GMT repräsentiert.

Methoden:

`long getTime()`
liefert die Anzahl Millisekunden, die seit dem 1.1.1970 um 00:00:00 Uhr GMT vergangen sind.

`void setTime(long time)`
stellt das `Date`-Objekt so ein, dass es den Zeitpunkt `time` Millisekunden nach dem 1.1.1970 um 00:00:00 Uhr GMT repräsentiert.

`String toString()`
liefert eine Zeichenkette aus Wochentag, Monat, Tag, Stunde, Minute, Sekunde, lokale Zeitzone und Jahr.
Beispiel: `Sun Feb 14 16:45:36 GMT+01:00 2000`.

TimeZone

Objekte der Klasse `TimeZone` repräsentieren Zeitzonen.

`static String[] getAvailableIDs()`
liefert ein Array aller Zeitzonennamen. So steht z.B. GMT für die *Greenwich Mean Time*, die der *Universal Time* (UT) entspricht, und "Europe/Berlin" (GMT+01:00) für die in Deutschland geltende Zeitzone.

`static TimeZone getDefault()`
liefert die lokale Zeitzone des Rechners.

`String getID()`
liefert den Zeitzonennamen. Beispiel: `Europe/Berlin`.

`String getDisplayName()`
liefert eine ausführliche Angabe zur Zeitzone.
Beispiel: `GMT+01:00`.

Gregorian-Calendar

Die Klasse `GregorianCalendar` implementiert den gregorianischen Kalender, der in der gesamten westlichen Welt verwendet wird.

Konstruktoren:

```
GregorianCalendar()
GregorianCalendar(int jahr, int monat,
  int tag)
GregorianCalendar(int jahr, int monat,
  int tag, int stunde, int minute)
GregorianCalendar(int jahr, int monat,
  int tag, int stunde, int minute,
  int sekunde)
GregorianCalendar(TimeZone zone)
```

Die Konstruktoren verwenden entweder die aktuelle oder die angegebene Zeit und beziehen sich auf die lokale Zeitzone oder die angegebene Zeitzone. Monate werden nicht von 1 bis 12, sondern von 0 bis 11 gezählt.

Die Superklasse `Calendar` enthält eine Reihe von ganzzahligen konstanten Klassenvariablen, die als Feldbezeichner von den Methoden `set` und `get` benutzt werden.

Tabelle 5.4: Konstanten

Konstante	Bedeutung
ERA	Epoche
BC	Before Christ
AD	Anno Domini
YEAR	Jahr
MONTH	Monat (0 ... 11)
WEEK_OF_MONTH	Woche innerhalb des Monats
WEEK_OF_YEAR	Kalenderwoche
DATE	Tag im Monat
DAY_OF_MONTH	Tag im Monat
DAY_OF_WEEK	Wochentag (1 = Sonntag)

DAY_OF_YEAR	Tag bezogen auf das Jahr
AM	Vormittag
PM	Nachmittag
AM_PM	AM bzw. PM
HOUR	Stunde (0 ... 12)
HOUR_OF_DAY	Stunde (0 ... 23)
MINUTE	Minute
SECOND	Sekunde
MILLISECOND	Millisekunde
ZONE_OFFSET	Zeitzonenabweichung in Millisekunden relativ zu GMT
DST_OFFSET	Sommerzeitabweichung in Millisekunden
JANUARY ... DECEMBER	Werte für Monat
MONDAY ... SUNDAY	Werte für Wochentag

Methoden:

`int get(int field)`
liefert den Wert des Feldes, das durch die Konstante `field` bezeichnet wird. Beispiel: `get(Calendar.YEAR)`

`void set(int field, int value)`
setzt den Wert des Feldes, das durch die Konstante `field` bezeichnet wird, auf `value`.
Beispiel: `set(Calendar.YEAR, 2000)`

`void add(int field, int amount)`
addiert den Wert `amount` zum Wert des Feldes, das durch die Konstante `field` bezeichnet wird.

`Date getTime()`
liefert den Zeitpunkt des Kalender-Objekts als `Date`-Objekt.

`boolean equals(Object obj)`
liefert `true`, wenn dieses Objekt und `obj` gleich sind.

`boolean before(Object obj)`
liefert `true`, wenn dieses Objekt einen früheren Zeitpunkt darstellt als `obj`.

`boolean after(Object obj)`
liefert `true`, wenn dieses Objekt einen späteren Zeitpunkt darstellt als `obj`.

Programm 5.18

```
// Datei: DatumTest.java
import java.util.*;

public class DatumTest {
   public static void main(String[] args) {
      Date datum;
      TimeZone zone;
      GregorianCalendar cal1, cal2;

   // aktuelles Rechnerdatum
      datum = new Date();
      System.out.println(datum.toString());

   // Standardzeitzone des Rechners
      zone = TimeZone.getDefault();
      System.out.println(zone.getID());
      System.out.println(zone.getDisplayName());

   // aktuelles Rechnerdatum
      cal1 = new GregorianCalendar();
      zeigeDatum(cal1);

   // aktuelles Datum in der Zeitzone America/New_York
      zone = TimeZone.getTimeZone("America/New_York");
      cal2 = new GregorianCalendar(zone);
      zeigeDatum(cal2);

   // Datum ändern
      cal1.set(Calendar.DATE, 1);
      cal1.set(Calendar.MONTH, 0);
      cal1.set(Calendar.YEAR, 2001);
      zeigeDatum(cal1);
```

```
        cal1.add(Calendar.DATE, 32);
        zeigeDatum(cal1);
    }

    public static void zeigeDatum(Calendar c) {
        System.out.println(
            setZero(c.get(Calendar.DATE)) + "." +
            setZero(c.get(Calendar.MONTH)+1) + "." +
            setZero(c.get(Calendar.YEAR)) + " " +
            setZero(c.get(Calendar.HOUR_OF_DAY)) + ":" +
            setZero(c.get(Calendar.MINUTE)) + ":" +
            setZero(c.get(Calendar.SECOND))
        );
    }

    private static String setZero(int zahl) {
        return (zahl >= 0 && zahl <= 9) ?
            "0" + zahl : "" + zahl;
    }
}
```

Ausgabebeispiel:

```
Mon Jul 24 17:27:50 GMT+02:00 2000
Europe/Berlin
GMT+01:00
24.07.2000 17:27:52
24.07.2000 11:27:52
01.01.2001 17:27:52
02.02.2001 17:27:52
```

5.11 Aufgaben

1. Schreiben Sie ein Programm, das einen beliebigen Text verschlüsselt, indem ein Buchstabe durch seine Positionsnummer im Alphabet ersetzt wird. Umlaute wie 'ä' sowie 'ß' sollen wie "ae" bzw. "ss" behandelt werden. Alle Zeichen, die keine Buchstaben sind, sollen ignoriert werden.

2. Schreiben Sie ein Programm, das nach einer als Kommando-Parameter übergebenen Zeichenkette in einer Textdatei sucht, die wie im Programm 5.7 eingelesen wird. Die Trefferzeilen sollen mit ihrer Zeilennummer ausgegeben werden.

3. Zerlegen Sie einen Text in die einzelnen Wörter, wobei vorher festgelegte Wörter wie die Artikel "ein", "eine", "der", "die", "das" usw. herauszufiltern sind. Die Wörter sind zunächst in einem Objekt vom Typ `Vector` zu sammeln und anschließend in sortierter Reihenfolge auszugeben. Der Text kann wie im Programm 5.7 eingelesen werden.

4. Schreiben Sie ein Programm, das die Grundrechenarten Addition, Subtraktion, Multiplikation und Division für Fließkommazahlen ausführen kann. Dabei soll das Programm jeweils mit drei Parametern aufgerufen werden, z.B. `java Aufg0504 114.5 + 2.5`.

5. Programmieren Sie ein Glücksspiel: Ausgehend von einem Punktekonto von 10000 Punkten kann der Spieler einen beliebigen Betrag größer 0 und kleiner gleich dem zur Verfügung stehenden Punktestand darauf wetten, dass er eine Zahl zwischen 0 und 9 errät. Falls er gewinnt, erhält er das 9-fache seines Einsatzes, sonst ist der Einsatz verloren. Die zu erratende Zahl soll durch einen Zufallsgenerator gezogen werden. Die Zahlen können ähnlich wie im Programm 5.7 über Tastatur eingelesen werden.

6. Leiten Sie aus der Klasse `Liste` in Programm 5.13 eine Klasse mit dem Namen `StringListe` ab, die die Methode `addElement` so implementiert, dass eine sortierte Liste für Strings entsteht.

7. Implementieren Sie eine Klasse `MailAdressen`, die E-Mail-Adressen zu Namen speichern kann und den Zugriff über Namen ermöglicht. Die Einträge sollen aus einer Datei geladen werden können. Benutzen Sie intern die Klasse `Properties`.

8. Implementieren Sie eine Klasse `AktuellesDatum` mit einer Klassenmethode, die das aktuelle Datum wie in folgendem Beispiel als Zeichenkette liefert: Montag, den 24.7.2000.

9. Berechnen Sie die Zeit in Tagen, Stunden, Minuten und Sekunden, die seit dem 1.1.1990 um 00:00:00 Uhr vergangen ist.

6 Dateiverarbeitung

Java bietet eine umfangreiche Bibliothek von Klassen zur Dateiverarbeitung mit sequentiellem oder wahlfreiem Zugriff. Die sequentielle Ein- und Ausgabe wird mittels so genannter Datenströme realisiert.

Datenstrom

Ein *Datenstrom* (*Stream*) kann als eine Folge von Bytes betrachtet werden, die aus Programmsicht von einer Datenquelle aus (*Eingabestrom*) oder in eine Datensenke (*Ausgabestrom*) fließen. Dabei ist es bei diesem abstrakten Konzept zunächst nicht wichtig, von welchem Eingabegerät gelesen bzw. auf welches Ausgabegerät geschrieben wird. Methoden konkreter Subklassen bieten die nötigen Zugriffsroutinen. Datenströme können so geschachtelt werden, dass lese- und schreibtechnische Erweiterungen, wie z.B. das Puffern von Zeichen, möglich sind.

Alle Klassen zur Dateiverarbeitung sind im Paket `java.io` enthalten.

Standard-datenströme

Die von der Klasse `System` bereitgestellten *Standarddatenströme* `System.in` (vom Typ `InputStream`), `System.out` und `System.err` (beide vom Typ `PrintStream`) zur Eingabe von der Tastatur bzw. zur Ausgabe auf dem Bildschirm (siehe auch Kapitel 5.8) können beim Aufruf des Programms mit Hilfe der Symbole < und > bzw. >> so umgelenkt werden, dass von einer Datei gelesen bzw. in eine Datei geschrieben wird.

Beispiel:

```
java Programm < ein > aus
```

Hier wird auf Betriebssystemebene die Tastatur durch die Datei `ein` und der Bildschirm durch die Datei `aus` ersetzt. Mittels Methoden der Klasse `System.err` erzeugte Fehler-

meldungen erscheinen auf dem Bildschirm. >> anstelle von > schreibt an das Ende einer bestehenden Datei.

6.1 Übersicht über Datenströme

Datenströme können nach Ein- und Ausgabe, nach der Art der Datenquelle bzw. -senke (z.B. Datei, Array, String), nach der Art der Übertragung (z.B. gepuffert, gefiltert) und nach der Art der Dateneinheiten, die sie behandeln, unterschieden werden.

byteorientiert

Byte-Ströme verwenden als Dateneinheit das Byte (8 Bit). Ihre Implementierung wird von den abstrakten Klassen `InputStream` und `OutputStream` vorgegeben.

Tabelle 6.1 gibt eine Übersicht über die vorhandenen Klassen. Die Namen der abstrakten Klassen sind kursiv gedruckt. Die Vererbungshierarchie wird durch Einrückungen wiedergegeben.

Tabelle 6.1: Byte-Ströme

Input	Output
InputStream	*OutputStream*
- ByteArrayInputStream	- ByteArrayOutputStream
- FileInputStream	- FileOutputStream
- FilterInputStream	- FilterOutputStream
- - BufferedInputStream	- - BufferedOutputStream
- - DataInputStream	- - DataOutputStream
- - PushbackInputStream	
	- - PrintStream
- ObjectInputStream	- ObjectOutputStream
- PipedInputStream	- PipedOutputStream
- SequenceInputStream	

Erläuterungen

`InputStream/OutputStream`
ist Superklasse aller Byte-Eingabe- bzw. -Ausgabeströme.

`ByteArrayInputStream/ByteArrayOutputStream`
liest aus bzw. schreibt in `byte`-Arrays.

`FileInputStream/FileOutputStream`
liest aus bzw. schreibt in Dateien.

`FilterInputStream/FilterOutputStream`
ist mit `InputStream` bzw. `OutputStream` verbunden und kann Daten manipulieren.

`BufferedInputStream/BufferedOutputStream`
verfügt über Puffer für Schreib- bzw. Leseoperationen.

`DataInputStream/DataOutputStream`
besitzt Methoden zum Lesen bzw. Schreiben von Werten primitiver Datentypen.

`PushbackInputStream`
kann bereits gelesene Daten in den Eingabestrom zurückgeben.

`PrintStream`
gibt Werte verschiedener Datentypen im Textformat aus.

`ObjectInputStream/ObjectOutputStream`
kann komplette Objekte schreiben bzw. wieder rekonstruieren.

`PipedInputStream/PipedOutputStream`
bieten Methoden, um Daten zwischen zwei unabhängig laufenden Programmen (*Threads*) über so genannte *Pipes* auszutauschen.

`SequenceInputStream`
kann aus mehreren Eingabeströmen sukzessive lesen.

Basismethoden

Alle Zugriffsroutinen lösen im Fehlerfall Ausnahmen vom Typ `IOException` aus, die abgefangen oder weitergeleitet werden müssen.

InputStream

Grundlegende Methoden der Klasse `InputStream` sind:

`int read()`
liest das nächste Byte aus dem Eingabestrom und gibt es als Wert vom Typ `int` zurück. Der Wert `-1` zeigt das Ende des Eingabestroms an.

`int read(byte[] b)`
liest maximal `b.length` Bytes, speichert sie in das Array `b` und liefert die Anzahl gelesener Bytes als Rückgabewert. Der Wert `-1` zeigt das Ende des Eingabestroms an.

`int read(byte[] b, int off, int len)`
liest maximal `len` Bytes, speichert sie beginnend bei Position `off` in das Array `b` und liefert die Anzahl gelesener Bytes als Rückgabewert. Der Wert `-1` zeigt das Ende des Eingabestroms an.

`void close()`
schließt den Eingabestrom.

OutputStream

Grundlegende Methoden der Klasse `OutputStream` sind:

`void write(int b)`
schreibt `b` als Byte in den Ausgabestrom.

`void write(byte[] b)`
schreibt die Bytes aus dem Array `b` in den Ausgabestrom.

`void write(byte[] b, int off, int len)`
schreibt beginnend bei Position `off` `len` Bytes aus dem Array `b` in den Ausgabestrom.

`void close()`
schließt den Ausgabestrom.

`void flush()`
schreibt in Puffern enthaltene Daten sofort in den Ausgabestrom.

zeichenorientiert

Character-Ströme sind von den abstrakten Klassen `Reader` und `Writer` abgeleitet und lesen bzw. schreiben Unicode-Zeichen vom Typ `char` (16 Bit).

Tabelle 6.2 gibt eine Übersicht über die vorhandenen Klassen. Die Namen der abstrakten Klassen sind kursiv gedruckt. Die Vererbungshierarchie wird durch Einrückungen wiedergegeben.

Tabelle 6.2: Character-Ströme

Input	Output
`Reader`	*`Writer`*
- `BufferedReader`	- `BufferedWriter`
- - `LineNumberReader`	
- `CharArrayReader`	- `CharArrayWriter`
- *`FilterReader`*	- *`FilterWriter`*
- - `PushbackReader`	
- `InputStreamReader`	- `OutputStreamWriter`
- - `FileReader`	- - `FileWriter`
- `PipedReader`	- `PipedWriter`
- `StringReader`	- `StringWriter`
	- `PrintWriter`

Erläuterungen

`Reader/Writer`
ist Superklasse aller Character-Eingabe- bzw. -Ausgabeströme.

`BufferedReader/BufferedWriter`
verfügt über Puffer für Lese- bzw. Schreiboperationen.

`LineNumberReader`
hat die Fähigkeit, Zeilen zu zählen.

`CharArrayReader/CharArrayWriter`
liest aus bzw. schreibt in `char`-Arrays.

`FilterReader/FilterWriter`
ist mit `Reader` bzw. `Writer` verbunden und kann Daten manipulieren.

`PushbackReader`
kann bereits gelesene Daten in den Eingabestrom zurückgeben.

`InputStreamReader/OutputStreamWriter`
liest Bytes von einem `InputStream` und wandelt sie in `char`-Werte bzw. wandelt `char`-Werte in Bytes und schreibt sie in einen `OutputStream`. Wenn keine Codierung spezifiziert ist, wird eine voreingestellte Codierung benutzt.

`FileReader/FileWriter`
liest aus einer bzw. schreibt in eine Datei.

`PipedReader/PipedWriter`
bieten Methoden, um Daten zwischen zwei unabhängig laufenden Programmen (*Threads*) über so genannte *Pipes* auszutauschen.

`StringReader/StringWriter`
liest Zeichen aus einem String bzw. schreibt Zeichen in einen String.

`PrintWriter`
gibt Werte verschiedener Datentypen im Textformat aus.

Basismethoden

Alle Zugriffsroutinen lösen im Fehlerfall Ausnahmen vom Typ `IOException` aus, die abgefangen oder weitergeleitet werden müssen.

Reader

Grundlegende Methoden der Klasse `Reader` sind:

`int read()`
liest das nächste Zeichen aus dem Eingabestrom und gibt es als Wert vom Typ `int` zurück. Der Wert `-1` zeigt das Ende des Eingabestroms an.

`int read(char[] c)`
liest maximal `c.length` Zeichen, speichert sie in das Array `c` und liefert die Anzahl gelesener Zeichen als Rückgabewert. Der Wert `-1` zeigt das Ende des Eingabestroms an.

`int read(char[] c, int off, int len)`
liest maximal `len` Zeichen, speichert sie beginnend bei Position `off` in das Array `c` und liefert die Anzahl gelesener Zeichen als Rückgabewert. Der Wert `-1` zeigt das Ende des Eingabestroms an.

`void close()`
schließt den Eingabestrom.

Writer

Grundlegende Methoden der Klasse `Writer` sind:

`void write(int c)`
schreibt `c` als Zeichen in den Ausgabestrom.

`void write(char[] c)`
schreibt die Zeichen aus dem Array `c` in den Ausgabestrom.

`void write(char[] c, int off, int len)`
schreibt beginnend bei Position `off` `len` Zeichen aus dem Array `c` in den Ausgabestrom.

`void write(String s)`
schreibt die Zeichen aus `s` in den Ausgabestrom.

`void write(String s, int off, int len)`
schreibt beginnend bei Position `off` `len` Zeichen aus `s` in den Ausgabestrom.

`void close()`
schließt den Ausgabestrom.

`void flush()`
schreibt in Puffern enthaltene Daten sofort in den Ausgabestrom.

6.2 Die Klasse File

Objekte der Klasse `File` repräsentieren eine Datei oder ein Verzeichnis des Dateisystems.

Konstruktoren:

```
File(String path)
File(String path, String name)
File(File dir, String name)
```

`path` enthält den Verzeichnisnamen und `name` den Dateinamen. `dir` ist das `File`-Objekt zum Verzeichnis.

Informationen über Datei und Verzeichnis

`String getName()`
liefert den Dateinamen bzw. den letzten Namensbestandteil des Verzeichnisses.

`String getPath()`
liefert den Pfadnamen.

`String getAbsolutePath()`
liefert die komplette Pfadangabe.

`String getParent()`
liefert den Namen des übergeordneten Verzeichnisses.

`boolean exists()`
liefert `true`, wenn die Datei bzw. das Verzeichnis existiert.

`boolean canRead()`
liefert `true`, wenn ein lesender Zugriff möglich ist.

`boolean canWrite()`
liefert `true`, wenn ein schreibender Zugriff möglich ist.

`boolean isFile()`
liefert `true`, wenn das Objekt eine Datei repräsentiert.

`boolean isDirectory()`
liefert `true`, wenn das Objekt ein Verzeichnis repräsentiert.

`boolean isAbsolute()`
liefert `true`, wenn das Objekt einen kompletten Pfad repräsentiert.

`long length()`
liefert die Länge der Datei in Bytes bzw. 0, wenn die Datei nicht existiert.

`long lastModified()`
liefert den Zeitpunkt der letzten Änderung der Datei in Millisekunden seit dem 1.1.1970 um 00:00:00 Uhr GMT bzw. 0, wenn die Datei nicht existiert.

Programm 6.1

```
// Datei: FileInfo.java
import java.io.*;
import java.util.*;

public class FileInfo {
   public static void main(String[] args)
      throws IOException {
      if (args.length < 1) {
         System.err.println("Parameter: Dateiname");
         System.exit(1);
      }

      File datei = new File(args[0]);
      if (!datei.isFile()) {
         System.err.println("Keine Datei");
         System.exit(1);
      }

      System.out.println("Name: " + datei.getName());
      System.out.println("Path: " + datei.getPath());
      System.out.println("AbsolutePath: " +
         datei.getAbsolutePath());
      System.out.println("Parent: " + datei.getParent());
      System.out.println("canWrite: " + datei.canWrite());
      System.out.println("lastModified: " +
         new Date(datei.lastModified()));
      System.out.println("length: " + datei.length());
   }
}
```

Der Aufruf

```
java FileInfo Sourcen\Test.java
```

liefert z.B.

```
Name: Test.java
Path: Sourcen\Test.java
AbsolutePath: C:\Projekt\Sourcen\Test.java
Parent: Sourcen
canWrite: true
lastModified: Tue Jul 25 18:00:34 GMT+02:00
2000
length: 768
```

Zugriffe auf Verzeichnisse

`String[] list()`
liefert ein Array von Datei- und Verzeichnisnamen, wenn das `File`-Objekt ein Verzeichnis repräsentiert, sonst `null`.

`String[] list(FilenameFilter filter)`
verhält sich wie obige Methode, nur dass ausschließlich Namen geliefert werden, bei denen ein Aufruf der Methode `boolean accept(File dir, String name)` für das Objekt `filter` den Wert `true` liefert. `FilenameFilter` ist ein Interface.

Das folgende Programm zeigt nur die Namen derjenigen Dateien bzw. Verzeichnisse im aktuellen Verzeichnis an, die mit einem vorgegebenem Suffix enden.

Programm 6.2

```
// Datei: Filter.java
import java.io.*;

public class Filter {
   public static void main(String[] args)
      throws IOException {
      File dir = new File(".");
      String[] liste;
      if (args.length < 1)
         liste = dir.list();
```

```
      else
         liste = dir.list(new DateiSuffixFilter(args[0]));
      for (int i = 0; i < liste.length; ++i)
         System.out.println(liste[i]);
   }
}

class DateiSuffixFilter implements FilenameFilter {
   private String suffix;

   DateiSuffixFilter(String suffix) {
      this.suffix = suffix;
   }

   public boolean accept(File dir, String name) {
      if (name.endsWith(suffix))
         return true;
      else
         return false;
   }
}
```

`boolean mkdir()`
liefert `true`, wenn das Verzeichnis erstellt werden konnte.

`boolean renameTo(File dest)`
liefert `true`, wenn das repräsentierte Objekt in das durch `dest` repräsentierte Objekt umbenannt werden konnte.

`boolean delete()`
liefert `true`, wenn das repräsentierte Objekt gelöscht werden konnte, sonst `false`. Verzeichnisse müssen zum Löschen leer sein.

6.3 Sequentielle Dateiverarbeitung

In diesem Kapitel werden Konstruktoren und Methoden ausgewählter Klassen (siehe Tabellen 6.1 und 6.2) schrittweise in Beispielen vorgestellt.

Interaktive Eingabe

Programm 6.3 liest Eingaben von der Tastatur. Die Eingabeschleife kann durch Eingabe von `Strg+Z` oder Leereingabe (Return-Taste) beendet werden.

Konstruktoren:
`InputStreamReader(InputStream in)`
`BufferedReader(Reader in)`
`BufferedReader(Reader in, int size)`
`size` ist die Größe des Eingabepuffers.

BufferedReader

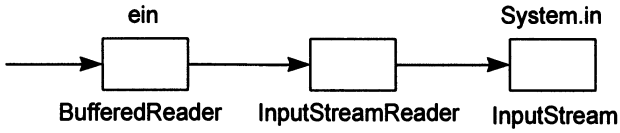
Methode der Klasse `BufferedReader`:
`String readLine()`
liest eine komplette Textzeile. Der zurückgegebene String enthält nicht das Zeilentrennzeichen. `readLine` gibt `null` zurück, wenn das Ende des Datenstroms erreicht ist.

Programm 6.3:

```
// Datei: Interaktiv.java
import java.io.*;

public class Interaktiv {
   public static void main(String[] args)
      throws IOException {
      BufferedReader ein = new BufferedReader(
         new InputStreamReader(System.in));
      String zeile;

      do {
         System.out.print("> ");
         zeile = ein.readLine();
         System.out.println(zeile);
      } while (zeile != null  && zeile.length() > 0);
      ein.close();
   }
}
```

Bild 6.1 zeigt wie die Leseaufrufe im Programm 6.3 für das `BufferedReader`-Objekt über den `InputStreamReader` an das `InputStream`-Objekt `System.in` weitergeleitet werden.

Bild 6.1: Weiterleitung von Leseaufrufen



Dateien kopieren

Die Programme 6.4 und 6.5 kopieren eine Textdatei zeichenweise bzw. zeilenweise.

Konstruktoren:

`FileReader(File datei)`
`FileReader(String dateiname)`
Wird die Datei nicht gefunden, so wird die Ausnahme `FileNotFoundException` ausgelöst.

```
FileWriter(File datei)
FileWriter(String dateiname)
FileWriter(String dateiname,
  boolean append)
```
Die Datei wird fortgeschrieben, wenn `append` den Wert `true` hat.

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```
`size` ist die Größe des Ausgabepuffers.

BufferedWriter

Methode der Klasse `BufferedWriter`:
`void newLine()`
schreibt ein Zeilentrennzeichen gemäß der System-Property `line.separator`.

Programm 6.4

```
// Datei: Kopie1.java
import java.io.*;

public class Kopie1 {
   public static void main(String[] args)
```

```
        throws IOException {
        int c;

        if (args.length != 2) {
           System.err.println("Parameter: eingabe ausgabe");
           System.exit(1);
        }

        BufferedReader ein = new BufferedReader(
           new FileReader(args[0]));
        BufferedWriter aus = new BufferedWriter(
           new FileWriter(args[1]));
        while ((c = ein.read()) != -1) {
           aus.write(c);
        }
        ein.close();
        aus.close();
     }
  }
```

Programm 6.5

```
// Datei: Kopie2.java
import java.io.*;

public class Kopie2 {
   public static void main(String[] args)
      throws IOException {
      String zeile;

      if (args.length != 2) {
         System.err.println("Parameter: eingabe ausgabe");
         System.exit(1);
      }

      BufferedReader ein = new BufferedReader(
         new FileReader(args[0]));
      BufferedWriter aus = new BufferedWriter(
         new FileWriter(args[1]));

      while ((zeile = ein.readLine()) != null) {
         aus.write(zeile);
         aus.newLine();
      }
```

```
        ein.close();
        aus.close();
    }
}
```

Zählen von Eingabezeilen

Programm 6.6 gibt gelesene Zeilen mit ihrer Zeilennummer aus.

Konstruktoren:

`LineNumberReader(Reader in)`
`LineNumberReader(Reader in, int size)`
`size` ist die Größe des Eingabepuffers.

`PrintWriter(Writer out)`
`PrintWriter(Writer out, boolean autoflush)`
Hat `autoflush` den Wert `true`, so werden gepufferte Daten immer dann geschrieben, wenn die Methode `println` aufgerufen wird.

Die beiden letzten Konstruktoren existieren auch mit `OutputStream` anstelle von `Writer`.

LineNumber-Reader

Methoden der Klasse `LineNumberReader`:

`String readLine()`
liest eine komplette Textzeile. Der zurückgegebene String enthält nicht das Zeilentrennzeichen. `readLine` gibt `null` zurück, wenn das Ende des Datenstroms erreicht ist.

`int getLineNumber()`
liefert den aktuellen Stand des Zeilenzählers.

PrintWriter

Methoden der Klasse `PrintWriter`:

`void print(`*Typ* `x)`
schreibt den angegebenen Parameter im Textformat in den Ausgabestrom. *Typ* steht für `boolean`, `char`, `int`, `long`, `float`, `double`, `char[]`, `String` oder `Object`. Im letzten Fall wird die Methode `String toString()` der Klasse `Object` bzw. die überschriebene Methode aufgerufen.

`void println(Typ x)`
verhält sich wie obige Methode mit dem Unterschied, dass zusätzlich das Zeilentrennzeichen geschrieben wird.

`void println()`
gibt das Zeilentrennzeichen aus.

Diese Methoden existieren auch für `PrintStream`.

Programm 6.6

```
// Datei: ZeilenNummern.java
import java.io.*;

public class ZeilenNummern {
   public static void main(String[] args)
      throws IOException {
      String zeile;

      if (args.length != 2) {
         System.err.println("Parameter: eingabe ausgabe");
         System.exit(1);
      }

      LineNumberReader ein = new LineNumberReader(
         new FileReader(args[0]));
      PrintWriter aus = new PrintWriter(
         new FileWriter(args[1]));
      while ((zeile = ein.readLine()) != null) {
         aus.println(ein.getLineNumber() + ": " + zeile);
      }
      ein.close();
      aus.close();
   }
}
```

gefilterte Datenströme

Programm 6.7 nutzt eine Subklasse der Klasse `FilterWriter`, um Umlaute und ß in die Zeichen `ae`, `oe`, `ue` bzw. `ss` zu wandeln.

Die abstrakte Klasse `FilterWriter` besitzt
- die Variable `protected Writer out` und den
- Konstruktor `protected FilterWriter(Writer out)`.

`out` ist der Ausgabestrom, in den geschrieben wird.

Die abstrakte Klasse `FilterReader` besitzt
- die Variable `protected Reader in` und den
- Konstruktor `protected FilterReader(Reader in)`.

`in` ist der Eingabestrom, aus dem gelesen wird.

Programm 6.7

```
// Datei: Umlaute.java
import java.io.*;

public class Umlaute {
   public static void main(String[] args)
      throws IOException {
      int c;

      if (args.length != 2) {
         System.err.println("Parameter: eingabe ausgabe");
         System.exit(1);
      }

      BufferedReader ein = new BufferedReader(
         new FileReader(args[0]));
      BufferedWriter aus = new BufferedWriter(
         new UmlautWriter(new FileWriter(args[1])));
      while ((c = ein.read()) != -1) {
         aus.write(c);
      }
      ein.close();
      aus.close();
   }
}

class UmlautWriter extends FilterWriter {
   public UmlautWriter(Writer out) {
      super(out);
   }

   public void write(int c) throws IOException {
      switch ((char)c) {
```

```
            case 'ä':
               super.write('a');
               super.write('e');
               break;
            ...
            case 'Ä':
               super.write('A');
               super.write('e');
               break;
            ...
            case 'ß':
               super.write('s');
               super.write('s');
               break;
            default:
               super.write(c);
         }
      }

      public void write(char[] c, int off, int len)
         throws IOException {
         for (int i = 0; i < len; ++i)
            write(c[off+i]);
      }

      public void write(char[] c) throws IOException {
         write(c, 0, c.length);
      }

      public void write(String s, int off, int len)
         throws IOException {
         for (int i = 0; i < len; ++i)
            write(s.charAt(off+i));
      }

      public void write(String s) throws IOException {
         write(s, 0, s.length());
      }
   }
```

Die Klasse `UmlautWriter` wird zwischen die Klassen `BufferedWriter` und `FileWriter` "eingeschoben". In der von `FilterWriter` abgeleiteten Klasse `UmlautWriter` werden die fünf `write`-Methoden überschrieben. Dabei werden die Ausgabezeichen vor der Übergabe an die Superklassenmethode manipuliert (Filterfunktion).

Komprimieren einer Datei

Programm 6.8 enthält einen einfachen Komprimierungsalgorithmus. Aufeinander folgende gleiche Zeichen werden durch drei Bytes ersetzt: `'@'`, das Wiederholungszeichen, die Anzahl der gleichen Zeichen (als Dezimalwert des Bytes). Die Komprimierung findet statt, wenn mindestens vier gleiche Zeichen aufeinander folgen. `'@'` darf nicht in der ursprünglichen Datei vorkommen.

Konstruktoren:
`PushbackReader(Reader in)`
`PushbackReader(Reader in, int size)`
`size` ist die Größe des Pushback-Puffers.

PushbackReader

Methoden der Klasse `PushbackReader`:

`void unread(int c)`
schreibt das Zeichen `c` wieder in den Eingabestrom zurück. Zurückgestellte Zeichen stehen beim nächsten Lesen wieder zur Verfügung.

`void unread(char[] c)`
schreibt die Zeichen aus dem Array `c` wieder in den Eingabestrom zurück.

`void unread(char[] c, int off, int len)`
schreibt `len` Zeichen aus dem Array `c` ab der Position `off` wieder in den Eingabestrom zurück.

Programm 6.8

```
// Datei: Kompression.java
import java.io.*;

public class Kompression {
   public static void main(String[] args)
      throws IOException {
      int z, c, next;
```

```
        if (args.length != 2) {
           System.err.println("Parameter: eingabe ausgabe");
           System.exit(1);
        }

        PushbackReader ein = new PushbackReader(
           new FileReader(args[0]));
        BufferedWriter aus = new BufferedWriter(
           new FileWriter(args[1]));

        while ((c = ein.read()) != -1) {
           for (z = 1; (next = ein.read()) != -1 &&
              c == next; ++z)
              if (z > 254) break;

           if (z > 3) {
              aus.write('@');
              aus.write(c);
              aus.write(z);
           }

           else
              for (int i = 0; i < z; ++i)
                 aus.write(c);

           if (next != -1)
              ein.unread(next);
        }

        ein.close();
        aus.close();
     }
  }
```

Binärdaten schreiben und lesen

Programm 6.9 schreibt und liest Daten im Binärformat.

Konstruktoren:

`FileInputStream(String dateiname)`
`FileInputStream(File datei)`
Wird die Datei nicht gefunden, so wird die Ausnahme `FileNotFoundException` ausgelöst.

```
FileOutputStream(File datei)
FileOutputStream(String dateiname)
FileOutputStream(String dateiname,
  boolean append)
```

Die Datei wird fortgeschrieben, wenn `append` den Wert `true` hat.

```
DataInputStream(InputStream in)
DataOutputStream(OutputStream out)
```

DataOutput-Stream

Methoden der Klasse `DataOutputStream`:

```
void writeBoolean(boolean x)
void writeChar(int x)
void writeByte(int x)
void writeShort(int x)
void writeInt(int x)
void writeLong(long x)
void writeFloat(float x)
void writeDouble(double x)
```

Diese Methoden schreiben Werte vom primitiven Datentyp im Binärformat. So schreibt z.B. `writeInt` 4 Bytes.

```
void writeUTF(String s)
```

schreibt eine Zeichenkette im *UTF-Format*. Die Ausgabe besteht aus zwei Bytes für die Länge der Zeichenkette, gefolgt von codierten Zeichen. Unicode-Zeichen werden als ein, zwei oder drei Bytes codiert.

DataInputStream

Methoden der Klasse `DataInputStream`:

```
boolean readBoolean()
char readChar()
byte readByte()
short readShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
```

Diese Methoden lesen Werte im Binärformat, die von entsprechenden Methoden der Klasse `DataOutputStream` geschrieben wurden.

`String readUTF()`
liest Zeichen im *UTF-Format*, wie sie von der Methode `writeUTF` der Klasse `DataOutputStream` geschrieben wurden.

EOFException

Wenn kein Byte gelesen werden kann, weil das Ende der Datei erreicht ist, wird die Ausnahme `EOFException` (Subklasse von `IOException`) ausgelöst.

Programm 6.9

```
// Datei: Binaer.java
import java.io.*;

public class Binaer {
   public static void main(String[] args)
      throws IOException {
      String s = "Heute ist das Wetter schön.";


      DataOutputStream aus = new DataOutputStream (
         new FileOutputStream("data.bin"));

      for (int i = 0; i < 10; ++i)
         aus.writeInt(i);
      aus.writeUTF(s);
      aus.close();

      DataInputStream ein = new DataInputStream (
         new FileInputStream("data.bin"));

      for (int i = 0; i < 10; ++i)
         System.out.print(ein.readInt() + " ");
      System.out.println();
      System.out.println(ein.readUTF());
      ein.close();
   }
}
```

Objekte serialisieren

Programm 6.10 schreibt und liest komplette Objekte im Binärformat.

Der Zustand eines Objekts (d.h. seine Attribute) wird in Bytes umgewandelt (*Serialisierung*) bzw. wieder zurückgewandelt, um das Objekt zu rekonstruieren (*Deserialisierung*).

Konstruktoren:
`ObjectInputStream(InputStream in)`
`ObjectOutputStream(OutputStream out)`

ObjectOutput-Stream

Methode der Klasse `ObjectOutputStream`:

`void writeObject(Object obj)`
schreibt das Objekt `obj` binär in den Ausgabestrom. Die Werte aller Attribute, die nicht als `static` bzw. `transient` deklariert sind, werden geschrieben.

Serializable

Die Klasse von `obj` muss das Interface `Serializable` implementieren. Dieses Interface besitzt keine Methoden.

ObjectInput-Stream

Methode der Klasse `ObjectInputStream`:

`Object readObject()`
liest ein Objekt aus dem Eingabestrom, das von der entsprechenden Methode der Klasse `ObjectOutputStream` geschrieben wurde. Ist das Ende der Datei erreicht, wird die Ausnahme `EOFException` ausgelöst.
Wenn die Klasse des Objekts nicht gefunden werden kann, wird eine `ClassNotFoundException` ausgelöst.

Programm 6.10

```
// Datei: Objekt.java
import java.io.*;

public class Objekt {
   public static void main(String[] args)
      throws IOException, ClassNotFoundException  {
      ObjectOutputStream aus = new ObjectOutputStream (
         new FileOutputStream("objekte"));

      Kunde a1 = new Kunde("Meier", 4711);
      Kunde a2 = new Kunde("Schmitz", 5020);
```

```
        aus.writeObject(a1);
        aus.writeObject(a2);
        aus.close();

        ObjectInputStream ein = new ObjectInputStream (
           new FileInputStream("objekte"));

        Kunde b1 = (Kunde)ein.readObject();
        Kunde b2 = (Kunde)ein.readObject();
        System.out.println(b1);
        System.out.println(b2);
        ein.close();
    }
}

class Kunde implements Serializable {
    String name;
    int nummer;

    public Kunde(String name, int nummer) {
        this.name = name;
        this.nummer = nummer;
    }

    public String toString() {
        return "Name: " + name + ", Nummer: " + nummer;
    }
}
```

6.4 Random-Access-Dateien

Objekte der Klasse `RandomAccessFile` ermöglichen den Zugriff auf Random-Access-Dateien. Eine *Random-Access-Datei* ist eine Datei mit wahlfreiem Zugriff, die zum Lesen oder zum Lesen und Schreiben geöffnet werden kann. Ein *Dateizeiger* (*Filepointer*) markiert die Stelle, an der das nächste Zeichen gelesen oder geschrieben wird.

Konstruktoren

```
RandomAccessFile(String dateiname,
  String modus)
RandomAccessFile(File datei, String modus)
```

`modus` gibt die Art des Zugriffs an. `"r"` steht für den Lesezugriff, `"rw"` für den Lese- und Schreibzugriff. Eine Datei wird neu angelegt, wenn sie beim Öffnen im Modus `"rw"` nicht existiert.

Methoden

Alle Zugriffsroutinen lösen im Fehlerfall Ausnahmen vom Typ `IOException` aus, die abgefangen oder weitergeleitet werden müssen.

`long getFilePointer()`
liefert die aktuelle Position des Dateizeigers. Das erste Byte der Datei hat die Position 0.

`void seek(long pos)`
setzt die Position des Dateizeigers auf `pos`.

`int skipBytes(int n)`
überspringt `n` Bytes.

`long length()`
liefert die Größe der Datei in Bytes.

`void close()`
schließt die Datei.

Die Klasse `RandomAccessFile` enthält die gleichen in Kapitel 6.3 spezifizierten Methoden zum Lesen und Schreiben wie die Klassen `DataInputStream` und `DataOutputStream`.

Das folgende Programm erstellt eine Artikeldatei, indem neu aufzunehmende Artikel an das Ende der Datei geschrieben werden. Auf Artikel kann mit Hilfe der Artikelnummer zugegriffen werden. Außerdem kann der Lagerbestand eines Artikels erhöht bzw. vermindert werden.

Programm 6.11

```
// Datei: ArtikelDatei.java
import java.io.*;
import java.util.*;

public class ArtikelDatei {
   private RandomAccessFile artikel;

   public ArtikelDatei(String datei) throws IOException {
      artikel = new RandomAccessFile(datei, "rw");
   }

   public void close() throws IOException {
      artikel.close();
   }

   public void addArtikel(Artikel a) throws IOException {
      artikel.seek(artikel.length());
      artikel.writeInt(a.getNr());
      artikel.writeDouble(a.getPreis());
      artikel.writeInt(a.getBestand());
   }

   public Artikel getArtikel(int nr) throws IOException {
      boolean found = false;
      int artNr, bestand;
      double preis;
      artikel.seek(0);

      try {
         while (!found) {
            artNr = artikel.readInt();
            if (artNr == nr)
               found = true;
            else
               artikel.skipBytes(12);
         }
      }
      catch (EOFException e) {
         return null;
      }

      preis = artikel.readDouble();
      bestand = artikel.readInt();
```

```
            return new Artikel(nr, preis, bestand);
        }

        public void addBestand(int nr, int zugang)
            throws IOException {
            Artikel a = getArtikel(nr);
            if (a != null) {
                artikel.seek(artikel.getFilePointer() - 4L);
                artikel.writeInt(a.getBestand() + zugang);
            }
        }

        public static void main(String[] args)
            throws IOException {
            ArtikelDatei artDatei = new ArtikelDatei("artikel");
            Artikel a1 = new Artikel(4711, 140.99, 1000);
            Artikel a2 = new Artikel(5011, 1000., 450);
            Artikel a3 = new Artikel(1112, 47.5, 1000);
            artDatei.addArtikel(a1);
            artDatei.addArtikel(a2);
            artDatei.addArtikel(a3);
            System.out.println(artDatei.getArtikel(5011));
            artDatei.addBestand(5011, -100);
            System.out.println(artDatei.getArtikel(5011));
            System.out.println(artDatei.getArtikel(9999));
            artDatei.close();
        }
    }

    class Artikel {
        private int nr;
        private double preis;
        private int bestand;

        public Artikel(int nr, double preis, int bestand) {
            this.nr = nr;
            this.preis = preis;
            this.bestand = bestand;
        }

        public int getNr() {
            return nr;
        }
```

```
    public double getPreis() {
        return preis;
    }

    public int getBestand() {
        return bestand;
    }
    public String toString() {
        return nr + ", " + preis + ", " + bestand;
    }
}
```

Die Ausgabe des Programms ist:

```
5011, 1000.0, 450
5011, 1000.0, 350
null
```

6.5 Aufgaben

1. Schreiben Sie ein Programm, das zuerst die Unterverzeichnisse und anschließend die Dateien eines vorgegebenen Verzeichnisses sortiert nach Namen auflistet.
2. Schreiben Sie ein Programm, das alle Buchstaben einer Textdatei in Großbuchstaben umwandelt.
3. Schreiben Sie ein Programm, das den Inhalt einer Datei an das Ende einer anderen Datei kopiert.
4. Eine Textdatei soll auf dem Bildschirm ausgegeben werden. Ist eine Bildschirmseite voll, soll das Programm anhalten, bis die Return-Taste gedrückt wird.
5. Nutzen Sie eine Instanz der Klasse `PrintWriter`, um Daten auf dem Bildschirm auszugeben.
6. Schreiben Sie ein Programm, das die Zeichen einer Textdatei beim Lesen über einen Filter sofort in Kleinbuchstaben umwandelt.
   Tipp: Benutzen Sie die `Character`-Methode
   `static toLowerCase(char c)`

7. Schreiben Sie ein Programm, das die mittels Programm 6.8 komprimierte Datei dekomprimiert.

8. Wie kann Programm 6.8 so erweitert werden, dass das Sonderzeichen `'@'` in der Datei als normales Zeichen vorkommen darf? Schreiben Sie ein Komprimierungs- und Dekomprimierungsprogramm. Tipp: Kommt `'@'` als normales Zeichen in der ursprünglichen Datei vor, so wird es verdoppelt.

9. In einem Programm erzeugte Objekte der Klasse `Konto` (siehe Kapitel 3.1 und 3.3) sollen in einer Datei gespeichert werden. Ein weiteres Programm soll die gespeicherten Objekte in einen Vektor einlesen.

7 Threads

Moderne Betriebssysteme können mehrere Programme quasi gleichzeitig (*Multitasking*) oder tatsächlich gleichzeitig (bei Mehrprozessorsystemen) ausführen. Die sequentielle Ausführung der Anweisungen eines Programms durch den Prozessor stellt einen *Prozess* dar, für den ein eigener Speicherbereich reserviert ist und der vom Betriebssystem verwaltet, gestartet und angehalten wird.

Thread

Ein *Thread* (Ausführungsfaden) ist ein einzelner in sich geschlossener Steuerfluss innerhalb eines Prozesses. Jeder Prozess besitzt einen Hauptthread, mit dem das Programm startet. Mehrere neue Threads können vom Programm selbst gestartet und angehalten werden (*Multithreading*). Diese Threads laufen dann alle parallel ab, besitzen jeweils einen eigenen Zustand mit Befehlszähler, Stack usw., arbeiten aber im Gegensatz zu Prozessen auf demselben Speicherbereich im Hauptspeicher.

In einem Einprozessorsystem kann die Gleichzeitigkeit dadurch simuliert werden, dass der Prozessor alle Prozesse und Threads reihum in schneller Folge schrittweise abarbeitet (*Time Slicing*).

Multithreading

In Java werden Threads durch die Klasse `Thread` und das Interface `Runnable` implementiert. Multithreading verbessert die Bedienbarkeit von grafischen Dialoganwendungen, insbesondere, wenn sie mit Animationen verbunden sind. Sie ermöglichen die Ausführung zeitintensiver Operationen im Hintergrund. Im Rahmen von Client-Server-Anwendungen müssen Serverprogramme Anfragen (z.B. Datenbankabfragen) verschiedener Clients gleichzeitig bearbeiten können. Zur parallelen Abarbeitung dieser Anfragen können clientbezogene Threads vom Serverprogramm gestartet werden.

7.1 Das Erzeugen von Threads

Problemstellung: Eine Dialoganwendung mit grafischer Benutzungsoberfläche soll laufend die aktuelle Zeit anzeigen. Über eine Schaltfläche soll diese Anzeige gestoppt werden.

Damit die Zeit laufend aktualisiert werden kann, müsste die Anweisung zur Zeitanzeige in einer Endlosschleife stehen. Dann könnte das Programm aber nicht auf die Betätigung der Schaltfläche reagieren, da es mit der Zeitanzeige ununterbrochen beschäftigt ist. Laufen die Zeitanzeige und die Ereignisverarbeitung jeweils in einem eigenen Thread, kann das Programm während der Ausgabe der aktuellen Zeit auf Benutzereingaben reagieren. Im Programm 7.1 wird diese Anwendung über Tastatureingabe simuliert.

Thread und Runnable

Objekte der Klasse `Thread` repräsentieren Threads. Diese Klasse implementiert das Interface `Runnable`, das die Methode `void run()` enthält.

Um einen konkreten Thread zu erzeugen, gibt es grundsätzlich zwei Möglichkeiten:

- Die Klasse, die die Threadausführung definiert, wird von `Thread` abgeleitet oder
- die Klasse implementiert das Interface `Runnable`.

run

In beiden Fällen muss die Methode `run` überschrieben werden. Sie enthält den Programmcode des Threads.

start

Die Methode `void start()` beginnt die Ausführung des Threads, indem sie die Methode `run` aufruft. Der Aufrufer der Methode `start` kann mit seiner Ausführung fortfahren, die nun parallel zur Ausführung des Threads läuft.

Programm 7.1 nutzt die erste Erzeugungsmöglichkeit, indem eine eigene Klasse von `Thread` abgeleitet wird.

Konstruktoren

Konstruktoren der Klasse `Thread` sind:

```
Thread()
Thread(String name)
```

`name` ersetzt den Standardnamen eines Threads durch einen eigenen Namen.

sleep

`static void sleep(long millis)`
hält die Ausführung des laufenden Threads für `millis` Millisekunden an. Wenn er durch einen anderen Thread unterbrochen wird, während er wartet, wird die Ausnahme `InterruptedException` ausgelöst.

Threads sollten so entwickelt werden, dass sie auf jedem System lauffähig sind. Ob *Time Slicing* unterstützt wird, hängt insbesondere vom Betriebssystem ab. Kann man sich nicht auf die Zeitscheibenzuteilung verlassen, sollte ein Thread in der `run`-Methode mittels `sleep` jeweils für kurze Zeit deaktiviert werden, so dass andere Threads zum Zuge kommen.

interrupt

`void interrupt()`
unterbricht den Thread, für den die Methode aufgerufen wurde. Wenn der Thread wartet, wird er aufgeweckt und löst eine `InterruptedException` aus.

join

`void join()`
wartet, bis der Thread, für den die Methode aufgerufen wurde, beendet ist. Bei Unterbrechung während der Wartezeit wird eine `InterruptedException` ausgelöst.

Programm 7.1

```
// Datei: ThreadTest1.java
import java.io.*;
import java.util.*;

public class ThreadTest1 {
   public static void main(String[] args)
      throws IOException {
      byte[] b = new byte[3];
      Zeitanzeige zeit = new Zeitanzeige();
      zeit.beenden = false;
      zeit.start();
      while (true) {
         System.in.read(b);
         if (b[0] == 'e') {
            zeit.beenden = true;
            break;
         }
      }
```

```
    }
}

class Zeitanzeige extends Thread {
   public boolean beenden;

   public void run() {
      while (!beenden) {
         System.out.println((new Date()).toString());
         try {
            Thread.sleep(1000);
         }
         catch (InterruptedException e) { }
      }
   }
}
```

Die Zeitanzeige wird durch Eingabe von `'e'` beendet.

Programm 7.2 nutzt für dieselbe Aufgabe die zweite Erzeugungsmöglichkeit. Die Klasse, die die Threadausführung definiert, implementiert das Interface `Runnable`. Diese zweite Möglichkeit muss immer dann genutzt werden, wenn die Klasse selbst Subklasse einer anderen Klasse ist.

Konstruktoren

In diesem Fall muss einer der folgenden Konstruktoren der Klasse `Thread` genutzt werden:

```
Thread(Runnable runObj)
Thread(Runnable runObj, String name)
```

`runObj` ist eine Referenz auf das Objekt, dessen `run`-Methode benutzt werden soll. `name` ersetzt den Standardnamen eines Threads durch einen eigenen Namen.

Programm 7.2

```
// Datei: ThreadTest2.java
import java.io.*;
import java.util.*;

public class ThreadTest2 {
   public static void main(String[] args)
      throws IOException {
      byte[] b = new byte[3];
```

```
        Zeitanzeige zeit = new Zeitanzeige();
        Thread t = new Thread(zeit);
        zeit.beenden = false;
        t.start();
        while (true) {
           System.in.read(b);
           if (b[0] == 'e') {
              zeit.beenden = true;
              break;
           }
        }
     }
  }

  class Zeitanzeige implements Runnable {
     public boolean beenden;

     public void run() {
        while (!beenden) {
           System.out.println(new Date().toString());
           try {
              Thread.sleep(1000);
           }
           catch (InterruptedException e) { }
        }
     }
  }
```

Das folgende Beispiel zeigt, wie bestehende Methoden genutzt werden können, um sie *threadfähig* zu machen und ihre Ausführung im Hintergrund asynchron laufen zu lassen.

Die Klasse `Primzahl` in Programm 7.3 enthält eine Methode, die feststellt, ob eine vorgegebene Zahl eine Primzahl ist. Das Programm verwendet keine Threads und läuft im Vordergrund.

Programm 7.3

```
// Datei: Primzahl.java

public class Primzahl {
   public void print(int zahl) {
      if (zahl > 1 && istPrim(zahl))
```

```
            System.out.println(zahl + " ist Primzahl");
         else
            System.out.println(zahl + " ist keine Primzahl");
      }

      private boolean istPrim(int zahl) {
         for (int i = 2; i < zahl; ++i)
            if (zahl % i == 0)
               return false;
         return true;
      }
   }

   // Datei: PrimTest1.java
   import java.io.*;

   public class PrimTest1 {
      public static void main(String[] args)
         throws IOException {
         BufferedReader ein = new BufferedReader(
            new InputStreamReader(System.in));
         Primzahl p = new Primzahl();
         int zahl;
         while (true) {
            System.out.print("> ");
            zahl = (new Integer(ein.readLine())).intValue();
            if (zahl == -1)
               break;
            p.print(zahl);
         }
         ein.close();
      }
   }
```

Ein-/Ausgabebeispiel:

```
> 1234567
1234567 ist keine Primzahl
> 100000007
100000007 ist Primzahl
> -1
```

Berechnungen im Hintergrund

Um die Berechnungen der Klasse `Primzahl` im Hintergrund asynchron auszuführen, wird eine neue Klasse entworfen, die von der Klasse `Primzahl` abgeleitet ist und das Interface `Runnable` implementiert. In der Klasse `PrimTest1` muss nur `Primzahl` durch `PrimzahlThread` ersetzt werden.

Programm 7.4

```
// Datei: PrimzahlThread.java

public class PrimzahlThread extends Primzahl
   implements Runnable {
   private int zahl;

   public void print(int zahl) {
      this.zahl = zahl;
      Thread t = new Thread(this);
      t.start();
   }

   public void run() {
      super.print(zahl);
   }
}

// Datei: PrimTest2.java
import java.io.*;

public class PrimTest2 {
   public static void main(String[] args)
      throws IOException {
      BufferedReader ein = new BufferedReader(
         new InputStreamReader(System.in));
      PrimzahlThread p = new PrimzahlThread();
      int zahl;
      while (true) {
         System.out.print("> ");
         zahl = (new Integer(ein.readLine())).intValue();
         if (zahl == -1)
            break;
         p.print(zahl);
      }
      ein.close();
   }
}
```

Ein-/Ausgabebeispiel:

```
> 100000007
> 100000037
> 100000039
> -1

100000007 ist Primzahl
100000037 ist Primzahl
100000039 ist Primzahl
```

Einige nützliche Methoden der Klasse `Thread`:

`String getName()`
liefert den Namen des Threads.

`void setName(String name)`
gibt dem Thread den neuen Namen `name`.

Sind mehrere Threads vorhanden, so werden diejenigen mit höherer Priorität vor denen mit niedrigerer Priorität ausgeführt.

`int getPriority()`
liefert die Priorität des Threads.

`void setPriority(int p)`
setzt die Priorität des Threads. `p` muss zwischen den Werten `Thread.MIN_PRIORITY` und `Thread.MAX_PRIORITY` liegen. `Thread.NORM_PRIORITY` ist der Normalwert.

7.2 Synchronisation von Threads

Beim Multithreading können Probleme durch den gleichzeitigen Zugriff auf Objekte und Variablen auftreten. Um undefinierte Ergebnisse zu vermeiden, müssen die Zugriffe bei Änderung gemeinsamer Daten synchronisiert werden.

Kritische Programmteile, die zu einer Zeit nur von einem Thread durchlaufen werden dürfen, müssen bei der Programmentwicklung erkannt und dann geschützt werden. Java bietet hierzu zwei Möglichkeiten:

- Schützen einer kompletten Methode oder
- Schützen eines Blocks innerhalb einer Methode.

synchronized für eine Methode

Wir beschäftigen uns zunächst mit der ersten Möglichkeit. Durch den Modifizierer `synchronized` können Instanzmethoden synchronisiert werden, so dass nur eine von ihnen zu einer Zeit von einem Thread benutzt werden kann. Wird eine solche Methode aufgerufen, wird versucht, das Objekt, für das sie aufgerufen wurde, für alle anderen `synchronized`-Methoden und -Blöcke zu sperren. Ist das Objekt bereits gesperrt, muss die Methode warten, bis der Konkurrent die Sperre aufgehoben hat. Nach Ausführung der Methode wird die Sperre für das Objekt aufgehoben.

Programm 7.5 demonstriert diesen Sachverhalt. Ein Objekt der Klasse `Zaehler` verwaltet in einer Datei den aktuellen Zählerstand. Dieser wird auf Anforderung um 1 erhöht und dann geliefert. Die `static`-Deklaration des Zählerobjekts in der Klasse `ZaehlerTest` garantiert, dass stets ein und dasselbe Objekt für die konkurrierenden Zugriffe beim Aufruf der Methode `getNext` gesperrt wird. Würde der Modifizierer `synchronized` für diese Methode fehlen, würden mehrfach gleiche Zählerstände zurückgeliefert sowie auch Ausnahmen vom Typ `IOException` ausgelöst.

Programm 7.5

```
// Datei: Zaehler.java
import java.io.*;

public class Zaehler {
   private String datei;

   public Zaehler(String datei) {
      this.datei = datei;
   }

   public void init(int zahl) throws IOException {
      DataOutputStream aus = new DataOutputStream(
         new FileOutputStream(datei));
      aus.writeInt(zahl);
      aus.close();
   }
```

```
    public synchronized int getNext() throws IOException {
        DataInputStream ein = new DataInputStream(
            new FileInputStream(datei));
        int zahl;
        zahl = ein.readInt();
        ein.close();
        DataOutputStream aus = new DataOutputStream(
            new FileOutputStream(datei));
        aus.writeInt(++zahl);
        aus.close();
        return zahl;
    }
}


// Datei: ZaehlerTest.java
import java.io.*;

public class ZaehlerTest extends Thread {
    static private Zaehler zaehler;
    private String name;

    static {
        zaehler = new Zaehler("zahl");
    }

    public ZaehlerTest(String name) {
        this.name = name;
    }

    public static void main(String[] args)
        throws IOException {
        zaehler.init(10000);
        Thread[] t = new Thread[5];

        for (int i = 0; i < 5; ++i) {
            t[i] = new ZaehlerTest("Thread-Nr. " + i);
            t[i].start();
        }
    }
```

```
    public void run() {
        for (int i = 0; i < 10; ++i) {
            try {
                System.out.println(name + ": " +
                    zaehler.getNext());
            }
            catch (IOException e) {
                System.err.println(e);
            }
        }
    }
}
```

Ausgabebeispiel:

```
Thread-Nr. 0: 10001
Thread-Nr. 1: 10002
Thread-Nr. 2: 10003
Thread-Nr. 4: 10004
Thread-Nr. 3: 10005
Thread-Nr. 0: 10006
Thread-Nr. 1: 10007
Thread-Nr. 2: 10008
Thread-Nr. 4: 10009
Thread-Nr. 3: 10010
Thread-Nr. 0: 10011
...
```

Producer/ Consumer

Das nächste Beispiel demonstriert ein so genanntes *Producer/ Consumer*-Problem. Ein Thread arbeitet als Produzent und erzeugt in zufälligen Abständen Zufallszahlen. Ein anderer Thread, der Konsument, "verbraucht" diese Zahlen in zufälligen Abständen. Die Zahlen werden in einem gemeinsamen Datenlager zwischengespeichert. Dort kann nur eine Zahl zu einer Zeit gespeichert sein. Die unabhängig voneinander laufenden Threads müssen sich synchronisieren. Der Konsument muss warten, bis eine neue Zahl erzeugt worden ist, und der Produzent muss warten, bis die vorher erzeugte und abgelegte Zahl verbraucht worden ist.

wait und notify

Die erforderliche Synchronisation erfolgt über die Methoden `wait` und `notify` bzw. `notifyAll` der Klasse `Object`. Sie dürfen nur innerhalb von `synchronized`-Methoden oder -Blöcken auftreten und werden für das gesperrte Objekt aufgerufen.

`void wait()`
Die Sperre für das Objekt wird aufgehoben und der Thread, der `wait` aufgerufen hat, wartet solange, bis er durch den Aufruf der Methode `notify` oder `notifyAll` durch einen anderen Thread, der in den Besitz der Sperre für dieses Objekt gelangt ist, aufgeweckt wird. Die Methode `wait` kann die Ausnahme `InterruptedException` auslösen.

`void notify()`
Einer von evtl. mehreren Threads, der für dasselbe Objekt, für das diese Methode aufgerufen wird, `wait` aufgerufen hat, wird aufgeweckt. Da der aufgeweckte Thread versucht, dieses Objekt wieder zu sperren, läuft er erst weiter, wenn der aufrufende Thread das Objekt freigegeben hat.

`void notifyAll()`
Alle wartenden Threads werden aufgeweckt und bewerben sich um eine Sperre.

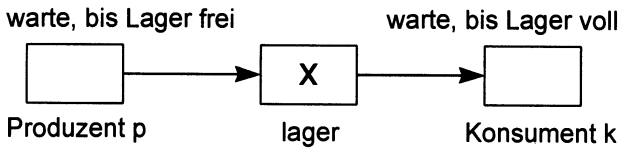


Bild 7.1: Kommunikation zwischen Threads

Programm 7.6

```
// Datei: Lager.java

public class Lager {
   private int wert;
   private boolean leer;

   public Lager() {
      leer = true;
   }
```

```
    public synchronized void put(int wert) {
        while (!leer) {
            try {
                wait();
            }
            catch (InterruptedException e) { }
        }
        this.wert = wert;
        leer = false;
        notify();
    }

    public synchronized int get() {
        while (leer) {
            try {
                wait();
            }
            catch (InterruptedException e) { }
        }
        leer = true;
        notify();
        return wert;
    }
}

// Datei: Produzent.java

public class Produzent extends Thread {
    private Lager lager;
    private int anz;

    public Produzent(Lager lager, int anz) {
        this.lager = lager;
        this.anz = anz;
    }

    public void run() {
        int wert = 0;

        for (int i = 0; i < anz; ++i) {
            wert = (int)(Math.random() * 100);
            try {
                Thread.sleep((int)(Math.random() * 1000));
```

```
            }
            catch (InterruptedException e) { }
            lager.put(wert);
            System.out.println((i+1) + ". Wert " + wert +
               " produziert");
         }
      }
   }

   // Datei: Konsument.java

   public class Konsument extends Thread {
      private Lager lager;
      private int anz;

      public Konsument(Lager lager, int anz) {
         this.lager = lager;
         this.anz = anz;
      }

      public void run() {
         int wert;
         for (int i = 0; i < anz; ++i) {
            wert = lager.get();
            try {
               Thread.sleep((int)(Math.random() * 1000));
            }
            catch (InterruptedException e) { }
            System.out.println("\t\t" + (i+1) + ". Wert " +
               wert + " konsumiert");
         }
      }
   }

   // Datei: PKTest.java

   public class PKTest {
      public static void main(String[] args) {
         Lager lager = new Lager();
         Produzent p = new Produzent(lager, 10);
         Konsument k = new Konsument(lager, 10);
         p.start();
         k.start();
```

```
    }
}
```

Ausgabebeispiel:

```
1. Wert 98 produziert
2. Wert 44 produziert
                    1. Wert 98 konsumiert
                    2. Wert 44 konsumiert
3. Wert 33 produziert
                    3. Wert 33 konsumiert
4. Wert 52 produziert
5. Wert 22 produziert
                    4. Wert 52 konsumiert
                    5. Wert 22 konsumiert
6. Wert 11 produziert
7. Wert 13 produziert
                    6. Wert 11 konsumiert
8. Wert 19 produziert
                    7. Wert 13 konsumiert
                    8. Wert 19 konsumiert
9. Wert 56 produziert
                    9. Wert 56 konsumiert
10. Wert 12 produziert
                    10. Wert 12 konsumiert
```

synchronized für einen Block

Programm 7.7 ist eine Variante, die anstelle von Methoden mit dem Modifizierer `synchronized` Anweisungsblöcke benutzt, die mit dem Schlüsselwort `synchronized` eingeleitet werden:

```
synchronized (obj) { ... }
```

`obj` wird vor dem Zugriff durch andere Threads geschützt, solange die Anweisungen im Block ausgeführt werden. Beim Eintritt in den Block wird versucht, das Objekt `obj` für alle anderen `synchronized`-Methoden bzw. -Blöcke zu sperren. Ist das Objekt bereits gesperrt, muss der Block warten, bis der Konkurrent die Sperre aufgehoben hat. Beim Austritt aus dem Block wird die Sperre für das Objekt aufgehoben.

Programm 7.7 enthält nun statt der Methodenaufrufe die entsprechenden `synchronized`-Blöcke innerhalb der Methode `run`.

Programm 7.7

```
// Datei: Lager2.java

public class Lager2 {
   public int wert;
   public boolean leer;

   public Lager2() {
      leer = true;
   }
}


// Datei: Produzent2.java
...
         synchronized (lager) {
            while (!lager.leer) {
               try {
                  lager.wait();
               }
               catch (InterruptedException e) { }
            }
            lager.wert = wert;
            lager.leer = false;

            lager.notify();
         }
...


// Datei: Konsument2.java
...
         synchronized(lager) {
            while (lager.leer) {
               try {
                  lager.wait();
               }
               catch (InterruptedException e) { }
            }
```

```
            lager.leer = true;
            lager.notify();
            wert = lager.wert;
         }
...
```

7.3 Das Unterbrechen von Threads

Die Methoden `wait` und `notify` können benutzt werden, um die Ausführung eines Threads vorübergehend anzuhalten und wieder fortzusetzen.

Programm 7.8 nutzt ein Objekt der Klasse `Status` als Sperrobjekt. Die Steuerung erfolgt über den Wert der `boolean`-Variablen `anhalten`.

Programm 7.8

```
// Datei: ZeitTest.java
import java.io.*;
import java.util.*;

public class ZeitTest {
   private Zeitanzeige zeit;
   private Status s;

   public static void main(String[] args)
      throws IOException {
      ZeitTest zt = new ZeitTest();
      byte[] b = new byte[3];

      zt.init();
      while (true) {
         System.in.read(b);
         switch (b[0]) {
            case 'a':
               zt.stop();
               break;
            case 'w':
               zt.start();
               break;
            case 'e':
               zt.ende();
               System.exit(0);
         }
```

```
            }
        }

        public void init() {
            s = new Status();
            zeit = new Zeitanzeige(s);
            zeit.start();
        }

        public void stop() {
            s.anhalten = true;
        }

        public void start() {
            synchronized(s) {
                s.anhalten = false;
                s.notify();
            }
        }

        public void ende() {
            s.beenden = true;
        }
    }

    class Status {
        public boolean anhalten, beenden;

        public Status() {
            anhalten = false;
            beenden = false;
        }
    }

    class Zeitanzeige extends Thread {
        private Status s;

        public Zeitanzeige(Status s) {
            this.s = s;
        }

        public void run() {
            while (!s.beenden) {
```

```
            synchronized(s) {
               try {
                  while (s.anhalten)
                     s.wait();
               }
               catch (InterruptedException e) { }
            }
            System.out.println(new Date().toString());
            try {
               Thread.sleep(1000);
            }
            catch (InterruptedException e) { }
         }
      }
   }
```

7.4 Aufgaben

1. Entwerfen Sie eine Klasse, die das Interface `Runnable` implementiert sowie die Methoden `main`, `start` und `stop`. `start` erzeugt einen neuen Thread, `stop` beendet ihn. Für jeden neu erzeugten Thread ist auch ein neuer Name zur Unterscheidung zu wählen. Der Thread soll seinen Namen in Abständen von zwei Sekunden ausgeben. In `main` können die Methoden `start` und `stop`, gesteuert über Tastatureingabe, aufgerufen werden.

2. Entwerfen Sie eine Klasse `Student` mit den Attributen `name`, `vorname` und `matrikelnummer` und den beiden `synchronized`-Methoden `setStudent` und `toString`. Die Methode `setStudent` initialisiert das Objekt, `toString` liefert die Daten des Objekts als Zeichenkette. Das Hauptprogramm (`main`) soll ein gemeinsam genutztes Objekt vom Typ `Student` erzeugen, das jeweils einem *Schreib-* und einem *Lese-Thread* übergeben wird. Benutzen Sie die Methoden `wait` und `notify`, um sicherzustellen, dass der Lese-Thread wartet, bis der Schreib-Thread das Objekt initialisiert hat,

und der Schreib-Thread erst wieder aktualisiert, wenn die vorhandenen Daten gelesen wurden.

3. Schreiben Sie ein Programm, das das folgende *Producer/Consumer*-Problem löst. Ein Thread (*Produzent*) erzeugt Zufallszahlen vom Typ `int` zwischen `0` und `60` und speichert sie als Objekte vom Typ `Integer` in einem Objekt vom Typ `Vector`. Ein anderer Thread (*Konsument*) verbraucht diese Zahlen, indem er sie in einem Balkendiagramm (Ausgabe von so vielen Zeichen `'*'` wie die Zahl angibt) darstellt. Ist die Zahl verbraucht, muss sie vom Konsumenten aus dem Vektor entfernt werden. Ist der Vektor leer, muss der Konsument so lange warten, bis der Produzent wieder eine Zahl gespeichert hat. Produzent und Konsument sollen nach jedem Schritt eine kleine Pause einlegen. Die Produktion soll zudem über Tastatureingabe angehalten und wieder fortgesetzt werden können. Nutzen Sie zur Synchronisation `synchronized`-Blöcke sowie die Methoden `wait` und `notify`.

8 Grafische Benutzungsoberflächen

AWT

Die Klassen des Pakets `java.awt` stellen grundlegende Elemente zur Erstellung grafischer Benutzungsoberflächen, auch GUI (*Graphical User Interface*) genannt, bereit. Die Abkürzung AWT steht dabei für *Abstract Window Toolkit*. Mit Hilfe dieser Klassen können Fenster mit Menüs, Textfelder und Bedienungselemente, wie z.B. Schaltflächen und Auswahllisten, *plattformunabhängig* realisiert werden. Zur Darstellung vieler AWT-Komponenten wird auf das jeweilige Betriebssystem zurückgegriffen (*Peer-Ansatz*), so dass z.B. ein Button unter Windows ein typischer Windows-Button und auf einem Macintosh ein typischer Mac-Button ist. Damit beschränkt sich AWT auf diejenigen GUI-Komponenten, die von allen unterstützten Betriebssystemen dargestellt werden können.

AWT ist Bestandteil der Sammlung *Java Foundation Classes* (JFC), die weitere GUI-Komponenten auf Basis eines neuen Konzepts (*Swing-Klassen*) und grafische Verbesserungen enthält.

Swing

Die *Swing-Klassen*, die sich im Paket `javax.swing` befinden, stellen die neuen Oberflächenkomponenten dar. Fast alle Swing-Komponenten sind vollständig in Java implementiert und vom verwendeten Betriebssystem unabhängig (*lightweight*), so dass sie auf allen Rechnern ein einheitliches Aussehen und Verhalten (*look and feel*) besitzen. Die Swing-Bibliothek enthält zahlreiche neue Komponenten. Swing ersetzt zwar alle Grundkomponenten des AWT, nutzt aber einige Leistungen des AWT, wie z.B. die Ereignisbehandlung, weiter.

Dieses Kapitel beschränkt sich auf die exemplarische Darstellung der Möglichkeiten, die *AWT* und insbesondere *Swing* bieten.

8.1 Übersicht

Dieses Kapitel behandelt:

- Methoden zum *Zeichnen* von einfachen 2-dimensionalen geometrischen Figuren,
- *GUI-Komponenten* (Bedienungselemente) für die Kommunikation mit dem Benutzer,
- *Container* zur Aufnahme anderer Komponenten,
- *Layout-Manager*, die die Komponenten in einem Container anordnen,
- Methoden zur Behandlung von Ereignissen (*Event-Handling*), z.B. Drücken eines Buttons, Mausklick,
- Methoden zur Bearbeitung und *Darstellung von Bildern* (Bitmaps) und zur *Wiedergabe von Tönen* (Audioclips).

Component

Alle Swing-Klassen sind Subklassen der Klasse `Component`. Diese enthält grundlegende Methoden, die eine Komponente am Bildschirm darstellen und sie für die Benutzerinteraktion vorbereiten.

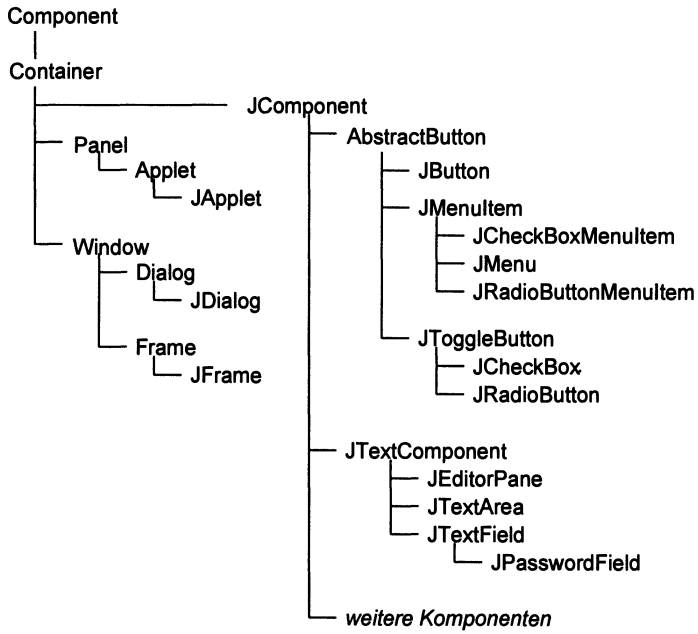
Container

Objekte der Klasse `Container` sind Komponenten, die wiederum Komponenten aufnehmen können. Sie dienen der Strukturierung der Benutzungsoberfläche. `Container` stellt Methoden zur Verfügung, um Komponenten hinzuzufügen, zu positionieren oder zu entfernen.

Bild 8.1 enthält die *Hierarchie der hier behandelten Klassen.*

Die Klassen `Component`, `Container`, `Panel`, `Window`, `Dialog` und `Frame` sind AWT-Klassen aus dem Paket `java.awt`, `Applet` liegt im Paket `java.applet`, alle anderen Klassen des Diagramms in Bild 8.1 liegen im Paket `javax.swing`.

Bild 8.1: Klassendiagramm



Weitere Komponenten sind:

`JComboBox`	`JOptionPane`	`JTable`
`JFileChooser`	`JPanel`	`JToolBar`
`JLabel`	`JScrollBar`	`JViewport`
`JList`	`JScrollPane`	
`JMenuBar`	`JTabbedPane`	

8.2 Fenster und Grafik

JFrame

Objekte der Klasse `JFrame` sind frei bewegliche Fenster mit Rahmen und Titelleiste.

Konstruktoren sind:

```
JFrame()
JFrame(String title)
```

Im zweiten Fall erhält das Fenster den Titel `title`.

Titel

`void setTitle(String title)`
setzt den Titel des Fensters.

`String getTitle()`
liefert den Titel des Fensters.

Dimension

Die Klasse `Dimension` aus `java.awt` repräsentiert eine Breite und eine Höhe. Sie besitzt die beiden `public`-Attribute `width` und `height` vom Typ `int` und den Konstruktor `Dimension(int width, int height)`.

Point

Die Klasse `Point` aus `java.awt` repräsentiert einen Punkt durch seine Koordinaten. Sie besitzt die beiden `public`-Attribute `x` und `y` vom Typ `int` und den Konstruktor `Point(int x, int y)`.

Größe und Position

Größe und Position einer Komponente (insbesondere eines Fensters) können mit folgenden Methoden der Klasse `Component` festgelegt bzw. abgefragt werden:

```
void setSize(int width, int height)
void setSize(Dimension d)
```

setzen die Größe der Komponente auf den Wert Breite mal Höhe (in Pixel) bzw. auf die Werte von `d`.

`Dimension getSize()`
liefert die aktuelle Größe der Komponente.

```
void setLocation(int x, int y)
void setLocation(Point p)
```

platzieren die linke obere Ecke der Komponente an Position `(x,y)` bzw. `p` der übergeordneten Komponente.

`Point getLocation()`
liefert die linke obere Ecke der Komponente relativ zum Koordinatensystem der übergeordneten Komponente.

Koordinatensystem

Der Ursprung `(0,0)` des Koordinatensystems liegt in der linken oberen Ecke. Positive x-Werte erstrecken sich nach rechts, positive y-Werte nach unten.

Insets

Die Breite der Randelemente eines Fensters (Rahmen und Titelleiste) kann mit der folgenden `Container`-Methode abgefragt werden:

```
Insets getInsets()
```

Die Klasse `Insets` repräsentiert die Breite der Ränder. Sie besitzt die `public`-Attribute `top`, `bottom`, `left` und `right` vom Typ `int`.

Anzeigen und Schließen

Die `Component`-Methode
`void setVisible(boolean b)`
macht die Komponente sichtbar (`true`) oder unsichtbar.

Die `Window`-Methode
`void dispose()`
löscht das Fenster und gibt die belegten Ressourcen frei.

Cursor

Mit der `Component`-Methode `setCursor` wird die Darstellung des Mauszeigers festgelegt:

```
void setCursor(Cursor c)
```

Die Klasse `Cursor` repräsentiert den Mauszeiger. Für `c` kann eine vorgegebene Konstante dieser Klasse eingesetzt werden, beispielsweise `CROSSHAIR_CURSOR`, `DEFAULT_CURSOR`, `HAND_CURSOR`, `MOVE_CURSOR`, `TEXT_CURSOR` oder `WAIT_CURSOR`.

Das folgende Beispielprogramm gibt Informationen zum Fenster aus und wechselt die Zeigerdarstellung nach einer Wartezeit. Dazu ist der Mauszeiger vom Benutzer in das Fenster zu bewegen. Nach einer bestimmten Zeit wird das Fenster automatisch geschlossen und das Programm beendet.

Die Methoden `getSize` und `getInsets` liefern erst dann richtige Werte, wenn das Fenster durch `setVisible` sichtbar gemacht worden ist.

Programm 8.1

```
// Datei: Fenster1.java
import java.awt.*;
import javax.swing.*;

public class Fenster1 {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Info");

      frame.setSize(300, 200);
      frame.setLocation(100, 100);
      frame.setVisible(true);

      druckeInfo(frame);
      warte(6000);
      frame.setTitle("Fenster mit HAND_CURSOR");
      frame.setCursor(new Cursor(Cursor.HAND_CURSOR));
      warte(6000);
      System.exit(0);
   }

   public static void druckeInfo(JFrame f) {
      Point p = f.getLocation();
      Dimension dim = f.getSize();
      Insets ins = f.getInsets();

      System.out.println("Titel: " + f.getTitle());
      System.out.println(
         "Position der linken oberen Ecke: (" +
         p.x + "," + p.y + ")");
      System.out.println("Breite: " + dim.width +
         ", Hoehe: " + dim.height);
      System.out.println("Breite des Randes:");
      System.out.println("oben: " + ins.top +
         ", unten: " + ins.bottom +
         ", links: " + ins.left +
         ", rechts: " + ins.right);
   }
```

```
public static void warte(long millis) {
      try {
         Thread.sleep(millis);
      }
      catch (InterruptedException e) { }
   }
}
```

Die Ausgabe des Programms ist:

```
Titel: Info
Position der linken oberen Ecke: (100,100)
Breite: 300, Hoehe: 200
Breite des Randes:
oben: 23, unten: 4, links: 4, rechts: 4
```

JPanel

Ein *Panel* ist die "Arbeitsfläche" für GUI-Komponenten. Die Klasse `JPanel` ist die einfachste Form eines Containers. Als Subklasse von `JComponent` können Panels in andere Container eingefügt werden und auch ineinander verschachtelt werden. Um ein Panel anzeigen zu können, muss es in ein Fenster eingefügt werden.

Komponenten hinzufügen

Mit der `Container`-Methode `add` können neue Komponenten in einen Container aufgenommen werden.

In ein Fenster können Komponenten jedoch nicht direkt aufgenommen werden. Ein `JFrame`-, `JDialog`- und `JApplet`-Objekt enthält einen speziellen Container (*content pane*), an den der `add`-Aufruf zu richten ist. Diesen Container erhält man mit der `JFrame`-Methode

```
Container getContentPane()
```

Beispiel:

Ist `komp` eine Komponente und `frame` ein `JFrame`-Objekt, so wird die Komponente in das Fenster wie folgt aufgenommen:

```
frame.getContentPane().add(komp);
```

Durch Überschreiben der `Component`-Methode

```
Dimension getPreferredSize()
```

kann die bevorzugte Größe der Komponente festgelegt werden. Diese Angabe wird bei der Anordnung der Komponente im Container verwendet.

Die `Window`-Methode

```
void pack()
```

gibt dem Fenster die Größe, die zur Darstellung der enthaltenen Komponenten nötig ist.

Zeichnen

In einem Panel können grafische Elemente ausgegeben werden. Diese Ausgabe erfolgt durch Überschreiben der `JComponent`-Methode `paintComponent`:

```
void paintComponent(Graphics g)
```

zeichnet die Komponente. `g` ist ein Objekt der Klasse `Graphics` aus `java.awt`, das den so genannten *Grafikkontext* darstellt und vom Java-Laufzeitsystem übergeben wird. Der Grafikkontext enthält diverse für das Zeichnen notwendige Basisinformationen.

Graphics

`Graphics` bietet Methoden zum Zeichnen von einfachen geometrischen Figuren und verwaltet Farbe und Schriftart, in der Grafik- und Textausgaben erfolgen.

`paintComponent` wird immer dann automatisch aufgerufen, wenn die Grafik ganz oder teilweise aktualisiert werden muss, was z.B. dann geschieht, wenn das Fenster zum ersten Mal angezeigt wird oder wenn es durch andere Elemente verdeckt war und wieder sichtbar wird.

Wenn `paintComponent` überschrieben wird, sollte als Erstes `super.paintComponent(g)` aufgerufen werden, damit die Komponente richtig dargestellt werden kann.

Anwendungen sollten `paintComponent` nie direkt aufrufen, sondern stattdessen – wenn neu gezeichnet werden muss – `void repaint()` nutzen.

Einige Methoden der Klasse `Graphics`:

Color

`void setColor(Color c)`
setzt die Farbe, die dann von allen folgenden Operationen benutzt wird. Für `c` kann z.B. eine der folgenden Konstanten der Klasse `Color` eingesetzt werden: `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white` und `yellow`.

Die Klasse `Color` stellt Farben nach dem *RGB-Farbmodell* dar. Der Konstruktor `Color(int r, int g, int b)` erzeugt eine Farbe aus der Mischung der Rot-, Grün- und Blauanteile `r`, `g` und `b`, die Werte im Bereich von `0` bis `255` annehmen können. Z.B. liefert das Tripel `(0,0,0)` die Farbe Schwarz, `(255,255,255)` die Farbe Weiß, `(255,0,0)` Rot, `(0,255,0)` Grün und `(0,0,255)` Blau.

`Color getColor()`
liefert die aktuelle Farbe.

Font

Mit der Klasse `Font` werden Schriftarten festgelegt. Der Konstruktor

`Font(String name, int style, int size)`

erzeugt ein neues `Font`-Objekt, wobei `name` den Namen der gewünschten Schriftart ("Dialog", "DialogInput", "SansSerif", "Serif", "Monospaced" oder "Symbol"), `style` die Schriftauszeichnung `Font.BOLD` (fett), `Font.ITALIC` (kursiv) und `Font.PLAIN` (normal) und `size` die Punktgröße angibt. `Font.BOLD` kann durch Addition mit `Font.ITALIC` kombiniert werden.

`void setFont(Font f)`
setzt die Schriftart, die dann von allen folgenden Operationen, die Text ausgeben, verwendet wird.

`Font getFont()`
liefert die aktuelle Schriftart.

`setFont` und `getFont` gibt es auch als `Component`-Methoden.

Zeichen-methoden

`void drawString(String s, int x, int y)`
zeichnet einen Schriftzug, wobei `(x,y)` das linke Ende der Grundlinie des ersten Zeichens von `s` ist.

```
void drawLine(int x1, int y1, int x2,
  int y2)
```
zeichnet eine Linie von `(x1,y1)` bis `(x2, y2)`.

```
void drawRect(int x, int y, int width,
  int height)
```
zeichnet ein Rechteck mit der Breite `width+1`, der Höhe `height+1` und der linken oberen Ecke `(x,y)`. Die Koordinaten der Eckpunkte sind also `x` und `x+width` bzw. `y` und `y+height`.

`void drawPolygon(int[] x, int[] y, int n)`
zeichnet ein geschlossenes Polygon mit den Eckpunktkoordinaten `x` und `y`. `n` ist die Anzahl der Eckpunkte.

```
void drawOval(int x, int y, int width,
  int height)
```
zeichnet ein Oval innerhalb des durch die Argumente spezifizierten Rechtecks. Ein Kreis entsteht, wenn `width` und `height` gleich sind.

```
void drawArc(int x, int y, int width,
  int height, int start, int arc)
```
zeichnet einen Kreis- bzw. Ellipsenbogen innerhalb des spezifizierten Rechtecks vom Startwinkel `start` bis zum Winkel `start + arc`. Positive Winkel (0 - 360) bedeuten eine Drehung gegen den Uhrzeigersinn, ein Winkel von 0 Grad bezeichnet die 3-Uhr-Position.

Rechtecke, Polygone, Ovale und Kreis- bzw. Ellipsenbögen können auch im Füllmodus mit der eingestellten Farbe gezeichnet werden. Die Methodennamen beginnen dann mit `fill` statt `draw`.

Vorder- und Hintergrundfarbe

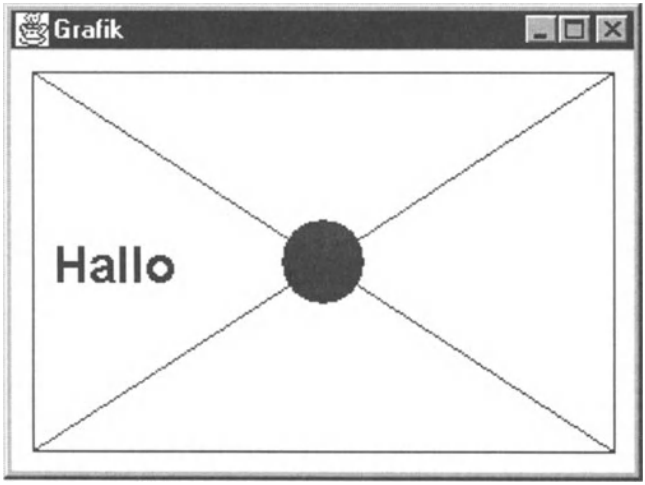
Die folgenden `Component`-Methoden setzen bzw. liefern die Vorder- bzw. Hintergrundfarbe einer Komponente:

```
void setForeground(Color c)
void setBackground(Color c)
Color getForeground()
Color getBackground()
```

Programm 8.2

```
// Datei: Fenster2.java
import java.awt.*;
import javax.swing.*;

public class Fenster2 extends JPanel {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Grafik");
      frame.getContentPane().add(new Fenster2());
      frame.pack();
      frame.setVisible(true);
   }

   public Fenster2() {
      setBackground(Color.white);
      setForeground(Color.red);
   }

   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      g.drawRect(10, 10, 280, 180);
      g.drawLine(10, 190, 290, 10);
      g.drawLine(10, 10, 290, 190);
      g.setColor(Color.black);
      g.setFont(new Font("SansSerif", Font.BOLD, 24));
      g.drawString("Hallo", 20, 110);
      g.setColor(Color.red);
      g.fillOval(130, 80, 40, 40);
   }

   public Dimension getPreferredSize() {
      return new Dimension(300, 200);
   }
}
```

Bild 8.2:
Grafik im Fenster



Das Programm enthält noch keine Funktionalität, um es mit dem Schließen des Fensters ordnungsgemäß zu beenden. Deshalb muss das Programm vom Benutzer abgebrochen werden, z.B. durch Eingabe von `Strg+C` im DOS-Fenster, aus dem es gestartet wurde.

Programm 8.3 erzeugt einen Kreis mit acht Segmenten, deren Farben über einen Zufallsgenerator bestimmt werden.

Programm 8.3

```
// Datei: Fenster3.java
import java.awt.*;
import javax.swing.*;

public class Fenster3 extends JPanel {
   private int start;
   private int[] f1 = new int[8];
   private int[] f2 = new int[8];
   private int[] f3 = new int[8];

   public static void main(String[] args) {
      JFrame frame = new JFrame("Farbrad");
      frame.getContentPane().add(new Fenster3());
```

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

    public Fenster3() {
        start = 0;
        for (int i = 0; i < 8; ++i) {
            f1[i] = (int)(Math.random() * 256);
            f2[i] = (int)(Math.random() * 256);
            f3[i] = (int)(Math.random() * 256);
        }
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int i = 0; i < 8; ++i) {
            Color c = new Color(f1[i], f2[i], f3[i]);
            g.setColor(c);
            g.fillArc(0, 0, 300, 300, start, 45);
            start += 45;
        }
    }

    public Dimension getPreferredSize() {
        return new Dimension(300, 300);
    }
}
```

8.3 GUI-Komponenten und Ereignisbehandlung

Die Kommunikation zwischen einem Benutzer und einem Anwendungsprogramm mit einer grafischen Oberfläche basiert auf einem *Event-Modell*, das neben einzelnen GUI-Komponenten in diesem Kapitel behandelt wird.

8.3.1 Das Event-Modell

Bisher haben wir das mit einem Fenster verbundene Programm nur gewaltsam abbrechen können. Nun wollen wir durch Ausführung des Befehls *Schließen* im Systemmenü des Fensters bzw. durch einen Mausklick auf das entsprechende

Symbol oder durch Drücken der *ESC-Taste* das Programm ordnungsgemäß beenden.

Ereignisse, Ereignisquellen und -empfänger

Das Paket `java.awt.event` enthält wesentliche Klassen, die die Reaktion auf derartige Ereignisse realisieren, und ist somit in jede Klasse einzubinden, die Ereignisse behandeln soll.

Eine Aktion, wie z.B. das Klicken auf einen Menüpunkt, löst ein *Ereignis* (*Event*) aus, auf das das Programm in bestimmter Weise reagieren kann. Ein Ereignis ist immer mit einer *Ereignisquelle* verbunden (z.B. einem Fenster oder einem Button), die das Ereignis ausgelöst hat. Ereignisse sind Objekte bestimmter Klassen. In unserem Beispiel handelt es sich um Objekte der Klassen `WindowEvent` und `KeyEvent`.

Die Reaktion auf Ereignisse erfolgt in so genannten *Ereignisempfängern* (*Listener*), die sich bei der Ereignisquelle registrieren müssen, wenn sie entsprechende Ereignisse mitgeteilt bekommen wollen.

Im Beispiel sind die Ereignisempfänger Objekte, die das Interface `WindowListener` bzw. `KeyListener` implementieren. Die zugehörigen Registrierungsmethoden, die in der Ereignisquelle aufgerufen werden müssen, sind:
`void addWindowListener(WindowListener l)` bzw.
`void addKeyListener(KeyListener l)`.

Die Mitteilung eines Ereignisses besteht im Aufruf der passenden Methode des Listener-Objekts, die als Argument die Referenz auf das entsprechende Ereignisobjekt erhält.

Alle Ereignisklassen besitzen eine Methode, mit deren Hilfe das Objekt ermittelt werden kann, das das Ereignis ausgelöst hat: `Object getSource()`.

Im Programm 8.4 ist das Objekt `frame` sowohl Ereignisquelle als auch -empfänger. Die Klasse `Schliessen1` implementiert die Interfaces `WindowListener` und `KeyListener`. Die einzig relevanten Methoden für die Reaktion auf die Ereignisse in diesem Beispiel sind `windowClosing` und `keyPressed`. Die anderen Methoden der beiden Interfaces haben deshalb leere Methodenrümpfe. Die `KeyEvent-`

Methode `getKeyCode` gibt den Code der gedrückten Taste zurück, der mit `KeyEvent.VK_ESCAPE` verglichen wird.

Programm 8.4

```
// Datei Schliessen1.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Schliessen1 extends JFrame implements
   WindowListener, KeyListener {
   public static void main(String[] args) {
      new Schliessen1();
   }

   public Schliessen1() {
      super("Demo");
      setSize(300, 200);
      setVisible(true);
      addWindowListener(this);
      addKeyListener(this);
   }

   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }

   public void windowActivated(WindowEvent e) { }
   public void windowClosed(WindowEvent e) { }
   public void windowDeactivated(WindowEvent e) { }
   public void windowDeiconified(WindowEvent e) { }
   public void windowIconified(WindowEvent e) { }
   public void windowOpened(WindowEvent e) { }

   public void keyPressed(KeyEvent e) {
      if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
         System.exit(0);
      }
   }

   public void keyReleased(KeyEvent e) { }
   public void keyTyped(KeyEvent e) { }
}
```

Adapterklasse

Eine *Adapterklasse* ist eine Klasse, die ein vorgegebenes Interface mit leeren Methodenrümpfen implementiert.

Die Adapterklassen `WindowAdapter` und `KeyAdapter` werden im Programm 8.5 genutzt, um nur die gewünschten Methoden in Subklassen zu überschreiben und damit Schreibaufwand für die nicht relevanten Methoden des Listener-Interface zu sparen. Diese Subklassen sind als *innere Klassen* (Instanzklassen) realisiert.

Programm 8.5

```
// Datei: Schliessen2.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Schliessen2 extends JFrame {
   public static void main(String[] args) {
      new Schliessen2();
   }

   public Schliessen2() {
      super("Demo");
      setSize(300, 200);
      setVisible(true);
      addWindowListener(new MyWindowListener());
      addKeyListener(new MyKeyListener());
   }

   class MyWindowListener extends WindowAdapter {
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      }
   }

   class MyKeyListener extends KeyAdapter {
      public void keyPressed(KeyEvent e) {
         if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
            System.exit(0);
         }
      }
   }
}
```

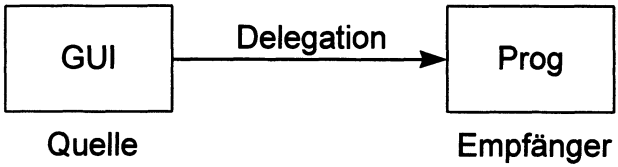
Wenn sehr wenig Code für die Ereignisempfänger benötigt wird, empfehlen sich auch *anonyme Klassen.*

Programm 8.6

```
// Datei: Schliessen3.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Schliessen3 extends JFrame {
   public static void main(String[] args) {
      new Schliessen3();
   }

   public Schliessen3() {
      super("Demo");
      setSize(300, 200);
      setVisible(true);

      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      } );

      addKeyListener(new KeyAdapter() {
         public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
               System.exit(0);
            }
         }
      } );
   }
}
```

Die Verarbeitung der Ereignisse wird also von der Ereignisquelle an die Ereignisempfänger delegiert. Das Event-Modell erlaubt damit eine klare Trennung zwischen Programmcode zur Gestaltung der grafischen Oberfläche und solchem zur Implementierung der Anwendungsfunktionalität. Eine solche Modularisierung ist insbesondere in großen Programmen wichtig.

Bild 8.3: Delegation der Ereignisverarbeitung



Programm 8.7 enthält die Klassen `Prog` (Anwendungsfunktionalität) und GUI (Gestaltung der Oberfläche). Das in der Methode `main` erzeugte `Prog`-Objekt wird an das `GUI`-Objekt zur Registrierung weitergereicht.

Programm 8.7

```
// Datei: Prog.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Prog implements WindowListener, KeyListener {
   public static void main(String[] args) {
      Prog anwendung = new Prog();
      GUI gui = new GUI(anwendung);
   }

   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
   ...

   public void keyPressed(KeyEvent e) {
      if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
         System.exit(0);
      }
   }
   ...
}
```

```
class GUI extends JFrame {
   private Prog anw;

   public GUI(Prog anw) {
      super("Demo");
      this.anw = anw;
      setSize(300, 200);
      setVisible(true);
      addWindowListener(this.anw);
      addKeyListener(this.anw);
   }
}
```

Die Tabellen 8.1 und 8.2 fassen die bisher behandelten Ereignisse zusammen.

Tabelle 8.1: WindowEvent

Ereignisklasse	**`WindowEvent`**
Ereignisquellen	`Window` und Subklassen
Registrierungsmethode	`addWindowListener`
Listener-Interface	`WindowListener`
Adapterklasse	`WindowAdapter`

Listener-Methoden	Beschreibung
`windowActivated`	Fenster wurde aktiviert
`windowClosed`	Fenster wurde geschlossen
`windowClosing`	Fenster wird geschlossen
`windowDeactivated`	Fenster wurde deaktiviert
`windowDeiconified`	Fenster wurde wieder hergestellt
`windowIconified`	Fenster wurde auf Symbolgröße verkleinert
`windowOpened`	Fenster wurde geöffnet

Tabelle 8.2: KeyEvent

Ereignisklasse	**`KeyEvent`**
Ereignisquellen	`Component` und Subklassen
Registrierungsmethode	`addKeyListener`
Listener-Interface	`KeyListener`
Adapterklasse	`KeyAdapter`

Listener-Methoden	Beschreibung
keyPressed	Taste wurde gedrückt
keyReleased	Taste wurde losgelassen
keyTyped	Taste wurde gedrückt und losgelassen

Zu allen Registrierungsmethoden `addXYZListener` existieren auch die dazu passenden Deregistrierungsmethoden `removeXYZListener`.

8.3.2 Buttons

Die Klasse `JButton` implementiert einen einfachen Button.

JButton

Konstruktoren sind:

```
JButton()
JButton(String text)
JButton(Icon icon)
JButton(String text, Icon icon)
```

Sie erzeugen einen Button ohne oder mit Beschriftung bzw. Icon.

Icon

`Icon` ist ein Interface, das z.B. von der Klasse `ImageIcon` implementiert wird.

Ein `ImageIcon`-Objekt kann mit den Konstruktoren `ImageIcon(String datei)` und `ImageIcon(URL url)` aus einer Bilddatei (gif- oder jpeg-Format) oder im Netzwerk (siehe Kapitel 9) erzeugt werden.

Tooltip

Mit der `JComponent`-Methode
`void setToolTipText(String text)`
wird eine Komponente mit einem so genannten *Tooltip* ausgestattet, der beim Bewegen des Mauszeigers über diese Komponente angezeigt wird.

Border

Die `JComponent`-Methode
`void setBorder(Border border)`
versieht eine Komponente mit einem Rand. Das Interface `Border` stammt aus dem Paket `javax.swing.border`.

Ränder können durch Aufruf von Methoden der Klasse `BorderFactory` erzeugt werden, z.B.

`static Border createBevelBorder(int type)`
erzeugt einen 3D-Rahmen, als `type` kann die Konstante `BevelBorder.LOWERED` oder `BevelBorder.RAISED` angegeben werden.

```
static Border createLineBorder(Color c, int
  thickness)
```
erzeugt einen einfachen, farbigen Rahmen mit vorgegebener Dicke (in Pixel).

AbstractButton

Die Superklasse `AbstractButton` bietet die folgenden Methoden:

`void setText(String text)`
legt die Beschriftung fest.

`String getText()`
liefert die Beschriftung.

`void setIcon(Icon icon)`
legt das Icon fest.

`Icon getIcon()`
liefert das Icon.

```
void setHorizontalTextPosition(int pos)
void setVerticalTextPosition(int pos)
```
legt die horizontale bzw. vertikale Position des Textes relativ zum Icon fest. Dabei werden für die horizontale Ausrichtung die Konstanten `LEFT`, `CENTER` oder `RIGHT` und für die vertikale Ausrichtung `TOP`, `CENTER` oder `BOTTOM` übergeben.

`void setActionCommand(String cmd)`
setzt den Kommandonamen des dem Button zugeordneten Action-Events. Als Voreinstellung ist das die Beschriftung des Buttons.

`String getActionCommand()`
liefert den Kommandonamen.

Ereignis-behandlung

Wird ein Button gedrückt, so sendet er ein Action-Event an seine Ereignisempfänger.

`void addActionListener(ActionListener l)`
registriert das Ereignisempfänger-Objekt `l`, das das Interface `ActionListener` implementiert.

`void actionPerformed(ActionEvent e)`
wird im Ereignisempfänger aufgerufen, wenn ein Action-Event auftritt.

Die Methode `String getActionCommand()` der Klasse `ActionEvent` liefert den Kommandonamen. Bei mehreren Buttons kann man hiermit feststellen, welcher Button gedrückt wurde.

Programm 8.8 demonstriert die Funktionsweise eines Buttons. Mit dem Drücken ändert der Button seinen Rahmen.

Programm 8.8

```
// Datei: ButtonTest1.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class ButtonTest1 extends JPanel
   implements ActionListener {
   private JButton button;
   private Border raised, lowered;
   private boolean on;

   public static void main(String[] args) {
      JFrame frame = new JFrame("Button");
      frame.getContentPane().add(new ButtonTest1());
      frame.setSize(300, 200);
      frame.setVisible(true);
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      } );
   }
```

```
    public ButtonTest1() {
        raised = BorderFactory.createBevelBorder(
            BevelBorder.RAISED);
        lowered = BorderFactory.createBevelBorder(
            BevelBorder.LOWERED);
        Icon ic = new ImageIcon("cross.gif");
        button = new JButton(" Test ", ic);
        button.setBorder(raised);
        on = false;
        button.setToolTipText("Bitte klicken");
        button.addActionListener(this);
        add(button);
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println(e.getActionCommand());
        if (on) {
            button.setBorder(raised);
            on = false;
        }
        else {
            button.setBorder(lowered);
            on = true;
        }
    }
}
```

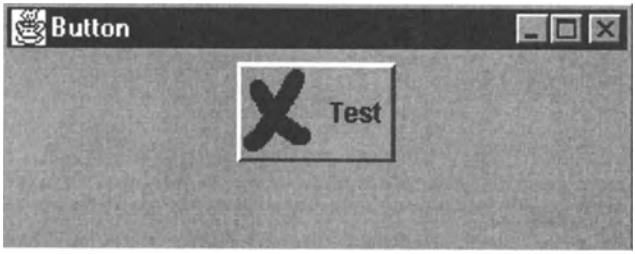


Bild 8.4: JButton

Die speziellen Buttons `JCheckBox` und `JRadioButton` können jeweils den Zustand "selektiert" oder "nicht selektiert" annehmen.

JCheckBox und JRadioButton

Neben den zu `JButton` analogen Konstruktoren existieren für `JCheckBox` und `JRadioButton` drei weitere mit einem zusätzlichen Parameter vom Typ `boolean`, der angibt, ob der Anfangszustand "selektiert" (`true`) eingestellt sein soll.

Alle Methoden der Klasse `AbstractButton` stehen zur Verfügung. Hier zwei weitere Methoden dieser Klasse:

`void setSelected(boolean state)`
setzt den Zustand des Buttons.

`boolean isSelected()`
liefert den Zustand des Buttons.

ButtonGroup

Mehrere Objekte vom Typ `JRadioButton` werden meist mit einem `ButtonGroup`-Objekt zu einer Gruppe zusammengefasst. Von den Buttons, die der gleichen Gruppe angehören, kann nur einer den Zustand "selektiert" haben. Das Selektieren eines neuen Buttons der Gruppe führt automatisch zum Deselektieren des bisher selektierten Buttons.

Ein Objekt der Klasse `ButtonGroup` wird mit dem Standardkonstruktor erzeugt.

`void add(AbstractButton b)`
nimmt den Button `b` in die Gruppe auf.

Ereignisbehandlung

Neben der Erzeugung eines Action-Events beim Drücken auf einen Button löst das Selektieren und Deselektieren eines `JCheckBox`- bzw. `JRadioButton`-Objekts ein Item-Event aus.

`void addItemListener(ItemListener l)`
registriert das Ereignisempfänger-Objekt `l`, das das Interface `ItemListener` implementiert.

`void itemStateChanged(ItemEvent e)`
wird im Ereignisempfänger aufgerufen, wenn ein Item-Event auftritt.

Die `ItemEvent`-Methode
`int getStateChange()`
liefert die Klassenkonstante `SELECTED`, wenn ausgewählt wurde, oder `DESELECTED`, wenn die Auswahl aufgehoben wurde.

Programm 8.9

```
// Datei: ButtonTest2.java
...

public class ButtonTest2 extends JPanel
   implements ItemListener {

   public static void main(String[] args) { ... }

   public ButtonTest2() {
      JCheckBox cb = new JCheckBox("Opt 0");
      cb.addItemListener(this);
      add(cb);

      ButtonGroup bg = new ButtonGroup();
      JRadioButton[] rb = new JRadioButton[3];
      for (int i = 0; i < rb.length; i++) {
         rb[i] = new JRadioButton("Opt " + (i+1));
         bg.add(rb[i]);
         add(rb[i]);
         rb[i].addItemListener(this);
      }
   }

   public void itemStateChanged(ItemEvent e) {
      if (e.getStateChange() == ItemEvent.SELECTED)
         System.out.print("SELECTED: ");
      else
         System.out.print("DESELECTED: ");
      Object obj = e.getSource();
      System.out.println(((AbstractButton)obj).getText());
   }
}
```

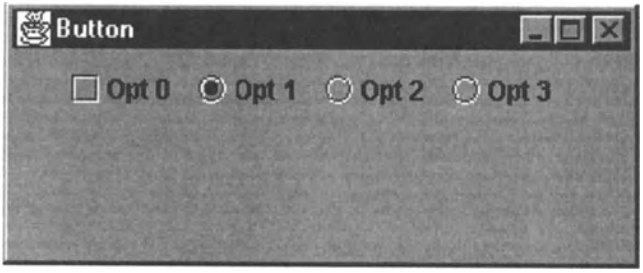


Bild 8.5: JCheckBox und JRadioButton

8.3.3 Labels

Die Klasse `JLabel` kann einen kurzen Text anzeigen.

JLabel

Konstruktoren sind:

```
JLabel()
JLabel(String text)
JLabel(String text, int align)
JLabel(Icon icon)
JLabel(Icon icon, int align)
JLabel(String text, Icon icon, int align)
```

Sie erzeugen ein Label ohne oder mit Beschriftung bzw. Icon. `align` bestimmt die horizontale Ausrichtung des Inhalts auf der verfügbaren Fläche mit einer der Konstanten `LEFT`, `CENTER` oder `RIGHT`.

`void setText(String text)`
legt den Text des Labels fest.

`String getText()`
liefert den Text des Labels.

`void setIcon(Icon icon)`
legt das Icon fest.

`Icon getIcon()`
liefert das Icon.

`void setHorizontalAlignment(int align)`
`void setVerticalAlignment(int align)`
legt die horizontals bzw. vertikale Ausrichtung des Inhalts auf der verfügbaren Fläche fest. Dabei werden für die horizontale Ausrichtung die Konstanten `LEFT`, `CENTER` oder `RIGHT` und für die vertikale Ausrichtung `TOP`, `CENTER` oder `BOTTOM` übergeben.

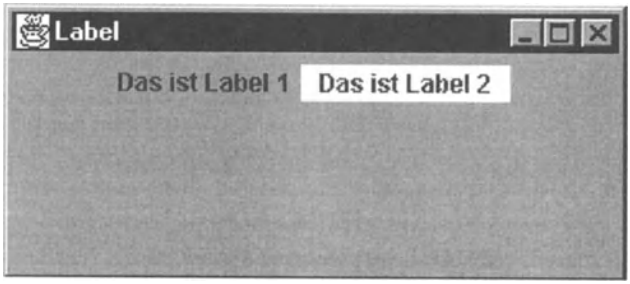
`void setHorizontalTextPosition(int pos)`
`void setVerticalTextPosition(int pos)`
legt die horizontale bzw. vertikale Position des Textes relativ zum Icon fest. Dabei werden für die horizontale Ausrichtung die Konstanten `LEFT`, `CENTER` oder `RIGHT` und für die vertikale Ausrichtung `TOP`, `CENTER` oder `BOTTOM` übergeben.

Die `JComponent`-Methode
`void setOpaque(boolean b)`
gibt der Komponente einen deckenden (`true`) oder durchsichtigen (`false`) Hintergrund.

Programm 8.10

```
// Datei: LabelTest.java
...

public class LabelTest extends JPanel {
   public static void main(String[] args) { ... }

   public LabelTest() {
      JLabel l1 = new JLabel("Das ist Label 1");
      add(l1);
      JLabel l2 = new JLabel("   Das ist Label 2   ");
      l2.setOpaque(true);
      l2.setBackground(Color.white);
      add(l2);
   }
}
```

Bild 8.6:
JLabel



8.3.4 Scrollbalken und spezielle Container

Die Klasse `JScrollBar` implementiert einen horizontalen oder vertikalen Scrollbalken, mit dem ganzzahlige Werte aus einem vorgegebenen Wertebereich eingestellt werden können.

JScrollBar

```
JScrollBar()
JScrollBar(int orientation)
JScrollBar(int orientation, int value,
  int visible, int min, int max)
```

erzeugen jeweils einen Scrollbalken.

Der parameterlose Konstruktor erzeugt einen vertikalen Scrollbalken. `orientation` legt die Ausrichtung fest. Mögliche Werte sind die Konstanten `HORIZONTAL` und `VERTICAL`. `value` ist der Anfangswert. `visible` bestimmt die Breite bzw. Höhe des Schiebers. `min` und `max` legen die Grenzen des Wertebereichs fest. Dabei ist der größte einstellbare Wert: `max - visible`.

Bei den ersten beiden Konstruktoren gelten die Werte: `value` = 0, `visible` = 10, `min` = 0 und `max` = 100.

`void setUnitIncrement(int v)`
stellt den Wert ein, der zum aktuellen Wert addiert oder von ihm subtrahiert wird, wenn eine Pfeilschaltfläche betätigt wird.

`void setBlockIncrement(int v)`
stellt den Wert ein, der zum aktuellen Wert addiert oder von ihm subtrahiert wird, wenn auf eine freie Fläche zwischen Schieber und Pfeil geklickt wird.

`void setOrientation(int orientation)`
legt die Ausrichtung fest. Mögliche Werte sind die Konstanten `HORIZONTAL` und `VERTICAL`.

`void setVisibleAmount()`
setzt die Breite bzw. Höhe des Schiebers.

`void setMinimum(int v)`
`void setMaximum(int v)`
setzt Minimum bzw. Maximum des Wertebereichs (siehe Erläuterungen zum Konstruktor).

`void setValue(int v)`
setzt den aktuellen Wert.

Zu diesen set-Methoden existieren auch die entsprechenden get-Methoden zur Abfrage der Werte.

```
void setValues(int value, int visible,
  int min, int max)
```
setzt die vier Werte des Scrollbalkens (siehe Erläuterungen zum Konstruktor).

Ereignisbehandlung

Bei Änderungen im Scrollbalken wird ein Adjustment-Event ausgelöst.

```
void addAdjustmentListener(
  AdjustmentListener l)
```
registriert das Ereignisempfänger-Objekt `l`, das das Interface `AdjustmentListener` implementiert.

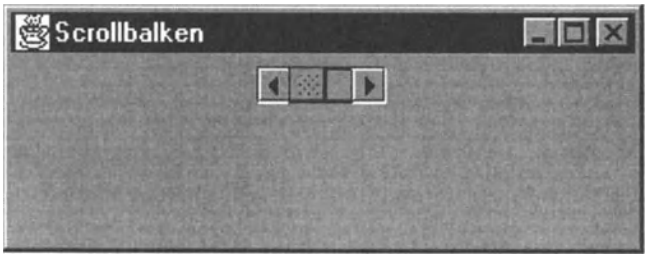
```
void adjustmentValueChanged(
  AdjustmentEvent e)
```

wird im Ereignisempfänger aufgerufen, wenn ein Adjustment-Event auftritt.

Programm 8.11

```
// Datei: ScrollTest.java
...

public class ScrollTest extends JPanel
   implements AdjustmentListener {
   private JScrollBar sb;

   public static void main(String[] args) { ... }

   public ScrollTest() {
      sb = new JScrollBar(JScrollBar.HORIZONTAL,
         0, 10, 0, 110);
      sb.addAdjustmentListener(this);
      add(sb);

      sb.setBlockIncrement(10);
      sb.setUnitIncrement(5);
      System.out.println("Value: " + sb.getValue());
      System.out.println("VisibleAmount: " +
         sb.getVisibleAmount());
      System.out.println("Minimum: " + sb.getMinimum());
      System.out.println("Maximum: " + sb.getMaximum());
      System.out.println("UnitIncrement: " +
         sb.getUnitIncrement());
      System.out.println("BlockIncrement: " +
         sb.getBlockIncrement());
   }

   public void adjustmentValueChanged(AdjustmentEvent e) {
      System.out.println(sb.getValue());
   }
}
```

Bild 8.7:
JScrollBar



JScrollPane

Die Klasse `JScrollPane` ist ein Container mit horizontalem und vertikalem Scrollbalken, der nur eine Komponente aufnimmt. Mit Hilfe der Scrollbalken kann die Komponente im Bildausschnitt verschoben werden, falls die Komponente aufgrund ihrer Größe nicht vollständig sichtbar ist.

Konstruktoren:

```
JScrollPane()
JScrollPane(int v, int h)
JScrollPane(Component c)
JScrollPane(Component c, int v, int h)
```

Der Standardkonstruktor erzeugt einen leeren Container, wobei die Scrollbalken nur bei Bedarf angezeigt werden. `c` ist die darzustellende Komponente. Die Konstanten `v` und `h` legen fest, ob und wann die Scrollbalken angezeigt werden sollen.

Gültige Werte für `v` sind:
`VERTICAL_SCROLLBAR_AS_NEEDED`
`VERTICAL_SCROLLBAR_NEVER`
`VERTICAL_SCROLLBAR_ALWAYS`

Gültige Werte für `h` sind:
`HORIZONTAL_SCROLLBAR_AS_NEEDED`
`HORIZONTAL_SCROLLBAR_NEVER`
`HORIZONTAL_SCROLLBAR_ALWAYS`

`JScrollBar getHorizontalScrollBar()`
`JScrollBar getVerticalScrollBar()`
liefern den horizontalen bzw. vertikalen Scrollbalken.

`JViewport getViewport()`
liefert den Anzeigebereich als Objekt der Klasse `JViewport`.

Wichtige `JViewport`-Methoden sind:

`Dimension getViewSize()`
liefert die Größe des Anzeigebereichs.

`Point getViewPosition()`
liefert die Position der Komponente, die an der Position `(0,0)` des Anzeigebereichs angezeigt wird.

`void setViewPosition(Point p)`
setzt die Position der Komponente, so dass diese der Position `(0,0)` des Anzeigebereichs entspricht.

`void setView(Component c)`
setzt die Komponente, die angezeigt werden soll.

Programm 8.12 implementiert ein `JScrollPane`-Objekt. Als Komponente wird eine Zeichnung (Subklasse von `JPanel`) verwendet. Die `Container`-Methode `setLayout` bestimmt für das Panel einen speziellen Layout-Manager, auf den an späterer Stelle eingegangen wird.

Programm 8.12

```
// Datei: ScrollPaneTest.java
...

public class ScrollPaneTest extends JPanel
   implements AdjustmentListener {
   private JScrollPane sp;

   public static void main(String[] args) { ... }

   public ScrollPaneTest() {
      setLayout(new BorderLayout());
      sp = new JScrollPane(new Zeichnung());
      JScrollBar hor = sp.getHorizontalScrollBar();
      JScrollBar ver = sp.getVerticalScrollBar();
```

```
        hor.addAdjustmentListener(this);
        ver.addAdjustmentListener(this);
        add(sp);
    }

    public void adjustmentValueChanged(AdjustmentEvent e) {
        Point p = sp.getViewport().getViewPosition();
        System.out.println("(" + p.x + "," + p.y + ")");
    }
}

class Zeichnung extends JPanel {
    public Zeichnung() {
        setBackground(Color.white);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawRect(10, 10, 400, 300);
        g.drawOval(20, 20, 380, 280);
    }

    public Dimension getPreferredSize() {
        return new Dimension(450, 350);
    }
}
```

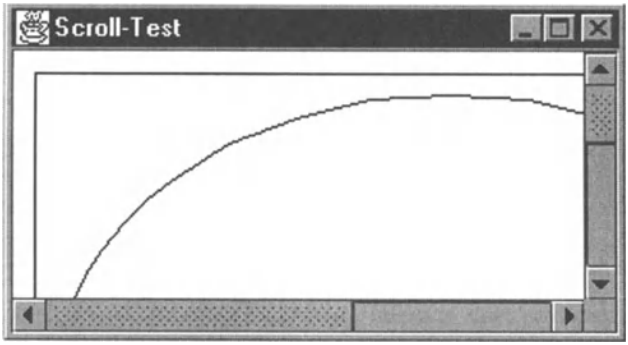


Bild 8.8: JScrollPane

JTabbedPane

`JTabbedPane` ist ein Container aus mehreren so genannten Karten, die wie in einem Stapel übereinander gelegt und abwechselnd sichtbar gemacht werden können (Karteiregister).

Konstruktoren:

```
JTabbedPane()
JTabbedPane(int placement)
```

erzeugen jeweils ein leeres Karteiregister. `placement` bestimmt den Ort der Registerlaschen; gültige Werte sind die Konstanten `TOP`, `BOTTOM`, `LEFT` oder `RIGHT`.

Folgende Methoden fügen neue Karten ein:

```
void addTab(String title, Component c)
void addTab(String title, Icon icon,
  Component c)
void addTab(String title, Icon icon,
  Component c, String tip)
```

Bei den beiden letzten Methoden kann auch der Titel oder das Icon `null` sein. `tip` ist ein Tooltip-Text.

```
void insertTab(String title, Icon icon,
  Component c, String tip, int pos)
```

fügt die Komponente `c` an der Position `pos` ein. Der Titel oder das Icon kann auch `null` sein.

Weitere Methoden sind:

```
void setSelectedIndex(int pos)
```

selektiert die Karte an der Position `pos`.

```
int getSelectedIndex()
```

liefert die Position der selektierten Karte.

```
void setTitleAt(int pos, String title)
```

setzt den Titel der Karte an der Position `pos`.

```
String getTitleAt(int pos)
```

liefert den Titel der Karte an der Position `pos`.

`void setIconAt(int pos, Icon icon)`
setzt das Icon der Karte an der Position `pos`.

`Icon getIconAt(int pos)`
liefert das Icon der Karte an der Position `pos`.

`void setEnabledAt(int pos, boolean b)`
deaktiviert die Karte an der Position `pos`, wenn `b` den Wert `false` hat.

`boolean isEnabledAt(int pos)`
liefert den Wert `true`, wenn die Karte an der Position `pos` aktiviert ist.

Ereignis-behandlung

Die Selektion einer Karte löst ein Change-Event aus.

`void addChangeListener(ChangeListener l)`
registriert das Ereignisempfänger-Objekt `l`, das das Interface `ChangeListener` aus dem Paket `javax.swing.event` implementiert.

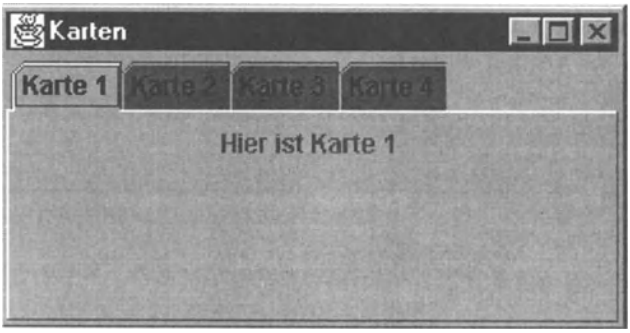
`void stateChanged(ChangeEvent e)`
wird im Ereignisempfänger aufgerufen, wenn ein Change-Event auftritt.

Programm 8.13

```
// Datei: KartenTest.java
...

public class KartenTest extends JPanel
   implements ChangeListener {
   private JTabbedPane tp;

   public static void main(String[] args) { ... }

   public KartenTest() {
      setLayout(new BorderLayout());
      tp = new JTabbedPane();
      for (int i = 0; i < 4; i++)
         tp.addTab("Karte " + (i+1),
            erzeugeKarte("Hier ist Karte " + (i+1)));
      tp.addChangeListener(this);
      add(tp);
   }
```

```
    private JPanel erzeugeKarte(String text) {
        JPanel p = new JPanel();
        p.add(new JLabel(text));
        return p;
    }

    public void stateChanged(ChangeEvent e) {
        System.out.println("Karte " +
            (tp.getSelectedIndex() + 1) + " selektiert");
    }
}
```

Bild 8.9: JTabbedPane



8.3.5 Textkomponenten

Die Klasse `JTextField` dient zur Eingabe und Anzeige einer Textzeile.

Konstruktoren:

```
JTextField()
JTextField(int cols)
JTextField(String text)
JTextField(String text, int cols)
```

erzeugen jeweils ein neues Textfeld, das ggf. den Text `text` enthält und `cols` Zeichen breit ist.

`void setColumns(int cols)`
setzt die Spaltenbreite des Textfeldes.

`int getColumns()`
liefert die Spaltenbreite des Textfeldes.

`void setHorizontalAlignment(int align)`
legt die horizontale Ausrichtung des Textes fest. Dabei werden die Konstanten `LEFT` (Voreinstellung), `CENTER` oder `RIGHT` übergeben.

JTextComponent

Die Superklasse `JTextComponent` (`javax.swing.text`) bietet die folgenden Methoden:

`void setText(String text)`
setzt den Text `text`.

`String getText()`
liefert den Text des Textfeldes.

`void selectAll()`
markiert den kompletten Text im Textfeld.

`String getSelectedText()`
liefert den markierten Text.

`int getSelectionStart()`
liefert die Position des ersten markierten Zeichens.

`int getSelectionEnd()`
liefert die Position des letzten markierten Zeichens + 1.

`void setCaretPosition(int pos)`
setzt die Position der Einfügemarke.

`int getCaretPosition()`
liefert die Position der Einfügemarke.

`void moveCaretPosition(int pos)`
bewegt die Einfügemarke zur Position `pos`. Der Text ab der beim letzten Aufruf von `setCaretPosition` gesetzten Position bis `pos - 1` wird markiert.

`void copy()`
kopiert den markierten Text in die Zwischenablage.

`void cut()`
kopiert den markierten Text in die Zwischenablage und löscht ihn im Original.

`void paste()`
ersetzt den markierten Text durch den Inhalt der Zwischenablage bzw. fügt diesen Inhalt an der aktuellen Position der Einfügemarke ein.

`void setEditable(boolean b)`
setzt das Textfeld als editierbar, falls `b` den Wert `true` hat. Hat `b` den Wert `false`, so kann das Textfeld nicht editiert werden.

`boolean isEditable()`
liefert `true`, wenn das Textfeld editiert werden kann, sonst `false`.

JPasswordField

Die Subklasse `JPasswordField` mit den zu `JTextField` analogen Konstruktoren implementiert ein Passwortfeld, in dem anstelle eines eingegebenen Zeichens ein "Echo-Zeichen" angezeigt wird, standardmäßig das Zeichen `'*'`.

`void setEchoChar(char c)`
setzt das Echo-Zeichen auf `c`.

`char getEchoChar()`
liefert das eingestellte Echo-Zeichen.

`char[] getPassword()`
liefert den Inhalt des Passwortfeldes.

Ereignis-behandlung

Wird die *Return-Taste* innerhalb des Textfeldes gedrückt, so erzeugt das Textfeld ein Action-Event. Ein Textfeld erlaubt die Registrierung eines `ActionListener`-Objekts.

Die Methode `String getActionCommand()` der Klasse `ActionEvent` liefert den Inhalt des Textfeldes.

Programm 8.14

```
// Datei: TextTest1.java
...

public class TextTest1 extends JPanel
   implements ActionListener {
   public static void main(String[] args) { ... }

   public TextTest1() {
      JTextField t1 = new JTextField("Hallo", 10);
      t1.addActionListener(this);
      add(t1);

      JTextField t2 = new JTextField("4711", 10);
      t2.setHorizontalAlignment(JTextField.RIGHT);
      t2.addActionListener(this);
      add(t2);

      JPasswordField t3 = new JPasswordField(10);
      t3.addActionListener(this);
      add(t3);
   }

   public void actionPerformed(ActionEvent e) {
      System.out.println(e.getActionCommand());
   }
}
```

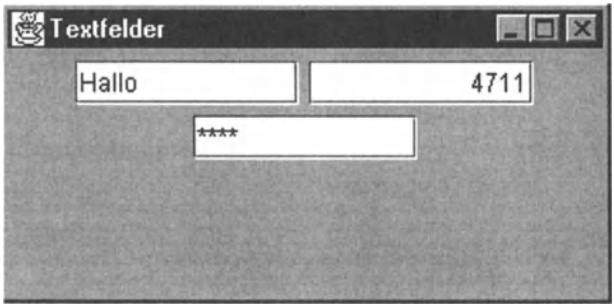


Bild 8.10: JTextField und JPasswordField

JTextArea

Die Klasse `JTextArea` (Subklasse von `JTextComponent`) dient zur Eingabe und Anzeige mehrerer Textzeilen.

Konstruktoren:

```
JTextArea()
JTextArea(int rows, int cols)
JTextArea(String text)
JTextArea(String text, int rows, int cols)
```

erzeugen jeweils ein neues Textfeld, das ggf. den Text `text` enthält und `rows` sichtbare Zeilen sowie `cols` sichtbare Spalten hat, falls diese beiden Werte angegeben sind.

`void setRows(int rows)`
setzt die Anzahl Zeilen des Textbereichs.

`int getRows()`
liefert die Anzahl Zeilen des Textbereichs.

`void setColumns(int cols)`
setzt die Anzahl Spalten des Textbereichs.

`int getColumns()`
liefert die Anzahl Spalten des Textbereichs.

`void setLineWrap(boolean b)`
Hat `b` den Wert `true`, werden Zeilen, die zu lang sind, automatisch umgebrochen.

`void setWrapStyleWord(boolean b)`
legt die Art des Zeilenumbruchs fest. Hat `b` den Wert `true`, werden Zeilen, die zu lang sind, am Ende eines Wortes umgebrochen.

`void append(String text)`
fügt `text` ans Ende des Textbereichs an.

`void insert(String text, int pos)`
fügt `text` an der Position `pos` im Textbereich ein.

`void replaceRange(String text, int start, int end)`
ersetzt den Text zwischen `start` und `end - 1` durch die Zeichenkette `text`.

Die `JTextComponent`-Methoden
`void read(Reader in, Object desc)`
`void write(Writer out)`
lesen Text in den Textbereich ein bzw. speichern ihn. Sie können eine `IOException` auslösen. `desc` beschreibt den Eingabestrom. `desc` kann auch den Wert `null` haben.

Programm 8.15

```
// Datei: TextTest2.java
...

public class TextTest2 extends JPanel {
   public static void main(String[] args) { ... }

   public TextTest2() {
      JTextArea t = new JTextArea(6, 15);
      t.setLineWrap(true);
      t.setWrapStyleWord(true);
      JScrollPane sp = new JScrollPane(t);
      add(sp);

      t.append("Das ist ein mehrzeiliges Textfeld.");
      t.insert("Hallo. ", 0);
      t.replaceRange("Dies", 7, 10);
   }
}
```

Bild 8.11: JTextArea

JEditorPane

`JEditorPane` implementiert einen Texteditor, der von der Klasse `JTextComponent` abgeleitet ist. Verschiedene Formate werden unterstützt: Plain-Text, Rich Text Format (RTF) und HTML.

Programm 8.16 realisiert einen einfachen Browser zur Anzeige von HTML-Seiten. Es werden nur die Methoden der Klasse `JEditorPane` vorgestellt, die im Beispiel benutzt werden.

Der Standardkonstruktor `JEditorPane()` erzeugt einen leeren Editor.

Die Navigation mittels Hyperlinks ist nur möglich, wenn die Komponente nicht editierbar ist: `setEditable(false)`.

`void setPage(String url)`
legt die HTML-Datei fest, die angezeigt werden soll. `url` ist die URL-Adresse der Datei (siehe Kapitel 9). Eine Ausnahme vom Typ `IOException` kann ausgelöst werden.

Ereignis-behandlung

Die Aktivierung eines Hyperlinks löst ein Hyperlink-Event aus.

```
void addHyperlinkListener(
  HyperlinkListener l)
```
registriert das Ereignisempfänger-Objekt `l`, das das Interface `HyperlinkListener` aus `javax.swing.event` implementiert.

`void hyperlinkUpdate(HyperlinkEvent e)`
wird im Ereignisempfänger aufgerufen, wenn ein Hyperlink-Event auftritt.

Methoden von `HyperlinkEvent`:

`HyperlinkEvent.EventType getEventType()`
liefert den Typ des Events: hier die Konstanten `ENTERED`, `EXITED` oder `ACTIVATED`.
(`EventType` ist eine "statische" innere Klasse von `HyperlinkEvent`.)

`URL getURL()`
liefert die URL-Adresse des Links.

Programm 8.16

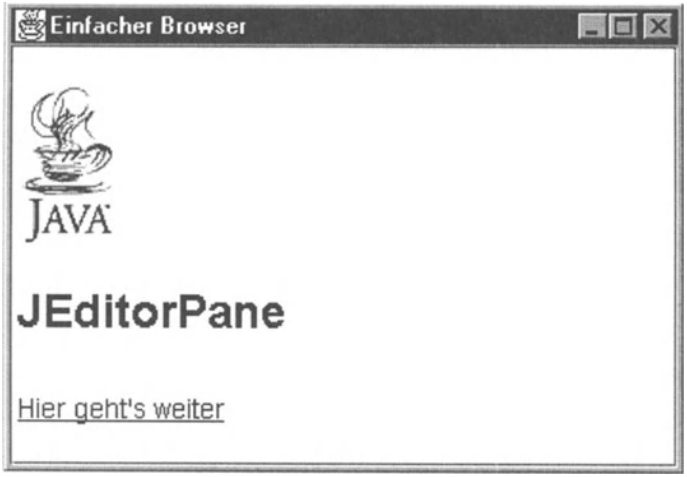
```
// Datei: EditorTest.java
...

public class EditorTest extends JPanel
   implements HyperlinkListener {
   private JEditorPane inhalt;

   public static void main(String[] args) { ... }

   public EditorTest(String seite) {
      setLayout(new BorderLayout());
      inhalt = new JEditorPane();
      inhalt.setEditable(false);
      inhalt.addHyperlinkListener(this);
      add(new JScrollPane(inhalt));
      ladeSeite(seite);
   }

   public void ladeSeite(String seite) {
      try {
         inhalt.setPage(seite);
      }
      catch (IOException e) {
         inhalt.setText(e.getMessage());
      }
   }

   public void hyperlinkUpdate(HyperlinkEvent e) {
      if (e.getEventType() ==
         HyperlinkEvent.EventType.ACTIVATED)
         ladeSeite(e.getURL().toString());
   }
}
```

Das Ergebnis des Aufrufs

`java EditorTest file:Test.html`

zeigt das folgende Bild.

Bild 8.12:
JEditorPane



8.3.6 Auswahlkomponenten

Komboboxen (`JComboBox`) und Listen (`JList`) sind Komponenten, die die Auswahl eines Eintrags aus einer Liste ermöglichen.

JComboBox

Die Klasse `JComboBox` realisiert ein Feld (Kombobox), das durch Mausklick eine Liste aufblendet, aus der ein Eintrag ausgewählt werden kann.

Konstruktoren:

```
JComboBox()
JComboBox(Object[] items)
JComboBox(Vector items)
```

erzeugen jeweils eine Kombobox mit leerer Liste, Elementen aus einem Array bzw. aus einem Vektor.

Wurde die Kombobox mit dem parameterlosen Konstruktor erzeugt, so können zum Einfügen und Löschen die folgenden Methoden benutzt werden:

```
void addItem(Object item)
void insertItemAt(Object item, int pos)
void removeItemAt(int pos)
void removeItem(Object item)
void removeAllItems()
```

`Object getItemAt(int pos)`
liefert den Eintrag an der Position `pos`.

`int getItemCount()`
liefert die Anzahl der Einträge.

`int getSelectedIndex()`
liefert die Position des ausgewählten Eintrags.

`Object getSelectedItem()`
liefert den ausgewählten Eintrag.

`void setSelectedIndex(int pos)`
wählt den Eintrag an der Position `pos` aus.

`void setSelectedItem(Object item)`
wählt den Eintrag `item` der Liste aus.

`void setMaximumRowCount(int count)`
setzt die maximale Anzahl von sichtbaren Zeilen der Liste. Enthält die Liste mehr als `count` Zeilen, wird ein Scrollbalken benutzt.

`void setEditable(boolean true)`
Hat `b` den Wert `true`, so kann das Feld editiert werden und die Auswahl über Tastatur eingegeben werden.

Ereignisbehandlung

Nach der Auswahl eines Eintrags wird ein Action-Event ausgelöst. Beim Wechsel des selektierten Eintrags wird ein Item-Event ausgelöst. Die entsprechenden Ereignisempfänger (Listener) können registriert werden (siehe Kapitel 8.3.2).

Programm 8.17 präsentiert eine Kombobox mit den Einträgen "rot", "grün" und "blau". Bei Auswahl eines Eintrags wird der Hintergrund des Fensters in der ausgewählten Farbe dargestellt.

Programm 8.17

```
// Datei: ComboBoxTest.java
...

public class ComboBoxTest extends JPanel
   implements ActionListener {
   private JComboBox cb;
   public static void main(String[] args) { ... }

   public ComboBoxTest() {
      cb = new JComboBox();
      cb.addActionListener(this);
      cb.addItem("rot");
      cb.addItem("grün");
      cb.addItem("blau");
      add(cb);
   }

   public void actionPerformed(ActionEvent e) {
      String farbe = (String)cb.getSelectedItem();
      if (farbe.equals("rot"))
         setBackground(Color.red);
      if (farbe.equals("grün"))
         setBackground(Color.green);
      if (farbe.equals("blau"))
         setBackground(Color.blue);
   }
}
```

Bild 8.13: JComboBox

JList

Die Klasse `JList` ermöglicht die Auswahl von einem oder mehreren Einträgen aus einer Liste, die mit einem Sichtfenster bestimmter Größe versehen ist.

Konstruktoren:

`JList()`
`JList(Object[] items)`
`JList(Vector items)`
erzeugen jeweils eine Liste mit leerer Liste, Elementen aus einem Array bzw. aus einem Vektor.

`void setListData(Object[] items)`
`void setListData(Vector items)`
füllt die Liste mit den Elementen aus dem Array bzw. aus dem Vektor `items`.

`void setVisibleRowCount(int count)`
legt die Anzahl Zeilen der Liste fest, die ohne Scrollbalken angezeigt werden können.

`void ensureIndexIsVisible(int pos)`
stellt sicher, dass der Eintrag an der Position `pos` sichtbar ist, wenn die Liste mit einem Scrollbalken versehen ist. (Die Komponente muss hierfür bereits sichtbar sein.)

`void setSelectionMode(int mode)`
bestimmt, ob ein (einfacher Mausklick) oder mehrere Einträge (Control-Taste und Mausklick bzw. Shift-Taste und Mausklick) ausgewählt werden können. Zulässige Werte von `mode` sind die `ListSelectionModel`-Konstanten: `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` und `MULTIPLE_INTERVAL_SELECTION`.

`int getSelectionMode()`
liefert den eingestellten Selektionsmodus.

`void clearSelection()`
hebt die aktuelle Auswahl in der Liste auf.

`boolean isSelectedIndex(int pos)`
liefert den Wert `true`, wenn der Eintrag an der Position `pos` ausgewählt ist.

`boolean isSelectionEmpty()`
liefert den Wert `true`, wenn nichts ausgewählt wurde.

`Object getSelectedValue()`
liefert den ersten ausgewählten Eintrag oder `null`, wenn kein Eintrag ausgewählt wurde.

`Object[] getSelectedValues()`
liefert die ausgewählten Einträge.

`int getSelectedIndex()`
liefert die Position des ausgewählten Eintrags oder `-1`, wenn kein Eintrag ausgewählt wurde.

`int[] getSelectedIndices()`
liefert die Positionen der ausgewählten Einträge.

`void setSelectedIndex(int pos)`
wählt den Eintrag an der Position `pos` aus.

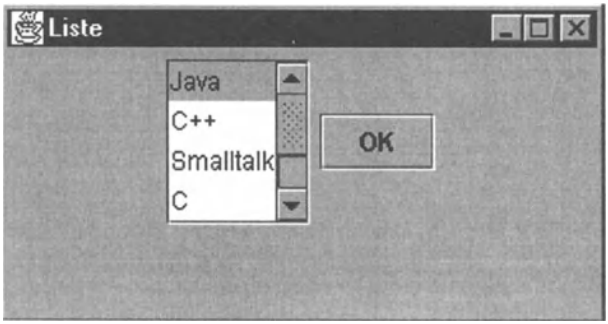
`void setSelectedIndices(int[] pos)`
wählt die Einträge an den Positionen `pos` aus.

Programm 8.18

```
// Datei: ListTest.java
...

public class ListTest extends JPanel
   implements ActionListener {
   private JButton b;
   private JList li;

   public static void main(String[] args) { ... }

   public ListTest() {
      String[] items = {"Java", "C++", "Smalltalk", "C",
         "Pascal", "Cobol"};
      li = new JList(items);
      li.setVisibleRowCount(4);
      add(new JScrollPane(li));
      b = new JButton(" OK ");
      b.addActionListener(this);
      add(b);
   }
```

```
    public void actionPerformed(ActionEvent e) {
        Object[] sel = li.getSelectedValues();
        for (int i = 0; i < sel.length; i++)
            System.out.println(sel[i]);
        System.out.println();
    }
}
```

Bild 8.14: JList



8.4 Layout-Manager

Die Anordnung von Komponenten in einem Container erfolgt mit Hilfe von so genannten *Layout-Managern*. Es gibt verschiedene Layout-Manager, denen jeweils ein anderes Konzept zu Grunde liegt. Allen gemeinsam ist, dass eine Platzierung durch Angabe der genauen Pixelwerte nicht erforderlich ist. Ein Vorteil ist die automatische Anpassung der Größe der Komponenten bei Verkleinerung bzw. Vergrößerung des sie enthaltenden Containers.

In diesem Kapitel behandeln wir die Layout-Manager `FlowLayout`, `BorderLayout` und `GridLayout` aus dem Paket `java.awt` sowie `BoxLayout` aus `javax.swing`.

Die Methode `void setLayout(LayoutManager mgr)` der Klasse `Container` setzt den Layout-Manager für den Container. Alle Layout-Manager implementieren das Interface `LayoutManager` bzw. ein Sub-Interface hiervon.

Null-Layout

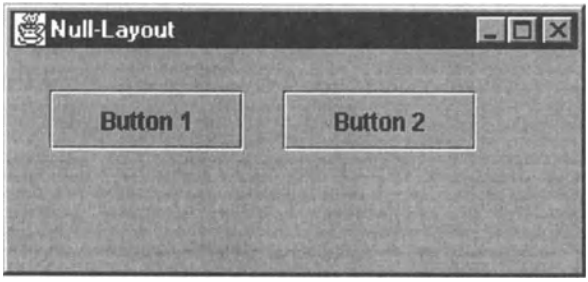
Ein so genanntes *Null-Layout* wird durch Aufruf der Methode `setLayout(null)` im Container erzeugt. Es wird kein Layout-Manager verwendet. Alle Komponenten werden dann mit Hilfe der `Component`-Methoden `setLocation` und `setSize` (siehe Kapitel 8.2) oder einfacher mit `setBounds` positioniert:

```
void setBounds(int x, int y, int width,
  int height)
```

Programm 8.19

```
// Datei: NullLayoutTest.java
...

public class NullLayoutTest extends JPanel {
   public static void main(String[] args) { ... }

   public NullLayoutTest() {
      setLayout(null);
      JButton b1 = new JButton("Button 1");
      JButton b2 = new JButton("Button 2");
      b1.setBounds(20, 20, 100, 30);
      b2.setBounds(140, 20, 100, 30);
      add(b1);
      add(b2);
   }
}
```

Bild 8.15: Null-Layout



FlowLayout

`FlowLayout` ist der Standard-Layout-Manager für Objekte der Klasse `JPanel`. Er ordnet die Komponenten zeilenweise von oben links nach unten rechts an. Passt eine Komponente

nicht mehr in eine Zeile (z.B. nach Verkleinerung des Containers), so wird sie automatisch in der nächsten Zeile angeordnet.

```
FlowLayout()
FlowLayout(int align)
FlowLayout(int align, int hgap, int vgap)
```

erzeugen jeweils das Layout. `align` bestimmt mit Hilfe von Konstanten dieser Klasse die Anordnung in einer Zeile: `CENTER` (zentriert), `LEFT` (linksbündig), `RIGHT` (rechtsbündig). Als Voreinstellung wird `CENTER` verwendet. `hgap` bestimmt den horizontalen und `vgap` den vertikalen Komponentenabstand. Die Voreinstellung ist 5.

Programm 8.20

```
// Datei: FlowLayoutTest.java
...

public class FlowLayoutTest extends JPanel {
   public static void main(String[] args) { ... }

   public FlowLayoutTest() {
      setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));
      add(new JButton("Button 1"));
      add(new JButton("Button 2"));
      add(new JButton("Button 3"));
      add(new JButton("Button 4"));
      add(new JButton("Button 5"));
   }
}
```

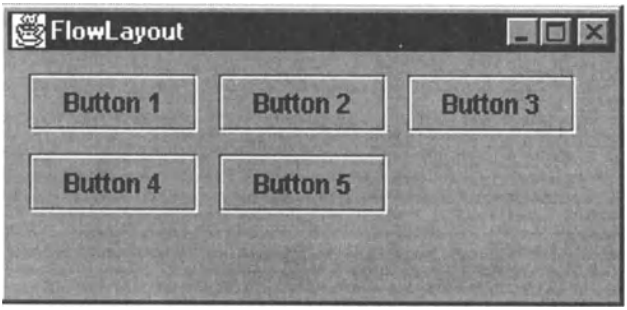


Bild 8.16: FlowLayout

BorderLayout

`BorderLayout` ist der Standard-Layout-Manager für die *content pane* (siehe Kapitel 8.2) von `JFrame` und `JDialog`. Er ordnet maximal fünf Komponenten an den vier Seiten und im Zentrum des Containers an.

Die Platzierung einer Komponente mit `add(comp)` erfolgt grundsätzlich im Zentrum. Mit `add(comp, pos)` wird die Komponente im Bereich `pos` platziert, wobei für `pos` eine der Konstanten `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER` der Klasse `BorderLayout` stehen darf.

`BorderLayout()`
`BorderLayout(int hgap, int vgap)`
erzeugen das Layout ohne bzw. mit Komponentenabstand.

Programm 8.21

```
// Datei: BorderLayoutTest.java
...

public class BorderLayoutTest extends JPanel {
   public static void main(String[] args) { ... }

   public BorderLayoutTest() {
      setLayout(new BorderLayout(10, 10));
      add(new JButton("Button 1"), BorderLayout.NORTH);
      add(new JButton("Button 2"), BorderLayout.SOUTH);
      add(new JButton("Button 3"), BorderLayout.WEST);
      add(new JButton("Button 4"), BorderLayout.EAST);
      add(new JButton("Button 5"), BorderLayout.CENTER);
   }
}
```

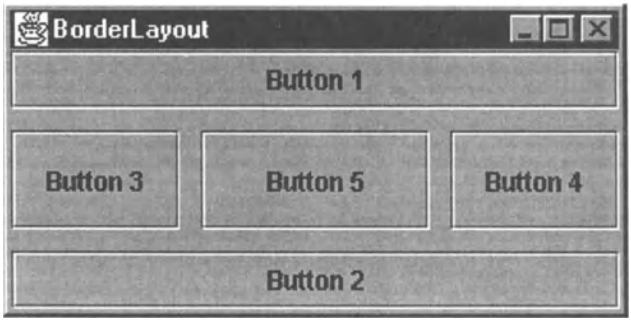


Bild 8.17: BorderLayout

GridLayout

`GridLayout` ordnet die Komponenten in einem Raster aus Zeilen gleicher Höhe und Spalten gleicher Breite an.

```
GridLayout()
GridLayout(int rows, int cols)
GridLayout(int rows, int cols, int hgap,
  int vgap)
```

erzeugen ein Layout, bei dem alle Komponenten in einer Zeile liegen bzw. `rows` Zeilen und `cols` Spalten haben. `hgap` und `vgap` geben den horizontalen bzw. vertikalen Komponentenabstand an.

Programm 8.22

```
// Datei: GridLayoutTest.java
...

public class GridLayoutTest extends JPanel {
   public static void main(String[] args) { ... }

   public GridLayoutTest() {
      setLayout(new GridLayout(2, 3, 20, 20));
      add(new JButton("Button 1"));
      add(new JButton("Button 2"));
      add(new JButton("Button 3"));
      add(new JButton("Button 4"));
      add(new JButton("Button 5"));
   }
}
```

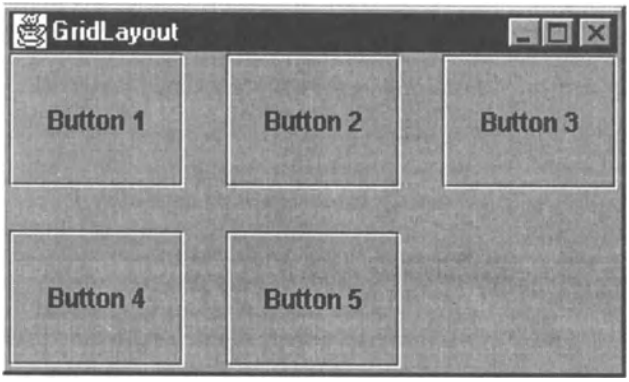


Bild 8.18: GridLayout

BoxLayout

Mit dem `BoxLayout` können alle Komponenten entweder in horizontaler oder in vertikaler Richtung angeordnet werden. Es wird dabei nie mehr als eine Zeile bzw. eine Spalte angelegt. Die Komponenten können unterschiedlich viel Platz belegen.

`BoxLayout(Container c, int axis)`
erzeugt das Layout für `c` mit der Richtung `axis`. Gültige Werte für `axis` sind die Konstanten `X_AXIS` bzw. `Y_AXIS`.

Die Klasse `Box` mit dem Konstruktor `Box(int axis)` ist ein Container, der das `BoxLayout` nutzt.

Klassenmethoden von `Box`:

`static Box createHorizontalBox()`
`static Box createVerticalBox()`
lieferen eine Box mit horizontaler bzw. vertikaler Ausrichtung.

```
static Component createHorizontalStrut(
  int width)
static Component createVerticalStrut(
  int height)
```
liefern einen festen Zwischenraum der angegebenen Größe.

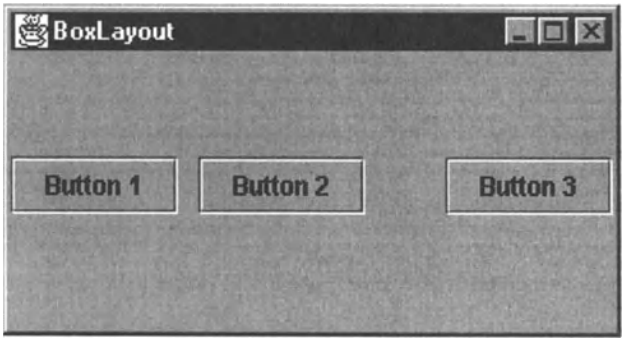
`static Component createHorizontalGlue()`
`static Component createVerticalGlue()`
liefern eine leere Komponente, die sich in der angegebenen Richtung ausdehnen kann.

Programm 8.23

```
// Datei: BoxLayoutTest.java
...

public class BoxLayoutTest extends JPanel {
   public static void main(String[] args) { ... }

   public BoxLayoutTest() {
      setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
      add(new JButton("Button 1"));
      add(Box.createHorizontalStrut(10));
      add(new JButton("Button 2"));
```

```
            add(Box.createHorizontalGlue());
            add(new JButton("Button 3"));
        }
    }
```

Bild 8.19: BoxLayout



Geschachtelte Layouts

Container mit eigenem Layout-Manager, wie z.B. `JPanel`, können ineinander geschachtelt werden.

Programm 8.24 ordnet zwei `JPanel`-Objekte gemäß dem `BorderLayout` an. Die Komponenten des ersten Panels werden gemäß dem `GridLayout`, die Komponenten des zweiten Panels gemäß dem `FlowLayout` platziert.

Programm 8.24

```
// Datei: LayoutTest.java
...

public class LayoutTest extends JPanel {
   public static void main(String[] args) { ... }

   public LayoutTest() {
      setLayout(new BorderLayout());
      JPanel p1 = new JPanel();
      JPanel p2 = new JPanel();
      add(p1, BorderLayout.CENTER);
      add(p2, BorderLayout.SOUTH);
      p1.setLayout(new GridLayout(2, 3, 5, 5));
```

```
        for (int i = 1; i <= 6; ++i)
           p1.add(new JButton("Button " + i));
        p2.setLayout(new FlowLayout(FlowLayout.RIGHT));
        p2.add(new JButton("Button 7"));
        p2.add(new JButton("Button 8"));
     }
  }
```

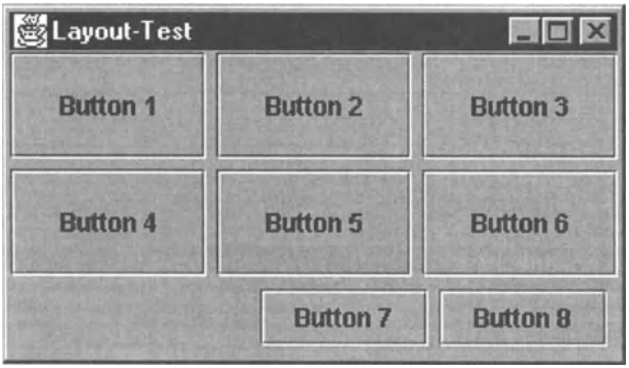


Bild 8.20: Geschachtelte Layouts

8.5 Menüs und Symbolleisten

Fenster können *Menüs* enthalten. Ihre Erstellung wird durch spezielle Klassen unterstützt. Die Klasse `JMenuBar` stellt die Menüleiste des Fensters dar, die Klasse `JMenu` ein einzelnes der darin enthaltenes Menüs, die Klassen `JMenuItem`, `JCheckBoxMenuItem` und `JRadioButtonMenuItem` implementieren die auswählbaren Menüeinträge eines Menüs.

Die Klassen `JFrame`, `JDialog` und `JApplet` besitzen die Methode `void setJMenuBar(JMenuBar mb)`, die die Menüleiste `mb` einbindet.

JMenuBar

`JMenuBar()`
erzeugt eine leere Menüleiste.

`JMenu add(JMenu m)`
fügt das Menü `m` der Menüleiste hinzu.

`void remove(Component m)`
entfernt das Menü `m` aus der Menüleiste.

`JMenu getMenu(int pos)`
liefert das Menü an der Position `pos`.

JMenu

`JMenu()`
`JMenu(String text)`
erzeugen ein Menü ohne bzw. mit Titel. `JMenu` ist Subklasse von `JMenuItem`.

`JMenuItem add(JMenuItem mi)`
fügt den Menüeintrag `mi` einem Menü hinzu. Dabei kann `mi` selbst ein Menü sein (*Untermenü*).

`JMenuItem insert(JMenuItem mi, int pos)`
fügt einen Menüeintrag an der Position `pos` in das Menü ein.

`void addSeparator()`
fügt eine Trennlinie dem Menü hinzu.

`void insertSeparator(int pos)`
fügt eine Trennlinie an der Position `pos` in das Menü ein.

`void remove(int pos)`
`void remove(JMenuItem mi)`
`void removeAll()`
entfernen den Menüpunkt an der Position `pos`, den Menüpunkt `mi` bzw. alle Menüpunkte aus dem Menü.

`JMenuItem getItem(int pos)`
liefert den Menüeintrag an der Position `pos`.

JMenuItem

Alle Einträge in einem Menü gehören zur Klasse `JMenuItem`. Wenn ein Menüeintrag ausgewählt wird, erzeugt er ein Action-Event.

`JMenuItem()`
`JMenuItem(Icon icon)`
`JMenuItem(String text)`
`JMenuItem(String text, Icon icon)`
erzeugen einen Menüeintrag mit oder ohne Beschriftung bzw. Icon.

Da JMenuItem Subklasse von AbstractButton ist, stehen auch die Methoden dieser Superklasse zur Verfügung.

Komponenten deaktivieren

Die beiden folgenden Methoden stammen aus der Klasse Component und sind für alle Komponenten aufrufbar:

```
void setEnabled(boolean b)
```
deaktiviert die Komponente (hier den Menüeintrag), wenn b den Wert false hat.

```
boolean isEnabled()
```
liefert den Wert true, wenn die Komponente (hier der Menüeintrag) aktiviert ist.

Hotkey

Menüeinträge können mit einem Tastaturkürzel (*Hotkey*) versehen werden. Damit kann dann ein Eintrag alternativ über die Tastatur ausgewählt werden. Ein KeyStroke-Objekt repräsentiert einen Hotkey. Die KeyStroke-Methode

```
static KeyStroke getKeyStroke(
  int code, int mod)
```
liefert einen Hotkey. Für code kann eine Konstante der Klasse KeyEvent eingesetzt werden, z.B. VK_0, ... , VK_9, VK_A, ... ,VK_Z. Gültige Werte für mod sind die Event-Konstanten CTRL_MASK, SHIFT_MASK oder ALT_MASK oder eine additive Kombination hiervon.

```
void setAccelerator(KeyStroke k)
```
stattet den Menüeintrag mit dem Hotkey k aus.

```
KeyStroke getAccelerator()
```
liefert den Hotkey des Menüeintrags.

JCheckBox-MenuItem

Die Klasse JCheckBoxMenuItem implementiert einen Menüeintrag mit dem Zustand "selektiert" oder "nicht selektiert". Das Ändern des Zustands löst wie bei einem JCheckBox-Objekt ein Item-Event aus.

JCheckBoxMenuItem hat die zu JMenuItem analogen Konstruktoren. Zusätzlich zu Text und Icon kann durch ein boolean-Argument der Auswahlzustand angegeben werden.

JRadioButton-MenuItem

Die Klasse `JRadioButtonMenuItem` ist analog zur Klasse `JCheckBoxMenuItem` aufgebaut. Mehrere dieser Menüeinträge werden in einer Gruppe (`ButtonGroup`, siehe Kapitel 8.3.2) zusammengefasst.

JToolBar

Symbolleisten werden von der Klasse `JToolBar` implementiert.

`JToolBar()`
`JToolBar(int orientation)`
erzeugen eine Symbolleiste mit horizontaler (`HORIZONTAL`) bzw. vertikaler (`VERTICAL`) Ausrichtung. Der parameterlose Konstruktor erzeugt eine horizontal ausgerichtete Leiste.

Symbolleisten werden mit der `Container`-Methode `add` gefüllt.

`void setOrientation(int orientation)`
legt die Ausrichtung fest.

`void setFloatable(boolean b)`
Hat `b` den Wert `true`, so kann die Symbolleiste mit der Maus an eine andere Position innerhalb ihres Containers gezogen werden.

`void addSeparator()`
`void addSeparator(Dimension size)`
fügen Abstände hinzu.

Das folgende Programm zeigt ein Fenster mit Menü- und Symbolleiste.

Programm 8.25

```
// Datei: MenuTest.java
...

public class MenuTest extends JFrame
   implements ActionListener {
   public static void main(String[] args) {
      new MenuTest();
   }

   public MenuTest() {
      super("Menü-Test");
```

```
        MyMenuBar menubar = new MyMenuBar(this);
        setJMenuBar(menubar);
        MyToolBar toolbar = new MyToolBar(this);
        toolbar.setFloatable(true);
        getContentPane().add(toolbar, BorderLayout.NORTH);
        ...
    }

    public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();
        String cmd = e.getActionCommand();
        if (obj instanceof JMenuItem) {
            System.out.println("Menue: " + cmd + " " +
                ((AbstractButton)obj).isSelected());
            if (cmd.equals("ende"))
                System.exit(0);
        }
        else
            System.out.println("Toolbar: " + cmd);
    }
}

class MyMenuBar extends JMenuBar {
    public MyMenuBar(ActionListener listener) {
        JMenu m;
        JMenuItem mi;
        m = new JMenu("Datei");
        mi = new JMenuItem("Neu");
        mi.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_N, Event.CTRL_MASK));
        mi.setActionCommand("neu");
        mi.addActionListener(listener);
        m.add(mi);
        ...
        m.addSeparator();
        mi = new JMenuItem("Beenden");
        mi.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_B, Event.CTRL_MASK));
        mi.setActionCommand("ende");
        mi.addActionListener(listener);
        m.add(mi);
        add(m);
        m = new JMenu("Bearbeiten");
```

```
        mi = new JMenuItem("Ausschneiden");
        mi.setActionCommand("ausschneiden");
        mi.addActionListener(listener);
        m.add(mi);
        ...
        mi = new JCheckBoxMenuItem("Invers");
        mi.setActionCommand("invers");
        mi.addActionListener(listener);
        m.add(mi);
        JMenu um = new JMenu("Formen");
        JRadioButtonMenuItem rb1, rb2, rb3;
        rb1 = new JRadioButtonMenuItem("Linie", true);
        rb1.setActionCommand("linie");
        rb1.addActionListener(listener);
        um.add(rb1);
        ...
        ButtonGroup bg = new ButtonGroup();
        bg.add(rb1);
        ...
        m.add(um);
        add(m);
    }
}

class MyToolBar extends JToolBar {
    public MyToolBar(ActionListener listener) {
        JButton b;
        b = new JButton(new ImageIcon("cut.gif"));
        b.setActionCommand("Ausschneiden");
        b.addActionListener(listener);
        add(b);
        addSeparator();
        b = new JButton(new ImageIcon("copy.gif"));
        b.setActionCommand("Kopieren");
        b.addActionListener(listener);
        add(b);
        b = new JButton(new ImageIcon("paste.gif"));
        b.setActionCommand("Einfuegen");
        b.addActionListener(listener);
        add(b);
    }
}
```

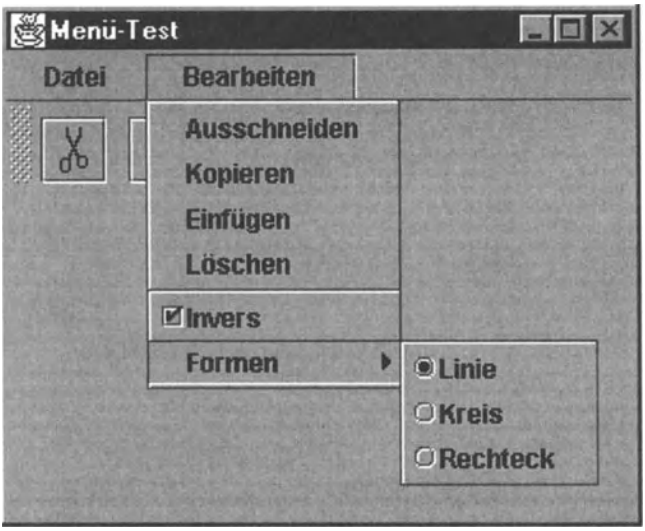


Bild 8.21: Menü und Symbolleiste

8.6 Dialogfenster

JDialog

Die Klasse `JDialog` erzeugt ein spezielles Fenster, das abhängig von einem anderen Fenster geöffnet werden kann. Ein solches Fenster kann als *modales* oder *nicht modales* Fenster erstellt werden. *Modal* bedeutet, dass man im anderen Fenster erst, nachdem das Dialogfenster geschlossen wurde, weiterarbeiten kann.

Konstruktoren:

```
Dialog(Typ owner)
Dialog(Typ owner, boolean modal)
Dialog(Typ owner, String title)
Dialog(Typ owner, String title,
  boolean modal)
```

erzeugen ein Dialogfenster ohne oder mit Titel. `owner` ist die übergeordnete Komponente. Für *Typ* ist entweder `JFrame`

oder `JDialog` einzusetzen. Hat `modal` den Wert `true`, so wird ein modales Fenster erzeugt.

`void setTitle(String title)`
setzt den Titel.

`String getTitle()`
liefert den Titel.

`void setModal(boolean b)`
gibt an, ob das Fenster modal sein soll. Hierzu muss `b` den Wert `true` haben.

`boolean isModal()`
liefert den Wert `true`, wenn das Fenster modal ist.

`void setResizable(boolean b)`
gibt an, ob die Größe des Fensters durch den Benutzer verändert werden kann. Hierzu muss `b` den Wert `true` haben.

`boolean isResizable()`
liefert den Wert `true`, wenn die Größe des Fensters vom Benutzer verändert werden kann.

`void setLocationRelativeTo(Component c)`
positioniert das Dialogfenster relativ zu `c`.

Programm 8.26 erzeugt ein modales Dialogfenster, in dem der Benutzername und ein zugehöriges Kennwort eingegeben werden kann.

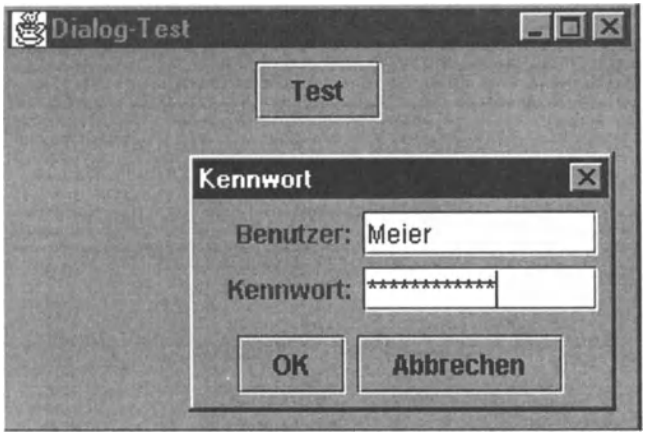
Programm 8.26

```
// Datei: DialogTest1.java
...

public class DialogTest1 extends JPanel
   implements ActionListener {
   private JFrame owner;
   private Kennwort kw;

   public static void main(String[] args) {
      JFrame frame = new JFrame("Dialog-Test");
      frame.getContentPane().add(new DialogTest1(frame));
```

```
        ...
    }

    public DialogTest1(JFrame owner) {
        this.owner = owner;
        JButton b = new JButton("Test");
        b.addActionListener(this);
        add(b);
    }

    public void actionPerformed(ActionEvent e) {
        if (kw == null)
            kw = new Kennwort(owner, "Kennwort");
        kw.clear();
        kw.setVisible(true);
        if (kw.isOK()) {
            System.out.println(kw.getUser());
            System.out.println(kw.getKennwort());
        }
    }
}

class Kennwort extends JDialog implements ActionListener {
    private JTextField user;
    private JPasswordField kennwort;
    private boolean result;

    public Kennwort(JFrame owner, String title) {
        super(owner, title, true);

        JPanel p1 = new JPanel();
        p1.setLayout(new FlowLayout(FlowLayout.RIGHT));
        p1.add(new JLabel("Benutzer:"));
        user = new JTextField(10);
        p1.add(user);
        p1.add(new JLabel("Kennwort:"));
        kennwort = new JPasswordField(10);
        p1.add(kennwort);

        JPanel p2 = new JPanel();
        JButton ok = new JButton("OK");
        ok.addActionListener(this);
        p2.add(ok);
```

```
        JButton abbr = new JButton("Abbrechen");
        abbr.addActionListener(this);
        p2.add(abbr);

        Container c = getContentPane();
        c.add(p1, BorderLayout.CENTER);
        c.add(p2, BorderLayout.SOUTH);
        Point p = owner.getLocation();
        setLocationRelativeTo(owner);
        setSize(200, 120);
        setResizable(false);
    }

    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if (cmd.equals("OK"))
            result = true;
        else
            result = false;
        setVisible(false);
        dispose();
    }

    public boolean isOK() {
        return result;
    }

    public String getUser() {
        return user.getText();
    }

    public String getKennwort() {
        return String.valueOf(kennwort.getPassword());
    }

    public void clear() {
        user.setText("");
        kennwort.setText("");
    }
}
```

Bild 8.22: JDialog



JOptionPane

Einfache Standarddialoge können mit Klassenmethoden der Klasse `JOptionPane` erstellt werden.

Bestätigung

```
static int showConfirmDialog(
  Component owner, Object msg, String title,
  int optType, int msgType)
```

zeigt einen *Bestätigungsdialog* im übergeordneten Fenster `owner` mit Anzeigetext `msg` und Titel `title`. Die verfügbaren Buttons und der Nachrichtentyp werden mittels `optType` bzw. `msgType` angegeben. Je nach ausgewähltem Button werden die Konstanten `YES_OPTION`, `NO_OPTION` oder `CANCEL_OPTION` geliefert. Gültige Werte für `optType` sind `YES_NO_OPTION` und `YES_NO_CANCEL_OPTION`.

Gültige Werte für `msgType` sind die Konstanten `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` und `PLAIN_MESSAGE`.

Eingabe

```
static String showInputDialog(
  Component owner, Object msg, String title,
  int msgType)
```

zeigt einen *Eingabedialog* im übergeordneten Fenster mit Anzeigetext, Titel und Nachrichtentyp. Der eingegebene Text wird zurückgeliefert.

Mitteilung

```
static void showMessageDialog(
  Component owner, Object msg, String title,
  int msgType)
```

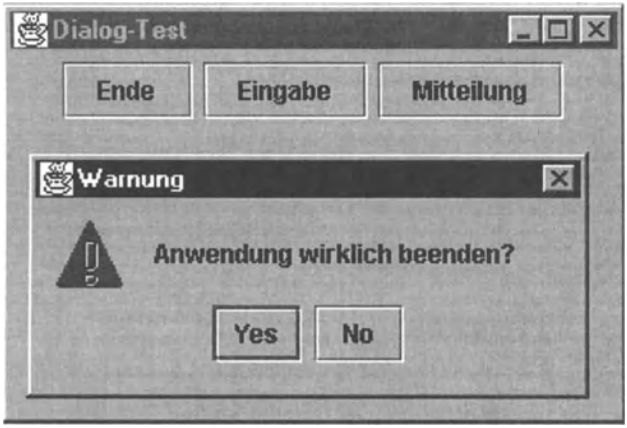
zeigt einen *Mitteilungsdialog* im übergeordneten Fenster mit Anzeigetext, Titel und Nachrichtentyp.

Programm 8.27

```
// Datei: DialogTest2.java
...

public class DialogTest2 extends JPanel
   implements ActionListener {
   private JFrame owner;

   public static void main(String[] args) {
      JFrame frame = new JFrame("Dialog-Test");
      frame.getContentPane().add(new DialogTest2(frame));
      ...
   }

   public DialogTest2(JFrame owner) {
      this.owner = owner;
      JButton ende = new JButton("Ende");
      ende.addActionListener(this);
      add(ende);
      JButton ein = new JButton("Eingabe");
      ein.addActionListener(this);
      add(ein);
      JButton msg = new JButton("Mitteilung");
      msg.addActionListener(this);
      add(msg);
   }

   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("Ende")) {
         int wahl = JOptionPane.showConfirmDialog(
```

```
                owner, "Anwendung wirklich beenden?",
                "Warnung", JOptionPane.YES_NO_OPTION,
                JOptionPane.WARNING_MESSAGE);
            if (wahl == JOptionPane.YES_OPTION)
                System.exit(0);
        }
        if (cmd.equals("Eingabe")) {
            String s = JOptionPane.showInputDialog(
                owner, "Bitte Kontonummer eingaben:", "",
                JOptionPane.PLAIN_MESSAGE);
            if (s != null)
                System.out.println(s);
        }
        if (cmd.equals("Mitteilung")) {
            JOptionPane.showMessageDialog(
                owner, "Das ist ein Test!", "Wichtige
                Mitteilung", JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

Bild 8.23: JOptionPane



JFileChooser

Die Klasse `JFileChooser` implementiert einen Dialog zur Auswahl von Dateien im Dateisystem. Der Standardkonstruktor erzeugt einen Dialog mit dem Home-Verzeichnis des Benutzers als Voreinstellung.

`void setDialogTitle(String title)`
`String getDialogTitle()`
setzt bzw. liefert den Titel des Dialogfensters.

`void setCurrentDirectory(File dir)`
`File getCurrentDirectory()`
setzt bzw. liefert das aktuelle Verzeichnis.

`void setSelectedFile(File file)`
`File getSelectedFile()`
setzt bzw. liefert die ausgewählte Datei.

`int showOpenDialog(Component c)`
`int showSaveDialog(Component c)`
`int showDialog(Component c, String text)`
zeigen jeweils das Dialogfenster an. Die Dialoge unterscheiden sich nur in der Beschriftung des Buttons, mit dem die Auswahl bestätigt wird: "Open", "Save" bzw. `text`. `c` kann auch `null` sein. Zurückgeliefert wird eine der Konstanten `APPROVE_OPTION` oder `CANCEL_OPTION`, je nachdem, ob die Auswahl bestätigt oder der Dialog abgebrochen wurde.

FileFilter

`void setFileFilter(FileFilter filter)`
legt einen Filter fest, der dazu dient, Dateien von der Anzeige im Dialogfenster auszuschließen.
`filter` ist ein Objekt der abstrakten Klasse `FileFilter` aus dem Paket `javax.swing.filechooser`. Folgende Methoden müssen implementiert werden:

`boolean accept(File file)`
liefert `true`, wenn `file` ausgewählt werden darf.

`String getDescription()`
liefert eine Beschreibung des Filters.

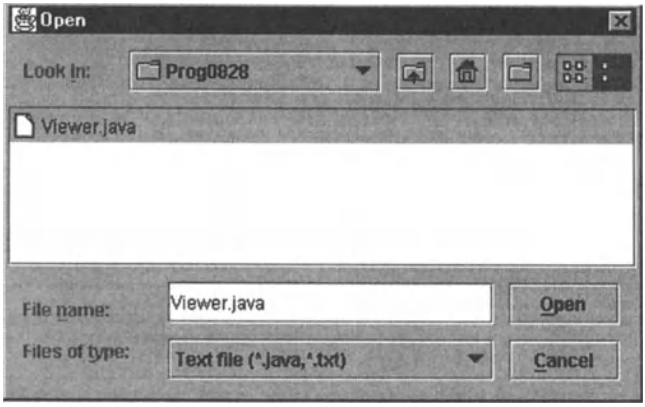
Programm 8.28 demonstriert einen Auswahldialog für Dateien mit der Endung ".java" bzw. ".txt". Die Inhalte werden in einem `JTextArea`-Objekt angezeigt.

Programm 8.28

```
// Datei: Viewer.java
...

public class Viewer extends JPanel
   implements ActionListener {
   private JFrame owner;
   private JTextArea text;

   public static void main(String[] args) {
      JFrame frame = new JFrame("Viewer");
      frame.getContentPane().add(new Viewer(frame));
      frame.pack();
      ...
   }

   public Viewer(JFrame owner) {
      this.owner = owner;
      setLayout(new BorderLayout());
      JButton open = new JButton("Öffnen");
      open.addActionListener(this);
      add(open, BorderLayout.NORTH);
      text = new JTextArea(20, 50);
      text.setEditable(false);
      text.setLineWrap(true);
      JScrollPane sp = new JScrollPane(text);
      add(sp, BorderLayout.CENTER);
   }

   public void actionPerformed(ActionEvent e) {
      JFileChooser fc = new JFileChooser();
      fc.setCurrentDirectory(new File("."));
      fc.setFileFilter(new DateiAuswahl());
      int auswahl = fc.showOpenDialog(owner);
      if (auswahl == JFileChooser.APPROVE_OPTION)
         oeffne(fc.getSelectedFile());
   }
```

```
    public void oeffne(File datei) {
        try {
            text.read(new FileReader(datei), null);
        }
        catch (IOException e) {
            text.setText(e.getMessage());
        }
    }
}

class DateiAuswahl extends FileFilter {
    public boolean accept(File f) {
        if (f.isDirectory())
            return true;
        String name = f.getName().toLowerCase();
        if (name.endsWith(".java"))
            return true;
        if (name.endsWith(".txt"))
            return true;
        return false;
    }

    public String getDescription() {
        return "Text file (*.java,*.txt)";
    }
}
```

Bild 8.24:
JFileChooser



8.7 Tabellen

Mit Hilfe der Klasse `JTable` können Daten tabellarisch angezeigt und einzelne Tabelleneinträge editiert werden.

Aus der umfangreichen Sammlung von Methoden zur Erstellung und Präsentation von Tabellendaten zeigt dieses Kapitel nur einen Teil, der für die Entwicklung eines Beispielprogramms erforderlich ist.

JTable

`JTable(TableModel m)`
erzeugt eine Tabelle auf der Basis des Datenmodells `m`.

Wird der Standardkonstruktor `JTable()` verwendet, so kann das Datenmodell mit der Methode

```
void setModel(TableModel m)
```

gesetzt werden.

`TableModel` ist ein Interface aus `javax.swing.table`. Es spezifiziert Methoden für die Anzeige und Änderung der Daten in der `JTable`-Komponente. Die abstrakte Klasse `AbstractTableModel` aus dem gleichen Paket implementiert dieses Interface nur zum Teil.

Einige Methoden dieser Klasse sind:

`int getColumnCount()`
liefert die Anzahl der Spalten im Datenmodell. Diese Methode muss implementiert werden.

`int getRowCount()`
liefert die Anzahl der Zeilen im Datenmodell. Diese Methode muss implementiert werden.

`Object getValueAt(int row, int col)`
liefert den aktuellen Wert in Zeile `row` und Spalte `col` im Datenmodell. Diese Methode muss implementiert werden.

`String getColumnName()`
liefert den Namen einer Spalte im Datenmodell.

`boolean isCellEditable(int row, int col)`
liefert `true`, falls die Zelle editiert werden kann.

```
void setValueAt(Object value, int row,
  int col)
```
modifiziert den Wert einer Zelle im Datenmodell.

```
void fireDataTableChanged()
```
benachrichtigt alle `TableModelListener`-Objekte darüber, dass Daten geändert wurden.

Insbesondere ist das `JTable`-Objekt als Listener mit dem Datenmodell verbunden.

Ereignis-behandlung

Bei einer Änderung des Datenmodells wird ein TableModel-Event erzeugt, sofern `fireDataTableChanged` aufgerufen wurde.

Änderung des Datenmodells

```
void addTableModelListener(
  TableModelListener l)
```
registriert das Ereignisempfänger-Objekt `l`, das das Interface `TableModelListener` aus `javax.swing.event` implementiert.

```
void tableChanged(TableModelEvent e)
```
wird im Ereignisempfänger aufgerufen, wenn ein Table-Model-Event auftritt.

Auswahl von Zeilen

Die Auswahl von Zeilen in der Tabelle löst ein ListSelection-Event aus.

Die `ListSelectionModel`-Methode
```
addListSelectionListener(
  ListSelectionListener l)
```
registriert das Ereignisempfänger-Objekt `l`, das das Interface `ListSelectionListener` aus `javax.swing.event` implementiert.

```
void valueChanged(ListSelectionEvent e)
```
wird im Ereignisempfänger aufgerufen, wenn ein List-Selection-Event auftritt.

Die `JTable`-Methode
```
ListSelectionModel getSelectionModel()
```
liefert das `ListSelectionModel`-Objekt.

Einige JTable-Methoden:

`void setPreferredScrollableViewportSize(Dimension size)`
stellt die bevorzugte Größe des Anzeigebereichs ein.

`int getSelectedRow()`
liefert den Index der ersten selektierten Zeile.

`int[] getSelectedRows()`
liefert die Indizes aller selektierten Zeilen.

`Object getValueAt(int row, int col)`
liefert den aktuellen Wert in Zeile `row` und Spalte `col`.

Programm 8.29 stellte eine Tabelle mit den Spalten "Name", "Vorname" und "Stunden" dar. Werte in der dritten Spalte können geändert werden. Die Spaltensumme wird automatisch aktualisiert.

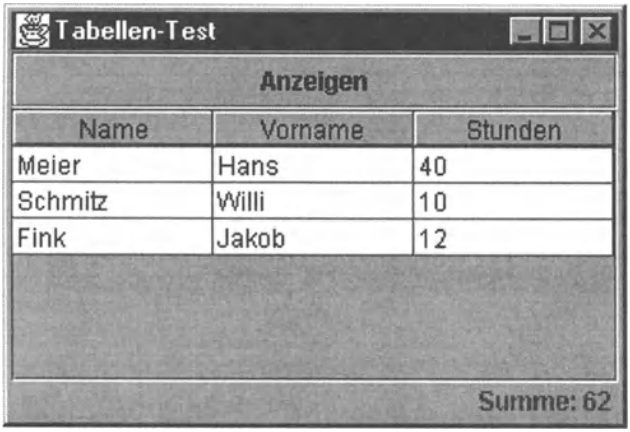
Programm 8.29

```
// Datei: TableTest.java
...

public class TableTest extends JPanel {
   private MyTableModel model;
   private JLabel summe;

   public static void main(String[] args) { ... }

   public TableTest() {
      model = new MyTableModel();
      setLayout(new BorderLayout());
      JButton b = new JButton("Anzeigen");
      b.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            for (int i = 0; i < model.getRowCount(); i++)
            {
               for (int j = 0; j < model.getColumnCount();
                  j++)
                  System.out.print(
                     model.getValueAt(i, j) + "\t");
               System.out.println();
            }
         }
```

```
        });
        add(b, BorderLayout.NORTH);

        JTable table = new JTable(model);
        add(new JScrollPane(table), BorderLayout.CENTER);
        summe = new JLabel(" ", JLabel.RIGHT);
        add(summe, BorderLayout.SOUTH);
        setSumme();

        model.addTableModelListener(new TableModelListener()
        {
           public void tableChanged(TableModelEvent e) {
              setSumme();
           }
        } );
     }

     public void setSumme() {
        int sum = 0;
        try {
           for (int i = 0; i < model.getRowCount(); i++) {
              String s = model.getValueAt(i, 2).toString();
              sum += Integer.valueOf(s).intValue();
           }
           summe.setText("Summe: " +
              String.valueOf(sum) + " ");
        }
        catch (NumberFormatException e) { }
     }
  }

  class MyTableModel extends AbstractTableModel {
     String[] namen = {"Name", "Vorname", "Stunden"};
     Object[][] daten = {
        {"Meier", "Hans", "40"},
        {"Schmitz", "Willi", "10"},
        {"Fink", "Jakob", "12"}
     };

     public int getColumnCount() {
        return namen.length;
     }
```

```
    public int getRowCount() {
       return daten.length;
    }

    public Object getValueAt(int zeile, int spalte) {
       return daten[zeile][spalte];
    }

    public String getColumnName(int spalte) {
       return namen[spalte];
    }

    public void setValueAt(Object value, int zeile,
       int spalte) {
       daten[zeile][spalte] = value;
       fireTableDataChanged();
    }

    public boolean isCellEditable(int zeile, int spalte) {
       if (spalte == 2)
          return true;
       else
          return false;
    }
}
```

Bild 8.25: JTable



8.8 Mausaktionen

Soll auf Mausaktionen, wie Drücken, Loslassen der Maustaste oder Bewegung des Mauszeigers, reagiert werden, so muss ein entsprechender Ereignisempfänger codiert und registriert werden. Die Registrierung bei der Ereignisquelle erfolgt mit Methoden der Klasse `Component`.

Maustasten-Behandlung

`void addMouseListener(MouseListener l)`
registriert das Ereignisempfänger-Objekt `l`, das das Interface `MouseListener` implementiert. Der Empfänger kann dann auf Mausklicks sowie das Betreten und Verlassen der Komponente reagieren.

Methoden von `MouseListener` sind:

`void mousePressed(MouseEvent e)`
wird beim Drücken der Maustaste aufgerufen.

`void mouseReleased(MouseEvent e)`
wird aufgerufen, wenn die gedrückte Maustaste losgelassen wurde.

`void mouseClicked(MouseEvent e)`
wird aufgerufen, wenn eine Maustaste gedrückt und wieder losgelassen wurde. Die Methode wird nach `mouseReleased` aufgerufen.

`void mouseEntered(MouseEvent e)`
wird aufgerufen, wenn der Mauszeiger sich in den Bereich der auslösenden Komponente hineinbewegt.

`void mouseExited(MouseEvent e)`
wird aufgerufen, wenn der Mauszeiger sich aus dem Bereich der auslösenden Komponente herausbewegt.

Die Adapterklasse `MouseAdapter` implementiert das Interface `MouseListener` mit leeren Methodenrümpfen.

MouseEvent

Methoden der Klasse `MouseEvent` sind:

`Point getPoint()`
liefert die x- und y-Koordinate der Position des Mauszeigers als Objekt der Klasse `Point`. Die Koordinaten werden relativ zum Ursprung der auslösenden Komponente angegeben.

`int getX()`
liefert die x-Koordinate der Position des Mauszeigers.

`int getY()`
liefert die y-Koordinate der Position des Mauszeigers.

`int getClickCount()`
liefert die Anzahl der hintereinander erfolgten Mausklicks.

`MouseEvent` ist von der Klasse `InputEvent` abgeleitet. Diese besitzt folgende Methoden:

```
boolean isAltDown()
boolean isShiftDown()
boolean isControlDown()
```
liefern den Wert `true`, wenn zusammen mit der Maustaste die Alt-, Shift- bzw. Control-Taste gedrückt wurde.

`boolean isMetaDown()`
liefert den Wert `true`, wenn die rechte Maustaste gedrückt wurde.

Maus-bewegungen

```
void addMouseMotionListener(
  MouseMotionListener l)
```
registriert das Ereignisempfänger-Objekt `l`, das das Interface `MouseMotionListener` implementiert. Der Empfänger kann dann auf die Bewegung des Mauszeigers reagieren.

Methoden von `MouseMotionListener` sind:

`void mouseMoved(MouseEvent e)`
wird aufgerufen, wenn die Maus bewegt wird, ohne dass dabei eine der Maustasten gedrückt wurde.

`void mouseDragged(MouseEvent e)`
wird aufgerufen, wenn die Maus bei gedrückter Maustaste bewegt wird.

Die Adapterklasse `MouseMotionAdapter` implementiert das Interface `MouseMotionListener` mit leeren Methodenrümpfen.

Programm 8.30 demonstriert die Mausbehandlung. Es können Ovale und Rechtecke auf einem Panel gezeichnet werden.

Das Drücken der Maustaste legt den Anfangspunkt des Ovals bzw. Rechtecks fest. Durch Ziehen der Maus wird die Größe der Figur bestimmt. Beim Loslassen der Maustaste wird die Figur in die Liste (ein Objekt der Klasse `Vector`) der bereits gezeichneten Figuren eingetragen. Bei jedem Aufruf von `paintComponent` werden alle gespeicherten Figuren sowie die aktuelle Figur neu gezeichnet.

Programm 8.30

```
// Datei: MouseTest.java
...

public class MouseTest extends JPanel {
   public static void main(String[] args) {
      ...
   }

   public MouseTest() {
      setLayout(new BorderLayout());
      Flaeche fl = new Flaeche();
      add(fl, BorderLayout.CENTER);
      JPanel p = new JPanel();
      JRadioButton oval = new JRadioButton("Oval", true);
      JRadioButton rechteck = new JRadioButton(
         "Rechteck");
      ButtonGroup formen = new ButtonGroup();
      formen.add(oval);
      formen.add(rechteck);
      p.add(oval);
      p.add(rechteck);
      add(p, BorderLayout.SOUTH);
      oval.addActionListener(fl);
      rechteck.addActionListener(fl);
   }
}
```

```
class Flaeche extends JPanel implements ActionListener {
   private Vector v;
   private int formwahl;
   private Form form;

   public Flaeche() {
      formwahl = 1;
      v = new Vector();
      form = new Form(0, 0, 0, 0, 0);
      setBackground(Color.white);
      addMouseListener(new MausTasten());
      addMouseMotionListener(new MausBewegungen());
   }

   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Form f;
      Enumeration enum;

      enum = v.elements();
      while (enum.hasMoreElements()) {
         f = (Form)enum.nextElement();
         if (f.kz == 1)
            g.drawOval(f.x, f.y, f.width, f.height);
         else if (f.kz == 2)
            g.drawRect(f.x, f.y, f.width, f.height);
      }

      if (form.width > 0 && form.height > 0)
         if (form.kz == 1)
            g.drawOval(form.x, form.y, form.width,
               form.height);
         else if (form.kz == 2)
            g.drawRect(form.x, form.y, form.width,
               form.height);
   }

   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("Oval"))
         formwahl = 1;
      else if (cmd.equals("Rechteck"))
         formwahl = 2;
```

```
    }

    class MausTasten extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            form = new Form(
                formwahl, e.getX(), e.getY(), 0, 0);
        }

        public void mouseReleased(MouseEvent e) {
            if (form.width > 0 && form.height > 0)
                v.addElement(form);
        }
    }

    class MausBewegungen extends MouseMotionAdapter {
        public void mouseDragged(MouseEvent e) {
            int x = e.getX();
            int y = e.getY();
            if (x > form.x && y > form.y) {
                form.width = x - form.x;
                form.height = y - form.y;
            }
            repaint();
        }
    }
}

class Form {
    public int kz, x, y, width, height;

    public Form(int kz, int x, int y, int width,
        int height) {
        this.kz = kz;
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

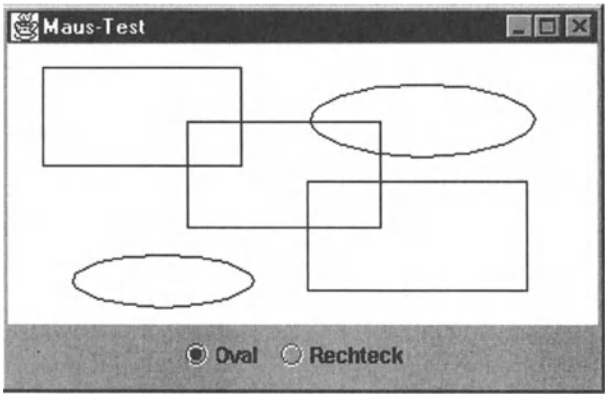
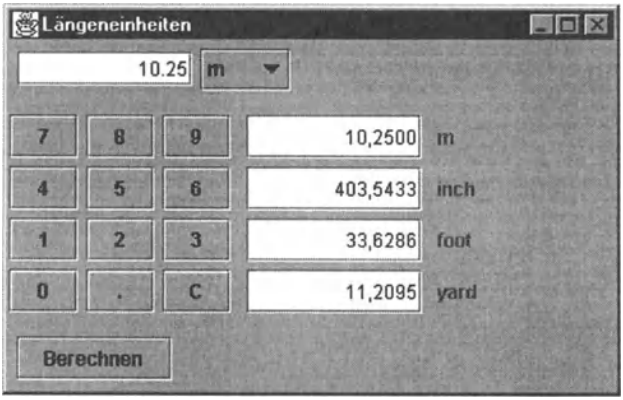


Bild 8.26: Ovale und Rechtecke zeichnen

8.9 Aufgaben

1. Schreiben Sie ein Programm, das Längenangaben in m, inch, foot und yard ineinander umrechnet. Es gelten folgende Beziehungen: 1 inch = 0,0254 m; 1 foot = 12 inch = 0,3048 m; 1 yard = 3 foot = 0,9144 m.



Die Eingabe der umzurechnenden Längenangabe kann direkt über Tastatur in einem Textfeld oder durch Drücken der Tasten eines Tastenfeldes erfolgen. Die Taste "C" löscht den Inhalt des Textfeldes. Die Maßeinheit kann über eine Auswahlliste eingestellt werden.

2. Erweitern Sie die Funktionalität von Programm 8.30. Ein Quadrat bzw. ein Kreis kann gezeichnet werden, wenn die Shift-Taste während des Aufziehens der Figur mit der Maus gedrückt wird. Die x-Koordinate des Mauszeigers bestimmt die Seitenlänge des Quadrats bzw. des den Kreis begrenzenden Quadrats. Mit Hilfe eines Markierungsfeldes (`JCheckBox`) kann bestimmt werden, ob die Figur mit der eingestellten Farbe ausgefüllt gezeichnet werden soll. Die Zeichenfarbe kann über eine Auswahlliste (`JComboBox`) eingestellt werden.

3. Schreiben Sie ein Programm, mit dem Freihandzeichnungen angefertigt werden können.

   Wie im Programm 8.30 legt das Drücken der Maustaste den Anfangspunkt der Kurve fest. Mit dem Ziehen der Maus bei gedrückter Maustaste werden die Punkte der Kurve in einen Vektor (Objekt der Klasse `Vector`) eingetragen. Beim Loslassen der Maustaste wird der Vektor, der die Punktkurve enthält, in einem Vektor der bereits gezeichneten Kurven eingetragen. Bei jedem Aufruf von `paintComponent` werden die gespeicherten Kurven sowie die aktuelle, noch nicht abgeschlossene Kurve neu gezeichnet. Dabei werden die einzelnen Punkte einer Kurve durch Linien miteinander verbunden, um evtl. Lücken zu vermeiden. Durch einen Mausklick bei gedrückter Control-Taste kann die letzte gezeichnete Kurve gelöscht werden.

   Das Fenster des Programms enthält ein Menü "Datei" mit den Menüpunkten "Neu", "Öffnen...", "Speichern", "Speichern unter..." und "Beenden". Damit kann eine neue Zeichnung angelegt werden, eine Zeichnung über einen Dialog (Klasse `JFileChooser`) geöffnet bzw.

gespeichert und das Programm beendet werden. Zum Öffnen bzw. Speichern einer Zeichnung wird der Vektor der gezeichneten Kurven mit Hilfe der Methoden der Klassen `ObjectInputStream` bzw. `ObjectOutputStream` aus einer Datei gelesen bzw. in eine Datei geschrieben.

4. Erstellen Sie einen Bildbetrachter, zur Anzeige von Bildern im gif- bzw. jpeg-Format. Die Bilddateien sollen in einem Dialog ausgewählt werden können (`JFileChooser`). Mit Hilfe eines Filters sollen nur die geeigneten Dateien zur Auswahl angeboten werden.

   Tipp: Zeigen Sie das Bild als Icon innerhalb eines mit Scrollbalken ausgestatteten Labels an.

9 Applets

Applets sind lauffähige Java-Programme, die aus einer Web-Seite heraus aufgerufen werden. Dazu muss der Bytecode des Applets mit Hilfe der Seitenbeschreibungssprache *HTML* (HyperText Markup Language) eingebunden werden. Applets werden vom Java-Interpreter eines Browsers ausgeführt. Die Ausgabe erfolgt in einem rechtwinkligen Bereich des Browserfensters. Zum Testen kann alternativ das Programm *appletviewer* aus dem Software Development Kit (SDK) benutzt werden. *appletviewer* interpretiert nur die Applet-Einträge in der HTML-Datei. Alle anderen HTML-Anweisungen werden ignoriert.

Dieses Kapitel behandelt die wesentlichen Eigenschaften eines Applets. Viele Anwendungen können so geschrieben werden, dass sie sowohl als Applet als auch als eigenständige Applikation laufen können. Die Wiedergabe von Bildern und das Abspielen von Sound-Dateien ist bei Applets und Applikationen möglich. Zuletzt wird eine Technik zur flackerfreien Bildschirmanzeige bei Animationen vorgestellt.

9.1 Der Lebenszyklus eines Applets

Um ein Applet zu erstellen, muss eine Subklasse der Klasse `JApplet`, einem speziellen Container mit *content pane* (siehe Kapitel 8.2), erstellt werden. `JApplet` selbst ist Subklasse von `Applet` aus dem Paket `java.applet`.

Das folgende Beispiel zeigt ein einfaches Applet, das einen Text in seinem Anzeigebereich ausgibt.

Programm 9.1

```
// Datei: Einfach.java
import java.awt.*;
import javax.swing.*;

public class Einfach extends JApplet {
   public void init() {
      JLabel label = new JLabel("Ein einfaches Applet",
         JLabel.CENTER);
      label.setBackground(Color.white);
      label.setForeground(Color.red);
      label.setOpaque(true);
      label.setFont(new Font("SansSerif", Font.BOLD, 24));
      getContentPane().add(label);
   }
}
```

Die HTML-Seite `Einfach.html` bindet das Applet ein:

```
<html>
<head>
<title>Ein einfaches Applet</title>
</head>
<body>
<applet code="Einfach" width="300" height="200">
</applet>
</body>
</html>
```

Die hierfür nötigen HTML-Anweisungen sind im Kapitel 9.2 beschrieben. Dort wird auch gezeigt, wie ein Applet mit einem Plug-in und speziellen Markierungen in der HTML-Seite für den *Internet Explorer* ausgeführt werden kann.

Die HTML-Datei `Einfach.html` kann von einem Browser geladen werden oder mit dem SDK-Tool *appletviewer* getestet werden: `appletviewer Einfach.html`

Ein Applet ist kein eigenständiges Programm. Die Methode `main` fehlt. Zum Starten des Applets erzeugt der Browser bzw. das Programm `appletviewer` eine Instanz der abgeleiteten Klasse.

Bild 9.1: Anzeige mit dem appletviewer

Das Applet durchläuft von dem Zeitpunkt, an dem es geladen wurde, bis zu dem, an dem kein Zugriff mehr möglich ist, einen bestimmten Zyklus. Die Übergänge zwischen den Lebensabschnitten werden durch vier Methoden der Klasse `JApplet` realisiert, die einen leeren Rumpf besitzen, jedoch in der abgeleiteten Klasse bei Bedarf überschrieben werden können.

init

`void init()`
wird genau einmal aufgerufen, nachdem das Applet geladen wurde. Hier können Instanzvariablen initialisiert, Bilder und Sounds geladen oder Parameter aus der HTML-Seite ausgewertet werden.

start

`void start()`
wird aufgerufen, nachdem die Initialisierung mit der Methode `init` abgeschlossen ist. Hier kann beispielsweise ein Thread gestartet werden. Im Gegensatz zur Initialisierung kann `start` vom Browser bzw. `appletviewer` mehrmals aufgerufen werden, z.B. nachdem das Applet gestoppt wurde.

stop

`void stop()`
wird aufgerufen, wenn die HTML-Seite, in der das Applet eingebunden ist, verlassen wird. Hier kann z.B. eine

Animation angehalten werden, um Rechenzeit zu sparen. Ein gestopptes Applet kann durch `start` wieder zum Leben erweckt werden. `stop` kann also vom Browser bzw. `appletviewer` mehrmals aufgerufen werden.

destroy

`void destroy()`
wird aufgerufen, wenn ein Applet beendet wird (z.B. beim Beenden des Browsers). Hier können alle vom Applet in Anspruch genommenen Ressourcen freigegeben und laufende Threads beendet werden.

Beim Start eines Applets werden `init` und dann `start`, beim Beenden eines Applets `stop` und dann `destroy` aufgerufen.

Das folgende Applet kann durch Aufruf der Menüpunkte *Start* und *Stop* im Programm `appletviewer` getestet werden.

Programm 9.2

```
// Datei: Zyklus.java
import java.awt.*;
import javax.swing.*;

public class Zyklus extends JApplet {
   private int zaehler;
   private JLabel label;

   public void init() {
      label = new JLabel("", JLabel.CENTER);
      label.setBackground(Color.white);
      label.setOpaque(true);
      label.setFont(new Font("SansSerif", Font.BOLD, 24));
      getContentPane().add(label);
      zaehler = 1;
      System.out.println("init");
   }

   public void start() {
      label.setText("start: " + zaehler);
      System.out.println("start");
   }
```

```
    public void stop() {
        zaehler++;
        label.setText("stop: " + zaehler);
        System.out.println("stop");
    }

    public void destroy() {
        System.out.println("destroy");
    }
}
```

9.2 Die Appletumgebung

Applet-Tag

Applets lassen sich mit HTML-Steueranweisungen (Tags genannt) in eine HTML-Seite einbinden:

`<applet Attribut=Wert ...> ... </applet>`

Innerhalb des Applet-Tags können mehrere Attribute auftreten:

Tabelle 9.1: Attribute des Applet-Tags

Attribut	Beschreibung
`code`	Name der Datei, die den ausführbaren Applet-Code (Startcode) enthält
`codebase`	URL des Verzeichnisses, das den Applet-Code enthält. Fehlt diese Angabe, wird das aktuelle Verzeichnis der HTML-Seite verwendet
`archive`	ein mit `jar` erstelltes Archiv `xyz.jar`, das die Klassen der Applet-Anwendung enthält
`name`	eindeutiger Name des Applets zur Unterscheidung mehrerer, miteinander kommunizierender Applets
`width`	Breite des Anzeigebereichs in Pixel
`height`	Höhe des Anzeigebereichs in Pixel
`align`	legt die Ausrichtung des Anzeigebereichs in einer Textzeile fest. Mögliche Werte sind:

	`left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`, `absbottom`
`vspace`	legt den freizulassenden Platz über und unter dem Anzeigebereich in Pixel fest
`hspace`	legt den freizulassenden Platz links und rechts vom Anzeigebereich in Pixel fest

Zwischen `<applet>` und `</applet>` können mehrere *Parameter-Tags* stehen:

`<param name=`*`Parameter`*` value=`*`Wert`*`>`
definiert einen Parameter, der an das Applet übergeben wird.

Object-Tag

Um ein Applet im *Internet Explorer* ausführen zu können, muss ein mit dem SDK zusammen installiertes Plug-in aktiviert werden. Die Einbindung des Applets erfolgt dann wie folgt:

```
<object classid="clsid:8AD9C840-044E-11D1-
B3E9-00805F499D93" width="nnn" height="nnn">
<param name="code" value="Klasse">
<param name="archive" value="xyz.jar">
...
</object>
```

Der *appletviewer* kann auch mit dieser Version starten.

Einige `Applet`-Methoden:

Parameter-übergabe

`String getParameter(String name)`
liefert zum Parameter `name`, der im Parameter-Tag angegeben ist, den zugehörigen Wert. Wird der angegebene Parameter nicht gefunden, wird `null` zurückgegeben.

Informations-methoden

`String[][] getParameterInfo()`
liefert ein zweidimensionales Array von Zeichenketten. Diese Methode sollte überschrieben werden und Informationen zu den vorhandenen Parametern enthalten. Jedes Element des Arrays beschreibt einen Parameter des Applets durch ein

Array mit drei Elementen: Name, Typ und Beschreibung des Parameters.

`String getAppletInfo()`
liefert einen String. Diese Methode sollte überschrieben werden und Informationen über das Applet (z.B. Name des Applets, Version, Datum, Autor, Copyright) enthalten.

Programm 9.3

```
...
<applet code="Param" width="300" height="200">
<param name="Text" value="Parameter-Test">
<param name ="Hoehe" value="100">
<param name ="Farbe" value="rot">
</applet>
...

// Datei: Param.java
import java.awt.*;
import javax.swing.*;

public class Param extends JApplet {
   private String text = "Hier steht ein Text";
   private int hoehe = 12;
   private String farbe = "schwarz";
   private Color color = Color.black;

   public String getAppletInfo() {
      return "Param, (C) Abts, 2000";
   }

   public String[][] getParameterInfo() {
      String[][] info = {
         {"Text", "String", "Anzeigetext"},
         {"Hoehe", "int", "Schriftgroesse"},
         {"Farbe", "String",
            "Schriftfarbe: rot, gelb, blau oder schwarz"}
      };
      return info;
   }
```

```
    public void init() {
        String s = getParameter("Text");

        if (s != null) {
            text = s;
        }

        s = getParameter("Hoehe");
        if (s != null) {
            try {
                hoehe = Integer.valueOf(s).intValue();
            }
            catch (NumberFormatException e) { }
        }

        s = getParameter("Farbe");
        if (s != null) {
            farbe = s;
            if (farbe.equals("rot"))
                color = Color.red;
            else if (farbe.equals("gelb"))
                color = Color.yellow;
            else if (farbe.equals("blau"))
                color = Color.blue;
            else if (farbe.equals("schwarz"))
                color = Color.black;
            else
                farbe = "schwarz";
        }

        JLabel label = new JLabel(text, JLabel.CENTER);
        label.setBackground(Color.white);
        label.setForeground(color);
        label.setOpaque(true);
        label.setFont(new Font("SansSerif", Font.BOLD,
            hoehe));
        getContentPane().add(label);
    }
}
```

Der Menüpunkt *Info* des Programms `appletviewer` zeigt die Applet- und Parameterinformationen an:

```
Param, (C) Abts, 2000

Text -- String - Anzeigetext
Hoehe -- int -- Schriftgroesse
Farbe -- String -- Schriftfarbe: rot, gelb,
  blau oder schwarz
```

Kommunikation zwischen Applets

Die Methoden des Interface `AppletContext` aus dem Paket `java.applet` können benutzt werden, um Informationen über die Web-Seite zu beschaffen, in der das Applet eingebunden ist.

`AppletContext getAppletContext()`
liefert den so genannten Applet-Kontext.

Methoden von `AppletContext`:

`Applet getApplet(String name)`
liefert, sofern vorhanden, eine Referenz auf das Applet mit dem Namen (Attribut `name` im Applet- bzw. Object-Tag) `name` oder `null`.

`Enumeration getApplets()`
liefert eine Aufzählung (siehe Kapitel 5.6) aller Applets der Web-Seite.

Programm 9.4 zeigt, wie ein Applet ein anderes Applet der gleichen Web-Seite steuern kann. Durch Drücken des Buttons *Stop* wird der Zähl-Thread des Applets `Applet2` beendet. *Start* erzeugt einen neuen Thread, der weiterzählt.

Programm 9.4

```
...
<applet code="Applet1" width="200" height="100">
</applet >
<applet code="Applet2" name="A2" width="200" height="100">
</applet>
...
```

```
// Datei: Applet1.java
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;

public class Applet1 extends JApplet
   implements ActionListener {
   public void init() {
      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      JButton start = new JButton("Start");
      start.addActionListener(this);
      c.add(start);
      JButton stop = new JButton("Stop");
      stop.addActionListener(this);
      c.add(stop);
   }

   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      AppletContext ac = getAppletContext();
      Applet2 a = (Applet2)ac.getApplet("A2");
      if (cmd.equals("Start"))
         a.start();
      else if (cmd.equals("Stop"))
         a.stop();
   }
}

// Datei: Applet2.java
import java.awt.*;
import javax.swing.*;

public class Applet2 extends JApplet implements Runnable {
   private JLabel label;
   private int zaehler;
   private Thread t;

   public void init() {
      zaehler = 0;
      label = new JLabel("", JLabel.CENTER);
```

```
        label.setForeground(Color.red);
        label.setFont(new Font("SansSerif", Font.BOLD, 24));
        getContentPane().add(label);
    }

    public void start() {
        if (t == null) {
            t = new Thread(this);
            t.start();
        }
    }

    public void stop() {
        t = null;
    }

    public void run() {
        while (t != null) {
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {}
            zaehler++;
            label.setText(String.valueOf(zaehler));
        }
    }
}
```

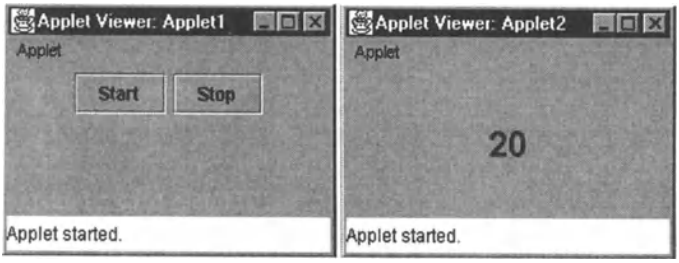


Bild 9.2: Applet-Kommunikation

Verweise auf Web-Seiten

Eine weitere Methode des Interface `AppletContext`:

`void showDocument(URL url)`
ersetzt die angezeigte Web-Seite durch die über den URL `url` angegebene Web-Seite.

Der *Uniform Resource Locator* (URL) ist die Adresse eines Objekts im *World Wide Web* (WWW). Sie hat folgende Form:

`Protokoll://Rechner[:Port]/[Datei]`

Beispiel: `http://www.fh-niederrhein.de`

Das Protokoll `http` überträgt eine Web-Seite aus dem Netz (`file` steht für die Übertragung einer Web-Seite vom lokalen Rechner). *`Port`* und *`Datei`* sind optional. *`Port`* steht für die Portnummer auf der entfernten Maschine. *`Datei`* steht für den Pfad- bzw. Dateinamen. Fehlt dieser oder ist nur der Name des Verzeichnisses angegeben, so wird ein Standardname wie z.B. `index.html` oder `default.htm` angehängt.

Objekte der Klasse `URL` (aus dem Paket `java.net`) können mit folgenden Konstruktoren erzeugt werden:

`URL(String spec)`
`URL(URL context, String spec)`
Im ersten Fall wird ein URL als Zeichenkette übergeben. Bei der zweiten Form werden in `spec` nicht enthaltene Adressteile aus `context` kopiert. Bei fehlerhaften Angaben wird die Ausnahme `MalformedURLException` ausgelöst.

Weitere Methoden der Klasse `Applet`:

`URL getDocumentBase()`
liefert den URL der Web-Seite, die das Applet eingebunden hat.

`URL getCodeBase()`
liefert den URL des Verzeichnisses, das das Applet enthält.

Programm 9.5 kann nur mit einem Browser getestet werden.

Programm 9.5

```
// Datei: Show.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.net.*;

public class Show extends JApplet
   implements ActionListener {
   private URL url1, url2;

   public void init() {
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
      JButton b = new JButton("Applet");
      b.addActionListener(this);
      c.add(b);
      b = new JButton("FH Niederrhein");
      b.addActionListener(this);
      c.add(b);
      try {
         url1 = new URL(getDocumentBase(), "Komm.html");
         url2 = new URL("http://www.fh-niederrhein.de");
      }
      catch (MalformedURLException e) { }
   }

   public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("Applet"))
         getAppletContext().showDocument(url1);
      else if (cmd.equals("FH Niederrhein"))
         getAppletContext().showDocument(url2);
   }
}
```

9.3 Hybridanwendungen

Hybridanwendungen sind ausführbare Java-Programme, die als Applet in Web-Seiten eingebunden und ohne Änderung auch als Applikation gestartet werden können. Dazu muss die von `JApplet` abgeleitete Klasse die Methode `main` enthalten. Programm 9.6 zeigt ein Beispiel.

Programm 9.6

```
// Datei RechnerApplet.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RechnerApplet extends JApplet {
   public RechnerApplet() {
      getContentPane().add(new Aufg0801());
   }

   public static void main(String[] args) {
      JFrame f = new JFrame("Rechner");
      f.getContentPane().add(new Aufg0801());
      f.pack();
      f.setVisible(true);
      f.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      } );
   }
}
```

Die Klasse `Aufg0801` implementiert die Lösung zur Aufgabe 1 in Kapitel 8.

Da zu dieser Anwendung mehrere Bytecode-Dateien gehören, fassen wir sie hier einmal in einem Archiv mittels

```
jar -cf Rechner.jar *.class
```

zusammen und nutzen den Parameter `archive` in der HTML-Seite (siehe Kapitel 9.2).

Als Applikation kann das Programm mit

```
java -cp Rechner.jar RechnerApplet
```

aufgerufen werden.

9.4 Wiedergabe von Bild- und Tondaten

Image

Objekte der Klasse `Image` aus dem Paket `java.awt` repräsentieren Bilder (Bitmaps).

```
int getHeight(ImageObserver observer)
int getWidth(ImageObserver observer)
```

liefern die Höhe bzw. Breite des Bildes oder `-1`, wenn diese noch nicht bekannt ist. `observer` vom Typ `ImageObserver` überwacht den Ladezustand des Bildes. Da die Klasse `Component` das Interface `ImageObserver` implementiert, kann bei allen Komponenten `this` übergeben werden.

Die `Applet`-Methoden

```
Image getImage(URL url)
Image getImage(URL url, String name)
```

liefern ein `Image`-Objekt zur Darstellung eines Bildes im Grafikformat `gif` oder `jpeg` (`jpg`) am Bildschirm, laden aber nicht das Bild. `url` ist die Adresse der Bilddatei bzw. des Verzeichnisses, in dem die Bilddatei `name` zu finden ist.

In Java-Applikationen werden entsprechende Methoden der Klasse `Toolkit` (aus `java.awt`) verwendet:

```
Image getImage(URL url)
Image getImage(String filename)
```

Die `Component`-Methode `Toolkit getToolkit()` liefert das `Toolkit`-Objekt für die aktuelle Umgebung.

Die folgenden `Graphics`-Methoden beginnen mit dem Zeichnen, auch wenn das gesamte Bild noch nicht geladen wurde. In diesem Fall wird `false` zurückgegeben. Der Ladevorgang wird erst bei der Darstellung des Bildes ausgelöst, so dass oft noch unfertige Bilder angezeigt werden.

```
boolean drawImage(Image img, int x, int y,
  ImageObserver observer)
boolean drawImage(Image img, int x, int y,
  int width, int height,
  ImageObserver observer)
```

zeichnen das Bild `img` mit der linken oberen Ecke an der Position `(x,y)`. Im zweiten Fall wird das Bild noch auf die Größe `width` mal `height` skaliert.

```
boolean drawImage(Image img,
  int dx1, int dy1, int dx2, int dy2,
  int sx1, int sy1, int sx2, int sy2,
  ImageObserver observer)
```

zeichnet den Ausschnitt eines Bildes, der durch die linke obere Ecke `(sx1,sy2)` und die rechte untere Ecke `(sx2,sy2)` bestimmt ist, und skaliert ihn so, dass die linke obere Ecke auf dem Punkt `(dx1,dy1)` und die rechte untere Ecke auf dem Punkt `(dx2,dy2)` liegt.

vollständiges Laden von Bildern abwarten

Oft will man mit der Anzeige warten, bis das Bild vollständig geladen ist. Hierbei helfen die Methoden der Klasse `MediaTracker` aus dem Paket `java.awt`.

```
MediaTracker(Component c)
```

erzeugt ein Objekt. `c` ist die Komponente, die die Bilder darstellen soll.

```
void addImage(Image img, int id)
```

fügt ein Bild `img` hinzu, dessen Ladevorgang überwacht werden soll. `id` bezeichnet die Priorität, mit der die Bilder geladen werden. Niedrigere Werte bezeichnen eine höhere Priorität.

```
void waitForID(int id)
  throws InterruptedException
```

startet das Laden der Bilder, die die Priorität `id` haben, und kehrt erst zurück, wenn alle Bilder vorliegen oder beim Laden Fehler aufgetreten sind.

```
void waitForAll()
  throws InterruptedException
```

startet das Laden aller Bilder und kehrt erst zurück, wenn alle Bilder vorliegen oder beim Laden Fehler aufgetreten sind.

AudioClip

Das Interface `AudioClip` aus `java.applet` deklariert drei Methoden zum Abspielen von Sound-Dateien. Die unterstützten Dateiformate sind: `AU`, `MIDI` (Typ 0 und Typ 1), `WAV`, `AIFF` und `RMF`.

`void play()`
spielt den Sound einmal ab.

`void loop()`
spielt den Sound in einer Schleife.

`void stop()`
stoppt das Abspielen des Sounds.

Die `Applet`-Methoden
`AudioClip getAudioClip(URL url)`
`AudioClip getAudioClip(URL url, String name)`
erzeugen ein `AudioClip`-Objekt für Tondaten. Beim Aufruf einer Abspielmethode werden die Tondaten geladen. `url` ist die Adresse der Sound-Datei bzw. des Verzeichnisses, in dem die Sound-Datei `name` zu finden ist.

Sound-Dateien können auch in Applikationen abgespielt werden. Die `Applet`-Klassenmethode
`static AudioClip newAudioClip(URL url)`
liefert ein `AudioClip`-Objekt.

Programm 9.7 demonstriert das Laden und die Wiedergabe von Bild und Ton. Die Namen der Bild- und Sound-Datei werden als Parameter aus der HTML-Seite gelesen.

Programm 9.7

```
// Datei: Media.java
import java.awt.*;
import java.applet.*;
import javax.swing.*;

public class Media extends JApplet {
   private Image img;
   private AudioClip ac;

   public void init() {
      String image = getParameter("Image");
      String audio = getParameter("AudioClip");
```

```
        ac = getAudioClip(getDocumentBase(), audio);
        img = getImage(getDocumentBase(), image);
        MediaTracker tracker = new MediaTracker(this);
        tracker.addImage(img, 0);
        try {
           tracker.waitForAll();
        }
        catch (InterruptedException e) { }
        getContentPane().add(new BildPanel(img));
     }

     public void start() {
        ac.play();
     }

     public void stop() {
        ac.stop();
     }
  }

  class BildPanel extends JPanel {
     private Image img;
     private int x, y;

     public BildPanel(Image img) {
        this.img = img;
     }

     public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int b = getSize().width;
        int h = getSize().height;
        int bImg = img.getWidth(this);
        int hImg = img.getHeight(this);
        x = (b - bImg) / 2;
        y = (h - hImg) / 2;
        g.drawImage(img, x, y, this);
     }
  }
```

9.5 Animation

Bewegungen (Animation) in einem Applet oder einer Applikation sollten in einem eigenen Thread laufen, um den Browser nicht lahmzulegen bzw. damit das Programm auch noch auf Benutzereingaben reagieren kann.

Die `Component`-Methode `repaint` wird immer dann aufgerufen, wenn neu gezeichnet werden muss.

Double-Buffering

Ein mit dem Standardkonstruktor erzeugtes `JPanel`-Objekt nutzt das so genannte *Double-Buffering*: Es wird zunächst in einen *Offscreen*-Puffer gezeichnet, der anschließend, wenn alle Ausgabeoperationen abgeschlossen sind, auf den Bildschirm kopiert wird. Bei einer Animation vermeidet man so ein unschönes Flackern der Bildschirmanzeige.

Programm 9.8 implementiert ein animiertes Applet. Ein Auto (`Auto.gif`) bewegt sich schrittweise (`x += 2`) von links nach rechts über den Anzeigebereich des Applets. Der Ablauf ist leicht verzögert.

Programm 9.8

```
// Datei: Auto.java
import java.awt.*;
import javax.swing.*;

public class Auto extends JApplet {
   private AutoPanel auto;
   private Thread t;

   public void init() {
      Image img = getImage(getDocumentBase(), "Auto.gif");
      MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(img, 0);
      try {
         tracker.waitForAll();
      }
      catch (InterruptedException e) { }
      auto = new AutoPanel(img);
      getContentPane().add(auto);
   }
```

```
    public void start() {
        if (t == null) {
            t = new Thread(auto);
            t.start();
            auto.setStop(false);
        }
    }

    public void stop() {
        t = null;
        auto.setStop(true);
    }
}

class AutoPanel extends JPanel implements Runnable {
    private Image img;
    private int x = -111;
    private boolean stop = false;

    public AutoPanel(Image img) {
        this.img = img;
        setBackground(Color.white);
    }

    public void setStop(boolean stop) {
        this.stop = stop;
    }

    public void run() {
        while (!stop) {
            if (x > 700)
                x = -111;
            x += 2;
            repaint();
            try {
                Thread.sleep(20);
            }
            catch (InterruptedException e) { }
        }
    }
```

```
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawLine(0, 126, 700, 126);
        g.drawImage(img, x, 80, 111, 46, this);
    }
}
```

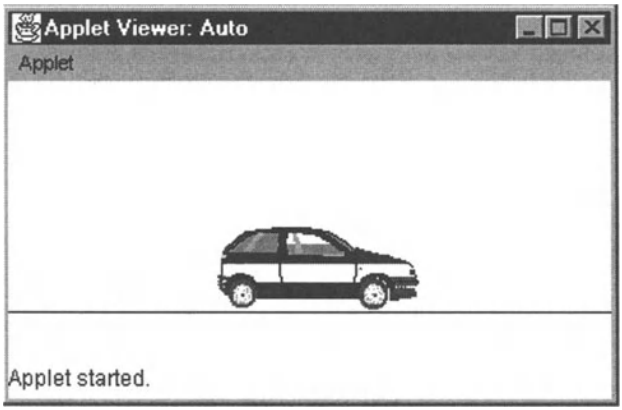


Bild 9.3: Ein animiertes Applet (Snapshot)

9.6 Aufgaben

1. Realisieren Sie ein Applet, das eine Ampel anzeigt, die automatisch von Rot über Gelb auf Grün und wieder zurück umschaltet. Dabei sollen die Rot- und Grünphase sechs Sekunden und die Gelbphase eine Sekunde dauern.

2. Schreiben Sie ein Applet, das farbige, kreisförmige Flecken zeichnet. Position, Größe des Radius und Farbe sollen zufällig sein. Ebenso sollen die Zeitabstände zwischen den Zeichenvorgängen zufällig verteilt sein. Bereits gezeichnete Flecken sollen sichtbar bleiben.

Tipp: Entfernen Sie den Aufruf von `super.paintComponent(g)` aus `paintComponent`.

3. Schreiben Sie Programm 8.30 aus Kapitel 8 als Hybridanwendung, die sowohl als Applet als auch als Applikation laufen kann.

4. Schreiben Sie ein Applet, das auf einer schwarzen Fläche von Zeit zu Zeit Blitze vom oberen bis zum unteren Rand des Anzeigebereichs zucken lässt. Position und Form der Blitze sind mit Hilfe eines Zufallsgenerators festzulegen.

5. Ausgehend von einer Startposition soll ein Ball (ausgefüllter Kreis) sich diagonal über den Applet-Anzeigebereich bewegen. Trifft der Ball den Rand des Bereichs, muss die Bewegungsrichtung so geändert werden, dass er wie ein Gummiball abprallt (Bouncing Ball).

6. In einem Applet soll eine Folge von Bildern (z.B. `Bild1.gif` bis `Bild10.gif`) mit Sound abgespielt werden. Als Parameter benötigt das Applet: den Namensstamm der Bilddateien (z.B. `Bild`), die Erweiterung (z.B. `gif`), die Nummer des ersten Bildes, die Nummer des letzten Bildes, die pro Sekunde anzuzeigende Anzahl Bilder und den Namen der Sound-Datei.

10 Datenbank- und Netzanwendungen

Dieses Kapitel enthält Anwendungsbeispiele, die den Zugriff auf relationale Datenbanken aus Java-Programmen heraus sowie die Erstellung von Client-Server-Anwendungen auf der Basis von Netzwerkverbindungen demonstrieren. Hierzu existieren umfangreiche Sammlungen von Klassen. Im Folgenden werden jedoch nur die Methoden vorgestellt, die für die Realisierung der Beispiele benötigt werden.

10.1 Datenbankzugriffe mit JDBC

JDBC API

Das Paket `java.sql` bietet eine Programmierschnittstelle für den Zugriff auf relationale Datenbanken mit Hilfe der Standard-Datenbanksprache SQL. Die hierzu erforderlichen Klassen und Methoden werden als *JDBC API* bezeichnet. Der Name *JDBC* wird als Abkürzung für *Java Database Connectivity* verwendet. Ein Programm kann mittels JDBC unabhängig vom verwendeten Datenbanksystem geschrieben werden. Somit ist ein Wechsel des Datenbanksystems ohne Änderung des Java-Programms möglich.

JDBC-Treiber

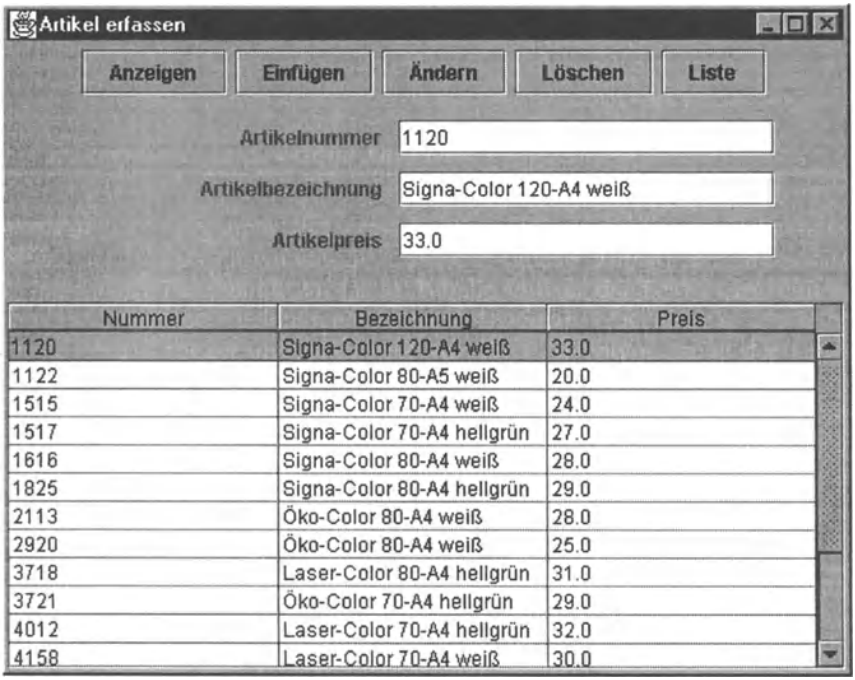
Die Datenbankanbindung wird über einen datenbanksystemspezifischen *JDBC-Treiber* realisiert. Dieser Treiber versteht die JDBC-Befehle, übersetzt sie in datenbanksystemspezifische Befehle und leitet sie an das Datenbanksystem zur Ausführung weiter. Mit der *JDBC-ODBC-Bridge* können vorhandene ODBC-Treiber verwendet werden. *ODBC* (Open Database Connectivity) ist eine von Microsoft definierte Datenbankschnittstelle. Die JDBC-Befehle werden von der JDBC-ODBC-Bridge in ODBC-Befehle übersetzt und an den ODBC-Treiber weitergeleitet.

In diesem Kapitel werden die einzelnen Klassen und Methoden vorgestellt, die für die Realisierung des Anwendungsbeispiels nötig sind.

Anwendungsbeispiel

Programm 10.1 ermöglicht die Pflege eines Artikelbestandes. Das Programm greift auf die Tabelle *Artikel* einer *MS-Access*-Datenbank zu. Diese Tabelle enthält die Felder *Artikelnummer* (Feldname: `ArtNr`, Datentyp: `Integer`), *Artikelbezeichnung* (Feldname: `ArtBez`, Datentyp: `Text` der Länge 40) und *Artikelpreis* (Feldname: `ArtPreis`, Datentyp: `Double`). Die Artikelnummer stellt den (eindeutigen) Primärschlüssel dar.

Bild 10.1: Formular zur Artikelpflege



Folgende Funktionen, die über Buttons ausgelöst werden, stehen zur Verfügung:

- *Anzeigen*
  Zur vorgegebenen Artikelnummer wird der Datensatz angezeigt.
- *Einfügen*
  Ein neuer Datensatz wird in die Tabelle Artikel aufgenommen.
- *Ändern*
  Ein bestehender Datensatz wird geändert.
- *Löschen*
  Ein Datensatz wird aus der Tabelle Artikel entfernt.
- *Liste*
  Alle Datensätze der Tabelle Artikel werden angezeigt.

Eine im Formular selektierte Tabellenzeile wird in die entsprechenden Formularfelder kopiert.

Für die MS-Access-Datenbank muss eine ODBC-Treiber-Verbindung eingerichtet werden. Bei Windows muss zu diesem Zweck das Programm *ODBC* aus der Systemsteuerung aufgerufen und der MS-Access-Treiber sowie die MS-Access-Datenbank ausgewählt werden. Der Name dieser Treiber-Verbindung (Datenquellenname) wird im Programm benutzt.

Treiber laden

Die statische Methode `Class.forName(`*`Treibername`*`)` lädt den JDBC-Treiber (siehe Kapitel 5.9).

Im Beispielprogramm 10.1 wird die JDBC-ODBC-Bridge `"sun.jdbc.odbc.JdbcOdbcDriver"` als Treiber gewählt.

Fehlermeldungen

Alle im Folgenden beschriebenen Methoden lösen die Ausnahme `SQLException` aus, wenn beim Zugriff auf die Datenbank ein Fehler aufgetreten ist.

Methoden der Klasse `SQLException`:

`String getSQLState()`
liefert eine Rückmeldung der Datenbank über den aufgetretenen Fehler.

`SQLException getNextException()`
liefert die nächste Ausnahme vom Typ `SQLException` oder `null`.

Verbindung herstellen

Die Methoden
```
static Connection getConnection(String url)
static Connection getConnection(String url,
  String user, String password)
```
der Klasse `DriverManager` stellen eine Verbindung zur Datenbank her und liefern ein Objekt, das das Interface `Connection` implementiert.

`url` ist der Name der Datenbank. Im Beispiel ist `url` die Zeichenkette `"jdbc:odbc:Artikel"`, wobei `Artikel` der Datenquellenname der ODBC-Treiber-Verbindung ist. Die zweite Methode ermöglicht die Angabe von Benutzername und Passwort für den Zugang zur Datenbank.

SQL-Anweisungen ausführen

Innerhalb einer Verbindung können SQL-Anweisungen ausgeführt und Ergebnisse gelesen werden.

Einige Methoden des Interface `Connection`:

`Statement createStatement()`
erzeugt ein Objekt, das das Interface `Statement` implementiert. Dieses Objekt repräsentiert eine SQL-Anweisung.

`void close()`
hebt die Verbindung zur Datenbank auf.

Einige Methoden des Interface `Statement`:

`ResultSet executeQuery(String sql)`
führt die SQL-Abfrage `sql` aus. Das Ergebniss wird als `ResultSet`-Objekt geliefert.

`void close()`
gibt die von der SQL-Anweisung benutzten Ressourcen frei.

Im Beispielprogramm werden folgende Abfragen verwendet:

select

`select * from Artikel order by ArtNr`
liefert alle Datensätze der Tabelle Artikel, sortiert nach der Artikelnummer.

`select * from Artikel where ArtNr = nr`
liefert den Datensatz zur Artikelnummer *nr*. *nr* steht hier für ein Literal.

`int executeUpdate(String sql)`
führt die `insert`-, `update`- oder `delete`-Anweisung `sql` aus und liefert die Anzahl der erfolgreich bearbeiteten Datensätze.

Im Beispielprogramm werden folgende SQL-Anweisungen verwendet:

insert

`insert into Artikel (ArtNr, ArtBez, ArtPreis) values (nr, bez, preis)`
fügt einen Datensatz mit den Werten *nr*, *bez* und *preis* in die Tabelle Artikel ein. *nr*, *bez* und *preis* stehen hier für Literale.

update

`update Artikel set ArtBez = bez, ArtPreis = preis where ArtNr = nr`
überschreibt die Artikelbezeichnung und den Artikelpreis mit den Werten *bez* bzw. *preis* im Datensatz mit der Artikelnummer *nr*. *nr*, *bez* und *preis* stehen hier für Literale.

delete

`delete from Artikel where ArtNr = nr`
löscht den Datensatz mit der Artikelnummer *nr*. *nr* steht hier für ein Literal.

Einige Methoden des Interface `ResultSet`:

`boolean next()`
stellt den nächsten Datensatz zur Verfügung und liefert `true` oder liefert `false`, wenn keine weiteren Datensätze vorliegen. Der erste Aufruf der Methode liefert den ersten Datensatz (falls vorhanden).

```
String getString(int n)
int getInt(int n)
double getDouble(int n)
```

lieferen den Wert in der n-ten Spalte als String, `int`-Wert bzw. `double`-Wert. Die Spaltennummerierung beginnt bei 1.

Programm 10.1

```
// Datei: DBDemo.java
...

public class DBDemo {
   private String treiber ="sun.jdbc.odbc.JdbcOdbcDriver";
   private String url = "jdbc:odbc:Artikel";
   private Connection con;
   private Datenmodell model;

   public static void main(String[] args) {
      new DBDemo();
   }

   public DBDemo() {
      try {
         Class.forName(treiber);
         con = DriverManager.getConnection(url);
      }
      catch (Exception ex) {
         System.out.println(getFehler(ex,
            "Verbindung zur DB herstellen"));
         System.exit(1);
      }

      model = new Datenmodell();
      JFrame frame = new JFrame("Artikel erfassen");
      frame.getContentPane().add(new GUI(this));
      frame.pack();
      frame.setVisible(true);
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            closeDB();
            System.exit(0);
         }
      } );
   }
```

```
public void closeDB() {
   try {
      con.close();
   }
   catch (Exception ex) {
      System.out.println(getFehler(ex,
         "Verbindung zur DB aufheben"));
   }
}

public String getFehler(Exception ex, String kennung) {
   StringBuffer sb = new StringBuffer();
   sb.append(kennung + "\n");
   sb.append(ex + "\n");
   if (ex instanceof SQLException) {
      sb.append("SQLState: " +
         ((SQLException)ex).getSQLState() + "\n");
      SQLException e;
      while (
         (e = ((SQLException)ex).getNextException()) !=
            null)
         sb.append(e.getSQLState() + "\n");
   }
   return sb.toString();
}

public Datenmodell getDatenmodell() {
   return model;
}

public boolean artikelVorhanden(int ArtNr)
   throws Exception {
   String sql =
      "select ArtNr from Artikel where ArtNr = " +
      ArtNr;
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery(sql);
   boolean found = false;
   if (rs.next())
      found = true;
   stmt.close();
   return found;
}
```

```
public Artikel getArtikel(int ArtNr) throws Exception {
   Artikel a = null;
   String sql = "select * from Artikel where ArtNr = "
      + ArtNr;
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery(sql);
   if (rs.next()) {
      a = new Artikel();
      a.setArtNr(rs.getInt(1));
      a.setArtBez(rs.getString(2));
      a.setArtPreis(rs.getDouble(3));
   }
   stmt.close();
   return a;
}

public void insertArtikel(Artikel a) throws Exception {
   String sql = "insert into Artikel (ArtNr, ArtBez, "
      + "ArtPreis) values (" + a.getArtNr() + ",'" +
      a.getArtBez() + "'," + a.getArtPreis() + ")";
   Statement stmt = con.createStatement();
   stmt.executeUpdate(sql);
   stmt.close();
}

public void updateArtikel(Artikel a) throws Exception {
   String sql = "update Artikel set ArtBez = '" +
      a.getArtBez() + "', ArtPreis = " +
      a.getArtPreis() + " where ArtNr = " +
      a.getArtNr();
   Statement stmt = con.createStatement();
   stmt.executeUpdate(sql);
   stmt.close();
}

public void deleteArtikel(int ArtNr) throws Exception {
   String sql = "delete from Artikel where ArtNr = " +
      ArtNr;
   Statement stmt = con.createStatement();
   stmt.executeUpdate(sql);
   stmt.close();
}
```

```
    public void setTabelle() throws Exception {
        Vector liste = model.getDaten();
        liste.removeAllElements();
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(
            "select * from Artikel order by ArtNr");
        Artikel a;
        while (rs.next()) {
            a = new Artikel();
            a.setArtNr(rs.getInt(1));
            a.setArtBez(rs.getString(2));
            a.setArtPreis(rs.getDouble(3));
            liste.add(a);
        }
        stmt.close();
        model.fireTableDataChanged();
    }
}

class Artikel {
    private int artNr;
    private String artBez;
    private double artPreis;

    public int getArtNr() {
        return artNr;
    }

    public String getArtBez() {
        return artBez;
    }

    public double getArtPreis() {
        return artPreis;
    }

    public void setArtNr(int artNr) {
        this.artNr = artNr;
    }

    public void setArtBez(String artBez) {
        this.artBez = artBez;
    }
```

```
    public void setArtPreis(double artPreis) {
        this.artPreis = artPreis;
    }
}

class Datenmodell extends AbstractTableModel {
    String[] namen = {"Nummer", "Bezeichnung", "Preis"};
    Vector daten = new Vector();

    public Vector getDaten() {
        return daten;
    }

    public int getColumnCount() {
        return namen.length;
    }

    public int getRowCount() {
        return daten.size();
    }

    public Object getValueAt(int zeile, int spalte) {
        Artikel a = (Artikel)daten.elementAt(zeile);
        if (spalte == 0)
            return String.valueOf(a.getArtNr());
        else if (spalte == 1)
            return a.getArtBez();
        else
            return String.valueOf(a.getArtPreis());
    }

    public String getColumnName(int spalte) {
        return namen[spalte];
    }
}

class GUI extends JPanel implements ActionListener,
    ListSelectionListener {
    private DBDemo anwendung;
    private JTextField t1 = new JTextField();
    private JTextField t2 = new JTextField();
    private JTextField t3 = new JTextField();
    private JLabel msg = new JLabel(" ");
```

```
private JTable table = new JTable();

public GUI(DBDemo anw) {
   anwendung = anw;
   setLayout(new BorderLayout(10, 10));

   JPanel p1 = new JPanel();
   p1.setLayout(
      new FlowLayout(FlowLayout.CENTER, 5, 5));
   JButton b1 = new JButton("Anzeigen");
   b1.addActionListener(this);
   JButton b2 = new JButton("Einfügen");
   b2.addActionListener(this);
   JButton b3 = new JButton("Ändern");
   b3.addActionListener(this);
   JButton b4 = new JButton("Löschen");
   b4.addActionListener(this);
   JButton b5 = new JButton("Liste");
   b5.addActionListener(this);
   p1.add(b1);
   p1.add(b2);
   p1.add(b3);
   p1.add(b4);
   p1.add(b5);

   JPanel p2 = new JPanel();
   p2.setLayout(new GridLayout(3, 2, 10, 10));
   p2.add(new JLabel("Artikelnummer", JLabel.RIGHT));
   p2.add(t1);
   p2.add(new JLabel("Artikelbezeichnung",
      JLabel.RIGHT));
   p2.add(t2);
   p2.add(new JLabel("Artikelpreis", JLabel.RIGHT));
   p2.add(t3);

   JPanel p3 = new JPanel();
   p3.setLayout(new BorderLayout());
   p3.add(msg, BorderLayout.NORTH);
   JScrollPane sp = new JScrollPane(table);
   p3.add(sp, BorderLayout.CENTER);

   add(p1, BorderLayout.NORTH);
   add(p2, BorderLayout.CENTER);
```

```
        add(Box.createHorizontalStrut(30),
            BorderLayout.EAST);
        add(p3, BorderLayout.SOUTH);
        msg.setForeground(Color.red);
        table.setModel(anwendung.getDatenmodell());
        table.setPreferredScrollableViewportSize(
            new Dimension(500, 200));
        table.getSelectionModel().addListSelectionListener(
            this);
    }

    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if (cmd.equals("Anzeigen"))
            doAnzeigen();
        else if (cmd.equals("Einfügen"))
            doEinfuegen();
        else if (cmd.equals("Ändern"))
            doAendern();
        else if (cmd.equals("Löschen"))
            doLoeschen();
        else if (cmd.equals("Liste"))
            doListe();
    }

    public void valueChanged(ListSelectionEvent e) {
        int row = table.getSelectedRow();
        if (row < 0)
            return;
        t1.setText((String)table.getValueAt(row, 0));
        t2.setText((String)table.getValueAt(row, 1));
        t3.setText((String)table.getValueAt(row, 2));
    }

    public void doAnzeigen() {
        int nr;
        t2.setText("");
        t3.setText("");
        msg.setText(" ");
        try {
            nr = Integer.valueOf(t1.getText()).intValue();
        }
        catch (NumberFormatException ex) {
```

```
            msg.setText(
               "Artikelnummer muss eine ganze Zahl sein");
            return;
         }

         try {
            Artikel a = anwendung.getArtikel(nr);
            if (a == null)
               msg.setText("Artikelnummer " + nr +
                  " nicht vorhanden");
            else {
               t2.setText(a.getArtBez());
               t3.setText(String.valueOf(a.getArtPreis()));
            }
         }
         catch (Exception ex) {
            System.out.println(
               anwendung.getFehler(ex, "Anzeigen"));
            return;
         }
      }

      public void doEinfuegen() {
         int nr;
         String bez;
         double preis;
         msg.setText(" ");

         try {
            nr = Integer.valueOf(t1.getText()).intValue();
         }
         catch (NumberFormatException ex) {
            msg.setText(
               "Artikelnummer muss eine ganze Zahl sein");
            return;
         }

         bez = t2.getText();
         if (bez.length() == 0) {
            msg.setText("Artikelbezeichnung fehlt");
            return;
         }
```

```
            try {
               preis = Double.valueOf(
                  t3.getText()).doubleValue();
            }
            catch (NumberFormatException ex) {
               msg.setText("Artikelpreis muss eine Zahl " +
                  "(z.B. 12.5) sein");
               return;
            }

            Artikel a = new Artikel();
            a.setArtNr(nr);
            a.setArtBez(bez);
            a.setArtPreis(preis);

            try {
               if (anwendung.artikelVorhanden(nr)) {
                  msg.setText("Artikelnummer " + nr +
                     " ist bereits vorhanden");
                  return;
               }
               anwendung.insertArtikel(a);
               msg.setText("Artikel wurde eingefügt");
            }
            catch (Exception ex) {
               System.out.println(
                  anwendung.getFehler(ex, "Einfuegen"));
               return;
            }
         }

         public void doAendern() {
            int nr;
            String bez;
            double preis;
            msg.setText(" ");

            try {
               nr = Integer.valueOf(t1.getText()).intValue();
            }
            catch (NumberFormatException ex) {
               msg.setText(
                  "Artikelnummer muss eine ganze Zahl sein");
```

```
            return;
         }

         bez = t2.getText();
         if (bez.length() == 0) {
            msg.setText("Artikelbezeichnung fehlt");
            return;
         }

         try {
            preis = Double.valueOf(
               t3.getText()).doubleValue();
         }
         catch (NumberFormatException ex) {
            msg.setText("Artikelpreis muss eine Zahl " +
               "(z.B. 12.5) sein");
            return;
         }

         Artikel a = new Artikel();
         a.setArtNr(nr);
         a.setArtBez(bez);
         a.setArtPreis(preis);

         try {
            if (!anwendung.artikelVorhanden(nr)) {
               msg.setText("Artikelnummer " + nr +
                  " nicht vorhanden");
               return;
            }
            anwendung.updateArtikel(a);
            msg.setText("Artikel wurde geändert");
         }
         catch (Exception ex) {
            System.out.println(
               anwendung.getFehler(ex, "Aendern"));
            return;
         }
      }

      public void doLoeschen() {
         int nr;
         msg.setText(" ");
```

```
            try {
               nr = Integer.valueOf(t1.getText()).intValue();
            }
            catch (NumberFormatException ex) {
               msg.setText(
                  "Artikelnummer muss eine ganze Zahl sein");
               return;
            }

            try {
               if (!anwendung.artikelVorhanden(nr)) {
                  msg.setText("Artikelnummer " + nr +
                     " nicht vorhanden");
                  return;
               }
               anwendung.deleteArtikel(nr);
               msg.setText("Artikel wurde gelöscht");
            }
            catch (Exception ex) {
               System.out.println(
                  anwendung.getFehler(ex, "Loeschen"));
               return;
            }
         }

         public void doListe() {
            t1.setText("");
            t2.setText("");
            t3.setText("");
            msg.setText(" ");

            try {
               anwendung.setTabelle();
            }
            catch (Exception ex) {
               System.out.println(
                  anwendung.getFehler(ex, "Liste"));
               return;
            }
         }
      }
```

10.2 Netzwerkprogrammierung

Für die Kommunikation über ein Netzwerk auf der Basis von *TCP/IP* (Transmission Control Protocol/Internet Protocol) stellt das Paket `java.net` die erforderlichen Klassen und Methoden zur Verfügung.

URL

Der *Uniform Resource Locator* (URL) wurde bereits im Kapitel 9.2 vorgestellt. Er stellt die Adresse eines Objekts im *World Wide Web* (WWW) dar. Objekte der Klasse `URL` repräsentieren solche Adressen.

Die `URL`-Methode
`InputStream openStream() throws IOException`
stellt eine Verbindung zum Objekt dieses `URL` her und liefert ein `InputStream`-Objekt (siehe Kapitel 6), um es zu lesen.

Programm 10.2 liest eine Textdatei, deren Name über einen URL angegeben wird.

Programm 10.2

```
// Datei: URLLesen.java
...

public class URLLesen extends JPanel
   implements ActionListener {
   private JTextField tf = new JTextField(30);
   private JTextArea ta = new JTextArea(15, 40);

   public static void main(String[] args) {
      JFrame frame = new JFrame("URL lesen");
      frame.getContentPane().add(new URLLesen());
      frame.pack();
      frame.setVisible(true);
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      } );
   }

   public URLLesen() {
      setLayout(new BorderLayout());
      JPanel p = new JPanel();
      p.add(new JLabel("URL", JLabel.RIGHT));
```

```
        p.add(tf);
        JButton b = new JButton("Öffnen");
        p.add(b);
        add(p, BorderLayout.CENTER);
        JScrollPane sp = new JScrollPane(ta);
        add(sp, BorderLayout.SOUTH);
        ta.setEditable(false);
        tf.addActionListener(this);
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        doOeffnen();
    }

    public void doOeffnen() {
        URL url;
        ta.setText("");
        try {
            url = new URL(tf.getText());
        }
        catch (MalformedURLException e) {
            ta.append("Fehlerhafte URL\n");
            ta.append(e.getMessage());
            return;
        }

        try {
            BufferedReader ein = new BufferedReader(
                new InputStreamReader(url.openStream()));
            String zeile;
            while ((zeile = ein.readLine()) != null)
                ta.append(zeile + "\n");
            ein.close();
        }
        catch (IOException e) {
            ta.append("I/O-Fehler\n");
            ta.append(e.getMessage());
        }
    }
}
```

Bild 10.2: Datei aus dem Netz lesen

Eine Datei des lokalen Rechners kann über das Protokoll `file` gelesen werden.

Beispiel: `file:C:/autoexec.bat`

Client und Server

Eine *Client-Server-Anwendung* ist eine *geteilte Anwendung*, in der die Verarbeitung zu einem Teil vom *Client* und zum anderen Teil vom *Server* vorgenommen wird. Das Client-Programm (z.B. ein Erfassungsformular mit Eingabeprüfung) erstellt einen Auftrag (z.B. "Ermittle den Namen des Kunden mit der Kundennummer 4711") und schickt ihn an das Server-Programm (z.B. ein Programm, das in einer Datenbank sucht), das ihn annimmt, bearbeitet und das Ergebnis als Antwort zurückschickt. Ein Client kann sich auf demselben Rechner wie der Server oder auf einem anderen, über das Netz verbundenen Rechner befinden.

Socket

So genannte *Sockets* stellen die Endpunkte einer TCP/IP-Verbindung zwischen Client und Server dar. Sie stellen eine Schnittstelle für den Datenaustausch (Lesen und Schreiben) über das Netz zur Verfügung.

Programm 10.3 implementiert einen Client, der eine Verbindung mit einem textgesteuerten Server aufnehmen kann. Eingaben für den Server werden über die Kommandozeile abgeschickt. Der Client schickt eine Textzeile und erwartet vom Server eine Textzeile als Antwort.

Die Klasse `Socket` implementiert das clientseitige Ende einer Netzwerkverbindung.

`Socket(String host, int port) throws UnknownHostException, IOException`
erzeugt ein `Socket`-Objekt. `host` ist der Name bzw. die IP-Adresse des Rechners, auf dem der Server läuft. `port` ist die Nummer des Netzwerk-Ports auf dem Zielrechner. Ein Server bietet seinen Dienst immer über eine Portnummer an.

Methoden der Klasse `Socket`:

`InputStream getInputStream() throws IOException`
liefert einen Datenstrom, um über diesen Socket zu lesen.

`OutputStream getOutputStream() throws IOException`
liefert einen Datenstrom, um über diesen Socket zu senden.

`void close() throws IOException`
schließt den Socket.

Programm 10.3

```
// Datei: Client.java
import java.net.*;
import java.io.*;

public class Client {
   public static void main(String[] args) {
      Socket so = null;

      String host = args[0];
      int port = Integer.valueOf(args[1]).intValue();
```

```
        // Socket erstellen
           try {
              so = new Socket(host, port);
           }
        catch (UnknownHostException e) {
           System.out.println(e);
           System.exit(1);
        }
        catch (IOException e) {
           System.out.println(e);
           System.exit(1);
        }

        // Verarbeitungsschleife
        try {
              BufferedReader ein = new BufferedReader(
              new InputStreamReader(System.in));
              BufferedReader einSo = new BufferedReader(
              new InputStreamReader(so.getInputStream()));
           PrintWriter ausSo = new PrintWriter(
              so.getOutputStream(), true);

           System.out.println(
              "> Bitte Auftrag an Server eingeben:");

           String eingabe;
              while ((eingabe = ein.readLine()) != null) {
              ausSo.println(eingabe);
                 System.out.println(
                    "> Antwort vom Server:");
              System.out.println(einSo.readLine());
              System.out.println(
                 "> Bitte Auftrag an Server eingeben:");
              }
              ein.close();
           so.close();
        }
        catch (IOException e) {
           System.out.println(e);
           System.exit(1);
        }
     }
  }
```

Zum Testen kann das Programm z.B. wie folgt aufgerufen werden:

```
java Client www.fh-niederrhein.de 7
```

Jeder Rechner im Internet verfügt über einen Dienst am Port 7, der die empfangenen Daten unverändert an den Sender zurückschickt (Echo-Server).

Beispiel-Sitzung:

```
> Bitte Auftrag an Server eingeben:
Hallo
> Antwort vom Server:
Hallo
> Bitte Auftrag an Server eingeben:
Das ist ein Test
> Antwort vom Server:
Das ist ein Test
> Bitte Auftrag an Server eingeben:
```

`Strg+Z` beendet die Eingabeschleife.

Programm 10.4 implementiert einen Server, der zu einer vom Client gesendeten Kundennummer den Kundennamen in einer Datei sucht und im Erfolgsfall zurückschickt.

ServerSocket

Die Klasse `ServerSocket` implementiert das serverseitige Ende einer Netzwerkverbindung.

`ServerSocket(int port) throws IOException`
erzeugt ein `ServerSocket`-Objekt für den Port `port`.

Die `ServerSocket`-Methode
`Socket accept() throws IOException`
wartet auf eine Verbindungsanforderung durch einen Client. Die Methode wartet, bis die Verbindung hergestellt ist. Sie liefert ein `Socket`-Objekt, über das Daten vom Client empfangen bzw. an den Client gesendet werden können.

Damit Server mehrere Clients gleichzeitig bedienen können, wird die eigentliche Berarbeitung des Client-Auftrags in einem Thread realisiert.

`void close() throws IOException`
schließt den Server-Socket.

Programm 10.4

```
// Datei: Server.java
import java.io.*;
import java.net.*;

public class Server {
   public static void main(String[] args) {
      ServerSocket serverSo = null;

      int port = Integer.valueOf(args[0]).intValue();

      // Server-Socket erzeugen
      try {
         serverSo = new ServerSocket(port);
      }
      catch (IOException e) {
         System.out.println(e);
         System.exit(1);
      }

      System.out.println(
         "Server lauscht auf Port " + port);

      // Verbindungsanforderungen neuer Clients annehmen
      // und bearbeiten
      int clientNr = 0;
      while (true) {
         try {
            Socket clientSo = serverSo.accept();
            clientNr++;
            System.out.println(
               "Mit Client " + clientNr + " verbunden");
            new ServerThread(clientSo, clientNr);
         }
         catch (IOException e) {
            System.out.println(e);
            System.exit(1);
         }
      }
   }
}
```

```
// Datei: ServerThread.java
import java.io.*;
import java.net.*;
import java.util.*;

public class ServerThread extends Thread {
   private Socket clientSo;
   private int clientNr;

   public ServerThread(Socket clientSo, int clientNr) {
      this.clientSo = clientSo;
      this.clientNr = clientNr;
      start();
   }

   public void run() {
      try {
         BufferedReader einSo =
            new BufferedReader(new InputStreamReader(
               clientSo.getInputStream()));
         PrintWriter ausSo = new PrintWriter(
               clientSo.getOutputStream(), true);
         String zeile, nr;
         while ((nr = einSo.readLine()) != null) {
            zeile = suche(nr);
            ausSo.println(zeile);
         }
         clientSo.close();
      }
      catch (IOException e) {
         System.out.println(e);
      }
      finally {
         System.out.println(
            "Client " + clientNr + " beendet");
      }
   }

   private String suche(String nr) {
      String zeile;
      StringTokenizer st;
      String knr = null, kname = null;
      boolean found = false;
```

```
        try {
            BufferedReader br = new BufferedReader(
                new FileReader("Kunden.txt"));
            while ((zeile = br.readLine()) != null) {
                st = new StringTokenizer(zeile, "#");
                knr = st.nextToken();
                if (nr.equals(knr)) {
                    kname = st.nextToken();
                    found = true;
                    break;
                }
            }
            br.close();
        }
        catch (IOException e) {
            System.out.println(e);
        }

        if (found)
            return knr + " " + kname;
        else
            return "Kundennummer nicht gefunden";
    }
}
```

Die Kundendatei hat die folgende Satzstruktur:

Kundennummer#Kundenname

Zum Testen des Programms muss eine freie Portnummer gewählt werden. Nummern kleiner als 1024 sind für Server des Systems reserviert und haben eine festgelegte Bedeutung. Um Client und Server zusammen auf einem Rechner (ohne Netzverbindung) zu testen, muss beim Start des Client als Rechnername `localhost` angegeben werden.

Aufruf des Server-Programms:
`java Server 4711`

Der Server meldet:
`Server lauscht auf Port 4711`

Aufruf des Client-Programms (in einem zweiten DOS-Fenster):

```
java Client localhost 4711
```

Der Server meldet:

```
Mit Client 1 verbunden
```

Der Client meldet:

```
> Bitte Auftrag an Server eingeben:
```

Eingabe beim Client:

```
1010
```

Der Client meldet:

```
> Antwort vom Server:
1010 Werner Schmitz
> Bitte Auftrag an Server eingeben:
```

`Strg+Z` beendet den Client.

Der Server meldet:

```
Client 1 beendet
```

`Strg+C` beendet den Server.

Statt des Programms `Client` (Programm 10.3) kann auch das Internet-Programm *telnet* benutzt werden:

```
telnet localhost 4711
```

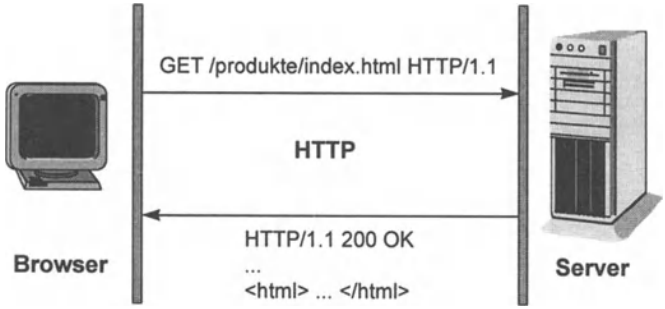
10.3 Datenbankabfragen über HTTP

Web-Browser kommunizieren mit einem Web-Server im Internet über das Protokoll *HTTP* (HyperText Transfer Protocol).

Der Web-Browser ermittelt aus der *URL-Adresse*, z.B. `http://www.xyz.de/produkte/index.html`, den Web-Server (im Beispiel: `www.xyz.de`) und stellt eine TCP-Verbindung zum Server her. Über diese Verbindung sendet er dann eine Anfrage, z.B. die Aufforderung, eine HTML-Seite zu übermitteln (im Beispiel: `GET /produkte/index.html HTTP/1.1`).

Der Server beantwortet die Anfrage mit der Übertragung der verlangten HTML-Seite, die der Browser nun für die Darstellung am Bildschirm aufbereitet.

Bild 10.3: Abfrage einer HTML-Seite über HTTP



statisch

Viele HTML-Seiten sind *statisch*, d.h. sie sind in Dateien gespeichert, auf die der Web-Server Zugriff hat. Diese Seiten enthalten für alle Benutzer dieselben Informationen (bis zum nächsten Update).

dynamisch

Eine *dynamische* Seite wird erst dann erzeugt, wenn der Browser diese Seite anfordert.

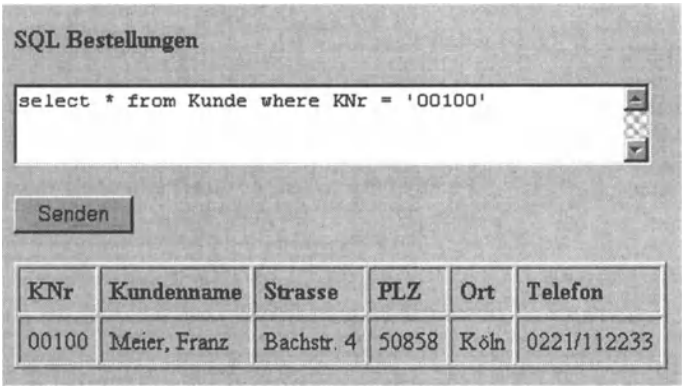
Die Seitenbeschreibungssprache *HTML* bietet die Möglichkeit, Formulare zu definieren, mit denen z.B. interaktiv Daten aus einer Datenbank abgefragt werden können. Die Eingabedaten des Formulars werden vom Browser in spezieller Weise codiert und zum Server geschickt. Bei einer Datenbankanbindung greift der Server zum Zeitpunkt der Benutzeranfrage auf die Datenbank zu und generiert aus dem Abfrageergebnis einen in HTML codierten Ausgabestrom.

Anwendungsbeispiel

Programm 10.7 ist ein spezieller Web-Server, der mit einer MS-Access-Datenbank verbunden ist, beliebige SQL-Anweisungen für diese Datenbank über HTTP empfängt und die Ausführungsergebnisse (Tabellen, Status- oder Fehlermeldungen) zum Browser zurückschickt. Dabei muss der Server den *HTTP-Request* interpretieren und die Antwort HTTP-gemäß aufbereiten (*HTTP-Response*). Die clientseitige

Eingabe der SQL-Anweisung erfolgt in einem HTML-Formular. Alle HTML-Seiten werden dynamisch erzeugt.

Bild 10.4: DB-Abfrage



Zunächst muss der hier relevante Teil des HTTP-Protokolls verstanden werden.

Analyse der HTTP-Anfrage

Programm 10.5 zeigt, was genau der Browser zum Server schickt, wenn Formulardaten eingegeben und gesendet werden.

Zum Testen wird eine statische HTML-Seite benutzt, die vom Browser lokal aufgerufen wird.

Testformular

Die HTML-Seite `Formular.html` hat folgenden Aufbau:

```
<html>
<head>
<title>Datenbankabfrage</title>
</head>
<body bgcolor="Lightgrey">
<b>SQL</b><br>
<form action="http://localhost:4711" method="get">
<textarea cols="50" rows="3" name="sql"></textarea><p>
<input type="submit" value=" Senden ">
</form>
</body>
</html>
```

Das Attribut `action` gibt den URL des Programms an, das die Eingabedaten verarbeiten soll.

`method` bestimmt die Methode, mit der das Formular verarbeitet werden soll. Bei der `get`-Methode werden die Eingabedaten direkt an den mit `action` spezifizierten URL angehängt. ("`post`" wird in Aufgabe 7 behandelt.)

Das Formular enthält ein mehrzeiliges Textfeld (`textarea`) und einen Submit-Button, der URL und Eingabedaten zum Server sendet.

Programm 10.5

```
// Datei: TestServer1.java
import java.io.*;
import java.net.*;

public class TestServer1 {
   public static void main(String[] args) {
      ServerSocket server = null;
      int port = Integer.valueOf(args[0]).intValue();

      try {
         server = new ServerSocket(port);
      }
      catch (IOException e) {
         System.out.println(e);
         System.exit(1);
      }

      while (true) {
         try {
            Socket client = server.accept();
            BufferedReader ein =
               new BufferedReader(new InputStreamReader(
                  client.getInputStream()));
            String zeile;
            while ((zeile = ein.readLine()) != null)
               System.out.println(zeile);
            client.close();
         }
         catch (IOException e) {
            System.out.println(e);
         }
```

```
        }
    }
}
```

Testablauf:

1. Aufruf des Servers: `java TestServer1 4711`
2. Öffnen des Formulars `Formular.html` mit dem Browser
3. Eingabe der SQL-Anweisung:

   `select * from Kunde where KNr = '00100'`

Der Server gibt folgende Daten aus:

HTTP-Request

```
GET /?sql=select+*+from+Kunde+where+KNr+%3D+%2700100%27
HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, */*
Accept-Language: de
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)
Host: localhost:4711
Connection: Keep-Alive
```

Die erste Zeile enthält die Anforderung. Die Daten aus einem Formular mit mehreren Eingabefeldern werden allgemein wie folgt codiert:

```
Name1=Wert1&Name2=Wert2&...
```

Alle Zeichen, die keine ASCII-Zeichen sind, und einige als Sonderzeichen verwendete Zeichen werden durch `%` gefolgt von ihrem Hexadezimalcode dargestellt. Leerzeichen werden als + codiert.

Diese Zeichenkette (*Query String*) muss nun der Server interpretieren.

Programm 10.6 extrahiert den Query String aus der Anforderungszeile und decodiert ihn mit Hilfe einer Methode der Klasse `URLDecoder` aus `java.net`:

```
static String decode(String s)
  throws Exception
```

Programm 10.6

```
// Datei: TestServer2.java
...

public class TestServer2 {
   public static void main(String[] args) {
      ServerSocket server = null;
      PrintWriter log = null;
      int port = Integer.valueOf(args[0]).intValue();

      try {
         server = new ServerSocket(port);
         log = new PrintWriter(new FileWriter(args[1]),
            true);
      }
      catch (IOException e) {
         System.out.println(e);
         System.exit(1);
      }

      while (true) {
         try {
            Socket client = server.accept();
            BufferedReader ein = new BufferedReader(
               new InputStreamReader(
                  client.getInputStream()));
            String zeile = ein.readLine();
            if (zeile == null)
               break;
            int x1 = zeile.indexOf('=');
            int x2 = zeile.indexOf(' ', x1);
            String query = zeile.substring(x1 + 1, x2);
            String decodedQuery =
               URLDecoder.decode(query);
            System.out.println(decodedQuery);
            log.println(decodedQuery);
            client.close();
         }
         catch (Exception e) {
            System.out.println(e);
         }
      }
   }
}
```

Die Ausgabe wird in eine Datei gespeichert. Nach der Decodierung des Query Strings stimmt er mit der Eingabe im Formular überein.

Datenbankabfrage mit Formular

Programm 10.7 implementiert das oben skizzierte Anwendungsbeispiel.

Nach Eingabe der Adresse

```
http://localhost:4711/
```

schickt der Server das dynamisch erzeugte "leere" Eingabeformular. Anstelle von `localhost` kann auch die IP-Adresse des Rechners stehen, falls er ans Netz (Internet oder Intranet) angeschlossen ist.

Die Datenbank "Bestellungen" enthält die Tabellen *Kunde*, *Artikel*, *Bestellkopf* und *Bestellposition*.

Weitere JDBC-Methoden

Das Programm benutzt weitere Klassen und Methoden des JDBC API.

`Statement`-Methoden:

`boolean execute(String sql)`
führt eine SQL-Anweisung aus. Der Rückgabewert kennzeichnet das Ausführungsergebnis.
Bei `true` kann das Ergebnis mit `getResultSet`, bei `false` mit `getUpdateCount` ermittelt werden.

`ResultSet getResultSet()`
liefert die Ergebnismenge der Abfrage.

`int getUpdateCount()`
liefert das Ergebnis der SQL-Anweisung als Update-Zähler.

`ResultSet`-Methode:

`ResultSetMetaData getMetaData()`
liefert die Anzahl der Spalten und Informationen über den Inhalt.

`ResultSetMetaData`-Methode:

`int getColumnCount()`
liefert die Anzahl der Spalten.

`String getColumnName(int col)`
liefert den Namen der Spalte `col`.

`int getColumnType()`
liefert den Typ der Spalte als Konstantenwert der Klasse `java.sql.Types`, z.B. `CHAR`, `VARCHAR`, `TINYINT`, `SMALLINT`, `INTEGER`, `REAL`, `DOUBLE`, `BIT`, `DATE`.

Diese Methode wird im Beispielprogramm benutzt, um die Ausrichtung der Daten in der angezeigten Tabelle (linksbündig oder rechtsbündig) festzulegen.

Der Server muss die Ergebnisdaten der SQL-Anweisung als HTML-Seite aufbereiten. Die Rückantwort an den Browser (HTTP-Response) setzt sich aus dem *HTTP-Header* und dem *HTTP-Body* zusammen. Header und Body sind durch eine Leerzeile getrennt. Der Header beginnt mit dem Statuscode (im Beispiel: `200 OK`) gefolgt von eine Information zum Typ der im Body enthaltenen Daten (im Beispiel: `Content-Type: text/html`). Weitere Kopfzeilen (z.B. die Länge des HTML-Dokuments: `Content-Length`) können folgen.

Leer-Formular

```
<html><head><title>Datenbankabfrage</title></head>
<body bgcolor="Lightgrey">
<b>SQL Bestellungen</b>
<form method="get">
<textarea cols="50" rows="3" name="sql"></textarea><p>
<input type="submit" value=" Senden ">
</form></body></html>
```

Hier fehlt das Attribut `action`. Der Browser setzt automatisch die Adresse des Servers ein, der das Formular geschickt hat.

Ergebnisseite

Die SQL-Anweisung
`select * from Kunde where KNr = '00100'`
führt zu folgendem Ergebnis:

```
<html><head><title>Datenbankabfrage</title></head>
<body bgcolor="Lightgrey">
<b>SQL Bestellungen</b>
<form method="get">
<textarea cols="50" rows="3" name="sql">select * from Kunde
where KNr = '00100'</textarea><p>
<input type="submit" value=" Senden ">
</form>
<table border cellpadding="5"><tr>
<th align="left">KNr</th>
<th align="left">Kundenname</th>
<th align="left">Strasse</th>
<th align="left">PLZ</th>
<th align="left">Ort</th>
<th align="left">Telefon</th>
</tr>
<tr>
<td valign="top" align="left">00100</td>
<td valign="top" align="left">Meier, Franz</td>
<td valign="top" align="left">Bachstr. 4</td>
<td valign="top" align="left">50858</td>
<td valign="top" align="left">Köln</td>
<td valign="top" align="left">0221/112233</td>
</tr>
</table></body></html>
```

Programm 10.7

```
// Datei: HttpServer.java
import java.io.*;
import java.net.*;
import java.sql.*;

public class HttpServer extends Thread {
   private String dbName;
   private int port;
   private Connection dbCon;
   private ServerSocket sock;
   private boolean stop = false;

   public static void main(String[] args) {
      if (args.length != 2) {
         System.out.println(
            "Aufruf: [start] java HttpServer " +
            "<ODBC-DBName> <Port>");
```

```
            System.exit(1);
        }

        HttpServer server = new HttpServer(
            args[0], Integer.valueOf(args[1]).intValue());

        System.out.println("Zum Beenden 'q' eingeben");
        try {
            while (System.in.read() != 'q') ;
        }
        catch (Exception e) { }

        server.close();
    }

    public HttpServer(String dbName, int port) {
        this.dbName = dbName;
        this.port = port;

        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            dbCon = DriverManager.getConnection("jdbc:odbc:"
                + dbName);
        }
        catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }

        try {
            sock = new ServerSocket(port);
        }
        catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }

        start();
        System.out.println("HttpServer gestartet");
        System.out.println("ODBC-DBName: " + dbName);
        System.out.println("Server lauscht auf Port " +
            port);
    }
```

```
    public void close() {
        stop = true;
        try {
            if (sock != null)
                sock.close();
            if (dbCon != null)
                dbCon.close();
        }
        catch (Exception e) { }
    }

    public void run() {
        while (!stop) {
            try {
                new HttpConnection(sock.accept(), dbName,
                    dbCon);
            }
            catch (Exception e) {
                System.out.println(e);
            }
        }
    }
}

// Datei: HttpConnection.java
import java.io.*;
import java.net.*;
import java.sql.*;

public class HttpConnection extends Thread {
    private Socket client;
    private String dbName;
    private Connection dbCon;
    private BufferedReader ein;
    private PrintWriter aus;

    public HttpConnection(Socket client, String dbName,
        Connection dbCon) {
        this.client = client;
        this.dbName = dbName;
        this.dbCon = dbCon;
        start();
    }
```

```
public void run() {
   try {
      ein = new BufferedReader(new InputStreamReader(
            client.getInputStream()));
      aus = new PrintWriter(client.getOutputStream(),
               true);

      String sql = readRequest();
      if (sql.length() == 0)
         sendForm(null);
      else {
         sendForm(sql);
         query(sql);
      }
      client.close();
   }
   catch (Exception e) {
      System.out.println(e);
   }
}

private String readRequest() throws Exception {
   String zeile = ein.readLine();
   if (zeile == null)
      return "";
   int x1 = zeile.indexOf('=');
   if (x1 < 0)
      return "";
   int x2 = zeile.indexOf(' ', x1);
   String query = zeile.substring(x1 + 1, x2);
   return URLDecoder.decode(query);
}

private void sendForm(String s) throws Exception {
   aus.println("HTTP/1.1 200 OK\nContent-Type: " +
      "text/html\n");
   aus.println("<html><head><title>Datenbankabfrage" +
      "</title></head>");
   aus.println("<body bgcolor=\"Lightgrey\">");
   aus.println("<b>SQL " + dbName + "</b>");
   aus.println("<form method=\"get\">");
   aus.print("<textarea cols=\"50\" rows=\"3\" " +
      name=\"sql\">");
```

```
      if (s != null)
         aus.print(s);
      aus.println("</textarea><p>");
      aus.println("<input type=\"submit\" " +
         "value=\" Senden \">");
      if (s == null)
         aus.println("</form></body></html>");
      else
         aus.println("</form>");
   }

   private void query(String sql) throws Exception {
      try {
         Statement s = dbCon.createStatement();

         if (!s.execute(sql)) {
            aus.println(s.getUpdateCount() + " Zeile(n)");
            aus.println("</body></html>");
            return;
         }

         ResultSet rs = s.getResultSet();
         ResultSetMetaData rm = rs.getMetaData();
         int n = rm.getColumnCount();
         String[] align = new String[n];

         aus.println("<table border cellpadding=\"5\"> " +
            "<tr>");
         for (int i = 1; i <= n; i++) {
            if (rm.getColumnType(i) == Types.TINYINT ||
                rm.getColumnType(i) == Types.SMALLINT ||
                rm.getColumnType(i) == Types.INTEGER ||
                rm.getColumnType(i) == Types.REAL ||
                rm.getColumnType(i) == Types.DOUBLE)
               align[i-1] = "right";
            else
               align[i-1] = "left";

            aus.println("<th align=\"" + align[i-1] +
               "\">" + rm.getColumnName(i) + "</th>");
         }
         aus.println("</tr>");
```

```
            while (rs.next()) {
               aus.println("<tr>");
               for (int i = 1; i <= n; i++) {
                  aus.println("<td valign=\"top\" align=\"" +
                     align[i-1] + "\">" + rs.getString(i) +
                     "</td>");
               }
               aus.println("</tr>");
            }
            aus.println("</table></body></html>");
         }
         catch (SQLException e) {
            aus.println(e);
         }
      }
}
```

Die Verbindung zur Datenbank wird beim Start des Servers hergestellt und erst beim Herunterfahren des Servers abgebaut. Damit die Datenbank am Ende ordnungsgemäß geschlossen werden kann, wird die Hauptschleife zur Annahme von Anfragen explizit durch Benutzereingabe beendet (und nicht durch Programmabbruch mittels `Strg+C`).

Der Server wird wie folgt aufgerufen:

```
java HttpServer Bestellungen 4711
```

Selbstverständlich können auch Änderungen der Datenbanktabellen vorgenommen werden, z.B.

```
update Bestellposition set Menge = Menge + 1
where BNr = 1 and BPos = 1
```

10.4 Aufgaben

1. Erstellen Sie ein Programm, das alle Datensätze einer Datenbanktabelle am Bildschirm anzeigt. Die Tabelle soll Angaben zu Büchern in den Feldern *Autor* und *Titel* enthalten. Primärschlüssel ist eine laufende *Nummer*. Die Liste soll nach Autor und Titel sortiert sein.

2. Realisieren Sie einen Server, der Eingaben eines Client empfängt und diese sofort unverändert wieder zurückgibt (*Echo-Server*). Der Server soll den Zeitpunkt der Aufnahme bzw. des Abbaus der Verbindung sowie jeder einzelnen Anforderung in einer Datei protokollieren.

3. Erstellen Sie eine *Client-Server-Anwendung*, bei der zu einer Kundennummer der zugehörige Kundenname aus einer Datenbank geliefert wird.

   Der Client präsentiert ein Formular mit den beiden Textfeldern *Kundennummer* und *Kundenname*. Mit dem Auslösen der Return-Taste im Feld Kundennummer wird diese an den Server geschickt. Die Antwort des Servers (der Kundenname oder die Meldung, dass die Kundennummer nicht vorhanden ist) wird im Feld Kundenname dargestellt.

   Der Server ermittelt mit Hilfe der vom Client gesendeten Kundennummer den Kundennamen und sendet diesen oder die Meldung, dass die Kundennummer nicht vorhanden ist, an den Client. Die Kundentabelle besteht aus den Feldern Kundennummer und Kundenname.

4. Erstellen Sie ein Programm, das beliebige Dateien über das Protokoll *HTTP* auf den lokalen Rechner herunterladen kann. Dabei soll in einem Fenster der Dateiname im URL-Format angegeben und die lokale Zieldatei zur Speicherung in einem Dialogfenster ausgewählt werden können.

5. Die Datenbankabfragen eines Benutzers sollen gezählt werden. Der aktuelle Zählerstand soll jeweils angezeigt werden. Ergänzen Sie das Programm 10.7.

Lösungsidee: HTTP ist ein zustandsloses Protokoll. Der Server will sich nicht alle benutzerspezifischen Zählerstände merken. Der aktuelle Zählerstand wird nun in das HTML-Formular in einem "versteckten" Eingabefeld gespeichert und dann beim nächsten Senden wieder zum Server geschickt, der den Zähler extrahiert, erhöht und zurückschickt. Das HTML-Formular dient also als "Gedächtnis".

Verstecktes Eingabefeld im Formular:

```
<input type="hidden" name="count" value="nnn">
```

Erste Zeile des HTTP-Request:

```
GET /?sql=xxx&count=nnn HTTP/1.1
```

6. Entwickeln Sie einen Server, der in einem HTML-Formular eingetragene Mail-Adressen entgegennimmt und in einer Datenbank speichert. Das Formular soll die Felder *Name* und *Mail-Adresse* enthalten. Diese Felder können auch genutzt werden, um über den Namen nach Mail-Adressen zu suchen. Über *Radiobuttons* soll die gewünschte Aktion (*Suchen* oder *Einfügen*) eingestellt werden können.

7. Entwickeln Sie eine Variante zu Aufgabe 6, bei der statt der `get`- die `post`-Methode im Formular verwendet wird. Die gesendeten Daten werden dann nicht an den URL angehängt und sind also auch nicht im Adressfeld des Browsers zu sehen.

   Der HTTP-Request hat den folgenden Aufbau:

```
POST ... HTTP/1.1
...
Content-Length: nnn
...
<Hier steht eine Leerzeile>
Name1=Wert1&Name2=Wert2&...
```

   Die Zahl *nnn* gibt die Länge der nach der Leerzeile folgenden Daten in Byte an.

Literatur- und Quellenverzeichnis

1. Beispielprogramme und Lösungen

`http://www08.mg.fh-niederrhein.de/dozenten/abts/buch.htm`

oder

`http://www.vieweg.de` (Rubrik: Links/Ressourcen)

2. Java-Entwicklungsumgebung

Java 2 SDK, Standard Edition v 1.3 mit Plug-in und API-Dokumentation
`http://java.sun.com/j2se/`

3. Literatur

Im Rahmen dieses Buches können einige Themen im Umfeld von Java – wie z.B. objektorientierte Modellierung, Kommunikationsprotokolle, Datenbanken – nicht behandelt werden. Die folgenden Quellen sind für eine Einführung bzw. Vertiefung geeignet. Nach den bibliographischen Angaben sind in Klammern stichwortartig die in der jeweiligen Quelle schwerpunktmäßig behandelten Themen aufgeführt.

Abts, D.; Mülder, W.: Aufbaukurs Wirtschaftsinformatik. Vieweg 2000 (*Internet, Datenbanken, SQL, objektorientierte Softwareentwicklung*)

Bengel, G.: Verteilte Systeme. Vieweg 2000 (*Client-Server-Computing*)

Comer, D.; Droms, R.: Computernetzwerke und Internets. Markt und Technik 2000 (*Internet, TCP/IP*)

Dehnhardt, W.: Anwendungsprogrammierung mit JDBC. Hanser 1999 (*JDBC, Client-Server-Computing*)

Erlenkötter, H.: HTML. rororo 2000 (*HTML*)

`ftp://ftp.denic.de/pub/rfc/rfc2616.txt`, 1999 (*HTTP/1.1*)

Hald, A.; Nevermann, W.: Datenbank-Engineering für Wirtschaftsinformatiker. Vieweg 2000 (*Datenbanken, SQL*)

Kuhlmann, G.: Müllmerstadt, F.: SQL. rororo 1999 (*SQL*)

Saake, G.; Sattler, K.-U.: Datenbanken & Java. JDBC, SQLJ und ODMG. dpunkt 2000 (*JDBC*)

Seemann, J.; Wolff von Gudenberg, J.: Software-Entwurf mit UML. Springer 2000 (*objektorientierte Softwareentwicklung*)

Steyer, R.: HTML 4. Data Becker 1999 (*HTML*)

Tolksdorf, R.: Die Sprachen des Web, HTML und XHTML. dpunkt 2000 (*HTML*)

Sachwortverzeichnis

A

abgeleitete Klasse 46
abstract 50, 54
Abstract Window Toolkit 4, 179
AbstractButton 199
AbstractTableModel 250
abstrakte Klasse 50
abstrakte Methode 50
Accessibility API 5
ActionEvent 200, 202, 216, 223, 235
ActionListener 200, 216
Adapterklasse 194
AdjustmentEvent 207
AdjustmentListener 207
Animation 281
anonyme Klasse 61
Anweisung 25
API 4
Applet 1, 2, 263
AppletContext 271
Applet-Tag 267
appletviewer 263, 264, 266, 270
Applikation 2, 45, 87
Archiv 65, 267, 276
arithmetischer Operator 17
Array 83
Arrays 88
Attribut 35
AudioClip 279
Ausdruck 17
Ausgabestrom 129
Ausnahme 67
AWT 4, 179

B

Basisklasse 46
Bedingungsoperator 24
Bestätigungsdialog 244
Bezeichner 12
Bitoperator 21
Block 25
boolean 14
Boolean 93
Border 198
BorderFactory 199
BorderLayout 230
Box 232
BoxLayout 232
break 31
BufferedInputStream 131
BufferedOutputStream 131
BufferedReader 133, 140
BufferedWriter 133, 141
ButtonGroup 202
byte 15
Byte 93
ByteArrayInputStream 131
ByteArrayOutputStream 131
Bytecode 1
Byte-Strom 130

C

call by value 39
Cast-Operator 24
catch 71
catch or throw 69
ChangeEvent 213
ChangeListener 213
char 14
Character 93
Character-Strom 133
CharArrayReader 133
CharArrayWriter 133
Class 117
ClassNotFoundException 151
CLASSPATH 63
Client 303
Client-Server-Anwendung 303
Color 187
Component 180
Connection 288
Container 180
content pane 185, 263

continue 31
CORBA 5
Cursor 183

D

DataInputStream 131, 149
DataOutputStream 131, 149
Date 121
Dateizeiger 152
Datenbank 285
Datenstrom 129
Datentyp 13
DecimalFormat 99
Dekrementoperator 18
Dimension 182
do 29
double 15
Double 93
Double-Buffering 281
Down-Cast 50
dynamisches Binden 53

E

Eingabedialog 245
Eingabestrom 129
Enumeration 103
EOFException 150, 151
Ereignis 192
Ereignisempfänger 192
Ereignisquelle 192
err 115
Error 68
Escape-Sequenz 14
Event 192
Event-Modell 191
Exception 4, 67, 68
extends 46, 56

F

false 14
File 136
file 274, 303
FileFilter 247
FileInputStream 131, 148
FilenameFilter 138
FileNotFoundException 141, 148
FileOutputStream 131, 149
Filepointer 152
FileReader 134, 141
FileWriter 134, 141
FilterInputStream 131
FilterOutputStream 131
FilterReader 134, 145
FilterWriter 134, 145
final 40, 55
finalize 44
finally 71
Fließkommatyp 15
float 15
Float 93
FlowLayout 228
Font 187
for 30

G

ganzzahliger Typ 15
Garbage Collection 4
Garbage Collector 43, 115
Grafikkontext 186
Graphical User Interface 179
Graphics 186
GregorianCalendar 122
GridLayout 231
GUI 179

H

Hashtabelle 109, 110
Hashtable 109
Hashwert 110
Hintergrundfarbe 189
Hotkey 236
HTML 220, 263, 264, 311
HTML-Formular 312
HTTP 274, 310
HTTP-Request 311
HTTP-Response 311, 317
Hüllklasse 92
Hybridanwendung 275
HyperlinkEvent 220
HyperlinkListener 220

I

Icon 198
if 26
Image 277
ImageIcon 198

implements 56
import 63
in 115
Initialisierung 13, 45, 84
Inkrementoperator 18
innere Klasse 59
InputEvent 256
InputStream 131, 132
InputStreamReader 134, 140
Insets 183
instanceof 50, 57
Instanz 35
Instanzklasse 59
Instanzmethode 45
Instanzvariable 44
int 15
Integer 93
Interface 56
Internet Explorer 264, 268
InterruptedException 161, 170
IOException 131, 134, 153
ItemEvent 202, 223, 236
ItemListener 202

J

J2EE 6
J2ME 6
J2SE 6
JApplet 263
jar 276
Java 2 Enterprise Edition 6
Java 2 Micro Edition 6
Java 2 Standard Edition 6
Java 2D API 4
Java 3D API 6
Java Advanced Imaging API 6
Java Bean 5
Java Database Connectivity 6
Java Development Kit 1
Java Foundation Classes 4, 179
Java IDL 5
Java Native Interface 5
Java Virtual Machine 1
Java-Compiler 1
javadoc 11
Java-Just-In-Time-Compiler 2
JButton 198
JCheckBox 202
JCheckBoxMenuItem 236
JComboBox 222
JDBC 285
JDBC API 6, 285
JDBC-ODBC-Bridge 285, 287
JDBC-Treiber 285
JDialog 240
JDK 1
JEditorPane 220
JFC 4, 179
JFileChooser 247
JFrame 182
JIT 2
JLabel 204
JList 225
JMenu 235
JMenuBar 234
JMenuItem 235
JNI 5
JOptionPane 244
JPanel 185
JPasswordField 216
JRadioButton 202
JRadioButtonMenuItem 237
JScrollBar 206
JScrollPane 209
JTabbedPane 212
JTable 250
JTextArea 218
JTextComponent 215
JTextField 214
JToolBar 237
JViewport 210
JVM 1

K

KeyEvent 197, 236
KeyListener 192, 197
KeyStroke 236
Klasse 35
Klassendeskriptor 117
Klassenliteral 118
Klassenmethode 45
Klassenvariable 44
Kommandozeilen-Parameter 87
Kommentar 11
Konstante 55
Konstruktor 42, 48

kontrollierte Ausnahme 68
Kontrollstruktur 25
Koordinatensystem 183

L

Label 216
Layout-Manager 227
LineNumberReader 133, 143
Liste 101
Listener 192
ListSelectionEvent 251
ListSelectionListener 251
Literal 14
localhost 309
logischer Operator 20
logischer Typ 14
lokale Variable 40
long 15
Long 93

M

main 45, 87
markierte Anweisung 31
Math 97
mathematische Funktion 97
Mausaktion 255
MediaTracker 278
mehrdimensionales Array 85
Menü 234
Methode 35, 39
Mitteilungsdialog 245
modales Fenster 240
Modifizierer 54
Modulo-Operator 18
MouseEvent 256
MouseListener 255
MouseMotionListener 256
Multithreading 4, 159

N

Namenskonvention 12
native 56
Netzwerkprogrammierung 301
new 37, 84
notify 170
notifyAll 170
null 38
Null-Layout 228

O

Object 49
ObjectInputStream 131, 151
ObjectOutputStream 131, 151
Object-Tag 268
Objekt 35, 37
ODBC 285, 287
Operator 17
out 115
OutputStream 131, 132
OutputStreamWriter 134

P

package 62
paintComponent 186
Paket 54, 62
Panel 185
Parameter 39
PipedInputStream 131
PipedOutputStream 131
PipedReader 134
PipedWriter 134
Plug-in 264, 268
Point 182
Polymorphismus 53
Port 304
Portnummer 274, 304, 309
primitiver Datentyp 13
PrintStream 131
PrintWriter 134, 143
private 55
Producer/Consumer 169
Properties 113
Property-Liste 113
protected 55
public 37, 55
Punktnotation 38
PushbackInputStream 131
PushbackReader 134, 147

R

Random 99
Random-Access-Datei 152
RandomAccessFile 152
Reader 133, 134
Referenztyp 37, 57
Referenzvariable 37

relationaler Operator 19
Remote Method Invocation 5
repaint 186, 281
ResultSet 289, 316
ResultSetMetaData 317
return 39
RGB-Farbmodell 187
RMI 5
Rückgabetyp 39
Rückgabewert 39
Runnable 159, 160, 162
RunTimeException 68

S

Sandbox 3
Schleife 28
Schlüsselwort 12
SDK 6, 263
Security API 6
SequenceInputStream 131
Serialisierung 151
Serializable 151
Server 303
ServerSocket 306
Servlet 5
short 15
Short 93
Sicherheitsmodell 3
Signatur 41
signiertes Applet 3
Smartcard Edition 6
Socket 303, 304
Software Development Kit 6
Sprunganweisung 31
SQL 6, 285, 288
SQLException 287
SQLJ 6
Stack 104
Standarddatenstrom 115, 129
Standardkonstruktor 42
Statement 288, 316
static 44, 45, 54
statische Initialisierung 45
Stream 129
String 16
String 77
StringBuffer 82
StringReader 134
StringTokenizer 90
StringWriter 134
Subklasse 46
super 47
Superklasse 46
Swing 4, 179
switch 27
Symbolleiste 237
Synchronisation 166, 170
synchronized 54, 167, 173
System 115
System-Properties 116

T

TableModel 250
TableModelEvent 251
TableModelListener 251
TCP/IP 301
telnet 310
this 41
Thread 4, 159, 160
throw 69
Throwable 67
throws 69
Time Slicing 159, 161
TimeZone 121
Toolkit 277
Tooltip 198
transient 54, 151
true 14
try 70
Typumwandlung 17, 24

Ü

Überladen 41
Überschreiben 46

U

Unicode 4, 12
Uniform Resource Locator 274, 301
Up-Cast 49
URL 274, 301
URLDecoder 314
UTF-Format 149, 150

V

Variable 13
Variablendeklaration 13

Vector 101
Vererbung 46
Verhalten 35
Verzweigung 25
void 39
Vordergrundfarbe 189

W

wait 170
while 28
WindowEvent 197
WindowListener 192, 197
Wrapper-Klasse 92
Writer 133, 135

Z

Zeichenkette 16
Zeichentyp 14
Zufallszahl 98, 99
Zustand 35
Zuweisungsoperator 23

Weitere Titel aus dem Programm

Dietmar Abts
Grundkurs JAVA
Die Einführung in das objektorientierte Programmieren
mit Beispielen und Übungsaufgaben
1999. X, 345 S. mit 30 Abb. Br. DM 39,80 ISBN 3-528-05711-4

Konzeption von Java - Die Sprache Java - Objektorientierte Programmierung - Ein- und Ausgabe - Multithreading - Erzeugung grafischer Benutzungsoberflächen - Entwicklung von Applets - Verschiedene Anwendungsbeispiele

Günther Bauer
Bausteinbasierte Software
Eine Einführung in moderne Konzepte des Software-Engineering
1999. VIII, 192 S. mit 128 Abb. Br. DM 59,80 ISBN 3-528-05722-X

Bausteine - Bausteinsysteme - Ein-Platz-Anwendungen - Mehr-Platz-Anwendungen - Multi-Media- und Intra- sowie Internet-Anwendungen

Anton Hald, Wolf Nevermann
Datenbanksysteme
Eine Einführung in die Entwicklung moderner Datenbank-Anwendungen
2., erw. Aufl. 2000. ca. 250 S. Br. ca. DM 64,00 ISBN 3-528-15436-5

Einführung in Aufbau und Aufgaben von Datenbanksystemen - Theorie und Praxis der Datenmodellierung - Konzeptionelles und logisches Datenbankdesign - physisches Datenbankdesign - Konzeption und Entwicklung von Datenbankapplikationen - verteilte Datenbanken - Client-Server-Architekturen - objektorientierte Datenbanken

vieweg

Abraham-Lincoln-Straße 46
65189 Wiesbaden
Fax 0611.7878-400
www.vieweg.de

Stand 1.4.2000
Änderungen vorbehalten.
Erhältlich im Buchhandel oder im Verlag.

Weitere Titel aus dem Programm

Dietmar Abts, Wilhelm Mülder

Aufbaukurs Wirtschaftsinformatik

Der kompakte und praxisorientierte Weg zum Diplom

2000. XII, 380 S. mit 120 Abb. Br. DM 59,00 ISBN 3-528-05592-8

Architekturen betrieblicher Informationssysteme - Workflow- und Dokumentationsmanagement - Managementunterstützungssysteme - Internet - Elektronische Märkte - Telekooperation in der mobilen Informationsgesellschaft - Datenmodellierung und Datenbanksysteme - Objektorientierte Software-Entwicklung

Doug Cooper, Michael Clancy

Pascal

Lehrbuch für das strukturiertes Programmieren

5. Aufl. 1999. X, 509 S. Br. DM 88,00 ISBN 3-528-44316-2

Vertraut werden mit Programmen - Prozeduren und Funktionen zum Lösen von Problemen - Wie man die Bearbeitung steuert: Die For-Anweisung - Auswahl treffen: Die Case-Anweisung - If-Anweisung - Bedingte Schleifen - Arrays - Reccords - Files und Text - Set-Typ - Pointer - Sortieren, Suchen, Vergleichen - Weitere Sprachelemente

Ingrid Gerdes, Frank Klawonn, Rudolf Kruse, Markus Schröder

Evolutionäre Algorithmen

Ihre Kopplung mit Fuzzy-Systemen und Neuronalen Netzen

2000. ca. 370 S. mit 40 Abb. Br. ca. DM 59,00 ISBN 3-528-05570-7

Optimierung - Genetische Algorithmen - Evolutionsstrategien - Genetische Programmierung - Fuzzy Systeme - Neuronale Netze

vieweg

Abraham-Lincoln-Straße 46
65189 Wiesbaden
Fax 0611.7878-400
www.vieweg.de

Stand 1.4.2000
Änderungen vorbehalten.
Erhältlich im Buchhandel oder im Verlag.