



Go To Java 2

<http://kickme.to/tiger/>

Inhaltsverzeichnis

- [Vorwort](#)
- [Änderungen und Versionen](#)
- [Inhaltsverzeichnis](#)
- [Verzeichnis der Abbildungen](#)
- [Verzeichnis der Tabellen](#)
- [Verzeichnis der Listings](#)
- [Die Icons in diesem Buch](#)
- [1 Was ist Java?](#)
 - [1.1 Historie](#)
 - [1.2 Eigenschaften von Java](#)
 - [1.3 Bewertung](#)
 - [1.4 Zusammenfassung](#)
- [2 Ein einleitendes Beispiel](#)
 - [2.1 Übersicht](#)
 - [2.2 Schritt 1: Die Programmstruktur](#)
 - [2.3 Schritt 2: Initialisierung des Applets](#)
 - [2.4 Schritt 3: Vorbereitung der Spielsteine](#)
 - [2.5 Schritt 4: Grafikausgabe](#)
 - [2.6 Schritt 5: Mausereignisse](#)
 - [2.7 Zusammenfassung](#)
- [3 Wie geht es weiter?](#)
 - [3.1 Wie sollte man dieses Buch lesen?](#)
 - [3.2 Was ist der Inhalt der einzelnen Kapitel?](#)
 - [3.3 Wie erhält man Hilfe?](#)
 - [3.4 Installation des JDK](#)
 - [3.5 Schnelleinstieg](#)
 - [3.6 Zusammenfassung](#)
- [4 Datentypen](#)
 - [4.1 Lexikalische Elemente eines Java-Programms](#)
 - [4.2 Primitive Datentypen](#)
 - [4.3 Variablen](#)
 - [4.4 Arrays](#)
 - [4.5 Referenztypen](#)
 - [4.6 Typkonvertierungen](#)
 - [4.7 Zusammenfassung](#)
- [5 Ausdrücke](#)
 - [5.1 Eigenschaften von Ausdrücken](#)
 - [5.2 Arithmetische Operatoren](#)
 - [5.3 Relationale Operatoren](#)
 - [5.4 Logische Operatoren](#)
 - [5.5 Bitweise Operatoren](#)
 - [5.6 Zuweisungsoperatoren](#)
 - [5.7 Sonstige Operatoren](#)

- [5.8 Operator-Vorrangregeln](#)
- [5.9 Zusammenfassung](#)
- **[6 Anweisungen](#)**
 - [6.1 Elementare Anweisungen](#)
 - [6.2 Verzweigungen](#)
 - [6.3 Schleifen](#)
 - [6.4 Zusammenfassung](#)
- **[7 Objektorientierte Programmierung](#)**
 - [7.1 Klassen und Objekte](#)
 - [7.2 Methoden](#)
 - [7.3 Vererbung](#)
 - [7.4 Attribute von Klassen, Methoden und Variablen](#)
 - [7.5 Klassen mit static-Elementen](#)
 - [7.6 Abstrakte Klassen und Methoden](#)
 - [7.7 Interfaces](#)
 - [7.8 Spezielle Klassen](#)
 - [7.9 Zusammenfassung](#)
- **[8 Bestandteile eines Programms](#)**
 - [8.1 Programmelemente](#)
 - [8.2 Pakete](#)
 - [8.3 Der Entwicklungszyklus](#)
 - [8.4 Zusammenfassung](#)
- **[9 Exceptions](#)**
 - [9.1 Grundlagen und Begriffe](#)
 - [9.2 Behandlung von Exceptions](#)
 - [9.3 Weitergabe von Exceptions](#)
 - [9.4 Zusammenfassung](#)
- **[10 Multithreading](#)**
 - [10.1 Grundlagen und Begriffe](#)
 - [10.2 Die Klasse Thread](#)
 - [10.3 Das Interface Runnable](#)
 - [10.4 Synchronisation](#)
 - [10.5 Verwalten von Threads](#)
 - [10.6 Zusammenfassung](#)
- **[11 String und StringBuffer](#)**
 - [11.1 Grundlegende Eigenschaften](#)
 - [11.2 Methoden der Klasse String](#)
 - [11.3 Weitere Eigenschaften](#)
 - [11.4 Die Klasse StringBuffer](#)
 - [11.5 Zusammenfassung](#)
- **[12 Utilities](#)**
 - [12.1 Die Klasse Vector](#)
 - [12.2 Die Klasse Stack](#)
 - [12.3 Die Klasse Hashtable](#)
 - [12.4 Die Klasse BitSet](#)
 - [12.5 Die Klasse StringTokenizer](#)
 - [12.6 Die Klasse Random](#)
 - [12.7 Die Klassen Date, Calendar und GregorianCalendar](#)

- [12.8 Die Klasse System](#)
- [12.9 Die Klasse Arrays](#)
- [12.10 Zusammenfassung](#)
- **[13 Dateiein-/ausgabe](#)**
 - [13.1 Allgemeine Konzepte](#)
 - [13.2 Ausgabe-Streams](#)
 - [13.3 Eingabe-Streams](#)
 - [13.4 Random-Access-Dateien](#)
 - [13.5 Datei- und Verzeichnis-Handling](#)
 - [13.6 Zusammenfassung](#)
- **[14 Grafikausgabe](#)**
 - [14.1 Das Abstract Windowing Toolkit](#)
 - [14.2 Grundlagen der Grafikausgabe](#)
 - [14.3 Elementare Grafikroutinen](#)
 - [14.4 Weiterführende Funktionen](#)
 - [14.5 Zusammenfassung](#)
- **[15 Textausgabe](#)**
 - [15.1 Ausgabefunktionen](#)
 - [15.2 Unterschiedliche Schriftarten](#)
 - [15.3 Eigenschaften von Schriftarten](#)
 - [15.4 Drucken in Java](#)
 - [15.5 Zusammenfassung](#)
- **[16 Farben](#)**
 - [16.1 Das Java-Farbmodell](#)
 - [16.2 Erzeugen von Farben](#)
 - [16.3 Verwenden von Farben](#)
 - [16.4 Systemfarben](#)
 - [16.5 Zusammenfassung](#)
- **[17 Fenster](#)**
 - [17.1 Die verschiedenen Fensterklassen](#)
 - [17.2 Aufrufen und Schließen eines Fensters](#)
 - [17.3 Visuelle Eigenschaften](#)
 - [17.4 Anzeigezustand](#)
 - [17.5 Fensterelemente](#)
 - [17.6 Zusammenfassung](#)
- **[18 Event-Handling](#)**
 - [18.1 Das Event-Handling im JDK 1.1](#)
 - [18.2 Entwurfsmuster für den Nachrichtenverkehr](#)
 - [18.3 Zusammenfassung](#)
- **[19 Low-Level-Events](#)**
 - [19.1 Window-Events](#)
 - [19.2 Component-Events](#)
 - [19.3 Mouse-Events](#)
 - [19.4 MouseMotion-Events](#)
 - [19.5 Focus-Events](#)
 - [19.6 Key-Events](#)
 - [19.7 Zusammenfassung](#)
- **[20 Menüs](#)**

- [20.1 Grundlagen](#)
- [20.2 Menüleiste](#)
- [20.3 Menüs](#)
- [20.4 Menüeinträge](#)
- [20.5 Action-Events](#)
- [20.6 Kontextmenüs](#)
- [20.7 Datenaustausch mit der Zwischenablage](#)
- [20.8 Zusammenfassung](#)
- **[21 Dialoge](#)**
 - [21.1 Erstellen eines Dialogs](#)
 - [21.2 Die Layoutmanager](#)
 - [21.3 Modale Dialoge](#)
 - [21.4 Zusammenfassung](#)
- **[22 Vordefinierte Dialogelemente](#)**
 - [22.1 Rahmenprogramm](#)
 - [22.2 Label](#)
 - [22.3 Button](#)
 - [22.4 Checkbox](#)
 - [22.5 CheckboxGroup](#)
 - [22.6 TextField](#)
 - [22.7 TextArea](#)
 - [22.8 Choice](#)
 - [22.9 List](#)
 - [22.10 Scrollbar](#)
 - [22.11 ScrollPane](#)
 - [22.12 Zusammenfassung](#)
- **[23 Eigene Dialogelemente](#)**
 - [23.1 Die Klasse Canvas](#)
 - [23.2 Entwicklung einer 7-Segment-Anzeige](#)
 - [23.3 Einbinden der Komponente](#)
 - [23.4 Zusammenfassung](#)
- **[24 Bitmaps und Animation](#)**
 - [24.1 Bitmaps](#)
 - [24.2 Animation](#)
 - [24.3 Zusammenfassung](#)
- **[25 Applets](#)**
 - [25.1 Die Architektur eines Applets](#)
 - [25.2 Einbinden eines Applets](#)
 - [25.3 Die Ausgabe von Sound](#)
 - [25.4 Verweise auf andere Seiten](#)
 - [25.5 Animation in Applets](#)
 - [25.6 Zusammenfassung](#)
- **[26 Die Werkzeuge des JDK](#)**
 - [26.1 Entwicklungswerkzeuge](#)
 - [26.2 Werkzeuge zur Dokumentation und Archivierung](#)
 - [26.3 Zusammenfassung](#)
- **[27 Collections](#)**

- [27.1 Grundlagen und Konzepte](#)
- [27.2 Die Collection des Typs List](#)
- [27.3 Iteratoren](#)
- [27.4 Eine eigene Queue-Klasse](#)
- [27.5 Die Collection des Typs Set](#)
- [27.6 Die Collection des Typs Map](#)
- [27.7 Sortierte Collections](#)
- [27.8 Die Klasse Collections](#)
- [27.9 Zusammenfassung](#)
- **[28 Serialisierung](#)**
 - [28.1 Grundlagen](#)
 - [28.2 Weitere Aspekte der Serialisierung](#)
 - [28.3 Anwendungen](#)
 - [28.4 Zusammenfassung](#)
- **[29 Performance-Tuning](#)**
 - [29.1 Einleitung](#)
 - [29.2 Tuning-Tips](#)
 - [29.3 Einsatz eines Profilers](#)
 - [29.4 Zusammenfassung](#)
- **[30 Datenbankzugriffe mit JDBC](#)**
 - [30.1 Einleitung](#)
 - [30.2 Grundlagen von JDBC](#)
 - [30.3 Die DirDB-Beispieldatenbank](#)
 - [30.4 Weiterführende Themen](#)
 - [30.5 Zusammenfassung](#)
- **[31 Reflection](#)**
 - [31.1 Einleitung](#)
 - [31.2 Die Klassen Object und Class](#)
 - [31.3 Methoden- und Konstruktorenaufrufe](#)
 - [31.4 Zugriff auf Membervariablen](#)
 - [31.5 Zusammenfassung](#)
- **[32 Netzwerkprogrammierung](#)**
 - [32.1 Grundlagen der Netzwerkprogrammierung](#)
 - [32.2 Client-Sockets](#)
 - [32.3 Server-Sockets](#)
 - [32.4 Daten mit Hilfe der Klasse URL lesen](#)
 - [32.5 Zusammenfassung](#)

Vorwort

Das vorliegende Buch »Go To Java 2« ist der Nachfolger von »Java 1.1 lernen«, das vor gut einem Jahr bei Addison-Wesley erschienen ist. Es verläßt damit die Reihe der Einsteigerbücher und gesellt sich zu den umfangreicheren Titeln der GoTo-Reihe. Die vielen Leser der Vorversion können allerdings beruhigt sein. Trotz der formalen Änderung ist die bewährte Struktur des Buches erhalten geblieben, und wesentliche Teile seines Inhalts konnten übernommen werden.

Natürlich wurden viele Korrekturen vorgenommen, und das Buch wurde an die Änderungen der Version 1.2 des JDK angepaßt (das von SUN seit einiger Zeit als *Java 2 platform* bezeichnet wird). Zudem gibt es sechs neue Kapitel, die sich mit fortgeschrittenen Themen befassen. Hier werden Collections, Tuning-Maßnahmen, Serialisierung, Reflection, JDBC-Datenbankprogrammierung und Netzwerkprogrammierung behandelt. Im großen und ganzen ist aber der Aufbau des Buches gleich geblieben und es eignet sich nach wie vor gleichermaßen als Lehr- und Nachschlagewerk.

Das Buch besteht aus 32 Kapiteln, die alle grundlegenden Aspekte von Java und seiner umfangreichen Klassenbibliothek erläutern. Die ersten drei Kapitel sind besonders wichtig. Sie geben einen Überblick über die Geschichte von Java, erläutern wichtige Spracheigenschaften und beschreiben den Aufbau des Buches. Sie erklären zudem die Installation des Java Development Kit, geben Hinweise auf weiterführende Dokumentationen und zeigen auf, wie die Beispielprogramme übersetzt und gestartet werden können. Hier finden Sie auch Hinweise auf den Inhalt der übrigen Kapitel und Anregungen zum Umgang mit dem Buch.

Ebenso wie in der Vorgängerversion wurde auch zu diesem Buch eine Online-Version erstellt, die auf der beigefügten CD-ROM enthalten ist. Sie stellt das komplette Buch in HTML-Form dar und ist wegen der großen Anzahl an Navigationshilfen und Querverweisen (es sind über 5000) gut als Referenz verwendbar. Daneben enthält die CD-ROM alle Beispiele aus dem Buch, das Java Development Kit 1.2 und weitere nützliche Werkzeuge und Hilfsmittel.

Was in »Java 1.1 lernen« als Experiment mit ungewissem Ausgang begonnen wurde, wird hier fortgesetzt: Auch die Online-Version von »Go To Java 2« steht wieder zum freien Download zur Verfügung! Studenten und Leser mit schmalen Budget haben so die Möglichkeit, Java zu lernen, ohne das Buch kaufen zu müssen oder können es vor dem Kauf erst einmal eingehend studieren. Auch Universitäten und vergleichbare Einrichtungen werden wieder die Möglichkeit zur Installation einer gespiegelten Version erhalten, um das Buch auf einfache Weise in der studentischen Ausbildung nutzen zu können.

Um weitere Informationen zum Buch und zur Online-Version zu erhalten, können Sie mich unter der Adresse <http://www.gkrueger.com> im Internet erreichen. Schreiben Sie mir eine Mail, wenn Sie Anregungen oder Kritik haben oder falls Sie einen Fehler gefunden haben. Sie dürfen mir natürlich auch schreiben, wenn Ihnen das Buch einfach nur gut gefallen hat! Verständnisfragen zu einzelnen Aspekten der Java-Programmierung kann ich aus Zeitgründen allerdings meist nicht beantworten. Sie sind besser in einer der zahlreichen Java-Newsgruppen aufgehoben (siehe [Abschnitt 3.3.2](#)).

Eine der größten Herausforderungen beim Schreiben des Buchs bestand in der Art und Weise seiner Herstellung. Anders als seine Vorgängerversionen wurde es nicht mit einer der üblichen Textverarbeitungen, sondern mit Hilfe eines gewöhnlichen ASCII-Editors in der Dokumentenbeschreibungssprache *SGML (Structured Generalized Markup Language)* geschrieben. Alle Werkzeuge zum Erstellen der Online- und Satzversion wurden selbst entwickelt und sind in Java geschrieben. Neben den zusätzlichen Möglichkeiten, die diese Vorgehensweise bietet, erhöht die Möglichkeit der automatisierten Verarbeitung des Quelltextes vor allem die *Konsistenz* der unterschiedlichen Buchbestandteile. Fehlerhafte Verweise, falsche Numerierungen oder ungültige Indexeinträge sollten nun der Vergangenheit angehören.

Auch die jetzt ohne zusätzlichen Aufwand erstellbare Online-Version kann bei Fehlern oder Erweiterungen sehr viel schneller als bisher aktualisiert werden. Wir wollen das nutzen, indem die Online-Version zukünftig unabhängig von den Nachdruckzyklen des Buchs gepflegt wird, wenn Verbesserungen oder Erweiterungen es sinnvoll erscheinen lassen. Alle Änderungen der Online-Version werden versioniert, die Papierform des Buchs entspricht der Version 1.0 der Online-Version.

Ich wünsche allen Lesern, daß ihnen dieses Buch beim Erlernen und Anwenden von Java ein unentbehrlicher Helfer sein wird und daß sie nach seiner Lektüre über solide Grundkenntnisse in der Java-Programmierung verfügen mögen.

Mein Dank gilt allen, die bei der Entstehung mitgewirkt haben. Dazu zählt vor allem Judith Stevens, die das Buch als Lektorin bei Addison-Wesley betreut hat. Auch für die Bereitschaft, mit SGML neue Wege in der Herstellung zu gehen, möchte ich mich bei den Beteiligten im Verlag bedanken. Zudem möchte ich Kollegen und Bekannten danken, die sich der Mühe unterzogen haben, einzelne Kapitel zu lesen und mit ihren Hinweisen und Anregungen zu seiner jetzigen Form beigetragen haben. Hier sind insbesondere Stefan Stark, Andi Müller, Jacques Nietsch, Carsten Leutzing und - für die neuen Kapitel - Thomas Backens zu nennen. Auch den vielen Lesern der Vorversion, die Fehler gefunden oder Anregungen gegeben haben, möchte ich an dieser Stelle danken. Wie immer gilt mein besonderer Dank meiner Familie, durch deren Hilfe das Buch überhaupt erst möglich gemacht wurde.

Guido Krüger im Januar 1999

Änderungen und Versionen

Aktuelle Version: **1.0.5 vom 31.10.1999**

31.10.1999	Version 1.0.5
31.10.1999	Die Methode <code>finalize</code> der Klasse <code>Object</code> wird in Abschnitt 7.2.7 nun korrekt als <code>public</code> deklariert.
31.10.1999	An mehreren Stellen im Buch wird nun bei der Beschreibung der Methodenüberlagerung anstelle des Begriffs "überlagert" der präzisere Terminus "überlagernd" verwendet.
31.10.1999	In Abschnitt 25.4.3 wurde die fehlerhafte Erläuterung der Methode <code>showDocument</code> korrigiert.
31.10.1999	In Tabelle 4.1 wurde die fehlerhafte Exponentialdarstellung des Wertebereichs der Fließkommazahlen korrigiert.
31.10.1999	In den Abschnitten 7.2.6 und 7.3.3 wurde die Beschreibung der <i>default</i> -Konstruktoren präzisiert.
31.10.1999	In Listing 24.14 wurde der Aufruf von <code>size</code> gegen <code>getSize</code> ausgetauscht.
31.10.1999	In Abschnitt 7.8.1 wurde die Eignung der Wrapper-Klassen zur Simulation von <i>call by reference</i> widerrufen, und es wurden weitere Beispiele zur Anwendung der Wrapper-Klassen gegeben.
03.08.1999	Version 1.0.4
01.08.1999	In Abschnitt 6.3.3 wurde der mittlerweile veraltete Vergleich mit C++ bezüglich der Sichtbarkeit und Lebensdauer von im Schleifenkopf deklarierten Variablen korrigiert.
01.08.1999	In Abschnitt 4.2.4 wurde die Beschreibung der Fließkommaliterale <code>NaN</code> , <code>POSITIVE_INFINITY</code> und <code>NEGATIVE_INFINITY</code> korrigiert.
01.08.1999	In Abschnitt 10.4.2 wurde der fehlerhafte Verweis auf Kapitel 7 korrigiert.
10.05.1999	Version 1.0.3
10.05.1999	In Kapitel 7 wird zur Altersberechnung mehrfach das Jahr 1996 verwendet. Es wurde durchgängig durch 1999 ersetzt.
10.05.1999	In Kapitel 7 wurde in Listing 7.30 das Schlüsselwort "Interface" versehentlich groß geschrieben. Korrekt lautet die erste Zeile des Listings "public interface Sammlerstueck".
09.03.1999	Version 1.0.2
09.03.1999	Im Buch ist in Abschnitt 9.2.2 das falsche Listing abgedruckt (im Buch auf Seite 211). Statt <code>Listing0903.java</code> sollte dort <code>RTErrrorProg2.java</code> stehen. Die Online-Version enthielt diesen Fehler nicht.
08.03.1999	In Abschnitt 6.2.1 muß es im zweiten Absatz des Abschnitts "Bedeutung" in der zweiten Zeile ", andernfalls <code>anweisung2</code> " statt ", andernfall <code>ausdruck2</code> " heißen (im Buch auf Seite 142, 2. Zeile v.u.).
08.03.1999	In Abschnitt 5.7.2 muß es im zweiten Absatz "Nicht-String-Operand" statt "Nicht-String-Operator" heißen (im Buch auf Seite 130, 14. Zeile v.u.). Eine Zeile weiter muß es dann "mit dem anderen Operanden" statt "mit dem anderen Operator" heißen.
02.02.1999	Version 1.0.1
02.02.1999	Ein paar kleinere Bugs aus den Klassennamen der Syntaxbeschreibungen entfernt.
31.01.1999	Version 1.0
31.01.1999	Buchversion fertiggestellt.

Verzeichnis der Abbildungen

- [2.1 Das unsortierte Schiebepuzzle](#)
- [2.2 Das sortierte Schiebepuzzle](#)
- [2.3 Ein leeres Applet](#)
- [2.4 Die Rahmendarstellung](#)
- [2.5 Die Darstellung des Drag & Drop](#)
- [4.1 Konvertierungen auf primitiven Datentypen](#)
- [8.1 Der Entwicklungszyklus in Java](#)
- [8.2 Plattformübergreifende Entwicklung in Java](#)
- [11.1 Ein Aufruf von substring\(begin, end\)](#)
- [11.2 Der Aufruf von regionMatches](#)
- [14.1 Das Koordinatensystem von Java](#)
- [14.2 Ein einfaches Fenster](#)
- [14.3 Ausgabe von Linien](#)
- [14.4 Ausgabe von Rechtecken](#)
- [14.5 Ausgabe eines Polygons](#)
- [14.6 Ausgabe von Kreisen](#)
- [14.7 Ausgabe von Kreisbögen](#)
- [14.8 Ausgabe von gefüllten Flächen](#)
- [14.9 Ausgabe des Java-Logos als Liniengrafik](#)
- [14.10 Kopieren von Grafiken](#)
- [14.11 Verwendung der Clipping-Funktionen](#)
- [15.1 Einfache Textausgabe](#)
- [15.2 Die Randelemente eines Fensters](#)
- [15.3 Ausgabe verschiedener Fonts](#)
- [15.4 Liste der Standardschriften](#)
- [15.5 Größenmaßzahlen für Fonts in Java](#)
- [15.6 Anzeige von Font-Metriken](#)
- [15.7 Zentrierte Textausgabe](#)
- [15.8 Das Programm zur Druckausgabe](#)
- [16.1 Der Farbenkreis](#)
- [16.2 Verwendung von Systemfarben](#)
- [17.1 Hierarchie der Fensterklassen](#)
- [17.2 Ein einfacher Bildschirmschoner](#)
- [17.3 Das Beispiel-Icon](#)
- [17.4 Ein Programm mit veränderten Fensterelementen](#)
- [18.1 Die Hierarchie der Ereignisklassen](#)
- [18.2 Die Hierarchie der ActionListener-Interfaces](#)
- [18.3 Das Programm für den Nachrichtentransfer](#)
- [19.1 Das Fenster sieht sich selbst aus der Vogelperspektive](#)
- [19.2 Die Ausgabe des Mausklick-Programms](#)
- [19.3 Die Ausgabe des Mausbewegungsprogramms](#)
- [19.4 Programm nach Erhalt des Eingabefokus](#)
- [19.5 Darstellung von Tastaturereignissen](#)

- [20.1 Erzeugen von Menüs](#)
- [20.2 Geschachtelte Menüs](#)
- [20.3 Ein Programm, das auf Action-Events reagiert](#)
- [20.4 Aufruf eines Kontextmenüs](#)
- [21.1 Ein Dialog mit zwei Buttons](#)
- [21.2 Verwendung der Klasse FlowLayout](#)
- [21.3 Verwendung der Klasse GridLayout](#)
- [21.4 Das FlowLayout in einem größeren Fenster](#)
- [21.5 Das GridLayout in einem größeren Fenster](#)
- [21.6 Verwendung der Klasse BorderLayout](#)
- [21.7 Ein BorderLayout mit Lücken](#)
- [21.8 Beispiel für GridBagLayout](#)
- [21.9 Zellschema für GridBagLayout-Beispiel](#)
- [21.10 Das GridBagLayout-Beispiel nach dem Skalieren](#)
- [21.11 Verwendung des Null-Layouts](#)
- [21.12 Verwendung eines geschachtelten Layouts](#)
- [21.13 Ein weiteres Beispiel für geschachtelte Layouts](#)
- [21.14 Das Vaterfenster für den modalen Dialog](#)
- [21.15 Ein einfacher Ja-/Nein-Dialog](#)
- [21.16 Ein Aufruf von OKDialog](#)
- [21.17 Ein Aufruf von YesNoDialog](#)
- [21.18 Ein Aufruf von YesNoCancelDialog](#)
- [22.1 Das Beispielprogramm zum Aufruf der Beispieldialoge](#)
- [22.2 Der noch leere Beispieldialog](#)
- [22.3 Ein Dialog mit Label-Komponenten](#)
- [22.4 Ein Dialog mit Checkbox-Komponenten](#)
- [22.5 Ein Dialog mit CheckboxGroup-Elementen](#)
- [22.6 Ein Dialog mit beschrifteten Textfeldern](#)
- [22.7 Ein Dialog mit einem TextArea-Objekt](#)
- [22.8 Ein Dialog mit einer Choice-Komponente](#)
- [22.9 Ein Dialog mit einer Listbox](#)
- [22.10 Ein Dialog mit zwei Schiebereglern](#)
- [22.11 ViewPort und virtueller Ausgabebereich beim ScrollPane](#)
- [22.12 Verwendung von ScrollPane](#)
- [23.1 Der Aufbau der 7-Segment-Anzeige](#)
- [23.2 Ein Beispiel für die Anwendung der 7-Segment-Anzeige](#)
- [24.1 Laden und Anzeigen einer Bitmap](#)
- [24.2 Verwendung von BitmapComponent](#)
- [24.3 Ein animierter Zähler](#)
- [24.4 Die Ausgabe während des Ladevorgangs](#)
- [24.5 Animation eines Schriftzugs, Schnappschuß 1](#)
- [24.6 Animation eines Schriftzugs, Schnappschuß 2](#)
- [24.7 Animation eines Schriftzugs, Schnappschuß 3](#)
- [24.8 Die animierte Schlange, Schnappschuß 1](#)
- [24.9 Die animierte Schlange, Schnappschuß 2](#)
- [24.10 Die animierte Schlange, Schnappschuß 3](#)
- [24.11 Die Lauflicht-Animation](#)
- [24.12 Eine Animation mit Doppelpufferung](#)

- [25.1 Ableitungsbaum der Applet-Klasse](#)
- [25.2 Darstellung des Schranken-Applets im Internet Explorer](#)
- [25.3 Das sprechende »Hello, World«-Programm](#)
- [25.4 Darstellung von URLLaden im Internet Explorer](#)
- [25.5 Das Wolkenkratzer-Beispielprogramm](#)
- [28.1 Das Programm serialver](#)
- [28.2 Eltern-Kind-Graph für Serialisierungsbeispiel](#)
- [30.1 E/R-Diagramm für DirDB](#)
- [32.1 Das ISO/OSI-7-Schichten-Modell](#)
- [32.2 Das vereinfachte 4-Ebenen-Modell](#)

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

Go To Java 2, Addison Wesley, Version 1.0.5, © 1999 Guido Krüger, <http://www.gkrueger.com>

Verzeichnis der Tabellen

- [3.1 Die comp.lang.java-Hierarchie im Usenet](#)
- [4.1 Primitive Datentypen](#)
- [4.2 Standard-Escape-Sequenzen](#)
- [4.3 Symbolische Fließkommaliterale](#)
- [5.1 Arithmetische Operatoren](#)
- [5.2 Relationale Operatoren](#)
- [5.3 Logische Operatoren](#)
- [5.4 Bitweise Operatoren](#)
- [5.5 Zuweisungsoperatoren](#)
- [5.6 Operator-Vorrangregeln](#)
- [7.1 Die Wrapper-Klassen](#)
- [8.1 Die vordefinierten Pakete des JDK](#)
- [12.1 Feldbezeichner der Klasse GregorianCalendar](#)
- [12.2 Standard-Properties](#)
- [13.1 Aus Writer abgeleitete Klassen](#)
- [13.2 Aus Reader abgeleitete Klassen](#)
- [15.1 Style-Parameter](#)
- [16.1 Gebräuchliche Farbwerte](#)
- [16.2 Liste der vordefinierten Systemfarben](#)
- [17.1 Konstanten zur Cursorauswahl](#)
- [18.1 Focus-Ereignisse](#)
- [18.2 Methoden für Focus-Ereignisse](#)
- [18.3 Key-Ereignisse](#)
- [18.4 Methoden für Key-Ereignisse](#)
- [18.5 Mouse-Ereignisse](#)
- [18.6 Methoden für Mouse-Ereignisse](#)
- [18.7 MouseMotion-Ereignisse](#)
- [18.8 Methoden für MouseMotion-Ereignisse](#)
- [18.9 Komponenten-Ereignisse](#)
- [18.10 Methoden für Komponenten-Ereignisse](#)
- [18.11 Container-Ereignisse](#)
- [18.12 Methoden für Container-Ereignisse](#)
- [18.13 Window-Ereignisse](#)
- [18.14 Methoden für Window-Ereignisse](#)
- [18.15 Action-Ereignisse](#)
- [18.16 Methode für Action-Ereignisse](#)
- [18.17 Adjustment-Ereignisse](#)
- [18.18 Methode für Adjustment-Ereignisse](#)
- [18.19 Item-Ereignisse](#)
- [18.20 Methode für Item-Ereignisse](#)
- [18.21 Text-Ereignisse](#)
- [18.22 Methode für Text-Ereignisse](#)
- [19.1 Methoden von WindowListener](#)

- [19.2 Methoden von ComponentListener](#)
- [19.3 Methoden von MouseListener](#)
- [19.4 Virtuelle Key-Codes](#)
- [19.5 Rückgabecodes bei Tastaturereignissen](#)
- [22.1 Konstanten für Schieberegler-Ereignisse](#)
- [22.2 Konstanten zur Anzeige der Schieberegler in ScrollPane](#)
- [25.1 Optionale Parameter des APPLETTags](#)
- [26.1 Optionen von javac](#)
- [26.2 Optionen des Java-Interpreters](#)
- [26.3 Optionen von appletviewer](#)
- [26.4 Kommandos von jdb](#)
- [26.5 Markierungen in Dokumentationskommentaren](#)
- [26.6 Einige Optionen von javadoc](#)
- [26.7 Kommandos von jar](#)
- [26.8 Optionen von javap](#)
- [29.1 Geschwindigkeit von Methodenaufrufen](#)
- [30.1 get-Methoden von ResultSet](#)
- [30.2 Die Struktur der dir-Tabelle](#)
- [30.3 Die Struktur der file-Tabelle](#)
- [30.4 SQL-Datentypen](#)
- [30.5 SQL-Aggregatfunktionen](#)
- [31.1 Klassenobjekte für die primitiven Typen](#)
- [32.1 Klassen von IP-Adressen](#)
- [32.2 Standard-Port-Nummern](#)
- [32.3 Liste wichtiger RFCs](#)

Verzeichnis der Listings

- [2.1 Grundstruktur des Schiebepuzzles](#)
- [2.2 HTML-Datei zum Laden des Schiebepuzzle-Applets](#)
- [2.3 Die init-Methode des Schiebepuzzle-Applets](#)
- [2.4 Variablendeklarationen für das Schiebepuzzle](#)
- [2.5 Vorbereitung der Spielsteine im Schiebepuzzle](#)
- [2.6 Mischen der Spielsteine im Schiebepuzzle](#)
- [2.7 Die Methode paint im Schiebepuzzle](#)
- [2.8 Zeichnen des Rahmens im Schiebepuzzle](#)
- [2.9 Zeichnen der Spielsteine des Schiebepuzzles](#)
- [2.10 Die überlagerte update-Methode im Schiebepuzzle](#)
- [2.11 Die Methode mousePressed im Schiebepuzzle](#)
- [2.12 Lokale Methoden im Schiebepuzzle](#)
- [2.13 Die Methode mouseReleased des Schiebepuzzles](#)
- [2.14 Test benachbarter Spielsteine im Schiebepuzzle](#)
- [2.15 Die Methode mouseDragged im Schiebepuzzle](#)
- [3.1 Hello, world](#)
- [3.2 Einrücken von Klassen und Methoden](#)
- [3.3 Einrücken von Kontrollanweisungen](#)
- [3.4 Einrücken fortgesetzter Anweisungen](#)
- [3.5 Einrücken langer Methodenaufrufe](#)
- [3.6 Einfache Ausgaben](#)
- [3.7 Einfache Eingaben](#)
- [4.1 Verwendung eines Dokumentationskommentars im Java-API](#)
- [4.2 Einfache Variablen ausgeben](#)
- [4.3 Initialisieren von Variablen](#)
- [4.4 Deklaration von Arrays](#)
- [4.5 Erzeugen von Arrays](#)
- [4.6 Deklaration und Initialisierung von Arrays](#)
- [4.7 Initialisierung mit literalen Arrays](#)
- [4.8 Deklaration und Zugriff auf Arrays](#)
- [4.9 Zugriff auf mehrdimensionale Arrays](#)
- [4.10 Ein nicht-rechteckiges Array](#)
- [4.11 Erzeugen eines Objekts mit dem new-Operator](#)
- [5.1 Fehler beim Kompilieren durch unvollständige Initialisierung](#)
- [5.2 Fehler beim Kompilieren durch unvollständige Datenflußanalyse](#)
- [5.3 String-Verkettung](#)
- [5.4 Vorsicht bei der String-Verkettung!](#)
- [5.5 Korrekte String-Verkettung bei gemischten Ausdrücken](#)
- [5.6 Vergleichen von Referenzen](#)
- [5.7 Vergleichen von Strings mit equals](#)
- [5.8 Bindungsprobleme bei den bitweisen Operatoren](#)
- [5.9 Korrekte Klammerung von bitweisen Operatoren](#)
- [6.1 Verdecken von Klassen- oder Instanzvariablen](#)

- [6.2 Dangling else](#)
- [6.3 Dangling else, ausgeschaltet](#)
- [6.4 Bedingtes Kompilieren](#)
- [6.5 Duff's Device](#)
- [6.6 Das gelabelte break](#)
- [6.7 Das gelabelte continue](#)
- [7.1 Eine einfache Klassendefinition](#)
- [7.2 Erzeugen eines Objekts mit new](#)
- [7.3 Kombinierte Deklaration und Initialisierung einer Objektvariablen](#)
- [7.4 Zuweisen von Werten an die Variablen eines Objekts](#)
- [7.5 Lesender Zugriff auf die Variablen eines Objekts](#)
- [7.6 Eine einfache Methode zur Altersberechnung](#)
- [7.7 Aufruf einer Methode](#)
- [7.8 Verwendung von this](#)
- [7.9 Eine Methode zur Ausgabe des Alters](#)
- [7.10 Wiederholter Aufruf der Methode zur Ausgabe des Alters](#)
- [7.11 Überladen einer Methode](#)
- [7.12 Definition eines parametrisierten Konstruktors](#)
- [7.13 Aufruf eines parametrisierten Konstruktors](#)
- [7.14 Eine Klasse mit mehreren Konstruktoren](#)
- [7.15 Verkettung von Konstruktoren](#)
- [7.16 Die finalize-Methode](#)
- [7.17 Ein einfaches Beispiel für Vererbung](#)
- [7.18 Zugriff auf geerbte Membervariablen](#)
- [7.19 Ableitung einer abgeleiteten Klasse](#)
- [7.20 Zugriff auf mehrfach vererbte Membervariablen](#)
- [7.21 Überlagern einer Methode in einer abgeleiteten Klasse](#)
- [7.22 Realisierung eines Instanzenzählers mit Klassenvariablen](#)
- [7.23 Verwendung von Klassenvariablen zur Definition von Konstanten](#)
- [7.24 Verwendung von Math.sqrt](#)
- [7.25 Definition eines statischen Konstruktors](#)
- [7.26 Abstrakte Klassen und Methoden](#)
- [7.27 Einfügen einer neuen Mitarbeiterklasse in die Gehaltsberechnung](#)
- [7.28 Definition eines Interfaces](#)
- [7.29 Implementieren eines Interfaces](#)
- [7.30 Definition eines weiteren Interfaces](#)
- [7.31 Implementieren mehrerer Interfaces](#)
- [8.1 Zugriff auf verdeckte Membervariablen](#)
- [8.2 Die Klasse A des Pakets demo](#)
- [8.3 Die Klasse B des Pakets demo](#)
- [8.4 Die Klasse C des Pakets demo.tools](#)
- [8.5 Verwendung der Klassen aus selbstdefinierten Paketen](#)
- [9.1 Die try-catch-Anweisung](#)
- [9.2 Ein Programm mit einem Laufzeitfehler](#)
- [9.3 Abfangen des Laufzeitfehlers mit einer try-catch-Anweisung](#)
- [9.4 Verwendung des Fehlerobjekts nach einem Laufzeitfehler](#)
- [9.5 Fortfahren nach Laufzeitfehlern](#)
- [9.6 Mehr als eine catch-Klausel](#)

- [9.7 Verwendung der finally-Klausel](#)
- [9.8 Verwendung der throws-Klausel](#)
- [9.9 Auslösen einer Ausnahme](#)
- [10.1 Ein einfacher Thread mit einem Zähler](#)
- [10.2 Beenden des Threads durch Aufruf von stop](#)
- [10.3 Anwendung der Methoden interrupt und isInterrupted](#)
- [10.4 Implementieren von Runnable](#)
- [10.5 Eine Klasse zur Primfaktorzerlegung](#)
- [10.6 Verwendung der Klasse zur Primfaktorzerlegung](#)
- [10.7 Primfaktorzerlegung mit Threads](#)
- [10.8 Verwendung der Klasse zur Primfaktorzerlegung mit Threads](#)
- [10.9 Zwei Zählerthreads](#)
- [10.10 Synchronisation von Threads mit Klassenobjekten](#)
- [10.11 Eine unzureichend synchronisierte Zählerklasse](#)
- [10.12 Synchronisieren der Zählermethode](#)
- [10.13 Ein Producer-/Consumer-Beispiel mit wait und notify](#)
- [11.1 String-Verkettung und die Methode substring](#)
- [11.2 Die Methode regionMatches der Klasse String](#)
- [11.3 Implementierung der String-Verkettung](#)
- [12.1 Die Methode elements der Klasse Vector](#)
- [12.2 Anwendung eines Stacks](#)
- [12.3 Anwendung der Klasse Hashtable](#)
- [12.4 Konstruktion von Primzahlen mit der Klasse BitSet](#)
- [12.5 Anwendung der Klasse StringTokenizer](#)
- [12.6 Zufallszahlen zur Generierung eines Lottotips](#)
- [12.7 Die Felder der Klasse Calendar](#)
- [12.8 Datumsarithmetik](#)
- [12.9 Ausgeben der System-Properties](#)
- [12.10 Die Auflösung des System-Timers bestimmen](#)
- [12.11 Verwendung von System.arraycopy](#)
- [12.12 Sortieren eines Arrays](#)
- [13.1 Erstellen einer Datei](#)
- [13.2 Gepufferte Ausgabe in eine Datei](#)
- [13.3 Schachteln von Writer-Konstruktoren](#)
- [13.4 Die Klasse PrintWriter](#)
- [13.5 Konstruktion einer eigenen FilterWriter-Klasse](#)
- [13.6 Anwendung der Klasse FileReader](#)
- [13.7 Verwendung der Klasse StringReader](#)
- [13.8 Eingabepufferung beim Lesen aus Dateien](#)
- [13.9 Die Klasse LineNumberReader](#)
- [13.10 Lesen einer .class-Datei mit der Klasse RandomAccessFile](#)
- [13.11 Verwendung der Klasse File](#)
- [13.12 Anlegen einer temporären Datei](#)
- [14.1 Ein einfaches Fenster erzeugen](#)
- [14.2 Rahmenprogramm für nachfolgende Beispiele](#)
- [14.3 Ausgabe von Linien](#)
- [14.4 Ausgabe von Rechtecken](#)
- [14.5 Ausgabe eines Polygons](#)

- [14.6 Ausgabe von Kreisen](#)
- [14.7 Ausgabe von Kreisbögen](#)
- [14.8 Ausgabe von gefüllten Flächen](#)
- [14.9 Kopieren von Flächen mit copyArea](#)
- [14.10 Verwendung der Clipping-Funktionen](#)
- [15.1 Einfache Textausgabe im Grafikfenster](#)
- [15.2 Ausgabe verschiedener Schriften](#)
- [15.3 Auflistung aller Standardschriften](#)
- [15.4 Auflistung aller Standardschriften im JDK 1.2](#)
- [15.5 Vergrößern der Schriftart](#)
- [15.6 Anzeige von Font-Metriken](#)
- [15.7 Zentrierte Textausgabe](#)
- [15.8 Ausdruck einer Testseite](#)
- [16.1 Darstellung des Farbkreises](#)
- [16.2 Verwendung von Systemfarben](#)
- [17.1 Anzeigen und Entfernen eines Frames](#)
- [17.2 Ein einfacher Bildschirmschoner](#)
- [17.3 Anzeigezustand eines Fensters umschalten](#)
- [17.4 Ein Programm mit veränderten Fensterelementen](#)
- [18.1 Basisprogramm für den Nachrichtentransfer](#)
- [18.2 Implementieren eines Listener-Interfaces](#)
- [18.3 Verwendung lokaler Klassen](#)
- [18.4 Verwendung anonymer Klassen](#)
- [18.5 Trennung von GUI- und Anwendungslogik](#)
- [18.6 Überlagern der Komponenten-Event-Handler](#)
- [19.1 Die Klasse CloseableFrame](#)
- [19.2 Das eigene Fenster aus der Vogelperspektive](#)
- [19.3 Reaktion auf Mausklicks](#)
- [19.4 Zeichnen von Rechtecken durch Ziehen der Maus](#)
- [19.5 Behandlung von Fokus-Ereignissen](#)
- [19.6 Reaktion auf Tastaturereignisse](#)
- [20.1 Erzeugen von Menüs](#)
- [20.2 Erzeugen eines Menüeintrags mit Beschleuniger](#)
- [20.3 Menüleisten mit zwei Menüs und Beschleunigertasten](#)
- [20.4 Geschachtelte Menüs](#)
- [20.5 Reaktion auf Action-Events aus einem Menü](#)
- [20.6 Einbinden eines Kontextmenüs](#)
- [20.7 Kommunikation mit der Zwischenablage](#)
- [21.1 Ein Dialog mit zwei Buttons](#)
- [21.2 Die Klasse FlowLayout](#)
- [21.3 Die Klasse GridLayout](#)
- [21.4 Das BorderLayout](#)
- [21.5 Umgang mit GridBagLayout und GridBagConstraints](#)
- [21.6 Beispiel für GridBagLayout](#)
- [21.7 Anordnen von Dialogelementen ohne Layoutmanager](#)
- [21.8 Schachteln von Layoutmanagern](#)
- [21.9 Eine weitere Anwendung für geschachtelte Layoutmanager](#)

- [21.10 Konstruktion modularer Dialoge](#)
- [21.11 Drei modale Standarddialoge](#)
- [22.1 Rahmenprogramm für Dialogelemente](#)
- [22.2 Verwendung von Label-Komponenten](#)
- [22.3 Verwendung von Checkbox-Komponenten](#)
- [22.4 Behandlung von Item-Events](#)
- [22.5 Verwendung einer CheckboxGroup](#)
- [22.6 Verwendung von Textfeldern](#)
- [22.7 Behandlung von Text-Events](#)
- [22.8 Textfelder mit Beschriftung](#)
- [22.9 Behandlung von Text-Events bei der Komponente TextArea](#)
- [22.10 Verwendung einer TextArea-Komponente](#)
- [22.11 Behandlung der Ereignisse einer Choice-Komponente](#)
- [22.12 Verwendung einer Choice-Komponente](#)
- [22.13 Behandlung der Ereignisse einer List-Komponente](#)
- [22.14 Verwendung einer List-Komponente](#)
- [22.15 Verwendung von Scrollbars](#)
- [22.16 Verwendung der Klasse ScrollPane](#)
- [23.1 Eine 7-Segment-Anzeige](#)
- [23.2 Einbinden der 7-Segment-Anzeige](#)
- [24.1 Laden einer Bitmap-Datei](#)
- [24.2 Laden und Anzeigen einer Bitmap](#)
- [24.3 Programm zum Laden und Anzeigen einer Bitmap](#)
- [24.4 Eine Komponente zum Anzeigen einer Bitmap](#)
- [24.5 Verwenden der Bitmap-Komponente](#)
- [24.6 Ein animierter Zähler](#)
- [24.7 Verwendung von Threads zur Animation](#)
- [24.8 Abspielen einer Folge von Bitmaps](#)
- [24.9 Die animierte Schlange](#)
- [24.10 Bildschirmflackern reduzieren bei stehenden Animationen](#)
- [24.11 Standard-Implementierung von update](#)
- [24.12 Modifizierte Version von update](#)
- [24.13 Modifizierte Schlangenanimation](#)
- [24.14 update-Methode mit Doppelpufferung](#)
- [24.15 Animation mit Doppelpufferung](#)
- [25.1 Ein einfaches Applet](#)
- [25.2 Verwendung von getParameterInfo](#)
- [25.3 Die Methode getAppletInfo](#)
- [25.4 Das APPLET-Tag](#)
- [25.5 Ein parametrisiertes Applet](#)
- [25.6 Die HTML-Datei zum Schranken-Applet](#)
- [25.7 Das sprechende »Hello, World«](#)
- [25.8 Soundausgabe aus einer Applikation](#)
- [25.9 Laden von Web-Seiten aus einem Applet](#)
- [25.10 Die HTML-Datei zum Laden der Webseiten](#)
- [25.11 Das Wolkenkratzer-Applet](#)
- [25.12 Die HTML-Datei zum Aufrufen des Wolkenkratzer-Applets](#)
- [26.1 HTML mit ARCHIVE-Tag](#)

- [27.1 Anlegen und Bearbeiten zweier Listen](#)
- [27.2 Zugriff auf eine Collection mit einem Iterator](#)
- [27.3 Das Interface Queue](#)
- [27.4 Die Klasse LinkedList](#)
- [27.5 Anwendung der Queue-Klasse](#)
- [27.6 Generierung eines Lottotips mit HashSet](#)
- [27.7 Anwendung der Klasse HashMap](#)
- [27.8 Die Klasse TreeSet](#)
- [27.9 Rückwärts sortieren mit einem Comparator](#)
- [27.10 Sortieren einer Liste](#)
- [28.1 Eine serialisierbare Uhrzeitklasse](#)
- [28.2 Serialisieren eines Time-Objekts](#)
- [28.3 Serialisieren mehrerer Objekte](#)
- [28.4 Deserialisieren eines Time-Objekts](#)
- [28.5 Deserialisieren mehrerer Elemente](#)
- [28.6 Die Uhrzeitklasse mit serialVersionUID](#)
- [28.7 Die Klasse Person](#)
- [28.8 Serialisieren von Objekten und Referenzen](#)
- [28.9 Ein einfacher Objektspeicher](#)
- [28.10 Beispielanwendung für den einfachen Objektspeicher](#)
- [28.11 Kopieren von Objekten durch Serialisierung](#)
- [29.1 Langsame String-Verkettung](#)
- [29.2 Wie der Java-Compiler String-Verkettungen übersetzt](#)
- [29.3 Performante String-Verkettungen mit StringBuffer.append](#)
- [29.4 Langsames Einfügen in einen String](#)
- [29.5 Schnelles Einfügen in einen String](#)
- [29.6 Vergleich von Listen und Vektoren](#)
- [29.7 Performance von Writer und OutputStream](#)
- [29.8 Gepufferter Zugriff auf Random-Access-Dateien](#)
- [29.9 Test-Programm für den Profiler](#)
- [29.10 Programm zum Sortieren der Profiling-Informationen](#)
- [29.11 Sortieren mit QuickSort](#)
- [29.12 Eine optimierte Ausgabemethode](#)
- [30.1 Behandeln einer SQLException](#)
- [30.2 Das Rahmenprogramm der DirDB-Datenbank](#)
- [30.3 Öffnen und Schließen der DirDB-Datenbank](#)
- [30.4 Anlegen der DirDB-Tabellen](#)
- [30.5 Füllen der DirDB-Tabellen](#)
- [30.6 Anzahl der Sätze in der DirDB-Datenbank](#)
- [30.7 Suchen nach Dateien in der DirDB-Datenbank](#)
- [30.8 Suchen nach Verzeichnissen in der DirDB-Datenbank](#)
- [30.9 Sortieren der Ergebnismenge](#)
- [30.10 Cluster-Berechnung mit der DirDB-Datenbank](#)
- [30.11 Die Klasse CachedConnection](#)
- [30.12 Verwenden eines PreparedStatement](#)
- [31.1 Dynamisches Laden von Klassen](#)
- [31.2 Testcode in der main-Methode](#)
- [31.3 Die Klasse Test](#)

- [31.4 Die Klasse TestQueue](#)
- [31.5 Funktionszeiger mit Reflection nachbilden](#)
- [31.6 Funktionszeiger mit Interfaces](#)
- [31.7 Parametrisierte Konstruktoren mit Reflection aufrufen](#)
- [31.8 Die Klasse PrintableObject](#)
- [32.1 IP-Adressenauflösung](#)
- [32.2 Abfrage des DayTime-Services](#)
- [32.3 Lesender und schreibender Zugriff auf einen Socket](#)
- [32.4 Laden einer Seite von einem Web-Server](#)
- [32.5 Ein ECHO-Server für Port 7](#)
- [32.6 Eine verbesserte Version des Echo-Servers](#)
- [32.7 Ein experimenteller Web-Server](#)
- [32.8 Daten von einem URL lesen](#)

Die Icons in diesem Buch

Um Ihnen die Orientierung in diesem Buch zu erleichtern, haben wir den Text in bestimmte Funktionsabschnitte gegliedert und diese durch entsprechende Symbole oder Icons gekennzeichnet. Folgende Icons finden Verwendung:

Beispiele helfen Ihnen, sich schneller und sicher im Feld der Java-Programmierung zu orientieren. Sie werden darum mit diesem Icon gekennzeichnet.

Manches ist von besonderer Bedeutung und verdient darum auch, besonders hervorgehoben zu werden. Solche **Hinweise** sind sehr nützlich, sie bringen einen geschwinder ans Ziel.

Manches geht ganz leicht. Wenn man nur weiß, wie, **Praxistips** finden Sie in den Abschnitten, wo dieses Icon steht.

Beim Erlernen einer Programmiersprache gibt es immer wieder Fallen, in die man als Ahnungsloser hineintreten kann. Die **Warnungen** im Buch führen Sie sicher an ihnen vorbei.

Zwischen den unterschiedlichen **Java-Versionen** gibt es teilweise beträchtliche Differenzen. Mit diesem Icon markieren wir die wichtigsten Unterschiede zwischen den Versionen 1.1 und 1.2 des JDK.

Kapitel 1

Was ist Java?

- [1 Was ist Java?](#)
 - [1.1 Historie](#)
 - [1.2 Eigenschaften von Java](#)
 - [1.2.1 Sprachmerkmale](#)
 - [1.2.2 Applets: Eine neue Klasse von Programmen](#)
 - [1.2.3 Grafikprogrammierung](#)
 - [1.2.4 Umfangreiche Klassenbibliothek](#)
 - [1.3 Bewertung](#)
 - [1.3.1 Einige weit verbreitete Mißverständnisse ...](#)
 - [These 1: Java == JavaScript](#)
 - [These 2: Java ist einfach zu lernen](#)
 - [These 3: Java ist portabel](#)
 - [These 4: Java-Programme sind langsam](#)
 - [These 5: Java ist nicht für ernsthafte Anwendungen geeignet](#)
 - [1.3.2 Wird Java erfolgreich sein?](#)
 - [1.3.3 Fazit](#)
 - [1.4 Zusammenfassung](#)

1.1 Historie

- [1.1 Historie](#)

Die Entstehungsgeschichte von Java begann im Jahre 1991, als eine Gruppe von Ingenieuren bei Sun Microsystems mit der Entwicklung von Software für interaktives Fernsehen und andere Geräte der Konsumelektronik begann. Bestandteile dieses Projekts, das später den Namen *Green-Projekt* bekam, waren ein Betriebssystem (Green-OS), ein Interpreter (Oak), ein Grafiksubsystem und diverse Hardwarekomponenten. Patrick Naughton, James Gosling und einige andere Mitglieder des Green-Teams entwickelten schließlich ein Gerät mit der Bezeichnung »*7« (*Star Seven*), das sie im Herbst 1992 firmenintern vorstellten.

Diese Vorstellung konnte einige der Sun-Manager - unter ihnen Bill Joy und Sun-Chef Scott McNealy - beeindrucken, und aus dem lockeren Team wurde im November die Firma *First Person, Inc.* Im Jahre 1993 versuchte First Person eine Reihe von Verträgen über die weitere Vermarktung von Star Seven unter Dach und Fach zu bringen, die aber allesamt scheiterten. Nach einigen Rettungsversuchen wurde die Arbeit des kleinen Unternehmens im April praktisch beendet und die Hälfte der Mitarbeiter in andere Projekte versetzt.

Mittlerweile hatte das World Wide Web ein kritische Größe erreicht. Nachdem NCSA Mosaic, der erste grafische Web-Browser, im April 1993 verfügbar war, konnte jedermann grafisch aufbereitete Informationen im Internet ansehen und auf einfachste Weise zwischen unterschiedlichsten Diensten und Anbietern wechseln. Das darin liegende Potential erkannten auch Patrick Naughton und sein Team, und sie fokussierten ihre Zielrichtung auf die Internet-Entwicklung. Im Herbst 1994 wurde die erste Version von *WebRunner* fertiggestellt, einem Browser, der in der Lage war, kleine Java-Programme, *Applets* genannt, aus dem World Wide Web zu laden und innerhalb des Browsers auszuführen.

Zu diesem Zeitpunkt war Oak, das später in Java umbenannt wurde, bereits eine relativ stabile Sprache. Sie wurde nicht nur dazu verwendet, WebRunner zu entwickeln, sondern von Arthur van Hoff, der Ende 1993 zum Team kam, zur Entwicklung des Java-Compilers selbst verwendet. WebRunner konnte die Verantwortlichen bei Sun überzeugen. Das Programm wurde nach der Umbenennung in *HotJava* in den nächsten Monaten stabilisiert und weiterentwickelt und konnte im Mai auf der *SunWorld '95* der Öffentlichkeit vorgestellt werden.

Trotz des technologischen Durchbruchs konnten sich zunächst nur wenige Anwender mit HotJava anfreunden. So war es ein großes Glück, daß Netscape sich entschied, die Java-Technologie von Sun zu lizenzieren und in der Version 2.0 des *Navigators*, die im Dezember 1995 auf den Markt kam, einem breiten Publikum zur Verfügung zu stellen. Nach einigen Monaten des Betatests für Java und HotJava wurde kurz darauf im Januar 1996 das *JDK 1.0*, die erste Version des *Java Development Kit*, freigegeben. Bereits während der Betatestphase wurden hunderte von Applets geschrieben, die über das Internet geladen werden konnten und schon früh einen Eindruck von den Möglichkeiten der Sprache vermittelten.

Kurz vor der Fertigstellung des JDK 1.0 wurde aus den verbliebenen Mitgliedern des Green-Teams die Firma *JavaSoft*, die von Sun mit der Weiterentwicklung von Java betraut wurde. Unter ihrem Präsidenten Alan Baratz entwickelte und pflegte JavaSoft das JDK und seine Werkzeuge und sollte fortan maßgeblich den weiteren Weg von Java bestimmen.

Tatsächlich stand die Entwicklung nun keinesfalls still, sondern nahm an Dynamik noch zu. In den folgenden Monaten bildeten sich eine Reihe von strategischen Allianzen zwischen Sun bzw. JavaSoft und vielen Großen der Branche. So wurde beispielsweise die im Mai 1996 angekündigte Komponentenarchitektur, die den Namen *JavaBeans* bekam, von so prominenten Firmen wie Borland, Lotus, Oracle, IBM, Netscape und Symantec unterstützt.

Im Laufe der nächsten Monate kam der »Java-Hype« so richtig in Fahrt, und Java wurde mit Vorschußlorbeeren überhäuft. In welcher Weise das Interesse an Java anstieg, mögen einige Kennzahlen verdeutlichen:

- Gamelan ist ein Verzeichnis von Java-Ressourcen, das Verweise auf Java-Applets oder -Applikationen verwaltet. Während die Zahl der Einträge im Januar 1996 unter 1000 lag, war sie bis August 1996 auf 3500 gestiegen und liegt heute (im November 1998) bei über 12000.
- Eine ähnliche Entwicklung nahm die Anzahl der Java-Newsgroups im Internet. Während im Januar 1996 lediglich eine einzige Gruppe [comp.lang.java](#) existierte, gibt es heute 17 weitere Newsgroups innerhalb der [comp.lang.java](#)-Hierarchie.
- Auch die Anzahl der Nachrichten hat sich ähnlich entwickelt. Während [comp.lang.java](#) Anfang 1996 etwa 100 Mails pro Tag hatte, sind es heute meist 400-500. Dabei sind noch nicht einmal die Nachrichten eingerechnet, die über diverse lokale News- oder List-Server ausgetauscht werden.
- Ein weiteres Beispiel für die rasante Verbreitung ist die Anzahl der bei SUN offiziell registrierten Java-Usergruppen. Sie lag im Februar 1996 bei 17, stieg bis Juli 1996 auf 36 und liegt heute bei etwa 140.

Nach einer Reihe von Ankündigungen im ersten Halbjahr wurden bis Ende 1996 zahlreiche Neuerungen vorgestellt. Unter ihnen waren die Datenbank-Spezifikation JDBC, die Komponentenarchitektur Beans, das Card API, HotJava Views, die »100 % Pure Java Initiative« und eine Reihe weiterer APIs. Zusätzlich kamen die ersten integrierten Entwicklungssysteme, wie *Cafe* und *Visual Cafe* von Symantec oder *J++* von Microsoft, auf den Markt.

Im Dezember 1996 wurde die Version 1.1 des Java Development Kit angekündigt. Sie sollte eine Reihe von Bugs der Vorgängerversion ausmerzen und weitere Funktionalitäten bringen. Im Februar 1997 standen die ersten Betaversionen des JDK 1.1 zur Verfügung und konnten von interessierten Entwicklern heruntergeladen werden. Im März 1997 wurde HotJava 1.0 herausgegeben (alle vorigen Versionen hatten offiziell Betacharakter) und auch das Java-Betriebssystem *JavaOS 1.0* wurde in diesem Monat der Öffentlichkeit vorgestellt.

Etwa zeitgleich konnte man auf der Cebit 1997 den ersten Prototypen der *JavaStation*, einer diskettenlosen Workstation, die ausschließlich auf Java basiert, bewundern. Mit der Ankündigung von Java-Prozessoren, wie dem *PicoJava*, eröffnete Sun die Perspektive, daß Java-Programme mittelfristig ebenso schnell laufen werden wie kompilierter C- oder C++-Code. Das für Java-Entwickler herausragende Ereignis des Jahres war die *JavaOne* im April 1997, die eine Vielzahl von Ankündigungen, Prototypen und Produkten hervorbrachte.

Die folgenden Monate standen für viele Entwickler und Tool-Hersteller unter dem Zeichen der Umstellung auf die Version 1.1 des JDK. Zwar gab es bereits Ende 1997 mehr als ein Dutzend integrierte Entwicklungsumgebungen, doch der Support für die Version 1.1 war längst noch nicht überall vorhanden. Auch die Browser-Hersteller taten sich schwer und haben erst zum Jahreswechsel 1997/98 mit den 4er-Versionen ihrer Browser erste Implementierungen des JDK 1.1 vorgestellt. Bis diese eine brauchbare Stabilität erreicht hatten, vergingen weitere Monate. Im Internet Explorer von Microsoft fehlen auch heute noch elementare Dinge wie JNI oder RMI.

Während sich 1998 die meisten Entwickler mit der Version 1.1 des JDK beschäftigten, wurde bei SUN bereits an der neuen Version 1.2 gearbeitet. Im Frühjahr 1998 stand deren erste öffentliche Version, das JDK 1.2 Beta 2, der Öffentlichkeit zur Verfügung. Wichtige Neuerungen waren die *Java Foundation Classes* mit dem *Swing Toolset*, dem *Java 2D API* und dem *Drag & Drop API*, das *Collection Framework* und das *Extension Framework*. Daneben gab es viele weitere Verbesserungen bestehender Pakete. Nach zwei weiteren Betas, die bis zum Juli erschienen, brachte SUN im Oktober und November die »Release Candidates« 1 und 2 heraus. Anfang Dezember 1998 wurde dann schließlich die erste finale Version des JDK 1.2 zur Verfügung gestellt und im Januar 1999 in *Java 2 platform* umbenannt.

Trotz der offiziellen Umbenennung in *Java 2 platform* werden wir in diesem Buch meist vom JDK 1.2 sprechen, wenn die aktuelle Version gemeint ist. Das entspricht weitgehend dem derzeitigen Sprachgebrauch und harmoniert mit weiten Teilen der offiziellen Dokumentation zum JDK.

Hinweis

1.2 Eigenschaften von Java

- [1.2 Eigenschaften von Java](#)
 - [1.2.1 Sprachmerkmale](#)
 - [1.2.2 Applets: Eine neue Klasse von Programmen](#)
 - [1.2.3 Grafikprogrammierung](#)
 - [1.2.4 Umfangreiche Klassenbibliothek](#)

1.2.1 Sprachmerkmale

Java wurde vollständig neu entworfen. Die Designer versuchten, die Syntax der Sprachen C und C++ soweit wie möglich nachzuahmen, verzichteten aber auf einen Großteil der komplexen und fehlerträchtigen Merkmale beider Sprachen. Das Ergebnis ihrer Bemühungen haben sie wie folgt zusammengefaßt:

»Java soll eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutrale, portable, performante, nebenläufige, dynamische Programmiersprache sein.«

Der Erfolg von Java hängt eng damit zusammen, daß ein wesentlicher Teil dieser Forderungen tatsächlich in einer für viele Programmierer akzeptablen Weise umgesetzt wurde - obgleich wir am Ende dieses Kapitels auch einige kritische Anmerkungen dazu geben werden.

Java ist sowohl eine objektorientierte Programmiersprache in der Tradition von Smalltalk als auch eine klassische imperative Programmiersprache nach dem Vorbild von C. Im Detail unterscheidet sich Java aber recht deutlich von C++, das denselben Anspruch erhebt. Durch die Integration einer großen Anzahl anspruchsvoller Features wie Multithreading, strukturiertem Exceptionhandling oder eingebauten grafischen Fähigkeiten implementiert Java eine Reihe interessanter Neuerungen auf dem Gebiet der Programmiersprachen.

Zudem profitiert Java davon, daß viele der Features von C++ *nicht* realisiert wurden und die Sprache dadurch schlank und übersichtlich wurde. So gibt es beispielsweise keine expliziten Pointer, keine separaten Header-Dateien, keine Mehrfachvererbung und keine Templates in Java. Wer allerdings glaubt, Java sei lediglich das Skelett einer Programmiersprache, die nur das Nötigste bietet, irrt. Tatsächlich ist Java eine elegante Sprache, die auch für größere Projekte und anspruchsvolle Aufgaben genügend Reserven besitzt.

In Java gibt es die meisten elementaren Datentypen, die auch C besitzt. Arrays und Strings sind als Objekte implementiert und sowohl im Compiler als auch im Laufzeitsystem verankert. Methodenlose Strukturtypen wie `struct` oder `union` gibt es in Java nicht. Alle primitiven Datentypen sind vorzeichenbehaftet und in ihrer Größe exakt spezifiziert. Java besitzt einen eingebauten logischen Datentyp `boolean`.

Java bietet semidynamische Arrays, deren initiale Größe zur Laufzeit festgelegt werden kann. Arrays werden als Objekte angesehen, die eine eingeschränkte Menge an Methoden bieten. Mehrdimensionale Arrays werden wie in C dadurch realisiert, daß einfache Arrays ineinander geschachtelt werden. Dabei können auch nicht-rechteckige Arrays erzeugt werden. Alle Array-Zugriffe werden zur Laufzeit auf Einhaltung der Bereichsgrenzen geprüft.

Die Ausdrücke in Java entsprechen weitgehend denen von C und C++. Java besitzt eine `if`-Anweisung, eine `while`-, `do`- und `for`-Schleife und ein `switch`-Statement. Es gibt die von C bekannten `break`- und `continue`-Anweisungen in normaler und mit einem Label versehenen Form. Letztere ermöglicht es, mehr als eine Schleifengrenze zu überspringen. Java besitzt allerdings kein allgemeines `goto`-Statement. Variablendeklarationen werden wie in C++ als Anweisungen angesehen und können an beliebiger Stelle innerhalb des Code-Parts eines Programms auftauchen.

Als OOP-Sprache besitzt Java alle Eigenschaften moderner objektorientierter Sprachen. Wie C++ erlaubt Java die Definition von Klassen, aus denen Objekte erzeugt werden können. Objekte werden dabei als Referenzdatentypen behandelt, die wie Variablen angelegt und verwendet werden können. Zur Initialisierung gibt es Konstruktoren, und es kann eine optionale *Finalizer*-Methode definiert werden, die aufgerufen wird, wenn das Objekt zerstört wird. Seit 1.1 gibt es lokale Klassen, die innerhalb einer anderen Klasse definiert werden.

Alle Methodenaufrufe in Java sind dynamisch. Methoden können überladen werden, Operatoren allerdings nicht. Anders als in C++ ist das Late-Binding standardmäßig aktiviert, kann aber unterdrückt werden. Java erlaubt zwar Einfach-, aber keine Mehrfachvererbung. Mit Hilfe von *Interfaces* (das sind abstrakte Klassendefinitionen, die nur aus Methoden bestehen) ist eine restriktive Form der Mehrfachvererbung möglich, die einen Kompromiß zwischen beiden Alternativen darstellt. Java erlaubt zudem die Definition abstrakter Basisklassen, die neben konkreten auch abstrakte Methoden enthalten, die in abgeleiteten Klassen ausdefiniert werden müssen.

Neben Instanzvariablen und -methoden können auch Klassenvariablen und -methoden definiert werden. Alle Elemente einer Klassendefinition können mit Hilfe der aus C++ bekannten Schlüsselwörter `public`, `private` und `protected` in ihrer Sichtbarkeit eingeschränkt werden. Objektvariablen werden als Referenzen implementiert. Mit ihrer Hilfe ist eine gegenüber C/C++ eingeschränkte Zeigerverarbeitung möglich, die das Erstellen dynamischer Datenstrukturen ermöglicht.

Das Speichermanagement in Java erfolgt automatisch . Während das Erzeugen von Objekten (von Ausnahmen wie String- und Array-Literalen

abgesehen) immer einen expliziten Aufruf des [new](#)-Operators erfordert, erfolgt die Rückgabe von nicht mehr benötigtem Speicher automatisch. Ein Garbage-Collector, der als niedrigpriorisierter Hintergrundprozeß läuft, sucht in regelmäßigen Abständen nach nicht mehr referenzierten Objekten und gibt den durch sie belegten Speicher an das Laufzeitsystem zurück. Viele der Fehler, die bei der Programmierung in C oder C++ dadurch entstehen, daß der Entwickler selbst für das Speichermanagement verantwortlich ist, können in Java nicht mehr auftreten.

In Java gibt es ein strukturiertes Exceptionhandling. Damit ist es möglich, Laufzeitfehler zu erkennen und in strukturierter Weise zu behandeln. Eine Methode muß jeden Laufzeitfehler, der während ihrer Abarbeitung auftreten kann, entweder abfangen oder durch eine geeignete Deklaration an den Aufrufer weitergeben. Dieser hat dann seinerseits die Pflicht, sich um den Fehler zu kümmern. Exceptions sind normale Objekte; die zugehörigen Klassen können erweitert und als Grundlage für anwendungsspezifische Fehler-Handler verwendet werden.

1.2.2 Applets: Eine neue Klasse von Programmen

Eine der am meisten gebrauchten Erklärungen für den überraschenden Erfolg von Java ist die enge Verbindung der Sprache zum Internet und zum World Wide Web. Mit Hilfe von Java ist es möglich, Programme zu entwickeln, die über das Web verbreitet und mit Hilfe eines Browsers wie Netscape Navigator, Sun HotJava oder Microsoft Internet Explorer gestartet werden können. Dazu wurde die Sprache HTML um das APPLET-Tag erweitert. Sie bietet so die Möglichkeit, kompilierten Java-Code in normale Web-Seiten einzubinden.

Ein Java-fähiger Browser enthält einen Java-Interpreter (die virtuelle Java-Maschine, auch kurz *VM* genannt) und die Laufzeitbibliothek, die benötigt wird, um die Ausführung des Programms zu unterstützen. Die genaue Beschreibung der virtuellen Maschine ist Bestandteil der Java-Spezifikation, und Java-VMs sind bereits auf eine große Anzahl unterschiedlicher Plattformen portiert worden. Ein Applet kann damit als eine neue Art von Binärprogramm angesehen werden, das über verschiedene Hardware- und Betriebssystemplattformen hinweg portabel ist und auf einfache Weise im Internet verteilt werden kann.

Im Gegensatz zu den eingeschränkten Möglichkeiten, die Script-Sprachen wie JavaScript bieten, sind Applets vollständige Java-Programme, die alle Merkmale der Sprache nutzen können. Insbesondere besitzt ein Applet alle Eigenschaften eines grafischen Ausgabefensters und kann zur Anzeige von Text, Grafik und Dialogelementen verwendet werden. Einer der großen Vorteile von Applets gegenüber herkömmlichen Programmen ist ihre einfache Distributierbarkeit. Anstelle explizit auszuführender Installationsroutinen lädt der *ClassLoader* des Browsers die Bestandteile eines Applets einfach aus dem Netz und führt sie direkt aus. Das ist vor allem bei kleineren und mittelgroßen Anwendungen in einer lokalen Netzwerkumgebung sehr hilfreich, insbesondere wenn diese sich häufig ändern.

Sicherheit war eines der wichtigsten Designziele bei der Entwicklung von Java, und es gibt eine ganze Reihe von Sicherheitsmechanismen, die verhindern sollen, daß Java-Applets während ihrer Ausführung Schaden anrichten. So ist es einem Applet, das in einem Web-Browser läuft, beispielsweise nicht erlaubt, Dateioperationen auf dem lokalen Rechner durchzuführen oder externe Programme zu starten.

Hinweis

1.2.3 Grafikprogrammierung

Die Java-Laufzeitbibliothek bietet umfassende grafische Fähigkeiten. Diese sind im wesentlichen plattformunabhängig und können dazu verwendet werden, portable Programme mit GUI-Fähigkeiten auszustatten. Seit der Version 1.2 des JDK werden diese Fähigkeiten unter dem Begriff *Java Foundation Classes* (kurz *JFC*) zusammengefaßt, deren drei wichtigste Komponenten die folgenden sind:

- Das *Abstract Windowing Toolkit* (kurz *AWT*) stellt elementare Grafik- und Fensterfunktionen auf der Basis der auf der jeweiligen Zielmaschine verfügbaren Fähigkeiten zur Verfügung.
- Das *Swing Toolset* stellt darüber hinaus eine Reihe zusätzlicher Dialogelemente zur Verfügung und ermöglicht die Konstruktion sehr komplexer grafischer Oberflächen. Mit seinem *Pluggable Look and Feel* bietet es die Möglichkeit, das Look and Feel eines Programmes zur Laufzeit umzuschalten und den Bedürfnissen des jeweiligen Benutzers und den Fähigkeiten der Systemumgebung anzupassen.
- Die dritte wichtige Komponente ist das *Java 2D API*, das komplexe Grafikoperationen und Bildbearbeitungsroutinen zur Verfügung stellt.

Es ist eine bemerkenswerte Innovation, daß Elemente für die GUI-Programmierung in einer Programmiersprache portabel zur Verfügung gestellt werden. Zwar gab es auch früher schon Programmiersprachen, die grafische Fähigkeiten hatten, aber wer einmal die Aufgabe hatte, eine grafische Benutzeroberfläche unter Windows, OS/2, UNIX und auf dem MAC zur Verfügung zu stellen, hatte meistens erheblichen Portierungsaufwand. Mit Standardmitteln der Sprachen C oder C++ und ihren Laufzeitbibliotheken war dies jedenfalls nicht möglich. Mit Java und ihren Klassenbibliotheken stand nun erstmals eine einfach zu verwendende Sprache zur Verfügung, die all diese Dinge bereits bietet.

Hinweis

Wir werden in diesem Buch ausführlich auf das AWT eingehen und damit die Grundlagen der Entwicklung grafischer Oberflächen in Java demonstrieren. Die Java Foundation Classes können aufgrund ihres großen Umfangs nur ansatzweise behandelt werden. Eine ausführliche Behandlung muß daher weiterführenden Titeln vorbehalten bleiben.

Das AWT bietet eine Reihe von elementaren Operationen, um grafische Ausgabeelemente, wie Linien, Polygone, Kreise, Ellipsen, Kreisabschnitte oder Rechtecke, zu erzeugen. Diese Methoden können auch in einem Füllmodus verwendet werden, der dafür sorgt, daß die gezeichneten Flächen mit Farbe ausgefüllt werden. Wie in den meisten Grafik-Libraries realisiert auch Java die Bildschirmausgabe mit Hilfe des Konzepts eines *Grafikkontexts*, der eine Abstraktion des tatsächlichen Ausgabegerätes bildet.

Neben grafischen Elementen kann natürlich auch Text ausgegeben und an beliebiger Stelle innerhalb der Fenster plaziert werden. Text kann skaliert werden, und es ist möglich, mit unterschiedlichen Fonts zu arbeiten. Das AWT bemüht sich, einen portablen Weg zur Font-Auswahl anzubieten,

indem eine Reihe von elementaren Schriftarten auch über Plattformgrenzen hinweg angeboten werden. Mit Hilfe von Font-Metriken können numerische Eigenschaften der verwendeten Schriftarten bestimmt und bei der Ausgabe berücksichtigt werden.

Das Farbmodell von Java basiert auf dem RGB-Modell, das seine Farben additiv auf der Basis der enthaltenen Rot-, Grün- und Blauanteile bestimmt. Daneben wird auch das HSB-Modell unterstützt (*hue, saturation, brightness*), und es gibt Methoden zur Konvertierung zwischen beiden. Das Farbsystem unterstützt eine Reihe von vordefinierten Farben, die plattformübergreifend zur Verfügung stehen.

Neben Grafik kann auch Sound ausgegeben werden. Java unterstützt die Wiedergabe von *au-Dateien* (ein von Sun eingeführtes Format zur Speicherung von digitalen Sound-Samples), und seit der Version 1.2 auch *wav-* und *aiff-Dateien*, die entweder über das Internet oder aus einer lokalen Datei geladen werden können. Die Samples können einmalig oder in einer Schleife wiederholt aufgerufen werden. Daneben ist es möglich, zwei oder mehr Sound-Dateien gleichzeitig abzuspielen. Neben Wave-Dateien gibt es seit dem JDK 1.2 auch die Möglichkeit, *Midi-Dateien* abzuspielen.

Das AWT erlaubt die Anzeige und Manipulation von Bilddaten. Mit Hilfe von Standardmethoden können Grafiken in elementaren Formaten wie GIF oder JPEG geladen, skaliert und auf dem Bildschirm angezeigt werden. Zusätzlich gibt es das Paket `java.awt.image`, das für die Manipulation von Bilddaten entworfen wurde und ausgefeilte Funktionen zur Bild- und Farbmanipulation zur Verfügung stellt.

Wie in den meisten grafischen Entwicklungsumgebungen wird auch beim AWT der Programmfluß durch Nachrichten gesteuert. Sie werden beim Auftreten bestimmter Ereignisse an das Programm gesendet und von diesem in geeigneter Weise behandelt. Java stellt Nachrichten zur Bearbeitung von Maus-, Tastatur, Fenster-, Dialog- und vielen anderen Ereignissen zur Verfügung. Das Event-Handling des JDK 1.1 erlaubt es, Nachrichten an jedes beliebige Objekt zu senden, das die Schnittstelle eines Nachrichtenempfängers implementiert.

Hinweis

1.2.4 Umfangreiche Klassenbibliothek

Die Java-Klassenbibliothek bietet eine ganze Reihe nützlicher Klassen und Interfaces, die eine problemnahe Programmierung ermöglichen. Einige dieser Features sind von Anfang an nützlich, andere erschließen sich erst nach einiger Einarbeitung.

Neben grafischen Ausgabemöglichkeiten stellt Java auch einfache Textausgaben zur Verfügung, ähnlich den entsprechenden Funktionen in C oder C++. Damit ist es möglich, Programme mit einfachen, zeilenorientierten Ein-/Ausgabemöglichkeiten auszustatten, wenn keine aufwendige Benutzerschnittstelle benötigt wird. Einfache Textausgaben werden mit den Methoden der Klasse `PrintStream` erzeugt. Diese erlauben es, alle gängigen Datentypen in ein Terminalfenster auszugeben. Die Klassenvariable `System.out` bietet einen vordefinierten `PrintStream`, der vom Laufzeitsystem initialisiert wird. In ähnlicher Weise steht mit `System.in` die Möglichkeit zur Verfügung, einfache Texteingaben von der Tastatur einzulesen.

Eines der wichtigsten Elemente der Klassenbibliothek ist die Klasse `String`, die Java-Implementierung von Zeichenketten. `String` bietet eine Vielzahl wichtiger Methoden zur Manipulation und zum Zugriff auf Zeichenketten, wie beispielsweise Operationen für numerische Konvertierungen, Zeichen- und Teilstringextraktion sowie für Textsuche und Stringvergleich.

Interessanterweise kann sich der Inhalt eines Strings in Java nicht verändern, sondern bleibt nach seiner Initialisierung fest. Das ist aber nicht weiter störend, denn in Zusammenarbeit mit der Klasse `StringBuffer` (die variabel lange Strings bietet) und der Fähigkeit des Compilers, String-Initialisierung, -Zuweisung und -Verkettung mit Hilfe der impliziten Verwendung der `StringBuffer`-Klasse auszuführen, fällt es gar nicht auf. Dank des automatischen Speichermanagements und der effizienten Konvertierung von `StringBuffer` nach `String` ähnelt der Umgang mit Strings aus der Sicht des Programmierers dem mit variabel langen Zeichenketten in anderen Programmiersprachen. Wegen des automatischen Speichermanagements sind Java-Strings insbesondere sehr viel sicherer als nullterminierte Strings in C oder C++.

Ein `Vector` in Java ist eine lineare Liste, die jede Art von Objekt aufnehmen kann und auf deren Elemente sowohl sequentiell als auch wahlfrei zugegriffen werden kann. Die Länge eines Vektors ist veränderlich, und Elemente können am Ende oder an einer beliebigen anderen Stelle eingefügt werden. Aufgrund dieser Flexibilität kann ein `Vector` oft da verwendet werden, wo ansonsten eine lineare Liste durch Verkettung von Objektreferenzen manuell erstellt werden müßte. Wie gewöhnlich erfolgt auch das Speichermanagement eines Vektors vollkommen automatisch. Neben `Vector` gibt es noch weitere Container-Klassen. So bietet beispielsweise `Hashtable` die Möglichkeit, beliebige Paare von Schlüsseln und Werten zusammenhängend zu speichern und effizient wieder aufzufinden.

Ein nützlicher Mechanismus zum Durchlaufen von Container-Klassen ist das `Enumeration`-Interface. Eine Klasse, die `Enumeration` implementiert, stellt die Methoden `hasMoreElements` und `nextElement` zur Verfügung. Diese können verwendet werden, um in einer Schleife alle Elemente des Containers zu durchlaufen. Die vordefinierten Container-Klassen von Java stellen meistens die Methode `elements` zur Verfügung, die ein Enumeration-Objekt zum Durchlaufen der eigenen Elemente zurückgibt.

Seit dem JDK 1.2 gibt es in Java eine eigene Bibliothek für Container-Klassen, das *Collection-API*. Sie stellt eine umfassende Sammlung an Interfaces für Container-Klassen zur Verfügung und bietet unterschiedliche Implementierungen für verschiedene Anwendungsfälle. Die zuvor erwähnte Klasse `Enumeration` wird hier durch das Interface `Iterator` ersetzt, das einfacher zu bedienen ist. Das Collection-API stellt daneben einige Algorithmen zur Verarbeitung von Containern zur Verfügung (z.B. Sortieren), die es in den einfachen Container-Klassen nicht gab.

Java stellt auch Zufallszahlen zur Verfügung. Das Paket `java.util` bietet eine Klasse `Random`, die das Initialisieren von Zufallszahlengeneratoren und den Zugriff auf ganzzahlige oder Fließkomma-Zufallszahlen ermöglicht. Neben *gleichverteilten* stellt die Klasse `Random` auch *normalverteilte* Zufallszahlen zur Verfügung.

Seit dem JDK 1.1 stehen darüber hinaus eine ganze Reihe hochspezialisierter (und teilweise sehr aufwendiger) Bibliotheken zur Verfügung. So bietet beispielsweise JDBC (*Java Database Connectivity*) den Zugriff auf relationale Datenbanken, *JavaBeans* stellt eine portable Komponentenarchitektur zur Verfügung, und mit dem *Networking-API* und RMI (*Remote Method Invocation*) kann unternehmensweit auf Netzwerkressourcen und verteilte Objekte zugegriffen werden. Per *Serialisierung* können Objekte persistent gemacht werden, und mit dem *Reflection-API* kann der Aufbau von Objekten und Klassen zur Laufzeit untersucht und dynamisch darauf zugegriffen werden. Wir werden in diesem Buch die wichtigsten dieser Bibliotheken ausführlich erläutern.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

1.3 Bewertung

- [1.3 Bewertung](#)
 - [1.3.1 Einige weit verbreitete Mißverständnisse ...](#)
 - [These 1: Java == JavaScript](#)
 - [These 2: Java ist einfach zu lernen](#)
 - [These 3: Java ist portabel](#)
 - [These 4: Java-Programme sind langsam](#)
 - [These 5: Java ist nicht für ernsthafte Anwendungen geeignet](#)
 - [1.3.2 Wird Java erfolgreich sein?](#)
 - [1.3.3 Fazit](#)

1.3.1 Einige weit verbreitete Mißverständnisse ...

Wie bereits im Vorwort erwähnt, wollen wir uns der Java-Euphorie gerne anschließen, sie aber nicht bedingungslos übernehmen. In diesem Abschnitt soll der Versuch unternommen werden, eine Bewertung zu geben und einige der häufigsten Mißverständnisse auszuräumen. Wir wollen dabei zunächst zu einigen Thesen Stellung nehmen, die immer wieder artikuliert werden und häufig zu überzogenen oder falschen Annahmen über das Wesen von Java führen.

These 1: Java == JavaScript

Das ist vollkommen falsch, denn JavaScript ist eine von Netscape entwickelte, proprietäre Script-Sprache zur Erweiterung von HTML-Seiten. Sie ist syntaktisch an Java angelehnt, bietet aber bei weitem nicht so viele Features. Sie wird nicht in Bytecode kompiliert, sondern vom Browser interpretiert und erlaubt weder die Konstruktion von Applets noch von eigenständigen Applikationen. Als Script-Sprache erlaubt sie einfache Manipulationen am Layout und der Interaktionsfähigkeit von HTML-Seiten.

These 2: Java ist einfach zu lernen

Diese gern postulierte These aus der Sprachbeschreibung ist nur bedingt richtig. Java ist eine objektorientierte Programmiersprache mit fortschrittlichen Eigenschaften und muß wie eine solche erlernt werden. Sie ist einfacher zu beherrschen als C oder C++, und es gibt weniger Raum für Anfängerfehler und für Fehler in sehr großen Programmen. Auch die Klassenbibliothek ist leichter zu verstehen, denn sie wurde neu entworfen und kommt ohne die Altlasten von gewachsenen Bibliotheken aus (allerdings merkt man ihr die drei JDK-Versionen 1.0, 1.1 und 1.2 in einigen Bereichen mittlerweile durchaus an). Trotz allem erfordert der Umgang mit Java und seinen Bibliotheken ein nicht zu unterschätzendes Maß an Einarbeitungszeit, bevor man als Entwickler zu produktiven Ergebnissen kommt.

These 3: Java ist portabel

Die Quellcode-Portabilität von Java-Programmen ist etwas höher als die von C- oder C++-Programmen. Das ist hauptsächlich dem Verzicht auf explizite Pointer, maschinennahe Datentypen und Operatoren zu verdanken. Auch die genaue Spezifikation der elementaren Datentypen und die sehr viel größere Zahl an Standard-Bibliotheken tragen zur höheren Quellcode-Portabilität bei.

Der wichtigere Portabilitätsvorteil von Java besteht jedoch darin, daß die kompilierten Programme *binärkompatibel* sind. Ein einmal übersetztes Programm, das innerhalb des Java-Standards entwickelt wurde, wird auf jeder Plattform laufen, die eine Java-VM und die erforderliche Laufzeitumgebung zur Verfügung hat.

These 4: Java-Programme sind langsam

Da Java-Programme in Bytecode übersetzt werden, der nicht direkt vom Prozessor ausgeführt werden kann, muß ein Interpreter diese Aufgabe übernehmen. Dadurch können rechenintensive Java-Programme um den Faktor 10 bis 20 langsamer sein als vergleichbare C/C++-Programme.

Dieses schlechte Ergebnis relativiert sich in der Praxis durch mehrere Umstände. Einerseits sind nur wenige Programme ausgesprochen rechenintensiv. Statt dessen verbringen sie oft einen Großteil ihrer Zeit damit, auf Eingaben des Benutzers oder langsame Datenbank- oder Festplattenzugriffe zu warten. Zudem steigt die Performance von Java-Programmen durch die Entwicklung von Just-In-Time-Compilern, Native-Code-Compilern oder Java-Prozessoren ständig und nähert sich zunehmend der von kompiliertem C/C++-Code an.

Das Performance-Problem kann daher als temporäres Problem angesehen werden - falls es für den speziellen Typ Anwendung überhaupt existiert. Viele Beobachter gehen heute davon aus, daß Java-Programme in einigen Jahren mit derselben Geschwindigkeit laufen werden wie kompilierte C/C++-Programme.

Allerdings kann der unachtsame Entwickler in Java sehr leicht zu einer schlechten Performance beitragen. Wer die abstrakten Designmöglichkeiten von Strings, Readern oder Writern, Collection-Klassen und vielen anderen Bestandteilen der Klassenbibliothek bedenkenlos verwendet, kann das Laufzeitverhalten seines Programmes stark beeinträchtigen. Schon mit der Kenntnis einiger Details über den Aufbau wichtiger JDK-Klassen lassen sich die vorhandenen Bibliotheken weit effizienter nutzen, und die Performance der Programme steigt an. In [Kapitel 29](#) gehen wir auf einige dieser Details ein und zeigen, wie man Java-Programme schreibt, deren Performance für viele Anwendungen adäquat ist.

These 5: Java ist nicht für ernsthafte Anwendungen geeignet

Diese Behauptung resultiert vor allem aus drei Beobachtungen. Zum einen hatten viele der zunächst in Java entwickelten Anwendungen Spielzeug- oder Democharakter. Meist als Applet realisiert, hatten sie lediglich die Aufgabe, eine Homepage zu verschönern oder die Java-Kenntnisse ihrer Entwickler zu demonstrieren. Echten Nutzwert boten dagegen nur wenige Applets. Größere Applikationen, die komplett in Java geschrieben wurden, waren zunächst kaum auf dem Markt.

Zweitens war das Look & Feel von Java-Programmen nicht ausgereift. Tatsächlich bildet das AWT nur einen geringen Anteil der in den jeweiligen plattformspezifischen Fenstersystemen verfügbaren Möglichkeiten ab. Die wenigen Dialogelemente stehen allerdings portabel zur Verfügung. Mit Hilfe des Swing-Toolsets sollte dieses Dilemma gelöst werden. Swing bietet einen umfassenden Satz komplexer Dialogelemente und stellt ihn plattformübergreifend zur Verfügung. Dabei ist es möglich, das Look&Feel der jeweiligen Anwendung zur Laufzeit umzustellen und so dem Geschmack und den Kenntnissen des jeweiligen Benutzers anzupassen. Allerdings hinkt sowohl die Performance als auch die Stabilität von Swing noch hinter der von vergleichbaren plattformspezifischen Libraries (insbesondere unter Windows) hinterher.

Die dritte Beobachtung besagt, daß Java voller Fehler steckt. Während dies weniger für die Sprache selbst, ihre Werkzeuge oder die elementaren Eigenschaften der Klassenbibliothek gilt, kann das AWT ein gewisses Maß an Fehlerhaftigkeit nicht verhehlen. Obwohl mit der Version 1.1 viele der frühen Bugs behoben wurden, kann es auch in seiner heutigen Fassung nicht als ausgesprochen stabiles System bezeichnet werden. Die aktuellen Auslieferungen des JDK 1.2 wurden einem aufwendigen Qualitätssicherungsprogramm mit mehrmonatigem Beta-Test und einer mehrstufigen Auslieferung des endgültigen Releases unterzogen (»release candidates«).

Tatsächlich hat sich in den letzten Monaten - zeitgleich zur Diskussion um die Eignung Javas als Entwicklungswerkzeug für grafische Oberflächen - bereits ein Wandel auf der Serverseite angedeutet. Mit dem *Servlet-API* hat sich beispielsweise im Bereich der Generierung dynamischer Web-Seiten eine Technik etabliert, die dem traditionellen CGI-Scripting ernsthafte Konkurrenz macht. Daneben setzen große Softwarehersteller auf mehrschichtige Client-Server-Architekturen mit Java-Unterstützung (Oracle Financials, Star Office) oder stattdessen ihre Datenbankserver (Oracle 8) oder Web-Server (Lotus Domino, SUN Java Web Server, Apache jserv) mit Java-Fähigkeiten aus. In vielen Unternehmen gibt es bereits verteilte Anwendungen auf der Basis von CORBA, die auf die plattformübergreifenden Fähigkeiten und die Binärkompatibilität von Java-Anwendungen setzen.

1.3.2 Wird Java erfolgreich sein?

Wenn man das Anwenderinteresse zugrunde legt, ist Java schon jetzt die mit Abstand erfolgreichste Programmiersprache der letzten Jahre. Wie oben gezeigt, hat Java in den ersten drei Jahren seit Ausgabe der Version 1.0 eine immense Verbreitung erfahren. Die Sprache war Anlaß für Hunderte von Büchern und Zeitschriftenartikeln, und es entstand eine Reihe von stark frequentierten Newsgroups, die das Interesse an Java deutlich machen. Alle größeren Softwarehersteller unterstützen die Sprache und haben konkrete Produkte realisiert. In vielen Stellenanzeigen werden bereits heute Java-Kenntnisse vom Bewerber verlangt.

Natürlich kann heute niemand sagen, ob Java auch langfristig erfolgreich sein wird und als Programmiersprache auch in zehn oder zwanzig Jahren noch eine Rolle spielen wird. Drei der Kernfaktoren, die den Erfolg von Java ausmachen, sprechen allerdings schon heute dafür:

- Die Geschichte der Programmiersprachen hat schon wiederholt gezeigt, daß umfangreiche und schwer zu lernende Sprachen wie PL/I oder ADA auf Dauer wenig Akzeptanz finden. Andererseits konnten sich auch sehr einfach strukturierte Sprachen nicht durchsetzen. C und C++ haben erfolgreich die Rolle zwischen beiden Extremen eingenommen, sind aber etwas in die Jahre gekommen. Möglicherweise besitzt Java genau die Mischung aus Ausdruckskraft und einfacher Bedien- und Erlernbarkeit, die derzeit gewünscht wird.
- Java-Bytecode ist portabel. Programme, die in Java geschrieben wurden, laufen auf vielen Plattformen, ohne neu kompiliert, neu entworfen oder neu programmiert werden zu müssen. Auch aufwendigere Bestandteile der Sprache, wie z.B. GUI-Fähigkeiten, Threads oder Netzwerkkommunikation, sind portabel. Java ist bereits jetzt auf vielen Plattformen verfügbar, und mit Java ist es für Entwickler einfach und kostengünstig, Programme plattformübergreifend zur Verfügung zu stellen.
- Java bietet völlig neue Perspektiven, Programme zu betreiben und zu verbreiten. Während das Geschäft mit Standardsoftware heute durchgehend von multifunktionalen, schwer zu beherrschenden Monolithen dominiert wird, bietet Java die Möglichkeit, Programme in kleinere Einheiten zu zerlegen, auf natürliche Weise im Netzwerk zu verteilen und sie über das Internet (oder Intranet) upzudaten. Als Sprache für anspruchsvolle Serverlösungen entwickelt sich Java zu einer ernstzunehmenden Konkurrenz zum weit verbreiteten CGI-Scripting mit Perl oder ähnlichen Sprachen.

1.3.3 Fazit

Was überwiegt nun? Die Vorteile oder die Nachteile? Diese Frage sollte jeder für sich selbst beantworten. Dieses Buch wäre nicht geschrieben worden, wenn der Autor nicht der Meinung wäre, daß es die Vorteile sind.

Tatsächlich läßt sich Java heute ohne weiteres dazu verwenden, anspruchsvolle Programme, Tools oder Applets zu entwickeln. Wie dabei mit den bekannten Nachteilen und Unzulänglichkeiten umzugehen ist, muß projektbezogen entschieden werden. Natürlich gibt es dabei eine Korrelation zwischen Innovativität und einer gewissen Unausgereiftheit.

Alleine die umfassenden Aktivitäten aller großen Hard- und Softwarehersteller und ihre Bekenntnisse zu Java lassen erwarten, daß die Kinderkrankheiten ausgemerzt werden und in nicht allzu ferner Zukunft eine Reihe stabiler und ausgereifter Entwicklungsumgebungen zur Verfügung stehen wird. Wer dann bereits auf Erfahrungen in Java-Programmierung zurückgreifen kann, hat möglicherweise den entscheidenden Vorteil im Wettbewerb mit der Konkurrenz.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

1.4 Zusammenfassung

- [1.4 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Historie von Java
- Grundlegende Eigenschaften der Sprache
- Die Bedeutung von Applets
- Eine Bewertung der Eigenschaften von Java
- Eine Prognose über die Zukunft von Java

Kapitel 2

Ein einleitendes Beispiel

- [2 Ein einleitendes Beispiel](#)
 - [2.1 Übersicht](#)
 - [2.2 Schritt 1: Die Programmstruktur](#)
 - [2.3 Schritt 2: Initialisierung des Applets](#)
 - [2.4 Schritt 3: Vorbereitung der Spielsteine](#)
 - [2.5 Schritt 4: Grafikausgabe](#)
 - [2.6 Schritt 5: Mausereignisse](#)
 - [2.7 Zusammenfassung](#)

2.1 Übersicht

- [2.1 Übersicht](#)

Dieses Kapitel gibt einen ersten Überblick über die Java-Programmierung. Mit Hilfe eines größeren Beispiels werden wichtige Spracheigenschaften erklärt und der Umgang mit den Werkzeugen des JDK und der Klassenbibliothek gezeigt. Dabei wurde bewußt ein etwas komplexeres Programm gewählt, das nicht mit der vollen Detailgenauigkeit erläutert wird. Die behandelten Themen werden an anderer Stelle im Buch wiederholt, und es wird jeweils genau erklärt, in welchem Kapitel weiterführende Informationen zu finden sind. Zunächst mag das Programm als Aperitif dienen und einen kleinen Vorgeschmack auf die Möglichkeiten von Java und die kommenden Kapitel geben.

Bitte tippen Sie die Listings in diesem Kapitel nicht ab! Ausnahmsweise dient der abgedruckte Code nur der reinen Anschauung und wurde aus Gründen der Übersichtlichkeit in sehr kleine Einheiten zerlegt. Verwenden Sie die Dateien `Puzzle.java` und `Puzzle.html` der beigefügten CD-ROM, um das Programm in der Praxis ausprobieren zu können. Falls Sie noch kein funktionierendes Java-Entwicklungssystem haben, lesen Sie nach Ende dieses Kapitels zunächst das dritte Kapitel. Dort wird erläutert, wie Sie das JDK installieren und einfache Programme erstellen, übersetzen und ausführen können.

Das vorliegende Kapitel soll Ihnen lediglich einen ersten Eindruck von den Fähigkeiten der Sprache Java und ihrem »Look and Feel« geben. Es macht überhaupt nichts, wenn Sie nicht alles verstehen. Sie könnten sogar das komplette Kapitel überspringen, ohne etwas wirklich Wichtiges zu versäumen. Nutzen Sie einfach diese letzte Gelegenheit, sich zu entspannen, und betrachten Sie die nächsten Seiten wie einen kleinen Film zu einem Thema, das Sie sehr interessiert. Ab [Kapitel 3](#) beginnt dann der eigentliche Lernstoff.

Als Beispielprogramm wurde ein Applet gewählt, das ein Schiebepuzzle realisiert (siehe [Abbildung 2.1](#)).



Abbildung 2.1: Das unsortierte Schiebepuzzle

Das Applet ist in der Lage, ein beliebiges Bild, das im `gif`- oder `jpeg`-Format vorliegt, zu laden, in 16 Spielsteine zu zerlegen und auf dem Bildschirm anzuzeigen. Eines der Spielfelder bleibt dabei unbesetzt und dient als Lücke, um die anderen Steine verschieben zu können. Ziel des Spiels ist es, die 15 Steine so zu ordnen, daß das ursprüngliche Bild wieder entsteht (siehe [Abbildung 2.2](#)). Das Verschieben der Steine erfolgt per Drag & Drop, indem ein benachbarter Stein mit der Maus in die vorhandene Lücke gezogen wird. Zusätzlich kann durch einen Klick auf den Rahmen das Spielfeld sortiert oder vermischt werden. Weitere Bedienelemente besitzt das Programm nicht.

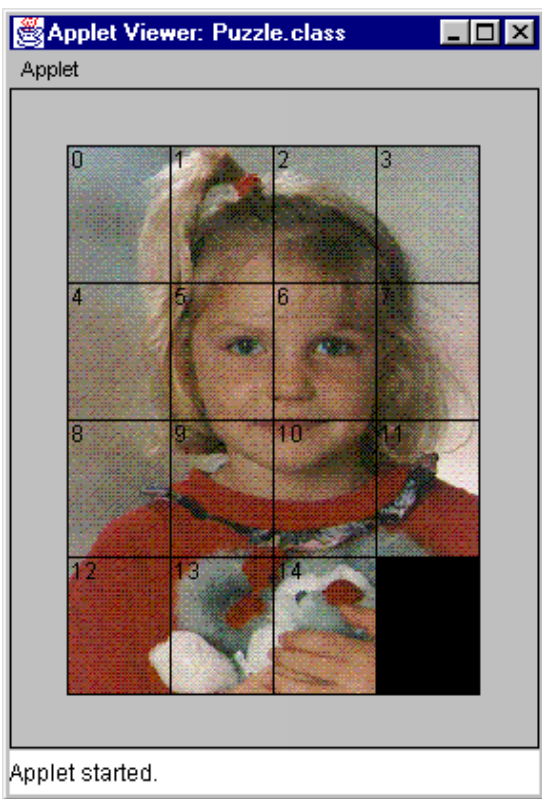


Abbildung 2.2: Das sortierte Schiebepuzzle

2.2 Schritt 1: Die Programmstruktur

- [2.2 Schritt 1: Die Programmstruktur](#)

In Java kann man entweder *Applets* oder *Applikationen* schreiben. Während es sich bei Applikationen um eigenständige Programme handelt, die mit Hilfe des Java-Interpreters gestartet werden können, laufen Applets innerhalb eines Browsers. Sie wurden erfunden, um interaktive Web-Seiten zu entwickeln, die neben Text auch Grafiken, Dialoge und andere Programmfunktionalitäten enthalten können.

Unser Beispielprogramm ist als Applet ausgelegt und sollte daher in jedem Browser ausgeführt werden können, der Java 1.1 beherrscht. Dazu zählen beispielsweise HotJava von Sun, die jüngeren 4er-Versionen des Netscape Navigator und (für unsere Zwecke) der Microsoft Internet Explorer 4.

Jedes Java-Programm - egal ob Applet oder Applikation - besteht aus einer Reihe von Quelldateien mit der Erweiterung `.java`. Diese werden vom Compiler in Bytecode übersetzt und in `.class`-Dateien gespeichert, die dann vom Interpreter ausgeführt werden können. Java ist eine vollständig objektorientierte Sprache, und zu jeder Klasse wird eine eigene `.class`-Datei angelegt. Unser Beispielprogramm besteht lediglich aus einer einzigen Quelldatei `Puzzle.java`, die insgesamt drei Klassen enthält:

- `Puzzle` ist die Hauptklasse. Sie übernimmt die Initialisierung des Applets, die Bildschirmausgabe und die Steuerung des Programms.
- Die anderen beiden Klassen, `MouseListener` und `MouseMotionListener`, dienen zur Behandlung von Mausereignissen.

Üblicherweise wird nur eine Klasse pro Quelldatei definiert. In diesem Fall aber wurden die Klassen `MouseListener` und `MouseMotionListener` lokal zur Hauptklasse deklariert und daher in derselben Quelldatei belassen. In jedem Fall darf nur eine Klasse innerhalb der Quelldatei als `public` deklariert und so anderen Programmen zur Verfügung gestellt werden.

Tip

Die Grundstruktur unseres Programmes sieht so aus:

```

001 /**
002  * @(#)Puzzle.java    1.000 97/07/23
003  *
004  * Copyright (c) 1997 Guido Krueger. All Rights Reserved.
005  *
006  * Dieses Applet ist die Implementierung eines Schiebepuzzles
007  * mit 4 x 4 Feldern. Auf den zunächst unsortierten Spielsteinen
008  * werden die Bestandteile eines Images angezeigt, die dann per
009  * Drag & Drop sortiert werden können. Der einzig erlaubte
010  * Zug besteht darin, einen Stein in die benachbarte Lücke zu
011  * verschieben. Durch Klicken auf den Rahmen kann die Sortierung
012  * umgekehrt werden.
013  *
014  * Das applet-Tag erwartet folgende Parameter:
015  *
016  * bordersize = Breite des Spielfeldrandes
017  * src        = Name der Bilddatei (gif oder jpeg)
018  *
019  */
020 import java.awt.*;
021 import java.awt.event.*;
022 import java.applet.*;
023 import java.util.*;
024
025 public class Puzzle
026 extends Applet
027 {
028     //Variablendeklarationen
029
030     public void init()
031     {
032         //Initialisierung des Applets
033     }
034
035     public void update(Graphics g)

```



```

036     {
037         //Wird überlagert, um die Grafikdarstellung
038         //flimmerfrei zu machen
039     }
040
041     public void paint(Graphics g)
042     {
043         //Grafikausgabe, ruft paintBorder und
044         //paintField auf
045     }
046
047     private void paintBorder(Graphics g)
048     {
049     }
050
051     private void paintField(Graphics g)
052     {
053         //Zeichnet die Spielsteine auf dem Brett.
054     }
055
056     private void prepareImage()
057     {
058         //Lädt das Bild.
059     }
060
061     private void randomizeField(boolean unordered)
062     {
063         //Mischt die Steine auf dem Spielfeld.
064     }
065
066     class MyMouseListener
067     extends MouseAdapter
068     {
069         public void mousePressed(MouseEvent event)
070         {
071             //Maustaste wurde gedrückt.
072         }
073
074         public void mouseReleased(MouseEvent event)
075         {
076             //Maustaste wurde losgelassen
077         }
078
079         private Point getFieldFromCursor(int x, int y)
080         {
081             //Liefert den zur Mausposition passenden horizontalen u.
082             //vertikalen Index des darunterliegenden Steins. Liegt
083             //der Punkt auf dem Rahmen, wird (-1,-1) zurückgegeben.
084             return null;
085         }
086
087         private boolean areNeighbours(Point p1, Point p2)
088         {
089             //Testet, ob die durch p1 und p2 bezeichneten Spielsteine
090             //Nachbarn sind.
091             return false;
092         }
093
094         private void swapRandomization()
095         {
096             //Kehrt die Steineordnung um: falls sie sortiert sind,
097             //werden sie gemischt, andernfalls werden sie sortiert.
098         }
099     }
100

```

```
101 class MyMouseMotionListener
102 extends MouseMotionAdapter
103 {
104     public void mouseDragged(MouseEvent event)
105     {
106         //Maus wurde bei gedrückter Taste bewegt.
107     }
108 }
109 }
```

Listing 2.1: Grundstruktur des Schiebepuzzles

Wir erkennen die drei Klassendefinitionen, die durch das Schlüsselwort `class` eingeleitet werden, und eine Reihe von Methoden, die innerhalb der Klassen definiert wurden. Anstelle des Methodenrumpfs haben wir hier lediglich einen Kommentar eingefügt, der ihre Aufgabe beschreibt. In den nachfolgenden Abschnitten werden alle Methoden näher beschrieben.

Am Anfang des Programms befindet sich ein umfangreicher Kommentar, der seine Aufgabe beschreibt. Dieser Kommentar wurde als Dokumentationskommentar angelegt, der mit dem Werkzeug `javadoc` zur Erzeugung von Sourcecode-Dokumentationen verwendet werden kann. Die nachfolgenden `import`-Anweisungen zeigen dem Java-Compiler, welches die Pakete sind, aus denen das Programm Klassen und Methoden verwenden will. Anders als in C oder C++ gibt es in Java keine separaten Headerfiles, sondern der Compiler liest zum Übersetzungszeitpunkt die `.class`-Dateien der eingebundenen Pakete.

Der abgebildete Programmrumph ist bereits übersetzbar und kann als Applet gestartet werden. Die Übersetzung erfolgt mit Hilfe des Compilerprogramms `javac`, das mit folgendem Kommando aufgerufen wird (wenigstens, wenn das JDK verwendet wird):

```
javac Puzzle.java
```

Nach der Übersetzung existieren drei Klassendateien `Puzzle.class`, `Puzzle$MyMouseMotionListener.class` und `Puzzle$MyMouseListener.class`, die im selben Verzeichnis wie die Quelldatei liegen. Mit Hilfe des AppletViewers könnte das übersetzte Programm nun gestartet werden. Zuvor benötigen wir allerdings noch eine passende HTML-Datei, die das Applet einbindet. Diese werden wir im nächsten Abschnitt vorstellen.

Weitere Informationen zur Strukturierung von Programmen und zur Einbindung von Paketen finden sich in [Kapitel 8](#). [Kapitel 7](#) liefert eine ausführliche Erklärung der objektorientierten Programmierung und erklärt die Verwendung von Klassen und Methoden. Die Entwicklung von Applets wird in [Kapitel 25](#) erklärt. [Kapitel 26](#) erläutert die Verwendung von `javadoc` zur Generierung von HTML-Dokumenten aus Java-Quelltexten und beschreibt den Aufruf des Compilers `javac` und des Interpreters `java`.

Hinweis

2.3 Schritt 2: Initialisierung des Applets

- [2.3 Schritt 2: Initialisierung des Applets](#)

Um das Beispielprogramm nach dem Übersetzen zu starten, wird der *Appletviewer* benötigt. Der Appletviewer ist ein Programm, das eine einfache HTML-Datei interpretiert, die darin befindlichen APPLET-Tags extrahiert und die zugehörigen Applets aufruft. Ein Applet erfordert zum Start also immer eine HTML-Datei, in die es eingebunden wird. Dazu wird ein oder mehrere APPLET-Tags eingebunden, die den Namen der zu verwendenden Klassendatei, die Größe des Applets und seine Parameter angeben. Alle übrigen HTML-Anweisungen werden vom Appletviewer, der natürlich kein vollständiger Web-Browser ist, ignoriert.

Eine einfache HTML-Datei, die unser Applet lädt, sieht so aus:

[Puzzle.html](#)

```
001 <!-- Puzzle.html -->
002 <html>
003 <head>
004 <title>Puzzle</title>
005 </head>
006 <body>
007 <h1>Puzzle</h1>
008 <applet code=Puzzle.class width=278 height=348>
009 <param name="borderSize" value=30>
010 <param name="src" value="mine.gif">
011 Hier steht das Applet Puzzle.class
012 </applet>
013 </body>
014 </html>
```

Listing 2.2: HTML-Datei zum Laden des Schiebepuzzle-Applets

Sie enthält die üblichen Formalien einer HTML-Datei wie [HTML](#)-, [HEAD](#)- und [BODY](#)-Tags und das APPLET-Tag zum Aufruf unseres Applets. Die erforderlichen Parameter [code](#), [width](#) und [height](#) geben dabei den Namen der Klassendatei sowie die Breite und Höhe des zur Ausgabe verfügbaren Bildschirmbereichs an. Die mit dem PARAM-Tag definierten Parameter [borderSize](#) und [name](#) werden an das Applet weitergereicht und von diesem zur Einstellung der Rahmengröße und zur Auswahl der Bilddatei verwendet. Das Applet kann mit folgendem Kommando gestartet werden:

```
appletviewer Puzzle.html
```

Da alle Methoden leer sind, hat das Applet natürlich noch keinerlei benutzerdefinierte Funktionalität und es wird eine leere Ausgabefläche angezeigt (siehe [Abbildung 2.3](#)):

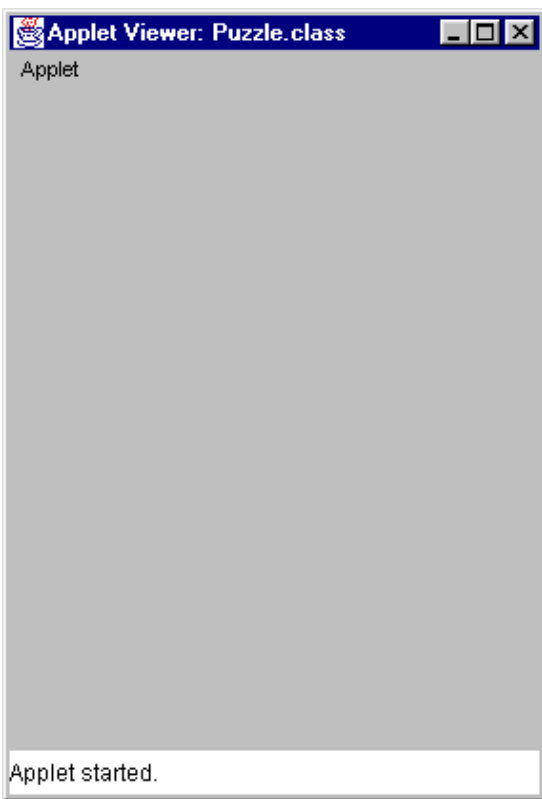


Abbildung 2.3: Ein leeres Applet

Ein Applet wird immer aus der Klasse `Applet` abgeleitet. Sie besitzt eine Reihe von Methoden, die zu Initialisierungszwecken überlagert werden können. Die Methode `init` ist eine davon. Sie wird lediglich ein einziges Mal aufgerufen und dient dazu, Einmalinitialisierungen vorzunehmen. Es gibt andere Methoden, die bei jedem Aufruf der zugehörigen HTML-Seite oder beim Verlassen derselben aufgerufen werden.

```

001 public void init()
002 {
003     aFields      = new int[4][4];
004     sourcefield  = new Point(-1, -1);
005     lastpoint    = new Point(-1, -1);
006     drawoffset   = new Point(0,0);
007     bordersize   = Integer.parseInt(getParameter("bordersize"));
008     if (bordersize < 1 || bordersize > 50) {
009         bordersize = 5;
010     }
011     setBackground(Color.lightGray);
012     addMouseListener(new MyMouseListener());
013     addMouseMotionListener(new MyMouseMotionListener());
014     prepareImage();
015     randomizeField(true);
016 }

```

Listing 2.3: Die `init`-Methode des Schiebepuzzle-Applets

In unserem Fall initialisiert `init` eine Reihe von Variablen bzw. erzeugt die erforderlichen Objektreferenzen. Objekte werden - von wenigen Ausnahmen abgesehen - immer mit Hilfe des `new`-Operators erzeugt, der von dem Klassennamen und der Liste der Parameter, die an den Konstruktor übergeben werden, gefolgt wird. Alle hier initialisierten Variablen wurden als Instanzvariablen im Kopf der Klasse deklariert (dort, wo im obigen Rumpf der Kommentar `//Variablendeklarationen` steht ([Zeile 028](#))):

```

001 int          aFields[][];           //Brett mit allen Feldern
002 Image        image;                 //Bildspeicher
003 int          bordersize;            //Randbreite
004 Dimension    fieldsize;             //Größe eines Feldes
005 Dimension    imagesize;             //Größe des Bildes
006 Point        sourcefield;           //Bei Mausklick ausgewähltes Feld
007 Point        lastpoint;             //Ursprung des letzten Rechtecks
008 Point        drawoffset;            //Offset zur Mausdragposition

```

Listing 2.4: Variablendeklarationen für das Schiebepuzzle

Hier sind sowohl einfache Typen als auch Objektvariablen zu finden. Als Membervariablen sind sie nur innerhalb der Klasse `Puzzle` sichtbar, andere Klassen haben auf sie keinen direkten Zugriff.

Innerhalb von `init` wird auch der Applet-Parameter `bordersize` gelesen, der mit Hilfe des `PARAM`-Tags in der HTML-Datei übergeben wurde. Mit diesem sehr allgemeinen Parameterübergabemechanismus ist es möglich, nahezu beliebige Argumente an Applets zu übergeben und diese so in weiten Bereichen konfigurierbar zu machen.

Hinweis

Die übrigen Anweisungen in `init` sind Methodenaufrufe. Mit `addMouseListener` und `addMouseMotionListener` werden die Objekte zur Behandlung von Mausereignissen erzeugt und registriert. Weiterhin wird durch Aufruf von `setBackground` die Hintergrundfarbe durch Übergabe eines `Color`-Objekts auf hellgrau eingestellt. Die anderen Methoden, `prepareImage` und `randomizeField`, sind lokale Methoden der Klasse `Puzzle`, die wir weiter unten erläutern.

Weiterführende Informationen zur Applet-Programmierung und den verschiedenen Parametern des `APPLET`-Tags finden sich in [Kapitel 25](#). Die Definition von Variablen wird in [Kapitel 4](#) erläutert, das Instanzieren von Objekten und der Aufruf von Methoden in [Kapitel 7](#). In [Kapitel 5](#) und [Kapitel 6](#) werden Ausdrücke und Anweisungen behandelt. Methoden zur Grafikausgabe finden sich in [Kapitel 14](#) und den folgenden Kapiteln.

Hinweis

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	<	>	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	----------------------	----------------------	--------------------------

2.4 Schritt 3: Vorbereitung der Spielsteine

- 2.4 Schritt 3: Vorbereitung der Spielsteine

Während der Initialisierung wird zunächst die Methode `prepareImage` aufgerufen, um die Bilddatei zu laden:

```

001 /**
002  * Lädt das Bild.
003  */
004 private void prepareImage()
005 {
006     //Bild laden
007     image = getImage(getDocumentBase(),getParameter("src"));
008     MediaTracker mt = new MediaTracker(this);
009     mt.addImage(image, 0);
010     try {
011         //Warten, bis das Image vollständig geladen ist,
012         mt.waitForAll();
013     } catch (InterruptedException e) {
014         //nothing
015     }
016     imagesize      = new Dimension();
017     imagesize.height = image.getHeight(this);
018     imagesize.width  = image.getWidth(this);
019 }

```

Listing 2.5: Vorbereitung der Spielsteine im Schiebepuzzle

Der eigentliche Ladevorgang ist ganz einfach und erfordert lediglich den Aufruf der Methode `getImage`. Diese erwartet als erstes Argument die Basisadresse des Bildes, in diesem Fall das mit `getDocumentBase` ermittelte Verzeichnis, aus dem die aktuelle HTML-Seite geladen wurde. Als zweiter Parameter wird der Name der Bilddatei übergeben, die hier durch Aufruf von `getParameter` aus dem Applet-Parameter `src` gewonnen wird.

Etwas aufwendig wird die Methode, um auf den Abschluß des Ladevorgangs zu warten. `getImage` geht davon aus, daß Bilder oftmals über langsame Netzwerkverbindungen geladen werden müssen und führt das Laden daher asynchron mit Hilfe eines eigenen Threads aus. Da wir am Ende der Methode die Größe des Bildes ermitteln müssen, um die Berechnung der Teilbilder auf den Spielsteinen korrekt durchführen zu können, muß das Programm warten, bis das Bild vollständig geladen ist. Dazu instanziert es ein Objekt der Klasse `MediaTracker`, das in der Lage ist, den Ladevorgang eines oder mehrerer Bilder durch Aufruf von `waitForAll` zu überwachen. Nach Abschluß des Ladevorgangs wird eine Ausnahme vom Typ `InterruptedException` gesendet, nach der das Programm fortfährt. Anschließend wird das fertige Bild in der Instanzvariablen `image` abgelegt.

Nach dem Laden des Bildes wird durch Aufruf von `randomizeField` die Anordnung der Spielsteine durcheinandergebracht. Die Spielsteine werden durch das zweidimensionale Array `aFields` repräsentiert, das eine Größe von 4 mal 4 Elementen hat. Arrays sind in Java *Objekte*, die wie andere Objekte dynamisch instanziiert werden müssen. Unser Programm hat dies in `init` mit Hilfe der folgenden Anweisung erledigt:

```
aFields      = new int[4][4];
```

In jedem Element von `aFields` steht einer der Werte 0 bis 15, der anzeigt, welches der 16 Teilbilder an der entsprechenden Position angezeigt werden soll. Das letzte Bild (also Nummer 15) repräsentiert die Lücke, die von den Anzeigeroutinen als schwarzes Rechteck dargestellt wird. `randomizeField` sortiert zunächst das Array, indem die Zahlen von 0 bis 15 in aufsteigender Reihenfolge hineingeschrieben werden. Anschließend werden die Steine vermischt, indem zwanzigmal nacheinander je zwei zufällig ausgewählte Elemente miteinander vertauscht werden:

```

001 /**
002  * Mischt die Steine auf dem Spielfeld.
003  */
004 private void randomizeField(boolean unordered)
005 {
006     int i, j, k, tmp;
007
008     //Zuerst sortieren...
009     for (i = 0; i <= 15; ++i) {
010         aFields[i / 4][i % 4] = i;
011     }
012     //Dann mischen...
013     if (unordered) {
014         Random rand = new Random(System.currentTimeMillis());
015         for (i = 0; i < 20; ++i) {
016             j = Math.abs(rand.nextInt()) % 16;
017             k = Math.abs(rand.nextInt()) % 16;
018             tmp = aFields[j / 4][j % 4];
019             aFields[j / 4][j % 4] = aFields[k / 4][k % 4];
020             aFields[k / 4][k % 4] = tmp;
021         }
022     }
023 }

```

Listing 2.6: Mischen der Spielsteine im Schiebepuzzle

Zufallszahlengeneratoren in Java sind Objekte der Klasse [Random](#). Der Zufallszahlengenerator, der in diesem Fall mit der aktuellen Systemzeit initialisiert wird, ist in der Lage, positive und negative Zufallszahlen für Ganz- und Fließkommazahlen zu erzeugen.

Die Klasse [Math](#) stellt eine Reihe von Methoden zur Verfügung, die bei der Verarbeitung von numerischen Werten hilfreich sind. Sie besitzt überwiegend *Klassenmethoden*, die ohne zugeordnetes Objekt aufgerufen werden können. Klassenmethoden sind das Pendant zu globalen Funktionen anderer Programmiersprachen, müssen aber immer zusammen mit dem Namen der Klasse, in der sie deklariert wurden, aufgerufen werden.

Tip

Weiterführende Informationen zum Laden und Anzeigen von Bildern finden sich in [Kapitel 24](#), dort wird auch das Konzept der Klasse [MediaTracker](#) genau erläutert. Die Behandlung von Ausnahmen und das [Exception](#)-Konzept werden in [Kapitel 9](#) vorgestellt. Die Instanzierung von Arrays wird in [Kapitel 4](#) behandelt und die Methoden der Klasse [Math](#) stammen aus [Kapitel 7](#), in dem auch die Verwendung von Klassenmethoden beschrieben wird. Der Zufallszahlengenerator und die Methoden der Klasse [System](#) werden in [Kapitel 12](#), das sich mit Utility-Klassen beschäftigt, vorgestellt.

Hinweis

2.5 Schritt 4: Grafikausgabe

- 2.5 Schritt 4: Grafikausgabe

Die Grafikausgabe erfolgt in allen Java-Programmen durch Aufruf der Methode `paint`. Ausnahmen davon sind lediglich Terminalausgaben, die vom Programm durch Aufruf von `System.out.println` erzeugt werden können. `paint` wird in abgeleiteten Klassen überlagert und sorgt für die ordnungsgemäße Darstellung der Bildschirmausgabe. Die Methode bekommt ein Objekt vom Typ `Graphics` übergeben, das einen *Grafikkontext* darstellt. Dieser stellt die Methoden zur Ausgabe von Text und Grafik zur Verfügung und kann am besten als abstraktes Ausgabegerät aufgefaßt werden. Ein Grafikkontext wird auch dann verwendet, wenn Druckausgaben erzeugt werden sollen oder die Ausgabe in einem Offscreen-Image erfolgt. In unserem Fall ruft `paint` die beiden Methoden `paintBorder` und `paintField` auf:

```
001 public void paint(Graphics g)
002 {
003     paintBorder(g);
004     paintField(g);
005 }
```

Listing 2.7: Die Methode `paint` im Schiebepuzzle

An beide Methoden wird der Grafikkontext übergeben, auf dem die Ausgabeoperationen vorgenommen werden. `paintBorder` ist dafür verantwortlich, den Rahmen um das Spielfeld zu zeichnen:

```
001 /**
002  * Zeichnet den Rahmen des Spielbretts.
003  */
004 private void paintBorder(Graphics g)
005 {
006     Insets insets    = getInsets();
007     Dimension size   = getSize();
008     size.height      -= insets.top + insets.bottom;
009     size.width       -= insets.left + insets.right;
010     fieldsize        = new Dimension();
011     fieldsize.width  = (size.width - (2 * bordersize)) / 4;
012     fieldsize.height = (size.height - (2 * bordersize)) / 4;
013     g.setColor(Color.black);
014     g.drawRect(
015         insets.left,
016         insets.top,
017         size.width - 1,
018         size.height - 2
019     );
020     g.drawRect(
021         insets.left + bordersize,
022         insets.top + bordersize,
023         4 * fieldsize.width,
024         4 * fieldsize.height
025     );
026 }
```

Listing 2.8: Zeichnen des Rahmens im Schiebepuzzle

Dazu ermittelt die Methode durch Aufruf von `getInsets` zunächst die Größe aller Randelemente wie Fensterrahmen, Menüzeile oder Statuszeile. Da das Java-Koordinatensystem in der linken oberen Ecke des zugrundliegenden Fensters beginnt (und nicht in der linken oberen Ecke des Client-Bereichs), müssen diese Werte berücksichtigt werden, damit nicht Teile der Ausgabe hinter den Randelementen verschwinden. Des weiteren ermittelt die Methode durch Aufruf von `getSize` die Größe des Applet-Fensters, um zusammen mit der als Parameter übergebenen Rahmengröße die Abmessung eines einzelnen Spielsteins zu berechnen. Diese wird der Instanzvariable `fieldsize` vom Typ `Dimension` zugewiesen und bei der späteren Anzeige des Spielfelds verwendet.

Anschließend wird die Zeichenfarbe auf schwarz geschaltet und die beiden Rechtecke zur Begrenzung des Rahmens gezeichnet. Bei diesem Rahmen handelt es sich natürlich nicht um die oben erwähnten Rahmenelemente des Java-Fensters, sondern um den Rand, den wir zur Begrenzung des Spielfelds selbst zeichnen. Würde innerhalb von `paint` lediglich `paintBorder` aufgerufen, so hätte das Applet das in [Abbildung 2.4](#) gezeigte

Aussehen.

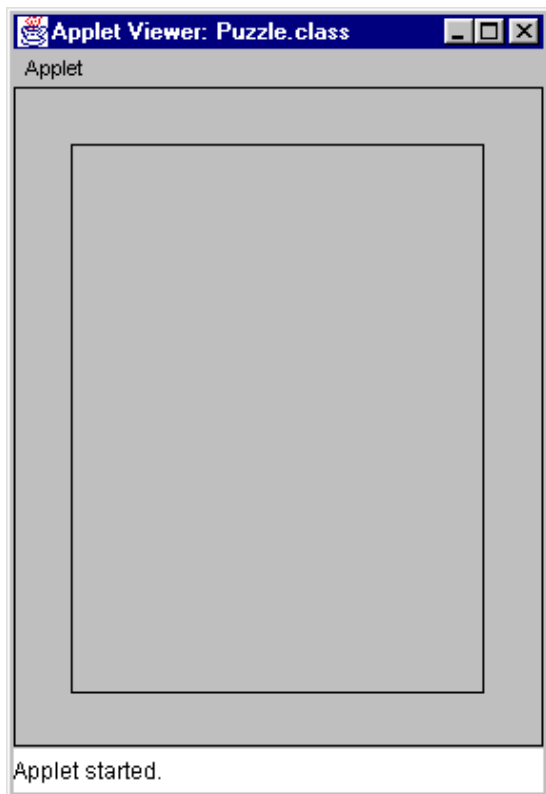


Abbildung 2.4: Die Rahmendarstellung

Die Darstellung des Spielfelds erfolgt in der Methode `paintField`, die ebenfalls den Grafikkontext aus `paint` übergeben bekommt. `paintField` bestimmt durch Aufruf von `getInsets` zunächst die linke obere Ecke des Client-Bereichs und setzt dann die Zeichenfarbe auf schwarz. Anschließend wird das Array `aFields` zeilenweise durchlaufen und für jede der 16 Positionen auf dem Bildschirm die Nummer des darauf befindlichen Spielsteins ermittelt. Falls es sich um die Nummer 15 handelt, wird mit `fillRect` ein schwarzes Rechteck in der Größe eines Spielsteins gezeichnet, um die Lücke darzustellen. Andernfalls wird der Spielstein gezeichnet. Dazu geht das Programm in drei Schritten vor:

- Zunächst wird die eigentliche Bildinformation gezeichnet. Dazu wird `drawImage` verwendet, um den passenden Teil aus dem ursprünglichen Image auszuschneiden und an der aktuellen Stelle in der Größe eines Spielsteins auf dem Bildschirm auszugeben. `drawImage` gibt es in verschiedenen Versionen, die eine recht flexible Ausgabe von Bildern erlauben. So kann neben einer kompletten Ausgabe ein Bild zuvor gespiegelt oder skaliert werden oder lediglich ein Teil des Bildes angezeigt werden.
- Anschließend wird der Rahmen um den Spielstein gezeichnet. Dazu wird ein Aufruf von `drawRect` verwendet, der genauso parametrisiert ist wie der Aufruf von `fillRect` zum Zeichnen der Lücke. Die meisten elementaren Grafikfunktionen des AWT gibt es wahlweise in einer `draw`- oder `fill`-Variante. Sie unterscheiden sich dadurch, daß die eine nur die Umrandung des Objekts zeichnet, während die andere es mit Farbe ausfüllt.
- Drittens wird die Beschriftung des Spielsteins ausgegeben. Dazu wird durch Aufruf von `drawString` die interne Nummer des Spielsteins in die linke obere Ecke geschrieben.

Hier ist der Quelltext von `paintField`:

```
001 /**
002  * Zeichnet die Spielsteine auf dem Brett.
003  */
004 private void paintField(Graphics g)
005 {
006     int imagenumber, image_i, image_j;
007     Insets insets = getInsets();
008     Point topleft = new Point();
009     topleft.x      = insets.left + bordersize;
010     topleft.y      = insets.top  + bordersize;
011     g.setColor(Color.black);
012     for (int i = 0; i <= 3; ++i) {
013         for (int j = 0; j <= 3; ++j) {
014             imagenumber = aFields[i][j];
015             if (imagenumber == 15) {
016                 //Lücke zeichnen
```

```

017         g.fillRect(
018             topleft.x + j * fieldsize.width,
019             topleft.y + i * fieldsize.height,
020             fieldsize.width,
021             fieldsize.height
022         );
023     } else {
024         //Image darstellen
025         image_i = imagenumber / 4;
026         image_j = imagenumber % 4;
027         g.drawImage(
028             image,
029             topleft.x + j * fieldsize.width,
030             topleft.y + i * fieldsize.height,
031             topleft.x+j*fieldsize.width+fieldsize.width,
032             topleft.y+i*fieldsize.height+fieldsize.height,
033             image_j*(imagesize.width/4),
034             image_i*(imagesize.height/4),
035             image_j*(imagesize.width/4)+imagesize.width/4,
036             image_i*(imagesize.height/4)+imagesize.height/4,
037             this
038         );
039         //Rahmen
040         g.drawRect(
041             topleft.x + j * fieldsize.width,
042             topleft.y + i * fieldsize.height,
043             fieldsize.width,
044             fieldsize.height
045         );
046         //Beschriftung
047         g.drawString(
048             "" + imagenumber,
049             topleft.x + j * fieldsize.width + 2,
050             topleft.y + i * fieldsize.height + 12
051         );
052     }
053 }
054 }
055 }

```

Listing 2.9: Zeichnen der Spielsteine des Schiebepuzzles

Mit den beiden bisher beschriebenen Methoden wäre die Grafikausgabe eigentlich schon komplett. Die zusätzlich vorhandene Methode [update](#), die normalerweise nicht überlagert werden muß, dient dazu, das Bildschirmflackern zu vermindern:

Tip

```

001 public void update(Graphics g)
002 {
003     Image      dbImage;
004     Graphics   dbGraphics;
005
006     //Double-Buffer initialisieren
007     dbImage = createImage(getSize().width,getSize().height);
008     dbGraphics = dbImage.getGraphics();
009     //Hintergrund löschen
010     dbGraphics.setColor(getBackground());
011     dbGraphics.fillRect(0,0,getSize().width,getSize().height);
012     //Vordergrund zeichnen
013     dbGraphics.setColor(getForeground());
014     paint(dbGraphics);
015     //Offscreen-Image anzeigen
016     g.drawImage(dbImage,0,0,this);
017     dbGraphics.dispose();
018 }

```

Listing 2.10: Die überlagerte update-Methode im Schiebepuzzle

Ihre Arbeitsweise beruht auf dem Einsatz eines *Offscreen-Images*, das zur Zwischenspeicherung der Grafikausgabe verwendet wird. Das Verfahren wird auch als *Doppelpufferung* bezeichnet, weil ein zweiter Bildschirmpuffer zur Ausgabe verwendet wird. [update](#) wird in der Aufrufkette, die bei einem Neuzeichnen des Bildschirms ausgelöst wird, unmittelbar vor [paint](#) aufgerufen. Ihre Aufgabe besteht darin, zunächst den Hintergrund mit der aktuellen Hintergrundfarbe zu füllen, dann die Vordergrundfarbe zu aktivieren und schließlich [paint](#) aufzurufen.

Genau das erledigt auch unsere Variante von [update](#). Alle Zeichenoperationen spielen sich dabei allerdings in einem separaten Puffer ab, der erst nach kompletter Fertigstellung in den Bildschirm eingeblendet wird. Das unschöne Bildschirmflackern, das dadurch entsteht, daß unmittelbar vor der Darstellung dunkler Flächen die darunterliegende helle Fläche des Hintergrunds neu gezeichnet wird, kann so vermieden werden. Der Nachteil ist die etwas verringerte Ausgabegeschwindigkeit und der recht hohe Speicherbedarf, denn bei jedem Neuzeichnen wird ein neues Offscreen-Image erzeugt.

Weiterführende Informationen zur Grafikausgabe finden sich in [Kapitel 14](#) und den folgenden **Hinweis** Kapiteln. [Kapitel 14](#) erläutert dabei insbesondere die grundlegenden Zeichenfunktionen und beschreibt den Grafikkontext. [Kapitel 15](#) erläutert die Textausgabe und [Kapitel 16](#) das Farbsystem von Java. Die zeilenorientierte Ausgabe von einfachen Texten wird in [Kapitel 3](#) erläutert und in [Kapitel 12](#) noch einmal aufgegriffen. In [Kapitel 24](#) werden verschiedene Techniken zur Reduzierung des Bildschirmflackerns vorgestellt.


```

023     }
024     }
025 }
026 }
027 return ret;
028 }
029
030
031 /**
032  * Kehrt die Steineordnung um: falls sie sortiert sind,
033  * werden sie gemischt und umgekehrt.
034  */
035 private void swapRandomization()
036 {
037     //Sind die Felder sortiert?
038     boolean sorted = true;
039     for (int i = 0; i <= 15; ++i) {
040         if (aFields[i / 4][i % 4] != i) {
041             sorted = false;
042             break;
043         }
044     }
045     //Neu mischen bzw. sortieren
046     randomizeField(sorted);
047 }

```

Listing 2.12: Lokale Methoden im Schiebepuzzle

`swaprandomization` ist sehr einfach aufgebaut. In einer Schleife wird zunächst überprüft, ob alle Spielsteine in aufsteigender Reihenfolge angeordnet sind. Abhängig vom Ergebnis wird `randomizeField` mit dem Sortieren oder Vermischen der Spielsteine beauftragt.

`getFieldFromCursor` bestimmt zunächst die linke obere Ecke des Client-Bereichs und die Größe des Spielfelds. Anschließend wird überprüft, ob zum Zeitpunkt des Mausklicks der Mauszeiger innerhalb dieses Bereichs gelegen hat. Ist dies nicht der Fall, wird (-1, -1) zurückgegeben, andernfalls wird der Index des Spielsteins ermittelt, über dem der Mauszeiger stand. Dazu wird der jeweilige `x`- bzw. `y`-Offset in die Client-Area durch die Breite bzw. Länge eines einzelnen Spielsteins dividiert.

Anschließend wird der `x`- und `y`-Abstand des Mauszeigers von der linken oberen Ecke des unter dem Mauszeiger befindlichen Spielsteins ermittelt und in `drawoffset` gespeichert. Dieser Wert wird benötigt, um beim Ziehen der Maus das Rechteck konsistent mit dem Mauszeiger mitführen zu können. Das zweite Mausereignis, auf das unser Programm reagieren muß, tritt auf, wenn die Maustaste losgelassen wird. In diesem Fall wird die Methode `mouseReleased` aufgerufen:

Hinweis

```

001 /**
002  * Maustaste losgelassen.
003  */
004 public void mouseReleased(MouseEvent event)
005 {
006     if (sourcefield.x != -1 && sourcefield.y != -1) {
007         Point destfield;
008         destfield = getFieldFromCursor(event.getX(), event.getY());
009         if (destfield.x != -1 && destfield.y != -1) {
010             if (aFields[destfield.y][destfield.x] == 15) {
011                 if (areNeighbours(sourcefield, destfield)) {
012                     aFields[destfield.y][destfield.x] =
013                         aFields[sourcefield.y][sourcefield.x];
014                     aFields[sourcefield.y][sourcefield.x] = 15;
015                 }
016             }
017         }
018         repaint();
019     }
020     sourcefield.x = -1;
021     sourcefield.y = -1;
022 }

```

Listing 2.13: Die Methode `mouseReleased` des Schiebepuzzles

[mouseReleased](#) überprüft zunächst, ob der zugehörige Mausklick *innerhalb* des Spielfelds ausgelöst wurde. Ist dies nicht der Fall (beispielsweise, weil die Maus vom Rand ins Spielfeld gezogen wurde), erfolgt keine weitere Bearbeitung. Andernfalls wird durch erneuten Aufruf von [getFieldFromCursor](#) der Spielstein ermittelt, auf dem die Maus gelandet ist. Falls dieser innerhalb des Spielfelds liegt, wird geprüft, ob die Maus auf der Lücke gelandet ist. Ist auch das der Fall, wird die Methode [areNeighbours](#) aufgerufen, um festzustellen, ob der Stein, auf dem die Maustaste gedrückt wurde, und die Lücke, auf der sie losgelassen wurde, Nachbarn auf dem Spielfeld sind:

```
001 /**
002  * Testet, ob die durch p1 und p2 bezeichneten Spielsteine
003  * Nachbarn sind.
004  */
005 private boolean areNeighbours(Point p1, Point p2)
006 {
007     int aNeighbours[][] = {{-1,0},{0,-1},{0,1},{1,0}};
008     for (int i = 0; i < aNeighbours.length; ++i) {
009         if (p1.x + aNeighbours[i][0] == p2.x) {
010             if (p1.y + aNeighbours[i][1] == p2.y) {
011                 return true;
012             }
013         }
014     }
015     return false;
016 }
```

Listing 2.14: Test benachbarter Spielsteine im Schiebepuzzle

Nur, wenn alle diese Bedingungen erfüllt sind, handelt es sich um eine gültige Operation, und der Spielstein darf verschoben werden. Dazu wird im Array [aFields](#) einfach an die Zielposition der Wert des Spielsteins, auf dem die Maus gedrückt wurde, abgelegt und der Inhalt des Quellsteins anschließend mit 15, also der Nummer der Lücke, überschrieben. Damit die Aktion auf dem Bildschirm sichtbar wird, ruft die Methode anschließend [repaint](#) auf, und das Fenster wird neu aufgebaut.

Um während des Ziehens der Maus einen visuellen Effekt zu erzielen, wird schließlich noch die Methode [mouseDragged](#) in der Klasse [MyMouseListener](#) implementiert. Diese wird immer dann aufgerufen, wenn eine Mausbewegung bei gedrückter Maustaste erfolgt:

```
001 /**
002  * Maus wurde bei gedrückter Taste bewegt.
003  */
004 public void mouseDragged(MouseEvent event)
005 {
006     if (sourcefield.x != -1 && sourcefield.y != -1) {
007         Graphics g = getGraphics();
008         g.setXORMode(getBackground());
009         g.setColor(Color.black);
010         //Das zuletzt gezeichnete Rechteck entfernen
011         if (lastpoint.x != -1) {
012             g.drawRect(
013                 lastpoint.x - drawoffset.x,
014                 lastpoint.y - drawoffset.y,
015                 fieldsize.width,
016                 fieldsize.height
017             );
018         }
019         //Neues Rechteck zeichnen
020         g.drawRect(
021             event.getX() - drawoffset.x,
022             event.getY() - drawoffset.y,
023             fieldsize.width,
024             fieldsize.height
025         );
026         lastpoint.x = event.getX();
027         lastpoint.y = event.getY();
028         g.dispose();
029     }
030 }
```

Listing 2.15: Die Methode mouseDragged im Schiebepuzzle

[mouseDragged](#) überprüft zunächst, ob das Ziehen der Maus wirklich innerhalb des Spielfeldes gestartet wurde. Ist dies nicht der Fall, wäre die in

`sourcefield` gespeicherte Position des Startfeldes (-1, -1) und `mouseDragged` würde keine weiteren Aktionen vornehmen. Ist dies aber der Fall, beschafft sich die Methode durch Aufruf von `getGraphics` zunächst einen Grafikkontext, um (außerhalb von `paint`) im Client-Bereich des Fensters Ausgaben vornehmen zu können.

Programme, die ein Drag & Drop implementieren, visualisieren dies in der Regel so, daß sie während der Drag-Operation eine vereinfachte Version des Quellobjekts anzeigen und synchron mit der Maus mitführen. Um dabei nicht jeweils ein vollständiges Neuzeichnen erforderlich zu machen, bietet der Device-Kontext einen *XOR-Modus* an, bei dem alle Zeichenoperationen die jeweilige Pixelfarbe des Untergrundes invertieren. Aus einem schwarzen Pixel wird ein weißes und umgekehrt, und farbige Pixel werden in eine gut zu unterscheidende Komplementärfarbe verwandelt. Wird im XOR-Modus eine Zeichenfunktion zweimal nacheinander aufgerufen, so hat der Bildschirm nach dem zweiten Aufruf exakt dasselbe Aussehen wie vor dem ersten.

Tip

Wir rufen also zunächst `setXORMode` auf, um den Grafikkontext in den XOR-Modus zu versetzen. Mit der erwähnten Technik zeichnen wir einen rechteckigen Rahmen, der im Abstand von `drawoffset` zur Mausposition liegt und synchron mit ihr bewegt wird. Erfolgt ein neuer Aufruf von `mouseDragged`, wird der beim vorigen Aufruf gezeichnete Rahmen erneut gezeichnet und damit unsichtbar gemacht. Anschließend wird ein Rahmen an der neuen Position gezeichnet und diese Position für den nächsten Aufruf in der Variable `lastpoint` gespeichert. Zum Schluß werden die vom Grafikkontext belegten Ressourcen durch Aufruf von `dispose` zurückgegeben. Dieser Aufruf ist nur nötig, wenn der Grafikkontext durch Aufruf von `getGraphics` selbst beschafft wurde. [Abbildung 2.5](#) zeigt die Bildschirmausgabe während einer Drag-Operation.

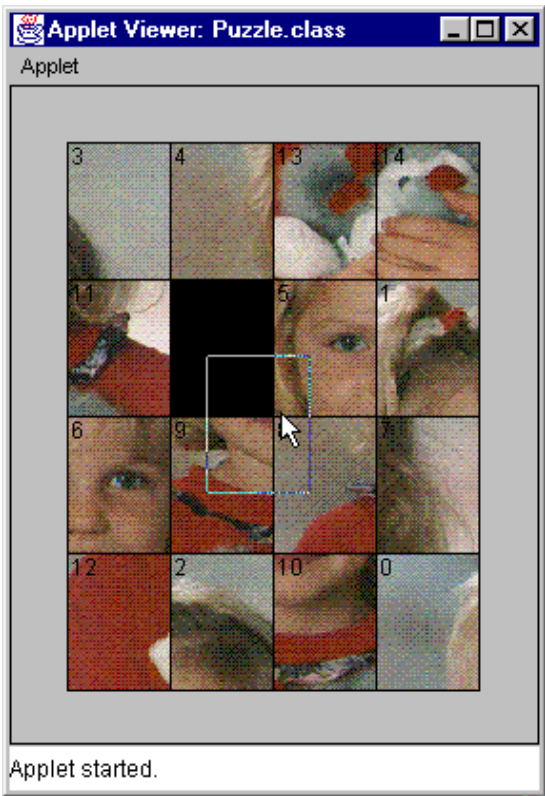


Abbildung 2.5: Die Darstellung des Drag & Drop

Weiterführende Informationen zur Ereignisbehandlung finden sich in [Kapitel 18](#) und [Kapitel 19](#). Dort werden die Grundlagen des Delegation Based Event Handling erläutert, konkrete Architekturvorschläge vorgestellt und die möglichen Ereignistypen und die zur Reaktion darauf erforderlichen Klassen, Interfaces und Methoden vorgestellt. Die Anwendung des XOR-Modus findet sich nur in diesem Beispiel, sie wird im Buch nicht weiter vertieft. Seit dem JDK 1.2 gibt es ein eigenes *Drag & Drop-API*, mit dem Daten zwischen verschiedenen Anwendungen oder mit dem Betriebssystem ausgetauscht werden können. Das API wird in diesem Buch nicht behandelt.

Hinweis

2.7 Zusammenfassung

- [2.7 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Installationsvoraussetzungen
- Installation des JDK und seiner Dokumentation
- Ein Überblick über Klassen und Methoden in Java
- Die Werkzeuge [javac](#), [appletviewer](#) und [javadoc](#)
- Initialisierung eines Applets
- Einbindung eines Applets in eine HTML-Datei
- Laden und Anzeigen von [gif](#)- und [jpeg](#)-Bildern
- Erzeugen von Zufallszahlen
- Die Methode [paint](#) und der Grafikkontext
- Einfache Zeichenmethoden
- Einsatz der Doppelpufferung zur Reduzierung des Bildschirmflackerns
- Reaktion auf Mausereignisse
- Drag & Drop und der XOR-Modus

Kapitel 3

Wie geht es weiter?

- [3 Wie geht es weiter?](#)
 - [3.1 Wie sollte man dieses Buch lesen?](#)
 - [3.2 Was ist der Inhalt der einzelnen Kapitel?](#)
 - [3.3 Wie erhält man Hilfe?](#)
 - [3.3.1 Die Dokumentation zum JDK](#)
 - [3.3.2 Weiterführende Informationen](#)
 - [Usenet](#)
 - [Meta-Ressourcen](#)
 - [FAQs](#)
 - [Online-Magazine und Dokumentationen](#)
 - [3.4 Installation des JDK](#)
 - [3.4.1 Hardware-Voraussetzungen](#)
 - [3.4.2 Installation](#)
 - [Installation des JDK](#)
 - [Installation der Dokumentation](#)
 - [Installation der Quelltexte](#)
 - [Deinstallation](#)
 - [3.5 Schnelleinstieg](#)
 - [3.5.1 Quelltext erstellen, übersetzen und ausführen](#)
 - [1. Vorbereitung](#)
 - [2. Erstellen des Quelltextes](#)
 - [3. Übersetzen des Quelltextes](#)
 - [4. Ausführen des erzeugten Programms](#)
 - [3.5.2 Die Beispielprogramme](#)
 - [Übersetzen der Beispiele](#)
 - [Quelltextformatierung](#)
 - [3.5.3 Einfache Ein-/Ausgaben](#)
 - [Einfache Ausgaben](#)
 - [Einfache Eingaben](#)
 - [3.6 Zusammenfassung](#)

3.1 Wie sollte man dieses Buch lesen?

- 3.1 Wie sollte man dieses Buch lesen?

Sie haben nun einen ersten Eindruck von Java gewonnen und wollen Ihr Wissen vervollständigen. Natürlich können Sie das Buch einfach weiterlesen und dabei Schritt für Schritt die Sprache Java und den Umgang mit ihrer umfassenden Klassenbibliothek erlernen.

Vielleicht wollen Sie aber gar nicht bis [Kapitel 25](#) warten, um zu erfahren, wie man ein Applet schreibt? Vermutlich wollen Sie auch nicht bis [Kapitel 26](#) warten, um den Umgang mit dem Debugger kennenzulernen? Auch kann es sein, daß Sie nicht an der Dateiein-/ausgabe interessiert sind und das [Kapitel 13](#) daher zunächst überspringen wollen. Je nach Ihren Vorkenntnissen und Präferenzen wird eine ganz bestimmte Vorgehensweise sinnvoll sein. Wir wollen in den nächsten Abschnitten ein paar Tips zum Lesen des Buchs und einige Hinweise zum Inhalt der einzelnen Kapitel geben.

Falls Sie zu dem Leserkreis gehören, der bereits einige Erfahrungen mit der Entwicklung von Programmen in einer konventionellen Hochsprache wie Pascal oder C hat und Sie dieses Buch vor allem lesen, um auf den Java-Zug aufzuspringen, sind Sie hier goldrichtig. Lesen Sie das Buch einfach von vorne nach hinten und lernen Sie in jedem Kapitel ein wenig dazu. Einige Dinge - insbesondere in den vorderen Kapiteln - werden Ihnen vertraut vorkommen. Aber bereits ab [Kapitel 7](#) werden Sie mit vielen Neuigkeiten konfrontiert, die den Reiz der Sprache ausmachen. Lesen Sie das Buch in aller Ruhe und nehmen Sie sich Zeit, die Beispiele zu verstehen und eigene Experimente zu machen. Sie werden am Ende des Buchs ein sicheres Verständnis für alle grundlegenden und einige weiterführende Belange der Java-Programmierung besitzen und können sich leicht in komplexere Themengebiete einarbeiten.

Falls Sie bereits weitreichende Programmiererfahrung in einer objektorientierten Sprache wie C++ oder SmallTalk haben, werden Sie sich am Anfang unterfordert fühlen. Die in [Kapitel 4](#) bis [Kapitel 6](#) eingeführten Grundlagen kommen Ihnen bekannt vor, die in [Kapitel 7](#) behandelte objektorientierte Programmierung kennen Sie im Schlaf, und die in [Kapitel 8](#) vorgestellten Techniken zur Entwicklung großer Programme sind ebenfalls nichts grundsätzlich Neues für Sie. Wenn Sie allerdings mit dem Multithreading, dem Abstract Windowing Toolkit, der Entwicklung von Animationen oder dem Exception-Konzept von Java noch nichts zu tun hatten, wird es sich auch für Sie lohnen, weiterzulesen. Auch die fortgeschrittenen Themen wie Serialisierung, Reflection, Datenbank-Anbindung oder Netzwerkprogrammierung sind sicherlich sehr interessant für Sie.

Falls Sie zu dem Leserkreis gehören, der hauptsächlich Applets entwickeln will, verspüren Sie wahrscheinlich keine Lust, viele Kapitel darauf warten zu müssen, das erste Applet vorgesetzt zu bekommen. Natürlich müssen Sie die Grundlagen der Sprache kennen, um Applets entwickeln zu können. Es spricht aber nichts dagegen, daß Sie sich bereits vor oder während des Studiums der grafikorientierten Programmierung das [Kapitel 25](#) über die Applet-Programmierung durchlesen. Die Unterschiede zwischen Applikationen und Applets halten sich in Grenzen, und sie werden genau erklärt.

Auch wenn Sie zu dem Typ Leser gehören, der Bücher nie am Stück liest, sondern nur dann zur Hand nimmt, wenn er nach einer Lösung für ein spezielles Problem sucht, kann dieses Buch für sie nützlich sein. Die Gliederung erleichtert auch das Wiederfinden spezieller Themen. Verwenden Sie einfach das Inhaltsverzeichnis oder den umfassenden Index, um das Thema zu finden, an dem Sie gerade besonders interessiert sind. Zögern Sie nicht, die Beispielprogramme in die Tat umzusetzen und eigene Experimente durchzuführen. Es gibt viele Leser, die auf diese Weise am besten lernen.

Wenn Sie dagegen überhaupt keine Programmiererfahrung haben, wird die Lektüre des Buchs nicht einfach werden. An vielen Stellen werden Grundkenntnisse in Datenstrukturen, Algorithmen und der Entwicklung von Computerprogrammen vorausgesetzt. Die Kapitel mit den fortgeschrittenen Themen setzen darüber hinaus ein gutes Verständnis der in den einführenden Kapiteln behandelten Themen voraus. Sollten Ihnen diese Kenntnisse fehlen, versuchen Sie, sie sich umgehend anzueignen. In [Abschnitt 3.3](#) finden Sie Hinweise auf weiterführende Dokumentationen und Online-Ressourcen, die Ihnen dabei helfen können.

Dieses Buch enthält an vielen Stellen Verweise auf Themen, die noch nicht behandelt wurden.

Hinweis

 Dadurch wird den Verflechtungen innerhalb der Themenbereiche Rechnung getragen, und man findet beim Nachschlagen schnell alle relevanten Textstellen wieder. Wenn Sie beim Lesen auf einen Vorwärtsverweis stoßen, können Sie normalerweise warten, bis die Erklärung nachgereicht wird (oftmals schon kurze Zeit später). Sie können den Begriff natürlich auch sofort nachschlagen, aber aus didaktischen Gründen ist es meist nicht notwendig.

3.2 Was ist der Inhalt der einzelnen Kapitel?

- [3.2 Was ist der Inhalt der einzelnen Kapitel?](#)

[Kapitel 1](#) und [Kapitel 2](#) haben Sie ja bereits gelesen, wir brauchen also nicht noch einmal darauf zurückzukommen. Auch der Inhalt von [Kapitel 3](#) ist Ihnen schon in einem gewissen Maße vertraut. Schauen wir uns also an, was die übrigen Kapitel zu bieten haben.

[Kapitel 4](#), [Kapitel 5](#) und [Kapitel 6](#) beschäftigen sich mit den elementarsten Eigenschaften der Sprache. Sie erklären die Datentypen von Java und stellen Ausdrücke und Anweisungen vor. Wenn Sie bereits mit C oder C++ vertraut sind, wird Ihnen das dort Geschriebene bekannt vorkommen, denn die Unterschiede sind in diesem Bereich gering. Dennoch gibt es einige elementare, aber wichtige Unterschiede, und auch als erfahrener C- oder C++-Programmierer sollten Sie nicht vollständig auf die Lektüre von [Kapitel 4](#) bis [Kapitel 6](#) verzichten.

[Kapitel 7](#) erklärt die objektorientierten Eigenschaften von Java und behandelt einige grundlegende Techniken der objektorientierten Programmierung. Wenn Sie bereits eine objektorientierte Sprache gut beherrschen, wird Ihnen [Kapitel 7](#) wenig Neues bieten. Was die Begriffe *Membervariable*, *Vererbung* oder *Polymorphismus* bedeuten, ist Ihnen ja längst bekannt. Kennen Sie aber auch schon das Konzept der *Interfaces*, Javas Beitrag zum Thema Mehrfachvererbung, oder wissen Sie, wie man in Java Methoden dynamisch instanziiert? Falls nicht, sollten Sie [Kapitel 7](#) und [Kapitel 31](#) nicht überschlagen. Darüber hinaus ist [Kapitel 7](#) für alle interessant, die bisher nur wenige oder gar keine Erfahrungen mit objektorientierter Programmierung gemacht haben. Sie erfahren dort, welche OOP-Features Java bietet und wie man sie sinnvoll anwendet.

Die [Kapitel 8](#), [Kapitel 9](#) und [Kapitel 10](#) sind auch für C- und C++-Profis wieder interessant. [Kapitel 8](#) erläutert das Package-Konzept von Java und leistet damit einen wichtigen Beitrag zur Strukturierung großer Programme sowie zur Verwendung von Bibliotheken von Drittanbietern. [Kapitel 9](#) behandelt den Exception-Mechanismus und erklärt, wie man in Java mit Laufzeitfehlern umgeht. In [Kapitel 10](#) schließlich lernen Sie, was es mit dem eingebauten Multithreading in Java auf sich hat und wie man dieses nützliche Feature am besten einsetzt.

Nach den Kapiteln [4](#) bis [10](#) kennen Sie alle grundlegenden Eigenschaften der Programmiersprache Java. Natürlich kommen Sie damit noch nicht sehr weit, denn die ebenso wichtigen Funktionen der Klassenbibliothek fehlen Ihnen noch. In [Kapitel 11](#) lernen Sie die String-Funktionen kennen, die zur Zeichenkettenverarbeitung unbedingt nötig sind. [Kapitel 12](#) stellt wichtige Klassen vor. Hier lernen Sie neben anderen Datenstrukturen beispielsweise die Klasse [Vector](#) kennen, mit der eine lineare Liste von Elementen beliebigen Typs realisiert werden kann. [Kapitel 13](#) schließlich stellt die Dateiein- und -ausgabe vor. Nach dem Studium der dort vorgestellten Klassen und Methoden sind Sie in der Lage, alle wichtigen Operationen auf sequentiellen und wahlfreien Dateien auszuführen. Die in diesen drei Kapiteln vorgestellten Konzepte unterscheiden sich wesentlich von den entsprechenden Konzepten in konventionellen Programmiersprachen. Selbst (oder gerade) wenn Sie ein versierter C- oder C++-Programmierer sind, werden Sie also an diesem Teil des Buches nicht vorbeikommen.

Wenn Sie bis hierhin durchgehalten haben, liegt die *Pflicht* hinter Ihnen, und Sie können sich der *Kür* zuwenden. Sie beherrschen die grundlegenden Funktionen der Sprache und ihrer Klassenbibliothek und können sich nun mit dem interessanten Thema der Programmierung unter einer grafischen Oberfläche beschäftigen. In den Kapiteln [14](#) bis [24](#) erfahren Sie alles über die Programmierung von Fenstern, Grafiken, Menüs, Dialogboxen, Bitmaps, Farben und Animationen. Sie lernen die Grundlagen der GUI-Programmierung kennen, erweitern Ihr Wissen um die Entwicklung von Benutzerschnittstellen und lernen schließlich, wie man bewegte und flackerfreie Grafiken auf den Bildschirm zaubert. Den Abschluß dieses Abschnitts bildet [Kapitel 25](#), das Sie in die Welt der Programmierung von Java-Applets einführt.

Der letzte Teil des Buchs behandelt eine Reihe weiterführender Themen. [Kapitel 29](#) beschäftigt sich mit der Performance von Java-Programmen und erklärt Techniken zur ihrer Beschleunigung. [Kapitel 27](#) erweitert die in [Kapitel 12](#) eingeführten Container-Klassen anhand des Collection-APIs des JDK 1.2. Die Kapitel [28](#) und [30](#) erweitern die Möglichkeiten, Daten persistent zu speichern. Sie erläutern das Serialisieren von Objekten und beschreiben den Zugriff auf relationale Datenbanken per JDBC. In [Kapitel 31](#) wird gezeigt, wie mit Hilfe des Reflection-APIs zur Laufzeit auf die Interna von Klassen und Objekten zugegriffen werden kann. Abgerundet wird das Buch durch die Kapitel [32](#) mit der Beschreibung der Netzwerkfähigkeiten von Java und das [Kapitel 26](#), in dem die Werkzeuge des JDK beschrieben werden.

Daß man Bücher unterschiedlich gliedern kann, ist kein Geheimnis. Daß die Beschreibung einer Programmiersprache eine andere Vorgehensweise erfordert als die eines Videospiels, ebenfalls nicht. Ist es nun besser, ein unbekanntes Entwicklungssystem aus einer abstrakten Perspektive kennenzulernen, um mit geringstem Mitteleinsatz frühzeitig sehenswerte Ergebnisse zu produzieren? Oder ist es besser, zunächst die Grundlagen der Programmiersprache zu erlernen und erst später die komplexeren Werkzeuge auf der Basis eines soliden Grundlagenwissens einzusetzen?

Wie die Kapitelaufteilung zeigt, wurde dieses Buch unter der Annahme geschrieben, daß der zweite Ansatz der sinnvollere ist. Das soll keinesfalls heißen, daß dies unter allen Umständen und für alle Leser der Fall ist. Wie oben erwähnt, mag es durchaus sinnvoll sein, zu einer anderen Vorgehensweise zu kommen. Gerade beim Einsatz im professionellen Umfeld hat man durch Termin- und Projektdruck nicht immer die Zeit, die Anwendung neuer Werkzeuge mit der nötigen Ruhe zu erlernen.

Letztendlich müssen Sie selbst entscheiden, welche der Kapitel Sie durchlesen und in welcher Reihenfolge Sie dies tun. Wählen Sie die Technik, die Ihren Neigungen und Erwartungen am meisten entgegenkommt und am besten mit den Beweggründen, aus denen heraus Sie dieses Buch lesen, in Einklang zu bringen ist. Dann werden Sie den größtmöglichen Nutzen aus diesem Buch ziehen.

3.3 Wie erhält man Hilfe?

- 3.3 Wie erhält man Hilfe?
 - 3.3.1 Die Dokumentation zum JDK
 - 3.3.2 Weiterführende Informationen
 - Usenet
 - Meta-Ressourcen
 - FAQs
 - Online-Magazine und Dokumentationen

3.3.1 Die Dokumentation zum JDK

Die Dokumentation zum JDK befindet sich auf der beigefügten CD-ROM. Sie liegt im Verzeichnis `\install\jdk12` und kann wie in [Abschnitt 3.4](#) beschrieben installiert werden. Zusätzlich befinden sich diverse weiterführende Informationen, Dokumentationen und Spezifikationen zu verschiedenen Aspekten der Java-Programmierung im Verzeichnis `\java` der CD-ROM. Es empfiehlt sich, die Datei `\readme.txt` zu lesen, um einen Überblick über den Inhalt der CD-ROM zu bekommen.

3.3.2 Weiterführende Informationen

Java ist die Sprache des Internet, und folglich gibt es unzählige Ressourcen im Internet, die sich in der einen oder anderen Weise mit Java beschäftigen. Leider veralten viele der Adressen fast ebenso schnell, wie sie erschienen sind, und ein Buch ist daher nur bedingt geeignet, sie aufzuzählen. Wir wollen uns daher auf einige der wichtigsten Adressen beschränken, die bei der Entwicklung von Java-Programmen nützlich sein können.

Usenet

Die offiziellen Usenet-Newsgroups zu Java beginnen mit dem Namen [comp.lang.java](#). Hier gibt es eine ganze Reihe von Untergruppen zu speziellen Themen (siehe [Tabelle 3.1](#)). Leider ist die Abgrenzung zwischen den einzelnen Untergruppen nicht immer klar, und es kommt regelmäßig zu Überschneidungen. [Tabelle 3.1](#) listet auch die veralteten Gruppen auf, weil immer noch Nachrichten in ihnen erscheinen.

Newsgroup	Inhalt	Nachrichten pro Tag
news:comp.lang.java.advocacy	Allgemeine Diskussionen über Java.	ca. 60
news:comp.lang.java.announce	Moderierte Newsgroup mit Ankündigungen und Vorstellungen von Neuentwicklungen.	< 1
news:comp.lang.java.api	Das Application Programming Interface und die Klassenbibliothek. Die Gruppe ist veraltet und sollte nicht mehr verwendet werden.	ca. 1
news:comp.lang.java.beans	Die Komponentenarchitektur Beans.	ca. 5-10
news:comp.lang.java.corba	Java, CORBA und Objektverteilung im Netz.	ca. 5
news:comp.lang.java.databases	Datenbankprogrammierung mit JDBC.	ca. 15
news:comp.lang.java.gui	Programmierung von grafischen Oberflächen und Diskussion von GUI-Buildern.	ca. 30
news:comp.lang.java.help	Allgemeine Quelle für Fragen aller Art, von der Installation bis zu Programmierproblemen.	ca. 60
news:comp.lang.java.machine	Diskussionen um VMs und alles, was sich unterhalb der Sprachebene abspielt. Ersetzt die Gruppe <i>comp.lang.java.tech</i> .	ca. 3

news:comp.lang.java.misc	Veraltete Gruppe mit Diskussionen zu unterschiedlichen Themen. Sollte eigentlich nicht mehr verwendet werden.	ca. 5
news:comp.lang.java.programmer	Stark frequentierte Newsgroup zu allen möglichen Aspekten der Java-Programmierung.	ca. 160
news:comp.lang.java.security	Diskussion von Sicherheitsaspekten.	ca. 5
news:comp.lang.java.setup	Diskussion von Installationsaspekten. Ist veraltet und sollte durch <i>comp.lang.java.help</i> ersetzt werden.	< 1
news:comp.lang.java.softwaretools	Diskussionen zu Tools, Werkzeugen und Entwicklungsumgebungen rund um Java.	ca. 10-15
news:comp.lang.java.tech	Veraltete Gruppe zu technischen Fragestellungen. Wurde durch news:comp.lang.java.machine ersetzt.	ca. 1-2
news:comp.lang.javascript	Hier dreht sich alles um die Script-Sprache JavaScript. Diese Gruppe hat daher keinen direkten Bezug zu Java, soll aber der Vollständigkeit halber erwähnt werden.	Nicht ermittelt
news:de.comp.lang.java	Es gibt auch eine deutsche Newsgroup, in der alle Aspekte von Java diskutiert werden.	ca. 35

Tabelle 3.1: Die *comp.lang.java*-Hierarchie im Usenet

Meta-Ressourcen

Unter <http://java.sun.com/> oder <http://www.javasoft.com/> finden Sie den Java-Server von Sun bzw. Suns Tochter JavaSoft. Hier sind Informationen aus erster Hand von den Entwicklern der Sprache zu finden. Dieser Server ist die erste Adresse, wenn es um Neuigkeiten, aktuelle Entwicklungen und Dokumentationen geht. Ein direkter Link auf die von Sun für Java zur Verfügung gestellten Entwicklungsumgebungen ist <http://java.sun.com/products/>. Unter der Adresse <http://java.sun.com/products/jdk/1.2/> gibt es alles rund um die aktuelle Java-Version 1.2.

Eine wichtige Adresse für Entwickler ist auch die der *Java Developers Connection (JDC)* unter <http://java.sun.com/jdc>. Diese Seiten werden von SUN gepflegt, um eine zentrale Anlaufstelle für Java-Entwickler zur Verfügung zu stellen. Es gibt dort Diskussionsforen, Schulungsangebote, weitere Software und jede Menge nützliche Informationen. Wichtiges »Organ« der JDC ist der *JDC-Newsletter*. Dabei handelt es sich um einen Newsletter, der per E-Mail regelmäßig über aktuelle Neuerungen informiert. Der Zutritt zur JDC ist kostenlos, erfordert aber das Ausfüllen einer Registrierungsseite.

Auch in Yahoo gibt es eine eigene Rubrik für die Programmiersprache Java. Unter der Adresse http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Java findet sich eine Vielzahl von Links zu Java-spezifischen Themen.

Eines der größten Java-Verzeichnisse im Internet nennt sich *Gamelan* und ist unter der Adresse <http://www.gamelan.com/> zu finden. Hier werden in einer ganzen Reihe von Rubriken über zehntausend Applets und Applikationen beschrieben. Gamelan bietet diverse Suchmöglichkeiten, und es gibt Listen der neuesten und interessantesten Applets.

Vergleichbar mit Gamelan ist *JARS*, der »Java Applet Rating Service«. Er befindet sich unter der Adresse <http://www.jars.com/> und bietet ebenfalls eine Vielzahl von Verweisen auf Java-Ressourcen im Internet.

JavaLobby ist ein Zusammenschluß von Java-Enthusiasten, der das Ziel verfolgt, die Sprache zu verbreiten und für ein »100 % Pure Java« einzutreten. Die Homepage unter <http://www.javalobby.org/> bietet auch eine ganze Menge Verweise zu Java-Ressourcen und interessante Artikel rund um Java.

Unter der Adresse <http://www.apl.jhu.edu/~hall/java/> verwaltet Marty Hall von der Johns Hopkins University eine umfassende Liste von Java-Ressourcen mit Links zu FAQs, weiteren Dokumentationen, Beispielanwendungen, Entwicklungsumgebungen, Klassenbibliotheken und vielem anderen mehr.

FAQs

Die FAQs (Frequently Asked Questions) der Newsgroup comp.lang.java stammen von Elliott Rusty Harold und sind unter der Adresse <http://sunsite.unc.edu/javafaq/javafaq.html> zu finden. Sie enthalten zahlreiche Hinweise zur Sprache, der Klassenbibliothek und zu den Java-Werkzeugen.

Von Sun selbst gibt es ebenfalls ein FAQ, das unter <http://www.javasoft.com/products/jdk/faq.html> zu finden ist. Dort sind auch einige Metainformationen und firmenbezogene Informationen über Java zu finden.

Auch Peter van der Linden bietet ein FAQ-Dokument an, das unter der Adresse <http://www.best.com/~pvd/Javafaq.txt> gefunden werden kann.

Online-Magazine und Dokumentationen

Unter <http://www.sys-con.com/java/index2.html> ist die Online-Version des *Java Developer's Journal* zu finden. Unter <http://www.javaworld.com/javasoft/index.html> findet sich die *Java World*, und die Adresse für den *Java Report Online* ist <http://www.javareport.com/>.

Auf dem SUN-Server gibt es weitere Dokumentationen zu Java. Unter <http://java.sun.com/docs/books/jls/index.html> ist die Sprachspezifikation zu finden, und die Spezifikation der virtuellen Maschine findet sich unter <http://java.sun.com/docs/books/vmspec/index.html> Unter <http://java.sun.com/docs/books/tutorial/index.html> findet sich ein Java-Tutorial von Sun. Einen Überblick über die insgesamt bei SUN verfügbare Java-Dokumentation gibt es unter der Adresse <http://java.sun.com/docs/index.html>.

3.4 Installation des JDK

- [3.4 Installation des JDK](#)
 - [3.4.1 Hardware-Voraussetzungen](#)
 - [3.4.2 Installation](#)
 - [Installation des JDK](#)
 - [Installation der Dokumentation](#)
 - [Installation der Quelltexte](#)
 - [Deinstallation](#)

3.4.1 Hardware-Voraussetzungen

Zur Installation des JDK ist ein vernünftig ausgestatteter PC oder eine Solaris-Workstation erforderlich. Alternativ kann auch eine der vielen anderen Plattformen verwendet werden, auf die das JDK portiert wurde, beispielsweise OS/2, Mac-OS, Linux und viele andere Unix-Derivate. Dieses Buch und die Beispiele darin wurden auf verschiedenen Rechnern unter Windows 95, 98, NT und unter LINUX entwickelt. Als eben noch brauchbares Minimalsystem kann folgende Konfiguration angesehen werden:

- Pentium-133
- 48 MB RAM
- Grafik mit 800 * 600 Pixeln, 8 Bit Farbtiefe

Diese Ausstattung liegt etwas über den Mindestanforderungen und erlaubt ein einigermaßen flüssiges Arbeiten. 64 oder 128 MB RAM und ein Pentium-II machen die Arbeit deutlich angenehmer. Soll viel mit den AWT- oder JFC-Klassen gearbeitet werden, ist eine bessere Grafikausstattung empfehlenswert. Der Compiler von Sun ist etwas langsam, denn er ist selbst in Java geschrieben. Auch auf einem schnelleren Rechner wird er nicht zum »Renner«. Sollen integrierte Entwicklungssysteme anderer Hersteller verwendet werden, liegen die Hardwareanforderungen meist ebenfalls deutlich über der oben angegebenen Konfiguration.

Die Installation des JDK 1.1 erfordert inklusive Dokumentation etwa 30 MB Plattenspeicher, die des JDK 1.2 etwa 150 MB. Zu beachten ist dabei, daß insbesondere nach Installation der Quelltexte und der Dokumentation sehr viele kleine Dateien vorhanden sind. Auf einem FAT-Dateisystem mit großer Clustergröße kann also noch erheblich mehr Plattenplatz verbraucht werden.

3.4.2 Installation

Wir wollen hier nur die Installation unter Windows 95 beschreiben, die mit Hilfe eines InstallShield-Skripts menügesteuert erfolgt. Anschließend muß die Dokumentation des JDK installiert werden. Dies geschieht per Hand und erfordert ein Programm zum Entpacken der HTML-Dateien.

Installation des JDK

Das JDK 1.2 befindet sich auf der CD-ROM im Unterverzeichnis `\install\jdk12`. Die Installation unter Windows ist sehr einfach und kann in folgenden Schritten erfolgen:

- Zunächst muß in das Verzeichnis `\install\jdk12` der CD-ROM gewechselt werden.
- Nun wird das Installationsprogramm `jdk12-win32.exe` gestartet.
- Nach Aufruf des InstallShield-Wizards wird mit »Next« die nächste Seite aufgerufen, um die Lizenzbedingungen anzuzeigen. Diese sollten bis zum Ende durchgelesen werden und die Installation ggfs. mit »Yes« fortgeführt werden.
- Nun kann das Installationsverzeichnis ausgewählt werden. Es empfiehlt sich, die Voreinstellung `\jdk1.2` zu akzeptieren und mit »Next« fortzufahren.
- Die nachfolgende Auswahl der Komponenten erlaubt eine Konfiguration der Installation. Anfangs empfiehlt es sich, alle Komponenten zu installieren, um auch die Beispielprogramme und den Quelltext einsehen zu können. Dazu werden etwa 60 MB Plattenspeicher benötigt. Nach Betätigen des »Next«-Buttons werden die ausgewählten Komponenten installiert.
- Nach erfolgter Installation werden die Lizenzbedingungen für das Java-Runtime-Environment angezeigt. Sie sollten ebenfalls durch Drücken des »Yes«-Buttons akzeptiert werden.
- Nun kann das Installationsverzeichnis ausgewählt werden. Es empfiehlt sich, die Voreinstellung `\programme\javasoft\jre\1.2` zu akzeptieren und mit »Next« fortzufahren.
- Nach erfolgter Installation kann die README-Datei gelesen und mit Hilfe des »Finish«-Buttons das Installationsprogramm beendet werden.

Alle Dateien befinden sich nun im ausgewählten Installationsverzeichnis, und die Installation des JDK ist abgeschlossen. Um mit dem JDK arbeiten zu können, muß noch das Verzeichnis `\jdk1.2\bin` in den Suchpfad für ausführbare Dateien eingetragen werden. Das kann direkt in der

`autoexec.bat` durch Modifikation des PATH-Statements erfolgen oder später mit Hilfe einer Batchdatei erledigt werden:

```
PATH=c:\jdk1.2\BIN;%PATH%
```

Anders als in den Vorgängerversionen benötigen die Werkzeuge des JDK bei einer Standardinstallation unter Windows 95 keine Umgebungsvariable `CLASSPATH` mehr, denn die erforderlichen Informationen werden bei der Installation in die Registry geschrieben (sie liegen in unterschiedlichen Abschnitten, eine Suche nach »javasoft« hilft weiter). Ist jedoch eine `CLASSPATH`-Variable vorhanden, so wird sie auch verwendet. Wird das JDK 1.2 also über eine ältere Version installiert, muß dafür gesorgt werden, daß eine eventuell gesetzte `CLASSPATH`-Variable modifiziert oder entfernt wird. Weitere Informationen zum Setzen der `CLASSPATH`-Umgebungsvariable finden sich ab [Abschnitt 8.2.2](#).

Hinweis

Installation der Dokumentation

Die Dokumentation des JDK besteht aus einer Sammlung von HTML-Dateien, die auf der CD-ROM im Verzeichnis `\install\jdk12` zu finden sind. Um sie zu installieren, muß die Datei `jdk12-doc.zip` ausgepackt werden. Da lange Dateinamen darin enthalten sind, muß zum Auspacken ein geeigneter Entpacker verwendet werden. Auf der CD-ROM befindet sich im Verzeichnis `\misc` eine Version von *WinZip95*, die dazu verwendet werden kann. Bitte beachten Sie vor der Verwendung dieses Programms die Lizenz- und Registrierungshinweise.

Die Dokumentation wurde bereits in der Verzeichnisstruktur `\jdk1.2\docs` verpackt. Falls das Installationsverzeichnis nicht geändert wurde, können die Dateien ohne weitere Änderungen im Root-Verzeichnis ausgepackt werden. Alle erforderlichen Unterverzeichnisse werden dann automatisch angelegt. Zur Anzeige der Dokumentation ist ein Browser wie Netscape Navigator oder Microsoft Internet Explorer erforderlich. Um leichter auf die Dokumentation zugreifen zu können, ist es sinnvoll, innerhalb des Browsers einen Verweis (Lesezeichen) auf das Hauptdokument `\jdk1.2\docs\index.html` anzulegen. Diese Datei kann zur Navigation auf alle Teile der Dokumentation verwendet werden und enthält zusätzlich eine Reihe von Verweisen auf externe Dokumente. Alternativ kann ein Icon auf dem Desktop angelegt werden, das den Browser mit dem Namen dieser Datei als Argument aufruft (die korrekte Schreibweise ist `file:///C:/jdk1.2/docs/index.html`).

Installation der Quelltexte

Das JDK 1.2 wird mit den vollständigen Java-Quelltexten ausgeliefert. Nicht enthalten sind dagegen die Quelltexte der Native Methods. Nach der Installation des JDK liegen die Quelltexte im Archiv `src.jar` im Installationsverzeichnis. Sie sind zur Arbeit mit dem JDK nicht unbedingt nötig, können aber hilfreich sein, um weitergehende Fragen zu beantworten. Die Datei `src.jar` kann beispielsweise mit *WinZip* geöffnet und ihr Inhalt nach `\jdk1.2` ausgepackt werden. Dadurch wird ein Unterverzeichnis `\jdk1.2\src` angelegt, in dem die Quelltexte zu finden sind. Alternativ kann natürlich auch das JDK-Werkzeug `jar` verwendet werden, dessen Bedienung in [Abschnitt 26.2.2](#) erklärt wird.

Deinstallation

Die Deinstallation ist denkbar einfach und kann mit Hilfe der Systemsteuerung erledigt werden. Der zu entfernende Eintrag hat die Bezeichnung »Java Development Kit 1.2«. Bis auf die separat installierte Dokumentation und einige kleinere Dateien entfernt die Deinstallationsroutine die Bestandteile des JDK vollständig vom Rechner.

3.5 Schnelleinstieg

- [3.5 Schnelleinstieg](#)
 - [3.5.1 Quelltext erstellen, übersetzen und ausführen](#)
 - [1. Vorbereitung](#)
 - [2. Erstellen des Quelltextes](#)
 - [3. Übersetzen des Quelltextes](#)
 - [4. Ausführen des erzeugten Programms](#)
 - [3.5.2 Die Beispielprogramme](#)
 - [Übersetzen der Beispiele](#)
 - [Quelltextformatierung](#)
 - [3.5.3 Einfache Ein-/Ausgaben](#)
 - [Einfache Ausgaben](#)
 - [Einfache Eingaben](#)

Falls Sie die ersten Gehversuche in Java machen wollen, ohne erst viele Grundlagen lernen zu müssen, oder wenn Sie einfach nur daran interessiert sind, möglichst schnell Ihr erstes Java-Programm auszuführen, dann sollten Sie diesen Abschnitt lesen. Sie erfahren hier in einer kurzen Anleitung, wie Sie ein einfaches Programm erstellen und mit den Werkzeugen des JDK übersetzen und ausführen. Zusätzlich gibt es einige Hinweise, um zeilenorientierte Ein- und Ausgaben durchzuführen.

3.5.1 Quelltext erstellen, übersetzen und ausführen

1. Vorbereitung

Installieren Sie das JDK wie in [Abschnitt 3.4](#) beschrieben und sorgen Sie dafür, daß in Ihrer Pfadangabe das Verzeichnis `\jdk1.2\bin` enthalten ist. Falls Sie das JDK nicht nach `c:\jdk1.2` installiert haben, passen Sie die Pfadangaben entsprechend an.

2. Erstellen des Quelltextes

Erstellen Sie mit einem beliebigen Texteditor die folgende Datei `Hello.java`:

[Hello.java](#)

```
001 /* Hello.java */
002
003 public class Hello
004 {
005     public static void main(String args[])
006     {
007         System.out.println("Hello, world");
008     }
009 }
```

Listing 3.1: Hello, world

Sie können dazu beispielsweise `notepad` unter Windows oder `vi` oder `emacs` unter UNIX verwenden. Die erstellte Datei enthält die Definition der Klasse `Hello`. Sie enthält lediglich eine einzige Methode `main`, die das Hauptprogramm unserer Applikation enthält.

Achten Sie bei der Vergabe der Datei- und Klassennamen auf korrekte Groß-/Kleinschreibung. Sowohl der Compiler als auch das Laufzeitsystem erwarten, daß die Hauptklasse einer Quelldatei exakt so geschrieben wird, wie der Name der Datei, in der sie sich befindet. Der Dateiname unserer Beispielklasse `Hello` lautet daher auch `Hello.java`, und nicht `hello.java` oder `HELLO.JAVA`. Erstellen Sie die Datei nicht auf Systemen oder Laufwerken (beispielsweise im Netz), die keine langen Dateinamen unterstützen. Weder der Compiler noch der Interpreter würde in diesem Fall die Klasse finden.

Warnung

3. Übersetzen des Quelltextes

Übersetzen Sie die Datei mit dem Kommando `javac` (so heißt der Java-Compiler des JDK):

```
javac Hello.java
```

Alle wichtigen Werkzeuge des JDK arbeiten kommandozeilenorientiert. Sie haben also keine grafische Oberfläche, sondern werden in einer DOS-Box aufgerufen und durch Aufrufparameter gesteuert. Eine integrierte Entwicklungsumgebung mit integriertem Editor, Compiler und Debugger bietet das JDK nicht. Eine Übersicht über die wichtigsten Werkzeuge und ihre Bedienung finden Sie in [Kapitel 26](#).

Hinweis

4. Ausführen des erzeugten Programms

Sie haben nun eine Datei `Hello.class` erzeugt, die mit dem Java-Interpreter ausgeführt werden kann:

```
java Hello
```

Das Programm gibt die gewünschte Meldung auf dem Bildschirm aus:

```
Hello, world
```

3.5.2 Die Beispielprogramme

Übersetzen der Beispiele

Auf die im vorigen Abschnitt beschriebene Weise können nun beliebige Java-Programme angelegt, übersetzt und ausgeführt werden. Die im Buch abgedruckten Beispielprogramme befinden sich auf der CD-ROM im Verzeichnis `\examples`. Kopieren Sie diese einschließlich der darin enthaltenen Unterverzeichnisse in ein beliebiges Verzeichnis auf Ihrer Festplatte. Benutzen Sie einen beliebigen Editor zur Eingabe oder Veränderung von `.java`-Dateien, übersetzen Sie die Datei mit dem Kommando `javac`, und starten Sie das fertige Programm mit dem Kommando `java`. Falls Sie ein Applet geschrieben haben, erstellen Sie zusätzlich eine passende HTML-Datei, und starten Sie das Programm mit dem Kommando `appletviewer` anstatt mit `java`.

Als Entwicklungssystem für dieses Buch wurden hauptsächlich die Versionen 1.1.2, 1.2 Beta 4 und 1.2 RC 2 des JDK unter Windows 95 verwendet. Die meisten Beispiele wurden auf diesem System entwickelt und getestet. Einige Beispiele wurden auch unter Windows 98, NT oder S.U.S.E Linux 5.2 entwickelt und getestet. In die Beispiellistings aus dem AWT sind einige Hinweise von Lesern mit SUN-Solaris-Plattformen eingeflossen. Keines der Programme wurde vom Autor auf einem Macintosh getestet (mangels Verfügbarkeit eines solchen). Bei Verwendung unterschiedlicher Plattformen könnte es theoretisch zu leichten Abweichungen bei der Installation, der Funktionalität der Entwicklungswerkzeuge oder den Eigenschaften der Standardbibliothek kommen. Diese sind aber in der tatsächlichen Praxis des Autors nicht aufgetreten.

Hinweis

Quelltextformatierung

Es ist bekannt, daß man sich über die Formatierung von Quelltexten und die Einrückung von Deklarationen und Anweisungen streiten kann. Jeder Entwickler hat seinen eigenen Stil und kennt gute Argumente, genau diesen zu verwenden. Mittlerweile gibt es einige große Lager, denen man sich anschließen kann, oder man ist gezwungen, seinen eigenen Stil einem vorgegebenen Style-Guide anzupassen. Wir wollen uns diesen fruchtlosen Diskussionen nicht anschließen und keinesfalls behaupten, die in diesem Buch verwendete Art, Sourcecode zu formatieren, wäre die einzig richtige. Dennoch wurde versucht, die Beispielprogramme einigermaßen konsistent zu formatieren und dabei einige wenige Regeln einzuhalten.

Bei Klassen- und Methodendefinitionen stehen die geschweiften Klammern unterhalb der Deklarationsanweisung, und die eigentliche Deklaration ist eingerückt:

```
001 import java.io.*;
002
003 public class Listing0302
004 {
005     public static void main(String args[])
006     {
007         //Hier steht der Methodenrumpf
008     }
009 }
```

Listing 3.2: Einrücken von Klassen und Methoden

Bei Kontrollanweisungen innerhalb einer Methode schreiben wir die öffnende Klammer dagegen in dieselbe Zeile wie die einleitende Anweisung:

```

001 for (int i = 0; i < aNeighbours.length; ++i) {
002     if (p1.x + aNeighbours[i][0] == p2.x) {
003         if (p1.y + aNeighbours[i][1] == p2.y) {
004             return true;
005         }
006     }
007 }

```

Listing 3.3: Einrücken von Kontrollanweisungen

Dies gilt auch für fortgesetzte Anweisungen wie beispielsweise `else` oder `else if`:

```

001 if (cmd.equals("Größer")) {
002     d.height *= 1.05;
003     d.width  *= 1.05;
004 } else if (cmd.equals("Kleiner")) {
005     d.height *= 0.95;
006     d.width  *= 0.95;
007 } else {
008     x = 10;
009 }

```

Listing 3.4: Einrücken fortgesetzter Anweisungen

Diese Technik verwenden wir meist auch, wenn bei einem Methodenaufruf nicht alle Argumente in eine Zeile passen:

```

001 System.out.println(
002     "Grüße aus Hamburg".regionMatches(
003         8,
004         "Greetings from Australia",
005         8,
006         2
007     )
008 );

```

Listing 3.5: Einrücken langer Methodenaufrufe

Diese einfachen Regeln lassen sich in den meisten Fällen anwenden, es gibt aber auch Fälle, in denen sie versagen. So zum Beispiel, wenn der Testausdruck einer `if`-Anweisung über mehrere Zeilen geht, wenn die Parameterdeklaration einer Methode nicht in eine Zeile paßt oder schlicht, wenn die Verschachtelung bereits sehr tief ist und keine weitere Einrückung zuläßt. In diesem Fall sei der Leser um Nachsicht gebeten und aufgefordert, den ästhetischen Anspruch an das Programm den jeweiligen pragmatischen Erwägungen unterzuordnen.

3.5.3 Einfache Ein-/Ausgaben

Für die ersten Schritte in einer neuen Sprache benötigt man immer auch I/O-Routinen, um einfache Ein- und Ausgaben vornehmen zu können. Glücklicherweise kann man in Java nicht nur grafikorientierte Programme schreiben, sondern auch auf die Standardein- und -ausgabe zugreifen. Damit stehen für kleine Programme einfache I/O-Routinen zur Verfügung, die wie in den meisten konventionellen Programmiersprachen verwendet werden können.

Einfache Ausgaben

Mit Hilfe des Kommandos `System.out.println` können einfache Ausgaben auf den Bildschirm geschrieben werden.

`System.out.println` erwartet als einziges Argument eine Zeichenkette. Mit Hilfe des Plus-Operators können Zeichenketten und numerische Argumente verknüpft werden, so daß man neben Text auch Zahlen ausgeben kann:

[Listing0306.java](#)

```

001 /* Listing0306.java */
002
003 public class Listing0306
004 {
005     public static void main(String args[])
006     {
007         System.out.println("1+2=" + (1+2));
008     }
009 }

```

Listing 3.6: Einfache Ausgaben

Die Ausgabe des Programms ist:

1+2=3

Einfache Eingaben

Leider ist es etwas komplizierter, Daten zeichenweise von der Tastatur zu lesen. Zwar steht ein vordefinierter Eingabe-Stream `System.in` zur Verfügung. Er ist aber nicht in der Lage, die eingelesenen Zeichen in primitive Datentypen zu konvertieren. Statt dessen muß zunächst eine Instanz der Klasse `InputStreamReader` und daraus ein `BufferedReader` erzeugt werden. Dieser kann dann dazu verwendet werden, die Eingabe zeilenweise zu lesen und das Ergebnis in einen primitiven Typ umzuwandeln. Weitere Information zur streambasierten Ein-/Ausgabe sind in [Kapitel 13](#) zu finden.

Das folgende Listing zeigt dies am Beispiel eines Programms, das zwei Ganzzahlen einliest, sie zusammenzählt und das Ergebnis auf dem Bildschirm ausgibt: Beispiel

[Listing0307.java](#)

```
001 /* Listing0307.java */
002 import java.io.*;
003
004 public class Listing0307
005 {
006     public static void main(String args[])
007     throws IOException
008     {
009         int a, b, c;
010         BufferedReader din = new BufferedReader(
011             new InputStreamReader(System.in));
012
013         System.out.println("Bitte a eingeben: ");
014         a = Integer.parseInt(din.readLine());
015         System.out.println("Bitte b eingeben: ");
016         b = Integer.parseInt(din.readLine());
017         c = a + b;
018         System.out.println("a+b="+c);
019     }
020 }
```

Listing 3.7: Einfache Eingaben

Werden die Zahlen 10 und 20 eingegeben, so lautet die Ausgabe des Programms:

Bitte a eingeben:
10
Bitte b eingeben:
20
a+b=30

Das Ergebnis von `din.readLine` ist ein `String`, der den Inhalt der Eingabezeile enthält . Sollen keine numerischen Datenwerte, sondern Zeichenketten eingelesen werden, so kann der Rückgabewert auch direkt verwendet werden.

Tip

3.6 Zusammenfassung

- [3.6 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Hinweise zum Lesen des Buchs
- Übersicht über den Inhalt der einzelnen Kapitel
- Installation des JDK und seiner Dokumentation, sowie der mitgelieferten Quelltexte
- Weiterführende Informationen
- Java-Ressourcen im Internet
- Ein Schnelleinstieg
- Übersetzen der Beispielprogramme
- Hinweise zur Formatierung des Quellcodes
- Einfache Ein-/Ausgaben

Kapitel 4

Datentypen

- [4 Datentypen](#)
 - [4.1 Lexikalische Elemente eines Java-Programms](#)
 - [4.1.1 Eingabezeichen](#)
 - [4.1.2 Kommentare](#)
 - [4.1.3 Bezeichner](#)
 - [4.1.4 Weitere Unterschiede zu C](#)
 - [4.2 Primitive Datentypen](#)
 - [4.2.1 Der logische Typ](#)
 - [Literele](#)
 - [4.2.2 Der Zeichentyp](#)
 - [Literele](#)
 - [4.2.3 Die integralen Typen](#)
 - [Literele](#)
 - [4.2.4 Die Fließkommazahlen](#)
 - [Literele](#)
 - [4.3 Variablen](#)
 - [4.3.1 Grundeigenschaften](#)
 - [4.3.2 Deklaration von Variablen](#)
 - [4.3.3 Lebensdauer/Sichtbarkeit](#)
 - [4.4 Arrays](#)
 - [4.4.1 Deklaration und Initialisierung](#)
 - [4.4.2 Zugriff auf Array-Elemente](#)
 - [4.4.3 Mehrdimensionale Arrays](#)
 - [4.5 Referenztypen](#)
 - [4.5.1 Beschreibung](#)
 - [4.5.2 Speichermanagement](#)
 - [4.6 Typkonvertierungen](#)
 - [4.7 Zusammenfassung](#)

4.1 Lexikalische Elemente eines Java-Programms

- 4.1 Lexikalische Elemente eines Java-Programms
 - 4.1.1 Eingabezeichen
 - 4.1.2 Kommentare
 - 4.1.3 Bezeichner
 - 4.1.4 Weitere Unterschiede zu C

Bevor wir uns in diesem Kapitel mit den Datentypen von Java befassen, sollen zunächst einmal die wichtigsten lexikalischen Eigenschaften der Sprache vorgestellt werden. Hierzu zählen der Eingabezeichensatz, die Kommentare und die Struktur von Bezeichnern.

4.1.1 Eingabezeichen

Ein Java-Programm besteht aus einer Folge von Unicode-Zeichen. Der Unicode-Zeichensatz faßt eine große Zahl internationaler Zeichensätze zusammen und integriert sie in einem einheitlichen Darstellungsmodell. Da die 256 verfügbaren Zeichen eines 8-Bit-Wortes bei weitem nicht ausreichen, um die über 30.000 unterschiedlichen Zeichen des Unicode-Zeichensatzes darzustellen, ist ein Unicode-Zeichen 2 Byte, also 16 Bit, lang. Der Unicode ist mit den ersten 128 Zeichen des ASCII- und mit den ersten 256 Zeichen des ISO-8859-1-Zeichensatzes kompatibel.

Die Integration des Unicode-Zeichensatzes geht in Java so weit, daß neben `String`- und `char`-Typen auch die literalen Symbole und Bezeichner der Programmiersprache im Unicode realisiert sind. Es ist daher ohne weiteres möglich, Variablen- oder Klassennamen mit nationalen Sonderzeichen oder anderen Symbolen zu versehen.

Hinweis

4.1.2 Kommentare

Es gibt in Java drei Arten von Kommentaren:

- Einzeilige Kommentare* beginnen mit `//` und enden am Ende der aktuellen Zeile.
- Mehrzeilige Kommentare* beginnen mit `/*` und enden mit `*/`. Sie können sich über mehrere Zeilen erstrecken.
- Dokumentationskommentare* beginnen mit `/**` und enden mit `*/` und können sich ebenfalls über mehrere Zeilen erstrecken.

Kommentare derselben Art sind nicht schachtelbar. Ein Java-Compiler akzeptiert aber einen einzeiligen innerhalb eines mehrzeiligen Kommentars und umgekehrt.

Dokumentationskommentare dienen dazu, Programme im Quelltext zu dokumentieren. Mit Hilfe des Tools `javadoc` werden sie aus der Quelle extrahiert und in ein HTML-Dokument umgewandelt (siehe [Kapitel 26](#)). Kapitel 18 der Sprachspezifikation erklärt die Verwendung von Dokumentationskommentaren ausführlich. Wir wollen uns hier lediglich auf ein kleines Beispiel beschränken, das besagter Beschreibung entnommen wurde:

```

001 /**
002  * Compares two Objects for equality.
003  * Returns a boolean that indicates whether this Object
004  * is equivalent to the specified Object. This method is
005  * used when an Object is stored in a hashtable.
006  * @param  obj      the Object to compare with
007  * @return          true if these Objects are equal;
008  *                 false otherwise.
009  * @see            java.util.Hashtable
010  */
011 public boolean equals(Object obj) {
012     return (this == obj);
013 }
    
```

Listing 4.1: Verwendung eines Dokumentationskommentars im Java-API

Dokumentationskommentare stehen immer *vor* dem Element, das sie beschreiben sollen. In diesem Fall ist das die Methode `equals` der Klasse `Object`. Der erste Satz ist eine Überschrift, dann folgt eine längere Beschreibung der Funktionsweise. Die durch `@` eingeleiteten Elemente sind Makros, die eine besondere Bedeutung haben. `@param` spezifiziert Methodenparameter, `@return` den Rückgabewert und `@see` einen Verweis. Daneben gibt es noch die Makros `@exception`, `@version` und `@author`, die hier aber nicht auftauchen.

4.1.3 Bezeichner

Ein Bezeichner ist eine Sequenz von Zeichen, die dazu dient, die Namen von Variablen, Klassen oder Methoden zu spezifizieren. Ein Bezeichner in Java kann beliebig lang sein, und alle Stellen sind signifikant. Bezeichner müssen mit einem *Unicode-Buchstaben* beginnen (das sind die Zeichen 'A' bis 'Z', 'a' bis 'z', '_' und '\$') und dürfen dann weitere Buchstaben oder Ziffern enthalten. Unterstrich und Dollarzeichen sollen nur aus historischen Gründen bzw. bei maschinell generiertem Java-Code verwendet werden.

Ein Buchstabe im Sinne des Unicode-Zeichensatzes muß nicht zwangsläufig aus dem lateinischen Alphabet stammen. Es ist auch zulässig, Buchstaben aus anderen Landessprachen zu verwenden. Java-Programme können daher ohne weiteres Bezeichner enthalten, die nationalen Konventionen folgen. Java-Bezeichner dürfen jedoch nicht mit Schlüsselwörtern, den booleschen Literalen [true](#) und [false](#) oder dem Literal [null](#) kollidieren.

Warnung

4.1.4 Weitere Unterschiede zu C

Nachfolgend seien noch einige weitere Unterschiede zu C und C++ aufgelistet, die auf der lexikalischen Ebene von Bedeutung sind:

- Es gibt keinen Präprozessor in Java und damit auch keine `#define`-, `#include`- und `#ifdef`-Anweisungen.
- Der Backslash `\` darf nicht zur Verkettung von zwei aufeinanderfolgenden Zeilen verwendet werden.
- Konstante Strings, die mit `+` verkettet werden, faßt der Compiler zu einem einzigen String zusammen.

4.2 Primitive Datentypen

- 4.2 Primitive Datentypen
 - 4.2.1 Der logische Typ
 - Literale
 - 4.2.2 Der Zeichentyp
 - Literale
 - 4.2.3 Die integralen Typen
 - Literale
 - 4.2.4 Die Fließkommazahlen
 - Literale

Java kennt acht elementare Datentypen, die gemäß Sprachspezifikation als *primitive Datentypen* bezeichnet werden. Daneben gibt es die Möglichkeit, Arrays zu definieren (die eingeschränkte Objekttypen sind), und als objektorientierte Sprache erlaubt Java natürlich die Definition von Objekttypen.

Im Gegensatz zu C und C++ gibt es die folgenden Elemente in Java jedoch nicht:

- explizite Zeiger
- Typdefinitionen (typedef)
- Aufzählungen (enum)
- Recordtypen (struct und union)
- Bitfelder

Hinweis

Was auf den ersten Blick wie eine Designschwäche aussieht, entpuppt sich bei näherem Hinsehen als Stärke von Java. Der konsequente Verzicht auf zusätzliche Datentypen macht die Sprache leicht erlernbar und verständlich. Die Vergangenheit hat mehrfach gezeigt, daß Programmiersprachen mit einem überladenen Typkonzept (zum Beispiel PL/I oder ADA) auf Dauer keine Akzeptanz finden.

Tatsächlich ist es ohne weiteres möglich, die unverzichtbaren Datentypen mit den in Java eingebauten Hilfsmitteln nachzubilden. So lassen sich beispielsweise Zeiger zur Konstruktion dynamischer Datenstrukturen mit Hilfe von Referenzvariablen simulieren, und Recordtypen sind nichts anderes als Klassen ohne Methoden. Der Verzicht auf Low-Level-Datenstrukturen, wie beispielsweise Zeigern zur Manipulation von Speicherstellen oder Bitfeldern zur Repräsentation von Hardwareelementen, ist dagegen gewollt.

Alle primitiven Datentypen in Java haben eine feste Länge, die von den Designern der Sprache ein- für allemal verbindlich festgelegt wurde. Ein `sizeof`-Operator, wie er in C vorhanden ist, wird in Java daher nicht benötigt und ist auch nicht vorhanden.

Ein weiterer Unterschied zu C und den meisten anderen Programmiersprachen besteht darin, daß Variablen in Java immer einen definierten Wert haben. Bei Membervariablen (also Variablen innerhalb von Klassen, siehe [Kapitel 7](#)) bekommt eine Variable einen Standardwert zugewiesen, wenn dieser nicht durch eine explizite Initialisierung geändert wird. Bei *lokalen* Variablen sorgt der Compiler durch eine Datenflußanalyse dafür, daß diese vor ihrer Verwendung explizit initialisiert werden. Eine Erläuterung dieses Konzepts, das unter dem Namen *Definite Assignment* in der Sprachdefinition beschrieben wird, ist Bestandteil von [Kapitel 5](#). [Tabelle 4.1](#) listet die in Java verfügbaren Basistypen und ihre Standardwerte auf:

Typname	Länge	Wertebereich	Standardwert
boolean	1	true , false	false
char	2	Alle Unicode-Zeichen	\u0000
byte	1	-2 ⁷ ...2 ⁷ -1	0
short	2	-2 ¹⁵ ...2 ¹⁵ -1	0
int	4	-2 ³¹ ...2 ³¹ -1	0
long	8	-2 ⁶³ ...2 ⁶³ -1	0
float	4	+/-3.40282347 * 10 ³⁸	0.0
double	8	+/-1.79769313486231570 * 10 ³⁰⁸	0.0

Tabelle 4.1: Primitive Datentypen

4.2.1 Der logische Typ

Mit `boolean` besitzt Java einen eigenen logischen Datentyp und beseitigt damit eine oft diskutierte Schwäche von C und C++. Der `boolean`-Typ muß zwangsweise dort verwendet werden, wo ein logischer Operand erforderlich ist. Ganzzahlige Typen mit den Werten 0 oder 1 dürfen nicht als Ersatz für einen logischen Typen verwendet werden.

Literale

Der Datentyp `boolean` kennt zwei verschiedene Werte, nämlich `true` und `false`. Neben den vordefinierten Konstanten gibt es keine weiteren Literale für logische Datentypen.

4.2.2 Der Zeichentyp

Java wurde mit dem Anspruch entworfen, bekannte Schwächen bestehender Programmiersprachen zu vermeiden, und der Wunsch nach Portabilität stand ganz oben auf der Liste der Designziele. Konsequenterweise wurde der Typ `char` in Java daher bereits von Anfang an 2 Byte groß gemacht und speichert seine Zeichen auf der Basis des Unicode-Zeichensatzes. Als einziger integraler Datentyp ist `char` nicht vorzeichenbehaftet.

Da das Sprachdesign und das Java-API so gestaltet wurden, daß die Verwendung des Unicode-Zeichensatzes weitgehend transparent bleibt, ergeben sich für die meisten Entwickler zunächst kaum Umstellungsprobleme. Ein `char` oder `String` kann in Java genauso intuitiv benutzt werden wie in Sprachen, die auf dem ASCII-Zeichensatz aufbauen. Unterschiede werden vor allem dann deutlich, wenn Berührungspunkte zwischen der internen Unicode-Darstellung und der Repräsentation auf Systemebene entstehen, beispielsweise beim Lesen oder Schreiben von Textdateien.

Tip

Literale

`char`-Literale werden grundsätzlich in einfache Hochkommata gesetzt. Daneben gibt es `String`-Literale, die in doppelten Hochkommata stehen. Ähnlich wie C stellt Java eine ganze Reihe von Standard-Escape-Sequenzen zur Verfügung, die zur Darstellung von Sonderzeichen verwendet werden können:

Zeichen	Bedeutung
<code>\b</code>	Rückschritt (Backspace)
<code>\t</code>	Horizontaler Tabulator
<code>\n</code>	Zeilenschaltung (Newline)
<code>\f</code>	Seitenumbruch (Formfeed)
<code>\r</code>	Wagenrücklauf (Carriage return)
<code>\"</code>	Doppeltes Anführungszeichen
<code>\'</code>	Einfaches Anführungszeichen
<code>\\</code>	Backslash
<code>\nnn</code>	Oktalzahl nnn (kann auch kürzer als 3 Zeichen sein, darf nicht größer als oktal 377 sein)

Tabelle 4.2: Standard-Escape-Sequenzen

Weiterhin können beliebige Unicode-Escape-Sequenzen der Form `\uxxxx` angegeben werden, wobei `xxxx` eine Folge von bis zu 4 hexadezimalen Ziffern ist. So steht beispielsweise `\u000a` für die Zeilenschaltung und `\u0020` für das Leerzeichen.

Eine wichtiger Unterschied zu Standard-Escape-Sequenzen besteht darin, daß Unicode-Escape-Sequenzen an beliebiger Stelle im Programm auftauchen dürfen, also auch außerhalb von `char`- oder `String`-Literalen. Wichtig ist außerdem, daß diese bereits vor der eigentlichen Interpretation des Quelltextes ausgetauscht werden. Es ist also beispielsweise nicht möglich, ein `char`-Literal, das ein Anführungszeichen darstellen soll, in der Form `"\u0027"` zu schreiben. Da die Unicode-Sequenzen bereits vor dem eigentlichen Compiler-Lauf ausgetauscht werden, würde der Compiler die Sequenz `"'` vorfinden und einen Fehler melden.

Warnung

4.2.3 Die integralen Typen

Java stellt vier ganzzahlige Datentypen zur Verfügung, und zwar `byte`, `short`, `int` und `long`, mit jeweils 1, 2, 4 und 8 Byte Länge. Alle ganzzahligen Typen sind vorzeichenbehaftet, und ihre Länge ist auf allen Plattformen gleich.

Anders als in C sind die Schlüsselworte `long` und `short` bereits Typenbezeichner und nicht nur Modifier. Es ist daher nicht erlaubt, `long int`

oder `short int` anstelle von `long` bzw. `short` zu schreiben. Auch den Modifier `unsigned` gibt es in Java nicht.

Literale

Ganzzahlige Literale können in Dezimal-, Oktal- oder Hexadezimalform geschrieben werden. Ein oktaler Wert beginnt mit dem Präfix `0`, ein hexadezimaler Wert mit `0x`. Dezimale Literale dürfen nur aus den Ziffern `0` bis `9`, oktale aus den Ziffern `0` bis `7` und hexadezimale aus den Ziffern `0` bis `9` und den Buchstaben `a` bis `f` und `A` bis `F` bestehen.

Durch Voranstellen eines `-` können negative Zahlen dargestellt werden, positive können wahlweise durch ein `+` eingeleitet werden. Ganzzahlige Literale sind grundsätzlich vom Typ `int`, wenn nicht der Suffix `L` oder `l` hinten angehängt wird. In diesem Fall sind sie vom Typ `long`.

4.2.4 Die Fließkommazahlen

Java kennt die beiden IEEE-754-Fließkommatypen `float` (einfache Genauigkeit) und `double` (doppelte Genauigkeit). Die Länge beträgt 4 Byte für `float` und 8 Byte für `double`.

Literale

Fließkommalliterale werden immer in Dezimalnotation aufgeschrieben. Sie bestehen aus einem Vorkommateil, einem Dezimalpunkt, einem Nachkommateil, einem Exponenten und einem Suffix. Um ein Fließkommalliteral von einem integralen Literal unterscheiden zu können, muß mindestens der Dezimalpunkt, der Exponent oder der Suffix vorhanden sein. Entweder der Vorkomma- oder der Nachkommateil darf ausgelassen werden, aber nicht beide. Vorkommateil und Exponent können wahlweise durch das Vorzeichen `+` oder `-` eingeleitet werden. Weiterhin ist der Exponent, der durch ein `e` oder `E` eingeleitet wird, optional. Auch der Suffix kann weggelassen werden, wenn durch die anderen Merkmale klar ist, daß es sich um eine Fließkommazahl handelt. Der Suffix kann entweder `f` oder `F` sein, um anzuzeigen, daß es sich um ein `float` handelt, oder `d` oder `D`, um ein `double` anzuzeigen. Fehlt er, so ist das Literal (unabhängig von seiner Größe) vom Typ `double`.

Gültige Fließkommazahlen sind:

- 3.14
- 2f
- 1e1
- .5f
- 6.

Beispiel

Neben diesen numerischen Literalen gibt es noch einige symbolische in den Klassen `Float` und `Double` des Pakets `java.lang`. [Tabelle 4.3](#) gibt eine Übersicht dieser vordefinierten Konstanten. `NaN` entsteht beispielsweise bei der Division durch 0, `POSITIVE_INFINITY` bzw. `NEGATIVE_INFINITY` sind Zahlen, die größer bzw. kleiner als der darstellbare Bereich sind.

Name	Verfügbar für	Bedeutung
<code>MAX_VALUE</code>	<code>Float</code> , <code>Double</code>	Größter darstellbarer positiver Wert
<code>MIN_VALUE</code>	<code>Float</code> , <code>Double</code>	Kleinsten darstellbarer positiver Wert
<code>NaN</code>	<code>Float</code> , <code>Double</code>	Not-A-Number
<code>NEGATIVE_INFINITY</code>	<code>Float</code> , <code>Double</code>	Negativ unendlich
<code>POSITIVE_INFINITY</code>	<code>Float</code> , <code>Double</code>	Positiv unendlich

Tabelle 4.3: Symbolische Fließkommalliterale

4.3 Variablen

- 4.3 Variablen
 - 4.3.1 Grundeigenschaften
 - 4.3.2 Deklaration von Variablen
 - 4.3.3 Lebensdauer/Sichtbarkeit

4.3.1 Grundeigenschaften

Variablen dienen dazu, Daten im Hauptspeicher eines Programms abzulegen und gegebenenfalls zu lesen oder zu verändern. In Java gibt es drei Typen von Variablen:

- Instanzvariablen*, die im Rahmen einer Klassendefinition definiert und zusammen mit dem Objekt angelegt werden.
- Klassenvariablen*, die ebenfalls im Rahmen einer Klassendefinition definiert werden, aber unabhängig von einem konkreten Objekt existieren.
- Lokale Variablen*, die innerhalb einer Methode oder eines Blocks definiert werden und nur dort existieren.

Daneben betrachtet die Sprachdefinition auch Array-Komponenten und die Parameter von Methoden und Exception-Handlern als Variablen.

Eine Variable in Java ist immer typisiert. Sie ist entweder von einem primitiven Typen oder von einem Referenztypen. Mit Ausnahme eines Spezialfalls bei Array-Variablen, auf den wir später zurückkommen, werden alle Typüberprüfungen zur Compile-Zeit vorgenommen. Java ist damit im klassischen Sinne eine typsichere Sprache.

Hinweis

Um einer Variable vom Typ `T` einen Wert `X` zuzuweisen, müssen `T` und `X` zuweisungskompatibel sein. Welche Typen zuweisungskompatibel sind, wird am Ende dieses Kapitels in [Abschnitt 4.6](#) erklärt.

Variablen können auf zwei unterschiedliche Arten verändert werden:

- durch eine Zuweisung
- durch einen Inkrement- oder Dekrement-Operator

Beide Möglichkeiten werden in [Kapitel 5](#) ausführlich erklärt.

4.3.2 Deklaration von Variablen

Die Deklaration einer Variable erfolgt in der Form

```
Typname Variablenname;
```

Dabei wird eine Variable des Typs `Typname` mit dem Namen `Variablenname` angelegt. Variablendeklarationen dürfen in Java an beliebiger Stelle im Programmcode erfolgen. Das folgende Beispielprogramm legt die Variablen `a`, `b`, `c` und `d` an und gibt ihren Inhalt auf dem Bildschirm aus:

[Listing0402.java](#) Beispiel

```

001 /* Listing0402.java */
002
003 public class Listing0402
004 {
005     public static void main(String[] args)
006     {
007         int a;
008         a = 1;
009         char b = 'x';
010         System.out.println(a);
011         double c = 3.1415;
012         System.out.println(b);
013         System.out.println(c);
014         boolean d = false;
015         System.out.println(d);
016     }
017 }

```

Listing 4.2: Einfache Variablen ausgeben

Die Ausgabe des Programms ist:

```
1
x
3.1415
false
```

Wie in diesem Beispiel zu sehen ist, dürfen Variablen gleich bei der Deklaration initialisiert werden. Dazu ist einfach der gewünschte Wert hinter einem Zuweisungsoperator an die Deklaration anzuhängen:

Tip

```
char b = 'x';
double c = 3.1415;
boolean d = false;
```

Listing 4.3: Initialisieren von Variablen

4.3.3 Lebensdauer/Sichtbarkeit

Die Sichtbarkeit lokaler Variablen erstreckt sich von der Stelle ihrer Deklaration bis zum Ende der Methode, in der sie deklariert wurden. Falls innerhalb eines Blocks lokale Variablen angelegt wurden, sind sie bis zum Ende des Blocks sichtbar. Die Lebensdauer einer lokalen Variable beginnt, wenn die zugehörige Deklarationsanweisung ausgeführt wird. Sie endet mit dem Ende des Methodenaufrufs. Falls innerhalb eines Blocks lokale Variablen angelegt wurden, endet ihre Lebensdauer mit dem Verlassen des Blocks. Es ist in Java nicht erlaubt, lokale Variablen zu deklarieren, die gleichnamige lokale Variablen eines weiter außen liegenden Blocks verdecken. Das Verdecken von Klassen- oder Instanzvariablen ist dagegen zulässig.

Instanzvariablen werden zum Zeitpunkt des Erzeugens einer neuen Instanz einer Klasse angelegt. Sie sind innerhalb der ganzen Klasse sichtbar, solange sie nicht von gleichnamigen lokalen Variablen verdeckt werden. In diesem Fall ist aber der Zugriff mit Hilfe des `this`-Zeigers möglich: `this.name` greift immer auf die Instanz- oder Klassenvariable `name` zu, selbst wenn eine gleichnamige lokale Variable existiert. Mit dem Zerstören des zugehörigen Objektes werden auch alle Instanzvariablen zerstört.

Klassenvariablen leben während der kompletten Laufzeit des Programms. Die Regeln für ihre Sichtbarkeit entsprechen denen von Instanzvariablen.

4.4 Arrays

- [4.4 Arrays](#)
 - [4.4.1 Deklaration und Initialisierung](#)
 - [4.4.2 Zugriff auf Array-Elemente](#)
 - [4.4.3 Mehrdimensionale Arrays](#)

Arrays in Java unterscheiden sich dadurch von Arrays in anderen Programmiersprachen, daß sie *Objekte* sind. Obwohl dieser Umstand in vielen Fällen vernachlässigt werden kann, bedeutet er dennoch:

- daß Array-Variablen *Referenzen* sind
- daß Arrays Methoden und Instanz-Variablen besitzen
- daß Arrays zur Laufzeit erzeugt werden

Dennoch bleibt ein Array immer eine (möglicherweise mehrdimensionale) Reihung von Elementen eines festen Grundtyps. Arrays in Java sind *semidynamisch*, d.h. ihre Größe kann zur Laufzeit festgelegt, später aber nicht mehr verändert werden.

Hinweis

4.4.1 Deklaration und Initialisierung

Die Deklaration eines Arrays in Java erfolgt in zwei Schritten:

- Deklaration einer Array-Variablen
- Erzeugen eines Arrays und Zuweisung an die Array-Variable

Die Deklaration eines Arrays entspricht der einer einfachen Variablen, mit dem Unterschied, daß an den Typnamen oder den Variablennamen (beide Varianten sind erlaubt) eckige Klammern gehängt werden:

```
001 int a[];
002 double b[];
003 boolean[] c;
```

Listing 4.4: Deklaration von Arrays

Zum Zeitpunkt der Deklaration wird noch nicht festgelegt, wie viele Elemente das Array haben soll. Dies geschieht erst später bei der Initialisierung des Arrays, die mit Hilfe des [new](#)-Operators oder durch literale Initialisierung ausgeführt wird. Sollen also beispielsweise die oben deklarierten Arrays 5, 10 und 15 Elemente haben, so würden wir das Beispiel wie folgt erweitern:

Hinweis

```
001 a = new int[5];
002 b = new double[10];
003 c = new boolean[15];
```

Listing 4.5: Erzeugen von Arrays

Ist bereits zum Deklarationszeitpunkt klar, wie viele Elemente das Array haben soll, können Deklaration und Initialisierung zusammen geschrieben werden:

```
001 int a[] = new int[5];
002 double b[] = new double[10];
003 boolean c[] = new boolean[15];
```

Listing 4.6: Deklaration und Initialisierung von Arrays

Alternativ zur Verwendung des [new](#)-Operators kann ein Array auch *literal* initialisiert werden. Dazu werden die Elemente des Arrays in geschweifte Klammern gesetzt und nach einem Zuweisungsoperator zur Initialisierung verwendet. Die Größe des Arrays ergibt sich aus der Anzahl der zugewiesenen Elemente:

```
001 int x[] = {1,2,3,4,5};
002 boolean y[] = {true, true};
```

Listing 4.7: Initialisierung mit literalen Arrays

Das Beispiel generiert ein `int`-Array `x` mit fünf Elementen und ein `boolean`-Array `y` mit zwei Elementen. Anders als bei der expliziten Initialisierung mit `new` muß die Initialisierung in diesem Fall unmittelbar bei der Deklaration erfolgen.

4.4.2 Zugriff auf Array-Elemente

Bei der Initialisierung eines Arrays von `n` Elementen werden die einzelnen Elemente von `0` bis `n-1` durchnummeriert. Der Zugriff auf jedes einzelne Element erfolgt über seinen numerischen Index, der nach dem Array-Namen in eckigen Klammern geschrieben wird. Das nachfolgende Beispiel deklariert zwei Arrays mit Elementen des Typs `int` bzw. `boolean`, die dann ausgegeben werden:

[Listing0408.java](#) **Beispiel**

```
001 /* Listing0408.java */
002
003 public class Listing0408
004 {
005     public static void main(String[] args)
006     {
007         int prim[] = new int[5];
008         boolean b[] = {true,false};
009
010         prim[0] = 2;
011         prim[1] = 3;
012         prim[2] = 5;
013         prim[3] = 7;
014         prim[4] = 11;
015
016         System.out.println("prim hat "+prim.length+" Elemente");
017         System.out.println("b hat "+b.length+" Elemente");
018         System.out.println(prim[0]);
019         System.out.println(prim[1]);
020         System.out.println(prim[2]);
021         System.out.println(prim[3]);
022         System.out.println(prim[4]);
023         System.out.println(b[0]);
024         System.out.println(b[1]);
025     }
026 }
```

Listing 4.8: Deklaration und Zugriff auf Arrays

Die Ausgabe des Programms ist:

```
prim hat 5 Elemente
b hat 2 Elemente
2
3
5
7
11
true
false
```

Der Array-Index muß vom Typ `int` sein. Aufgrund der vom Compiler automatisch vorgenommenen Typkonvertierungen sind auch `short`, `byte` und `char` zulässig. Jedes Array hat eine Instanzvariable `length`, die die Anzahl seiner Elemente angibt. Indexausdrücke werden vom Laufzeitsystem auf Einhaltung der Array-Grenzen geprüft. Sie müssen größer gleich 0 und kleiner als `length` sein.

Tip

4.4.3 Mehrdimensionale Arrays

Mehrdimensionale Arrays werden erzeugt, indem zwei oder mehr Paare eckiger Klammern bei der Deklaration angegeben werden. Mehrdimensionale Arrays werden als Arrays von Arrays angelegt. Die Initialisierung erfolgt analog zu eindimensionalen Arrays durch Angabe der Anzahl der Elemente je Dimension.

Der Zugriff auf mehrdimensionale Arrays geschieht durch Angabe aller erforderlichen Indizes, jeweils in eigenen eckigen Klammern. Auch bei mehrdimensionalen Arrays kann eine literale Initialisierung durch Schachtelung der Initialisierungssequenzen erreicht werden. Das folgende Beispiel erzeugt ein Array der Größe `2 * 3` und gibt dessen Elemente aus:


```

001 /* Listing0409.java */
002
003 public class Listing0409
004 {
005     public static void main(String[] args)
006     {
007         int a[][] = new int[2][3];
008
009         a[0][0] = 1;
010         a[0][1] = 2;
011         a[0][2] = 3;
012         a[1][0] = 4;
013         a[1][1] = 5;
014         a[1][2] = 6;
015         System.out.println(""+a[0][0]+a[0][1]+a[0][2]);
016         System.out.println(""+a[1][0]+a[1][1]+a[1][2]);
017     }
018 }

```

Listing 4.9: Zugriff auf mehrdimensionale Arrays

Die Ausgabe des Programms ist:

```

123
456

```

Da mehrdimensionale Arrays als geschachtelte Arrays gespeichert werden, ist es auch möglich, *nicht-rechteckige* Arrays zu erzeugen. Das folgende Beispiel deklariert und initialisiert ein zweidimensionales dreieckiges Array und gibt es auf dem Bildschirm aus. Gleichzeitig zeigt es die Verwendung der `length`-Variable, um die Größe des Arrays oder Sub-Arrays herauszufinden:

Tip

Listing0410.java Beispiel

```

001 /* Listing0410.java */
002
003 public class Listing0410
004 {
005     public static void main(String[] args)
006     {
007         int a[][] = { {0},
008                       {1,2},
009                       {3,4,5},
010                       {6,7,8,9}
011                     };
012         for (int i=0; i<a.length; ++i) {
013             for (int j=0; j<a[i].length; ++j) {
014                 System.out.print(a[i][j]);
015             }
016             System.out.println();
017         }
018     }
019 }

```

Listing 4.10: Ein nicht-rechteckiges Array

Die Arbeitsweise des Programms ist trotz der erst in [Kapitel 6](#) vorzustellenden `for`-Schleife unmittelbar verständlich. Die Ausgabe des Programms lautet:

```

0
12
345
6789

```

4.5 Referenztypen

- [4.5 Referenztypen](#)
 - [4.5.1 Beschreibung](#)
 - [4.5.2 Speichermanagement](#)

4.5.1 Beschreibung

Referenztypen sind neben den primitiven Datentypen die zweite wichtige Klasse von Datentypen in Java. Zu den Referenztypen gehören Objekte, Strings und Arrays. Weiterhin gibt es die vordefinierte Konstante `null`, die eine *leere* Referenz bezeichnet.

Eigentlich sind auch Strings und Arrays Objekte, aber es gibt bei ihnen einige Besonderheiten, die eine Unterscheidung von normalen Objekten rechtfertigen:

- Sowohl bei Strings als auch bei Arrays kennt der Compiler Literale, die einen expliziten Aufruf des `new`-Operators überflüssig machen.
- Arrays sind »klassenlose« Objekte. Sie können ausschließlich vom Compiler erzeugt werden, besitzen aber keine explizite Klassendefinition. Dennoch haben sie eine öffentliche Instanzvariable `length` und werden vom Laufzeitsystem wie normale Objekte behandelt.
- Die Klasse `String` ist zwar wie eine gewöhnliche Klasse in der Laufzeitbibliothek von Java vorhanden. Der Compiler hat aber Kenntnisse über den inneren Aufbau von Strings und generiert bei Stringoperationen Code, der auf Methoden der Klassen `String` und `StringBuffer` zugreift. Eine ähnlich enge Zusammenarbeit zwischen Compiler und Laufzeitbibliothek gibt es auch bei Threads und Exceptions. Wir werden auf diese Besonderheiten in den nachfolgenden Kapiteln noch einmal zurückkommen.

Das Verständnis für Referenztypen ist entscheidend für die Programmierung in Java.

Referenztypen können prinzipiell genauso benutzt werden wie primitive Typen. Da sie jedoch lediglich einen Verweis darstellen, ist die Semantik einiger Operatoren anders als bei primitiven Typen:

Hinweis

- Die Zuweisung einer Referenz kopiert lediglich den Verweis auf das betreffende Objekt, das Objekt selbst dagegen bleibt unkopiert. Nach einer Zuweisung zweier Referenztypen zeigen diese also auf ein und dasselbe Objekt. Sollen Referenztypen kopiert werden, so ist ein Aufruf der Methode `clone` erforderlich (siehe [Kapitel 7](#)).
- Der Gleichheitstest zweier Referenzen testet, ob beide Verweise gleich sind, d.h. auf ein und dasselbe Objekt zeigen. Das ist aber eine strengere Forderung als inhaltliche Gleichheit. Soll lediglich auf inhaltliche Gleichheit getestet werden, kann dazu die `equals`-Methode verwendet werden, die von den meisten Klassen implementiert wird (ebenfalls in [Kapitel 7](#) erläutert). Analoges gilt für den Test auf Ungleichheit.

Anders als in C und C++, wo der `*`-Operator zur Dereferenzierung eines Zeigers nötig ist, erfolgt in Java der Zugriff auf Referenztypen in der gleichen Weise wie der auf primitive Typen. Einen expliziten Dereferenzierungsoperator gibt es dagegen nicht.

4.5.2 Speichermanagement

Während primitive Typen lediglich deklariert werden, reicht dies bei Referenztypen nicht aus. Sie müssen mit Hilfe des `new`-Operators oder - im Falle von Arrays und Strings - durch Zuweisung von Literalen zusätzlich noch explizit erzeugt werden.

```
Vector v = new Vector();
```

Beispiel

Listing 4.11: Erzeugen eines Objekts mit dem new-Operator

Java verfügt über ein automatisches Speichermanagement. Dadurch braucht man sich als Java-Programmierer nicht um die Rückgabe von Speicher zu kümmern, der von Referenzvariablen belegt wird. Ein mit niedriger Priorität im Hintergrund arbeitender *Garbage Collector* sucht periodisch nach Objekten, die nicht mehr referenziert werden, um den durch sie belegten Speicher freizugeben.

4.6 Typkonvertierungen

- 4.6 Typkonvertierungen

Es gibt diverse Konvertierungen zwischen unterschiedlichen Datentypen in Java. Diese werden einerseits vom Compiler automatisch vorgenommen, beispielsweise bei der Auswertung von numerischen Ausdrücken. Andererseits können sie verwendet werden, um mit Hilfe des Type-Cast-Operators (siehe [Kapitel 5](#)) eigene Konvertierungen vorzunehmen.

Java unterscheidet prinzipiell zwischen *erweiternden* und *einschränkenden Konvertierungen* und diese noch einmal nach primitiven Typen und Referenztypen. Zunächst zu den Referenztypen:

- Als erweiternde Konvertierung eines Referenztyps `T` wird vor allem die Umwandlung eines Objekts vom Typ `T` in eine seiner Vaterklassen angesehen.
- Als einschränkende Konvertierung eines Referenztyps `T` wird vor allem die Umwandlung eines Objekts vom Typ `T` in eine der aus `T` abgeleiteten Klassen angesehen.

Daneben gibt es noch eine ganze Reihe weiterer Regeln zur Definition von erweiternden und einschränkenden Konvertierungen von Referenztypen. Die Bedeutung von Vaterklassen und den daraus abgeleiteten Unterklassen wird in [Kapitel 7](#) ausführlich erläutert.

Konvertierungen auf primitiven Datentypen sind etwas aufwendiger zu erklären. Wir benutzen dazu [Abbildung 4.1](#):

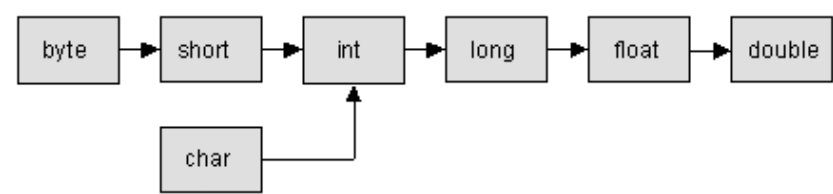


Abbildung 4.1: Konvertierungen auf primitiven Datentypen

Jede Konvertierung, die in Pfeilrichtung erfolgt, beschreibt eine erweiternde Konvertierung, und jede Konvertierung, die entgegen der Pfeilrichtung erfolgt, beschreibt eine einschränkende Konvertierung. Andere Konvertierungen zwischen primitiven Datentypen sind nicht erlaubt. Insbesondere gibt es also keine legale Konvertierung von und nach [boolean](#) und auch keine Konvertierung zwischen primitiven Typen und Referenztypen.

Welche Bedeutung haben nun aber die verschiedenen Konvertierungen zwischen unterschiedlichen Typen? Wir wollen uns an dieser Stelle lediglich mit den Konvertierungen zwischen primitiven Typen beschäftigen. Wie aus [Abbildung 4.1](#) ersichtlich ist, beschränken sich diese auf Umwandlungen zwischen numerischen Typen. Die Anwendung einer erweiternden Konvertierung wird in folgenden Fällen vom Compiler automatisch vorgenommen:

- Bei einer Zuweisung, wenn der Typ der Variablen und des zugewiesenen Ausdrucks nicht identisch ist.
- Bei der Auswertung eines arithmetischen Ausdrucks, wenn Operanden unterschiedlich typisiert sind.
- Beim Aufruf einer Methode, falls die Typen der aktuellen Parameter nicht mit denen der formalen Parameter übereinstimmen.

Es ist daher beispielsweise ohne weiteres möglich, ein [short](#) und ein [int](#) gemeinsam in einem Additionsausdruck zu verwenden, da ein [short](#) mit Hilfe einer erweiternden Konvertierung in ein [int](#) verwandelt werden kann. Ebenso ist es möglich, ein [char](#) als Array-Index zu verwenden, da es erweiternd in ein [int](#) konvertiert werden kann. Auch die Arithmetik in Ausdrücken, die sowohl integrale also auch Fließkommawerte enthalten, ist möglich, da der Compiler alle integralen Parameter erweiternd in Fließkommawerte umwandeln kann.

Es ist dagegen nicht ohne weiteres möglich, einer [int](#)-Variablen einen [double](#)-Wert zuzuweisen. Die hierzu erforderliche einschränkende Konvertierung nimmt der Compiler nicht selbst vor; sie kann allerdings mit Hilfe des Type-Cast-Operators manuell durchgeführt werden. Auch die Verwendung eines [long](#) als Array-Index verbietet sich aus diesem Grund.

Bei den einschränkenden Konvertierungen kann es passieren, daß ein Wert verfälscht wird, da der Wertebereich des Zielobjekts kleiner ist. Aber auch erweiternde Konvertierungen sind nicht immer gefahrlos möglich. So kann zwar beispielsweise ein [float](#) mindestens genauso große Werte aufnehmen wie ein [long](#). Seine Genauigkeit ist aber auf ca. 8 Stellen beschränkt, und daher können größere Ganzzahlen (z.B. 1000000123) nicht mehr mit voller Genauigkeit dargestellt werden.

Warnung

4.7 Zusammenfassung

- [4.7 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Der Unicode-Zeichensatz
- Namenskonventionen für Bezeichner
- Kommentare in Java
- Die primitiven Datentypen [boolean](#), [char](#), [byte](#), [short int](#), [long](#), [float](#) und [double](#)
- Die boolschen Literale [true](#) und [false](#) sowie die Fließkommalliterale [MAX_VALUE](#), [MIN_VALUE](#), [NaN](#), [NEGATIVE_INFINITY](#) und [POSITIVE_INFINITY](#)
- Die Standard-Escape-Sequenzen [\b](#), [\t](#), [\n](#), [\f](#), [\r](#), [\"](#), [\'](#), [\\](#) und [\nnn](#)
- Deklaration, Lebensdauer und Sichtbarkeit von Variablen
- Deklaration, Initialisierung und Zugriff auf Arrays
- Referenztypen und automatisches Speichermanagement
- Typkonvertierungen

Kapitel 5

Ausdrücke

- [5 Ausdrücke](#)
 - [5.1 Eigenschaften von Ausdrücken](#)
 - [5.2 Arithmetische Operatoren](#)
 - [5.3 Relationale Operatoren](#)
 - [5.4 Logische Operatoren](#)
 - [5.5 Bitweise Operatoren](#)
 - [5.6 Zuweisungsoperatoren](#)
 - [5.7 Sonstige Operatoren](#)
 - [5.7.1 Weitere Operatoren für primitive Typen](#)
 - [Fragezeichenoperator](#)
 - [Type-Cast-Operator](#)
 - [5.7.2 Operatoren für Objekte](#)
 - [String-Verkettung](#)
 - [Referenzgleichheit und -ungleichheit](#)
 - [Der instanceof-Operator](#)
 - [Der new-Operator](#)
 - [Member-Zugriff](#)
 - [Methodenaufruf](#)
 - [Zugriff auf Array-Elemente](#)
 - [5.7.3 Welche Operatoren es nicht gibt](#)
 - [5.8 Operator-Vorrangregeln](#)
 - [5.9 Zusammenfassung](#)

5.1 Eigenschaften von Ausdrücken

- 5.1 Eigenschaften von Ausdrücken

Wie in den meisten anderen Programmiersprachen gehören auch in Java *Ausdrücke* zu den kleinsten ausführbaren Einheiten eines Programms. Sie dienen dazu, Variablen einen Wert zuzuweisen, numerische Berechnungen durchzuführen oder logische Bedingungen zu formulieren.

Ein Ausdruck besteht immer aus mindestens einem Operator und einem oder mehreren Operanden, auf die der Operator angewendet wird. Nach den Typen der Operanden unterscheidet man *numerische*, *relationale*, *logische*, *bitweise*, *Zuweisungs*- und *sonstige* Operatoren. Jeder Ausdruck hat einen Rückgabewert, der durch die Anwendung des Operators auf die Operanden entsteht. Der Typ des Rückgabewerts bestimmt sich aus den Typen der Operanden und der Art des verwendeten Operators.

Neben der Typisierung ist die *Stelligkeit eines Operators* von Bedeutung. Operatoren, die lediglich ein Argument erwarten, nennt man *einstellig*, solche mit zwei Argumenten *zweistellig*. Beispiele für einstellige Operatoren sind das unäre Minus (also das negative Vorzeichen) oder der logische Nicht-Operator. Arithmetische Operatoren wie Addition oder Subtraktion sind zweistellig. Darüber hinaus gibt es in Java - wie in C - auch den dreistelligen Fragezeichenoperator.

Für die richtige Interpretation von Ausdrücken muß man die *Bindungs*- und *Assoziativitätsregeln* der Operatoren kennen. Bindungsregeln beschreiben die Reihenfolge, in der verschiedene Operatoren innerhalb eines Ausdrucks ausgewertet werden. So besagt beispielsweise die bekannte Regel »Punktrechnung vor Strichrechnung«, daß der Multiplikationsoperator eine höhere Bindungskraft hat als der Additionsoperator und demnach in Ausdrücken zuerst ausgewertet wird. Assoziativität beschreibt die Auswertungsreihenfolge von Operatoren derselben Bindungskraft, also beispielsweise die Auswertungsreihenfolge einer Kette von Additionen und Subtraktionen. Da Summationsoperatoren linksassoziativ sind, wird beispielsweise der Ausdruck `a-b+c` wie `(a-b)+c` ausgewertet und nicht wie `a-(b+c)`.

Neben ihrer eigentlichen Funktion, einen Rückgabewert zu produzieren, haben einige Operatoren auch *Nebeneffekte*. Als Nebeneffekt bezeichnet man das Verhalten eines Ausdrucks, auch ohne explizite Zuweisung die Inhalte von Variablen zu verändern. Meist sind diese Nebeneffekte erwünscht, wie etwa bei der Verwendung der Inkrement- und Dekrementoperatoren. Komplexere Ausdrücke können wegen der oftmals verdeckten Auswertungsreihenfolge der Teilausdrücke jedoch unter Umständen schwer verständliche Nebeneffekte enthalten und sollten deshalb mit Vorsicht angewendet werden.

Im Gegensatz zu vielen anderen Programmiersprachen ist in Java die *Reihenfolge der Auswertung* der Operanden innerhalb eines Teilausdrucks wohldefiniert. Die Sprachdefinition schreibt explizit vor, den linken Operanden eines Ausdrucks vollständig vor dem rechten Operanden auszuwerten. Falls also beispielsweise `i` den Wert 2 hat, dann ergibt der Ausdruck `(i=3) * i` in jedem Fall den Wert 9 und nicht 6.

Neben der natürlichen Auswertungsreihenfolge, die innerhalb eines Ausdrucks durch die Bindungs- und Assoziativitätsregeln vorgegeben wird, läßt sich durch eine explizite Klammerung jederzeit eine andere Reihenfolge erzwingen. Während das Ergebnis des Ausdrucks `4+2*5` wegen der Bindungsregeln 14 ist, liefert `(4+2)*5` das Resultat 30. Die Regeln für die Klammerung von Teilausdrücken in Java gleichen denen aller anderen Programmiersprachen und brauchen daher nicht weiter erläutert zu werden.

In Java gibt es ein Konzept, das sich *Definite Assignment* nennt. Gemeint ist damit die Tatsache, daß jede lokale Variable vor ihrer ersten Verwendung definitiv initialisiert sein muß. Das wünscht sich eigentlich zwar auch jeder Programmierer, aber in Java wird dies durch den Compiler sichergestellt! Dazu muß im Quelltext eine Datenflußanalyse durchgeführt werden, die jeden möglichen Ausführungspfad von der Deklaration einer Variablen bis zu ihrer Verwendung ermittelt und sicherstellt, daß kein Weg existiert, der eine Initialisierung auslassen würde.

Die folgende Methode `Test` läßt sich beispielsweise deshalb nicht fehlerfrei kompilieren, weil `k` vor der Ausgabeanweisung nicht initialisiert wird, wenn `i` kleiner 2 ist.

Beispiel

```
001 public static void Test(int i)
002 {
003     int k;
004     if (i >= 2) {
005         k = 5;
006     }
007     System.out.println(k);
008 }
```

Listing 5.1: Fehler beim Kompilieren durch unvollständige Initialisierung

Ein solches Verhalten des Compilers ist natürlich höchst wünschenswert, denn es zeigt einen *tatsächlichen* Fehler an, der sonst unbemerkt geblieben wäre. Daß die Datenflußanalyse allerdings auch Grenzen hat, zeigt das folgende Beispiel, bei dem ebenfalls ein Compiler-Fehler auftritt:

Tip

```
001 public static void Test(int i)
002 {
003     int k;
004     if (i < 2) {
005         k = 5;
006     }
007     if (i >= 2) {
008         k = 6;
009     }
010     System.out.println(k);
011 }
```

Listing 5.2: Fehler beim Kompilieren durch unvollständige Datenflußanalyse

Natürlich kann hier `k` nicht uninitialisiert bleiben, denn eine der beiden Bedingungen ist immer wahr. Leider gehen die Fähigkeiten der Compiler noch nicht so weit, dies zu erkennen.

Es wäre in diesem Fall natürlich vernünftiger gewesen, den `else`-Zweig an die erste Verzweigung anzuhängen und damit die zweite ganz einzusparen.

In Wirklichkeit funktioniert die Datenflußanalyse bei vernünftigem Programmierstil recht gut. Die wenigen Fälle, in denen der Compiler sich irrt, können in der Regel durch explizite Initialisierungen aufgelöst werden.

5.2 Arithmetische Operatoren

- 5.2 Arithmetische Operatoren

Java kennt die üblichen arithmetischen Operatoren der meisten imperativen Programmiersprachen, nämlich die *Addition*, *Subtraktion*, *Multiplikation*, *Division* und den *Restwertoperator*. Zusätzlich gibt es die einstelligen Operatoren für positives und negatives Vorzeichen sowie die nebeneffektbehafteten *Prä-* und *Postinkrement-* und *Prä-* und *Postdekrement-*Operatoren.

Die arithmetischen Operatoren erwarten numerische Operanden und liefern einen numerischen Rückgabewert. Haben die Operanden unterschiedliche Typen, beispielsweise `int` und `float`, so entspricht der Ergebnistyp des Teilausdrucks dem größeren der beiden Operanden. Zuvor wird der kleinere der beiden Operanden mit Hilfe einer erweiternden Konvertierung in den Typ des größeren konvertiert.

[Tabelle 5.1](#) gibt eine Übersicht der in Java verfügbaren arithmetischen Operatoren.

Operator	Bezeichnung	Bedeutung
+	Positives Vorzeichen	+n ist gleichbedeutend mit n
-	Negatives Vorzeichen	-n kehrt das Vorzeichen von n um
+	Summe	a + b ergibt die Summe von a und b
-	Differenz	a - b ergibt die Differenz von a und b
*	Produkt	a * b ergibt das Produkt aus a und b
/	Quotient	a / b ergibt den Quotienten von a und b
%	Restwert	a % b ergibt den Rest der ganzzahligen Division von a durch b. In Java läßt sich dieser Operator auch auf Fließkommazahlen anwenden.
++	Präinkrement	++a ergibt a+1 und erhöht a um 1
++	Postinkrement	a++ ergibt a und erhöht a um 1
--	Prädekrement	--a ergibt a-1 und verringert a um 1
--	Postdekrement	a-- ergibt a und verringert a um 1

Tabelle 5.1: Arithmetische Operatoren

5.3 Relationale Operatoren

- 5.3 Relationale Operatoren

Relationale Operatoren dienen dazu, Ausdrücke miteinander zu vergleichen und in Abhängigkeit davon einen logischen Rückgabewert zu produzieren. Java stellt den *Gleichheits-* und *Ungleichheitstest* sowie die Vergleiche *Größer* und *Kleiner* sowie *Größer gleich* und *Kleiner gleich* zur Verfügung. Die relationalen Operatoren arbeiten auf beliebigen - auch gemischten - numerischen Typen. Im Fall von Gleichheit und Ungleichheit funktionieren sie auch auf Objekttypen. [Tabelle 5.2](#) gibt eine Übersicht der in Java verfügbaren relationalen Operatoren.

Operator	Bezeichnung	Bedeutung
==	Gleich	a == b ergibt true, wenn a gleich b ist. Sind a und b Referenztypen, so ist der Rückgabewert true, wenn beide Werte auf dasselbe Objekt zeigen.
!=	Ungleich	a != b ergibt true, wenn a ungleich b ist. Sind a und b Objekte, so ist der Rückgabewert true, wenn beide Werte auf unterschiedliche Objekte zeigen.
<	Kleiner	a < b ergibt true, wenn a kleiner b ist.
<=	Kleiner gleich	a <= b ergibt true, wenn a kleiner oder gleich b ist.
>	Größer	a > b ergibt true, wenn a größer b ist.
>=	Größer gleich	a >= b ergibt true, wenn a größer oder gleich b ist.

Tabelle 5.2: Relationale Operatoren

5.4 Logische Operatoren

- 5.4 Logische Operatoren

Logische Operatoren dienen dazu, *boolesche* Werte miteinander zu verknüpfen. Im Gegensatz zu den relationalen Operatoren, die durch Vergleiche erst Wahrheitswerte produzieren, werden logische Operatoren zur Weiterverarbeitung von Wahrheitswerten verwendet.

Java stellt die Grundoperationen *UND*, *ODER* und *NICHT* zur Verfügung und bietet darüber hinaus die Möglichkeit, das Auswertungsverhalten der Operanden zu beeinflussen. Anders als die meisten anderen Programmiersprachen stellt Java die UND- und ODER-Verknüpfungen in zwei verschiedenen Varianten zur Verfügung, nämlich mit *Short-Circuit-Evaluation* oder ohne.

Bei der Short-Circuit-Evaluation eines logischen Ausdrucks wird ein weiter rechts stehender Teilausdruck nur dann ausgewertet, wenn er für das Ergebnis des Gesamtausdrucks noch von Bedeutung ist. Falls in dem Ausdruck `A && B` also bereits `A` falsch ist, wird zwangsläufig immer auch `A && B` falsch sein, unabhängig von dem Resultat von `B`. Bei der Short-Circuit-Evaluation wird in diesem Fall `B` gar nicht mehr ausgewertet. Analoges gilt bei der Anwendung des ODER-Operators.

Hinweis

Der in Java ebenfalls verfügbare *EXKLUSIV-ODER-Operator* muß natürlich immer in der langen Variante ausgewertet werden. [Tabelle 5.3](#) gibt eine Übersicht der logischen Operatoren.

Operator	Bezeichnung	Bedeutung
!	Logisches NICHT	!a ergibt false, wenn a wahr ist und true, wenn a false ist.
&&	UND mit Short-Circuit-Evaluation	a && b ergibt true, wenn sowohl a als auch b wahr sind. Ist a bereits falsch, so wird false zurückgegeben und b nicht mehr ausgewertet.
	ODER mit Short-Circuit-Evaluation	a b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Ist bereits a wahr, so wird true zurückgegeben und b nicht mehr ausgewertet.
&	UND ohne Short-Circuit-Evaluation	a & b ergibt true, wenn sowohl a als auch b wahr sind. Beide Teilausdrücke werden ausgewertet.
	ODER ohne Short-Circuit-Evaluation	a b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Beide Teilausdrücke werden ausgewertet.
^	EXKLUSIV-ODER	a ^ b ergibt true, wenn beide Ausdrücke einen unterschiedlichen Wahrheitswert haben.

Tabelle 5.3: Logische Operatoren

5.5 Bitweise Operatoren

- 5.5 Bitweise Operatoren

Mit Hilfe der bitweisen Operatoren kann auf die Binärdarstellung von numerischen Operanden zugegriffen werden. Ein numerischer Datentyp wird dabei als Folge von Bits angesehen, die mit Hilfe der bitweisen Operatoren einzeln abgefragt und manipuliert werden können.

Java hat dieselben bitweisen Operatoren wie C und C++ und stellt daher *Schiebeoperationen*, *logische Verknüpfungen* und das *Einerkomplement* zur Verfügung. Da alle numerischen Typen in Java vorzeichenbehaftet sind, gibt es einen zusätzlichen *Rechtsschiebeoperator* `>>>`, der das höchstwertige Bit nach der Verschiebung auf 0 setzt - und zwar auch dann, wenn es vorher auf 1 stand. [Tabelle 5.4](#) gibt eine Übersicht über die bitweisen Operatoren in Java.

Operator	Bezeichnung	Bedeutung
~	Einerkomplement	~a entsteht aus a, indem alle Bits von a invertiert werden.
	Bitweises ODER	a b ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander ODER-verknüpft werden.
&	Bitweises UND	a & b ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander UND-verknüpft werden.
^	Bitweises EXKLUSIV-ODER	a ^ b ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander EXKLUSIV-ODER-verknüpft werden.
>>	Rechtsschieben mit Vorzeichen	a >> b ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach rechts geschoben werden. Falls das höchstwertige Bit gesetzt ist (a also negativ ist), wird auch das höchstwertige Bit des Results gesetzt.
>>>	Rechtsschieben ohne Vorzeichen	a >>> b ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach rechts geschoben werden. Dabei wird das höchstwertige Bit des Results immer auf 0 gesetzt.
<<	Linksschieben	a << b ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach links geschoben werden. Das niederwertigste Bit wird mit 0 aufgefüllt, das höchstwertige Bit (also das Vorzeichen) wird von a übernommen.

Tabelle 5.4: Bitweise Operatoren

5.6 Zuweisungsoperatoren

- 5.6 Zuweisungsoperatoren

Auch die Zuweisungsoperatoren in Java entsprechen im großen und ganzen den Zuweisungsoperatoren von C und C++. Ebenso gilt die Zuweisung nicht als *Anweisung*, sondern als *Ausdruck*, der einen Rückgabewert erzeugt.

Die Verwechslung der relationalen Operatoren *Zuweisung* und *Gleichheitstest* (= und ==) war in C eines der Kardinalprobleme, in Java kann sie nicht mehr so leicht passieren. Sind beispielsweise `a` und `b` vom Typ `int`, so hat zwar der Ausdruck `a = b` einen definierten Rückgabewert wie in C. Er darf jedoch nicht als Kontrollausdruck einer Schleife oder Verzweigung verwendet werden, da er nicht vom Typ `boolean` ist. Anders als in C, wo boolesche Werte durch Ganzzahlen simuliert werden, schließt Java diese Art von Fehler also von vornherein aus. Nur wenn `a` und `b` vom Typ `boolean` sind, wird das Verwechseln von Zuweisung und Gleichheitstest vom Compiler nicht bemerkt.

Hinweis

Ebenso wie in C können auch in Java numerische bzw. bitweise Operatoren mit der Zuweisung kombiniert werden. Der Ausdruck `a+=b` addiert `b` zu `a`, speichert das Ergebnis in `a` und liefert es ebenfalls als Rückgabewert zurück. [Tabelle 5.5](#) gibt eine Übersicht der in Java verfügbaren Zuweisungsoperatoren.

Operator	Bezeichnung	Bedeutung
=	Einfache Zuweisung	a = b weist a den Wert von b zu und liefert b als Rückgabewert.
+=	Additionszuweisung	a += b weist a den Wert von a + b zu und liefert a + b als Rückgabewert.
-=	Subtraktionszuweisung	a -= b weist a den Wert von a - b zu und liefert a - b als Rückgabewert.
*=	Multiplikationszuweisung	a *= b weist a den Wert von a * b zu und liefert a * b als Rückgabewert.
/=	Divisionszuweisung	a /= b weist a den Wert von a / b zu und liefert a / b als Rückgabewert.
%=	Modulozuweisung	a %= b weist a den Wert von a % b zu und liefert a % b als Rückgabewert.
&=	UND-Zuweisung	a &= b weist a den Wert von a & b zu und liefert a & b als Rückgabewert.
=	ODER-Zuweisung	a = b weist a den Wert von a b zu und liefert a b als Rückgabewert.
^=	EXKLUSIV-ODER-Zuweisung	a ^= b weist a den Wert von a ^ b zu und liefert a ^ b als Rückgabewert.
<<=	Linksschiebezuweisung	a <<= b weist a den Wert von a << b zu und liefert a << b als Rückgabewert.
>>=	Rechtsschiebezuweisung	a >>= b weist a den Wert von a >> b zu und liefert a >> b als Rückgabewert.
>>>=	Rechtsschiebezuweisung mit Nullexpansion	a >>>= b weist a den Wert von a >>> b zu und liefert a >>> b als Rückgabewert.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

5.7 Sonstige Operatoren

- [5.7 Sonstige Operatoren](#)
 - [5.7.1 Weitere Operatoren für primitive Typen](#)
 - [Fragezeichenoperator](#)
 - [Type-Cast-Operator](#)
 - [5.7.2 Operatoren für Objekte](#)
 - [String-Verkettung](#)
 - [Referenzgleichheit und -ungleichheit](#)
 - [Der instanceof-Operator](#)
 - [Der new-Operator](#)
 - [Member-Zugriff](#)
 - [Methodenaufruf](#)
 - [Zugriff auf Array-Elemente](#)
 - [5.7.3 Welche Operatoren es nicht gibt](#)

Neben den bisher vorgestellten Operatoren stellt Java noch eine Reihe weiterer Operatoren zur Verfügung, die in diesem Abschnitt erläutert werden sollen.

5.7.1 Weitere Operatoren für primitive Typen

Fragezeichenoperator

Der Fragezeichenoperator `?:` ist der einzige dreistellige Operator in Java. Er erwartet einen logischen Ausdruck und zwei weitere Ausdrücke, die beide entweder numerisch, von einem Referenztyp oder vom Typ `boolean` sind.

Bei der Auswertung wird zunächst der Wert des logischen Operators ermittelt. Ist dieser wahr, so wird der erste der beiden anderen Operanden ausgewertet, sonst der zweite. Das Ergebnis des Ausdrucks `a ? b : c` ist also `b`, falls `a` wahr ist und `c`, falls `a` falsch ist. Der Typ des Rückgabewerts entspricht dem Typ des größeren der beiden Ausdrücke `a` und `b`.

Type-Cast-Operator

Ebenso wie in C gibt es auch in Java einen Type-Cast-Operator, mit dessen Hilfe explizite Typumwandlungen vorgenommen werden können. Der Ausdruck `(type) a` wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um. Auch wenn `a` eine Variable ist, ist das Ergebnis von `(type) a` ein Ausdruck, der nicht mehr auf der linken, sondern nur noch auf der rechten Seite eines Zuweisungsoperators stehen darf.

Wie in [Kapitel 4](#) erklärt, gibt es verschiedene Arten von Typkonvertierungen in Java. Mit Hilfe des Type-Cast-Operators dürfen alle legalen Typkonvertierungen vorgenommen werden. Der Type-Cast-Operator wird vor allem dann angewendet, wenn der Compiler keine impliziten Konvertierungen vornimmt; beispielsweise bei der Zuweisung von größeren an kleinere numerische Typen oder bei der Umwandlung von Objekttypen.

5.7.2 Operatoren für Objekte

Es gibt in Java einige Ausdrücke und Operatoren, die mit Objekten arbeiten oder Objekte produzieren. Die meisten von ihnen können erst dann erläutert werden, wenn die entsprechenden Konzepte in späteren Kapiteln eingeführt wurden. Der Vollständigkeit halber sollen sie dennoch an dieser Stelle erwähnt werden.

Hinweis

String-Verkettung

Der `+`-Operator kann nicht nur mit numerischen Operanden verwendet werden, sondern auch zur Verkettung von Strings. Ist wenigstens einer der beiden Operatoren in `a + b` ein String, so wird der gesamte Ausdruck als String-Verkettung ausgeführt. Hierzu wird gegebenenfalls zunächst der Nicht-String-Operand in einen String umgewandelt und anschließend mit dem anderen Operanden verkettet. Das Ergebnis der Operation ist wieder ein String, in dem beide Operanden hintereinander stehen.

In Java ist die Konvertierung in einen String für nahezu jeden Typen definiert. Bei primitiven Typen wird die Umwandlung vom Compiler und bei Referenztypen durch die Methode `toString` ausgeführt. Die String-Verkettung ist daher sehr universell zu verwenden und ermöglicht (beispielsweise zu Ausgabezwecken) eine sehr bequeme Zusammenfassung von Ausdrücken unterschiedlichen Typs. Ein typisches Beispiel für die Verwendung der String-Verkettung ist die Ausgabe von numerischen Ergebnissen auf dem Bildschirm:

Tip

[Listing0503.java](#)

Beispiel

```
001 /* Listing0503.java */
002
003 public class Listing0503
004 {
005     public static void main(String[] args)
006     {
007         int a = 5;
008         double x = 3.14;
009
010         System.out.println("a = " + a);
011         System.out.println("x = " + x);
012     }
013 }
```

Listing 5.3: String-Verkettung

Die Ausgabe des Programms lautet:

```
a = 5
x = 3.14
```

Etwas Vorsicht ist geboten, wenn sowohl String-Verkettung als auch Addition in einem Ausdruck verwendet werden sollen, da die in diesem Fall geltenden Vorrang- und Assoziativitätsregeln zu unerwarteten Ergebnissen führen können. Das folgende Programm gibt daher nicht `3 + 4 = 7`, sondern `3 + 4 = 34` aus.

Warnung

[Listing0504.java](#)

```
001 /* Listing0504.java */
002
003 public class Listing0504
004 {
005     public static void main(String[] args)
006     {
007         System.out.println("3 + 4 = " + 3 + 4);
008     }
009 }
```

Listing 5.4: Vorsicht bei der String-Verkettung!

Um das gewünschte Ergebnis zu erzielen, müßte der Teilausdruck `3 + 4` geklammert werden:

[Listing0505.java](#)

```
001 /* Listing0505.java */
002
003 public class Listing0505
004 {
005     public static void main(String[] args)
006     {
007         System.out.println("3 + 4 = " + (3 + 4));
008     }
009 }
```

Listing 5.5: Korrekte String-Verkettung bei gemischten Ausdrücken

Referenzgleichheit und -ungleichheit

Die Operatoren `==` und `!=` können auch auf Objekte, also auf Referenztypen, angewendet werden. In diesem Fall ist zu beachten, daß dabei lediglich die Gleichheit oder Ungleichheit der Referenz getestet wird. Es wird also überprüft, ob die Objektzeiger auf ein und dasselbe Objekt zeigen und nicht, ob die Objekte *inhaltlich* übereinstimmen.

Hinweis

Ein einfaches Beispiel ist der Vergleich zweier Strings `a` und `b`, die beide den Inhalt "hallo" haben:

Beispiel

[Listing0506.java](#)

```
001 /* Listing0506.java */
002
003 public class Listing0506
004 {
005     public static void main(String[] args)
006     {
007         String a = new String("hallo");
008         String b = new String("hallo");
009         System.out.println("a == b liefert " + (a == b));
010         System.out.println("a != b liefert " + (a != b));
011     }
012 }
```

Listing 5.6: Vergleichen von Referenzen

Werden sie zur Laufzeit angelegt (wie in diesem Beispiel), liefert das Programm das erwartete Ergebnis, denn `a` und `b` sind Referenzen auf unterschiedliche Objekte, also Zeiger auf unterschiedliche Instanzen derselben Klasse.

```
a == b liefert false
a != b liefert true
```

Dies ist das erwartete Verhalten für fast alle Objektreferenzen. Werden die Strings als Literale dagegen zur Compile-Zeit angelegt und damit vom Compiler als *konstant* erkannt, sind sie genau dann Instanzen derselben Klasse, wenn sie tatsächlich inhaltlich gleich sind. Dies liegt daran, daß String-Literale mit Hilfe der Methode [String.intern](#) angelegt werden, die einen Puffer von [String](#)-Objekten verwaltet, in dem jede Zeichenkette nur einmal vorkommt. Wird ein bereits existierender String noch einmal angelegt, so findet die Methode den Doppelgänger und liefert einen Zeiger darauf zurück. Dieses Verhalten ist so nur bei Strings zu finden, andere Objekte besitzen keine konstanten Werte und keine literalen Darstellungen. Die korrekte Methode, Strings auf inhaltliche Übereinstimmung zu testen, besteht darin, die Methode [equals](#) der Klasse [String](#) aufzurufen (siehe [Listing 5.7](#)).

Warnung

[Listing0507.java](#)

```
001 /* Listing0507.java */
002
003 public class Listing0507
004 {
005     public static void main(String[] args)
006     {
007         String a = new String("hallo");
008         String b = new String("hallo");
009         System.out.println("a.equals(b) liefert " + a.equals(b));
010     }
011 }
```

Listing 5.7: Vergleichen von Strings mit equals

Der instanceof-Operator

Der [instanceof](#)-Operator kann verwendet werden, um herauszufinden, zu welcher Klasse ein bestimmtes Objekt gehört. Der Ausdruck `a instanceof b` liefert genau dann `true`, wenn `a` und `b` Referenztypen sind und `a` eine Instanz der Klasse `b` oder einer ihrer Unterklassen ist. Falls das Ergebnis des [instanceof](#)-Operators nicht bereits zur Compile-Zeit ermittelt werden kann, generiert der Java-Compiler Code, um den entsprechenden Check zur Laufzeit durchführen zu können.

Der new-Operator

In Java werden Objekte und Arrays mit Hilfe des *new*-Operators erzeugt. Sowohl das Erzeugen eines Arrays als auch das Erzeugen eines Objekts sind Ausdrücke, deren Rückgabewert das gerade erzeugte Objekt bzw. Array ist.

Member-Zugriff

Der Zugriff auf Klassen- oder Instanzvariablen wird mit Hilfe des *Punkt*-Operators ausgeführt und hat die Form `a.b`. Dabei ist `a` der Name einer Klasse bzw. der Instanz einer Klasse, und `b` ist der Name einer Klassen- oder Instanzvariable. Der Typ des Ausdrucks entspricht dem Typ der Variable, und zurückgegeben wird der Inhalt dieser Variable.

Methodenaufruf

In Java gibt es keine Funktionen, sondern nur *Methoden*. Der Unterschied zwischen beiden besteht darin, daß Methoden immer an eine Klasse oder die Instanz einer Klasse gebunden sind und nur in diesem Kontext aufgerufen werden können. Die Syntax des Methodenaufrufs gleicht der anderer Programmiersprachen und erfolgt in der (etwas vereinfachten) Form `f()` bzw. `f(parameterliste)`. Der Typ des Ausdrucks entspricht dem vereinbarten Rückgabetyt der Methode. Der Wert des Ausdrucks ist der von der Methode mit Hilfe der `return`-Anweisung zurückgegebene Wert.

Methoden können selbstverständlich Nebeneffekte haben und werden in vielen Fällen ausschließlich zu diesem Zweck geschrieben. Ist dies der Fall, so sollte eine Methode als `void` deklariert werden und damit anzeigen, daß sie keinen Rückgabewert produziert. Die einzig sinnvolle Verwendung einer solchen Methode besteht darin, sie innerhalb einer *Ausdrucksanweisung* (siehe [Kapitel 6](#)) aufzurufen.

Da die Realisierung der Methodenaufrufe in Java recht kompliziert ist (die Sprachspezifikation [Hinweis](#) widmet diesem Thema mehr als 10 Seiten), werden wir in [Kapitel 7](#) noch einmal ausführlich darauf eingehen.

Zugriff auf Array-Elemente

Wie in anderen Programmiersprachen erfolgt auch in Java der Zugriff auf Array-Elemente mit Hilfe eckiger Klammern in der Form `a[b]` (bzw. `a[b][c]`, `a[b][c][d]` usw. bei mehrdimensionalen Arrays). Dabei ist `a` der Name eines Arrays oder ein Ausdruck, der zu einem Array ausgewertet wird, und `b` ein Ausdruck, der zu einem `int` evaluiert werden kann. Der Typ des Ausdrucks entspricht dem Basistyp des Arrays, zurückgegeben wird der Inhalt des Array-Elements, das sich an Position `b` befindet. Wie in C und C++ beginnt die Zählung mit dem ersten Element bei Position 0.

5.7.3 Welche Operatoren es nicht gibt

Da es in Java keine expliziten Pointer gibt, fehlen auch die aus C bekannten Operatoren `*` zur Dereferenzierung eines Zeigers und `&` zur Bestimmung der Adresse einer Variablen. Des weiteren fehlt ein `sizeof`-Operator, denn da alle Typen eine genau spezifizierte Länge haben, ist dieser überflüssig. Der Kommaoperator von C ist ebenfalls nicht vorhanden, er taucht aber als syntaktischer Bestandteil der `for`-Schleife wieder auf und erlaubt es, im Initialisierungsteil der Schleife mehr als eine Zuweisung vorzunehmen. [Hinweis](#)

5.8 Operator-Vorrangregeln

- 5.8 Operator-Vorrangregeln

[Tabelle 5.6](#) listet alle Operatoren in der Reihenfolge ihrer Vorrangregeln auf. Weiter oben stehende Operatoren haben dabei Vorrang vor weiter unten stehenden Operatoren. Innerhalb derselben Gruppe stehende Operatoren werden entsprechend ihrer Assoziativität ausgewertet.

Die Spalte *Typisierung* gibt die möglichen Operandentypen an. Dabei steht »N« für numerische, »I« für integrale (also ganzzahlig numerische), »L« für logische, »S« für String-, »R« für Referenz-, »P« für primitive und »A« für alle Typen.

Gruppe	Operator	Typisierung	Assoziativität	Bezeichnung
1	++	N	R	Inkrement
	--	N	R	Dekrement
	+	N	R	Unäres Plus
	-	N	R	Unäres Minus
	~	I	R	Einerkomplement
	!	L	R	Logisches NICHT
	(type)	A	R	Type-Cast
2	*	N,N	L	Multiplikation
	/	N,N	L	Division
	%	N,N	L	Modulo
3	+	N,N	L	Addition
	-	N,N	L	Subtraktion
	+	S,I	L	String-Verkettung
4	<<	I,I	L	Linksschieben
	>>	I,I	L	Rechtsschieben
	>>>	I,I	L	Rechtsschieben mit Nullexpansion
5	<	L,L	L	Kleiner
	<=	L,L	L	Kleiner gleich
	>	L,L	L	Größer
	>=	L,L	L	Größer gleich
	instanceof	R,R	L	Klassenzugehörigkeit
6	==	P,P	L	Gleich
	!=	P,P	L	Ungleich
	==	R,R	L	Referenzgleichheit
	!=	R,R	L	Referenzungleichheit
7	&	I,I	L	Bitweises UND
	&	L,L	L	Logisches UND mit vollständiger Auswertung
8	^	I,I	L	Bitweises EXKLUSIV-ODER
	^	L,L	L	Logisches EXKLUSIV-ODER
9		I,I	L	Bitweises ODER
		L,L	L	Logisches ODER mit vollständiger Auswertung
10	&&	L,L	L	Logisches UND mit Short-Circuit-Evaluation
11		L,L	L	Logisches ODER mit Short-Circuit-Evaluation
12	?:	L,A,A	R	Bedingte Auswertung
13	=	V,A	R	Zuweisung

	+=	V,N	R	Additionszuweisung
	-=	V,N	R	Subtraktionszuweisung
	*=	V,N	R	Multiplikationszuweisung
	/=	V,N	R	Divisionszuweisung
	%=	V,N	R	Restwertzuweisung
	&=	N,N u. L,L	R	Bitweises-UND-Zuweisung und Logisches-UND-Zuweisung
	=	N,N u. L,L	R	Bitweises-ODER-Zuweisung und Logisches-ODER-Zuweisung
	^=	N,N u. L,L	R	Bitweises-EXKLUSIV-ODER-Zuweisung und Logisches-EXKLUSIV-ODER-Zuweisung
	<<=	V,I	R	Linksschiebezuweisung
	>>=	V,I	R	Rechtsschiebezuweisung
	>>>=	V,I	R	Rechtsschiebezuweisung mit Nullexpansion

Tabelle 5.6: Operator-Vorrangregeln

Etwas unschön ist die Tatsache, daß die bitweisen Operatoren schwächer binden als die relationalen Operatoren. Da sie auf einer Stufe mit den zugehörigen logischen Operatoren stehen, gibt es beim Übersetzen des folgenden Programms den Fehler »Incompatible type for &. Can't convert int to boolean«:

Warnung

```
001 /* Listing0508.java */
002
003 public class Listing0508
004 {
005     public static void main(String[] args)
006     {
007         int i = 55;
008         int j = 97;
009         if (i & 15 < j & 15) {
010             System.out.println("LowByte(55) < LowByte(97)");
011         } else {
012             System.out.println("LowByte(55) >= LowByte(97)");
013         }
014     }
015 }
```

Listing 5.8: Bindungsprobleme bei den bitweisen Operatoren

Bei der Verwendung der bitweisen Operatoren sind also zusätzliche Klammern erforderlich. Die korrekte Version des Programms zeigt [Listing 5.9](#) (verbessert wurde [Zeile 009](#)):

[Listing0509.java](#)

```
001 /* Listing0509.java */
002
003 public class Listing0509
004 {
005     public static void main(String[] args)
006     {
007         int i = 55;
008         int j = 97;
009         if ((i & 15) < (j & 15)) {
010             System.out.println("LowByte(55) < LowByte(97)");
011         } else {
012             System.out.println("LowByte(55) >= LowByte(97)");
013         }
014     }
015 }
```

Listing 5.9: Korrekte Klammerung von bitweisen Operatoren

Die Ausgabe des Programms ist nun erwartungsgemäß:

LowByte(55) >= LowByte(97)

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

5.9 Zusammenfassung

- [5.9 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Grundlegende Eigenschaften von Ausdrücken
- Die arithmetischen Operatoren +, -, *, /, %, ++ und --
- Die relationalen Operatoren ==, !=, <, <=, > und >=
- Die logischen Operatoren !, &&, ||, &, | und ^
- Die bitweisen Operatoren ~, |, &, ^, >>, >>> und <<
- Die Zuweisungsoperatoren =, +=, -=, *=, /=, %, &=, |=, ^=, <<=, >>= und >>>=
- Der Fragezeichenoperator ?:, der Operator für Typumwandlungen und der String-Verkettungs-Operator +
- Die Operatoren [new](#) und [instanceof](#)
- Die Operatoren für Member- und Array-Zugriff und der Operator zum Aufrufen von Methoden
- Die Operator-Vorrangregeln von Java

Kapitel 6

Anweisungen

- [6 Anweisungen](#)
 - [6.1 Elementare Anweisungen](#)
 - [6.1.1 Die leere Anweisung](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.1.2 Der Block](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.1.3 Variablendeklarationen](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.1.4 Ausdrucksanweisungen](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.2 Verzweigungen](#)
 - [6.2.1 Die if-Anweisung](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [Dangling else](#)
 - [Bedingtes Kompilieren](#)
 - [6.2.2 Die switch-Anweisung](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.3 Schleifen](#)
 - [6.3.1 Die while-Schleife](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.3.2 Die do-Schleife](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.3.3 Die for-Schleife](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [break und continue](#)
 - [6.4 Zusammenfassung](#)

6.1 Elementare Anweisungen

- [6.1 Elementare Anweisungen](#)
 - [6.1.1 Die leere Anweisung](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.1.2 Der Block](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.1.3 Variablendeklarationen](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.1.4 Ausdrucksanweisungen](#)
 - [Syntax](#)
 - [Bedeutung](#)

6.1.1 Die leere Anweisung

Syntax

```
;
```

Bedeutung

Die einfachste Anweisung in Java ist die leere Anweisung. Sie besteht nur aus dem Semikolon und hat keinerlei Effekt auf das laufende Programm. Eine leere Anweisung kann da verwendet werden, wo syntaktisch eine Anweisung erforderlich ist, aber von der Programmlogik her *nichts* zu tun ist.

6.1.2 Der Block

Syntax

```
{
    Anweisung1;
    Anweisung2;
    ...
}
```

Bedeutung

Ein Block ist eine Zusammenfassung von Anweisungen, die nacheinander ausgeführt werden. Alle im Block zusammengefaßten Anweisungen gelten in ihrer Gesamtheit als *eine einzelne* Anweisung und dürfen überall da verwendet werden, wo syntaktisch eine einzelne elementare Anweisung erlaubt wäre. Ein Block darf auch Anweisungen zur Deklaration von lokalen Variablen haben. Diese sind dann nur innerhalb des Blocks gültig und sichtbar.

6.1.3 Variablendeklarationen

Syntax

```
Typname VariablenName;
```

oder

```
Typname VariablenName = InitialerWert;
```

Bedeutung

Die Deklaration einer lokalen Variable gilt in Java als ausführbare Anweisung. Sie darf daher überall dort erfolgen, wo eine Anweisung verwendet werden darf. Die Sichtbarkeit einer lokalen Variable erstreckt sich von der Deklaration bis zum Ende des umschließenden Blocks.

Lokale Variablen dürfen sich nicht gegenseitig verdecken. Es ist also nicht erlaubt, eine bereits deklarierte Variable `x` in einem tiefer geschachtelten Block erneut zu deklarieren. Das Verdecken von Klassen- oder Instanzvariablen dagegen ist zulässig und wird besonders häufig in Konstruktoren bei der Initialisierung von Instanzvariablen verwendet:

Hinweis

Beispiel

```
001 class Punkt
002 {
003     int x;
004     int y;
005
006     Punkt(int x, int y)
007     {
008         this.x = x;
009         this.y = y;
010     }
011 }
```

Listing 6.1: Verdecken von Klassen- oder Instanzvariablen

Hier wird eine Klasse `Punkt` definiert, die zwei Instanzvariablen `x` und `y` enthält. Der Konstruktor initialisiert die Instanzvariablen mit den gleichnamigen Parametern. Um nun nicht für die formalen Parameter neue Namen erfinden zu müssen, lassen sich durch lokale Variablen verdeckte Instanzvariablen dadurch wieder aufdecken, daß sie mit dem `this`-Pointer qualifiziert werden.

Das vorliegende Beispiel erscheint zum jetzigen Zeitpunkt sicherlich schwer verständlich, denn bisher wurden weder Klassen noch Methoden noch Konstruktoren eingeführt. Nach Lektüre von [Kapitel 7](#), das sich mit den objektorientierten Eigenschaften von Java beschäftigt, wird es verständlicher sein. Einprägen sollte man sich diesen Stil aber jetzt schon, denn er wird in Java häufig verwendet.

6.1.4 Ausdrucksanweisungen

Syntax

Ausdruck;

Bedeutung

Ausdrucksanweisungen dienen dazu, Ausdrücke in einem Anweisungskontext auszuführen. Sie werden erzeugt, indem an den Ausdruck ein Semikolon angehängt wird. Da der Rückgabewert des Ausdrucks dabei ignoriert wird, macht eine Ausdrucksanweisung nur dann Sinn, wenn der Ausdruck Nebeneffekte hat. Tatsächlich ist es in Java so, daß innerhalb einer Ausdrucksanweisung nur ganz bestimmte Ausdrücke auftauchen dürfen:

- Zuweisung
- Inkrement und Dekrement
- Methodenaufruf
- Instanzerzeugung

Während es in C beispielsweise möglich ist, den Ausdruck `2+4` in eine Ausdrucksanweisung `2+4;` zu verpacken, ist dies in Java nicht erlaubt. Eine solche Anweisung ist aber auch wenig sinnvoll, und es war daher nur vernünftig, dies in Java nicht zuzulassen.

6.2 Verzweigungen

- 6.2 Verzweigungen
 - 6.2.1 Die if-Anweisung
 - Syntax
 - Bedeutung
 - Dangling else
 - Bedingtes Kompilieren
 - 6.2.2 Die switch-Anweisung
 - Syntax
 - Bedeutung

Verzweigungen in Java dienen wie in allen Programmiersprachen dazu, bestimmte Programmteile nur beim Eintreten vorgegebener Bedingungen, die erst zur Laufzeit bekannt werden, auszuführen. An Verzweigungen bietet Java die `if`- und `if-else`-Anweisung sowie die `switch`-Anweisung.

6.2.1 Die if-Anweisung

Syntax

```
if (ausdruck)
    anweisung;
```

oder

```
if (ausdruck)
    anweisung1;
else
    anweisung2;
```

Bedeutung

Die `if`-Anweisung wertet zunächst den Ausdruck `ausdruck` aus. Danach führt sie die Anweisung `anweisung` genau dann aus, wenn das Ergebnis des Ausdrucks `true` ist. Ist `ausdruck` hingegen `false`, so wird die Anweisung nicht ausgeführt, sondern mit der ersten Anweisung nach der `if`-Anweisung fortgefahren.

Mit der `if-else`-Anweisung gibt es eine weitere Verzweigung in Java. Falls `ausdruck` wahr ist, wird `anweisung1` ausgeführt, andernfalls `anweisung2`. Eine der beiden Anweisungen wird also in jedem Fall ausgeführt.

Anstelle einer einzelnen Anweisung kann jeweils auch eine Folge von Anweisungen angegeben werden, wenn sie innerhalb eines Blocks steht. Dieser wird als Einheit betrachtet und komplett ausgeführt, wenn die entsprechende Bedingung zutrifft.

Zu beachten ist, daß der Testausdruck in der Schleife vom Typ `boolean` sein muß. Anders als in C ist es in Java nicht erlaubt, einen *numerischen* Ausdruck an seiner Stelle zu verwenden.

Hinweis

Dangling else

Eine der Mehrdeutigkeiten, die in fast allen blockstrukturierten Programmiersprachen auftauchen können, wurde auch von den Java-Entwicklern nicht beseitigt. Als Beispiel wollen wir uns das folgende Codefragment ansehen, das leider nicht so ausgeführt wird, wie es die Einrückung erwarten läßt:

Warnung

```

001 if (a)
002     if (b)
003         s1;
004 else
005     s2;

```

Listing 6.2: Dangling else

Der [else](#)-Zweig gehört zu der *innersten* Verzweigung `if (b)...`, und die korrekte Einrückung würde lauten:

```

001 if (a)
002     if (b)
003         s1;
004     else
005         s2;

```

Listing 6.3: Dangling else, ausgeschaltet

Dieses Problem ist in der Literatur unter dem Namen *dangling else* bekannt und kann nur auftauchen, wenn eine [if](#)- und eine [if-else](#)-Verzweigung ineinander geschachtelt werden und beide Anweisungen nicht durch Blockklammern begrenzt wurden. Um die Mehrdeutigkeit zu beseitigen, wird in Java wie auch in C oder C++ ein »freies« [else](#) immer an das am weitesten innen liegende [if](#) angehängt.

Bedingtes Kompilieren

Eine weitere Besonderheit der Verzweigungen in Java rührt daher, daß die Sprache keinen Präprozessor besitzt und daher kein `#ifdef` kennt. Um eine eingeschränkte Form der bedingten Kompilierung zu verwirklichen, wird in Java das folgende Programmfragment in der Weise kompiliert, daß die Anweisung [anweisung](#) nicht mitübersetzt wird, da der Testausdruck konstant [false](#) ist:

Tip

```

001 if (false)
002     anweisung

```

Listing 6.4: Bedingtes Kompilieren

Allerdings sollte man hinzufügen, daß ein solches Verhalten in der Sprachspezifikation zwar dringend empfohlen wird, für die Compiler-Bauer aber nicht zwangsläufig verpflichtend ist.

Das hier beschriebene Verhalten eines Java-Compilers steht im Widerspruch zu einer anderen Forderung, nämlich der, nur *erreichbare Anweisungen* zu akzeptieren. Gemäß Sprachspezifikation soll der Compiler alle Anweisungen, die *nicht erreichbar* sind, ablehnen, also einen Fehler melden.

Nicht erreichbar im technischen Sinne sind dabei Anweisungen in Schleifen, deren Testausdruck zur Compile-Zeit [false](#) ist, und Anweisungen, die hinter einer [break](#)-, [continue](#)-, [throw](#)- oder [return](#)-Anweisung liegen, die *unbedingt* angesprungen wird. Die einzige Ausnahme von dieser Regel ist die im vorigen Absatz erwähnte Variante der konstant unwahren Verzweigung, die zur bedingten Kompilierung verwendet werden kann.

6.2.2 Die switch-Anweisung

Syntax

```

switch (ausdruck)
{
    case constant:
        anweisung;
    ...
    default:
}

```

Bedeutung

Die [switch](#)-Anweisung ist eine Mehrfachverzweigung. Zunächst wird der Ausdruck [ausdruck](#), der vom Typ [byte](#), [short](#), [char](#) oder [int](#) sein muß, ausgewertet. In Abhängigkeit vom Ergebnis wird dann die Sprungmarke angesprungen, deren Konstante mit dem Ergebnis des Ausdrucks übereinstimmt. Die Konstante und der Ausdruck müssen dabei zuweisungskompatibel sein.

Das optionale [default](#)-Label wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird. Ist kein [default](#)-Label vorhanden und wird auch keine passende Sprungmarke gefunden, so wird keine der Anweisungen innerhalb der [switch](#)-Anweisung ausgeführt. Jede Konstante eines [case](#)-Labels darf nur einmal auftauchen. Das [default](#)-Label darf maximal einmal verwendet werden.

Nachdem ein `case`- oder `default`-Label angesprungen wurde, werden alle dahinter stehenden Anweisungen ausgeführt. Im Gegensatz zu Sprachen wie PASCAL erfolgt auch dann keine Unterbrechung, wenn das nächste Label erreicht wird. Wenn dies erwünscht ist, muß der Kontrollfluß wie in C und C++ mit Hilfe einer `break`-Anweisung unterbrochen werden. Jedes `break` innerhalb einer `switch`-Anweisung führt dazu, daß zum Ende der `switch`-Anweisung verzweigt wird.

Warnung

Wie aus den bisherigen Ausführungen deutlich wurde, ist die Semantik der `switch`-Anweisung in Java der in C und C++ sehr ähnlich. Ein wichtiger Unterschied besteht darin, daß in Java alle Anweisungen, die unmittelbar innerhalb des `switch` liegen, `case`- oder `default`-Labels sein müssen. Der Trick, in `switch`-Anweisungen Schleifen zu packen, die sich über mehrere Labels erstrecken, funktioniert in Java nicht. Die Sprachspezifikation erläutert dies am Beispiel von *Duff's Device*, das so in Java nicht kompilierbar ist:

```
001 int q = (n+7)/8;
002 switch (n%8) {
003 case 0: do { foo();
004 case 1:      foo();
005 case 2:      foo();
006 case 3:      foo();
007 case 4:      foo();
008 case 5:      foo();
009 case 6:      foo();
010 case 7:      foo();
011           } while (--q >= 0);
012 }
```

Listing 6.5: *Duff's Device*

Glücklicherweise ist derartige Code auch mehr zur Verwirrung ahnungsloser Programmierer geeignet als zur ernsthaften Anwendung und kommt in der Praxis normalerweise kaum vor.

6.3 Schleifen

- [6.3 Schleifen](#)
 - [6.3.1 Die while-Schleife](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.3.2 Die do-Schleife](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [6.3.3 Die for-Schleife](#)
 - [Syntax](#)
 - [Bedeutung](#)
 - [break und continue](#)

Java besitzt die drei üblichen Schleifenanweisungen prozeduraler Programmiersprachen: eine nichtabweisende, eine abweisende und eine Zählschleife. Die Syntax und Semantik der Schleifen ist fast vollständig identisch zu den entsprechenden Anweisungen in C.

6.3.1 Die while-Schleife

Syntax

```
while (ausdruck)
    anweisung;
```

Bedeutung

Zuerst wird der Testausdruck, der vom Typ [boolean](#) sein muß, geprüft. Ist er [true](#), wird die Anweisung ausgeführt, andernfalls wird mit der ersten Anweisung hinter der Schleife weitergemacht. Nachdem die Anweisung ausgeführt wurde, wird der Testausdruck erneut geprüft usw. Die Schleife wird beendet, sobald der Test [false](#) ergibt.

6.3.2 Die do-Schleife

Syntax

```
do
    anweisung;
while (ausdruck);
```

Bedeutung

Die [do](#)-Schleife arbeitet *nichtabweisend*, d.h., sie wird mindestens einmal ausgeführt. Da zunächst die Schleifenanweisung ausgeführt und erst dann der Testausdruck überprüft wird, kann die [do](#)-Schleife frühestens nach einem Durchlauf regulär beendet werden. Die Bearbeitung der Schleife wird immer dann beendet, wenn der Test des Schleifenausdrucks [false](#) ergibt.

6.3.3 Die for-Schleife

Syntax

```
for (init; test; update)
    anweisung;
```

Bedeutung

Der Kopf der `for`-Schleife besteht aus drei Ausdrücken, die jeder für sich optional sind:

- Der `init`-Ausdruck wird einmal vor dem Start der Schleife aufgerufen. Er dient dazu, Initialisierungen durchzuführen, die durch die Auswertung von Ausdrücken mit Nebeneffekten verursacht werden. Der Rückgabewert der Ausdrücke wird vollständig ignoriert.
- Der `init`-Teil darf auch aus mehreren Ausdrücken bestehen, wenn die einzelnen Teilausdrücke durch Kommata getrennt sind. Diese syntaktische Erweiterung ist allerdings nur innerhalb des Initialisierungsteils einer `for`-Schleife erlaubt. Einen allgemeinen Komma-Operator (wie in C und C++) gibt es in Java nicht.
- Fehlt der Initialisierungsteil, wird keine Initialisierung im Kopf der Schleife durchgeführt.
- Der `init`-Teil darf auch Variablendeklarationen enthalten, beispielsweise, um einen Schleifenzähler zu erzeugen. Die Variablen müssen bei der Deklaration initialisiert werden. Sichtbarkeit und Lebensdauer erstrecken sich auf den Block, der die Schleifenanweisungen enthält. Damit ist es möglich, den Namen einer Schleifenvariablen innerhalb einer Methode mehrfach zu deklarieren.
- Der `test`-Teil bildet den Testausdruck der Schleife. Analog zur `while`-Schleife wird er am Anfang der Schleife ausgeführt, und die Schleifenanweisung wird nur ausgeführt, wenn die Auswertung des Testausdrucks `true` ergibt. Fehlt der Testausdruck, so setzt der Compiler an seiner Stelle die Konstante `true` ein.
- Der `update`-Ausdruck dient dazu, den Schleifenzähler zu verändern. Er wird nach jedem Durchlauf der Schleife ausgewertet, bevor der Testausdruck das nächste Mal ausgewertet wird. Im Gegensatz zum `init`-Ausdruck darf der `update`-Ausdruck nicht aus mehreren Komponenten bestehen. Der Rückgabewert des Ausdrucks wird ignoriert. Fehlt der `update`-Ausdruck, so wird keine automatische Modifikation des Schleifenzählers durchgeführt.

break und continue

In Java gibt es zwei weitere Möglichkeiten, die normale Auswertungsreihenfolge in einer Schleife zu verändern. Taucht innerhalb einer Schleife eine `break`-Anweisung auf, wird die Schleife verlassen und das Programm mit der ersten Anweisung nach der Schleife fortgesetzt. Taucht dagegen eine `continue`-Anweisung auf, springt das Programm an das Ende des Schleifenrumpfs und beginnt mit der nächsten Iteration.

In der einfachsten Form arbeiten `break` und `continue` genauso wie in C und C++. Beide Anweisungen können innerhalb von `do`-, `while`- und `for`-Schleifen verwendet werden. Befindet sich ein `break` innerhalb einer mehrfach geschachtelten Schleife, so verläßt es die innerste Schleife. Dies gilt analog für `continue`. Neben dieser einfachen Form von `break` und `continue` gibt es in Java noch die mit einem Label versehene Form:

```
break Label;  
  
continue Label;
```

Zu jedem mit einem Label versehenen `break` oder `continue` muß es eine mit einem Label versehene Kontrollstruktur geben, die diese Anweisung umschließt. Üblicherweise wird ein mit einem Label versehenes `break` verwendet, um zwei oder mehr geschachtelte Schleifen zu beenden:

```
001 loop1: for (int i=0; i<100; ++i) {  
002     for (int j=0; j<100; ++j) {  
003         if (A[i][j] > 42) {  
004             break;  
005         }  
006         if (A[i][j] == null) {  
007             break loop1;  
008         }  
009     }  
010 }
```

Beispiel

Listing 6.6: Das gelabelte break

Dieses Programm enthält zwei geschachtelte `for`-Schleifen. Die äußere von beiden hat das Label `loop1`, und die innere ist ohne Label. Die erste `break`-Anweisung hat kein Label und verläßt daher nur die innere der beiden Schleifen. Das Programm fährt anschließend mit der nächsten Iteration der äußeren Schleife fort. Die zweite Form der `break`-Anweisung enthält das Label `loop1` und verläßt daher die Anweisung mit eben diesem Label, in diesem Fall also die äußere `for`-Schleife.

Vollkommen analog zu der mit einem Label versehenen `break`-Anweisung kann auch die mit einem Label versehene Form der `continue`-Anweisung verwendet werden, um die nächste Iteration einer weiter außen liegenden Schleife einzuleiten:

```

001 loop1: for (int i=0; i<100; ++i) {
002     for (int j=0; j<100; ++j) {
003         if (A[i][j] > 42) {
004             continue;
005         }
006         if (A[i][j] == null) {
007             continue loop1;
008         }
009     }
010 }

```

Listing 6.7: Das gelabelte *continue*

Die [break](#)- und [continue](#)-Anweisungen, insbesondere die mit einem Label versehenen, stellen *Sprunganweisungen* dar. Allerdings sind ihre Möglichkeiten im Vergleich zum [goto](#) anderer Programmiersprachen auf kontrollierte Sprünge in Schleifenanweisungen beschränkt. Eine *allgemeine* Sprunganweisung gibt es in Java nicht (obwohl die Entwickler sich ein Hintertürchen offengelassen haben, indem sie [goto](#) zu einem reservierten Wort erklärt haben).

Die Bedeutung von gelabelten [break](#)- und [continue](#)-Anweisungen ist in der Praxis nicht so groß, wie man vermuten könnte. Unbedacht eingesetzt, können sie die Lesbarkeit eines Programmes sogar vermindern. Eine nützliche Anwendung des gelabelten [break](#) besteht darin, eine Schleife aus einer darin liegenden [switch](#)-Anweisung verlassen zu können.

6.4 Zusammenfassung

- [6.4 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die leere Anweisung
- Blöcke von Anweisungen
- Variablendeklarationen und die Bedeutung lokaler Variablen
- Ausdrucksanweisungen
- Die [if](#)-Anweisung
- Die [switch](#)-Anweisung
- Die [while](#)-, [do](#)- und [for](#)-Schleifen
- Die Anweisungen [break](#) und [continue](#) sowie gelabelte Schleifen

Kapitel 7

Objektorientierte Programmierung

- [7 Objektorientierte Programmierung](#)
 - [7.1 Klassen und Objekte](#)
 - [7.1.1 Klassen](#)
 - [7.1.2 Objekte](#)
 - [7.2 Methoden](#)
 - [7.2.1 Definition](#)
 - [7.2.2 Aufruf](#)
 - [7.2.3 Parameter](#)
 - [7.2.4 Rückgabewert](#)
 - [7.2.5 Überladen von Methoden](#)
 - [Die Signatur einer Methode](#)
 - [7.2.6 Konstruktoren](#)
 - [Default-Konstruktoren](#)
 - [Verkettung von Konstruktoren](#)
 - [7.2.7 Destruktoren](#)
 - [7.3 Vererbung](#)
 - [7.3.1 Ableiten einer Klasse](#)
 - [7.3.2 Methodenüberlagerung](#)
 - [Dynamische Methodensuche](#)
 - [Aufrufen einer verdeckten Methode](#)
 - [7.3.3 Vererbung von Konstruktoren](#)
 - [Konstruktorenverkettung](#)
 - [Der default-Konstruktor](#)
 - [Überlagerte Konstruktoren](#)
 - [Destruktorenverkettung](#)
 - [7.4 Attribute von Klassen, Methoden und Variablen](#)
 - [7.4.1 Sichtbarkeit](#)
 - [7.4.2 Die Attribute im Überblick](#)
 - [private](#)
 - [protected](#)
 - [public](#)
 - [static](#)
 - [final](#)
 - [7.5 Klassen mit static-Elementen](#)
 - [7.5.1 Klassenvariablen](#)
 - [7.5.2 Konstanten](#)
 - [7.5.3 Klassenmethoden](#)
 - [7.5.4 Statische Konstruktoren](#)
 - [7.6 Abstrakte Klassen und Methoden](#)
 - [7.7 Interfaces](#)
 - [7.7.1 Definition eines Interfaces](#)

- [7.7.2 Verwendung von Interfaces](#)
- [7.7.3 Implementieren mehrerer Interfaces](#)
- [7.7.4 Interfaces sind Klassen](#)
- [7.7.5 Konstanten in Interfaces](#)
- [7.7.6 Beispiele aus der Java-Klassenbibliothek](#)
 - [Runnable](#)
 - [Enumeration](#)
 - [Cloneable](#)
- [7.8 Spezielle Klassen](#)
 - [7.8.1 Die Wrapper-Klassen](#)
 - [Instanziierung](#)
 - [Rückgabe des Wertes](#)
 - [Parsen von Strings](#)
 - [Konstanten](#)
 - [7.8.2 Die Klasse java.lang.Math](#)
 - [Winkelfunktionen](#)
 - [Minimum und Maximum](#)
 - [Arithmetik](#)
 - [Runden und Abschneiden](#)
- [7.9 Zusammenfassung](#)

7.1 Klassen und Objekte

- [7.1 Klassen und Objekte](#)
 - [7.1.1 Klassen](#)
 - [7.1.2 Objekte](#)

7.1.1 Klassen

Eine Klassendefinition in Java wird durch das Schlüsselwort `class` eingeleitet. Anschließend folgt innerhalb von geschweiften Klammern eine beliebige Anzahl an Variablen- und Methodendefinitionen. Das folgende Listing ist ein Beispiel für eine einfache Klassendefinition:

Beispiel

```

001 public class Auto
002 {
003     public String name;
004     public int    erstzulassung;
005     public int    leistung;
006 }
    
```

Listing 7.1: Eine einfache Klassendefinition

Diese Klasse enthält keine Methoden, sondern lediglich die Variablen `name`, `erstzulassung` und `leistung`. Eine solche methodenlose Klassendefinition entspricht dem Konzept des Verbunddatentyps aus C oder PASCAL (`struct` bzw. `record`). Die innerhalb einer Klasse definierten Variablen wollen wir im folgenden (analog zu C++) als *Membervariablen* bezeichnen.

7.1.2 Objekte

Um von einer Klasse ein Objekt anzulegen, muß eine Variable vom Typ der Klasse deklariert und ihr mit Hilfe des `new`-Operators ein neu erzeugtes Objekt zugewiesen werden:

```

001 Auto meinKombi;
002 meinKombi = new Auto();
    
```

Listing 7.2: Erzeugen eines Objekts mit new

Die erste Anweisung ist eine normale Variablendeklaration, wie sie aus [Kapitel 4](#) bekannt ist. Anstelle eines primitiven Typs wird hier der Typname einer zuvor definierten Klasse verwendet. Im Unterschied zu primitiven Datentypen wird die Objektvariable `meinKombi` als *Referenz* gespeichert. Die zweite Anweisung generiert mit Hilfe des `new`-Operators eine neue Instanz der Klasse `Auto` und weist sie der Variablen `meinKombi` zu.

In Java wird jede *selbstdefinierte* Klasse mit Hilfe des `new`-Operators instanziiert. Mit Ausnahme von Strings und Arrays, bei denen der Compiler auch *Literale* zur Objekterzeugung zur Verfügung stellt, gilt dies auch für alle vordefinierten Klassen der Java-Klassenbibliothek.

Wie bei primitiven Variablen lassen sich beide Anweisungen auch kombinieren. Das nachfolgende

Tip

 Beispiel deklariert und initialisiert die Variable `meinKombi`:

```

001 Auto meinKombi = new Auto();
    
```

Listing 7.3: Kombinierte Deklaration und Initialisierung einer Objektvariablen

Nach der Initialisierung haben alle Variablen des Objekts zunächst Standardwerte. Referenztypen haben den Standardwert `null`, die Standardwerte der primitiven Typen können [Tabelle 4.1](#) entnommen werden. Der Zugriff auf sie erfolgt mit Hilfe der Punktnotation `Objekt.Variable`. Um unser `Auto`-Objekt in einen 250 PS starken Mercedes 600 des Baujahrs 1972 zu verwandeln, müßten folgende Anweisungen ausgeführt werden:

```

001 meinKombi.name = "Mercedes 600";
002 meinKombi.erstzulassung = 1972;
003 meinKombi.leistung = 250;
    
```

Listing 7.4: Zuweisen von Werten an die Variablen eines Objekts

Ebenso wie der schreibende erfolgt auch der lesende Zugriff mit Hilfe der Punktnotation. Die Ausgabe des aktuellen Objektes auf dem Bildschirm könnte also mit den folgenden Anweisungen erledigt werden:

```
001 System.out.println("Name.....: "+meinKombi.name);
002 System.out.println("Zugelassen..: "+meinKombi.erstzulassung);
003 System.out.println("Leistung....: "+meinKombi.leistung);
```

Listing 7.5: Lesender Zugriff auf die Variablen eines Objekts

7.2 Methoden

- [7.2 Methoden](#)
 - [7.2.1 Definition](#)
 - [7.2.2 Aufruf](#)
 - [7.2.3 Parameter](#)
 - [7.2.4 Rückgabewert](#)
 - [7.2.5 Überladen von Methoden](#)
 - [Die Signatur einer Methode](#)
 - [7.2.6 Konstruktoren](#)
 - [Default-Konstruktoren](#)
 - [Verkettung von Konstruktoren](#)
 - [7.2.7 Destruktoren](#)

7.2.1 Definition

Methoden definieren das *Verhalten* von Objekten. Sie werden innerhalb einer Klassendefinition angelegt und haben Zugriff auf alle Variablen des Objekts. Methoden sind das Pendant zu den *Funktionen* anderer Programmiersprachen, arbeiten aber immer mit den Variablen des aktuellen Objekts. *Globale Funktionen*, die vollkommen unabhängig von einem Objekt oder einer Klasse existieren, gibt es in Java ebensowenig wie globale Variablen. Wir werden später allerdings Klassenvariablen und -methoden kennenlernen, die nicht an eine konkrete Instanz gebunden sind.

Die Syntax der Methodendefinition in Java ähnelt der von C:

```
{Modifier}
Typ Name([Parameter])
{
    {Anweisung;}
}
```

Nach einer Reihe von *Modifiern* (wir kommen weiter unten in [Abschnitt 7.4](#) darauf zurück) folgen der *Typ* des Rückgabewerts der Funktion, ihr *Name* und eine optionale *Parameterliste*. In geschweiften Klammern folgt dann der *Methodenrumpf*, also die Liste der Anweisungen, die das Verhalten der Methode festlegen. Die Erweiterung unserer Beispielklasse um eine Methode zur Berechnung des Alters des [Auto](#)-Objekts würde beispielsweise so aussehen:

```
001 public class Auto
002 {
003     public String name;
004     public int    erstzulassung;
005     public int    leistung;
006
007     public int alter()
008     {
009         return 1999 - erstzulassung;
010     }
011 }
```

Beispiel

Listing 7.6: Eine einfache Methode zur Altersberechnung

Hier wird eine Methode `alter` definiert, die einen ganzzahligen Wert zurückgibt, der sich aus der Differenz von 1999 und dem Jahr der Erstzulassung errechnet.

7.2.2 Aufruf

Der Aufruf einer Methode erfolgt ähnlich der Verwendung einer Instanzvariablen in Punktnotation. Zur Unterscheidung von einem Variablenzugriff müssen zusätzlich die Parameter der Methode in Klammern angegeben werden, selbst wenn die Liste leer ist. Das folgende Programm würde demnach die Zahl 9 auf dem Bildschirm ausgeben.

```

001 Auto golf1 = new Auto();
002 golf1.erstzulassung = 1990;
003 System.out.println(golf1.alter());

```

Listing 7.7: Aufruf einer Methode

Wie an der Definition von `alter` zu erkennen ist, darf eine Methode auf die Instanzvariablen ihrer Klasse zugreifen, ohne die Punktnotation zu verwenden. Das funktioniert deshalb, weil der Compiler alle nicht in Punktnotation verwendeten Variablen `x`, die nicht lokale Variablen sind, auf das Objekt `this` bezieht und damit als `this.x` interpretiert.

Tip

Bei `this` handelt es sich um einen Zeiger, der beim Anlegen eines Objekts automatisch generiert wird. `this` ist eine Referenzvariable, die auf das aktuelle Objekt zeigt und dazu verwendet wird, die eigenen Methoden und Instanzvariablen anzusprechen. Der `this`-Zeiger ist auch *explizit* verfügbar und kann wie eine ganz normale Objektvariable verwendet werden. Er wird als versteckter Parameter an jede nicht-statische Methode übergeben. Die Methode `alter` hätte also auch so geschrieben werden können:

```

001 public int alter()
002 {
003     return 1999 - this.erstzulassung;
004 }

```

Listing 7.8: Verwendung von `this`

Es ist tatsächlich manchmal sinnvoll, `this` explizit zu verwenden, wenn hervorgehoben werden soll, daß es sich um den Zugriff auf eine Instanzvariable handelt.

Tip

7.2.3 Parameter

Eine Methode kann mit Parametern definiert werden. Dazu wird bei der Methodendefinition eine Parameterliste innerhalb der Klammern angegeben. Jeder formale Parameter besteht aus einem Typnamen und dem Namen des Parameters. Soll mehr als ein Parameter definiert werden, so sind die einzelnen Definitionen durch Kommata zu trennen.

Alle Parameter werden in Java per *call by value* übergeben. Beim Aufruf einer Methode wird also der aktuelle Wert in die Parametervariable kopiert und an die Methode übergeben. Veränderungen der Parametervariablen innerhalb der Methode bleiben lokal und wirken sich nicht auf den Aufrufer aus. Das folgende Beispiel definiert eine Methode `printAlter`, die das Alter des Autos insgesamt `wieoft` mal auf dem Bildschirm ausgibt:

```

001 public void printAlter(int wieoft)
002 {
003     while (wieoft-- > 0) {
004         System.out.println("Alter = " + alter());
005     }
006 }

```

Listing 7.9: Eine Methode zur Ausgabe des Alters

Obwohl der Parameter `wieoft` innerhalb der Methode verändert wird, merkt ein Aufrufer nichts von diesen Änderungen, da innerhalb der Methode mit einer Kopie gearbeitet wird. Das folgende Programm würde das Alter des Objekts `auto` daher insgesamt neunmal auf dem Bildschirm ausgeben:

```

001 ...
002 int a = 3;
003
004 auto.printAlter(a);
005 auto.printAlter(a);
006 auto.printAlter(a);
007 ...

```

Listing 7.10: Wiederholter Aufruf der Methode zur Ausgabe des Alters

Wie bereits erwähnt, sind Objektvariablen Referenzen, also Zeiger. Zwar werden auch sie bei der Übergabe an eine Methode per Wert übergeben, da innerhalb der Methode aber der Zeiger auf das Originalobjekt zur Verfügung steht (wenn auch in kopierter Form), sind Veränderungen an dem Objekt natürlich auch für den Aufrufer der Methode sichtbar. Wie in allen anderen Programmiersprachen entspricht die *call by value*-Übergabe eines *Zeigers* damit natürlich genau der Semantik von *call by reference*.

Die Übergabe von Objekten an Methoden hat damit zwei wichtige Konsequenzen:

- Die Methode erhält keine Kopie, sondern arbeitet mit dem Originalobjekt.
- Die Übergabe von Objekten ist performant, gleichgültig wie groß sie sind.

Hinweis

Sollen Objekte kopiert werden, so muß dies explizit durch Aufruf der Methode `clone` der Klasse `Object` erfolgen.

7.2.4 Rückgabewert

Jede Methode in Java ist typisiert. Der Typ einer Methode wird zum Zeitpunkt der Definition festgelegt und bestimmt den Typ des Rückgabewerts. Dieser kann von einem beliebigen primitiven Typ, einem Objekttyp oder vom Typ `void` sein. Die Methoden vom Typ `void` haben gar keinen Rückgabewert und dürfen nicht in Ausdrücken verwendet werden. Sie sind lediglich wegen ihrer Nebeneffekte von Interesse und dürfen daher nur als Ausdrucksanweisung verwendet werden.

Hat eine Methode einen Rückgabewert (ist also nicht vom Typ `void`), so kann sie mit Hilfe der `return`-Anweisung einen Wert an den Aufrufer zurückgeben. Die `return`-Anweisung hat folgende Syntax:

```
return Ausdruck;
```

Wenn diese Anweisung ausgeführt wird, führt dies zum Beenden der Methode, und der Wert des angegebenen Ausdrucks wird an den Aufrufer zurückgegeben. Der Ausdruck muß dabei zuweisungskompatibel zum Typ der Funktion sein. Die in [Kapitel 5](#) erläuterte Datenflußanalyse sorgt dafür, daß hinter der `return`-Anweisung keine unerreichbaren Anweisungen stehen und daß jeder mögliche Ausgang einer Funktion mit einem `return` versehen ist. Der in C beliebte Fehler, die `return`-Anweisung einer Funktion zu vergessen und damit einen undefinierten Rückgabewert zu erzeugen, kann in Java nicht auftreten.

7.2.5 Überladen von Methoden

In Java ist es erlaubt, Methoden zu *überladen*, d.h. innerhalb einer Klasse zwei unterschiedliche Methoden mit demselben Namen zu definieren. Der Compiler unterscheidet die verschiedenen Varianten anhand der Anzahl und Typisierung der Parameter. Es ist also nicht erlaubt, zwei Methoden mit exakt demselben Namen und identischer Parameterliste zu definieren. Dabei werden auch zwei Methoden, die sich nur durch den Typ ihres Rückgabewertes unterscheiden, als gleich angesehen.

Das Überladen von Methoden ist dann sinnvoll, wenn die gleichnamigen Methoden auch eine vergleichbare Funktionalität haben. Eine typische Anwendung von überladenen Methoden besteht in der Simulation von variablen Parameterlisten (die als Feature direkt in Java nicht zur Verfügung stehen). Auch, um eine Funktion, die bereits an vielen verschiedenen Stellen im Programm aufgerufen wird, um einen weiteren Parameter zu erweitern, ist es nützlich, diese Funktion zu überladen, um nicht alle Aufrufstellen anpassen zu müssen.

Tip

Das folgende Beispiel erweitert die Klasse `Auto` um eine weitere Methode `alter`, die das Alter des Autos nicht nur zurückgibt, sondern es auch mit einem als Parameter übergebenen Titel versieht und auf dem Bildschirm ausgibt:

Beispiel

```
001 public int alter(String titel)
002 {
003     int alter = alter();
004     System.out.println(titel+alter);
005     return alter;
006 }
```

Listing 7.11: Überladen einer Methode

Die Signatur einer Methode

Innerhalb dieser Methode wird der Name `alter` in drei verschiedenen Bedeutungen verwendet. Erstens ist `alter` der Name der Methode selbst. Zweitens wird die lokale Variable `alter` definiert, um drittens den Rückgabewert der parameterlosen `alter`-Methode aufzunehmen. Der Compiler kann die Namen in allen drei Fällen unterscheiden, denn er arbeitet mit der *Signatur* der Methode. Unter der Signatur einer Methode versteht man ihren *internen* Namen. Dieser setzt sich aus dem nach außen sichtbaren Namen plus codierter Information über die Reihenfolge und Typen der formalen Parameter zusammen. Die Signaturen zweier gleichnamiger Methoden sind also immer dann unterscheidbar, wenn sie sich wenigstens in einem Parameter voneinander unterscheiden.

7.2.6 Konstruktoren

In jeder objektorientierten Programmiersprache lassen sich spezielle Methoden definieren, die bei der Initialisierung eines Objekts aufgerufen werden: die *Konstruktoren*. In Java werden Konstruktoren als Methoden ohne Rückgabewert definiert, die den Namen der Klasse erhalten, zu der sie gehören. Konstruktoren dürfen eine beliebige Anzahl an Parametern haben und können überladen werden. Die Erweiterung unserer `Auto`-Klasse um einen Konstruktor, der den Namen des `Auto`-Objekts vorgibt, sieht beispielsweise so aus:

```

001 public class Auto
002 {
003     public String name;
004     public int     erstzulassung;
005     public int     leistung;
006
007     public Auto(String name)
008     {
009         this.name = name;
010     }
011 }

```

Listing 7.12: Definition eines parametrisierten Konstruktors

Soll ein Objekt unter Verwendung eines parametrisierten Konstruktors instanziiert werden, so sind die Argumente wie bei einem Methodenaufruf in Klammern nach dem Namen des Konstruktors anzugeben:

```

001 Auto dasAuto = new Auto("Porsche 911");
002 System.out.println(dasAuto.name);

```

Listing 7.13: Aufruf eines parametrisierten Konstruktors

In diesem Fall wird zunächst Speicher für das `Auto`-Objekt beschafft und dann der Konstruktor aufgerufen. Dieser initialisiert seinerseits die Instanzvariable `name` mit dem übergebenen Argument "Porsche 911". Der nachfolgende Aufruf schreibt dann diesen Text auf den Bildschirm.

Explizite Konstruktoren werden immer dann eingesetzt, wenn zur Initialisierung eines Objektes besondere Aufgaben zu erledigen sind. Es ist dabei durchaus gebräuchlich, Konstruktoren zu überladen und mit unterschiedlichen Parameterlisten auszustatten. Beim Ausführen der `new`-Anweisung wählt der Compiler anhand der aktuellen Parameterliste den passenden Konstruktor und ruft ihn mit den angegebenen Argumenten auf.

Tip

Wir wollen das vorige Beispiel um einen Konstruktor erweitern, der *alle* Instanzvariablen initialisiert:

Beispiel

```

001 public class Auto
002 {
003     public String name;
004     public int     erstzulassung;
005     public int     leistung;
006
007     public Auto(String name)
008     {
009         this.name = name;
010     }
011
012     public Auto(String name,
013                 int     erstzulassung,
014                 int     leistung)
015     {
016         this.name = name;
017         this.erstzulassung = erstzulassung;
018         this.leistung = leistung;
019     }
020 }

```

Listing 7.14: Eine Klasse mit mehreren Konstruktoren

Default-Konstruktoren

Falls eine Klasse überhaupt keinen *expliziten* Konstruktor besitzt, wird beim Anlegen des Objekts ein parameterloser *default*-Konstruktor zur Verfügung gestellt. Seine einzige Aufgabe besteht darin, den parameterlosen Konstruktor der Superklasse aufzurufen. Enthält eine Klassendeklaration dagegen ausschließlich *parametrisierte* Konstruktoren, wird kein *default*-Konstruktor erzeugt, und die Klassendatei besitzt letztendlich überhaupt keinen parameterlosen Konstruktor.

Definiert die Klasse dagegen einen eigenen parameterlosen Konstruktor, überlagert dieser den *default*-Konstruktor und wird bei allen parameterlosen Instanzierungen dieses Objekts verwendet.

Hinweis

Verkettung von Konstruktoren

Konstruktoren können in Java verkettet werden, d.h., sie können sich gegenseitig aufrufen. Der aufzurufende Konstruktor wird dabei als eine normale Methode angesehen, die über den Namen `this` aufgerufen werden kann. Die Unterscheidung zum bereits vorgestellten `this`-Pointer nimmt der Compiler anhand der runden Klammern vor, die dem Aufruf folgen. Der im vorigen Beispiel vorgestellte Konstruktor hätte damit auch so geschrieben werden können:

```
001 public Auto(String name,  
002             int   erstzulassung,  
003             int   leistung)  
004 {  
005     this(name);  
006     this.erstzulassung = erstzulassung;  
007     this.leistung = leistung;  
008 }
```

Beispiel

Listing 7.15: Verkettung von Konstruktoren

Der Vorteil der Konstruktorenverkettung besteht darin, daß vorhandener Code wiederverwendet wird. Führt ein parameterloser Konstruktor eine Reihe von nichttrivialen Aktionen durch, so ist es natürlich sinnvoller, diesen in einem spezialisierteren Konstruktor durch Aufruf wiederzuverwenden, als den Code zu duplizieren.

Hinweis

Wird ein Konstruktor in einem anderen Konstruktor derselben Klasse explizit aufgerufen, muß dies als erste Anweisung innerhalb der Methode geschehen. Steht der Aufruf nicht an erster Stelle, gibt es einen Compiler-Fehler.

Warnung

7.2.7 Destruktoren

Neben Konstruktoren, die während der Initialisierung eines Objekts aufgerufen werden, gibt es in Java auch *Destruktoren*. Sie werden unmittelbar vor dem Zerstören eines Objekts aufgerufen.

Ein Destruktor wird als geschützte parameterlose Methode mit dem Namen `finalize` definiert:

```
001 protected void finalize()  
002 {  
003     ...  
004 }
```

Listing 7.16: Die finalize-Methode

Da Java über ein automatisches Speichermanagement verfügt, kommt den Destruktoren eine viel geringere Bedeutung zu als in anderen objektorientierten Sprachen. Anders als etwa in C++ muß sich der Entwickler ja nicht um die Rückgabe von belegtem Speicher kümmern; und das ist sicher eine der Hauptaufgaben von Destruktoren in C++.

Hinweis

Tatsächlich garantiert die Sprachspezifikation nicht, daß ein Destruktor überhaupt aufgerufen wird. Wenn er aber aufgerufen wird, so erfolgt dies nicht, wenn die Lebensdauer des Objektes endet, sondern dann, wenn der Garbage Collector den für das Objekt reservierten Speicherplatz zurückgibt. Dies kann unter Umständen nicht nur viel später der Fall sein (der Garbage Collector läuft ja als asynchroner Hintergrundprozeß), sondern auch gar nicht. Wird nämlich das Programm beendet, bevor der Garbage Collector das nächste Mal aufgerufen wird, werden auch keine Destruktoren aufgerufen. Selbst wenn Destruktoren aufgerufen werden, ist die Reihenfolge oder der Zeitpunkt ihres Aufrufs undefiniert. Der Einsatz von Destruktoren in Java sollte also mit der nötigen Vorsicht erfolgen.

7.3 Vererbung

- [7.3 Vererbung](#)
 - [7.3.1 Ableiten einer Klasse](#)
 - [7.3.2 Methodenüberlagerung](#)
 - [Dynamische Methodensuche](#)
 - [Aufrufen einer verdeckten Methode](#)
 - [7.3.3 Vererbung von Konstruktoren](#)
 - [Konstruktorenverkettung](#)
 - [Der default-Konstruktor](#)
 - [Überlagerte Konstruktoren](#)
 - [Destruktorenverkettung](#)

Eines der wesentlichen Designmerkmale objektorientierter Sprachen ist die Zusammenfassung von Variablen und Methoden zu Klassen. Eine weiteres wichtiges Merkmal ist das der *Vererbung*, also der Möglichkeit, Eigenschaften vorhandener Klassen auf neue Klassen zu übertragen. Fehlt dieses Merkmal, bezeichnet man die Sprache (gem. *Booch*) auch als lediglich *objektbasiert*.

Man unterscheidet dabei zwischen *einfacher Vererbung*, bei der eine Klasse von maximal einer anderen Klasse abgeleitet werden kann, und *Mehrfachvererbung*, bei der eine Klasse von mehr als einer anderen Klasse abgeleitet werden kann. In Java gibt es lediglich Einfachvererbung, um den Problemen aus dem Weg zu gehen, die durch Mehrfachvererbung entstehen können. Um die Einschränkungen in den Designmöglichkeiten, die bei Einfachvererbung entstehen, zu vermeiden, wurde mit Hilfe der *Interfaces* eine neue, restriktive Art der Mehrfachvererbung eingeführt. Wir werden später darauf zurückkommen.

7.3.1 Ableiten einer Klasse

Um eine neue Klasse aus einer bestehenden abzuleiten, ist im Kopf der Klasse mit Hilfe des Schlüsselworts `extends` ein Verweis auf die Basisklasse anzugeben. Hierdurch erbt die abgeleitete Klasse alle Eigenschaften der Basisklasse, d.h. alle Variablen und alle Methoden. Durch Hinzufügen neuer Elemente oder Überladen der vorhandenen kann die Funktionalität der abgeleiteten Klasse erweitert werden.

Als Beispiel wollen wir eine neue Klasse `Cabrio` definieren, die sich von `Auto` nur dadurch unterscheidet, daß sie zusätzlich die Zeit, die zum Öffnen des Verdecks benötigt wird, speichern soll:

Beispiel

```
001 class Cabrio
002 extends Auto
003 {
004     int vdauer;
005 }
```

Listing 7.17: Ein einfaches Beispiel für Vererbung

Wir können nun nicht nur auf die neue Variable `vdauer`, sondern auch auf alle Elemente der Basisklasse `Auto` zugreifen:

```
001 Cabrio kfz1 = new Cabrio();
002 kfz1.name = "MX5";
003 kfz1.erstzulassung = 1994;
004 kfz1.leistung = 115;
005 kfz1.vdauer = 120;
006 System.out.println("Alter = "+kfz1.alter());
```

Listing 7.18: Zugriff auf geerbte Membervariablen

Die Vererbung von Klassen beliebig tief geschachtelt werden. Eine abgeleitete Klasse erbt dabei jeweils die Eigenschaften der unmittelbaren Vaterklasse, die ihrerseits die Eigenschaften ihrer unmittelbaren Vaterklasse erbt usw. Wir können also beispielsweise die Klasse `Cabrio` verwenden, um daraus eine neue Klasse `ZweisitzerCabrio` abzuleiten:

Hinweis

```

001 class ZweisitzerCabrio
002 extends Cabrio
003 {
004     boolean notsitze;
005 }

```

Listing 7.19: Ableitung einer abgeleiteten Klasse

Diese könnte nun verwendet werden, um ein Objekt zu instanzieren, das die Eigenschaften der Klassen `Auto`, `Cabrio` und `ZweisitzerCabrio` hat:

```

001 ZweisitzerCabrio kfz1 = new ZweisitzerCabrio();
002 kfz1.name = "911-T";
003 kfz1.erstzulassung = 1982;
004 kfz1.leistung = 94;
005 kfz1.vdauer = 50;
006 kfz1.notsitze = true;
007 System.out.println("Alter = "+kfz1.alter());

```

Listing 7.20: Zugriff auf mehrfach vererbte Membervariablen

Nicht jede Klasse darf zur Ableitung neuer Klassen verwendet werden. Besitzt eine Klasse das Attribut `final`, ist es nicht erlaubt, eine neue Klasse aus ihr abzuleiten. Die möglichen Attribute einer Klasse werden im nächsten Abschnitt erläutert.

Hinweis

7.3.2 Methodenüberlagerung

Neben den Membervariablen erbt eine abgeleitete Klasse auch die Methoden ihrer Vaterklasse (wenn dies nicht durch spezielle Attribute verhindert wird). Daneben dürfen auch neue Methoden definiert werden. Die Klasse besitzt dann alle Methoden, die aus der Vaterklasse geerbt wurden und zusätzlich die, die in der Methode neu definiert wurden.

Daneben dürfen auch bereits von der Vaterklasse geerbte Methoden neu definiert werden. In diesem Fall spricht man von *Überlagerung* der Methode. Wurde eine Methode überlagert, wird beim Aufruf der Methode auf Objekten dieses Typs immer die überlagernde Version verwendet.

Das folgende Beispiel erweitert die Klasse `ZweisitzerCabrio` um die Methode `alter`, das nun in Monaten ausgegeben werden soll:

Beispiel

```

001 class ZweisitzerCabrio
002 extends Cabrio
003 {
004     boolean notsitze;
005
006     public int alter()
007     {
008         return 12 * (1999 - erstzulassung);
009     }
010 }

```

Listing 7.21: Überlagern einer Methode in einer abgeleiteten Klasse

Da die Methode `alter` bereits aus der Klasse `Cabrio` geerbt wurde, die sie ihrerseits von `Auto` geerbt hat, handelt es sich um eine *Überlagerung*. Zukünftig würde dadurch in allen Objekten vom Typ `ZweisitzerCabrio` bei Aufruf von `alter` die überlagernde Version, bei allen Objekten des Typs `Auto` oder `Cabrio` aber die alte Version verwendet werden. Es wird immer die Variante aufgerufen, die dem aktuellen Objekt beim Zurückverfolgen der Vererbungslinie am nächsten liegt.

Dynamische Methodensuche

Nicht immer kann bereits der Compiler entscheiden, welche Variante einer überlagerten Methode er aufrufen soll. In [Kapitel 4](#) wurde bereits erwähnt, daß ein Objekt einer abgeleiteten Klasse zuweisungskompatibel zu einer Variablen einer übergeordneten Klasse ist. Wir dürfen also beispielsweise ein `Cabrio`-Objekt ohne weiteres einer Variablen vom Typ `Auto` zuweisen.

Die Variable vom Typ `Auto` kann während ihrer Lebensdauer also Objekte verschiedenen Typs enthalten (insbesondere vom Typ `Auto`, `Cabrio` und `ZweisitzerCabrio`). Damit kann natürlich nicht schon zur Compile-Zeit entschieden werden, welche Version einer überlagerten Methode aufgerufen werden soll. Der Compiler muß also Code generieren, um dies zur Laufzeit zu entscheiden. Man bezeichnet dies auch als *dynamisches Binden*.

Dieses Verhalten wird in C++ durch virtuelle Funktionen realisiert und muß mit Hilfe des Schlüsselworts `virtual` explizit angeordnet werden. In Java ist eine explizite Deklaration nicht nötig, denn Methodenaufrufe werden immer dynamisch interpretiert. Der dadurch verursachte Aufwand zur Laufzeit ist nicht zu vernachlässigen und liegt deutlich über den Kosten eines Funktionsaufrufs in C. Um das Problem zu umgehen, gibt es drei

Möglichkeiten, dafür zu sorgen, daß eine Methode nicht dynamisch interpretiert wird. Dabei wird mit Hilfe zusätzlicher Attribute dafür gesorgt, daß die betreffende Methode nicht überlagert werden kann:

- Methoden vom Typ `private` sind in abgeleiteten Klassen nicht sichtbar und können daher nicht überlagert werden.
- Bei Methoden vom Typ `final` deklariert der Anwender explizit, daß sie nicht überlagert werden sollen.
- Auch bei `static`-Methoden, die ja unabhängig von einer Instanz existieren, besteht das Problem nicht.

Aufrufen einer verdeckten Methode

Wird eine Methode `x` in einer abgeleiteten Klasse überlagert, wird die ursprüngliche Methode `x` verdeckt. Aufrufe von `x` beziehen sich immer auf die überlagernde Variante. Oftmals ist es allerdings nützlich, die verdeckte Superklassenmethode aufrufen zu können, beispielsweise, wenn deren Funktionalität nur leicht verändert werden soll. In diesem Fall kann mit Hilfe des Ausdrucks `super.x()` die Methode der Vaterklasse aufgerufen werden. Der kaskadierte Aufruf von Superklassenmethoden (wie in `super.super.x()`) ist nicht erlaubt.

7.3.3 Vererbung von Konstruktoren

Konstruktorenverkettung

Wenn eine Klasse instanziiert wird, garantiert Java, daß ein zur Parametrisierung des `new`-Operators passender Konstruktor aufgerufen wird. Daneben garantiert der Compiler, daß auch der Konstruktor der Vaterklasse aufgerufen wird. Dieser Aufruf kann entweder explizit oder implizit geschehen.

Falls als erste Anweisung innerhalb eines Konstruktors ein Aufruf der Methode `super` steht, wird dies als Aufruf des Superklassenkonstruktors interpretiert. `super` wird wie eine normale Methode verwendet und kann mit oder ohne Parameter aufgerufen werden. Der Aufruf muß natürlich zu einem in der Superklasse definierten Konstruktor passen.

Falls als erste Anweisung im Konstruktor kein Aufruf von `super` steht, setzt der Compiler an dieser Stelle einen impliziten Aufruf `super()`; ein und ruft damit den parameterlosen Konstruktor der Vaterklasse auf. Falls ein solcher Konstruktor in der Vaterklasse nicht definiert wurde, gibt es einen Compiler-Fehler. Das ist genau dann der Fall, wenn in der Superklassendeklaration lediglich *parametrisierte* Konstruktoren angegeben wurden und daher ein parameterloser *default*-Konstruktor nicht automatisch erzeugt wurde.

Alternativ zu diesen beiden Varianten, einen Superklassenkonstruktor aufzurufen, ist es auch erlaubt, mit Hilfe der `this`-Methode einen anderen Konstruktor der eigenen Klasse aufzurufen. Um die oben erwähnten Zusagen einzuhalten, muß dieser allerdings selbst direkt oder indirekt schließlich einen Superklassenkonstruktor aufrufen.

Hinweis

Der default-Konstruktor

Das Anlegen von Konstruktoren in einer Klasse ist optional. Falls in einer Klasse überhaupt kein Konstruktor definiert wurde, erzeugt der Compiler beim Übersetzen der Klasse automatisch einen parameterlosen *default*-Konstruktor. Dieser enthält lediglich einen Aufruf des parameterlosen Superklassenkonstruktors.

Überlagerte Konstruktoren

Konstruktoren werden nicht vererbt. Alle Konstruktoren, die in einer abgeleiteten Klasse benötigt werden, müssen neu definiert werden, selbst wenn sie nur aus einem Aufruf des Superklassenkonstruktors bestehen.

Hinweis

Durch diese Regel wird bei jedem Neuanlegen eines Objekts eine ganze Kette von Konstruktoren aufgerufen. Da nach den obigen Regeln jeder Konstruktor zuerst den Superklassenkonstruktor aufruft, wird die Initialisierung von oben nach unten in der Vererbungshierarchie durchgeführt: zuerst wird der Konstruktor der Klasse `Object` ausgeführt, dann der der ersten Unterklasse usw., bis zuletzt der Konstruktor der zu instanzierenden Klasse ausgeführt wird.

Destruktorenverkettung

Im Gegensatz zu den Konstruktoren werden die Destruktoren eines Ableitungszweiges nicht automatisch verkettet. Falls eine Destruktorenverkettung erforderlich ist, kann sie durch explizite Aufrufe des Superklassendestruktors mit Hilfe der Anweisung `super.finalize()` durchgeführt werden.

Hinweis

7.4 Attribute von Klassen, Methoden und Variablen

- [7.4 Attribute von Klassen, Methoden und Variablen](#)
 - [7.4.1 Sichtbarkeit](#)
 - [7.4.2 Die Attribute im Überblick](#)
 - [private](#)
 - [protected](#)
 - [public](#)
 - [static](#)
 - [final](#)

In diesem Kapitel wurden an verschiedenen Stellen Beispiele gezeigt, die Attribute wie [public](#) oder [private](#) verwenden. Mit Hilfe dieser Attribute können die Eigenschaften von Klassen, Methoden und Variablen verändert werden. Sie haben insbesondere Einfluß auf die *Lebensdauer*, *Sichtbarkeit* und *Veränderbarkeit* dieser Programmelemente. Der vorliegende Abschnitt erläutert alle Attribute im Zusammenhang und macht ihre Wirkungsweise auf die verschiedenen Elemente eines Java-Programms deutlich.

7.4.1 Sichtbarkeit

Die eingangs erwähnte Tatsache, daß in einer abgeleiteten Klasse alle Eigenschaften der Basisklasse übernommen werden, ist nicht in allen Fällen korrekt. Zwar besitzt eine abgeleitete Klasse immer alle Variablen und Methoden der Basisklasse, sie kann aber unter Umständen nicht darauf zugreifen, wenn deren Sichtbarkeit eingeschränkt wurde.

Die Sichtbarkeit von Variablen und Methoden wird mit Hilfe von *Attributen* (die auch *Modifier* genannt werden) geregelt. In Java gibt es drei Sichtbarkeits Ebenen für den Zugriff auf Elemente einer Klasse:

- Elemente des Typs [public](#) sind in der Klasse selbst (also in ihren Methoden), in Methoden abgeleiteter Klassen und für den Aufrufer von Instanzen der Klasse sichtbar.
- Elemente des Typs [protected](#) sind in der Klasse selbst und in Methoden abgeleiteter Klassen sichtbar. Der Aufrufer einer Instanz der Klasse kann auf Elemente der Klasse [protected](#) dagegen nur noch dann zugreifen, wenn er in demselben Paket definiert wurde.
- Elemente des Typs [private](#) sind lediglich in der Klasse selbst sichtbar. Für abgeleitete Klassen und für Aufrufer von Instanzen bleiben [private](#)-Variablen verdeckt.

Mit Hilfe dieser Sichtbarkeits Ebenen kann der Zugriff auf Klassenelemente eingeschränkt werden. [private](#)-Elemente sollten immer dann verwendet werden, wenn implementierungsabhängige Details zu verstecken sind, die auch in abgeleiteten Klassen nicht sichtbar sein sollen. [protected](#)-Elemente sind vor Zugriffen von außen geschützt, können aber von abgeleiteten Klassen verwendet werden. Die [public](#)-Elemente schließlich bilden die für alle sichtbaren Teile einer Klassendefinition und können daher als ihre Schnittstelle angesehen werden. Nachfolgend werden die verschiedenen Sichtbarkeitsattribute noch einmal genau beschrieben.

Tip

7.4.2 Die Attribute im Überblick

Nachfolgend wollen wir die wichtigsten Attribute noch einmal zusammenfassend darstellen und ihre jeweiligen Auswirkungen auf die Sichtbarkeit, Lebensdauer oder Veränderbarkeit von Variablen, Methoden und Klassen beschreiben. Einige der für uns weniger wichtigen Attribute, wie [transient](#) oder [volatile](#), wollen wir nicht näher betrachten.

private

Methoden oder Variablen vom Typ [private](#) sind nur in der aktuellen Klasse sichtbar. Sowohl für Aufrufer von Objekten der Klasse als auch für abgeleitete Klassen bleiben sie unsichtbar.

protected

Methoden oder Variablen vom Typ [protected](#) sind in der aktuellen Klasse und in abgeleiteten Klassen sichtbar. Darüber hinaus sind sie für Methoden anderer Klassen innerhalb desselben Pakets sichtbar (das ist ein wichtiger Unterschied beispielsweise zu C++). Sie sind jedoch nicht für Aufrufer der Klasse sichtbar, die in anderen Paketen definiert wurden.

public

Membervariablen und Methoden vom Typ `public` sind im Rahmen ihrer Lebensdauer überall sichtbar. Sie können daher in der eigenen Klasse und von beliebigen Methoden anderer Klassen verwendet werden. Das Attribut `public` ist zusätzlich auch bei der Klassendefinition selbst von Bedeutung, denn nur Klassen, die als `public` deklariert wurden, sind außerhalb des Paktes sichtbar, in dem sie definiert wurden. In jeder Quelldatei darf nur eine Klasse mit dem Attribut `public` angelegt werden.

Werden Methoden oder Variablen ohne eines der drei Sichtbarkeitsattribute definiert, so entspricht ihr Verhalten im wesentlichen dem von Elementen des Typs `protected`. Damit sind sie - und das ist ihre wichtigste Eigenschaft - innerhalb desselben Pakets überall sichtbar. Der Unterschied zu `protected` besteht darin, daß sie in Unterklassen, die in anderen Paketen definiert werden, unsichtbar bleiben.

static

Variablen und Methoden mit dem Attribut `static` sind nicht an die Existenz eines konkreten Objekts gebunden, sondern existieren vom Laden der Klasse bis zum Beenden des Programms. Das `static`-Attribut beeinflußt bei Membervariablen ihre Lebensdauer und erlaubt bei Methoden den Aufruf, ohne daß der Aufrufer ein Objekt der Klasse besitzt, in der die Methode definiert wurde.

Wird das Attribut `static` nicht verwendet, so sind Variablen innerhalb einer Klasse immer an eine konkrete Instanz gebunden. Ihre Lebensdauer beginnt mit dem Anlegen des Objekts und dem Aufruf eines Konstruktors und endet mit der Freigabe des Objekts durch den Garbage Collector.

final

Variablen mit dem Attribut `final` dürfen nicht verändert werden, sind also als *Konstanten* anzusehen. Methoden des Typs `final` dürfen nicht überlagert werden; ebensowenig dürfen Klassen des Typs `final` zur Ableitung neuer Klassen verwendet werden. Wird das Attribut `final` dagegen nicht verwendet, sind Variablen veränderbar, können Methoden überlagert und Klassen abgeleitet werden.

Falls eine Methode oder Klasse das Attribut `final` besitzt, kann der Compiler in der Regel auf die dynamische Methodensuche verzichten; `final`-Methoden können daher schneller aufgerufen werden als normale Methoden. Dies ist einer der Gründe dafür, daß die Java-Designer einige der mitgelieferten Klassen als `final` deklariert haben. Es führt aber gleichzeitig dazu, daß die entsprechenden Klassen nicht mehr erweitert werden können. Prominentestes Beispiel aus der Laufzeitbibliothek ist sicherlich die als `final` deklarierte Klasse `String`.

Hinweis

7.5 Klassen mit static-Elementen

- [7.5 Klassen mit static-Elementen](#)
 - [7.5.1 Klassenvariablen](#)
 - [7.5.2 Konstanten](#)
 - [7.5.3 Klassenmethoden](#)
 - [7.5.4 Statische Konstruktoren](#)

Java ist eine konsequent objektorientierte Sprache, in der es weder globale Funktionen noch globale Variablen gibt. Da es aber mitunter sinnvoll ist, Methoden aufzurufen, die nicht an Instanzen einer Klasse gebunden sind, haben die Sprachdesigner das Attribut `static` für Methoden und Variablen eingeführt.

7.5.1 Klassenvariablen

Variablen, die innerhalb einer Klasse mit dem Attribut `static` versehen werden, nennt man *Klassenvariablen*. Im Gegensatz zu den Instanzvariablen, die an ein Objekt gebunden sind, existieren Klassenvariablen unabhängig von einem Objekt.

Jede Klassenvariable wird nur einmal angelegt und kann von allen Methoden der Klasse aufgerufen werden. Da sich alle Methoden die Variable »teilen«, sind Veränderungen, die eine Instanz vornimmt, auch in allen anderen Instanzen sichtbar. Klassenvariablen sind daher vergleichbar mit globalen Variablen, denn ihre Lebensdauer erstreckt sich auf das gesamte Programm. Namenskollisionen können allerdings nicht auftreten, denn der Zugriff von außen erfolgt durch Qualifizierung mit dem Klassennamen in der Form `Klassenname.Variablenname`.

Ein gutes Beispiel für die Verwendung von Klassenvariablen besteht darin, einen Instanzenzähler in eine Klasse einzubauen. Hierzu wird eine beliebige Klassenvariable eingeführt, die beim Erzeugen eines Objekts hoch- und beim Zerstören heruntergezählt wird. Das folgende Beispiel demonstriert das für die Klasse `Auto`:

Beispiel

[Testauto.java](#)

```

001 /* Testauto.java */
002
003 public class Testauto
004 {
005     static private int objcnt = 0;
006
007     public Testauto()
008     {
009         ++objcnt;
010     }
011
012     public void finalize()
013     {
014         --objcnt;
015     }
016
017     public static void main(String args[])
018     {
019         Testauto auto1;
020         Testauto auto2 = new Testauto();
021         System.out.println(
022             "Anzahl Testauto-Objekte: " + Testauto.objcnt
023         );
024     }
025 }

```

Listing 7.22: Realisierung eines Instanzenzählers mit Klassenvariablen

Die Ausgabe des Programms ist:

Anzahl Testauto-Objekte: 1

Mit `auto2` wurde eine Instanz der Klasse erzeugt, `auto1` ist dagegen zum Zeitpunkt der Ausgabeanweisung lediglich eine noch nicht initialisierte

Objektreferenz.

7.5.2 Konstanten

Eine andere Anwendung von Klassenvariablen besteht in der Deklaration von Konstanten. Dazu wird das `static`-Attribut mit dem `final`-Attribut kombiniert, um eine unveränderliche Variable mit unbegrenzter Lebensdauer zu erzeugen:

```
001 public class Auto
002 {
003     private static final STEUERSATZ = 18.9;
004 }
```

Listing 7.23: Verwendung von Klassenvariablen zur Definition von Konstanten

Durch die Anwendung von `final` wird verhindert, daß der Konstanten `STEUERSATZ` während der Ausführung des Programms ein anderer Wert zugewiesen wird. Da Java keinen Präprozessor enthält und damit keine `#define`-Anweisung kennt, ist die beschriebene Methode das einzige Verfahren zur Deklaration von Konstanten in Java. Die Konvention, Konstantennamen groß zu schreiben, wurde dabei von C übernommen.

Hinweis

7.5.3 Klassenmethoden

Neben Klassenvariablen gibt es in Java auch *Klassenmethoden*, d.h. Methoden, die unabhängig von einer bestimmten Instanz existieren. Klassenmethoden werden ebenfalls mit Hilfe des `static`-Attributs deklariert und - analog zu Klassenvariablen - durch Voranstellen des Klassennamens aufgerufen.

Da Klassenmethoden unabhängig von konkreten Instanzen ihrer Klasse existieren, ist ein Zugriff auf Instanzvariablen nicht möglich. Diese Trennung äußert sich darin, daß Klassenmethoden keinen `this`-Zeiger besitzen. Der Zugriff auf Instanzvariablen und der Aufruf von Instanzmethoden wird daher schon zur Compile-Zeit als Fehler erkannt.

Klassenmethoden werden häufig da eingesetzt, wo Funktionalitäten zur Verfügung gestellt werden, die nicht datenzentriert arbeiten oder auf primitiven Datentypen operieren. Beispiele für beide Arten sind in der Klassenbibliothek zu finden. Zur ersten Gruppe gehören beispielsweise die Methoden der Klasse `System`. Die Klasse `System` ist eine Art Toolbox, die Funktionen wie *Aufruf des Garbage Collectors* oder *Beenden des Programms* zur Verfügung stellt. Zur zweiten Gruppe gehören beispielsweise die Methoden der Klasse `Math`, die eine große Anzahl an Funktionen zur Fließkomma-Arithmetik zur Verfügung stellt. Da die Fließkommatypen primitiv sind, hätte ein instanzbasiertes Methodendesign an dieser Stelle wenig Sinn gemacht.

Tip

Das folgende Listing zeigt die Verwendung der Klassenmethode `sqrt` der Klasse `Math` zur Ausgabe einer Tabelle von Quadratwurzeln:

Beispiel

[Listing0724.java](#)

```
001 /* Listing0724.java */
002
003 public class Listing0724
004 {
005     public static void main(String[] args)
006     {
007         double x, y;
008
009         for (x = 0.0; x <= 10.0; x = x + 1.0) {
010             y = Math.sqrt(x);
011             System.out.println("sqrt("+x+") = "+y);
012         }
013     }
014 }
```

Listing 7.24: Verwendung von `Math.sqrt`

Die Ausgabe des Programms ist:

```
sqrt(0.0) = 0.0
sqrt(1.0) = 1.0
sqrt(2.0) = 1.4142135623730951
sqrt(3.0) = 1.7320508075688772
sqrt(4.0) = 2.0
sqrt(5.0) = 2.23606797749979
sqrt(6.0) = 2.449489742783178
```

```
sqrt(7.0) = 2.6457513110645907
sqrt(8.0) = 2.8284271247461903
sqrt(9.0) = 3.0
sqrt(10.0) = 3.1622776601683795
```

7.5.4 Statische Konstruktoren

Bisher haben wir nur die Möglichkeit kennengelernt, statischen Variablen während der Deklaration einen Wert zuzuweisen. Falls komplexere Initialisierungen benötigt werden, können zur Initialisierung von statischen Variablen *statische Konstruktoren* definiert werden. Sie ähneln gewöhnlichen Konstruktoren und dienen wie diese dazu, Variablen mit einem definierten Startwert zu belegen. Im Gegensatz zu einem gewöhnlichen Konstruktor erfolgt der Aufruf eines statischen Konstruktors aber nicht mit jedem neu angelegten Objekt, sondern nur einmal zu Beginn des Programms.

Ein statischer Konstruktor wird als parameterlose Methode mit dem Namen `static` definiert:

```
001 class Test
002 {
003     static int i;
004     static int j;
005
006     static
007     {
008         i = 5;
009         j = 3 * i;
010     }
011 }
```

Listing 7.25: Definition eines statischen Konstruktors

Da er vom Laufzeitsystem aufgerufen wird, wenn die Klasse geladen wird, ist eine benutzerdefinierte Parametrisierung nicht möglich. Eine Klasse darf mehr als einen statischen Konstruktor definieren, sie werden in der Reihenfolge ihrer Deklaration aufgerufen.

7.6 Abstrakte Klassen und Methoden

- 7.6 Abstrakte Klassen und Methoden

In Java ist es möglich, *abstrakte* Methoden zu definieren. Im Gegensatz zu *konkreten* Methoden enthalten sie nur die Deklaration des Methodenkopfes, aber keine Implementierung des Methodenrumpfes. Syntaktisch unterscheiden sich abstrakte Methoden dadurch, daß anstelle der geschweiften Klammern mit den auszuführenden Anweisungen lediglich ein Semikolon steht. Zusätzlich wird die Definition mit dem Attribut `abstract` versehen.

Abstrakte Methoden können nicht aufgerufen werden. So definieren sie nur eine Schnittstelle, die durch Überlagerung in einer abgeleiteten Klasse implementiert werden kann.

Eine Klasse, die mindestens eine abstrakte Methode enthält, wird selbst als abstrakt angesehen und muß ebenfalls mit dem Schlüsselwort `abstract` versehen werden. Abstrakte Klassen können nicht instanziiert werden, da sie Methoden enthalten, die nicht implementiert wurden. Statt dessen werden abstrakte Klassen abgeleitet, und in der abgeleiteten Klasse werden eine oder mehrere der abstrakten Methoden implementiert. Eine abstrakte Klasse wird konkret, wenn alle ihre Methoden implementiert sind. Die Konkretisierung kann dabei auch schrittweise über mehrere Vererbungsstufen erfolgen.

Hinweis

Wir wollen uns ein Beispiel ansehen, um diese Ausführungen zu verdeutlichen. Zum Aufbau einer Mitarbeiterdatenbank soll zunächst eine Basisklasse definiert werden, die jene Eigenschaften implementiert, die für alle Mitarbeiter zutreffen, wie beispielsweise *persönliche Daten* oder der *Eintrittstermin in das Unternehmen*. Gleichzeitig soll diese Klasse als Basis für spezialisierte Unterklassen verwendet werden, um die Besonderheiten spezieller Mitarbeiterarten, wie *Arbeiter*, *Angestellte* oder *Manager*, abzubilden. Da die Berechnung des monatlichen Gehalts zwar für jeden Mitarbeiter erforderlich, in ihrer konkreten Realisierung aber abhängig vom Typ des Mitarbeiters ist, soll eine abstrakte Methode `monatsBrutto` in der Basisklasse definiert werden, die in den abgeleiteten Klassen konkretisiert wird.

Beispiel

Das folgende Beispiel zeigt die Implementierung der Klassen `Mitarbeiter`, `Arbeiter`, `Angestellter` und `Manager` zur Realisierung der verschiedenen Mitarbeiterarten. Zusätzlich wird die Klasse `Gehaltsberechnung` definiert, um das Hauptprogramm zur Verfügung zu stellen, in dem die Gehaltsberechnung durchgeführt wird. Dazu wird ein Array `ma` mit konkreten Untertypen der Klasse `Mitarbeiter` gefüllt (hier nur angedeutet) und dann für alle Elemente das Monatsgehalt durch Aufruf von `monatsBrutto` ermittelt.

[Gehaltsberechnung.java](#)

```

001 /* Gehaltsberechnung.java */
002
003 import java.util.Date;
004
005 abstract class Mitarbeiter
006 {
007     int persnr;
008     String name;
009     Date eintritt;
010
011     public Mitarbeiter()
012     {
013     }
014
015     public abstract double monatsBrutto();
016 }
017
018 class Arbeiter
019 extends Mitarbeiter
020 {
021     double stundenlohn;
022     double anzahlstunden;
023     double ueberstundenzuschlag;
024     double anzahlueberstunden;
025     double schichtzulage;
026

```

```

027     public double monatsBrutto()
028     {
029         return stundenlohn*anzahlstunden+
030                ueberstundenzuschlag*anzahlueberstunden+
031                schichtzulage;
032     }
033 }
034
035 class Angestellter
036 extends Mitarbeiter
037 {
038     double grundgehalt;
039     double ortszuschlag;
040     double zulage;
041
042     public double monatsBrutto()
043     {
044         return grundgehalt+
045                ortszuschlag+
046                zulage;
047     }
048 }
049
050 class Manager
051 extends Mitarbeiter
052 {
053     double fixgehalt;
054     double provision1;
055     double provision2;
056     double umsatz1;
057     double umsatz2;
058
059     public double monatsBrutto()
060     {
061         return fixgehalt+
062                umsatz1*provision1/100+
063                umsatz2*provision2/100;
064     }
065 }
066
067 public class Gehaltsberechnung
068 {
069     private static final int ANZ_MA = 100;
070     private static Mitarbeiter ma[];
071     private static double bruttosumme;
072
073     public static void main(String args[])
074     {
075         ma = new Mitarbeiter[ANZ_MA];
076
077         //Mitarbeiter-Array füllen, z.B.
078         //ma[0] = new Manager();
079         //ma[1] = new Arbeiter();
080         //ma[2] = new Angestellter();
081         //...
082
083         //Bruttosumme berechnen
084         bruttosumme = 0.0;
085         for (int i=0; i<ma.length; ++i) {
086             bruttosumme += ma[i].monatsBrutto();
087         }
088         System.out.println("Bruttosumme = "+bruttosumme);
089     }
090 }

```

Listing 7.26: Abstrakte Klassen und Methoden

Unabhängig davon, ob in einem Array-Element ein Arbeiter, Angestellter oder Manager gespeichert wird, führt der Aufruf von `monatsBrutto` dank der dynamischen Methodensuche die zum Typ des konkreten Objekts passende Berechnung aus. Auch weitere Verfeinerungen der Klassenhierarchie durch Ableiten neuer Klassen erfordern keine Veränderung in der Routine zur Berechnung der monatlichen Bruttosumme.

So könnte beispielsweise eine neue Klasse `GFManager` (ein Manager, der Mitglied der Geschäftsführung ist) aus `Manager` abgeleitet und problemlos in die Gehaltsberechnung integriert werden:

```
001 public class GFManager
002 extends Manager
003 {
004     double gfzulage;
005
006     public double monatsBrutto()
007     {
008         return super.monatsBrutto()+gfzulage;
009     }
010 }
```

Listing 7.27: Einfügen einer neuen Mitarbeiterklasse in die Gehaltsberechnung

In der abgedruckten Form ist das Programm in [Listing 7.26](#) natürlich nicht lauffähig, denn das Array `ma` wird nicht vollständig initialisiert. Ansonsten ist es aber korrekt und illustriert beispielhaft die Anwendung abstrakter Klassen und Methoden.

Hinweis

7.7 Interfaces

- 7.7 Interfaces
 - 7.7.1 Definition eines Interfaces
 - 7.7.2 Verwendung von Interfaces
 - 7.7.3 Implementieren mehrerer Interfaces
 - 7.7.4 Interfaces sind Klassen
 - 7.7.5 Konstanten in Interfaces
 - 7.7.6 Beispiele aus der Java-Klassenbibliothek
 - Runnable
 - Enumeration
 - Cloneable

Es wurde bereits erwähnt, daß es in Java keine *Mehrfachvererbung* von Klassen gibt. Die möglichen Schwierigkeiten beim Umgang mit mehrfacher Vererbung und die Einsicht, daß das Ererben von nichttrivialen Methoden aus mehr als einer Klasse in der Praxis selten zu realisieren ist, haben die Designer dazu veranlaßt, dieses Feature nicht zu implementieren. Andererseits sah man es sehr wohl als wünschenswert an, Methodendeklarationen von mehr als einer Klasse zu erben und hat mit den *Interfaces* ein Ersatzkonstrukt geschaffen, das genau dieses Feature bietet.

7.7.1 Definition eines Interfaces

Ein Interface ist eine besondere Form einer Klasse, die ausschließlich abstrakte Methoden und Konstanten enthält. Anstelle von `class` wird zur Definition eines Interfaces das Schlüsselwort `interface` verwendet. Alle Methoden sind daraufhin standardmäßig abstrakt. Ein Interface kann allerdings keine Konstruktoren enthalten.

Das folgende Listing definiert ein Interface `Fortbewegungsmittel`, das die Methoden `kapazitaet` und `kilometerPreis` definiert:

```
001 public interface Fortbewegungsmittel
002 {
003     public int kapazitaet();
004     public double kilometerPreis();
005 }
```

Listing 7.28: Definition eines Interfaces

7.7.2 Verwendung von Interfaces

Was bei der Vererbung von Klassen als *Ableitung* bezeichnet wird, nennt man bei Interfaces *Implementierung*. Durch das Implementieren eines Interfaces verpflichtet sich die Klasse, *alle* Methoden, die im Interface definiert sind, zu implementieren. Fehlt eine Methode, so gibt es einen Compilerfehler. Die Implementierung eines Interfaces wird durch das Schlüsselwort `implements` bei der Klassendefinition angezeigt.

Als Beispiel wollen wir die Klasse `Auto` um das neue Interface `Fortbewegungsmittel` erweitern und die Methoden `kapazitaet` und `kilometerPreis` implementieren:

```
001 public class Auto
002 implements Fortbewegungsmittel
003 {
004     public String name;
005     public int    erstzulassung;
006     public int    leistung;
007     private int   anzahlSitze;
008     private double spritVerbrauch;
009     private double spritPreis;
010
011     public double kapazitaet()
012     {
013         return anzahlSitze;
014     }
```

Beispiel

```

015
016     public double kilometerPreis()
017     {
018         return spritVerbrauch*spritPreis/100;
019     }
020
021 }

```

Listing 7.29: Implementieren eines Interfaces

Ebenso wie die Klasse `Auto` könnte auch jede andere Klasse das Interface `Fortbewegungsmittel` implementieren und Konkretisierungen der beiden Methoden vornehmen. Nützlich ist dies insbesondere für Klassen, die in keinem direkten Zusammenhang mit der Klasse `Auto` und ihrer Vererbungshierarchie stehen. Um ihr gewisse Eigenschaften eines Fortbewegungsmittels zu verleihen, könnte also beispielsweise auch die Klasse `Teppich` dieses Interface implementieren.

Eine Klasse kann auch dann ein Interface implementieren, wenn sie bereits von einer anderen Klasse abgeleitet ist. In diesem Fall erbt die neue Klasse wie gewohnt alle Eigenschaften der Basisklasse und hat zusätzlich die Aufgabe, die abstrakten Methoden des Interfaces zu implementieren.

Hinweis

Das folgende Interface `Sammlerstueck` mag bei gewöhnlichen Autos keine Anwendung finden, ist bei einem Oldtimer aber durchaus sinnvoll:

Beispiel

```

001 public interface Sammlerstueck
002 {
003     public double sammlerWert();
004     public String bisherigeAusstellungen();
005 }
006
007 public class Oldtimer
008     extends Auto
009     implements Sammlerstueck
010 {
011     //...
012 }

```

Listing 7.30: Definition eines weiteren Interfaces

Da ein `Sammlerstueck` aber durchaus auch in ganz anderen Vererbungshierarchien auftauchen kann als bei Autos (beispielsweise bei Briefmarken, Schmuck oder Telefonkarten), macht es keinen Sinn, diese Methoden in den Ableitungsbäumen all dieser Klassen wiederholt zu deklarieren. Statt dessen sollten die Klassen das Interface `Sammlerstueck` implementieren und so garantieren, daß die Methoden `sammlerWert` und `bisherigeAusstellungen` zur Verfügung stehen.

7.7.3 Implementieren mehrerer Interfaces

Eine Klasse kann nicht nur ein einzelnes Interface, sondern eine beliebige Anzahl von ihnen implementieren. So ist es beispielsweise problemlos möglich, eine aus `Flugzeug` abgeleitete Klasse `Doppeldecker` zu definieren, die sowohl `Sammlerstueck` als auch `Fortbewegungsmittel` implementiert:

```

001 public class Doppeldecker
002     extends Flugobjekt
003     implements Fortbewegungsmittel, Sammlerstueck
004 {
005     //...
006 }

```

Listing 7.31: Implementieren mehrerer Interfaces

Die Klasse `Doppeldecker` muß dann alle in `Sammlerobjekt` und `Fortbewegungsmittel` deklarierten Methoden implementieren.

7.7.4 Interfaces sind Klassen

Interfaces besitzen zwei wichtige Eigenschaften, die auch Klassen haben:

- Sie lassen sich vererben. Wird ein Interface `B` aus einem Interface `A` abgeleitet, so erbt es alle Deklarationen von `A` und kann diese um eigene Erweiterungen ergänzen.
- Variablen können vom Typ eines Interfaces sein. In diesem Fall kann man ihnen Objekte zuweisen, die aus Klassen abstammen, die dieses oder eines der daraus abgeleiteten Interfaces implementieren. Sie selbst können Variablen zugewiesen werden, die zu Klassen gehören, die dieses Interface implementieren.

Es macht also Sinn, ein Interface als eine Typvereinbarung anzusehen. Eine Klasse, die dieses Interface implementiert, ist dann vom Typ des Interfaces. Wegen der Mehrfachvererbung von Interfaces kann eine Instanzvariable damit insbesondere mehrere Typen haben und zu mehr als einem Typen zuweisungskompatibel sein.

Hinweis

7.7.5 Konstanten in Interfaces

Neben abstrakten Methoden können Interfaces auch Konstanten, also Variablen mit den Attributen `static` und `final`, enthalten. Wenn eine Klasse ein solches Interface implementiert, erbt es gleichzeitig auch all seine Konstanten. Es ist auch erlaubt, daß ein Interface ausschließlich Konstanten enthält.

Dieses Feature kann zum Beispiel nützlich sein, wenn ein Programm sehr viele Konstanten definiert. Anstatt diese in ihren korrespondierenden Klassen zu belassen und mit `Klasse.Name` aufzurufen, könnte ein einzelnes Interface definiert werden, das alle Konstantendefinitionen vereinigt. Wenn nun jede Klasse, die eine der Konstanten benötigt, dieses Interface implementiert, so stehen alle darin definierten Konstanten direkt zur Verfügung und können ohne die Qualifizierung mit einem Klassennamen aufgerufen werden.

Tip

7.7.6 Beispiele aus der Java-Klassenbibliothek

Die Java-Klassenbibliothek definiert und implementiert selbst eine ganze Reihe von Interfaces. Drei von ihnen sollen hier kurz vorgestellt werden.

Runnable

Das Interface `Runnable` wird zur Implementierung von Threads verwendet, und zwar insbesondere dann, wenn die betreffende Klasse selbst nicht direkt aus `Thread` abgeleitet werden kann. `Runnable` deklariert lediglich die Methode `run`, deren Rumpf den ausführbaren Code des Threads enthält. Der Umgang mit Threads wird in [Kapitel 10](#) erklärt.

Enumeration

Das Interface `Enumeration` wird zur Deklaration von *Enumeratoren* verwendet. Enumeratoren dienen dazu, alle Elemente von Kollektionen (z.B. `Hashtable` oder `Vector`) nacheinander zu durchlaufen. `Enumeration` definiert die Methoden `nextElement` und `hasMoreElements`. Sie dienen dazu, das nächste Element der Kollektion zu holen bzw. zu testen, ob weitere Elemente vorhanden sind.

Cloneable

Dieses Interface ist etwas untypisch, denn es besitzt weder Methoden noch Konstanten. Es dient lediglich dazu, dem Compiler anzuzeigen, daß eine Klasse die Methode `clone` unterstützt und damit die Fähigkeit besitzt, sich selbst zu kopieren.

7.8 Spezielle Klassen

- 7.8 Spezielle Klassen
 - 7.8.1 Die Wrapper-Klassen
 - Instanzierung
 - Rückgabe des Wertes
 - Parsen von Strings
 - Konstanten
 - 7.8.2 Die Klasse `java.lang.Math`
 - Winkelfunktionen
 - Minimum und Maximum
 - Arithmetik
 - Runden und Abschneiden

7.8.1 Die Wrapper-Klassen

Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende *Wrapper*-Klasse. Diese kapselt die primitive Variable in einer objektorientierten Hülle und stellt eine Reihe von Methoden zum Zugriff auf die Variable zur Verfügung. Zwar wird man bei der Programmierung meist die primitiven Typen verwenden, doch gibt es einige Situationen, in denen die Anwendung einer Wrapper-Klasse sinnvoll sein kann:

- Das Paket `java.util` stellt eine Reihe von Verbundklassen zur Verfügung, die beliebige Objekttypen speichern können. Um darin auch elementare Typen ablegen zu können, ist es notwendig, anstelle der primitiven Typen ihre Wrapper-Klassen zu verwenden.
- Da Objektreferenzen den Wert `null` haben können, kann die Verwendung der Wrapper-Klassen beispielsweise bei der Datenbankprogrammierung nützlich sein. Damit lassen sich primitive Feldtypen darzustellen, die NULL-Werte enthalten können.
- Das Reflection-API (siehe [Kapitel 31](#)) verwendet Wrapper-Klassen zum Zugriff auf Membervariablen oder Methodenargumente primitiver Typen.

Da Objektparameter im Gegensatz zu primitiven Typen per Referenz übergeben werden, bieten die Wrapper-Klassen *prinzipiell* die Möglichkeit, Methodenparameter per *call by reference* zu übergeben, und damit Änderungen von primitiven Parametern an den Aufrufer zurückzugeben. In der Praxis funktioniert das allerdings nicht, denn alle vordefinierten Wrapper-Klassen sind unveränderlich (sie werden als *immutable* bezeichnet). Als Alternative bietet es sich lediglich an, eigene Wrapper-Klassen zu definieren, die eine Möglichkeit zum Ändern des eingepackten Wertes bieten.

Hinweis

Wrapper-Klassen existieren zu allen numerischen Typen und zu den Typen `char` und `boolean`:

Wrapper-Klasse	Primitiver Typ
Byte	byte
Short	short
Integer	int
Long	long
Double	double
Float	float
Boolean	boolean
Character	char
Void	void

Tabelle 7.1: Die Wrapper-Klassen

Instanzierung

Die Instanzierung einer Wrapper-Klasse kann meist auf zwei unterschiedliche Arten erfolgen. Einerseits ist es möglich, den korrespondierenden primitiven Typ an den Konstruktor zu übergeben, um ein Objekt desselben Werts zu erzeugen. Alternativ kann meist auch ein String an den

Konstruktor übergeben werden. Dieser wird in den entsprechenden primitiven Typ konvertiert und dann zur Initialisierung der Wrapper-Klasse verwendet.

```
public Integer(int i)

public Integer(String s)
    throws NumberFormatException

public Long(long l)

public Long(String s)
    throws NumberFormatException

public Float(float f)

public Float(double d)

public Float(String s)
    throws NumberFormatException

public Double(double d)

public Double(String s)
    throws NumberFormatException

public Boolean(boolean b)

public Boolean(String s)

public Character(char c)
```

Das in diesem Beispiel mehrfach verwendete Schlüsselwort [throws](#) deklariert *Ausnahmen*, die von einer Methode verursacht werden können. Wir werden darauf in [Kapitel 9](#) zurückkommen und uns dort ausführlich mit dem Konzept der Ausnahmen beschäftigen.

Hinweis

Rückgabe des Wertes

Die meisten Wrapper-Klassen besitzen zwei Methoden, um den internen Wert abzufragen. Eine der beiden liefert ihn passend zum korrespondierenden Grundtyp, die andere als [String](#). Der Name von Methoden der ersten Art setzt sich aus dem Namen des Basistyps und der Erweiterung [Value](#) zusammen, beispielsweise [charValue](#), [booleanValue](#) oder [intValue](#). Die numerischen Methoden [intValue](#), [longValue](#), [floatValue](#) und [doubleValue](#) stehen dabei für alle numerischen Wrapper-Klassen zur Verfügung.

```
public boolean booleanValue()
public char charValue()
public int intValue()
public long longValue()
public float floatValue()
public double doubleValue()
```

Der Name der Methode, die den internen Wert als [String](#) zurückgibt, ist [toString](#). Diese Methode steht in allen Wrapper-Klassen zur Verfügung:

```
public String toString()
```

Parsen von Strings

Neben der Möglichkeit, aus Strings Objekte zu erzeugen, können die meisten Wrapper-Klassen auch primitive Datentypen erzeugen. Dazu gibt es statische Methoden mit den Namen [parseByte](#), [parseInt](#), [parseLong](#), [parseFloat](#) und [parseDouble](#) in den zugehörigen Klassen [Byte](#), [Integer](#), [Long](#), [Float](#) und [Double](#):


```

public static byte parseByte(String s)
    throws NumberFormatException

public static int parseInt(String s)
    throws NumberFormatException

public static long parseLong(String s)
    throws NumberFormatException

public static float parseFloat(String s)
    throws NumberFormatException

public static double parseDouble(String s)
    throws NumberFormatException

```

Konstanten

Die numerischen Wrapper-Klassen stellen Konstanten zur Bezeichnung spezieller Elemente zur Verfügung. So gibt es in jeder der Klassen [Byte](#), [Short](#), [Integer](#), [Long](#), [Float](#) und [Double](#) die Konstanten [MIN_VALUE](#) und [MAX_VALUE](#), die das kleinste bzw. größte Element des Wertebereichs darstellen. In den Klassen [Float](#) und [Double](#) gibt es zusätzlich die Konstanten [NEGATIVE_INFINITY](#), [POSITIVE_INFINITY](#) und [NaN](#). Sie stellen die Werte *minus unendlich*, *plus unendlich* und *undefiniert* dar.

7.8.2 Die Klasse java.lang.Math

Die Klasse [java.lang.Math](#) enthält Funktionen zur Fließkomma-Arithmetik. Alle Methoden dieser Klasse sind vom Typ [static](#) und können daher ohne konkretes Objekt verwendet werden. Nachfolgend wollen wir die wichtigsten von ihnen auflisten und eine kurze Beschreibung ihrer Funktionsweise geben.

Winkelfunktionen

[java.lang.Math](#) stellt die üblichen Winkelfunktionen und ihre Umkehrungen zur Verfügung. Winkelwerte werden dabei im Bogenmaß übergeben.

[java.lang.Math](#)

```

public static double sin(double x)
public static double cos(double x)
public static double tan(double x)
public static double asin(double x)
public static double acos(double x)
public static double atan(double x)

```

Minimum und Maximum

Die Methoden [min](#) und [max](#) erwarten zwei numerische Werte als Argument und geben das kleinere bzw. größere von beiden zurück.

[java.lang.Math](#)

```

public static int min(int a, int b)
public static long min(long a, long b)
public static float min(float a, float b)
public static double min(double a, double b)

public static int max(int a, int b)
public static long max(long a, long b)
public static float max(float a, float b)
public static double max(double a, double b)

```

Arithmetik

Die nachfolgend aufgelisteten Methoden dienen zur Berechnung der Exponentialfunktion zur Basis *e*, zur Berechnung des natürlichen Logarithmus und zur Berechnung der Exponentialfunktion zu einer beliebigen Basis. Mit [sqrt](#) kann die Quadratwurzel berechnet werden.

```
public static double exp(double a)

public static double log(double a)
    throws ArithmeticException

public static double pow(double a, double b)
    throws ArithmeticException

public static double sqrt(double a)
    throws ArithmeticException
```

Runden und Abschneiden

Mit Hilfe der Methode [abs](#) wird der absolute Betrag eines numerischen Werts bestimmt. [ceil](#) liefert die kleinste ganze Zahl größer und [floor](#) die größte ganze Zahl kleiner oder gleich dem übergebenen Argument. Mit Hilfe von [round](#) kann ein Wert gerundet werden.

[java.lang.Math](#)

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

public static double ceil(double a)
public static double floor(double a)
public static int round(float a)
```

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	<	>	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	----------------------	----------------------	--------------------------

7.9 Zusammenfassung

- [7.9 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Bedeutung von Klassen und Objekten
- Definition und Aufruf von Methoden
- Methodenparameter und die Übergabearten *call by value* und *call by reference*
- Der Rückgabewert einer Methode und die [return](#)-Anweisung
- Überladen von Methoden
- Konstruktoren, Default-Konstruktoren und die Verkettung von Konstruktoren
- Destruktoren und das Schlüsselwort [finalize](#)
- Das Prinzip der Vererbung und das Schlüsselwort [extends](#) zum Ableiten einer Klasse
- Methodenüberlagerung, die Bedeutung des statischen und dynamischen Bindens und das Schlüsselwort [super](#)
- Die Attribute [private](#), [protected](#), [public](#), [static](#) und [final](#)
- Klassenvariablen und -methoden und statische Konstruktoren
- Abstrakte Klassen und die Definition und Verwendung von Interfaces
- Die speziellen Interfaces [Enumeration](#), [Runnable](#) und [Cloneable](#)
- Die Wrapper-Klassen [Byte](#), [Short](#), [Integer](#), [Long](#), [Double](#), [Float](#), [Boolean](#), [Character](#) und [Void](#)
- Die Klasse [Math](#)

Kapitel 8

Bestandteile eines Programms

- [8 Bestandteile eines Programms](#)
 - [8.1 Programmelemente](#)
 - [8.1.1 Anweisungen](#)
 - [8.1.2 Blöcke](#)
 - [8.1.3 Methoden](#)
 - [8.1.4 Klassen](#)
 - [8.1.5 Pakete](#)
 - [8.1.6 Applikationen](#)
 - [8.1.7 Applets](#)
 - [8.2 Pakete](#)
 - [8.2.1 Verwendung von Paketen](#)
 - [Der Import von java.lang](#)
 - [Die vordefinierten Pakete im JDK](#)
 - [8.2.2 Die Bedeutung der Paketnamen](#)
 - [Laden von Klassen im JDK 1.0 und 1.1](#)
 - [Laden von Klassen im JDK 1.2](#)
 - [Die Dateien classes.zip und rt.jar](#)
 - [Umgekehrte Domain-Namen](#)
 - [8.2.3 Einbinden zusätzlicher Pakete](#)
 - [8.2.4 Erstellen eigener Pakete](#)
 - [Benannte Pakete](#)
 - [Das Default-Paket](#)
 - [Das public-Attribut](#)
 - [8.3 Der Entwicklungszyklus](#)
 - [8.3.1 Schematische Darstellung](#)
 - [8.3.2 Projektverwaltung](#)
 - [Getrenntes Kompilieren](#)
 - [Java und make](#)
 - [8.4 Zusammenfassung](#)

8.1 Programmelemente

- [8.1 Programmelemente](#)
 - [8.1.1 Anweisungen](#)
 - [8.1.2 Blöcke](#)
 - [8.1.3 Methoden](#)
 - [8.1.4 Klassen](#)
 - [8.1.5 Pakete](#)
 - [8.1.6 Applikationen](#)
 - [8.1.7 Applets](#)

Wie in jeder Programmiersprache kann man auch in Java Strukturen erkennen, die auf den unterschiedlichen Abstraktionsebenen die Bestandteile eines Programms bilden. Die Kenntnis und Bezeichnung dieser Strukturen ist hilfreich für das Verständnis der entwickelten Programme und für die Kommunikation mit anderen Programmierern.

Wir wollen uns dazu in diesem Kapitel die folgenden Programmelemente ansehen:

- Anweisungen
- Blöcke
- Methoden
- Klassen
- Pakete
- Applikationen
- Applets

8.1.1 Anweisungen

Anweisungen gehören zu den elementarsten ausführbaren Programmelementen in Java. Eine Anweisung kann eine Deklaration enthalten, einen Ausdruck auswerten oder den Programmablauf steuern. Wir wollen Anweisungen an dieser Stelle als die kleinsten Bausteine von Java betrachten, und alle anderen Elemente sollen darauf aufbauen.

8.1.2 Blöcke

Ein Block ist eine Kollektion von Anweisungen, die nacheinander ausgeführt werden. Anders als eine einfache Anweisung kann ein Block eigene Variablen definieren, die nur innerhalb des Blocks sichtbar sind. Sie werden angelegt, wenn der Block aufgerufen wird, und zerstört, wenn er wieder verlassen wird.

Interessant ist ein Block vor allem deshalb, weil er selbst eine *Anweisung* ist. Während er in seiner Zusammensetzung aus vielen verschiedenen Anweisungen bestehen kann, stellt sich ein Block nach außen hin als eine einzige Anweisung dar. Die ganze Semantik einer blockstrukturierten Sprache ist darauf ausgelegt, daß ein Block für die Anweisungen, die ihn verwenden, wie eine homogene Einzelanweisung aussieht.

Rekursive Konstruktionsschemata, wie das hier beschriebene, treten an vielen verschiedenen Stellen und in vielen verschiedenen Erscheinungsformen bei der Programmentwicklung auf. Es gibt Methoden, die sich selbst aufrufen, Strukturvariablen, die andere Variablen enthalten, oder eben Blöcke, die Anweisungen *enthalten* und Anweisungen *sind*. Grundlage von rekursiven *Part-of*-Beziehungen ist immer der Umstand, daß die Kollektion selbst vom Typ der darin enthaltenen Elemente ist.

Die Sichtbarkeit und Lebensdauer von Blöcken in Java entspricht den üblichen Regeln, die auch in anderen Programmiersprachen gültig sind. Lokale Variablen werden angelegt, wenn die Ausführung des Blocks beginnt, und wieder zerstört, wenn der Block verlassen wird. Innerhalb eines Blocks sind die lokalen Variablen des Blocks und die lokalen Variablen der umgebenden Blöcke bzw. der umgebenden Methode sichtbar. Zusätzlich sind die Variablen der Klasse sichtbar, in der die den Block umschließende Methode enthalten ist.

Lokale Variablen verdecken gleichnamige Instanz- oder Klassenvariablen. Durch Voranstellen des `this`-Zeigers kann trotzdem auf sie zugegriffen werden. Dies wird in Java oft ausgenutzt, um im Konstruktor einer Methode Membervariablen zu initialisieren, die denselben Namen wie formale Parameter haben:

Tip

```
001 class Point
002 {
003     private int x, y;
004     public Point(int x, int y)
005     {
006         this.x = x;
007         this.y = y;
008     }
009 }
```

Listing 8.1: Zugriff auf verdeckte Membervariablen

Diese Vorgehensweise hat den Vorteil, daß man sich für die formalen Parameter nicht extra Namen ausdenken muß, die sich von den zugehörigen Instanzvariablen unterscheiden.

Im Gegensatz zu den meisten anderen Programmiersprachen gibt es in Java die Regel, daß *lokale* Variablen sich nicht gegenseitig verdecken dürfen. Es ist also nicht erlaubt, innerhalb eines Blocks eine lokale Variable zu deklarieren, die unter demselben Namen bereits als lokale Variable sichtbar ist. Weitere Details zu Blöcken sind in [Kapitel 6](#) zu finden.

Warnung

8.1.3 Methoden

Methoden sind wie Blöcke Kollektionen, die Deklarationen und Anweisungen enthalten können. Genaugenommen enthalten sie neben dem Funktionskopf genau *einen* Block, der alle anderen Elemente enthält. Sie unterscheiden sich von Blöcken folgendermaßen:

- sie haben einen Namen und können von verschiedenen Stellen des Programms aus aufgerufen werden
- sie sind parametrisierbar, und ihr Verhalten ist so zur Laufzeit strukturiert veränderbar
- sie können einen Rückgabewert haben, mit dem ein Wert an den Aufrufer zurückgegeben werden kann

Methoden werden in Java immer *lokal zu einer Klasse* definiert. Klassenlose Funktionen, wie sie beispielsweise in C++ zur Verfügung stehen, gibt es in Java nicht. In diesem Sinne ist Java eine wirklich objektorientierte Programmiersprache, denn die Methoden operieren immer auf den Daten eines bestimmten Objekts, zu dem sie aufgerufen werden.

Hinweis

Allerdings gibt es noch die *Klassenmethoden*. Sie werden zwar auch innerhalb einer Klasse definiert, benötigen aber später kein Objekt zur Ausführung. Statt dessen entsprechen Klassenmethoden eher den globalen Funktionen anderer Programmiersprachen und haben somit keinen Zugriff auf die Membervariablen eines Objekts. Im Unterschied zu gewöhnlichen Funktionen werden Klassenmethoden aber innerhalb einer Klasse definiert und besitzen damit einen eigenen Namensraum, der sich über die Methoden der aktuellen Klasse erstreckt. Der Zugriff auf eine Klassenmethode erfordert immer die Verwendung eines qualifizierten Namens, der sich aus dem Klassennamen, einem Punkt und dem eigentlichen Methodennamen zusammensetzt.

8.1.4 Klassen

Klassen sind das wichtigste Strukturierungsmittel objektorientierter Sprachen. Eine Klasse enthält eine Menge von Variablen, die den *Zustand* von Objekten dieser Klasse beschreiben, und eine Menge von Methoden, die das *Verhalten* der Objekte festlegen.

Klassen sind insofern schachtelbar, als ihre Instanzvariablen vom Typ einer Klasse sein können. Dabei ist es insbesondere erlaubt, daß die Membervariablen von demselben Typ wie die zu definierende Klasse sind. Da in Java alle Objekte als Referenzen abgelegt werden, können auf diese Weise rekursive Datentypen erzeugt und zur Konstruktion von dynamischen Datenstrukturen verwendet werden.

8.1.5 Pakete

In großen Programmsystemen reichen Klassen als Strukturelemente alleine nicht aus. Deshalb bietet Java mit den *Packages* (oder Paketen) oberhalb der Ebene der Klassen eine weitere Kollektion für Programmelemente an.

Ein Paket ist eine Sammlung von Klassen, die einen gemeinsamen Zweck verfolgen oder aus anderen Gründen zusammengefaßt werden sollen. Jede Klasse in Java ist Bestandteil genau eines Pakets. Paketnamen können aus mehreren Teilen bestehen und beliebig tiefe Hierarchien ausdrücken.

Der Name einer Methode oder einer Variablen besteht damit grundsätzlich aus drei Elementen:

- Paketname
- Klassen- oder Objektname
- Methoden- bzw. Variablenname

Ein Beispiel für einen Methodennamen ist `java.lang.Math.sqrt`. Wir werden später Mechanismen kennenlernen, mit denen es möglich ist, die Namen bei ihrer Verwendung abzukürzen.

8.1.6 Applikationen

Applikationen bilden eigenständige Programme in Java. Sie benötigen keinen Browser zur Ausführung, sondern nur den Java-Interpreter und die `.class`-Files der verwendeten Klassen.

Technisch betrachtet ist eine Applikation nicht mehr als eine einzelne Klasse, in der eine Methode vom Typ `public static void main` definiert wurde. Jede Klasse, die eine solche Methode enthält, kann als Applikation verwendet werden.

Durch einfaches Hinzufügen einer Methode `public static void main` kann also jede beliebige Klasse sehr leicht in eine Applikation verwandelt und vom Java-Interpreter aufgerufen werden. Dies kann beispielsweise nützlich sein, um in Low-Level-Klassen, die eigentlich nur als Dienstleister auftreten, eigenständig ausführbaren Testcode unterzubringen, oder um eine solche Klasse mit Benutzungshinweisen auszustatten, die der Entwickler durch einfaches Starten der Klasse als Applikation abrufen kann.

Tip

8.1.7 Applets

Applets sind ebenfalls lauffähige Java-Programme. Anders als Applikationen werden sie aus einer HTML-Seite heraus aufgerufen und benötigen zur Ausführung einen Web-Browser (oder ein Werkzeug wie den *Appletviewer*, das sich in gewissem Sinne wie ein Web-Browser verhält).

Applets werden nicht durch die Methode `public static void main` gestartet, sondern müssen aus der Klasse `Applet` abgeleitet und nach deren Architekturmerkmalen konstruiert werden. Zum Starten des Programms erzeugt der Browser eine Instanz der abgeleiteten Klasse und ruft nacheinander eine Reihe vordefinierter *Callback-Methoden* auf. Callback-Methoden sind Methoden, die von der abgeleiteten Klasse zur Verfügung gestellt und vom Browser oder *AppletViewer* aufgerufen werden. Weitere Details zur Applet-Programmierung finden sich in [Kapitel 25](#).

Hinweis

8.2 Pakete

- [8.2 Pakete](#)
 - [8.2.1 Verwendung von Paketen](#)
 - [Der Import von java.lang](#)
 - [Die vordefinierten Pakete im JDK](#)
 - [8.2.2 Die Bedeutung der Paketnamen](#)
 - [Laden von Klassen im JDK 1.0 und 1.1](#)
 - [Laden von Klassen im JDK 1.2](#)
 - [Die Dateien classes.zip und rt.jar](#)
 - [Umgekehrte Domain-Namen](#)
 - [8.2.3 Einbinden zusätzlicher Pakete](#)
 - [8.2.4 Erstellen eigener Pakete](#)
 - [Benannte Pakete](#)
 - [Das Default-Paket](#)
 - [Das public-Attribut](#)

8.2.1 Verwendung von Paketen

Jede Klasse in Java ist Bestandteil eines Pakets. Der vollständige Name einer Klasse besteht aus dem Namen des Pakets, gefolgt von einem Punkt, der vom eigentlichen Namen der Klasse gefolgt wird. Der Name des Pakets selbst kann ebenfalls einen oder mehrere Punkte enthalten.

Damit eine Klasse verwendet werden kann, muß angegeben werden, in welchem Paket sie liegt. Hierzu gibt es zwei unterschiedliche Möglichkeiten:

- Die Klasse wird über ihren vollen (qualifizierten) Namen angesprochen:

```
java.util.Date d = new java.util.Date();
```

- Am Anfang des Programms werden die gewünschten Klassen mit Hilfe einer [import](#)-Anweisung eingebunden:

```
import java.util.*;
...
Date d = new Date();
```

Die Verwendung voll qualifizierter Namen hat den Nachteil, daß die Klassennamen sehr lang und unhandlich werden. Bequemer ist daher die Anwendung der zweiten Variante, bei der die benötigten Klassen mit Hilfe einer [import](#)-Anweisung dem Compiler bekannt gemacht werden.

Die [import](#)-Anweisung gibt es in zwei unterschiedlichen Ausprägungen:

- Mit der Syntax `import Paket.Klasse;` wird genau eine Klasse importiert. Alle anderen Klassen des Pakets bleiben unsichtbar:

```
import java.util.Date;
...
Date d = new Date();
java.util.Vector v = new java.util.Vector();
```

- Unter Verwendung der Syntax `import Paket.*;` können alle Klassen des angegebenen Pakets auf einmal importiert werden:

```
import java.util.*;
...
Date d = new Date();
Vector v = new Vector();
```


Im Gegensatz zu ähnlichen Konstrukten in anderen Sprachen ist die Verwendung der zweiten Variante der `import`-Anweisung nicht zwangsläufig wesentlich ineffizienter als die der ersten. In der Sprachspezifikation wird sie als *type import on demand* bezeichnet, was bedeutet, daß die Klasse erst dann in den angegebenen Paketen gesucht wird, wenn das Programm sie wirklich benötigt. Keinesfalls muß der Compiler beim Parsen der `import`-Anweisung zwangsläufig alle Klassendateien des angegebenen Pakets in den Hauptspeicher laden oder ähnlich zeit- und speicheraufwendige Dinge machen. Es schadet also im allgemeinen nichts, die zweite Variante der `import`-Anweisung zu verwenden.

Hinweis

Der Import von java.lang

In vielen verschiedenen Beispielen in diesem Buch werden Klassennamen (wie beispielsweise `String`, `Thread` oder `Object`) verwendet, ohne daß eine zugehörige `import`-Anweisung zu erkennen wäre. In diesem Fall entstammen die Klassen dem Paket `java.lang`.

Dieses Paket wurde von den Entwicklern der Sprache als so wichtig angesehen, daß es bei jedem Compilerlauf automatisch importiert wird. Man kann sich das so vorstellen, als wenn am Anfang jeder Quelldatei implizit die folgende Anweisung stehen würde:

```
import java.lang.*;
```

Ein expliziter Import von `java.lang` ist daher niemals nötig. Alle anderen Pakete müssen jedoch vor ihrer Verwendung importiert werden, wenn auf die Anwendung voll qualifizierter Klassennamen verzichtet werden soll.

Tip

Die vordefinierten Pakete im JDK

Während beim Wechsel der Java-Versionen die Sprachspezifikation relativ stabil geblieben ist, hat sich der Umfang der Laufzeitbibliothek um ein Vielfaches erhöht. Dies zeigt sich unter anderem an der gestiegenen Anzahl an vordefinierten Paketen, die mit dem JDK ausgeliefert werden (siehe [Tabelle 8.1](#)). Sie stieg von 8 Standardpaketen im JDK 1.0 auf 22 im JDK 1.1 und auf über 50 im JDK 1.2.

Paket	Bedeutung
java.applet	Applets
java.awt	Abstract Windowing Toolkit
java.awt.color	Unterstützung für unterschiedliche Farbsysteme
java.awt.datatransfer	AWT Clipboard-Funktionen
java.awt.dnd	Drag & Drop
java.awt.font	Erweiterte Unterstützung für Schriftarten
java.awt.geom	2D-Bildverarbeitung
java.awt.im	Eingabe von fernöstlichen Schriftzeichen
java.awt.image	AWT-Bildverarbeitung
java.awt.image.renderable	Darstellungsunabhängige Bildverarbeitung
java.awt.print	Druckausgabe
java.beans	Java Beans
java.beans.beancontext	Java Beans-Unterstützung
java.io	Bildschirm- und Datei-I/O
java.lang	Elementare Sprachunterstützung
java.lang.reflect	Reflection-API
java.lang.ref	Referenz-Objekte
java.math	Fließkomma-Arithmetik
java.net	Netzwerkunterstützung
java.rmi	Remote Method Invocation (RMI)
java.rmi.dgc	RMI Distributed Garbage Collection
java.rmi.registry	RMI Service Registry
java.rmi.server	RMI-Support
java.security	Security-Dienste
java.security.acl	Access Control Lists

java.security.cert	Zertifikate
java.security.interfaces	DSA-Interfaces
java.security.spec	Sicherheitsspezifikationen
java.sql	Datenbankzugriff (JDBC)
java.text	Internationalisierung
java.util	Diverse Utilities und Datenstrukturen
java.util.jar	Zugriff auf JAR-Files
java.util.mime	Multipurpose Internet Mail Extensions (MIME)
java.util.zip	Zugriff auf ZIP-Files

Tabelle 8.1: Die vordefinierten Pakete des JDK

Neben den Standardpaketen gibt es seit der Version 1.2 des JDK eine Reihe von *Standarderweiterungen*, deren Paktenamen mit [javax.](#) beginnt. Sie stellen erweiterte Funktionalitäten in einem oder mehreren .jar-Dateien zur Verfügung und werden typischerweise im Unterverzeichnis [lib/ext](#) des JDK installiert. Im Gegensatz zu den Standardpaketen des JDK sind sie nicht unbedingt Bestandteil jedes Java-Entwicklungssystems und müssen nicht auf allen Plattformen zur Verfügung stehen. Sie stellen häufig gebrauchte Erweiterungen zur Verfügung, deren Umfang die reinen Kernbibliotheken um ein Vielfaches übertrifft.

JDK1.1/1.2

8.2.2 Die Bedeutung der Paketnamen

Paketnamen bestehen in Java aus mehreren Komponenten, die jeweils durch einen Punkt voneinander getrennt sind. Neben der Aufgabe, die Paketnamen *visuell* zu strukturieren, hat die Unterteilung aber noch eine andere, sehr viel wichtigere Bedeutung.

Jeder Teil eines mehrstufigen Paketnamens bezeichnet nämlich ein Unterverzeichnis auf dem Weg zu der gewünschten Klassendatei. Soll beispielsweise eine Klasse aus dem Paket [java.awt.image](#) eingebunden werden, sucht es der Java-Compiler im Unterverzeichnis [java\awt\image](#). Soll dagegen eine Klasse aus dem Paket [com.sun.image.codec.jpeg](#) geladen werden, wird es im Unterverzeichnis [com\sun\image\codec\jpeg](#) gesucht. Interessant ist in diesem Zusammenhang natürlich die Frage, in welchem Verzeichnis der Compiler mit der Suche beginnt. Bei der Antwort darauf muß zwischen dem aktuellen JDK 1.2 und seinen Vorgängerversionen 1.0 und 1.1 unterschieden werden.

Laden von Klassen im JDK 1.0 und 1.1

Wurde keine Umgebungsvariable [CLASSPATH](#) angegeben und der Schalter `-classpath` beim Compiler-Aufruf nicht verwendet, so suchen der Java-Compiler und die übrigen Tools in einem systemspezifischen Installationsverzeichnis (z.B. [c:\java1.1.7\lib](#) beim JDK 1.1.7) und zusätzlich im aktuellen Verzeichnis nach den Klassendateien.

Anders als im JDK 1.0 ist es bei einer Standardinstallation des JDK 1.1 unter Windows 95 nicht erforderlich, den [CLASSPATH](#) explizit zu setzen. Alle Tools generieren den [CLASSPATH](#) implizit aus der Position des [bin](#)-Verzeichnisses (z.B. [c:\java1.1.7\bin](#)) nach folgendem Schema:

```
.;[bin]\..\classes;[bin]\..\lib\classes.zip
```

[\[bin\]](#) steht hier für den Pfad des [bin](#)-Verzeichnisses. Nur wenn die Klassendateien in einem anderen Verzeichnis liegen oder Klassendateien in weiteren Verzeichnissen eingebunden werden sollen, muß die [CLASSPATH](#)-Variable manuell geändert werden.

Existiert zum Zeitpunkt des Compiler-Aufrufs die Umgebungsvariable [CLASSPATH](#), beginnt der Compiler die Suche nach den eingebundenen Klassen in allen im [CLASSPATH](#) angegebenen Verzeichnissen. Die einzelnen Verzeichnisse werden durch ein Semikolon (bzw. einen Doppelpunkt unter UNIX) voneinander getrennt.

Beim Aufruf des Compilers kann der Schalter `-classpath`, gefolgt von einer Liste von Verzeichnisnamen, übergeben werden. Er hat dieselbe Bedeutung wie die Umgebungsvariable [CLASSPATH](#) und definiert die Verzeichnisse, in denen der Compiler nach den [.class](#)-Dateien sucht. Der Compiler-Schalter hat dabei Vorrang gegenüber einer möglichen Umgebungsvariablen.

Laden von Klassen im JDK 1.2

Die Verwendung der [CLASSPATH](#)-Variable hat immer wieder zu Schwierigkeiten und Mißverständnissen geführt. Es ist insbesondere häufig passiert, daß durch falsches Setzen der Umgebungsvariable die Systemklassen selbst nicht mehr gefunden werden konnten und auf diese Weise das gesamte Laufzeitsystem unbenutzbar wurde.

Seit dem JDK 1.2 wurde daher die Bedeutung der [CLASSPATH](#)-Umgebungsvariable dahingehend verändert, daß sie nur noch zur Suche der *benutzerspezifischen* Klassen verwendet wird. Alle Standardpakete und Standarderweiterungen (beide zusammen werden im JDK 1.2 *Bootstrap Classes* genannt) werden dagegen unabhängig vom [CLASSPATH](#) mit Hilfe der auf das Installationsverzeichnis verweisenden Systemeigenschaft

[sun.boot.class.path](#) gefunden. Sie wird bei der JDK-Installation automatisch gesetzt und sollte später nicht mehr verändert werden. Der [CLASSPATH](#) braucht also nur noch dann explizit gesetzt zu werden, wenn benutzerspezifische Klassen vorhanden sind, die nicht im aktuellen Verzeichnis liegen (letzeres wird ebenfalls automatisch durchsucht).

Falls Sie vor dem JDK 1.2 ältere Versionen des Java Development Kit auf ihrem Rechner installiert hatten, überprüfen Sie bitte nach der Installation die Umgebungsvariablen. Sorgen Sie dafür, daß nicht ein veralteter [CLASSPATH](#) auf Verzeichnisse oder Dateien verweist, aus denen das Laufzeitsystem versehentlich unbrauchbare Klassendateien laden würde.

Warnung

Die Dateien `classes.zip` und `rt.jar`

Im Installationsverzeichnis von JDKs der Versionen 1.0 und 1.1 findet man meist eine Datei [classes.zip](#) anstelle der erwähnten Unterverzeichnisse mit den Klassendateien. Aus Gründen der Performance beim Übersetzen haben sich die Entwickler entschlossen, alle Standardklassendateien in diesem Archiv abzulegen, um einen schnelleren Lesezugriff auf sie zu ermöglichen.

Die Datei [classes.zip](#) sollte nicht ausgepackt werden, denn der Compiler verwendet sie in archivierter Form. Obwohl beim Einpacken der Klassen auf die Komprimierung verzichtet wurde, bietet diese Form der Archivierung den Vorteil, daß viele kleine Dateien in einer einzigen großen zusammengefaßt werden. Zeitaufwendige Verzeichniszugriffe und -wechsel können so entfallen, wenn der Compiler nach Klassennamen suchen muß oder den Inhalt einer Klassendatei lesen will.

Warnung

Um dem Compiler diese Art der Speicherung der Klassendateien bekannt zu machen, muß in der [CLASSPATH](#)-Umgebungsvariable nicht nur das Verzeichnis, sondern auch der Name der `.zip`-Datei angegeben werden, z.B.:

`CLASSPATH=.;c:\java\LIB\CLASSES.ZIP`

Seit dem JDK 1.2 gibt es die Datei [classes.zip](#) nicht mehr. Die Klassenbibliotheken liegen nun als `.jar`-Dateien (z.B. [rt.jar](#)) vor und befinden sich im Unterverzeichnis `jre/lib` der JDK-Installation. Wie oben erwähnt, werden sie unabhängig vom Inhalt der [CLASSPATH](#)-Umgebungsvariable gefunden. Weitere Informationen zu `.jar`-Dateien finden Sie in [Kapitel 26](#).

JDK1.1/1.2

Umgekehrte Domain-Namen

Die Entwickler von Java haben sich einen Mechanismus ausgedacht, um auch bei sehr großen Projekten, an denen möglicherweise viele Entwickler beteiligt sind, Namenskollisionen zwischen den beteiligten Klassen und Paketen zu vermeiden. Auch die Verwendung einer großen Anzahl unterschiedlicher Klassenbibliotheken von verschiedenen Herstellern sollte möglich sein, ohne daß Namensüberschneidungen dies schon im Keim ersticken.

Um diese Probleme zu lösen, hat sich das Java-Team eine Vorgehensweise zur Vergabe von Paketnamen überlegt, die an das *Domain-Namen*-System bei Internet-Adressen angelehnt ist. Danach sollte jeder Anbieter seine Pakete entsprechend dem eigenen Domain-Namen benennen, dabei allerdings die Namensbestandteile in umgekehrter Reihenfolge verwenden.

So sollten beispielsweise die Klassen der Firma Sun, deren Domain-Name [sun.com](#) ist, in einem Paket `com.sun` oder in darunter befindlichen Subpaketen liegen. Da die Domain-Namen weltweit eindeutig sind, werden Namenskollisionen zwischen Paketen unterschiedlicher Hersteller auf diese Weise von vornherein vermieden. Beispiele für derartige Paketnamen liefert die Standardinstallation gleich mit. So stellt das JDK 1.2 diverse Pakete `com.sun.*` zur Verfügung. Sie gehören nicht zum Standard-Sprachumfang eines Java-Entwicklungssystems, sondern werden von SUN als eigenständige Erweiterung mit dem JDK ausgeliefert.

Unterhalb des Basispakets können Unterpakete beliebig geschachtelt werden. Die Namensvergabe liegt dabei in der Entscheidung des Unternehmens. Gibt es beispielsweise die Abteilungen `is`, `se` und `tx` in einem Unternehmen mit der Domain `tl.de`, kann es sinnvoll sein, diese Abteilungsnamen auch als Unterprojekte zu verwenden. Die von diesen Abteilungen erstellten Klassen würden dann in den Paketen `de.tl.is`, `de.tl.se` und `de.tl.tx` liegen.

Hinweis

Der Vollständigkeit halber sollte man anmerken, daß sich das hier beschriebene System in der Praxis noch nicht komplett durchgesetzt hat und insbesondere die Klassen der `java.*`- und `javax.*`-Hierarchie Ausnahmen bilden. Es wird mit der zunehmenden Anzahl allgemein verfügbarer Pakete jedoch an Bedeutung gewinnen. Manche Entwickler verwenden ein leicht abgewandeltes Schema, bei dem nur die Top-Level-Domain ausgelassen wird. Die Paketnamen beginnen in diesem Fall nicht mit `org` oder `com`, sondern direkt mit dem zweiten Teil des Domain-Namens (oder einem ähnlichen herstellerspezifischen Kürzel). Dadurch werden sie etwas kürzer und sind leichter zu handhaben.

8.2.3 Einbinden zusätzlicher Pakete

In der Praxis wird man neben der Klassenbibliothek des JDK häufig zusätzliche Pakete von Drittanbietern verwenden wollen. Um den Klassenpfad nicht unnötig lang werden zu lassen, empfiehlt es sich, die Pakete in einem gemeinsamen Verzeichnis abzulegen. Falls sich alle Entwickler von Libraries an das oben besprochene Schema zur Vergabe von Paketnamen halten, kann es keine Überschneidungen geben.

Als Beispiel wollen wir einige Java-Klassen des Autors (zu finden als Datei `gkjava.zip` unter <http://www.gkrueger.com> oder auf der CD-ROM zum Buch im Verzeichnis `\misc`) und die Utilities der »ACME-Labs« von Jef Poskanzer (<http://www.acme.com>) installieren:

- Zunächst legen wir ein gemeinsames Unterverzeichnis für unsere Klassendateien an, beispielsweise `c:\classes`.
- Nun wird die `.zip`-Datei mit den Klassen des Autors geladen und im Verzeichnis `c:\classes` ausgepackt. Da alle Paketenamen mit `gk` beginnen, landen alle Dateien im Unterverzeichnis `gk`.
- Auf die gleiche Weise packen wir die Klassendateien von Jef Poskanzer aus. Alle Klassen liegen im Paket `acme` oder in Unterpaketen und landen damit im Unterverzeichnis `acme`.
- Nun braucht lediglich noch der `CLASSPATH` so gesetzt zu werden, daß er das Verzeichnis `c:\classes` beinhaltet:

```
set CLASSPATH=.;c:\classes
```

Alle Libraries, die sich an diese Konventionen halten, können in der beschriebenen Weise installiert werden. Probleme gibt es nur, wenn ein Anbieter seine Klassendateien nicht in Paketen ablegt. In diesem Fall müßten die Klassendateien in das aktuelle Verzeichnis oder nach `c:\classes` kopiert werden. Das würde bei Klassendateien von mehr als einem Anbieter natürlich schnell zum Chaos führen, läßt sich aber nicht so einfach ändern. Es bieten sich zwei Handlungsalternativen an:

- Wenn man die Pakete in unterschiedlichen Unterverzeichnissen ablegt, benötigen die Programme einen relativ langen und möglicherweise schwer zu pflegenden Klassenpfad, in dem alle benötigten Verzeichnisse verzeichnet sind.
- Wenn man die »paketlosen« Dateien verschiedener Anbieter in unterschiedliche Unterverzeichnisse von `c:\classes` legen will, müßte man dazu die Quelldateien um die entsprechenden `package`-Anweisungen (siehe nachfolgenden Abschnitt) ergänzen.

Beide Varianten sind unbefriedigend, und es bleibt zu hoffen, daß die Anbieter von Java-Klassenbibliotheken sich verstärkt an die Namenskonventionen des Java-Teams halten werden.

8.2.4 Erstellen eigener Pakete

Benannte Pakete

Bisher wurde nur gezeigt, wie man Klassen aus fremden Paketen verwendet, nicht aber, wie man selbst Pakete erstellt. Glücklicherweise ist das aber keine Aufgabe für Spezialisten, sondern sehr einfach mit Bordmitteln realisierbar.

Um eine Klasse einem ganz bestimmten Paket zuzuordnen, muß lediglich am Anfang des Quelltextes eine geeignete `package`-Anweisung verwendet werden. Diese besteht (analog zur `import`-Anweisung) aus dem Schlüsselwort `package` und dem Namen des Pakets, dem die nachfolgende Klasse zugeordnet werden soll. Die `package`-Anweisung muß als erste Anweisung in einer Quelldatei stehen, so daß der Compiler sie noch vor den `import`-Anweisungen findet.

Der Aufbau und die Bedeutung der Paketnamen in der `package`-Anweisung entspricht exakt dem der `import`-Anweisung. Der Compiler löst ebenso wie beim `import` den dort angegebenen hierarchischen Namen in eine Kette von Unterverzeichnissen auf, an deren Ende die Quelldatei steht. Neben der Quelldatei wird auch die Klassendatei in diesem Unterverzeichnis erstellt.

Da der Java-Compiler eingebundene Quelldateien, die noch nicht übersetzt sind, während der Übersetzung einer anderen Klasse automatisch mit übersetzt, ist das Erstellen eines neuen Pakets sehr einfach. Wir wollen uns ein Beispiel ansehen, bei dem zwei Pakete `demo` und `demo.tools` angelegt werden und die darin enthaltenen Klassen `A`, `B` und `C` in einer Klasse `PackageDemo` verwendet werden. Am einfachsten ist es, in den folgenden Schritten vorzugehen:

Beispiel

Wir gehen davon aus, daß der `CLASSPATH` das aktuelle Verzeichnis enthält (also beispielsweise den Inhalt `.;c:\classes` hat) und legen im aktuellen Verzeichnis die Unterverzeichnisse `demo` und `demo\tools` an.

- Zunächst wird im Unterverzeichnis `demo` die Datei `A.java` angelegt:

[demo\A.java](#)

```
001    package demo;
002
003    public class A
004    {
005        public void hello()
006        {
007            System.out.println("Hier ist A");
008        }
009    }
010
```

Listing 8.2: Die Klasse A des Pakets demo

Sie enthält die Anweisung `package demo;`, um anzuzeigen, daß die Klasse `A` zum Paket `demo` gehört.

- Im Unterverzeichnis `demo` wird weiterhin die Datei `B.java` angelegt:

[demo/B.java](#)

```
001    package demo;
002
003    public class B
004    {
005        public void hello()
006        {
007            System.out.println("Hier ist B");
008        }
009    }
010
```

Listing 8.3: Die Klasse B des Pakets demo

Auch diese Quelldatei enthält die Anweisung `package demo;`, um anzuzeigen, daß die Klasse zum Paket `demo` gehört.

- Im Unterverzeichnis `demo\tools` wird die Datei `C.java` angelegt:

[demo/tools/C.java](#)

```
001    package demo.tools;
002
003    public class C
004    {
005        public void hello()
006        {
007            System.out.println("Hier ist C");
008        }
009    }
010
```

Listing 8.4: Die Klasse C des Pakets demo.tools

Diese Quelldatei enthält die Anweisung `package demo.tools;`, um anzuzeigen, daß die Klasse zum Paket `demo.tools` gehört.

- Nun wird im Stammverzeichnis die Datei `PackageDemo.java` angelegt:

[PackageDemo.java](#)

```
001    import demo.*;
002    import demo.tools.*;
003
004    public class PackageDemo
005    {
006        public static void main(String[] args)
007        {
008            (new A()).hello();
009            (new B()).hello();
010            (new C()).hello();
011        }
012    }
013
```

Listing 8.5: Verwendung der Klassen aus selbstdefinierten Paketen

Ohne vorher die Klassen `A`, `B` oder `C` separat übersetzen zu müssen, kann nun einfach `PackageDemo` kompiliert werden. Der Compiler erkennt die Verwendung von `A`, `B` und `C`, findet die Paketverzeichnisse `demo` und `demo\tools` und erkennt, daß die Quellen noch nicht übersetzt wurden. Er erzeugt dann aus den `.java`-Dateien die zugehörigen `.class`-Dateien (in demselben Verzeichnis wie die Quelldateien), bindet sie ein und übersetzt schließlich die Klasse `PackageDemo`. Die Ausgabe des Programms ist:

```
Hier ist A
Hier ist B
Hier ist C
```

Das Default-Paket

Würde nur dann ein eigenes Paket erzeugt werden, wenn die Quelldatei eine `package`-Anweisung enthält, müßte man sich fragen, zu welchem Paket die Klassen gehören, in deren Quelldatei die `package`-Anweisung fehlt (was ja erlaubt ist). Die Antwort auf diese Frage liegt darin, daß es in Java ein *Default-Paket* gibt, das genau dann verwendet wird, wenn keine andere Zuordnung getroffen wurde.

Das Default-Paket ist ein Zugeständnis an kleinere Programme oder einfache Programmierprojekte, bei denen es sich nicht lohnt, eigene Pakete anzulegen. Ohne Teile des Projektes in Unterverzeichnissen abzulegen und durch `import`- und `package`-Anweisungen unnötigen Aufwand zu

treiben, ist es auf diese Weise möglich, Quelldateien einfach im aktuellen Verzeichnis abzulegen, dort zu kompilieren und automatisch einzubinden. Klassen des Default-Pakets können ohne explizite `import`-Anweisung verwendet werden.

Ein Java-Compiler braucht laut Spezifikation nur ein einziges Default-Paket zur Verfügung zu stellen. Typischerweise wird dieses Konzept aber so realisiert, daß jedes unterschiedliche Verzeichnis die Rolle eines Default-Paketes übernimmt. Auf diese Weise lassen sich beliebig viele Default-Pakete erzeugen, indem bei Bedarf einfach ein neues Unterverzeichnis angelegt wird und die Quelldateien eines Java-Projektes dort abgelegt werden.

Hinweis

Das public-Attribut

Es gibt noch eine wichtige Besonderheit bei der Deklaration von Klassen, die von anderen Klassen verwendet werden sollen. Damit eine Klasse `A` eine andere Klasse `B` einbinden darf, muß nämlich eine der beiden folgenden Bedingungen erfüllt sein:

- entweder gehören `A` und `B` zu demselben Paket oder
- die Klasse `B` wurde als `public` deklariert

Wenn also nur Default-Pakete verwendet werden, spielt es keine Rolle, ob eine Klasse vom Typ `public` ist, denn alle Klassen liegen in demselben Paket. Werden aber Klassen aus externen Paketen eingebunden, so gelingt das nur, wenn die einzubindende Klasse vom Typ `public` ist. Andernfalls verweigert der Compiler deren Einbindung und bricht die Übersetzung mit einer Fehlermeldung ab.

Warnung

8.3 Der Entwicklungszyklus

- 8.3 Der Entwicklungszyklus
 - 8.3.1 Schematische Darstellung
 - 8.3.2 Projektverwaltung
 - Getrenntes Kompilieren
 - Java und make

8.3.1 Schematische Darstellung

Der Turn-around-Zyklus beim Entwickeln von Java-Programmen unterscheidet sich in mehrfacher Hinsicht von dem in traditionellen kompilierten Programmiersprachen.

- Der Compiler erzeugt keine direkt ausführbaren Programme, sondern `.class`-Files, die von einem Java-Interpreter ausgeführt werden. Mittlerweile arbeiten bereits mehrere Betriebssystem-Hersteller daran, `.class`-Files direkt ausführbar zu machen, und es gibt schon Prozessoren, die den Maschinencode in `.class`-Files ohne zwischengeschalteten Interpreter verstehen.
- Es gibt keinen expliziten Link-Lauf, denn die verschiedenen `.class`-Files werden zur Ausführungszeit gebunden.
- Die `.class`-Files eines einzigen Projekts können auf unterschiedlichen Plattformen mit unterschiedlichen Compilern erzeugt werden.
- Die `.class`-Files lassen sich auf allen Plattformen ausführen, die einen Java-Interpreter besitzen; unabhängig davon, wo sie ursprünglich entwickelt wurden.

[Abbildung 8.1](#) zeigt den schematischen Ablauf bei der Entwicklung eines Java-Programms, das aus den Klassen `A`, `B` und `C` besteht.

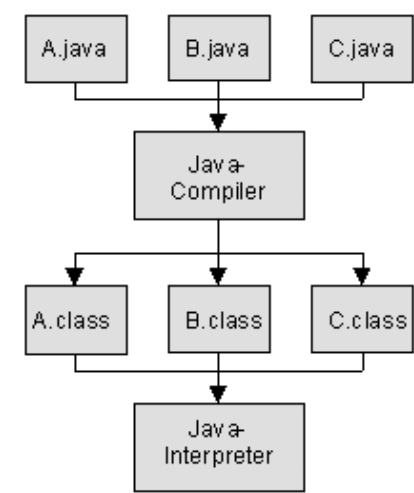


Abbildung 8.1: Der Entwicklungszyklus in Java

Da der Java-Bytecode plattformunabhängig ist, hätten die Klassen `A`, `B` und `C` aber auch ebenso gut von verschiedenen Entwicklern auf drei unterschiedlichen Plattformen entwickelt werden können, um später auf einer vierten Plattform ausgeführt zu werden. [Abbildung 8.2](#) zeigt dies beispielhaft.

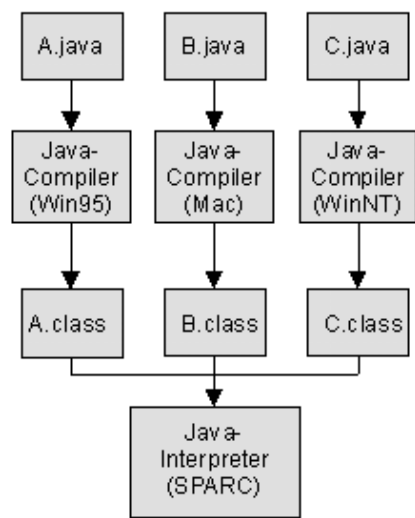


Abbildung 8.2: Plattformübergreifende Entwicklung in Java

8.3.2 Projektverwaltung

Getrenntes Kompilieren

Wie die Abbildungen deutlich machen, spielen die `.class`-Files für den Entwicklungsprozeß und die Portierbarkeit von Java-Programmen eine entscheidende Rolle. Jedes `.class`-File enthält den Bytecode für eine übersetzte Klasse. Zusätzlich sind in ihr Informationen enthalten, um dynamisches Linken und Late-Bindung zu unterstützen und gegebenenfalls Symbole, Zeilennummern und andere Informationen für den Debugger zur Verfügung zu stellen.

Eine der Grundregeln bei der Entwicklung von Java-Programmen ist es, genau eine Klassendefinition je Quelldatei aufzunehmen. Anders als etwa in C++ ist es in Java grundsätzlich nicht möglich, die Quellen einer einzigen Klasse auf mehrere Dateien zu verteilen (und auf diese Weise *weniger* als eine Klasse in einer Quelldatei zu speichern). Obwohl es - wie wir gleich sehen werden - möglich ist, *mehr* als eine Klasse in einer Quelldatei abzulegen, sollte dies in der Regel nicht getan werden. Wenn diese Regeln eingehalten werden, ergibt sich eine eindeutige Abbildung zwischen Klassen und Quelldateien, die bei der sauberen Strukturierung eines Projektes hilft. Durch die Verwendung von Paketen kann eine Verteilung der Quelltexte auf verschiedene Verzeichnisse und Unterverzeichnisse erreicht werden.

Hinweis

Wie bereits angedeutet, dürfen tatsächlich auch zwei oder mehr Klassen in einer Quelldatei enthalten sein. Voraussetzung dafür ist allerdings, daß höchstens eine von ihnen als `public` deklariert wurde. Eine solche Vorgehensweise kann beispielsweise bei sehr kleinen Projekten, die nur aus ganz wenigen Klassen bestehen, sinnvoll sein, um alle Klassen in eine einzige Datei zu bekommen. Sie kann auch angewendet werden, wenn kleine Hilfsklassen benötigt werden, die nur für eine einzige andere Klasse von Bedeutung sind. Der Compiler erzeugt in jedem Fall eine separate `.class`-Datei für jede Klasse, die in einer Quelldatei enthalten ist. Wurde in einer Quelldatei mehr als eine Klasse `public` deklariert, so gibt es einen Compiler-Fehler.

Wichtig ist in dem Zusammenhang auch, daß der Name der `public`-Klasse und der Name der Quelldatei identisch sein müssen. Dabei muß die Groß- und Kleinschreibung eingehalten werden und auch bei Klassennamen, die länger als 8 Zeichen sind, muß der Dateiname so lang wie der Klassename sein. Klassen- und Dateiname unterscheiden sich also nur durch die Extension `.java`. So befindet sich beispielsweise die Klasse `Integer` in einer Datei mit dem Namen `Integer.java` und die Klasse `InterruptedException` in einer Datei mit dem Namen `InterruptedException.java`. Da die Extension einer Java-Quelldatei in jedem Fall `.java` (und damit vierstellig) ist, ist eine vernünftige Portierung von Java auf Plattformen, die nur 8+3-Dateinamen unterstützen, kaum sinnvoll möglich und bisher auch nur ansatzweise gelungen.

Warnung

Interessanterweise bietet Java volle Typsicherheit auch über die Grenzen von Quelldateien hinweg, ohne daß dazu Header-Dateien oder andere Interface-Beschreibungen nötig wären. Der Compiler verwendet während der Übersetzung einer Java-Klasse die `.class`-Dateien aller eingebundenen Klassen und entnimmt diesen die Signatur der aufgerufenen Methoden. Die Notwendigkeit zur Bereitstellung von separaten Header-Dateien (wie beispielsweise in C++) und das fehlerträchtige Management der Abhängigkeiten zwischen ihnen und ihren Quelltexten entfällt daher.

Java und make

Genau aus diesem Grund ist aber auch die Verwendung des [make](#)-Tools zur Verwaltung der Dateiabhängigkeiten in Java-Projekten schwierig. Anstelle der klar definierten Abhängigkeit einer Quelldatei von einigen Headerdateien besitzt eine Java-Quelle meist eine Vielzahl von (teilweise impliziten oder indirekten) Abhängigkeiten zu anderen Java-Quellen. Diese korrekt zu pflegen ist nicht einfach, und ein [makefile](#) kann leicht unvollständige Abhängigkeitsregeln enthalten. Die daraus entstehenden Inkonsistenzen können zu schwer zu lokalisierenden Laufzeitfehlern führen.

Ein anderes Problem bei der Verwendung von [make](#) ist die Vielzahl der separaten Compiler-Aufrufe. Da der Java-Compiler selbst in Java geschrieben wurde, verursacht jeder Aufruf einen größeren Overhead durch das erforderliche Starten der virtuellen Maschine, der sich bei umfangreichen Projekten in unzumutbaren Wartezeiten niederschlägt.

Aus diesen Gründen ist es bei kleineren und mittleren Projekten manchmal sinnvoll, auf den Einsatz von [make](#) zu verzichten, und die Programme bei Bedarf durch Aufruf von `javac` * `.java` komplett neu zu kompilieren. Der Aufruf-Overhead entsteht in diesem Fall nur einmal, und der Compiler braucht nicht wesentlich länger, als wenn lediglich eine einzige Quelle übersetzt würde.

Tip

8.4 Zusammenfassung

- [8.4 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Anweisungen, Blöcke und Methoden als Bestandteile von Klassen
- Klassen als Bestandteile von Paketen
- Grundsätzlicher Aufbau von Applets und Applikationen
- Verwendung von Pakten und die [import](#)-Anweisung
- Die vordefinierten Pakete der Java-Klassenbibliothek
- Die Struktur der Paketnamen
- Erstellen eigener Pakete
- Schematische Darstellung des Entwicklungszyklus
- Getrenntes Kompilieren und Porjektorganisation

Kapitel 9

Exceptions

- [9 Exceptions](#)
 - [9.1 Grundlagen und Begriffe](#)
 - [9.2 Behandlung von Exceptions](#)
 - [9.2.1 Die try-catch-Anweisung](#)
 - [9.2.2 Das Fehlerobjekt](#)
 - [9.2.3 Die Fehlerklassen von Java](#)
 - [9.2.4 Fortfahren nach Fehlern](#)
 - [9.2.5 Mehr als eine catch-Klausel](#)
 - [9.2.6 Die finally-Klausel](#)
 - [9.3 Weitergabe von Exceptions](#)
 - [9.3.1 Die catch-or-throw-Regel](#)
 - [9.3.2 Weitergabe einer Exception](#)
 - [Die Klasse RuntimeException](#)
 - [9.3.3 Auslösen von Ausnahmen](#)
 - [9.4 Zusammenfassung](#)

9.1 Grundlagen und Begriffe

- [9.1 Grundlagen und Begriffe](#)

Mit den *Exceptions* besitzt Java einen Mechanismus zur strukturierten Behandlung von Fehlern, die während der Programmausführung auftreten. Tritt etwa ein Laufzeitfehler auf, weil ein Array-Zugriff außerhalb der definierten Grenzen erfolgte oder weil eine Datei, die geöffnet werden sollte, nicht gefunden wurde, so gibt es in Java Sprachmittel, die eine systematische Behandlung solcher Ausnahmen ermöglichen.

Da Exceptions ein relativ neues Feature von Programmiersprachen sind, ist es sinnvoll, zunächst die in diesem Zusammenhang verwendeten Begriffe vorzustellen. Als *Exception* wird dabei die eigentliche Ausnahme bezeichnet, die durch ein Programm zur Laufzeit verursacht werden kann. Das *Auslösen einer Ausnahme* wird im Java-Sprachgebrauch als *throwing* bezeichnet, wir werden meist die deutsche Bezeichnung *auslösen* verwenden. Das *Behandeln einer Ausnahme*, also die explizite Reaktion auf das Eintreten einer Ausnahme, wird als *catching* bezeichnet. Schließlich werden wir auch die Begriffe *Ausnahme* und *Exception* synonym verwenden.

Das Grundprinzip des Exception-Mechanismus in Java kann wie folgt beschrieben werden:

- Ein Laufzeitfehler oder eine vom Entwickler gewollte Bedingung löst eine Ausnahme aus.
- Diese kann nun entweder von dem Programmteil, in dem sie ausgelöst wurde, behandelt werden, oder sie kann weitergegeben werden.
- Wird die Ausnahme weitergegeben, so hat der Empfänger der Ausnahme erneut die Möglichkeit, sie entweder zu behandeln oder selbst weiterzugeben.
- Wird die Ausnahme von keinem Programmteil behandelt, so führt sie zum Abbruch des Programms und zur Ausgabe einer Fehlermeldung.

Die folgenden Abschnitte erläutern die Details des Auftretens, der Behandlung und der Weitergabe von Ausnahmen.

9.2 Behandlung von Exceptions

- [9.2 Behandlung von Exceptions](#)
 - [9.2.1 Die try-catch-Anweisung](#)
 - [9.2.2 Das Fehlerobjekt](#)
 - [9.2.3 Die Fehlerklassen von Java](#)
 - [9.2.4 Fortfahren nach Fehlern](#)
 - [9.2.5 Mehr als eine catch-Klausel](#)
 - [9.2.6 Die finally-Klausel](#)

9.2.1 Die try-catch-Anweisung

Das Behandeln von Ausnahmen erfolgt mit Hilfe der [try-catch](#)-Anweisung:

```
001 try {
002     Anweisung;
003     ...
004 } catch (Ausnahmetyp x) {
005     Anweisung;
006     ...
007 }
```

Listing 9.1: Die try-catch-Anweisung

Der [try](#)-Block enthält dabei eine oder mehrere Anweisungen, bei deren Ausführung ein Fehler des Typs [Ausnahmetyp](#) auftreten kann. In diesem Fall wird die normale Programmausführung unterbrochen, und der Programmablauf fährt mit der ersten Anweisung nach der [catch](#)-Klausel fort, die den passenden Ausnahmetyp deklariert hat. Hier kann nun Code untergebracht werden, der eine angemessene Reaktion auf den Fehler realisiert.

Wir wollen das folgende fehlerhafte Programm betrachten:

[RTErrrorProg1.java](#) Beispiel

```
001 /* RTErrrorProg1.java */
002
003 public class RTErrrorProg1
004 {
005     public static void main(String[] args)
006     {
007         int i, base = 0;
008
009         for (base = 10; base >= 2; --base) {
010             i = Integer.parseInt("40",base);
011             System.out.println("40 base "+base+" = "+i);
012         }
013     }
014 }
```

Listing 9.2: Ein Programm mit einem Laufzeitfehler

Hier soll der String "40" aus verschiedenen Zahlensystemen in ein [int](#) konvertiert und als Dezimalzahl ausgegeben werden. Die dazu verwendete Methode [parseInt](#) überprüft, ob der übergebene String einen gültigen Zahlenwert zur angegebenen Basis darstellt. Ist dies nicht der Fall, löst sie eine Ausnahme des Typs [NumberFormatException](#) aus. Ohne weitere Maßnahmen stürzt das Programm dann beim Versuch, den String "40" als Zahl zur Basis 4 anzusehen, ab:

```
40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
```

```
Exception in thread "main" java.lang.NumberFormatException: 40
    at java.lang.Integer.parseInt(Compiled Code)
    at RTErrrorProgl.main(Compiled Code)
```

Das Programm lässt sich nun leicht gegen solche Fehler absichern, indem die Programmsequenz, die den Fehler verursacht, in eine [try-catch](#)-Anweisung eingeschlossen wird:

[Listing0903.java](#)

```
001 /* Listing0903.java */
002
003 public class Listing0903
004 {
005     public static void main(String[] args)
006     {
007         int i, base = 0;
008
009         try {
010             for (base = 10; base >= 2; --base) {
011                 i = Integer.parseInt("40",base);
012                 System.out.println("40 base "+base+" = "+i);
013             }
014         } catch (NumberFormatException e) {
015             System.out.println(
016                 "40 ist keine Zahl zur Basis "+base
017             );
018         }
019     }
020 }
```

Listing 9.3: Abfangen des Laufzeitfehlers mit einer try-catch-Anweisung

Zwar ist 40 immer noch keine Zahl zur Basis 4, aber das Programm fängt diesen Fehler nun ab und gibt eine geeignete Fehlermeldung aus:

```
40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
40 ist keine Zahl zur Basis 4
```

9.2.2 Das Fehlerobjekt

In der [catch](#)-Klausel wird nicht nur die Art des abzufangenden Fehlers definiert, sondern auch ein formaler Parameter angegeben, der beim Auftreten der Ausnahme ein Fehlerobjekt übernehmen soll. Fehlerobjekte sind dabei Instanzen der Klasse [Throwable](#) oder einer ihrer Unterklassen. Sie werden vom Aufrufer der Ausnahme erzeugt und als Parameter an die [catch](#)-Klausel übergeben. Das Fehlerobjekt enthält Informationen über die Art des aufgetretenen Fehlers. So liefert beispielsweise die Methode [getMessage](#) einen Fehlertext (wenn dieser explizit gesetzt wurde), und [printStackTrace](#) druckt einen Auszug aus dem Laufzeit-Stack. Am einfachsten kann der Fehlertext mit der Methode [toString](#) der Klasse [Throwable](#) ausgegeben werden:

[java.lang.Throwable](#)

```
public String getMessage()

public void printStackTrace()

public String toString()
```

Unser Beispielprogramm könnte dann wie folgt umgeschrieben werden:

Beispiel

[RTErrrorProg2.java](#)

```
001 /* RTErrrorProg2.java */
002
003 public class RTErrrorProg2
004 {
005     public static void main(String[] args)
006     {
007         int i, base = 0;
008
009         try {
010             for (base = 10; base >= 2; --base) {
011                 i = Integer.parseInt("40",base);
012                 System.out.println("40 base "+base+" = "+i);
013             }
014         } catch (NumberFormatException e) {
015             System.out.println("***Fehler aufgetreten***");
016             System.out.println("Ursache: "+e.getMessage());
017             e.printStackTrace();
018         }
019     }
020 }
```

Listing 9.4: Verwendung des Fehlerobjekts nach einem Laufzeitfehler

Die Ausgabe des Programms wäre dann:

```
40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
***Fehler aufgetreten***
Ursache: 40
java.lang.NumberFormatException: 40
    at java.lang.Integer.parseInt(Compiled Code)
    at RTErrrorProg2.main(Compiled Code)
```

Wie man sieht, ähnelt die Ausgabe des Programms der ersten Version, die ohne expliziten Fehler-Handler geschrieben wurde. Das liegt daran, daß das Java-Laufzeitsystem beim Auftreten eines Fehlers, der von keiner Methode behandelt wurde, [printStackTrace](#) aufruft, bevor es das Programm beendet.

9.2.3 Die Fehlerklassen von Java

Alle Laufzeitfehler in Java sind Unterklassen der Klasse [Throwable](#). [Throwable](#) ist eine allgemeine Fehlerklasse, die im wesentlichen eine Klartext-Fehlermeldung speichern und einen Auszug des Laufzeit-Stacks ausgeben kann. Unterhalb von [Throwable](#) befinden sich zwei große Vererbungshierarchien:

- Die Klasse [Error](#) ist Superklasse aller *schwerwiegenden* Fehler. Diese werden hauptsächlich durch Probleme in der virtuellen Java-Maschine ausgelöst. Fehler der Klasse [Error](#) sollten in der Regel nicht abgefangen werden, sondern (durch den Standard-Fehlerhandler) nach einer entsprechenden Meldung zum Abbruch des Programms führen.
- Alle Fehler, die möglicherweise für die Anwendung selbst von Interesse sind, befinden sich in der Klasse [Exception](#) oder einer ihrer Unterklassen. Ein Fehler dieser Art signalisiert einen abnormen Zustand, der vom Programm abgefangen und behandelt werden kann.

Viele Pakete der Java-Klassenbibliothek definieren ihre eigenen Fehlerklassen. So gibt es spezielle Fehlerklassen für die Dateiein- und -ausgabe, die Netzwirkommunikation oder den Zugriff auf Arrays. Wir werden diese speziellen Fehlerklassen immer dann erläutern, wenn eine Methode besprochen wird, die diese Art von Fehler erzeugt.

9.2.4 Fortfahren nach Fehlern

Die Reaktion auf eine Ausnahme muß keinesfalls zwangsläufig darin bestehen, das Programm zu beenden. Statt dessen kann auch versucht werden, den Fehler zu beheben oder zu umgehen, um dann mit dem Programm fortzufahren. Wird im obigen Programm die [try-catch](#)-Anweisung in die Schleife gesetzt, so fährt das Programm nach jedem Fehler fort und versucht, die Konvertierung zur nächsten Basis vorzunehmen:

```

001 /* Listing0905.java */
002
003 public class Listing0905
004 {
005     public static void main(String[] args)
006     {
007         int i, base = 0;
008
009         for (base = 10; base >= 2; --base) {
010             try {
011                 i = Integer.parseInt("40",base);
012                 System.out.println("40 base "+base+" = "+i);
013             } catch (NumberFormatException e) {
014                 System.out.println(
015                     "40 ist keine Zahl zur Basis "+base
016                 );
017             }
018         }
019     }
020 }

```

Listing 9.5: Fortfahren nach Laufzeitfehlern

Die Ausgabe des Programms lautet nun:

```

40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
40 ist keine Zahl zur Basis 4
40 ist keine Zahl zur Basis 3
40 ist keine Zahl zur Basis 2

```

Ob es sinnvoller ist, nach einem Fehler mit der Programmausführung fortzufahren, oder das Programm abubrechen, ist von der Art des Fehlers und der Stelle im Programm, an der er aufgetreten ist, abhängig. Hier muß von Fall zu Fall entschieden werden, wie vorgegangen werden soll.

Hinweis

9.2.5 Mehr als eine catch-Klausel

Bisher sind wir davon ausgegangen, daß innerhalb eines `try`-Blocks nur eine Ausnahme auftreten kann. Tatsächlich ist es natürlich ohne weiteres möglich, daß zwei oder mehrere unterschiedliche Ausnahmen ausgelöst werden. Das Programm kann auf verschiedene Fehler reagieren, indem es mehr als eine `catch`-Klausel verwendet. Jede `catch`-Klausel fängt die Fehler ab, die zum Typ des angegebenen Fehlerobjekts zuweisungskompatibel sind. Dazu gehören alle Fehler der angegebenen Klasse und all ihrer Unterklassen (das wichtige Konzept der Zuweisungskompatibilität von Objekttypen wurde in [Kapitel 7](#) erläutert). Die einzelnen `catch`-Klauseln werden in der Reihenfolge ihres Auftretens abgearbeitet.

Wir wollen das Beispielprogramm nun so erweitern, daß nicht nur ein einzelner String in eine Zahl konvertiert wird, sondern ein Array von Strings. Aufgrund eines Fehlers in der verwendeten `for`-Schleife verursacht das Programm einen Zugriff auf ein nicht vorhandenes Array-Element und löst damit eine Ausnahme des Typs `IndexOutOfBoundsException` aus. Diese kann zusammen mit der Ausnahme `NumberFormatException` in einer gemeinsamen `try-catch`-Anweisung behandelt werden:

Beispiel


```

001 /* Listing0906.java */
002
003 public class Listing0906
004 {
005     public static void main(String[] args)
006     {
007         int i, j, base = 0;
008         String numbers[] = new String[3];
009
010         numbers[0] = "10";
011         numbers[1] = "20";
012         numbers[2] = "30";
013         try {
014             for (base = 10; base >= 2; --base) {
015                 for (j = 0; j <= 3; ++j) {
016                     i = Integer.parseInt(numbers[j],base);
017                     System.out.println(
018                         numbers[j]+" base "+base+" = "+i
019                     );
020                 }
021             }
022         } catch (IndexOutOfBoundsException e1) {
023             System.out.println(
024                 "***IndexOutOfBoundsException: " + e1.toString()
025             );
026         } catch (NumberFormatException e2) {
027             System.out.println(
028                 "***NumberFormatException: " + e2.toString()
029             );
030         }
031     }
032 }

```

Listing 9.6: Mehr als eine catch-Klausel

Die Ausgabe des Programms ist nun:

```

10 base 10 = 10
20 base 10 = 20
30 base 10 = 30
***IndexOutOfBoundsException: java.lang.ArrayIndexOutOfBoundsException

```

9.2.6 Die finally-Klausel

Die [try-catch](#)-Anweisung enthält einen optionalen Bestandteil, der bisher noch nicht erläutert wurde. Mit Hilfe der [finally](#)-Klausel, die als letzter Bestandteil einer [try-catch](#)-Anweisung verwendet werden darf, kann ein Programmfragment definiert werden, das immer dann ausgeführt wird, wenn die zugehörige [try](#)-Klausel betreten wurde. Dabei spielt es keine Rolle, *welches* Ereignis dafür verantwortlich war, daß die [try](#)-Klausel verlassen wurde. Die [finally](#)-Klausel wird insbesondere dann ausgeführt, wenn der [try](#)-Block durch eine der folgenden Anweisungen verlassen wurde:

- wenn das normale Ende des [try](#)-Blocks erreicht wurde
- wenn eine Ausnahme aufgetreten ist, die durch eine [catch](#)-Klausel behandelt wurde
- wenn eine Ausnahme aufgetreten ist, die nicht durch eine [catch](#)-Klausel behandelt wurde
- wenn der [try](#)-Block durch eine der Sprunganweisungen [break](#), [continue](#) oder [return](#) verlassen werden soll

Die [finally](#)-Klausel ist also der ideale Ort, um Aufräumarbeiten durchzuführen. Hier können beispielsweise Dateien geschlossen oder Ressourcen freigegeben werden.

Hinweis

Die folgende Variation unseres Beispielprogramms benutzt finally dazu, am Ende des Programms eine Meldung auszugeben: Beispiel

Listing0907.java

```
001 /* Listing0907.java */
002
003 public class Listing0907
004 {
005     public static void main(String[] args)
006     {
007         int i, base = 0;
008
009         try {
010             for (base = 10; base >= 2; --base) {
011                 i = Integer.parseInt("40",base);
012                 System.out.println("40 base "+base+" = "+i);
013             }
014         } catch (NumberFormatException e) {
015             System.out.println(
016                 "40 ist keine Zahl zur Basis "+base
017             );
018         } finally {
019             System.out.println(
020                 "Sie haben ein einfaches Beispiel " +
021                 "sehr glücklich gemacht."
022             );
023         }
024     }
025 }
```

Listing 9.7: Verwendung der finally-Klausel

Die Ausgabe des Programms ist:

```
40 base 10 = 40
40 base 9 = 36
40 base 8 = 32
40 base 7 = 28
40 base 6 = 24
40 base 5 = 20
40 ist keine Zahl zur Basis 4
Sie haben ein einfaches Beispiel sehr glücklich gemacht.
```

9.3 Weitergabe von Exceptions

- [9.3 Weitergabe von Exceptions](#)
 - [9.3.1 Die catch-or-throw-Regel](#)
 - [9.3.2 Weitergabe einer Exception](#)
 - [Die Klasse RuntimeException](#)
 - [9.3.3 Auslösen von Ausnahmen](#)

9.3.1 Die catch-or-throw-Regel

Bei der Behandlung von Ausnahmen in Java gibt es die Grundregel *catch or throw*. Sie besagt, daß jede Ausnahme entweder *behandelt* oder *weitergegeben* werden muß. Wie man Ausnahmen behandelt, wurde anhand der [try-catch](#)-Anweisung in den vorherigen Abschnitten erklärt. Soll eine Ausnahme nicht behandelt, sondern weitergegeben werden, so kann dies einfach dadurch geschehen, daß eine geeignete [try-catch](#)-Anweisung nicht verwendet wird.

In der Regel gibt es dann jedoch zunächst einen Fehler beim Übersetzen des Programms, denn der Compiler erwartet, daß jede mögliche Ausnahme, die nicht behandelt, sondern weitergegeben wird, mit Hilfe der [throws](#)-Klausel zu deklarieren ist. Dazu wird an das Ende des Methodenkopfes das Schlüsselwort [throws](#) mit einer Liste aller Ausnahmen, die nicht behandelt werden sollen, angehängt:

```
001 public void SqrtTable()
002 throws ArithmeticException
003 {
004     double x = -1.0;
005
006     while (x <= 10.0) {
007         System.out.println("sqrt( "+x+" )="+Math.sqrt(x));
008         x += 1.0;
009     }
010 }
```

Listing 9.8: Verwendung der throws-Klausel

9.3.2 Weitergabe einer Exception

Tritt eine Ausnahme ein, sucht das Laufzeitsystem zunächst nach einer die Anweisung unmittelbar umgebenden [try-catch](#)-Anweisung, die diesen Fehler behandelt. Findet es eine solche nicht, wiederholt es die Suche sukzessive in allen umgebenden Blöcken. Ist auch das erfolglos, wird der Fehler an den Aufrufer der Methode weitergegeben. Dort beginnt die Suche nach einem Fehler-Handler von neuem, und der Fehler wird an die umgebenden Blöcke und schließlich an den Aufrufer weitergereicht. Enthält auch die Hauptmethode keinen Code, um den Fehler zu behandeln, bricht das Programm mit einer Fehlermeldung ab.

Da alle Fehler, die nicht innerhalb einer Methode behandelt werden, dem Compiler mit Hilfe der [throws](#)-Klausel bekannt gemacht werden, kennt dieser zu jeder Methode die potentiellen Fehler, die von ihr verursacht werden können. Mit diesen Informationen kann der Compiler bei jedem Methodenaufwurf sicherstellen, daß der Aufrufer seinerseits die *catch-or-throw*-Regel einhält.

Hinweis

Die Klasse RuntimeException

Um den Aufwand durch explizites Fehler-Handling bei der Entwicklung von Java-Programmen nicht zu groß werden zu lassen, gibt es eine Ausnahme von der *catch-or-throw*-Regel. Direkt unterhalb der Klasse [Exception](#) gibt es die Klasse [RuntimeException](#). Sie ist die Vaterklasse aller Laufzeitfehler, die zwar behandelt werden *können*, aber nicht *müssen*. Der Programmierer kann in diesem Fall selbst entscheiden, ob er den entsprechenden Fehler behandeln will oder nicht.

9.3.3 Auslösen von Ausnahmen

Wie schon erwähnt, sind Ausnahmeobjekte unter Java Instanzen bestimmter Klassen und können somit behandelt werden wie andere Objekte auch. Es ist also insbesondere möglich, ein Fehlerobjekt zu instanzieren oder eigene Fehlerklassen aus den vorhandenen abzuleiten.

Das Erzeugen eines Objekts aus einer Fehlerklasse gleicht dem Erzeugen eines Objekts aus einer beliebigen anderen Klasse. So legt die Anweisung

new `ArithmeticException` ein neues Fehlerobjekt der Klasse `ArithmeticException` an, das einer Variablen des entsprechenden Typs zugewiesen werden kann.

Mit Hilfe der `throw`-Anweisung kann ein solches Objekt dazu verwendet werden, eine Ausnahme zu erzeugen. Die Syntax der `throw`-Anweisung ist:

```
throw AusnahmeObjekt;
```

Die Behandlung dieser Fehler folgt den üblichen Regeln. Sie entspricht damit genau dem Fall, als wenn anstelle der `throw`-Anweisung eine aufgerufene Methode denselben Fehler ausgelöst hätte: Zunächst wird in den umgebenden Blöcken nach einem Fehler-Handler gesucht. Falls das erfolglos ist, wird der Fehler an den Aufrufer weitergegeben.

Die `throw`-Anweisung kann nicht nur dazu verwendet werden, *neue* Fehler auszulösen. Sie kann ebenfalls eingesetzt werden, um innerhalb der `catch`-Klausel einer `try-catch`-Anweisung das übergebene Fehlerobjekt erneut zu senden. In diesem Fall wird nicht noch einmal dieselbe `catch`-Klausel ausgeführt, sondern der Fehler wird gemäß den oben genannten Regeln an den umgebenden Block bzw. den Aufrufer weitergegeben.

Tip

Auch selbstdefinierte Ausnahmen müssen sich an die *catch-or-throw*-Regel halten. Wird die Ausnahme nicht innerhalb derselben Methode behandelt, ist sie mit Hilfe der `throws`-Klausel im Methodenkopf zu deklarieren und weiter oben in der Aufrufkette zu behandeln.

Das folgende Beispiel definiert eine Funktion `isPrim`, die ermittelt, ob der übergebene Parameter eine Primzahl ist. Wird ein negativer Wert übergeben, verursacht die Methode eine Ausnahme des Typs `ArithmeticException`:

Beispiel

```
001 public boolean isPrim(int n)
002 throws ArithmeticException
003 {
004     if (n <= 0) {
005         throw new ArithmeticException("isPrim: Parameter <= 0");
006     }
007     if (n == 1) {
008         return false;
009     }
010     for (int i = 2; i <= n/2; ++i) {
011         if (n % i == 0) {
012             return false;
013         }
014     }
015     return true;
016 }
```

Listing 9.9: Auslösen einer Ausnahme

Die `throw`-Anweisung hat dabei den Charakter einer Sprunganweisung. Sie unterbricht das Programm an der aktuellen Stelle und verzweigt unmittelbar zu der umgebenden `catch`-Klausel. Gibt es eine solche nicht, wird der Fehler an den Aufrufer weitergegeben. Tritt der Fehler innerhalb einer `try-catch`-Anweisung mit einer `finally`-Klausel auf, wird diese noch vor der Weitergabe des Fehlers ausgeführt.

9.4 Zusammenfassung

- [9.4 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Der Exception-Mechanismus in Java
- Die [try-catch](#)-Anweisung
- Fehlerobjekte und die Klasse [Throwable](#)
- Die Fehlerklassen [Error](#) und [Exception](#)
- Die [finally](#)-Klausel
- Die catch-or-throw-Regel, das Schlüsselwort [throws](#) und die Klasse [RuntimeException](#)
- Eigene Fehlerklassen und die [throw](#)-Anweisung

Kapitel 10

Multithreading

- [10 Multithreading](#)
 - [10.1 Grundlagen und Begriffe](#)
 - [10.2 Die Klasse Thread](#)
 - [10.2.1 Erzeugen eines neuen Threads](#)
 - [10.2.2 Abbrechen eines Threads](#)
 - [10.2.3 Anhalten eines Threads](#)
 - [10.2.4 Weitere Methoden](#)
 - [sleep](#)
 - [isAlive](#)
 - [join](#)
 - [10.3 Das Interface Runnable](#)
 - [10.3.1 Implementieren von Runnable](#)
 - [10.3.2 Multithreading durch Wrapper-Klassen](#)
 - [10.4 Synchronisation](#)
 - [10.4.1 Synchronisationsprobleme](#)
 - [10.4.2 Monitore](#)
 - [Anwendung von synchronized auf einen Block von Anweisungen](#)
 - [Anwendung von synchronized auf eine Methode](#)
 - [10.4.3 wait und notify](#)
 - [10.5 Verwalten von Threads](#)
 - [10.5.1 Priorität und Name](#)
 - [10.5.2 Thread-Gruppen](#)
 - [10.6 Zusammenfassung](#)

10.1 Grundlagen und Begriffe

- 10.1 Grundlagen und Begriffe

Kaum eine wichtige Programmiersprache der letzten Jahre hat das Konzept der *Nebenläufigkeit* innerhalb der Sprache implementiert. Mit Nebenläufigkeit bezeichnet man die Fähigkeit eines Systems, zwei oder mehr Vorgänge gleichzeitig oder quasi-gleichzeitig ausführen zu können. Lediglich ADA stellt sowohl parallele Prozesse als auch Mechanismen zur Kommunikation und Synchronisation zur Verfügung, die direkt in die Sprache eingebettet sind. Durch Weiterentwicklungen im Bereich der Betriebssystemtechnologie wurde allerdings das Konzept der *Threads* immer populärer und auf der Basis von Library-Routinen auch konventionellen Programmiersprachen zur Verfügung gestellt.

Java hat Threads direkt in die Sprache integriert und mit den erforderlichen Hilfsmitteln als Konstrukt zur Realisierung der Nebenläufigkeit implementiert. Ein Thread ist ein eigenständiges Programmfragment, das parallel zu anderen Threads laufen kann. Ein Thread ähnelt damit einem *Prozeß*, arbeitet aber auf einer feineren Ebene. Während ein Prozeß das Instrument zur Ausführung eines kompletten Programms ist, können innerhalb dieses Prozesses mehrere Threads parallel laufen. Der Laufzeit-Overhead zur Erzeugung und Verwaltung eines Threads ist relativ gering und kann in den meisten Programmen vernachlässigt werden. Ein wichtiger Unterschied zwischen Threads und Prozessen ist der, daß alle Threads eines Programmes sich einen gemeinsamen Adreßraum teilen, also auf dieselben Variablen zugreifen, während die Adreßräume unterschiedlicher Prozesse streng voneinander getrennt sind.

Die Implementierung von Threads war eine explizite Anforderung an das Design der Sprache. Threads sollen unter anderem die Implementierung grafischer Anwendungen erleichtern, die durch Simulationen komplexer Abläufe oft inhärent nebenläufig sind. Threads können auch dazu verwendet werden, die Bedienbarkeit von Dialoganwendungen zu verbessern, indem rechenintensive Anwendungen im Hintergrund ablaufen.

Threads werden in Java durch die Klasse [Thread](#) und das Interface [Runnable](#) implementiert. In beiden Fällen wird der Thread-Body, also der parallel auszuführende Code, in Form der überlagerten Methode [run](#) zur Verfügung gestellt. Die Kommunikation kann dann durch Zugriff auf die Instanz- oder Klassenvariablen oder durch Aufruf beliebiger Methoden, die innerhalb von [run](#) sichtbar sind, erfolgen. Zur Synchronisation stellt Java das aus der Betriebssystemtheorie bekannte Konzept des *Monitors* zur Verfügung, das es erlaubt, kritische Abschnitte innerhalb korrekt geklammerter Programmfragmente und Methoden zu kapseln und so den Zugriff auf gemeinsam benutzte Datenstrukturen zu koordinieren.

Darüber hinaus bietet Java Funktionen zur Verwaltung von Threads. Diese erlauben es, Threads in Gruppen zusammenzufassen, zu priorisieren und Informationen über Eigenschaften von Threads zu gewinnen. Das Scheduling kann dabei wahlweise *unterbrechend* oder *nichtunterbrechend* implementiert sein. Die Sprachspezifikation legt dies nicht endgültig fest, aber in den meisten Java-Implementierungen wird dies von den Möglichkeiten des darunter liegenden Betriebssystems abhängen.

10.2 Die Klasse Thread

- [10.2 Die Klasse Thread](#)
 - [10.2.1 Erzeugen eines neuen Threads](#)
 - [10.2.2 Abbrechen eines Threads](#)
 - [10.2.3 Anhalten eines Threads](#)
 - [10.2.4 Weitere Methoden](#)
 - [sleep](#)
 - [isAlive](#)
 - [join](#)

10.2.1 Erzeugen eines neuen Threads

Die Klasse [Thread](#) ist Bestandteil des Pakets [java.lang](#) und steht damit allen Anwendungen standardmäßig zur Verfügung. [Thread](#) stellt die Basismethoden zur Erzeugung, Kontrolle und zum Beenden von Threads zur Verfügung. Um einen konkreten Thread zu erzeugen, muß eine eigene Klasse aus [Thread](#) abgeleitet und die Methode [run](#) überlagert werden.

Mit Hilfe eines Aufrufs der Methode [start](#) wird der Thread gestartet und die weitere Ausführung an die Methode [run](#) übertragen. [start](#) wird nach dem Starten des Threads beendet, und der Aufrufer kann parallel zum neu erzeugten Thread fortfahren.

Das folgende Beispiel zeigt einen einfachen Thread, der in einer Endlosschleife einen Zahlenwert hochzählt:

Beispiel

[Listing1001.java](#)

```

001 /* Listing1001.java */
002
003 class MyThread1001
004 extends Thread
005 {
006     public void run()
007     {
008         int i = 0;
009         while (true) {
010             System.out.println(i++);
011         }
012     }
013 }
014
015 public class Listing1001
016 {
017     public static void main(String[] args)
018     {
019         MyThread1001 t = new MyThread1001();
020         t.start();
021     }
022 }

```

Listing 10.1: Ein einfacher Thread mit einem Zähler

Zunächst wird hier ein neues Objekt vom Typ [MyThread](#) instanziiert. Die Ausführung eines Threads ist damit vorbereitet, aber noch nicht tatsächlich erfolgt. Erst durch den Aufruf von [start](#) wird ein neuer Thread erzeugt und durch einen impliziten Aufruf von [run](#) der Thread-Body gestartet. Da das Programm in einer Endlosschleife läuft, läßt es sich nur gewaltsam abbrechen (beispielsweise durch Drücken von [\[STRG\]+\[C\]](#)).

Im Gegensatz zu unseren bisherigen Beispielen wird dieses Programm nicht automatisch beendet, nachdem `main` beendet wurde. Es gibt eine einfache Regel, die besagt, daß eine Java-Applikation immer dann beendet wird, wenn der letzte Thread beendet wurde, der kein *Hintergrund-Thread* (*Dämon*) ist. Da ein einfaches Programm nur einen einzigen Vordergrund-Thread besitzt (nämlich den, in dem `main` läuft), wird es demnach beendet, wenn `main` beendet wird. Das Beispielprogramm erzeugt dagegen einen zusätzlichen Vordergrund-Thread und kann damit vom Interpreter erst dann beendet werden, wenn dieser Thread beendet wurde.

Hinweis

10.2.2 Abbrechen eines Threads

Zunächst einmal wird ein Thread dadurch beendet, daß das Ende seiner `run`-Methode erreicht ist. In manchen Fällen ist es jedoch erforderlich, den Thread von außen abzubreaken. Die bis zum JDK 1.1 übliche Vorgehensweise bestand darin, die Methode `stop` der Klasse `Thread` aufzurufen. Dadurch wurde der Thread abgebrochen und aus der Liste der aktiven Threads entfernt.

Wir wollen das vorige Beispiel erweitern und den Thread nach zwei Sekunden durch Aufruf von `stop` beenden:

Beispiel

[Listing1002.java](#)

```
001 /* Listing1002.java */
002
003 class MyThread1002
004 extends Thread
005 {
006     public void run()
007     {
008         int i = 0;
009         while (true) {
010             System.out.println(i++);
011         }
012     }
013 }
014
015 public class Listing1002
016 {
017     public static void main(String[] args)
018     {
019         MyThread1002 t = new MyThread1002();
020         t.start();
021         try {
022             Thread.sleep(2000);
023         } catch (InterruptedException e) {
024             //nichts
025         }
026         t.stop();
027     }
028 }
```

Listing 10.2: Beenden des Threads durch Aufruf von `stop`

An diesem Beispiel kann man gut erkennen, daß der Thread tatsächlich parallel zum Hauptprogramm ausgeführt wird. Nach dem Aufruf von `start` beginnt einerseits die Zählschleife mit der Bildschirmausgabe, aber gleichzeitig fährt das Hauptprogramm mit dem Aufruf der `sleep`-Methode und dem Aufruf von `stop` fort. Beide Programmteile laufen also parallel ab.

Im JDK 1.2 wurde die Methode `stop` als `deprecated` markiert, d.h., sie sollte nicht mehr verwendet werden. Der Grund dafür liegt in der potentiellen *Unsicherheit* des Aufrufs, denn es ist nicht voraussagbar und auch nicht definiert, an welcher Stelle ein Thread unterbrochen wird, wenn ein Aufruf von `stop` erfolgt. Es kann nämlich insbesondere vorkommen, daß der Abbruch innerhalb eines kritischen Abschnitts erfolgt (der mit dem `synchronized`-Schlüsselwort geschützt wurde) oder in einer anwendungsspezifischen Transaktion auftritt, die aus Konsistenzgründen nicht unterbrochen werden darf.

JDK1.1/1.2

Die alternative Methode, einen Thread abzubreaken, besteht darin, im Thread selbst auf Unterbrechungsanforderungen zu reagieren. So könnte beispielsweise eine Membervariable `cancelled` eingeführt und beim Initialisieren des Threads auf `false` gesetzt werden. Mit Hilfe einer Methode `cancel` kann der Wert der Variable zu einem beliebigen Zeitpunkt auf `true` gesetzt werden. Aufgabe der Bearbeitungsroutine in `run` ist es nun, an geeigneten Stellen diese Variable abzufragen und für den Fall, daß sie `true` ist, die Methode `run` konsistent zu beenden.

Dabei darf `cancelled` natürlich nicht zu oft abgefragt werden, um das Programm nicht unnötig aufzublähen und das Laufzeitverhalten des Threads nicht zu sehr zu verschlechtern. Andererseits darf die Abfrage nicht zu selten erfolgen, damit es nicht zu lange dauert, bis auf eine Abbruchanforderung reagiert wird. Insbesondere darf es keine potentiellen Endlosschleifen geben, in den `cancelled` überhaupt nicht abgefragt wird. Die Kunst besteht darin, diese gegensätzlichen Anforderungen sinnvoll zu vereinen.

Glücklicherweise gibt es in der Klasse `Thread` bereits einige Methoden, die einen solchen Mechanismus standardmäßig unterstützen:

[java.lang.Thread](#)

```
public void interrupt()

public boolean isInterrupted()

public static boolean interrupted()
```

Durch Aufruf von `interrupt` wird ein Flag gesetzt, das eine Unterbrechungsanforderung signalisiert. Durch Aufruf von `isInterrupted` kann der Thread feststellen, ob das Abbruchflag gesetzt wurde und der Thread beendet werden soll. Die statische Methode `interrupted` stellt den Status des Abbruchsflags beim *aktuellen* Thread fest. Ihr Aufruf entspricht dem Aufruf von `currentThread().isInterrupted()`, setzt aber zusätzlich das Abbruchflag auf seinen initialen Wert `false` zurück.

Wir wollen uns den Gebrauch dieser Methoden an einem Beispiel ansehen. Dazu soll ein Programm geschrieben werden, das in einem separaten Thread ununterbrochen Textzeilen auf dem Bildschirm ausgibt. Das Hauptprogramm soll den Thread erzeugen und nach 2 Sekunden durch einen Aufruf von `interrupt` eine Unterbrechungsanforderung erzeugen. Der Thread soll dann die aktuelle Zeile fertig ausgeben und anschließend terminieren.

[Listing1003.java](#)

```
001 /* Listing1003.java */
002
003 public class Listing1003
004 extends Thread
005 {
006     int cnt = 0;
007
008     public void run()
009     {
010         while (true) {
011             if (isInterrupted()) {
012                 break;
013             }
014             printLine(++cnt);
015         }
016     }
017
018     private void printLine(int cnt)
019     {
020         //Zeile ausgeben
021         System.out.print(cnt + ": ");
022         for (int i = 0; i < 30; ++i) {
023             System.out.print(i == cnt % 30 ? "* " : ". ");
024         }
025         System.out.println();
026         //100 ms. warten
027         try {
028             Thread.sleep(100);
029         } catch (InterruptedException e) {
030             interrupt();
031         }
032     }
033
034     public static void main(String args[])
035     {
036         Listing1003 th = new Listing1003();
037         {
038             //Thread starten
039             th.start();
040             //2 Sekunden warten
041             try {
042                 Thread.sleep(2000);
```

```

043     } catch (InterruptedException e) {
044     }
045     //Thread unterbrechen
046     th.interrupt();
047 }
048 }
049 }

```

Listing 10.3: Anwendung der Methoden interrupt und isInterrupted

Die `main`-Methode ist leicht zu verstehen. Sie startet den Thread, wartet 2 Sekunden und ruft dann die Methode `interrupt` auf. In der Methode `run` wird in einer Endlosschleife durch Aufruf von `println` jeweils eine neue Zeile ausgegeben. Zuvor wird bei jedem Aufruf mit `isInterrupted` geprüft, ob das Abbruchflag gesetzt wurde. Ist das der Fall, wird keine weitere Zeile ausgegeben, sondern die Schleife (und mit ihr der Thread) beendet.

Innerhalb von `println` wird zunächst die Textzeile ausgegeben und dann eine Pause von 100 Millisekunden eingelegt. Da in der Methode keine Abfrage des Abbruchflags erfolgt, ist sichergestellt, daß die aktuelle Zeile selbst dann bis zum Ende ausgegeben wird, wenn der Aufruf von `interrupt` mitten in der Schleife zur Ausgabe der Bildschirmzeile erfolgt.

Da die Pause nach der Bildschirmausgabe mit 100 Millisekunden vermutlich länger dauert als die Bildschirmausgabe selbst, ist es recht wahrscheinlich, daß der Aufruf von `interrupt` während des Aufrufs von `sleep` erfolgt. Ist das der Fall, wird `sleep` mit einer `InterruptedException` abgebrochen (auch wenn die geforderte Zeitspanne noch nicht vollständig verstrichen ist). Wichtig ist hier, daß das Abbruchflag zurückgesetzt wird und der Aufruf von `interrupt` somit eigentlich verlorengehen würde, wenn er nicht direkt in der `catch`-Klausel behandelt würde. Wir rufen daher innerhalb der `catch`-Klausel `interrupt` erneut auf, um das Flag wieder zu setzen und `run` die Abbruchanforderung zu signalisieren. Alternativ hätten wir auch die Ausnahme an den Aufrufer weitergeben können und sie dort als Auslöser für das Ende der Ausgabeschleife betrachten können.

Die beiden anderen Methoden, die eine Ausnahme des Typs `InterruptedException` auslösen können, sind `join` der Klasse `Thread` und `wait` der Klasse `Object`. Auch sie setzen beim Auftreten der Ausnahme das Abbruchflag zurück und müssen daher in ähnlicher Weise behandelt werden.

Hinweis

10.2.3 Anhalten eines Threads

Die Klasse `Thread` besitzt zwei Methoden `suspend` und `resume`, mit deren Hilfe es möglich ist, einen Thread vorübergehend anzuhalten und anschließend an der Unterbrechungsstelle fortzusetzen. Beide Methoden sind nicht ganz ungefährlich und können unbemerkt Deadlocks verursachen. Sie wurden daher im JDK 1.2 als `deprecated` markiert und sollten nicht mehr verwendet werden. Ihre Funktionalität muß - wenn erforderlich - manuell nachgebildet werden.

JDK1.1/1.2

10.2.4 Weitere Methoden

sleep

Sowohl innerhalb der Threads als auch innerhalb der Methode `main` wird ein Aufruf von `Thread.sleep` verwendet, um das Programm pausieren zu lassen. `sleep` ist eine statische Methode der Klasse `Thread`, die mit einem oder zwei Parametern aufgerufen werden kann:

`java.lang.Thread`

```
public static void sleep(long millis)
```

```
public static void sleep(long millis, int nanos)
```

Die erste Version sorgt dafür, daß der aktuelle Prozeß für die (in Millisekunden angegebene) Zeit unterbrochen wird. Die zweite erlaubt eine noch genauere Eingabe der Wartezeit, indem auch Bruchteile im Nanosekundenbereich angegeben werden können. In beiden Fällen wird die tatsächlich erzielbare Genauigkeit allerdings durch Hardwarerestriktionen der Zielmaschine begrenzt. Im Fall von MS-DOS/Windows entspricht sie in der Regel der Genauigkeit des System-Tickers, liegt also bei etwa 55 ms.

Die Kapselung des Aufrufs von `Thread.sleep` innerhalb eines `try-catch`-Blocks ist erforderlich, weil `sleep` nach Ablauf der Zeit eine Ausnahme vom Typ `InterruptedException` erzeugt. Ohne den `try-catch`-Block würde diese an den Aufrufer weitergegeben werden. Als Klassenmethode kann `sleep` aufgerufen werden, ohne daß eine Instanz der Klasse `Thread` verfügbar ist. Insbesondere kann die Methode auch dazu verwendet werden, das Hauptprogramm pausieren zu lassen, das ja nicht explizit als Thread erzeugt wurde. Dies funktioniert deshalb, weil beim Starten eines Java-Programms automatisch ein Thread für die Ausführung des Hauptprogramms angelegt wurde.

Hinweis

isAlive

Mit dieser Methode kann festgestellt werden, ob der aktuelle Thread noch läuft.

[java.lang.Thread](#)

```
public final boolean isAlive()
```

[isAlive](#) gibt immer dann [true](#) zurück, wenn der aktuelle Thread gestartet, aber noch nicht wieder beendet wurde. Beendet wird ein Thread, wenn das Ende der [run](#)-Methode erreicht ist oder wenn (in Prä-1.2-JDKs) die Methode [stop](#) aufgerufen wurde.

join

[java.lang.Thread](#)

```
public final void join()  
    throws InterruptedException
```

Die Methode [join](#) wartet auf das Ende des Threads, für den sie aufgerufen wurde. Sie ermöglicht es damit, einen Prozeß zu starten und (ähnlich einem Funktionsaufruf) mit der weiteren Ausführung so lange zu warten, bis der Prozeß beendet ist. [join](#) gibt es auch mit einem [long](#) als Parameter. In diesem Fall wartet die Methode maximal die angegebene Zeit in Millisekunden und fährt nach Ablauf der Zeit auch dann fort, wenn der Prozeß noch nicht beendet ist.

10.3 Das Interface Runnable

- 10.3 Das Interface Runnable
 - 10.3.1 Implementieren von Runnable
 - 10.3.2 Multithreading durch Wrapper-Klassen

Nicht immer ist es möglich, eine Klasse, die als Thread laufen soll, von [Thread](#) abzuleiten. Dies ist insbesondere dann nicht möglich, wenn die Klasse Bestandteil einer Vererbungshierarchie ist, die eigentlich nichts mit Multithreading zu tun hat. Da Java keine Mehrfachvererbung kennt, kann eine bereits abgeleitete Klasse nicht von einer weiteren Klasse erben. Da sehr unterschiedliche Klassen als Thread parallel zu vorhandenem Code ausgeführt werden können, ist dies eine sehr unschöne Einschränkung des Multithreading-Konzepts von Java.

Glücklicherweise gibt es einen Ausweg. Er besteht darin, einen Thread nicht durch Ableiten aus [Thread](#), sondern durch Implementierung des Interfaces [Runnable](#) zu erzeugen. [Runnable](#) enthält nur eine einzige Deklaration, nämlich die der Methode [run](#):

```

public abstract void run()
                                     java.lang.Runnable
```

10.3.1 Implementieren von Runnable

Tatsächlich muß jede Klasse, deren Instanzen als Thread laufen sollen, das Interface [Runnable](#) implementieren (sogar die Klasse [Thread](#) selbst). Um eine nicht von [Thread](#) abgeleitete Instanz in dieser Weise als Thread laufen zu lassen, ist in folgenden Schritten vorzugehen:

- Zunächst wird ein neues Thread-Objekt erzeugt.
- An den Konstruktor wird das Objekt übergeben, das parallel ausgeführt werden soll.
- Die Methode [start](#) des neuen Thread-Objekts wird aufgerufen.

Nun startet das [Thread](#)-Objekt die [run](#)-Methode des übergebenen Objekts, das sie ja durch die Übergabe im Konstruktor kennt. Da dieses Objekt das Interface [Runnable](#) implementiert, ist garantiert, daß eine geeignete Methode [run](#) zur Verfügung steht.

Wir wollen dies an einem Beispiel deutlich machen:

Beispiel

[Listing1004.java](#)

```

001 /* Listing1004.java */
002
003 class A1004
004 {
005     int irgendwas;
006     //...
007 }
008
009 class B1004
010 extends A1004
011 implements Runnable
012 {
013     public void run()
014     {
015         int i = 0;
016         while (true) {
017             if (Thread.interrupted()) {
018                 break;
019             }
020             System.out.println(i++);
021         }
022     }
023 }
024
025 public class Listing1004
026 {
027     public static void main(String args[])
028     {
029         B1004 b = new B1004();
```

```

030     Thread t = new Thread(b);
031     t.start();
032     try {
033         Thread.sleep(1000);
034     } catch (InterruptedException e){
035         //nichts
036     }
037     t.interrupt();
038 }
039 }

```

Listing 10.4: Implementieren von Runnable

Die Klasse B1004 ist von A1004 abgeleitet und kann daher nicht von [Thread](#) abgeleitet sein. Statt dessen implementiert sie das Interface [Runnable](#). Um nun ein Objekt der Klasse B1004 als Thread auszuführen, wird in [main](#) von Listing1004 eine Instanz dieser Klasse erzeugt und an den Konstruktor der Klasse [Thread](#) übergeben. Nach dem Aufruf von [start](#) wird die [run](#)-Methode von B1004 aufgerufen.

10.3.2 Multithreading durch Wrapper-Klassen

Auf eine ähnliche Weise lassen sich auch Methoden, die ursprünglich nicht als Thread vorgesehen waren, in einen solchen umwandeln und im Hintergrund ausführen. Der Grundstein für die Umwandlung eines gewöhnlichen Objekts in einen Thread wird dabei immer bei der Übergabe eines [Runnable](#)-Objekts an den Konstruktor des [Thread](#)-Objekts gelegt. Das folgende Beispiel demonstriert, wie eine zeitintensive Primfaktorzerlegung im Hintergrund laufen kann.

Zunächst benötigen wir dazu eine Klasse [PrimeNumberTools](#), die Routinen zur Berechnung von Primzahlen und zur Primfaktorzerlegung zur Verfügung stellt. Diese Klasse ist weder von [Thread](#) abgeleitet, noch implementiert sie [Runnable](#):

Beispiel

[PrimeNumberTools.java](#)

```

001 /* PrimeNumberTools.java */
002
003 public class PrimeNumberTools
004 {
005     public void printPrimeFactors(int num)
006     {
007         int whichprime = 1;
008         int prime;
009         String prefix;
010
011         prefix = "primeFactors("+num+")= ";
012         while (num > 1) {
013             prime = getPrime(whichprime);
014             if (num % prime == 0) {
015                 System.out.print(prefix+prime);
016                 prefix = " ";
017                 num /= prime;
018             } else {
019                 ++whichprime;
020             }
021         }
022         System.out.println();
023     }
024
025     public int getPrime(int cnt)
026     {
027         int i = 1;
028         int ret = 2;
029
030         while (i < cnt) {
031             ++ret;
032             if (isPrime(ret)) {
033                 ++i;
034             }
035         }
036         return ret;
037     }
038 }

```

```

039     private boolean isPrime(int num)
040     {
041         for (int i = 2; i < num; ++i) {
042             if (num % i == 0) {
043                 return false;
044             }
045         }
046         return true;
047     }
048
049 }

```

Listing 10.5: Eine Klasse zur Primfaktorzerlegung

Ohne Hintergrundverarbeitung könnte `PrimeNumberTools` instanziiert und ihre Methoden durch einfachen Aufruf verwendet werden:

[Listing1006.java](#)

```

001 /* Listing1006.java */
002
003 import java.io.*;
004
005 public class Listing1006
006 {
007     public static void main(String[] args)
008     {
009         PrimeNumberTools pt = new PrimeNumberTools();
010         BufferedReader in = new BufferedReader(
011             new InputStreamReader(
012                 new DataInputStream(System.in)));
013
014         int num;
015
016         try {
017             while (true) {
018                 System.out.print("Bitte eine Zahl eingeben: ");
019                 System.out.flush();
020                 num = (new Integer(in.readLine())).intValue();
021                 if (num == -1) {
022                     break;
023                 }
024                 pt.printPrimeFactors(num);
025             } catch (IOException e) {
026                 //nichts
027             }
028         }
029 }

```

Listing 10.6: Verwendung der Klasse zur Primfaktorzerlegung

Das Programm erzeugt eine Instanz der Klasse `PrimeNumberTools` und führt für jeden eingelesenen Zahlenwert durch Aufruf der Methode `printPrimeFactors` die Primfaktorzerlegung durch. Daß hier einige I/O-Routinen von Java verwendet wurden, braucht Sie nicht zu beunruhigen; wir kommen in [Kapitel 13](#) auf sie zurück.

Um nun diese Berechnungen asynchron durchzuführen, entwerfen wir eine Wrapper-Klasse, die von `PrimeNumberTools` abgeleitet wird und das Interface `Runnable` implementiert:

[ThreadedPrimeNumberTools.java](#)

```

001 /* ThreadedPrimeNumberTools.java */
002
003 public class ThreadedPrimeNumberTools
004     extends PrimeNumberTools
005     implements Runnable
006 {
007     private int arg;
008     private int func;
009
010     public void printPrimeFactors(int num)
011     {
012         execAsynchron(1, num);

```

```

013     }
014
015     public void printPrime(int cnt)
016     {
017         execAsynchron(2,cnt);
018     }
019
020     public void run()
021     {
022         if (func == 1) {
023             super.printPrimeFactors(arg);
024         } else if (func == 2) {
025             int result = super.getPrime(arg);
026             System.out.println("prime number #" + arg + " is: " + result);
027         }
028     }
029
030     private void execAsynchron(int func, int arg)
031     {
032         Thread t = new Thread(this);
033         this.func = func;
034         this.arg = arg;
035         t.start();
036     }
037 }

```

Listing 10.7: Primfaktorzerlegung mit Threads

Hier wurde die Methode `printPrimeFactors` überlagert, um den Aufruf der Superklasse asynchron ausführen zu können. Dazu wird in `execAsynchron` ein neuer Thread generiert, dem im Konstruktor das aktuelle Objekt übergeben wird. Durch Aufruf der Methode `start` wird der Thread gestartet und die `run`-Methode des aktuellen Objekts aufgerufen. Diese führt die gewünschten Aufrufe der Superklasse aus und schreibt die Ergebnisse auf den Bildschirm. So ist es möglich, bereits während der Berechnung der Primfaktoren einer Zahl eine neue Eingabe zu erledigen und eine neue Primfaktorberechnung zu beginnen.

Um dies zu erreichen, ist in der Klasse `Listing1006` lediglich die Deklaration des Objekts vom Typ `PrimeNumberTools` durch eine Deklaration vom Typ der daraus abgeleiteten Klasse `ThreadedPrimeNumberTools` zu ersetzen:

[Listing1008.java](#)

```

001 /* Listing1008.java */
002
003 import java.io.*;
004
005 public class Listing1008
006 {
007     public static void main(String[] args)
008     {
009         ThreadedPrimeNumberTools pt;
010         BufferedReader in = new BufferedReader(
011             new InputStreamReader(
012                 new DataInputStream(System.in)));
013         int num;
014
015         try {
016             while (true) {
017                 System.out.print("Bitte eine Zahl eingeben: ");
018                 System.out.flush();
019                 num = (new Integer(in.readLine())).intValue();
020                 if (num == -1) {
021                     break;
022                 }
023                 pt = new ThreadedPrimeNumberTools();
024                 pt.printPrimeFactors(num);
025             }
026         } catch (IOException e) {
027             //nichts
028         }
029     }

```


Listing 10.8: Verwendung der Klasse zur Primfaktorzerlegung mit Threads

Wenn alle Eingaben erfolgen, bevor das erste Ergebnis ausgegeben wird, könnte eine Beispielsitzung etwa so aussehen (Benutzereingaben sind fett gedruckt):

```
Bitte eine Zahl eingeben: 991
Bitte eine Zahl eingeben: 577
Bitte eine Zahl eingeben: 677
Bitte eine Zahl eingeben: -1
primeFactors(577)= 577
primeFactors(677)= 677
primeFactors(991)= 991
```

Obwohl das gewünschte Verhalten (nämlich die asynchrone Ausführung einer zeitaufwendigen Berechnung im Hintergrund) realisiert wird, ist dieses Beispiel nicht beliebig zu verallgemeinern. Die Ausgabe erfolgt beispielsweise nur dann ohne Unterbrechung durch Benutzereingaben, wenn alle Eingaben vor der ersten Ausgabe abgeschlossen sind. Selbst in diesem Fall funktioniert das Programm nicht immer zuverlässig. Es ist generell problematisch, Hintergrundprozessen zu erlauben, auf die Standardein- oder -ausgabe zuzugreifen, die ja vorwiegend vom Vordergrund-Thread verwendet wird. Ein- und Ausgaben könnten durcheinander geraten und es könnte zu Synchronisationsproblemen kommen, die die Ausgabe verfälschen. Wir haben nur ausnahmsweise davon Gebrauch gemacht, um das Prinzip der Hintergrundverarbeitung an einem einfachen Beispiel darzustellen.

Hinweis

Das nächste Problem ist die Realisierung des Dispatchers in `run`, der mit Hilfe der Instanzvariablen `func` und `arg` die erforderlichen Funktionsaufrufe durchführt. Dies funktioniert hier recht problemlos, weil alle Methoden dieselbe Parametrisierung haben. Im allgemeinen wäre hier ein aufwendigerer Übergabemechanismus erforderlich.

Des weiteren sind meistens Vorder- und Hintergrundverarbeitung zu *synchronisieren*, weil der Vordergrundprozeß die Ergebnisse des Hintergrundprozesses benötigt. Auch hier haben wir stark vereinfacht, indem die Ergebnisse einfach direkt nach der Verfügbarkeit vom Hintergrundprozeß auf den Bildschirm geschrieben wurden. Das Beispiel zeigt jedoch, wie *prinzipiell* vorgegangen werden könnte und ist vorwiegend als Anregung für eigene Experimente anzusehen.

10.4 Synchronisation

- [10.4 Synchronisation](#)
 - [10.4.1 Synchronisationsprobleme](#)
 - [10.4.2 Monitore](#)
 - [Anwendung von synchronized auf einen Block von Anweisungen](#)
 - [Anwendung von synchronized auf eine Methode](#)
 - [10.4.3 wait und notify](#)

10.4.1 Synchronisationsprobleme

Wenn man sich mit Nebenläufigkeit beschäftigt, muß man sich in aller Regel auch mit Fragen der *Synchronisation* nebenläufiger Prozesse beschäftigen. In Java erfolgt die Kommunikation zweier Threads auf der Basis gemeinsamer Variablen, die von beiden Threads erreicht werden können. Führen beide Prozesse Änderungen auf den gemeinsamen Daten durch, so müssen sie synchronisiert werden, denn andernfalls können undefinierte Ergebnisse entstehen.

Wir wollen uns als einleitendes Beispiel ein kleines Programm ansehen, bei dem zwei Threads Beispiel einen gemeinsamen Zähler hochzählen:

[Listing1009.java](#)

```

001 /* Listing1009.java */
002
003 public class Listing1009
004 extends Thread
005 {
006     static int cnt = 0;
007
008     public static void main(String[] args)
009     {
010         Thread t1 = new Listing1009();
011         Thread t2 = new Listing1009();
012         t1.start();
013         t2.start();
014     }
015
016     public void run()
017     {
018         while (true) {
019             System.out.println(cnt++);
020         }
021     }
022 }
    
```

Listing 10.9: Zwei Zählerthreads

Läßt man das Programm eine Weile laufen, könnte es beispielsweise zu folgender Ausgabe kommen:

```

0
1
2
3
4
5
6
7
8
9
10
11
12
    
```

```

13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
33     <-- Nanu? Wo ist die 32?
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
32     <-- Ach so, hier!
58
59

```

Beide Prozesse greifen unsynchronisiert auf die gemeinsame Klassenvariable `cnt` zu. Da die Operation `System.out.println(cnt++);` nicht *atomic* ist, kommt es zu dem Fall, daß die Operation mitten in der Ausführung unterbrochen wird und der Scheduler mit dem anderen Thread fortfährt. Erst später, wenn der unterbrochene Prozeß wieder Rechenzeit erhält, kann er seinen vor der Unterbrechung errechneten Zählerwert von 32 ausgeben. Sein Pendant war in der Zwischenzeit allerdings bereits bis 56 fortgefahren. Um diese Art von Inkonsistenzen zu beseitigen, bedarf es der *Synchronisation* der beteiligten Prozesse.

10.4.2 Monitore

Zur Synchronisation nebenläufiger Prozesse hat Java das Konzept des *Monitors* implementiert. Ein Monitor ist die Kapselung eines *kritischen Bereichs* (also eines Programnteils, der nur von jeweils einem Prozeß zur Zeit durchlaufen werden darf) mit Hilfe einer automatisch verwalteten Sperre. Diese Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen. Ist sie beim Eintritt in den Monitor bereits von einem anderen Prozeß gesetzt, so muß der aktuelle Prozeß warten, bis der Konkurrent die Sperre freigegeben und den Monitor verlassen hat.

Das Monitor-Konzept wird mit Hilfe des in die Sprache integrierten Schlüsselworts `synchronized` realisiert. Durch `synchronized` kann entweder eine komplette Methode oder ein Block innerhalb einer Methode geschützt werden. Der Eintritt in den so deklarierten Monitor wird durch das Setzen einer Sperre auf einer Objektvariablen erreicht. Bezieht sich `synchronized` auf eine komplette Methode, wird als Sperre der `this`-Pointer verwendet, andernfalls ist eine Objektvariable explizit anzugeben.

Anwendung von synchronized auf einen Block von Anweisungen

Wir wollen uns diese Art der Verwendung an einem Beispiel ansehen, welches das oben demonstrierte Synchronisationsproblem löst. Die naheliegende Lösung, die Anweisung `System.out.println(cnt++)`; durch einen `synchronized`-Block auf der Variablen `this` zu synchronisieren, funktioniert leider nicht. Da der Zeiger `this` für jeden der beiden Threads, die ja unterschiedliche Instanzen repräsentieren, neu vergeben wird, wäre für jeden Thread der Eintritt in den Monitor grundsätzlich erlaubt.

Statt dessen verwenden wir die aus [Abschnitt 31.2.2](#) bekannte Methode `getClass`, die uns ein Klassenobjekt beschafft (ein und dasselbe für alle Instanzen), mit dem wir die Klassenvariable `cnt` schützen können:

[Listing1010.java](#)

```
001 /* Listing1010.java */
002
003 public class Listing1010
004 extends Thread
005 {
006     static int cnt = 0;
007
008     public static void main(String[] args)
009     {
010         Thread t1 = new Listing1010();
011         Thread t2 = new Listing1010();
012         t1.start();
013         t2.start();
014     }
015
016     public void run()
017     {
018         while (true) {
019             synchronized (getClass()) {
020                 System.out.println(cnt++);
021             }
022         }
023     }
024 }
```

Listing 10.10: Synchronisation von Threads mit Klassenobjekten

Nun werden alle Zählerwerte in aufsteigender Reihenfolge ausgegeben.

Anwendung von synchronized auf eine Methode

Ein anderer Fall ist der, bei dem der Zugriff auf ein Objekt selbst synchronisiert werden muß, weil damit zu rechnen ist, daß mehr als ein Thread zur gleichen Zeit das Objekt verwenden will.

Im folgenden werden die potentiellen Probleme am Beispiel eines Zählerobjekts erläutert, dessen Aufgabe es ist, einen internen Zähler zu kapseln, auf Anforderung den aktuellen Zählerstand zu liefern und den internen Zähler zu inkrementieren. Hierbei handelt es sich um eine Aufgabe, die beispielsweise in der Datenbankprogrammierung sehr häufig vorkommt, um Schlüsselnummern zu generieren.

Typischerweise wird das Synchronisationsproblem dadurch verschärft, daß die Verwendung des Zählers einige vergleichsweise langsame Festplattenzugriffe erforderlich macht. In unserem Beispiel wird der Zähler von fünf Threads verwendet. Die Langsamkeit und damit die Wahrscheinlichkeit, daß der Scheduler die Zugriffsoperation unterbricht, wird in unserem Beispiel durch eine Sequenz eingestreuter Fließkommaoperationen erhöht:

Beispiel

Beispiel

```

001 /* Listing1011.java */
002
003 class Counter1011
004 {
005     int cnt;
006
007     public Counter1011(int cnt)
008     {
009         this.cnt = cnt;
010     }
011
012     public int nextNumber()
013     {
014         int ret = cnt;
015         //Hier erfolgen ein paar zeitaufwendige Berechnungen, um
016         //so zu tun, als sei das Errechnen des Nachfolgezählers
017         //eine langwierige Operation, die leicht durch den
018         //Scheduler unterbrochen werden kann.
019         double x = 1.0, y, z;
020         for (int i= 0; i < 1000; ++i) {
021             x = Math.sin((x*i%35)*1.13);
022             y = Math.log(x+10.0);
023             z = Math.sqrt(x+y);
024         }
025         //Jetzt ist der Wert gefunden
026         cnt++;
027         return ret;
028     }
029 }
030
031 public class Listing1011
032 extends Thread
033 {
034     private String name;
035     private Counter1011 counter;
036
037     public Listing1011(String name, Counter1011 counter)
038     {
039         this.name = name;
040         this.counter = counter;
041     }
042
043     public static void main(String[] args)
044     {
045         Thread t[] = new Thread[5];
046         Counter1011 cnt = new Counter1011(10);
047
048         for (int i = 0; i < 5; ++i) {
049             t[i] = new Listing1011("Thread-"+i,cnt);
050             t[i].start();
051         }
052     }
053
054     public void run()
055     {
056         while (true) {
057             System.out.println(counter.nextNumber()+" for "+name);
058         }
059     }
060 }

```

Listing 10.11: Eine unzureichend synchronisierte Zählerklasse

Das Ergebnis des Programms ist - wie nicht anders zu erwarten - schlecht, denn es werden sehr viele doppelte Schlüssel produziert. Ein Beispiellauf brachte bereits in den ersten 15 Aufrufen 6 doppelte Zählerwerte:

```

10 for Thread-2
11 for Thread-4
10 for Thread-0
10 for Thread-1
11 for Thread-2
11 for Thread-3
12 for Thread-4
13 for Thread-0
14 for Thread-1
15 for Thread-2
16 for Thread-3
17 for Thread-4
18 for Thread-0
19 for Thread-1
20 for Thread-2

```

Auch hier gibt es eine einfache Lösung für das Synchronisationsproblem. Eine einfache Markierung der Methode `nextNumber` als `synchronized` macht diese zu einem Monitor und sorgt dafür, daß der komplette Code innerhalb der Methode als atomares Programmfragment behandelt wird. Eine Unterbrechung des kritischen Abschnitts durch einen anderen Thread ist dann nicht mehr möglich:

```

001 public synchronized int nextNumber()
002 {
003     int ret = cnt;
004     //Hier erfolgen ein paar zeitaufwendige Berechnungen, um so
005     //zu tun, als sei das Errechnen des Nachfolgezählerstandes
006     //eine langwierige Operation, die leicht durch den
007     //Scheduler unterbrochen werden kann.
008     double x = 1.0, y, z;
009     for (int i= 0; i < 1000; ++i) {
010         x = Math.sin((x*i%35)*1.13);
011         y = Math.log(x+10.0);
012         z = Math.sqrt(x+y);
013     }
014     //Jetzt ist der Wert gefunden
015     cnt++;
016     return ret;
017 }

```

Listing 10.12: Synchronisieren der Zählermethode

Durch das `synchronized`-Attribut wird beim Aufruf der Methode die Instanzvariable `this` gesperrt und damit der Zugriff für andere Threads unmöglich gemacht. Erst nach Verlassen der Methode und Entsperren von `this` kann `nextNumber` wieder von anderen Threads aufgerufen werden.

Diese Art des Zugriffsschutzes wird in Java von vielen Klassen verwendet, um ihre Methoden *thread-sicher* zu machen. Nach Aussage der Sprachspezifikation kann davon ausgegangen werden, daß die gesamte Java-Klassenbibliothek in diesem Sinne thread-sicher ist.

Hinweis

10.4.3 wait und notify

Neben dem Monitorkonzept stehen mit den Methoden `wait` und `notify` der Klasse `Object` noch weitere Synchronisationsprimitve zur Verfügung. Zusätzlich zu der bereits erwähnten Sperre, die einem Objekt zugeordnet ist, besitzt ein Objekt nämlich auch noch eine *Warteliste*. Dabei handelt es sich um eine (möglicherweise leere) Menge von Threads, die vom Scheduler unterbrochen wurden und auf ein Ereignis warten, um fortgesetzt werden zu können.

Sowohl `wait` als auch `notify` dürfen nur aufgerufen werden, wenn das Objekt bereits gesperrt ist, also nur innerhalb eines `synchronized`-Blocks für dieses Objekt. Ein Aufruf von `wait` nimmt die bereits gewährten Sperren (temporär) zurück und stellt den Prozeß, der den Aufruf von `wait` verursachte, in die Warteliste des Objekts. Dadurch wird er unterbrochen und im Scheduler als *wartend* markiert. Ein Aufruf von `notify` entfernt einen (beliebigen) Prozeß aus der Warteliste des Objekts, stellt die (temporär) aufgehobenen Sperren wieder her und führt ihn dem normalen Scheduling zu. `wait` und `notify` sind damit für elementare Synchronisationsaufgaben geeignet, bei denen es weniger auf die Kommunikation als auf die Steuerung der zeitlichen Abläufe ankommt.

Das folgende Beispiel demonstriert den Einsatz von [wait](#) und [notify](#) an einem *Producer/Consumer-Beispiel*.

Ein Prozeß arbeitet dabei als Produzent, der Fließkommazahlen »herstellt«, und ein anderer als Konsument, der die produzierten Daten verbraucht. Die Kommunikation zwischen beiden erfolgt über ein gemeinsam verwendetes [Vector](#)-Objekt, das die produzierten Elemente zwischenspeichert und als Medium für die [wait](#)-/[notify](#)-Aufrufe dient:

[Listing1013.java](#)

```

001 /* Listing1013.java */
002
003 import java.util.*;
004
005 class Producer1013
006 extends Thread
007 {
008     private Vector v;
009
010     public Producer1013(Vector v)
011     {
012         this.v = v;
013     }
014
015     public void run()
016     {
017         String s;
018
019         while (true) {
020             synchronized (v) {
021                 s = "Wert "+Math.random();
022                 v.addElement(s);
023                 System.out.println("Produzent erzeugte "+s);
024                 v.notify();
025             }
026             try {
027                 Thread.sleep((int)(100*Math.random()));
028             } catch (InterruptedException e) {
029                 //nichts
030             }
031         }
032     }
033 }
034
035 class Consumer1013
036 extends Thread
037 {
038     private Vector v;
039
040     public Consumer1013(Vector v)
041     {
042         this.v = v;
043     }
044
045     public void run()
046     {
047         while (true) {
048             synchronized (v) {
049                 if (v.size() < 1) {
050                     try {
051                         v.wait();
052                     } catch (InterruptedException e) {
053                         //nichts
054                     }
055                 }
056                 System.out.print(
057                     " Konsument fand "+(String)v.elementAt(0)
058                 );
059                 v.removeElementAt(0);

```

```

060         System.out.println(" (verbleiben: "+v.size()+")");
061     }
062     try {
063         Thread.sleep((int)(100*Math.random()));
064     } catch (InterruptedException e) {
065         //nichts
066     }
067 }
068 }
069 }
070
071 public class Listing1013
072 {
073     public static void main(String[] args)
074     {
075         Vector v = new Vector();
076
077         Producer1013 p = new Producer1013(v);
078         Consumer1013 c = new Consumer1013(v);
079         p.start();
080         c.start();
081     }
082 }

```

Listing 10.13: Ein Producer-/Consumer-Beispiel mit wait und notify

Um die Arbeitsverteilung zwischen den Prozessen etwas interessanter zu gestalten, werden beide gezwungen, nach jedem Schritt eine kleine Pause einzulegen. Da die Wartezeit zufällig ausgewählt wird, kann es durchaus dazu kommen, daß der Produzent eine größere Anzahl an Elementen anhäuft, die der Konsument noch nicht abgeholt hat. Der umgekehrte Fall ist natürlich nicht möglich, da der Konsument warten muß, wenn keine Elemente verfügbar sind. Eine Beispielsitzung könnte etwa so aussehen:

```

Produzent erzeugte Wert 0.09100924684649958
Konsument fand Wert 0.09100924684649958 (verbleiben: 0)
Produzent erzeugte Wert 0.5429652807455857
Konsument fand Wert 0.5429652807455857 (verbleiben: 0)
Produzent erzeugte Wert 0.6548096532111007
Konsument fand Wert 0.6548096532111007 (verbleiben: 0)
Produzent erzeugte Wert 0.02311095955845288
Konsument fand Wert 0.02311095955845288 (verbleiben: 0)
Produzent erzeugte Wert 0.6277057416210464
Konsument fand Wert 0.6277057416210464 (verbleiben: 0)
Produzent erzeugte Wert 0.6965546173953919
Produzent erzeugte Wert 0.6990053250441516
Produzent erzeugte Wert 0.9874467815778902
Produzent erzeugte Wert 0.12110075531692543
Produzent erzeugte Wert 0.5957795111549329
Konsument fand Wert 0.6965546173953919 (verbleiben: 4)
Produzent erzeugte Wert 0.019655027417308846
Konsument fand Wert 0.6990053250441516 (verbleiben: 4)
Konsument fand Wert 0.9874467815778902 (verbleiben: 3)
Produzent erzeugte Wert 0.14247583735074354
Konsument fand Wert 0.12110075531692543 (verbleiben: 3)

```

Durch eine konstante Pause nach jedem produzierten Element könnte der Produzent bewußt langsamer gemacht werden. Der schnellere Konsument würde dann einen Großteil seiner Zeit damit verbringen, festzustellen, daß keine Elemente verfügbar sind. Zwar würde das Beispiel (in leicht modifizierter Form) auch ohne den Einsatz von `wait/notify` funktionieren. Durch ihre Verwendung aber ist der Konsumentenprozeß nicht gezwungen, *aktiv* zu warten, sondern wird vom Produzenten benachrichtigt, wenn ein neues Element verfügbar ist. Der Rechenzeitbedarf reduziert sich dadurch auf einen Bruchteil dessen, was andernfalls benötigt würde.

Hinweis

10.5 Verwalten von Threads

- [10.5 Verwalten von Threads](#)
 - [10.5.1 Priorität und Name](#)
 - [10.5.2 Thread-Gruppen](#)

Threads können in Java nicht nur ausgeführt und synchronisiert werden, sondern besitzen auch eine Reihe administrativer Eigenschaften, die besonders dann nützlich sind, wenn das Thread-Konzept stark genutzt wird. Diese administrativen Eigenschaften lassen sich in zwei Gruppen unterteilen. Zum einen gibt es Eigenschaften, die bei den Threads selbst zu finden sind, beispielsweise die *Priorität* oder der *Name* eines Threads. Zum anderen gibt es Eigenschaften, die darauf begründet sind, daß jeder Thread in Java zu einer *Thread-Gruppe* gehört, die untereinander über Vater-Kind-Beziehungen miteinander in Verbindung stehen. Beide Gruppen von Eigenschaften sollen im folgenden kurz erklärt werden.

10.5.1 Priorität und Name

Jeder Thread in Java hat einen eindeutigen Namen. Wird dieser nicht explizit vergeben (beispielsweise im Konstruktor), so vergibt Java einen automatisch generierten Namen der Form "**Thread-**" + n, wobei n ein [int](#) ist. Alternativ kann einem Thread mit der Methode [setName](#) auch vom Programm selbst ein Name zugewiesen werden:

```
public void setName(String name)
```

[java.lang.Thread](#)

Mit Hilfe der Methode [getName](#) kann der Name eines Threads abgefragt werden:

```
public String getName()
```

[java.lang.Thread](#)

Neben einem Namen besitzt ein Thread auch eine *Priorität*. Die Priorität steuert den Scheduler in der Weise, daß bei Vorhandensein mehrerer bereiter Threads diejenigen mit höherer Priorität vor denen mit niedrigerer Priorität ausgeführt werden. Beim Anlegen eines neuen Threads bekommt dieser die Priorität des aktuellen Threads. Mit Hilfe der Methode [setPriority](#) kann die Priorität gesetzt und mit [getPriority](#) abgefragt werden:

```
public void setPriority(int newPriority)

public int getPriority()
```

[java.lang.Thread](#)

Als Parameter kann an [setPriority](#) ein Wert übergeben werden, der zwischen den Konstanten [Thread.MIN_PRIORITY](#) und [Thread.MAX_PRIORITY](#) liegt. Der Normalwert wird durch die Konstante [Thread.NORM_PRIORITY](#) festgelegt. In der aktuellen Version des JDK haben diese Konstanten die Werte 1, 5 und 10.

10.5.2 Thread-Gruppen

Thread-Gruppen dienen dazu, Informationen zu verwalten, die nicht nur für einen einzelnen Thread von Bedeutung sind, sondern für eine ganze Gruppe. Weiterhin bieten sie Methoden, die auf alle Threads einer Gruppe angewendet werden können, und sie erlauben es, die Threads einer Gruppe aufzuzählen. In der Java-Klassenbibliothek gibt es zu diesem Zweck eine eigene Klasse [ThreadGroup](#).

Der Einstieg zur Anwendung von Objekten der Klasse [ThreadGroup](#) ist die Methode [getThreadGroup](#) der Klasse [Thread](#). Sie liefert das [ThreadGroup](#)-Objekt, dem der aktuelle [Thread](#) angehört. Alle Thread-Gruppen sind über eine Vater-Kind-Beziehung miteinander verkettet. Mit Hilfe der Methode [getParent](#) von [ThreadGroup](#) kann die Vater-Thread-Gruppe ermittelt werden:

```
public ThreadGroup getParent()
```

[java.lang.ThreadGroup](#)

Durch mehrfache Anwendung kann so die Wurzel aller Thread-Gruppen ermittelt werden. Steht eine [ThreadGroup](#) ganz oben in der Hierarchie, so liefert die Methode [getParent](#) den Wert [null](#).

Um die Menge aller Threads zu bestimmen, die in der aktuellen Thread-Gruppe und ihren Untergruppen enthalten sind, kann die Methode [enumerate](#) verwendet werden:

```
public int enumerate(Thread list[])
```

[java.lang.ThreadGroup](#)

[enumerate](#) erwartet ein Array von Threads, das beim Aufruf mit allen Threads, die zu dieser Thread-Gruppe oder einer ihrer Untergruppen

gehören, gefüllt wird. Das Array sollte vor dem Aufruf bereits auf die erforderliche Größe dimensioniert werden. Als Anhaltspunkt kann dabei das Ergebnis der Methode [activeCount](#) dienen, die die Anzahl der aktiven Threads in der aktuellen Thread-Gruppe und allen Thread-Gruppen, die die aktuelle Thread-Gruppe als Vorfahr haben, angibt:

[java.lang.ThreadGroup](#)

```
public int activeCount()
```

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	<	>	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	----------------------	----------------------	--------------------------

10.6 Zusammenfassung

- [10.6 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Grundlagen des Multithreadings
- Die Klasse [Thread](#) und die Methoden [start](#), [run](#) und [stop](#)
- Die Methoden [interrupt](#), [isInterrupted](#) und die statische Methode [interrupted](#)
- Die Methoden [suspend](#) und [resume](#) sowie [sleep](#), [isAlive](#) und [join](#)
- Das Interface [Runnable](#)
- Erzeugen eines Threads, wenn die Klasse nicht aus [Thread](#) abgeleitet ist
- Synchronisationsprobleme und die Arbeitsweise eines Monitors
- Das Schlüsselwort [synchronized](#)
- Die Methoden [wait](#) und [notify](#)
- Name und Priorität von Threads
- Thread-Gruppen

Kapitel 11

String und StringBuffer

- [11 String und StringBuffer](#)
 - [11.1 Grundlegende Eigenschaften](#)
 - [11.2 Methoden der Klasse String](#)
 - [11.2.1 Konstruktoren](#)
 - [11.2.2 Zeichenextraktion](#)
 - [11.2.3 Die Länge der Zeichenkette](#)
 - [11.2.4 Vergleichen von Zeichenketten](#)
 - [11.2.5 Suchen in Zeichenketten](#)
 - [11.2.6 Ersetzen von Zeichenketten](#)
 - [11.2.7 Konvertierungsfunktionen](#)
 - [11.3 Weitere Eigenschaften](#)
 - [11.3.1 Die Klasse String ist final](#)
 - [11.3.2 Was ist ein String für den Compiler?](#)
 - [String-Literale](#)
 - [String-Verkettung und -Zuweisung](#)
 - [11.3.3 String-Objekte sind nicht dynamisch](#)
 - [11.4 Die Klasse StringBuffer](#)
 - [11.4.1 Konstruktoren](#)
 - [11.4.2 Einfügen von Elementen](#)
 - [11.4.3 Löschen von Elementen](#)
 - [11.4.4 Verändern von Elementen](#)
 - [11.4.5 Längeninformationen](#)
 - [11.4.6 Konvertierung in einen String](#)
 - [11.5 Zusammenfassung](#)

11.1 Grundlegende Eigenschaften

- 11.1 Grundlegende Eigenschaften

Wie in allen anderen Programmiersprachen gibt es auch in Java *Zeichenketten*. Ähnlich wie in C und C++ kennt der Compiler nur einige ihrer elementaren Eigenschaften und überläßt der Laufzeitbibliothek einen Großteil der Implementierung. Wir werden uns in diesem Kapitel die Implementierung und Verwendung von Zeichenketten in Java ansehen und uns dabei der üblichen Praxis anschließen, die Begriffe *String* und *Zeichenkette* synonym zu verwenden.

In Java werden Zeichenketten durch die Klasse `String` repräsentiert. Sie bietet Methoden zum Erzeugen von Zeichenketten, zur Extraktion von Teilstrings, zum Vergleich mit anderen Strings und zur Erzeugung von Strings aus primitiven Typen. Der Compiler erkennt und interpretiert `String`-Litereale und ist in der Lage, `String`-Objekte zu erzeugen und zu verketten.

Eine Zeichenkette hat in Java prinzipiell dieselbe Bedeutung wie in anderen Sprachen. Als Reihung von Elementen des Typs `char` ist sie die wichtigste Datenstruktur für alle Aufgaben, die etwas mit der Ein- und Ausgabe oder der Verarbeitung von Zeichen zu tun haben.

Da der `char`-Typ in Java durch ein Unicode-Zeichen repräsentiert wird, besteht auch ein String aus einer Kette von Unicode-Zeichen. Abgesehen davon, daß er dadurch doppelt soviel Speicher belegt wie ein ASCII-String, braucht man sich darüber aber nicht allzu viele Gedanken zu machen. Wegen der bereits in [Kapitel 4](#) erwähnten Kompatibilität zwischen Unicode-Standard und ASCII-Zeichensatz sind die Unterschiede bei der normalen Verwendung von Strings ohne Belang.

Anders als in C braucht der Java-Programmierer keine Kenntnisse über den internen Aufbau von Strings zu haben. Insbesondere ist nicht davon auszugehen, daß ein String durch ein nullterminiertes Array von `char`-Elementen dargestellt wird. Natürlich heißt dies nicht, daß die interne Struktur von Strings *vollkommen* anonym ist, denn sowohl Compiler als auch Laufzeitsystem müssen sie kennen. Für den Programmierer, der lediglich Java-Programme schreiben will, ist sie aber bedeutungslos.

Hinweis

11.2 Methoden der Klasse String

- [11.2 Methoden der Klasse String](#)
 - [11.2.1 Konstruktoren](#)
 - [11.2.2 Zeichenextraktion](#)
 - [11.2.3 Die Länge der Zeichenkette](#)
 - [11.2.4 Vergleichen von Zeichenketten](#)
 - [11.2.5 Suchen in Zeichenketten](#)
 - [11.2.6 Ersetzen von Zeichenketten](#)
 - [11.2.7 Konvertierungsfunktionen](#)

Die Klasse `String` definiert eine Vielzahl von Methoden zur Manipulation und zur Bestimmung der Eigenschaften von Zeichenketten. Da Strings und die auf ihnen definierten Operationen bei der Anwendung einer Programmiersprache ein wichtiges und für die Effizienz der Programmentwicklung kritisches Merkmal sind, sollen hier die wichtigsten String-Operationen vorgestellt werden. Weitere Methoden dieser Klasse können den Referenzhandbüchern der Java-Klassenbibliothek oder der Online-Hilfe entnommen werden.

11.2.1 Konstruktoren

Die Klasse `String` bietet eine Reihe von Möglichkeiten, neue Instanzen zu erzeugen. Neben der bereits in [Kapitel 4](#) erläuterten Initialisierung mit Hilfe von Literalen besitzt die Klasse eine Reihe von Konstruktoren, die dazu verwendet werden können, `String`-Objekte explizit zu erzeugen.

```
String()
```

Erzeugt ein leeres String-Objekt.

```
String(String value)
```

Erzeugt einen neuen String durch Duplizierung eines bereits vorhandenen.

```
String(char[] value)
```

Erzeugt einen neuen String aus einem vorhandenem Zeichen-Array. Dabei werden alle Elemente des Arrays in den String übernommen.

Nachdem ein `String` erzeugt wurde, kann er wie jedes andere Objekt verwendet werden.

Insbesondere erfolgt damit im Gegensatz zu C oder C++ das Speichermanagement vollautomatisch, ohne daß sich der Programmierer darum kümmern muß.

Hinweis

11.2.2 Zeichenextraktion

```
char charAt(int index)
    throws StringIndexOutOfBoundsException
```

Liefert das Zeichen an Position `index`. Dabei hat das erste Element eines Strings den Index 0 und das letzte den Index `length() - 1`. Falls der String kürzer als `index + 1` Zeichen ist, wird eine Ausnahme des Typs `StringIndexOutOfBoundsException` erzeugt.

```
String substring(int begin, int end)
```

Liefert den Teilstring, der an Position `begin` beginnt und an Position `end` endet. Wie bei allen Zugriffen über einen numerischen Index beginnt auch hier die Zählung bei 0.

Ungewöhnlich bei der Verwendung dieser Methode ist die Tatsache, daß der Parameter `end` auf das erste Zeichen *hinter* den zu extrahierenden Teilstring verweist (siehe [Abbildung 11.1](#)).

Der Rückgabewert ist also die Zeichenkette, die von Indexposition `begin` bis Indexposition `end - 1` reicht.

Hinweis

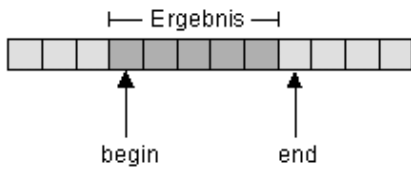


Abbildung 11.1: Ein Aufruf von `substring(begin, end)`

Es gibt noch eine zweite Variante der Methode `substring`, die mit nur einem einzigen Parameter aufgerufen wird. Sie liefert den Teilstring von der angegebenen Position bis zum Ende des Strings.

`java.lang.String`

`String trim()`

Die Methode liefert den String, der entsteht, wenn auf beiden Seiten der Zeichenkette jeweils alle zusammenhängenden Leerzeichen entfernt werden. Dabei werden alle Zeichen, die einen Code kleiner gleich 32 haben, als Leerzeichen angesehen. Leider gibt es keine separaten Methoden für das rechts- oder linksbündige Entfernen von Leerzeichen, `trim` entfernt immer die Leerzeichen auf beiden Seiten. Da die Klasse `String` als `final` deklariert wurde, gibt es auch keine Möglichkeit, entsprechende Methoden nachzurüsten.

11.2.3 Die Länge der Zeichenkette

`java.lang.String`

`int length()`

Liefert die aktuelle Länge des `String`-Objekts. Ist der Rückgabewert 0, so bedeutet dies, daß der String leer ist. Wird ein Wert n größer 0 zurückgegeben, so enthält der String n Zeichen, die an den Indexpositionen 0 bis $n - 1$ liegen.

Das folgende Beispiel zeigt die Verwendung der Methoden `substring` und `length` und der `String`-Verkettung auf (letztere wird in [Abschnitt 11.3.2](#) noch einmal genauer behandelt).

Beispiel

`Listing1101.java`

```
001 /* Listing1101.java */
002
003 public class Listing1101
004 {
005     public static void main(String[] args)
006     {
007         String s1;
008         s1 = "Auf der Mauer";
009         s1 += ", auf der Lauer";
010         s1 += ", sitzt 'ne kleine Wanze";
011         System.out.println(s1);
012
013         for (int i = 1; i <= 5; ++i) {
014             s1 = s1.substring(0,s1.length()-1);
015             System.out.println(s1);
016         }
017     }
018 }
```

Listing 11.1: String-Verkettung und die Methode `substring`

Die Ausgabe des Programmes ist:

```
Auf der Mauer, auf der Lauer, sitzt 'ne kleine Wanze
Auf der Mauer, auf der Lauer, sitzt 'ne kleine Wanz
Auf der Mauer, auf der Lauer, sitzt 'ne kleine Wan
Auf der Mauer, auf der Lauer, sitzt 'ne kleine Wa
Auf der Mauer, auf der Lauer, sitzt 'ne kleine W
Auf der Mauer, auf der Lauer, sitzt 'ne kleine
```

11.2.4 Vergleichen von Zeichenketten

[java.lang.String](#)

```
boolean equals(Object anObject)

boolean equalsIgnoreCase(String s)

boolean startsWith(String s)

boolean endsWith(String s)

int compareTo(String s)

int regionMatches(
    int toffset,
    String other,
    int ooffset,
    int len
)
```

[equals](#) liefert [true](#), wenn das aktuelle String-Objekt und [anObject](#) identisch sind. Analog zu der gleichnamigen Methode der Klasse [Object](#) überprüft [equals](#) also nicht, ob beide Strings dasselbe Objekt referenzieren, sondern testet auf inhaltliche Gleichheit. Interessant an [equals](#) ist die Tatsache, daß als Parameter ein Objekt der Klasse [Object](#) erwartet wird. [equals](#) kann einen String also nicht nur mit einem anderen String vergleichen, sondern mit der String-Darstellung eines beliebigen Objekts.

Neben [equals](#) gibt es noch eine ähnliche Methode, [equalsIgnoreCase](#), die eventuell vorhandene Unterschiede in der Groß-/Kleinschreibung beider Zeichenketten ignoriert.

[startsWith](#) testet, ob das String-Objekt mit der Zeichenkette [s](#) beginnt. Ist das der Fall, so gibt die Methode [true](#) zurück, andernfalls [false](#). [endsWith](#) überprüft dagegen, ob das String-Objekt mit der Zeichenkette [s](#) endet. Ist das der Fall, gibt die Methode ebenfalls [true](#) zurück, andernfalls [false](#).

Die wichtige Methode [compareTo](#) führt einen *lexikalischen* Vergleich beider Strings durch. Bei einem lexikalischen Vergleich werden die Zeichen paarweise von links nach rechts verglichen. Tritt ein Unterschied auf oder ist einer der Strings beendet, wird das Ergebnis ermittelt. Ist das aktuelle String-Objekt dabei kleiner als [s](#), wird ein negativer Wert zurückgegeben. Ist es größer, wird ein positiver Wert zurückgegeben. Bei Gleichheit liefert die Methode den Rückgabewert 0.

[regionMatches](#) vergleicht zwei gleich lange String-Regionen, die in zwei unterschiedlichen Strings an zwei unterschiedlichen Positionen liegen können. Die Region im ersten String beginnt dabei an der Position [toffset](#), die im zweiten String [other](#) an der Position [ooffset](#). Verglichen werden [len](#) Zeichen. Das Ergebnis ist [true](#), wenn beide Teilstrings identisch sind, andernfalls ist es [false](#). Die Methode steht auch in einer Variante zur Verfügung, bei der Groß- und Kleinschreibung ignoriert werden. In diesem Fall wird als zusätzlicher Parameter an erster Stelle ein boolescher Wert [ignoreCase](#) übergeben.

Die Anwendung von [regionMatches](#) soll an folgendem Beispiel erläutert werden. Der erste Vergleich liefert [true](#) (siehe [Abbildung 11.2](#)), der zweite [false](#):

Beispiel

[Listing1102.java](#)

```
001 /* Listing1102.java */
002
003 public class Listing1102
004 {
005     public static void main(String[] args)
006     {
007         System.out.println(
008             "Grüße aus Hamburg".regionMatches(
009                 8,
010                 "Greetings from Australia",
011                 8,
012                 2
013             )
014         );
015         System.out.println(
016             "Grüße aus Hamburg".regionMatches(
017                 6,
018                 "Greetings from Australia",
019                 15,
020                 3
```



```
021      )
022      );
023  }
024 }
```

Listing 11.2: Die Methode `regionMatches` der Klasse `String`

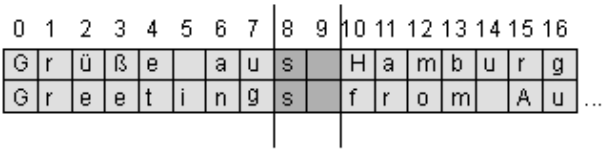


Abbildung 11.2: Der Aufruf von `regionMatches`

11.2.5 Suchen in Zeichenketten

```
int indexOf(String s)
int indexOf(String s, int fromIndex)
int lastIndexOf(String s)
```

Die Methode `indexOf` sucht das erste Vorkommen der Zeichenkette `s` innerhalb des `String`-Objekts. Wird `s` gefunden, liefert die Methode den Index des ersten übereinstimmenden Zeichens zurück, andernfalls wird `-1` zurückgegeben. Die Methode gibt es auch in einer Version, die einen Parameter vom Typ `char` akzeptiert. In diesem Fall sucht sie nach dem ersten Auftreten des angegebenen Zeichens.

Die zweite Variante von `indexOf` arbeitet wie die erste, beginnt mit der Suche aber erst ab Position `fromIndex`. Wird `s` beginnend ab dieser Position gefunden, liefert die Methode den Index des ersten übereinstimmenden Zeichens, andernfalls `-1`. Auch diese Methode gibt es in einer Variante, die anstelle eines `String`-Parameters ein Argument des Typs `char` erwartet. Ihr Verhalten ist analog zur vorherigen.

Die Methode `lastIndexOf` sucht nach dem letzten Vorkommen des Teilstrings `s` im aktuellen `String`-Objekt. Wird `s` gefunden, liefert die Methode den Index des ersten übereinstimmenden Zeichens, andernfalls `-1`. Wie die beiden vorherigen Methoden gibt es auch `lastIndexOf` wahlweise mit einem Parameter vom Typ `char` und mit einem zweiten Parameter, der die Startposition der Suche bestimmt.

11.2.6 Ersetzen von Zeichenketten

```
String toLowerCase()
String toUpperCase()
String replace(char oldchar, char newchar)
```

Die Methode `toLowerCase` liefert den String zurück, der entsteht, wenn alle Zeichen des Argumentstrings in Kleinbuchstaben umgewandelt werden. Besitzt der String keine umwandelbaren Zeichen, wird der Original-String zurückgegeben. `toUpperCase` arbeitet ganz ähnlich, liefert aber den String, der entsteht, wenn alle Zeichen in Großbuchstaben umgewandelt werden. Besitzt der String keine umwandelbaren Zeichen, wird der Original-String zurückgegeben.

Mit Hilfe von `replace` wird eine zeichenweise Konvertierung des aktuellen String-Objekts durchgeführt. Dabei wird jedes Auftreten des Zeichens `oldchar` durch das Zeichen `newchar` ersetzt.

Leider gibt es in der Klasse `String` keine Methode, die das Ersetzen von Teilstrings durch andere Teilstrings (von möglicherweise unterschiedlicher Länge) ermöglicht. Die auf Einzelzeichen basierende Methode `replace` ist das einzige Mittel, um überhaupt Ersetzungen vornehmen zu können. Seit der Version 1.2 gibt es im JDK aber in der Klasse `StringBuffer` eine entsprechende Methode. Sie wird in [Abschnitt 11.4.4](#) beschrieben.

Tip

11.2.7 Konvertierungsfunktionen

Da es oftmals nötig ist, primitive Datentypen in Zeichenketten umzuwandeln, bietet Java eine ganze Reihe von Methoden, die zu diesem Zweck entwickelt wurden. Allen Methoden ist der Name `valueOf` und die Tatsache, daß sie exakt einen Parameter erwarten, gemein.

Alle `valueOf`-Methoden sind als Klassenmethoden implementiert, d.h., sie können auch ohne ein `String`-Objekt aufgerufen werden. Dies macht Sinn, denn sie werden ja in aller Regel erst dazu verwendet, Strings zu erzeugen. Ein Aufruf von `valueOf` wandelt ein primitives Objekt mit Hilfe

der Methode [toString](#), die von der zugehörigen Wrapper-Klasse zur Verfügung gestellt wird, in eine Zeichenkette um. Die wichtigsten [valueOf](#)-Methoden sind:

[java.lang.String](#)

static String valueOf(boolean b)

static String valueOf(char c)

static String valueOf(char c[])

static String valueOf(double d)

static String valueOf(float f)

static String valueOf(int i)

static String valueOf(long l)

static String valueOf(Object obj)

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	<	>	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	----------------------	----------------------	--------------------------

11.3 Weitere Eigenschaften

- [11.3 Weitere Eigenschaften](#)
 - [11.3.1 Die Klasse String ist final](#)
 - [11.3.2 Was ist ein String für den Compiler?](#)
 - [String-Literale](#)
 - [String-Verkettung und -Zuweisung](#)
 - [11.3.3 String-Objekte sind nicht dynamisch](#)

11.3.1 Die Klasse String ist final

Beim Studium der Klassenbibliotheken erkennt man, daß die Klasse [String](#) mit dem Attribut [final](#) belegt ist. Einer der Gründe für diese Maßnahme ist die dadurch gesteigerte Effizienz beim Aufruf der Methoden von [String](#)-Objekten zu sehen. Anstelle der dynamischen Methodensuche, die normalerweise erforderlich ist, kann der Compiler [final](#)-Methoden statisch kompilieren und dadurch schneller aufrufen. Daneben spielten aber auch Sicherheitsüberlegungen und Aspekte des Multithreadings eine Rolle.

Leider hat dies den großen Nachteil, daß aus der [String](#)-Klasse keine neuen Klassen abgeleitet werden können. Es gibt also keine Möglichkeit, die vorhandenen Methoden auf *natürliche* Art und Weise zu ergänzen oder zu modifizieren. Soll beispielsweise eine neue Methode [replace](#) geschrieben werden, die in der Lage ist, Strings (und nicht nur einzelne Zeichen) gegeneinander auszutauschen, so bleibt nur der Umweg über eine Methode einer anderen Klasse, die den zu modifizierenden String als Parameter übergeben bekommt. Alternativ könnte man natürlich auch die komplette [String](#)-Klasse neu schreiben. Leider sind beide Varianten unschön und konterkarieren die Vorteile der objektorientierten Implementierung von Zeichenketten in Java.

Hinweis

11.3.2 Was ist ein String für den Compiler?

In Java gibt es an vielen Stellen Verbindungen zwischen dem Compiler und der Laufzeitbibliothek, und zwar insofern, als der Compiler Kenntnis über interne Eigenschaften bestimmter Klassen hat und die Fähigkeit besitzt, Instanzen dieser Klassen zu erzeugen und zu manipulieren. So ist er beispielsweise in der Lage, Code zu generieren, um [String](#)-Objekte zu erzeugen, einander zuzuweisen oder mit Hilfe des [+](#)-Operators zu verketteten.

String-Literale

Jedes [String](#)-Literal ist eine Referenz auf ein Objekt der Klasse [String](#). Wenn der Compiler beim Übersetzen des Quelltextes ein [String](#)-Literal findet, erzeugt er ein neues [String](#)-Objekt und verwendet es anstelle des Literals.

String-Verkettung und -Zuweisung

In Java ist der Operator [+](#) auch auf Strings definiert. Auf zwei [String](#)-Objekte angewendet, liefert er die Verkettung beider Objekte, d.h. ihre Hintereinanderschreibung. Eine Möglichkeit, die [String](#)-Verkettung zu übersetzen, könnte darin bestehen, ein temporäres [StringBuffer](#)-Objekt zu konstruieren und die Operanden mit Hilfe der [append](#)-Methode anzuhängen. Das resultierende Objekt könnte dann mit der [toString](#)-Methode effizient in einen [String](#) konvertiert werden. Die Klasse [StringBuffer](#) dient dazu, *veränderliche* Strings zu implementieren. Sie wird im nächsten Abschnitt vorgestellt.

Diese Vorgehensweise soll an dem folgenden Beispiel erläutert werden. Das Programm gibt zweimal hintereinander »Hallo, Welt« aus. Beim ersten Mal wird der [+](#)-Operator verwendet, beim zweiten Mal die Variante mit dem temporären [StringBuffer](#)-Objekt:

Beispiel

```

001 /* Listing1103.java */
002
003 public class Listing1103
004 {
005     public static void main(String[] args)
006     {
007         String a, b, c;
008
009         //Konventionelle Verkettung
010         a = "Hallo";
011         b = "Welt";
012         c = a + ", " + b;
013         System.out.println(c);
014
015         //So könnte es der Compiler übersetzen
016         a = "Hallo";
017         b = "Welt";
018         c =(new StringBuffer(a)).append(", ").append(b).toString();
019         System.out.println(c);
020     }
021 }

```

Listing 11.3: Implementierung der String-Verkettung

Allerdings ist ein Java-Compiler nicht gezwungen, genau so vorzugehen. Die Sprachspezifikation gibt lediglich die Empfehlung, daß es aus Effizienzgründen sinnvoll sein kann, eine solche oder ähnliche Implementierung zu wählen.

11.3.3 String-Objekte sind nicht dynamisch

Interessanterweise werden durch die Klasse [String](#) keine *dynamischen* Zeichenketten implementiert. Nach der Initialisierung eines [String](#) bleiben dessen Länge und Inhalt konstant. Wie ist dann aber die genaue Arbeitsweise von Funktionen wie [substring](#) oder [replace](#) zu erklären, und was genau passiert bei einer Anweisung wie der folgenden:

```

String s = "hello, world";
s = s.substring(0,5);

```

Die Antwort darauf ist ganz einfach. Die [substring](#)-Methode nimmt die Veränderungen nicht auf dem Original-String vor, sondern erzeugt eine Kopie, die mit dem gewünschten Inhalt gefüllt wird. Diese gibt sie dann an den Aufrufer zurück, der das Ergebnis erneut an [s](#) zuweist und damit die Originalinstanz für den Garbage Collector freigibt. Durch den Referenzcharakter von Objekten und das automatische Speichermanagement entsteht also der Eindruck, als wären [String](#)-Objekte veränderlich.

Als Entwickler braucht man sich hierüber normalerweise keine Gedanken zu machen. Dennoch ist es manchmal nützlich, Zeichenketten zur Verfügung zu haben, die sich dynamisch verändern können (beispielsweise benötigt sie der Compiler selbst zur Implementierung der String-Verkettung). In Java gibt es zu diesem Zweck die Klasse [StringBuffer](#). Sie arbeitet ähnlich wie [String](#), implementiert aber Zeichenketten, die ihre Länge zur Laufzeit ändern können. [StringBuffer](#) besitzt nicht so viele Methoden zur Auswertung der Zeichenkette, sondern legt den Schwerpunkt auf Operationen zur Veränderung ihres Inhalts. Die wichtigsten Methoden sollen im folgenden vorgestellt werden.

11.4 Die Klasse StringBuffer

- [11.4 Die Klasse StringBuffer](#)
 - [11.4.1 Konstruktoren](#)
 - [11.4.2 Einfügen von Elementen](#)
 - [11.4.3 Löschen von Elementen](#)
 - [11.4.4 Verändern von Elementen](#)
 - [11.4.5 Längeninformationen](#)
 - [11.4.6 Konvertierung in einen String](#)

11.4.1 Konstruktoren

```

StringBuffer()
StringBuffer(String s)

```

Der parameterlose Konstruktor erzeugt einen leeren [StringBuffer](#). Wird dagegen ein [String](#) übergeben, erzeugt der Konstruktor ein [StringBuffer](#)-Objekt, das eine Kopie der übergebenen Zeichenkette darstellt.

11.4.2 Einfügen von Elementen

```

StringBuffer append(String s)
StringBuffer insert(int offset, String s)

```

Mit [append](#) wird der String [s](#) an das Ende des [StringBuffer](#)-Objekts angehängt. Zurückgegeben wird das auf diese Weise verlängerte [StringBuffer](#)-Objekt [s](#). Zusätzlich gibt es die Methode [append](#) in Varianten für das Anhängen aller Arten von primitiven Typen. Anstelle eines [String](#)-Objekts wird hier der entsprechende primitive Typ übergeben, in einen String konvertiert und an das Ende des Objekts angehängt. [insert](#) fügt den String [s](#) an der Position [index](#) in den aktuellen [StringBuffer](#) ein. Zurückgegeben wird das auf diese Weise verlängerte [StringBuffer](#)-Objekt [s](#). Auch diese Methode gibt es für primitive Typen. Der anstelle eines [String](#) übergebene Wert wird zunächst in einen [String](#) konvertiert und dann an der gewünschten Stelle eingefügt.

11.4.3 Löschen von Elementen

```

public StringBuffer deleteCharAt(int index)
public StringBuffer delete(int start, int end)

```

Mit [deleteCharAt](#) wird das an Position [index](#) stehende Zeichen entfernt und der [StringBuffer](#) um ein Zeichen verkürzt. [delete](#) entfernt den Teilstring, der von Position [start](#) bis [end](#) reicht, aus dem [StringBuffer](#) und verkürzt ihn um die entsprechende Anzahl Zeichen.

11.4.4 Verändern von Elementen

```

void setCharAt(int index, char c)
    throws StringIndexOutOfBoundsException
public StringBuffer replace(int start, int end, String str)

```

Mit der Methode [setCharAt](#) wird das an Position [index](#) stehende Zeichen durch das Zeichen [c](#) ersetzt. Falls der [StringBuffer](#) zu kurz ist (also [index](#) hinter das Ende des [StringBuffer](#)-Objekts zeigt), löst die Methode eine Ausnahme des Typs [StringIndexOutOfBoundsException](#) aus.

Seit dem JDK 1.2 gibt es zusätzlich eine Methode [replace](#), mit der ein Teil des [StringBuffer](#)-Objekts durch einen anderen [String](#) ersetzt werden kann. Dabei wird das von Position `start` bis Position `end - 1` gehende Teilstück durch den [String](#) `str` ersetzt. Falls erforderlich, wird der ursprüngliche [StringBuffer](#) verkürzt oder verlängert.

JDK1.1/1.2

11.4.5 Längeninformationen

[java.lang.StringBuffer](#)

```
int length()  
  
public int capacity()
```

[length](#) liefert die Länge des Objekts, also die Anzahl der Zeichen, die im [StringBuffer](#) enthalten sind. Mit [capacity](#) wird dagegen die Größe des belegten Pufferspeichers ermittelt. Dieser Wert ist typischerweise größer als der von [length](#) zurückgegebene Wert.

11.4.6 Konvertierung in einen String

[java.lang.StringBuffer](#)

```
String toString()
```

Nachdem die Konstruktion eines [StringBuffer](#)-Objekts abgeschlossen ist, kann es mit Hilfe dieser Methode effizient in einen [String](#) verwandelt werden. Die Methode legt dabei keine Kopie des [StringBuffer](#)-Objekts an, sondern liefert einen Zeiger auf den internen Zeichenpuffer. Erst wenn der [StringBuffer](#) erneut verändert werden soll, wird tatsächlich eine Kopie erzeugt.

Die sinnvolle Anwendung der Klassen [String](#) und [StringBuffer](#) hat bei vielen Programmen großen Einfluß auf ihr Laufzeitverhalten. In [Kapitel 29](#) gehen wir daher noch einmal auf die speziellen Performanceaspekte beider Klassen ein und geben Hinweise zu ihrer sinnvollen Anwendung.

Hinweis

11.5 Zusammenfassung

- [11.5 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Klasse [String](#) als Instrument zur Verarbeitung von Zeichenketten
- [String](#)-Literele und Konstruktion von [String](#)-Objekten
- Extraktion von Teilzeichenketten mit [charAt](#), [substring](#) und [trim](#)
- Die Methode [length](#) zur Bestimmung der Länge eines Strings
- Vergleichen von Zeichenketten mit den Methoden [equals](#), [startsWith](#), [endsWith](#), [compareTo](#) und [regionMatches](#)
- Suchen in Strings mit [indexOf](#) und [lastIndexOf](#)
- Konvertierung von Strings mit [toLowerCase](#), [toUpperCase](#) und [replace](#)
- Die Umwandlungsmethoden [toString](#) und [valueOf](#)
- Die Klasse [StringBuffer](#) zum Erzeugen von dynamischen Strings
- Die Methoden [append](#), [insert](#) und [setCharAt](#) der Klasse [StringBuffer](#)
- Die effiziente Konvertierung zwischen [StringBuffer](#) und [String](#) mit Hilfe von [toString](#)

Kapitel 12

Utilities

- [12 Utilities](#)
 - [12.1 Die Klasse Vector](#)
 - [12.1.1 Einfügen von Elementen](#)
 - [12.1.2 Zugriff auf Elemente](#)
 - [12.1.3 Der Vektor als Iterator](#)
 - [12.2 Die Klasse Stack](#)
 - [12.3 Die Klasse Hashtable](#)
 - [12.3.1 Einfügen von Elementen](#)
 - [12.3.2 Zugriff auf Elemente](#)
 - [12.3.3 Hashtable als Iterator](#)
 - [12.3.4 Die Klasse Properties](#)
 - [12.4 Die Klasse BitSet](#)
 - [12.4.1 Elementweise Operationen](#)
 - [12.4.2 Mengenorientierte Operationen](#)
 - [12.5 Die Klasse StringTokenizer](#)
 - [12.5.1 Anlegen eines StringTokenizerers](#)
 - [12.5.2 Zugriff auf Tokens](#)
 - [12.6 Die Klasse Random](#)
 - [12.6.1 Initialisierung des Zufallszahlengenerators](#)
 - [12.6.2 Erzeugen von Zufallszahlen](#)
 - [Gleichverteilte Zufallszahlen](#)
 - [Normalverteilte Zufallszahlen](#)
 - [12.7 Die Klassen Date, Calendar und GregorianCalendar](#)
 - [12.7.1 Konstruktoren](#)
 - [12.7.2 Abfragen und Setzen von Datumsbestandteilen](#)
 - [12.7.3 Vergleiche und Datums-/Zeitarithmetik](#)
 - [12.7.4 Umwandlung zwischen Date und Calendar](#)
 - [12.8 Die Klasse System](#)
 - [12.8.1 System-Properties](#)
 - [12.8.2 in, err und out](#)
 - [12.8.3 exit](#)
 - [12.8.4 gc](#)
 - [12.8.5 currentTimeMillis](#)
 - [12.8.6 arraycopy](#)
 - [12.9 Die Klasse Arrays](#)
 - [12.10 Zusammenfassung](#)

12.1 Die Klasse Vector

- [12.1 Die Klasse Vector](#)
 - [12.1.1 Einfügen von Elementen](#)
 - [12.1.2 Zugriff auf Elemente](#)
 - [12.1.3 Der Vektor als Iterator](#)

Die Klasse [Vector](#) aus dem Paket [java.util](#) ist die Java-Repräsentation einer *linearen Liste*. Die Liste kann Elemente beliebigen Typs enthalten, und ihre Länge ist zur Laufzeit veränderbar. [Vector](#) erlaubt das Einfügen von Elementen an beliebiger Stelle und bietet sowohl sequentiellen als auch wahlfreien Zugriff auf die Elemente. Das JDK realisiert einen [Vector](#) als Array von Elementen des Typs [Object](#). Daher sind Zugriffe auf vorhandene Elemente und das Durchlaufen der Liste schnelle Operationen. Löschungen und Einfügungen, die die interne Kapazität des Arrays überschreiten, sind dagegen relativ langsam, weil Teile des Arrays umkopiert werden müssen. In der Praxis können diese implementierungsspezifischen Details allerdings meist vernachlässigt werden, und ein [Vector](#) kann als konkrete Implementierung einer linearen Liste angesehen werden.

Neben der Klasse [Vector](#) werden in den ersten Abschnitten dieses Kapitels weitere Klassen zum Speichern und Bearbeiten einer *Menge von Objekten* behandelt. Diese Klassen waren in den JDKs der Versionen 1.0 und 1.1 die einzigen Collections in der Java-Klassenbibliothek. Seit dem JDK 1.2 gibt es darüber hinaus ein eigenes *Collection-API*, das weitere Klassen und zusätzliche Schnittstellen zur Verfügung stellt. Wir werden diese neuen Features in [Kapitel 27](#) ausführlich erläutern.

JDK1.1/1.2

12.1.1 Einfügen von Elementen

Das Anlegen eines neuen Vektors kann mit Hilfe des parameterlosen Konstruktors erfolgen:

```
public Vector()
```

[java.util.Vector](#)

Nach dem Anlegen ist ein [Vector](#) zunächst leer, d.h., er enthält keine Elemente. Durch Aufruf von [isEmpty](#) kann geprüft werden, ob ein [Vector](#) leer ist; [size](#) liefert die Anzahl der Elemente:

```
public final boolean isEmpty()

public final int size()
```

[java.util.Vector](#)

Elemente können an beliebiger Stelle in die Liste eingefügt werden. Ein [Vector](#) erlaubt die Speicherung beliebiger Objekttypen, denn die Einfüge- und Zugriffsmethoden arbeiten mit Instanzen der Klasse [Object](#). Da jede Klasse letztlich aus [Object](#) abgeleitet ist, können auf diese Weise beliebige Objekte in die Liste eingefügt werden.

Leider ist der Zugriff auf die gespeicherten Elemente damit natürlich nicht *typsicher*. Der Compiler kann nicht wissen, welche Objekte an welcher Stelle im [Vector](#) gespeichert wurden und geht daher davon aus, daß beim Zugriff auf Elemente eine Instanz der Klasse [Object](#) geliefert wird. Mit Hilfe des Typkonvertierungsoperators muß diese dann in das ursprüngliche Objekt zurückverwandelt werden. Die Verantwortung für korrekte Typisierung liegt damit beim Entwickler. Mit Hilfe des Operators [instanceof](#) kann bei Bedarf zumindest eine Laufzeit-Typüberprüfung vorgeschaltet werden.

Warnung

Neue Elemente können wahlweise an das Ende des Vektors oder an einer beliebigen anderen Stelle eingefügt werden. Das Einfügen am Ende erfolgt mit der Methode [addElement](#):

```
public void addElement(Object obj)
```

[java.util.Vector](#)

In diesem Fall wird das Objekt `obj` an das Ende der bisherigen Liste von Elementen angehängt.

Soll ein Element dagegen an einer beliebigen Stelle innerhalb der Liste eingefügt werden, ist die Methode [insertElementAt](#) zu verwenden:

```
public void insertElementAt(Object obj, int index)
    throws ArrayIndexOutOfBoundsException
```

[java.util.Vector](#)

Diese Methode fügt das Objekt `obj` an der Position `index` in den Vektor ein. Alle bisher an dieser oder einer dahinter liegenden Position befindlichen Elemente werden um eine Position weitergeschoben.

12.1.2 Zugriff auf Elemente

Ein Vektor bietet sowohl *sequentiellen* als auch *wahlfreien* Zugriff auf seine Elemente. Für den sequentiellen Zugriff bietet es sich an, den im nachfolgenden Abschnitt beschriebenen *Iterator* zu verwenden. Der wahlfreie Zugriff erfolgt mit einer der Methoden `firstElement`, `lastElement` oder `elementAt`:

```
public Object firstElement()
    throws ArrayIndexOutOfBoundsException

public Object lastElement()
    throws ArrayIndexOutOfBoundsException

public Object elementAt(int index)
    throws ArrayIndexOutOfBoundsException
```

`firstElement` liefert das erste Element, `lastElement` das letzte. Mit Hilfe von `elementAt` wird auf das Element an Position `index` zugegriffen. Alle drei Methoden verursachen eine Ausnahme, wenn das gesuchte Element nicht vorhanden ist.

12.1.3 Der Vektor als Iterator

Für den sequentiellen Zugriff auf die Elemente des Vektors steht ein *Iterator* zur Verfügung. Ein Iterator ist eine Abstraktion für den aufeinanderfolgenden Zugriff auf alle Elemente einer komplexen Datenstruktur. Ein Iterator wird in Java durch das Interface `Enumeration` zur Verfügung gestellt und deshalb in der Java-Welt oft auch *Enumerator* genannt.

Das Interface `Enumeration` (das bereits in [Kapitel 7](#) kurz vorgestellt wurde) besitzt die Methoden `hasMoreElements` und `nextElement`. Nach der Initialisierung zeigt ein `Enumeration`-Objekt auf das erste Element der Aufzählung. Durch Aufruf von `hasMoreElements` kann geprüft werden, ob weitere Elemente in der Aufzählung enthalten sind, und `nextElement` setzt den internen Zeiger auf das nächste Element:

```
public boolean hasMoreElements()

public Object nextElement()
    throws NoSuchElementException
```

In der Klasse `Vector` liefert die Methode `elements` einen Enumerator für alle Elemente, die sich im Vektor befinden:

```
public Enumeration elements()
```

Das folgende Beispiel verdeutlicht die Anwendung von `elements`:

Listing 12.1: Die Methode `elements` der Klasse `Vector`

Beispiel

```
001 /* Listing1201.java */
002
003 import java.util.*;
004
005 public class Listing1201
006 {
007     public static void main(String[] args)
008     {
009         Vector v = new Vector();
010
011         v.addElement("eins");
012         v.addElement("drei");
013         v.insertElementAt("zwei",1);
014         for (Enumeration el=v.elements(); el.hasMoreElements(); ) {
015             System.out.println((String)el.nextElement());
016         }
017     }
018 }
```

Das Programm erzeugt einen `Vector`, fügt die Werte "eins", "zwei" und "drei" ein und gibt sie anschließend auf dem Bildschirm aus:

eins
zwei
drei

Ein Enumerator ist immer dann nützlich, wenn die Elemente eines zusammengesetzten Datentyps nacheinander aufgezählt werden sollen. Enumeratoren werden in Java noch an verschiedenen anderen Stellen zur Verfügung gestellt, beispielsweise in den Klassen [Hashtable](#) oder [StringTokenizer](#).

Tip

12.2 Die Klasse Stack

- 12.2 Die Klasse Stack

Ein Stack ist eine Datenstruktur, die nach dem *LIFO-Prinzip* (last-in-first-out) arbeitet. Die Elemente werden am vorderen Ende der Liste eingefügt und von dort auch wieder entnommen. Das heißt, die zuletzt eingefügten Elemente werden zuerst entnommen und die zuerst eingefügten zuletzt.

In Java ist ein [Stack](#) eine Ableitung eines Vektors, der um neue Zugriffsfunktionen erweitert wurde, um das typische Verhalten eines Stacks zu implementieren. Obwohl dies eine ökonomische Vorgehensweise ist, bedeutet es, daß ein [Stack](#) alle Methoden eines Vektors erbt und damit auch wie ein Vektor verwendet werden kann. Wir wollen diese Tatsache hier ignorieren und uns mit den spezifischen Eigenschaften eines Stacks beschäftigen.

Der Konstruktor der Klasse [Stack](#) ist parameterlos:

[java.util.Stack](#)

```
public Stack()
```

Das Anfügen neuer Elemente wird durch einen Aufruf der Methode [push](#) erledigt und erfolgt wie üblich am oberen Ende des Stacks. Die Methode liefert als Rückgabewert das eingefügte Objekt:

[java.util.Stack](#)

```
public Object push(Object item)
```

Der Zugriff auf das oberste Element kann mit einer der Methoden [pop](#) oder [peek](#) erfolgen. Beide liefern das oberste Element des Stacks, [pop](#) entfernt es anschließend vom Stack:

[java.util.Stack](#)

```
public Object pop()
```

```
public Object peek()
```

Des weiteren gibt es eine Methode [empty](#), um festzustellen, ob der [Stack](#) leer ist, und eine Methode [search](#), die nach einem beliebigen Element sucht und als Rückgabewert die Distanz zwischen gefundenem Element und oberstem Stack-Element angibt:

[java.util.Stack](#)

```
public boolean empty()
```

```
public int search(Object o)
```

Das folgende Beispiel verdeutlicht die Anwendung eines Stacks:

Beispiel

[Listing1202.java](#)

```
001 /* Listing1202.java */
002
003 import java.util.*;
004
005 public class Listing1202
006 {
007     public static void main(String[] args)
008     {
009         Stack s = new Stack();
010
011         s.push("Erstes Element");
012         s.push("Zweites Element");
013         s.push("Drittes Element");
014         while (true) {
015             try {
016                 System.out.println(s.pop());
017             } catch (EmptyStackException e) {
018                 break;
019             }
020         }
021     }
022 }
```

Listing 12.2: Anwendung eines Stacks

Das Programm erzeugt einen [Stack](#) und fügt die Werte "Erstes Element", "Zweites Element" und "Drittes Element" ein. Anschließend entfernt es so lange Elemente aus dem Stack, bis die Ausgabeschleife durch eine Ausnahme des Typs [EmptyStackException](#) beendet wird. Durch die Arbeitsweise des [Stack](#) werden die Elemente in der umgekehrten Eingabereihenfolge auf dem Bildschirm ausgegeben:

Drittes Element
Zweites Element
Erstes Element

12.3 Die Klasse Hashtable

- [12.3 Die Klasse Hashtable](#)
 - [12.3.1 Einfügen von Elementen](#)
 - [12.3.2 Zugriff auf Elemente](#)
 - [12.3.3 Hashtable als Iterator](#)
 - [12.3.4 Die Klasse Properties](#)

Die Klasse [Hashtable](#) ist eine Konkretisierung der abstrakten Klasse [Dictionary](#). Diese stellt einen *assoziativen* Speicher dar, der *Schlüssel* auf *Werte* abbildet und über den Schlüsselbegriff einen effizienten Zugriff auf den Wert ermöglicht. Ein Dictionary speichert also immer zusammengehörige Paare von Daten, bei denen der Schlüssel als Name des zugehörigen Wertes angesehen werden kann. Über den Schlüssel kann später der Wert leicht wiedergefunden werden.

Da ein Dictionary auf unterschiedliche Weise implementiert werden kann, haben die Java-Designer entschieden, dessen abstrakte Eigenschaften in einer Basisklasse zusammenzufassen. Die Implementierung [Hashtable](#) benutzt das Verfahren der *Schlüsseltransformation*, also die Verwendung einer Transformationsfunktion (auch *Hash-Funktion* genannt), zur Abbildung von Schlüsseln auf Indexpositionen eines Arrays. Weitere Konkretisierungen der Klasse [Dictionary](#), etwa auf der Basis binärer Bäume, gibt es in Java derzeit nicht.

Neben den erwähnten abstrakten Eigenschaften besitzt [Hashtable](#) noch die konkreten Merkmale *Kapazität* und *Ladefaktor*. Die Kapazität gibt die Anzahl der Elemente an, die insgesamt untergebracht werden können. Der Ladefaktor zeigt dagegen an, bei welchem Füllungsgrad die Hash-Tabelle vergrößert werden muß. Das Vergrößern erfolgt automatisch, wenn die Anzahl der Elemente innerhalb der Tabelle größer ist als das Produkt aus Kapazität und Ladefaktor. Seit dem JDK 1.2 darf der Ladefaktor auch größer als 1 sein. In diesem Fall wird die [Hashtable](#) also erst dann vergrößert, wenn der Füllungsgrad größer als 100 % ist und bereits ein Teil der Elemente in den Überlaufbereichen untergebracht wurde.

Wichtig bei der Verwendung der [Dictionary](#)-Klassen ist, daß das Einfügen und der Zugriff auf Schlüssel nicht auf der Basis des Operators `==`, sondern mit Hilfe der Methode [equals](#) erfolgt. Schlüssel müssen daher lediglich *inhaltlich* gleich sein, um als identisch angesehen zu werden. Eine Referenzgleichheit ist dagegen nicht erforderlich.

Hinweis

12.3.1 Einfügen von Elementen

Eine Instanz der Klasse [Hashtable](#) kann mit Hilfe eines parameterlosen Konstruktors angelegt werden:

```

public Hashtable()

```

Das Einfügen von Elementen erfolgt durch Aufruf der Methode [put](#):

```

public Object put(Object key, Object value)

```

Dieser Aufruf fügt das Schlüssel-Werte-Paar (*key*, *value*) in die [Hashtable](#) ein. Weder *key* noch *value* dürfen dabei [null](#) sein. Falls bereits ein Wertepaar mit dem Schlüssel *key* enthalten ist, wird der bisherige Wert gegen den neuen ausgetauscht, und [put](#) liefert in diesem Fall den Wert zurück, der bisher dem Schlüssel zugeordnet war. Falls der Schlüssel bisher noch nicht vorhanden ist, ist der Rückgabewert [null](#).

12.3.2 Zugriff auf Elemente

Der Zugriff auf ein Element erfolgt mit Hilfe der Methode [get](#) über den ihm zugeordneten Schlüssel. [get](#) erwartet ein Schlüsselobjekt und liefert den dazu passenden Wert. Falls der angegebene Schlüssel nicht enthalten war, ist der Rückgabewert [null](#):

```

public Object get(Object key)

```

Zusätzlich zu den bisher erwähnten Methoden gibt es noch zwei weitere mit den Namen [contains](#) und [containsKey](#). Sie überprüfen, ob ein bestimmter Wert bzw. ein bestimmter Schlüssel in der [Hashtable](#) enthalten ist:

```

public boolean contains(Object value)

public boolean containsKey(Object key)

```

Der Rückgabewert ist [true](#), falls das gesuchte Element enthalten ist, andernfalls ist er [false](#). Bei der Verwendung dieser Funktionen ist zu

beachten, daß die Suche nach einem Wert sehr wahrscheinlich viel ineffizienter ist, als die Suche nach einem Schlüssel. Während der Schlüssel über die Transformationsfunktion sehr schnell gefunden wird, erfordert die Suche nach einem Wert einen sequentiellen Durchlauf durch die Tabelle.

12.3.3 Hashtable als Iterator

In der Klasse [Hashtable](#) gibt es zwei Iteratoren, die zur Auflistung von Schlüsseln und Werten verwendet werden können:

```
public Enumeration elements()  
  
public Enumeration keys()
```

Die Methode [elements](#) liefert einen Iterator für die Auflistung aller Werte in der [Hashtable](#). In welcher Reihenfolge die Elemente dabei durchlaufen werden, ist nicht definiert. Da eine Hash-Funktion die Eigenschaft hat, Schlüssel gleichmäßig über den verfügbaren Speicher zu verteilen, ist davon auszugehen, daß die Iteratoren ihre Rückgabewerte in einer zufälligen Reihenfolge liefern.

Analog zu [elements](#) liefert [keys](#) eine Auflistung aller Schlüssel, die sich in der Hash-Tabelle befinden. Wie üblich liefern beide Methoden ein Objekt, welches das Interface [Enumeration](#) implementiert. Wie weiter oben erklärt, erfolgt der Zugriff daher mit Hilfe der Methoden [hasMoreElements](#) und [nextElement](#).

Das folgende Beispiel verdeutlicht die Anwendung einer [Hashtable](#):

Beispiel
[Listing1203.java](#)

```
001 /* Listing1203.java */  
002  
003 import java.util.*;  
004  
005 public class Listing1203  
006 {  
007     public static void main(String[] args)  
008     {  
009         Hashtable h = new Hashtable();  
010  
011         //Pflege der Aliase  
012         h.put("Fritz","f.mueller@test.de");  
013         h.put("Franz","fk@b-blabla.com");  
014         h.put("Paula","user0125@mail.uofm.edu");  
015         h.put("Lissa","lb3@gateway.fhdto.northsurf.dk");  
016  
017         //Ausgabe  
018         Enumeration e = h.keys();  
019         while (e.hasMoreElements()) {  
020             String alias = (String)e.nextElement();  
021             System.out.println(  
022                 alias + " --> " + h.get(alias)  
023             );  
024         }  
025     }  
026 }
```

Listing 12.3: Anwendung der Klasse Hashtable

Das Programm legt eine leere [Hashtable](#) an, die zur Aufnahme von Mail-Aliasen verwendet werden soll. Dazu soll zu jeder E-Mail-Adresse ein kurzer Aliasname gepflegt werden, unter dem die lange Adresse später angesprochen werden kann. Das Programm legt zunächst die Aliase »Fritz«, »Franz«, »Paula« und »Lissa« an und assoziiert jeden mit der zugehörigen E-Mail-Adresse. Anschließend durchläuft es alle Schlüssel und gibt zu jedem den dazu passenden Wert aus. Die Ausgabe des Programms ist:

```
Franz --> fk@b-blabla.com  
Fritz --> f.mueller@test.de  
Paula --> user0125@mail.uofm.edu  
Lissa --> lb3@gateway.fhdto.northsurf.dk
```

12.3.4 Die Klasse Properties

Die Klasse [Properties](#) ist aus [Hashtable](#) abgeleitet und repräsentiert ein auf [String](#)-Paare spezialisiertes Dictionary, das es erlaubt, seinen Inhalt auf einen externen Datenträger zu speichern oder von dort zu laden. Ein solches Objekt wird auch als *Property-Liste* (oder *Eigenschaften-Liste*) bezeichnet. Zur Instanzierung stehen zwei Konstruktoren zur Verfügung:

```
public Properties()
```

```
public Properties(Properties defaults)
```

Der erste legt eine leere Property-Liste an, der zweite füllt sie mit den übergebenen Default-Werten. Der Zugriff auf die einzelnen Elemente erfolgt mit den Methoden [getProperty](#) und [propertyNames](#):

java.util.Properties

```
public String getProperty(String key)
```

```
public String getProperty(String key, String defaultValue)
```

```
public Enumeration propertyNames()
```

Die erste Variante von [getProperty](#) liefert die Eigenschaft mit der Bezeichnung [key](#). Ist sie nicht vorhanden, wird [null](#) zurückgegeben. Die zweite Variante hat dieselbe Aufgabe, gibt aber den Standardwert [defaultValue](#) zurück, wenn die gesuchte Eigenschaft nicht gefunden wurde. Mit [propertyNames](#) kann ein [Enumeration](#)-Objekt beschafft werden, mit dem alle Eigenschaften der Property-Liste aufgezählt werden können.

Zum Speichern einer Property-Liste steht die Methode [store](#) zur Verfügung:

java.util.Properties

```
public void store(OutputStream out, String header)
    throws IOException
```

Sie erwartet einen [OutputStream](#) zur Ausgabe (siehe [Kapitel 13](#)) und einen Header, der als Kommentar in die Ausgabedatei geschrieben wird. Das Gegenstück zu [store](#) ist [load](#), mit der eine Property-Liste gelesen werden kann:

java.util.Properties

```
public void load(InputStream in)
    throws IOException
```

Hier muß ein [InputStream](#) übergeben werden (siehe ebenfalls [Kapitel 13](#)), der die Daten der Property-Liste zur Verfügung stellt. Wir wollen an dieser Stelle die Betrachtung der Klasse [Properties](#) abschließen. Ein ausführliches Beispiel zu ihrer Verwendung findet sich am Ende dieses Kapitels in [Abschnitt 12.8.1](#).

Im JDK 1.1 wurde statt der Methode [store](#) die Methode [save](#) zum Speichern einer Property-Liste verwendet. Diese hatte dieselbe Signatur, löste aber bei I/O-Problemen keine [IOException](#) aus. Im JDK 1.2 wurde [save](#) als [deprecated](#) deklariert und durch [store](#) ersetzt.

JDK1.1/1.2

12.4 Die Klasse BitSet

- 12.4 Die Klasse BitSet
 - 12.4.1 Elementweise Operationen
 - 12.4.2 Mengenorientierte Operationen

Die Klasse `BitSet` dient dazu, *Mengen ganzer Zahlen* zu repräsentieren. Sie erlaubt es, Zahlen einzufügen oder zu löschen und zu testen, ob bestimmte Werte in der Menge enthalten sind. Die Klasse bietet zudem die Möglichkeit, Teil- und Vereinigungsmengen zu bilden.

Ein `BitSet` erlaubt aber auch noch eine andere Interpretation. Sie kann nämlich auch als eine Menge von Bits, die entweder gesetzt oder nicht gesetzt sein können, angesehen werden. In diesem Fall entspricht das Einfügen eines Elements dem Setzen eines Bits, das Entfernen dem Löschen und die Vereinigungs- und Schnittmengenoperationen den logischen ODER- bzw. UND-Operationen.

Tip

Diese Analogie wird insbesondere dann deutlich, wenn man eine Menge von ganzen Zahlen in der Form repräsentiert, daß in einem booleschen Array das Element an Position *i* genau dann auf `true` gesetzt wird, wenn die Zahl *i* Element der repräsentierten Menge ist. Mit `BitSet` bietet Java nun eine Klasse, die sowohl als Liste von Bits als auch als Menge von Ganzzahlen angesehen werden kann.

12.4.1 Elementweise Operationen

Ein neues Objekt der Klasse `BitSet` kann mit dem parameterlosen Konstruktor angelegt werden. Die dadurch repräsentierte Menge ist zunächst leer.

`java.util.BitSet`

```
public BitSet()
```

Das Einfügen einer Zahl (bzw. das Setzen eines Bits) erfolgt mit Hilfe der Methode `set`. Das Entfernen einer Zahl (bzw. das Löschen eines Bits) erfolgt mit der Methode `clear`. Die Abfrage, ob eine Zahl in der Menge enthalten ist (bzw. die Abfrage des Zustands eines bestimmten Bits), erfolgt mit Hilfe der Methode `get`:

`java.util.BitSet`

```
public void set(int bit)
```

```
public void clear(int bit)
```

```
public boolean get(int bit)
```

12.4.2 Mengenorientierte Operationen

Die mengenorientierten Operationen benötigen zwei Mengen als Argumente, nämlich das aktuelle Objekt und eine weitere Menge, die als Parameter übergeben wird. Das Ergebnis der Operation wird dem aktuellen Objekt zugewiesen. Das Bilden der Vereinigungsmenge (bzw. die bitweise ODER-Operation) erfolgt durch Aufruf der Methode `or`, das Bilden der Durchschnittsmenge (bzw. die bitweise UND-Operation) mit Hilfe von `and`. Zusätzlich gibt es die Methode `xor`, die ein bitweises EXKLUSIV-ODER durchführt. Deren mengentheoretisches Äquivalent ist die Vereinigung von Schnittmenge und Schnittmenge der Umkehrmengen.

Seit dem JDK 1.2 gibt es des weiteren die Methode `andNot`, mit der alle Bits der Ursprungsmenge gelöscht werden, deren korrespondierendes Bit in der Argumentmenge gesetzt ist.

JDK1.1/1.2

`java.util.BitSet`

```
public void or(BitSet set)
```

```
public void and(BitSet set)
```

```
public void xor(BitSet set)
```

```
public void andNot(BitSet set)
```

Das folgende Programm verdeutlicht die Anwendung der Klasse [BitSet](#) am Beispiel der Konstruktion der Menge der Primzahlen kleiner gleich 20. Dabei werden besagte Primzahlen einfach als Menge X der natürlichen Zahlen bis 20 angesehen, bei der jedes Element keinen echten Teiler in X enthält:

[Listing1204.java](#)

```

001 /* Listing1204.java */
002
003 import java.util.*;
004
005 public class Listing1204
006 {
007     private final static int MAXNUM = 20;
008
009     public static void main(String[] args)
010     {
011         BitSet b;
012         boolean ok;
013
014         System.out.println("Die Primzahlen <= " + MAXNUM + ":");
015         b = new BitSet();
016         for (int i = 2; i <= MAXNUM; ++i) {
017             ok = true;
018             for (int j = 2; j < i; ++j) {
019                 if (b.get(j) && i % j == 0) {
020                     ok = false;
021                     break;
022                 }
023             }
024             if (ok) {
025                 b.set(i);
026             }
027         }
028         for (int i = 1; i <= MAXNUM; ++i) {
029             if (b.get(i)) {
030                 System.out.println("  " + i);
031             }
032         }
033     }
034 }

```

Listing 12.4: Konstruktion von Primzahlen mit der Klasse BitSet

Die Ausgabe des Programms ist:

```

Die Primzahlen <= 20:
2
3
5
7
11
13
17
19

```

12.5 Die Klasse StringTokenizer

- [12.5 Die Klasse StringTokenizer](#)
 - [12.5.1 Anlegen eines StringTokenizers](#)
 - [12.5.2 Zugriff auf Tokens](#)

Die Klasse [StringTokenizer](#) ist eine nützliche Hilfsklasse, mit der Strings in einzelne *Tokens* zerlegt werden können. Ein Token wird dabei als zusammenhängende Sequenz von Zeichen angesehen, die durch Trennzeichen oder durch das Ende der Zeichenkette begrenzt ist. Die Klasse [StringTokenizer](#) implementiert das Interface [Enumeration](#), so daß sie genauso benutzt werden kann wie die Iteratoren in den Klassen [Vector](#) oder [Hashtable](#).

12.5.1 Anlegen eines StringTokenizers

Die Klasse [StringTokenizer](#) besitzt drei Konstruktoren:

```

public StringTokenizer(String str)
public StringTokenizer(String str, String delim)
public StringTokenizer(String str,
                        String delim,
                        boolean returnTokens)

```

Die erste Variante übergibt den String `str`, der tokenisiert werden soll, und bereitet das Objekt für die nachfolgenden Zugriffe auf die einzelnen Tokens vor. Die zweite Variante erwartet zusätzlich die Übergabe einer Zeichenkette `delim`, die alle Zeichen enthält, die als Trennzeichen zwischen zwei aufeinanderfolgenden Tokens angesehen werden sollen. Wird der parameterlose Konstruktor verwendet, so werden die Zeichen `'\n'`, `'\r'`, `'\t'` und das Leerzeichen als Begrenzer verwendet.

Der dritte Konstruktor enthält einen weiteren Parameter, `returnTokens`. Wird er auf `true` gesetzt, geben die Funktionen zur Extraktion der Tokens auch die Trennzeichen zwischen zwei Tokens zurück. Falls der Parameter `false` ist, werden die Trennzeichen lediglich als Begrenzer angesehen, ohne an den Aufrufer zurückgegeben zu werden.

12.5.2 Zugriff auf Tokens

Wie beim Interface [Enumeration](#) üblich, erfolgt der Zugriff auf die Tokens mit Hilfe der Methoden [hasMoreElements](#) und [nextElement](#):

```

public boolean hasMoreElements()
public Object nextElement()
    throws NoSuchElementException

```

[hasMoreElements](#) gibt genau dann `true` zurück, wenn noch mindestens ein weiteres Token zur Verfügung steht, andernfalls wird `false` zurückgegeben. [nextElement](#) liefert das nächste Token, also den Teilstring von der aktuellen Position bis zum nächsten Trennzeichen bzw. bis zum Ende der Zeichenkette.

Obwohl sehr elegant, ist die Verwendung der Methoden des Interfaces [Enumeration](#) leider etwas umständlich, weil [nextElement](#) Instanzen der Klasse [Object](#) zurückgibt. Um die Anwendung des [StringTokenizer](#) bequemer zu machen, existiert mit den Methoden [hasMoreTokens](#) und [nextToken](#) dieselbe Funktionalität noch einmal in einer Form, bei der ein Rückgabewert vom Typ [String](#) geliefert wird:

Tip

```

public boolean hasMoreTokens()
public String nextToken()
    throws NoSuchElementException

```

Hier liefert [nextToken](#) kein [Object](#), sondern einen [String](#), und eine explizite Typkonvertierung erübrigt sich damit.

Das folgende Programm verdeutlicht die Anwendung eines [StringTokenizer](#), der eine einfache Zeichenkette in ihre Bestandteile zerlegt:

[Listing1205.java](#)

```
001 /* Listing1205.java */
002
003 import java.util.*;
004
005 public class Listing1205
006 {
007     public static void main(String[] args)
008     {
009         String s = "Dies ist nur ein Test";
010         StringTokenizer st = new StringTokenizer(s);
011         while (st.hasMoreTokens()) {
012             System.out.println(st.nextToken());
013         }
014     }
015 }
```

Listing 12.5: Anwendung der Klasse StringTokenizer

Die Programmausgabe ist:

Dies
ist
nur
ein
Test

12.6 Die Klasse Random

- [12.6 Die Klasse Random](#)
 - [12.6.1 Initialisierung des Zufallszahlengenerators](#)
 - [12.6.2 Erzeugen von Zufallszahlen](#)
 - [Gleichverteilte Zufallszahlen](#)
 - [Normalverteilte Zufallszahlen](#)

12.6.1 Initialisierung des Zufallszahlengenerators

Zufallszahlen werden beim Programmieren erstaunlich häufig gebraucht, beispielsweise für Simulationen, für Spiele oder zum Aufbau von Testszenarien. Java stellt zu diesem Zweck eine Klasse [Random](#) zur Verfügung, mit der Zufallszahlen erzeugt werden können. Sie basiert auf dem *Linear-Kongruenz-Algorithmus*, wie er beispielsweise in D.E. Knuth's »The Art Of Computer Programming, Vol. 2« beschrieben ist.

Die Klasse [Random](#) erlaubt das Instanzieren eines Zufallszahlengenerators mit oder ohne manuelles Setzen des `seed`-Wertes (also der internen Zufallsvariable):

```

public Random()
public Random(long seed)

```

Wird der `seed`-Parameter übergeben, initialisiert der Zufallszahlengenerator seinen internen Zähler mit diesem Wert, und die anschließend erzeugte Folge von Zufallszahlen ist *reproduzierbar*. Wird dagegen der parameterlose Konstruktor aufgerufen, initialisiert er den Zufallszahlengenerator auf der Basis der aktuellen Systemzeit. Die Zufallszahlenfolge ist in diesem Fall nicht reproduzierbar.

12.6.2 Erzeugen von Zufallszahlen

Gleichverteilte Zufallszahlen

Der Zufallszahlengenerator kann Zufallszahlen für die numerischen Grundtypen [int](#), [long](#), [float](#) oder [double](#) erzeugen. Durch Aufruf einer der Methoden [nextInt](#), [nextLong](#), [nextFloat](#) oder [nextDouble](#) wird die jeweils nächste Zufallszahl des entsprechenden Typs ermittelt und an den Aufrufer zurückgegeben:

```

public int nextInt()
public int nextInt(int n)
public long nextLong()
public float nextFloat()
public double nextDouble()

```

Anders als in vielen anderen Programmiersprachen liefern diese Methoden Ergebnisse aus dem gesamten Wertebereich des entsprechenden Grundtyps, also auch negative Zufallszahlen. Soll der Rückgabewert auf Werte größer gleich Null beschränkt werden, kann das Ergebnis mit [Math.abs](#) in einen positiven Wert umgewandelt werden.

Hinweis

Sollen nur ganzzahlige Zufallswerte unterhalb einer bestimmten Obergrenze *n* erzeugt werden, kann das Ergebnis zusätzlich modulo *n* genommen werden. Sollen nur positive Ganzzahlen (inclusive 0) erzeugt werden, kann die mit einem [int](#) parametrisierte Variante von [nextInt](#) verwendet werden. Zurückgegeben werden gleichverteilte Zufallszahlen zwischen 0 (inklusive) und *n* (exklusive).

Das folgende Beispiel zeigt die Verwendung von Zufallszahlen zur Generierung eines Lottotips. Hierbei werden durch Aufruf von [nextInt](#) ganzzahlige Zufallszahlen erzeugt und mit Hilfe der [abs](#)-Methode und des Modulo-Operators auf den Wertebereich von 1 bis 49 beschränkt. [BitSet](#) `b` verhindert, daß Zahlen doppelt in den Tip eingehen und wird zur sortierten Ausgabe des Ergebnisses verwendet:

Beispiel

[Listing1206.java](#)

```
001 /* Listing1206.java */
002
003 import java.util.*;
004
005 public class Listing1206
006 {
007     public static void main(String[] args)
008     {
009         BitSet b = new BitSet();
010         Random r = new Random();
011
012         System.out.print("Mein Lottotip: ");
013         int cnt = 0;
014         while (cnt < 6) {
015             int num = 1 + Math.abs(r.nextInt()) % 49;
016             if (!b.get(num)) {
017                 b.set(num);
018                 ++cnt;
019             }
020         }
021         for (int i = 1; i <= 49; ++i) {
022             if (b.get(i)) {
023                 System.out.print(i + " ");
024             }
025         }
026         System.out.println("");
027     }
028 }
```

Listing 12.6: Zufallszahlen zur Generierung eines Lottotips

Eine etwas einfacher zu handhabende Methode zur Erzeugung von Fließkomma-Zufallszahlen befindet sich in der Klasse [Math](#). Die Klassenmethode [Math.random](#) kann als Klassenmethode ohne Instanzierung eines Objekts verwendet werden und erzeugt gleichverteilte Zufallszahlen auf der Basis von [nextDouble](#).

Tip

Normalverteilte Zufallszahlen

Während diese Methoden *gleichverteilte* Zufallszahlen erzeugen, liefert die Methode [nextGaussian](#) die Zufallszahlen auf der Basis der *Normalverteilung* mit einem Mittelwert von 0.0 und der Standardabweichung von 1.0:

[java.util.Random](#)

```
public double nextGaussian()
```

Auch hier erzeugt ein Aufruf die jeweils nächste Zufallszahl auf der Basis des vorherigen Werts.

12.7 Die Klassen Date, Calendar und GregorianCalendar

- [12.7 Die Klassen Date, Calendar und GregorianCalendar](#)
 - [12.7.1 Konstruktoren](#)
 - [12.7.2 Abfragen und Setzen von Datumsbestandteilen](#)
 - [12.7.3 Vergleiche und Datums-/Zeitarithmetik](#)
 - [12.7.4 Umwandlung zwischen Date und Calendar](#)

Bis zum JDK 1.0.2 war die Klasse [Date](#) zur Darstellung und Manipulation von Datumswerten vorgesehen. Leider war sie nicht ganz fehlerfrei und aufgrund diverser Einschränkungen nur sehr bedingt zu gebrauchen. Ab der Version 1.1 des JDK gibt es neben [Date](#) die Klasse [Calendar](#) zur Verarbeitung von Datumswerten. Obgleich der Name den Anschein erwecken mag, daß ein Objekt vom Typ [Calendar](#) ein visueller Kalender ist, der als Komponente in GUI-basierten Programmen verwendet werden kann, ist dies nicht richtig. Statt dessen stellt [Calendar](#) eine Kapselung für [Date](#) dar, deren Aufgabe es ist, ein Datum-/Uhrzeitobjekt zu realisieren und Methoden zur Konstruktion, zum Verändern und Auslesen von Datums-/Uhrzeitbestandteilen und für die Zeit- und Datumsarithmetik zur Verfügung zu stellen.

Zunächst ist [Calendar](#) nichts weiter als eine abstrakte Basisklasse. Sie enthält die Methoden, mit denen auf die einzelnen Elemente konkreter Kalenderklassen zugegriffen werden kann bzw. mit denen diese Klassen manipuliert werden können. Als einzige konkrete Ableitung von [Calendar](#) steht die Klasse [GregorianCalendar](#) zur Verfügung, die ein Datum nach dem hierzulande verwendeten gregorianischen Kalender implementiert. Die Komplexität der Klassen [Calendar](#) und [GregorianCalendar](#) kommt vor allem durch folgende Ursachen zustande:

- Datumsarithmetik ist grundsätzlich eine nicht ganz triviale Angelegenheit
- Lokale Besonderheiten sind zu berücksichtigen (Ausgabeformatierung, an welchem Tag beginnt die Woche, 12/24-Stunden-Darstellung, wie viele Tage muß die erste Woche des Jahres mindestens haben usw.)
- Zeitzonen sind zu berücksichtigen
- Schaltjahre oder -sekunden und ähnliche Abweichungen zu astronomischen Kalendern müssen berücksichtigt werden
- Weltweit kommen viele unterschiedliche Kalender zum Einsatz

In der Tat ist die Implementierung der Kalenderklassen des JDK komplex und war lange Zeit fehlerbehaftet. Sie wird insbesondere dadurch erschwert, daß ein Datumsobjekt nicht nur aus den einzelnen Feldern für Tag, Monat, Jahr usw. besteht, sondern zusätzlich einen ganzzahligen Wert des Typs [long](#) enthält, der das Datum als *Anzahl der Millisekunden seit dem 1.1.1970* speichert. Beide Werte müssen auch nach Veränderungen einzelner Bestandteile des Datumsobjekts konsistent gehalten werden.

Auch die Bedienung der Kalenderklassen ist nicht so eingängig wie in vielen anderen Programmiersprachen. Hinderlich ist dabei oft die Tatsache, daß neben Datum und Uhrzeit grundsätzlich auch die Zeitzone mit betrachtet wird. Wir wollen in diesem Abschnitt einen pragmatischen Ansatz wählen und nur die wesentlichen Eigenschaften der beiden Klassen vorstellen. Fortgeschrittenere Themen wie Zeitzonekalkulation oder Lokalisierung werden außen vor bleiben.

12.7.1 Konstruktoren

Da die Klasse [Calendar](#) abstrakt ist, müssen konkrete Datumsobjekte aus der Klasse [GregorianCalendar](#) erzeugt werden. Dazu stehen folgende Konstruktoren zur Verfügung:

[java.util.GregorianCalendar](#)

```

public GregorianCalendar()

public GregorianCalendar(int year, int month, int date)

public GregorianCalendar(
    int year, int month, int date,
    int hour, int minute
)

public GregorianCalendar(
    int year, int month, int date,
    int hour, int minute, int second
)

```

Der parameterlose Konstruktor initialisiert das Datumsobjekt mit dem aktuellen Datum und der aktuellen Uhrzeit. Die übrigen Konstruktoren weisen die als Parameter übergebenen Werte zu. Neben den hier vorgestellten Konstruktoren gibt es noch weitere, die es erlauben, die Zeitzone und

Lokalisierung zu verändern. Standardmäßig werden die lokale Zeitzone und die aktuelle Lokalisierung verwendet.

12.7.2 Abfragen und Setzen von Datumsbestandteilen

Das Abfragen und Setzen von Datumsbestandteilen erfolgt mit den Methoden `set` und `get`:

```
public final int get(int field)

public final void set(int field, int value)
```

`get` und `set` erwarten als erstes Argument einen Feldbezeichner, der angibt, auf welches der diversen Datums-/Zeitfelder des Objektes zugegriffen werden soll. Als Rückgabewert liefert `get` den Inhalt des angegebenen Feldes; `set` schreibt den als zweiten Parameter `value` übergebenen Wert in das Feld hinein. [Tabelle 12.1](#) gibt eine Übersicht der in `GregorianCalendar` vorgesehenen Feldbezeichner und ihrer Wertegrenzen:

Feldbezeichner	Minimalwert	Maximalwert
<code>Calendar.ERA</code>	0	1
<code>Calendar.YEAR</code>	1	5,000,000
<code>Calendar.MONTH</code>	0	11
<code>Calendar.WEEK_OF_YEAR</code>	1	54
<code>Calendar.WEEK_OF_MONTH</code>	1	6
<code>Calendar.DAY_OF_MONTH</code>	1	31
<code>Calendar.DAY_OF_YEAR</code>	1	366
<code>Calendar.DAY_OF_WEEK</code>	1	7
<code>Calendar.DAY_OF_WEEK_IN_MONTH</code>	-1	6
<code>Calendar.AM_PM</code>	0	1
<code>Calendar.HOUR</code>	0	12
<code>Calendar.HOUR_OF_DAY</code>	0	23
<code>Calendar.MINUTE</code>	0	59
<code>Calendar.SECOND</code>	0	59
<code>Calendar.MILLISECOND</code>	0	999
<code>Calendar.ZONE_OFFSET</code>	-12*60*60*1000	12*60*60*1000
<code>Calendar.DST_OFFSET</code>	0	1*60*60*1000

Tabelle 12.1: Feldbezeichner der Klasse `GregorianCalendar`

Hierbei gibt es einige Besonderheiten zu beachten. So wird beispielsweise der Monat nicht von 1 bis 12 gemessen, sondern von 0 bis 11. Das Feld `ERA` gibt an, ob das Datum vor Christi Geburt oder danach liegt. `DAY_OF_WEEK` geht von 1 = Sonntag bis 7 = Samstag, das nachfolgende Beispiel zeigt die Verwendung von symbolischen Konstanten. `ZONE_OFFSET` und `DST_OFFSET` sind die Zeitzone- und Sommerzeitabweichungen, die in Millisekunden gemessen werden.

Hinweis

Wir wollen uns die Verwendung der verschiedenen Felder an einem einfachen Beispiel ansehen. Das Programm zeigt auch die Verwendung einiger symbolischer Konstanten zur Darstellung von Wochentagen und der Ära (vor/nach Christi Geburt):

Beispiel

```
001 /* Listing1207.java */
002
003 import java.util.*;
004
005 public class Listing1207
006 {
007     public static void main(String[] args)
008     {
009         //Zuerst Ausgabe des aktuellen Datums
010         GregorianCalendar cal = new GregorianCalendar();
011         cal.setTimeZone(TimeZone.getTimeZone("ECT"));
012         printCalendarInfo(cal);
013     }
```

[Listing1207.java](#)


```

014     System.out.println("---");
015
016     //Nun Ausgabe der Informationen zum 22.6.1910 (dem
017     //Geburtstag von Konrad Zuse).
018     cal.set(Calendar.DATE, 22);
019     cal.set(Calendar.MONTH, 6 - 1);
020     cal.set(Calendar.YEAR, 1910);
021     printCalendarInfo(cal);
022
023     //cal.setTime(cal.getTime());
024 }
025
026 public static void printCalendarInfo(GregorianCalendar cal)
027 {
028     int value;
029
030     //Aera
031     value = cal.get(Calendar.ERA);
032     if (value == cal.BC) {
033         System.out.println("Aera.....: vor Christi Geburt");
034     } else if (value == cal.AD) {
035         System.out.println("Aera.....: Anno Domini");
036     } else {
037         System.out.println("Aera.....: unbekannt");
038     }
039     //Datum
040     System.out.println(
041         "Datum.....: " +
042         cal.get(Calendar.DATE) + "." +
043         (cal.get(Calendar.MONTH)+1) + "." +
044         cal.get(Calendar.YEAR)
045     );
046     //Zeit
047     System.out.println(
048         "Zeit.....: " +
049         cal.get(Calendar.HOUR_OF_DAY) + ":" +
050         cal.get(Calendar.MINUTE) + ":" +
051         cal.get(Calendar.SECOND) + " (" +
052         cal.get(Calendar.MILLISECOND) + " ms)"
053     );
054     //Zeit, amerikanisch
055     System.out.print(
056         "Am.Zeit....: " +
057         cal.get(Calendar.HOUR) + ":" +
058         cal.get(Calendar.MINUTE) + ":" +
059         cal.get(Calendar.SECOND)
060     );
061     value = cal.get(Calendar.AM_PM);
062     if (value == cal.AM) {
063         System.out.println(" AM");
064     } else if (value == cal.PM) {
065         System.out.println(" PM");
066     }
067     //Woche
068     System.out.println(
069         "Woche.....: " +
070         cal.get(Calendar.WEEK_OF_YEAR) + ". im Jahr"
071     );
072     System.out.println(
073         "          " +
074         cal.get(Calendar.WEEK_OF_MONTH) + ". im Monat"
075     );
076     //Tag
077     System.out.println(
078         "Tag.....: " +

```

```

079     cal.get(Calendar.DAY_OF_YEAR) + ". im Jahr"
080 );
081 System.out.println(
082     "          " +
083     cal.get(Calendar.DAY_OF_MONTH) + ". im Monat"
084 );
085 System.out.println(
086     "          " +
087     cal.get(Calendar.DAY_OF_WEEK_IN_MONTH) + ". in der Woche"
088 );
089 //Wochentag
090 value = cal.get(Calendar.DAY_OF_WEEK);
091 if (value == cal.SUNDAY) {
092     System.out.println("Wochentag..: Sonntag");
093 } else if (value == cal.MONDAY) {
094     System.out.println("Wochentag..: Montag");
095 } else if (value == cal.TUESDAY) {
096     System.out.println("Wochentag..: Dienstag");
097 } else if (value == cal.WEDNESDAY) {
098     System.out.println("Wochentag..: Mittwoch");
099 } else if (value == cal.THURSDAY) {
100     System.out.println("Wochentag..: Donnerstag");
101 } else if (value == cal.FRIDAY) {
102     System.out.println("Wochentag..: Freitag");
103 } else if (value == cal.SATURDAY) {
104     System.out.println("Wochentag..: Samstag");
105 } else {
106     System.out.println("Wochentag..: unbekannt");
107 }
108 //Zeitzone
109 System.out.println(
110     "Zeitzone...: " +
111     cal.get(Calendar.ZONE_OFFSET)/3600000 +
112     " Stunden"
113 );
114 System.out.println(
115     "Sommerzeit.: " +
116     cal.get(Calendar.DST_OFFSET)/3600000 +
117     " Stunden"
118 );
119 }
120 }

```

Listing 12.7: Die Felder der Klasse Calendar

Das Programm erzeugt zunächst ein [GregorianCalendar](#)-Objekt für das aktuelle Tagesdatum und gibt die internen Feldwerte aus. Anschließend ändert es per Aufruf von [set](#) das Datum in den 22.6.1910 ab und gibt die Felder erneut aus. Die Ausgabe des Programms lautet:

```

Aera.....: Anno Domini
Datum.....: 2.1.1998
Zeit.....: 18:12:53 (+560 ms)
Am.Zeit....: 6:12:53 PM
Woche.....: 53. im Jahr
            0. im Monat
Tag.....: 2. im Jahr
            2. im Monat
            1. in der Woche
Wochentag..: Samstag
Zeitzone...: 1 Stunden
Sommerzeit.: 0 Stunden
---
Aera.....: Anno Domini
Datum.....: 22.6.1910
Zeit.....: 12:14:41 (+880 ms)
Am.Zeit....: 0:14:41 PM
Woche.....: 25. im Jahr

```

```

        4. im Monat
Tag.....: 173. im Jahr
        22. im Monat
        4. in der Woche
Wochentag.: Mittwoch
Zeitzone..: 1 Stunden
Sommerzeit.: 1 Stunden

```

Wie man sieht, werden sowohl Datums- als auch Zeitwerte korrekt ausgegeben. Damit dieses und vergleichbare Programme auch in den JDK-1.1-Versionen korrekt laufen, sind einige Besonderheiten zu beachten:

JDK1.1/1.2

- Wird das Datumsobjekt ohne explizite Zeitzonesangabe konstruiert, so verwendet der Konstruktor die von der Methode [TimeZone.getDefault](#) gelieferte Zeitzone, die ihrerseits aus dem System-Property [user.timezone](#) generiert wird. Unter dem JDK 1.1 wurde dieses Property aber nicht immer gefüllt und es war erforderlich, die Zeitzone per Hand zu setzen. Im JDK 1.2 könnten wir dagegen auf den Aufruf von [setTimeZone](#) in [Zeile 011](#) verzichten.
- Das zweite Problem rührt aus der oben erwähnten Schwierigkeit, den internen Zeitwert und die Feldwerte korrekt zu synchronisieren. Nach der Zuweisung eines neuen Datums im Beispielprogramm werden zwar die Felder für Tag, Monat und Jahr korrekt gesetzt, die übrigen aber leider nicht. Nach dieser Zuweisung wäre also die Ausgabe des Wochentags fehlerhaft gewesen. Als Workaround könnte das Beispielprogramm nach dem Aufruf der [set](#)-Methoden einen Aufruf von [setTime\(getTime\(\)\)](#) verwenden, der interne Uhrzeit und Feldwerte abgleicht. Wir haben das im Listing in der [Zeile 023](#) hinter dem Kommentar angedeutet. Wie gesagt, auch das scheint im JDK 1.2 nicht mehr nötig zu sein.

12.7.3 Vergleiche und Datums-/Zeitarithmetik

Die Methoden [equals](#), [before](#) und [after](#) erlauben es, zwei Datumswerte auf ihre relative zeitliche Lage zueinander zu vergleichen:

[java.util.Calendar](#)

```
public boolean equals(Object obj)
```

```
public boolean before(Object obj)
```

```
public boolean after(Object obj)
```

Mit Hilfe der Methode [add](#) kann zu einem beliebigen Feld eines [Calendar](#)- oder [GregorianCalendar](#)-Objekts ein beliebiger positiver oder negativer Wert hinzugezählt werden:

[java.util.Calendar](#)

```
public abstract void add(int field, int amount)
```

Dabei ist es auch erlaubt, daß die Summe den für dieses Feld erlaubten Grenzwert über- bzw. unterschreitet. In diesem Fall wird der nächsthöherwertige Datums- bzw. Zeitbestandteil entsprechend angepaßt.

Das folgende Programm konstruiert zunächst ein Datum für den 30.10.1908 und gibt es aus. Anschließend wird zunächst der Tag, dann der Monat und schließlich das Jahr je zweimal um 1 erhöht. Nach erfolgter Ausgabe wird die Änderung schrittweise wieder rückgängig gemacht und der ursprüngliche Wert wieder erzeugt:

Beispiel

[Listing1208.java](#)

```

001 /* Listing1208.java */
002
003 import java.util.*;
004
005 public class Listing1208
006 {
007     public static void main(String[] args)
008     {
009         GregorianCalendar cal = new GregorianCalendar();
010         cal.set(Calendar.DATE, 30);
011         cal.set(Calendar.MONTH, 10 - 1);
012         cal.set(Calendar.YEAR, 1908);
013         showDate(cal);
014         addOne(cal, Calendar.DATE);
015         addOne(cal, Calendar.DATE);

```

```

016     addOne(cal, Calendar.MONTH);
017     addOne(cal, Calendar.MONTH);
018     addOne(cal, Calendar.YEAR);
019     addOne(cal, Calendar.YEAR);
020
021     cal.add(Calendar.DATE, -2);
022     cal.add(Calendar.MONTH, -2);
023     cal.add(Calendar.YEAR, -2);
024     showDate(cal);
025 }
026
027 public static void addOne(Calendar cal, int field)
028 {
029     cal.add(field,1);
030     showDate(cal);
031 }
032
033 public static void showDate(Calendar cal)
034 {
035     String ret = "";
036     int value = cal.get(Calendar.DAY_OF_WEEK);
037
038     switch (value) {
039     case cal.SUNDAY:
040         ret += "Sonntag";
041         break;
042     case cal.MONDAY:
043         ret += "Montag";
044         break;
045     case cal.TUESDAY:
046         ret += "Dienstag";
047         break;
048     case cal.WEDNESDAY:
049         ret += "Mittwoch";
050         break;
051     case cal.THURSDAY:
052         ret += "Donnerstag";
053         break;
054     case cal.FRIDAY:
055         ret += "Freitag";
056         break;
057     case cal.SATURDAY:
058         ret += "Samstag";
059         break;
060     }
061     ret += ", den ";
062     ret += cal.get(Calendar.DATE) + ".";
063     ret += (cal.get(Calendar.MONTH)+1) + ".";
064     ret += cal.get(Calendar.YEAR);
065     System.out.println(ret);
066 }
067 }

```

Listing 12.8: Datumsarithmetik

Die Ausgabe des Programms lautet:

```

Freitag, den 30.10.1908
Samstag, den 31.10.1908
Sonntag, den 1.11.1908
Dienstag, den 1.12.1908
Freitag, den 1.1.1909
Samstag, den 1.1.1910
Sonntag, den 1.1.1911
Freitag, den 30.10.1908

```

12.7.4 Umwandlung zwischen Date und Calendar

In der Praxis ist es mitunter erforderlich, zwischen den beiden konkurrierenden Zeitdarstellungen der Klassen [Date](#) und [Calendar](#) hin- und herzuschalten. So ist beispielsweise der in JDBC (siehe [Kapitel 30](#)) häufig verwendete SQL-Datentyp [java.sql.Date](#) aus [java.util.Date](#) abgeleitet und repräsentiert ein Datum als Anzahl der Millisekunden seit dem 1.1.1970. Mit Hilfe der Methoden [setTime](#) und [getTime](#) können beide Darstellungen ineinander überführt werden:

java.util.Calendar

```
public final Date getTime()  
  
public final void setTime(Date date)
```

Ein Aufruf von [getTime](#) liefert das Datum als Objekt des Typs [Date](#). Soll das Datum aus einem vorhandenen [Date](#)-Objekt in ein [Calendar](#)-Objekt übertragen werden, kann dazu die Methode [setTime](#) aufgerufen werden. Die Klasse [Date](#) kann weiterhin dazu verwendet werden, auf die Anzahl der Millisekunden seit dem 1.1.1970 zuzugreifen:

java.util.Date

```
public Date(long date)  
  
public long getTime()
```

Der Konstruktor erzeugt aus dem [long](#) ein [Date](#)-Objekt, und die Methode [getTime](#) kann dazu verwendet werden, zu einem gegebenen [Date](#)-Objekt die Anzahl der Millisekunden seit dem 1.1.1970 zu ermitteln.

12.8 Die Klasse System

- 12.8 Die Klasse System
 - 12.8.1 System-Properties
 - 12.8.2 in, err und out
 - 12.8.3 exit
 - 12.8.4 gc
 - 12.8.5 currentTimeMillis
 - 12.8.6 arraycopy

Neben den bisher vorgestellten Datenstrukturen des Pakets `java.util` gibt es in `java.lang` eine Klasse `System`, die ebenfalls eine Reihe nützlicher Hilfsmittel zur Verfügung stellt. Die wichtigsten von ihnen sollen in den folgenden Abschnitten besprochen werden.

12.8.1 System-Properties

In Java gibt es keine Möglichkeit, direkt auf die *Umgebungsvariablen* eines Programms zuzugreifen. Ein solcher Zugriff wurde von den Java-Designern als nichtportabel angesehen und statt dessen durch das Konzept der *Properties* ersetzt. Properties sind Listen von *Eigenschaften*, die dem Programm vom Java-Laufzeitsystem zur Verfügung gestellt werden. Jede Eigenschaft besitzt einen Namen, unter dem auf sie zugegriffen werden kann. Das Java-Laufzeitsystem stellt standardmäßig die folgenden Properties zur Verfügung:

Property	Bedeutung
<code>java.version</code>	Java-Versionsnummer
<code>java.vendor</code>	Herstellerspezifische Zeichenkette
<code>java.vendor.url</code>	URL (also ein Internet-Link) zum Hersteller
<code>java.home</code>	Installationsverzeichnis
<code>java.class.version</code>	Versionsnummer der Java-Klassenbibliothek
<code>java.class.path</code>	Aktueller Klassenpfad
<code>os.name</code>	Name des Betriebssystems
<code>os.arch</code>	Betriebssystem-Architektur
<code>os.version</code>	Versionsnummer des Betriebssystems
<code>file.separator</code>	Trennzeichen für die Bestandteile eines Pfadnamens
<code>path.separator</code>	Trennzeichen für die Laufwerksangabe eines Pfadnamens
<code>line.separator</code>	Zeichenkette für Zeilenschaltung
<code>user.name</code>	Name des angemeldeten Benutzers
<code>user.home</code>	Home-Verzeichnis
<code>user.dir</code>	Aktuelles Arbeitsverzeichnis
<code>java.vm.specification.version</code>	Version der VM-Spezifikation
<code>java.vm.specification.vendor</code>	Hersteller der VM-Spezifikation
<code>java.vm.specification.name</code>	Bezeichnung der VM-Spezifikation
<code>java.vm.version</code>	VM-Version
<code>java.vm.vendor</code>	Hersteller der VM
<code>java.vm.name</code>	Name der VM-Implementierung
<code>java.specification.version</code>	Version der Spezifikation der Laufzeitumgebung
<code>java.specification.vendor</code>	Hersteller der Spezifikation der Laufzeitumgebung
<code>java.specification.name</code>	Bezeichnung der Spezifikation der Laufzeitumgebung

Tabelle 12.2: Standard-Properties

Für den Zugriff auf diese Eigenschaften steht die Klasse [Properties](#) aus dem Paket [java.util](#) zur Verfügung. Sie bietet die Möglichkeit, Property-Listen zu erzeugen, mit Werten zu füllen und vorhandene Werte auszulesen. Die Klasse [Properties](#) ist eine Ableitung der Klasse [Hashtable](#) und stellt damit eine Tabelle von Schlüssel-/Wertepaaren dar.

Für den Zugriff auf einzelne Properties reicht meist die einfach zu bedienende Klassenmethode [getProperty](#) der Klasse [System](#) in [java.lang](#) aus:

```
java.lang.System
public static String getProperty(String key)
public static String getProperty(String key, String default)
```

Die erste Variante liefert die Eigenschaft mit dem Namen [key](#) in Form einer Zeichenkette. Falls keine Eigenschaft mit diesem Namen gefunden wurde, wird [null](#) zurückgegeben. Die zweite Variante erlaubt die Übergabe eines Standardwertes. Der Unterschied zur ersten Variante besteht darin, daß nicht [null](#), sondern der Standardwert zurückgegeben wird, wenn die gesuchte Eigenschaft nicht gefunden wurde.

Die Methode [getProperties](#) liefert das komplette [Properties](#)-Objekt mit den System-Properties:

```
java.lang.System
public static Properties getProperties()
```

Das folgende Programm gibt eine Liste aller System-Properties auf dem Bildschirm aus. Es verwendet dazu zunächst die Methode [getProperties](#), um das System-[Properties](#)-Objekt zu beschaffen. Anschließend erzeugt es durch Aufruf von [propertyNames](#) einen Enumerator, mit dem alle Schlüsselwerte durchlaufen werden können und mit dem durch Aufruf von [getProperty](#) der zugehörige Wert ermittelt werden kann. Auf diese Weise listet das Programm alle verfügbaren System-Properties auf (das sind in der Regel sehr viel mehr als die plattformübergreifend spezifizierten):

Beispiel

```
Listing1209.java
001 /* Listing1209.java */
002
003 import java.util.*;
004
005 public class Listing1209
006 {
007     public static void main(String[] args)
008     {
009         Properties sysprops = System.getProperties();
010         Enumeration propnames = sysprops.propertyNames();
011         while (propnames.hasMoreElements()) {
012             String propname = (String)propnames.nextElement();
013             System.out.println(
014                 propname + "=" + System.getProperty(propname)
015             );
016         }
017     }
018 }
```

Listing 12.9: Ausgeben der System-Properties

12.8.2 in, err und out

In den vorangegangenen Kapiteln wurde schon häufig der Aufruf [System.out.println](#) verwendet, um Daten auf die Standardausgabe bzw. in ein Debug-Fenster auszugeben. Diese Anweisung ruft die Methode [println](#) des Objekts [out](#) der Klasse [System](#) auf. Dabei ist [out](#) eine statische Variable vom Typ [PrintStream](#), die beim Starten des Programms so initialisiert wird, daß ihre Ausgabe auf die Standardausgabe geleitet wird.

Analog zu [out](#) gibt es die statischen Variablen [err](#) und [in](#). Dabei dient [err](#) zur Ausgabe von Fehlermeldungen, und [in](#) ist ein Standardeingabekanal, der dazu verwendet werden kann, Eingaben von der Tastatur zu lesen.

Mit Hilfe der Methoden [setIn](#), [setOut](#) und [setErr](#) ist es sogar möglich, die Standardein- und -ausgabe aus dem Programm heraus umzuleiten:

```
java.lang.System
public static void setIn(InputStream in)
public static void setOut(PrintStream out)
public static void setErr(PrintStream err)
```

Die Verwendung dieser Klassenvariablen ist im Grunde genommen nicht konform mit dem Dialogkonzept einer GUI-Anwendung. Ihr Einsatz kann aber immer dann sinnvoll sein, wenn Java-Programme geschrieben werden sollen, die keine ausgefeilte Oberfläche benötigen. In diesem Fall sind sie ein nützliches Hilfsmittel, um einfache Ein-/Ausgaben ohne großen Aufwand realisieren zu können. Bereits in [Kapitel 3](#) wurde gezeigt, wie man diese Routinen zur Ein- und Ausgabe verwenden kann. Weitere Informationen darüber sind in [Kapitel 13](#) zu finden, das sich mit der Dateiein- und -ausgabe beschäftigt.

Hinweis

12.8.3 exit

Auch Aufrufe der Methode [System.exit](#) sind uns schon begegnet:

[java.lang.System](#)

```
public static void exit(int status)
```

Mit [System.exit](#) wird das laufende Programm beendet. Der Aufrufparameter `status` dient als Fehleranzeige und wird als Exitcode an den Aufrufer des Programms zurückgegeben. Gemäß Konvention zeigt dabei ein Wert größer oder gleich 1 einen Fehler während der Programmausführung an, während 0 ein fehlerfreies Programmende signalisiert.

12.8.4 gc

[java.lang.System](#)

```
public static void gc()
```

Ein Aufruf der Methode [gc](#) führt einen expliziten Aufruf des Garbage Collectors durch. Dieser sucht dann nach freiem Speicher und gibt diesen an das Laufzeitsystem zurück. Normalerweise ist ein Aufruf dieser Methode nicht erforderlich, denn der Garbage Collector läuft ständig als niedrig priorisierter Thread im Hintergrund. Der Aufruf von [gc](#) ist immer dann sinnvoll, wenn eine explizite Kontrolle über den Zeitpunkt der Speicherfreigabe gewünscht ist.

12.8.5 currentTimeMillis

Die Methode [currentTimeMillis](#) liefert die Anzahl Millisekunden, die zum Zeitpunkt des Aufrufs seit Mitternacht des 1.1.1970 vergangen sind:

[java.lang.System](#)

```
public static long currentTimeMillis()
```

Ob dabei tatsächlich eine Auflösung von einer Millisekunde erreicht wird, ist von der konkreten Java-Implementierung abhängig. In PC-basierten Java-Systemen orientiert sie sich meist an der Auflösung des System-Timers. Dieser wird 18,2mal pro Sekunde aufgerufen, so daß die Auflösung damit bei etwa 55 ms. liegt.

Hinweis

Mit dem folgenden Beispielprogramm kann die Auflösung des System-Timers ermittelt werden:

Beispiel

[Listing1210.java](#)

```
001 /* Listing1210.java */
002
003 public class Listing1210
004 {
005     public static void main(String[] args)
006     {
007         long t1, t2;
008         int  actres, sumres = 0, i = 0;
009         while (true) {
010             ++i;
011             t1 = System.currentTimeMillis();
012             while (true) {
013                 t2 = System.currentTimeMillis();
014                 if (t2 != t1) {
015                     actres = (int)(t2 - t1);
016                     break;
017                 }
018             }
019             sumres += actres;
020             System.out.print("it="+i+", ");
021             System.out.print("actres="+actres+" msec., ");
022             System.out.print("avgres="+sumres/i+" msec.");
023             System.out.println("");
024         }
025     }
026 }
```



```

024         try {
025             Thread.sleep(500);
026         } catch (InterruptedException e) {
027             //nichts
028         }
029     }
030 }
031 }

```

Listing 12.10: Die Auflösung des System-Timers bestimmen

Das Programm bestimmt zunächst die aktuelle Systemzeit und merkt sich den Wert in der Variablen `t1`. Nun wird die Systemzeit in einer Schleife erneut so oft gemessen, bis sie sich geändert hat. Die Differenz zwischen beiden Werten wird als Auflösung des aktuellen Durchgangs angesehen und der Variablen `actres` zugewiesen.

Um eine größere Genauigkeit zu erzielen, führt das Programm die Bestimmung der Auflösung mit Hilfe der äußeren Schleife viele Male durch. Die dabei jeweils ermittelte Auflösung wird in der Variablen `sumres` addiert und durch Division durch die Anzahl der Schleifendurchläufe zur Ermittlung des gleitenden Durchschnitts verwendet. Das so errechnete Ergebnis pendelt sich tatsächlich bereits nach wenigen Durchläufen auf den vorausgesagten Wert von 55 ms. ein:

Hinweis

```

it=1, actres=60 msec., avgres=60 msec.
it=2, actres=50 msec., avgres=55 msec.
it=3, actres=50 msec., avgres=53 msec.
it=4, actres=50 msec., avgres=52 msec.
it=5, actres=50 msec., avgres=52 msec.
it=6, actres=50 msec., avgres=51 msec.
...
it=65, actres=50 msec., avgres=55 msec.
it=66, actres=50 msec., avgres=55 msec.
it=67, actres=50 msec., avgres=55 msec.
it=68, actres=60 msec., avgres=55 msec.
it=69, actres=60 msec., avgres=55 msec.
it=70, actres=60 msec., avgres=55 msec.

```

12.8.6 arraycopy

Als letzte Methode der Klasse `System` soll `arraycopy` vorgestellt werden:

[java.lang.System](#)

```

public static native void arraycopy(
    Object src, int src_position,
    Object dst, int dst_position,
    int length
)

```

`arraycopy` kann dazu verwendet werden, Arrays oder Teile davon zu kopieren. Dabei können die Elemente sowohl innerhalb desselben Arrays als auch in ein anderes Array kopiert werden. Falls innerhalb desselben Arrays kopiert wird, dürfen sich Quell- und Zielbereich auch überlappen. Die Methode arbeitet sowohl mit elementaren als auch mit Objekttypen, Ziel- und Quellarray müssen lediglich zuweisungskompatibel sein. Da die Methode in C bzw. Assembler implementiert ist, arbeitet sie recht performant.

Als erste Argumente werden das Quellarray `src` und die Startposition `src_position` angegeben. Anschließend folgen das Zielarray `dst` und die Zielposition `dst_position`. Als letztes Argument wird die Länge `length` des zu kopierenden Bereichs angegeben. Falls die Argumente auf Elemente zeigen, die außerhalb des Arrays liegen, wird eine Ausnahme des Typs `ArrayIndexOutOfBoundsException` ausgelöst. Falls ein Element aufgrund eines Typfehlers nicht gespeichert werden kann, gibt es eine Ausnahme des Typs `ArrayStoreException`.

Beispiel:

[Listing1211.java](#)

```
001 /* Listing1211.java */
002
003 public class Listing1211
004 {
005     public static void main(String[] args)
006     {
007         int ar[] = {0,0,0,0,0,0,0,0,0,0};
008
009         for (int i = 0; i < 10; ++i) {
010             System.arraycopy(ar,0,ar,1,9);
011             ar[0] = i;
012         }
013         System.out.print("ar = ");
014         for (int i = 0; i < 10; ++i) {
015             System.out.print(ar[i] + " ");
016         }
017         System.out.println("");
018     }
019 }
```

Listing 12.11: Verwendung von System.arraycopy

Das Programm füllt ein 10-elementiges Array von Ganzzahlen, das zunächst nur Nullen enthält, mit den Zahlen 0 bis 9. Dabei wird jeweils durch Kopieren der ersten neun Elemente an die zweite Position des Arrays an der ersten Position Platz gemacht, um dort den Inhalt des fortlaufenden Schleifenzählers abzulegen. Nach 10 Durchläufen stehen somit die Zahlen 0 bis 9 verkehrt herum im Array. Die Ausgabe des Programms ist:

ar = 9 8 7 6 5 4 3 2 1 0

12.9 Die Klasse Arrays

- 12.9 Die Klasse Arrays

Seit dem JDK 1.2 gibt es die Klasse [Arrays](#) im Paket [java.util](#), die einige nützliche Methoden zum Zugriff auf Arrays zur Verfügung stellt. Sie kann beispielsweise ein Array mit vorgegebenen Werten füllen, eine binäre Suche durchführen, das Array sortieren oder zwei Arrays miteinander vergleichen.

JDK1.1/1.2

Die wichtigsten Methoden sind [fill](#), [binarySearch](#), [sort](#) und [equals](#):

[java.util.Arrays](#)

```
public static void fill(int[] a, int val)
```

```
public static int binarySearch(int[] a, int key)
```

```
public static void sort(int[] a)
```

```
public static boolean equals(int[] a, int[] a2)
```

Alle Methoden stehen auch in vergleichbaren Versionen für die anderen primitiven Typen zur Verfügung. Wir wollen uns die Verwendung der Klasse [Arrays](#) am Beispiel eines einfachen Programms ansehen, das ein Array von Ganzzahlen sortiert:

[Listing1212.java](#)

```
001 /* Listing1212.java */
002
003 import java.util.*;
004
005 public class Listing1212
006 {
007     public static void main(String[] args)
008     {
009         final int SIZE = 20;
010         int values[] = new int[SIZE];
011         Random rand = new Random();
012         //Erzeugen uns Ausgeben des unsortierten Arrays
013         for (int i = 0; i < SIZE; ++i) {
014             values[i] = rand.nextInt(10 * SIZE);
015         }
016         for (int i = 0; i < SIZE; ++i) {
017             System.out.println(values[i]);
018         }
019         //Sortieren des Arrays
020         Arrays.sort(values);
021         //Ausgeben der Daten
022         System.out.println("---");
023         for (int i = 0; i < SIZE; ++i) {
024             System.out.println(values[i]);
025         }
026     }
027 }
```

Listing 12.12: Sortieren eines Arrays

Die Sortiermethoden der Klasse [Arrays](#) können Arrays mit primitiven Datentypen nur in *aufsteigender* Reihenfolge sortieren. Zusätzlich gibt es eine Variante, die ein Array von Elementen des Typ [Object](#) sortiert und dazu als zweites Argument ein [Comparator](#)-Objekt erwartet. Auf die Bedeutung und Anwendung dieser Klasse und ihre Anwendung im Rahmen der Collection-Klassen des JDK 1.2 werden wir in [Abschnitt 27.7](#) zurückkommen.

Hinweis

12.10 Zusammenfassung

- [12.10 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Klasse [Vector](#) zum Erzeugen von dynamischen Listen
- Verwendung des [Enumeration](#)-Interfaces
- Die Klasse [Stack](#)
- Die Klassen [Hashtable](#) und [Properties](#) zum Erzeugen von Dictionaries
- Die Klasse [BitSet](#) zur Darstellung von Mengen ganzer Zahlen
- Die Klasse [StringTokenizer](#)
- Erzeugen gleich- und normalverteilter Zufallszahlen mit der Klasse [Random](#)
- Datumsbearbeitung mit den Klassen [Date](#), [Calendar](#) und [GregorianCalendar](#)
- Die Klasse [System](#)
- System-Properties
- Zugriff auf die Standardein- und -ausgabe mit [in](#), [out](#) und [err](#)
- Die Methoden [exit](#), [gc](#), [currentTimeMillis](#) und [arraycopy](#) der Klasse [System](#)
- Die Klasse [Arrays](#)

Kapitel 13

Datei-/-ausgabe

- [13 Datei-/-ausgabe](#)
 - [13.1 Allgemeine Konzepte](#)
 - [13.2 Ausgabe-Streams](#)
 - [13.2.1 Die abstrakte Klasse Writer](#)
 - [13.2.2 Auswahl des Ausgabegerätes](#)
 - [OutputStreamWriter und FileWriter](#)
 - [StringWriter und CharArrayWriter](#)
 - [13.2.3 Schachteln von Ausgabe-Streams](#)
 - [BufferedWriter](#)
 - [PrintWriter](#)
 - [FilterWriter](#)
 - [13.3 Eingabe-Streams](#)
 - [13.3.1 Die abstrakte Klasse Reader](#)
 - [13.3.2 Auswahl des Eingabegerätes](#)
 - [InputStreamReader und FileReader](#)
 - [StringReader und CharArrayReader](#)
 - [13.3.3 Schachteln von Eingabe-Streams](#)
 - [BufferedReader](#)
 - [LineNumberReader](#)
 - [FilterReader und PushbackReader](#)
 - [13.4 Random-Access-Dateien](#)
 - [13.4.1 Öffnen, Neuanlegen und Schließen](#)
 - [13.4.2 Positionierung des Dateizeigers](#)
 - [13.4.3 Lesezugriffe](#)
 - [13.4.4 Schreibzugriffe](#)
 - [13.5 Datei- und Verzeichnis-Handling](#)
 - [13.5.1 Konstruktion eines File-Objektes](#)
 - [13.5.2 Zugriff auf Teile des Pfadnamens](#)
 - [13.5.3 Informationen über die Datei](#)
 - [13.5.4 Zugriff auf Verzeichnisse](#)
 - [13.5.5 Anlegen von Dateien](#)
 - [Temporäre Dateien](#)
 - [Lockdateien](#)
 - [13.6 Zusammenfassung](#)

13.1 Allgemeine Konzepte

- 13.1 Allgemeine Konzepte

Als Sprache, die nicht nur das Erstellen vorwiegend grafikorientierter Web-Applets ermöglicht, sondern auch zur Entwicklung von eigenständigen Anwendungen eingesetzt werden soll, bietet Java eine umfangreiche Bibliothek zum sequentiellen und wahlfreien Zugriff auf Dateien und zur Verwaltung von Verzeichnissen. Während der wahlfreie Zugriff ähnlich wie in anderen Sprachen gelöst ist, wurden bei der sequentiellen Ein-/Ausgabe neue Wege beschritten. Die dafür verwendeten Klassen realisieren das aus anderen Sprachen bekannte Konzept der *Streams* mit Hilfe objektorientierter Techniken.

Ein *Stream* wird dabei zunächst als abstraktes Konstrukt eingeführt, dessen Fähigkeit darin besteht, Zeichen auf ein imaginäres Ausgabegerät zu schreiben oder von diesem zu lesen. Erst konkrete Unterklassen binden die Zugriffsroutinen an echte Ein- oder Ausgabegeräte, wie beispielsweise an Dateien, Strings oder Kommunikationskanäle im Netzwerk.

Des weiteren bietet das Klassenkonzept von Java die Möglichkeit, Streams zu verketten oder zu schachteln. Die Verkettung von Streams ermöglicht es, mehrere Dateien zusammenzufassen und für den Aufrufer als einen einzigen Stream darzustellen. Das Schachteln von Streams erlaubt die Konstruktion von Filtern, die bei der Ein- oder Ausgabe bestimmte Zusatzfunktionen übernehmen, beispielsweise das Puffern von Zeichen, das Zählen von Zeilen oder die Interpretation binärer Daten.

Beide Konzepte sind mit normalen Sprachmitteln realisiert und können selbst erweitert werden. So ist es ohne weiteres möglich, eigene Filter zu schreiben, die den Ein- oder Ausgabestrom analysieren und anwendungsbezogene Funktionalitäten realisieren.

Alle Klassen zur Dateiein- und -ausgabe befinden sich im Paket `java.io`. Um sie zu verwenden, sollte daher folgende Anweisung an den Anfang eines Programms gestellt werden:

```
import java.io.*;
```

Bis zur Version 1.0 des JDK gab es nur *Byte-Streams* in Java. Wesentliches Merkmal eines Byte-Streams war es dabei, daß die Transporteinheit 8 Bit lange Bytes waren, wie sie in den meisten anderen Programmiersprachen verwendet werden. Während damit die Kompatibilität zu Textdateien, die mit konventionellen Programmiersprachen erstellt wurden oder von diesen gelesen werden sollten, gewährleistet war, gab es natürlich Reibungsverluste bei der Umwandlung zwischen Bytes und 16 Bit langen Unicode-Zeichen, wie sie innerhalb von Java benutzt werden. Zudem war die Abbildung zwischen Bytes und Characters eher unsystematisch gelöst und bot wenig Unterstützung für die Anpassung an unterschiedliche Zeichensätze und nationale Gegebenheiten.

All dies hat die JDK-Designer dazu bewogen, das Konzept der Streams in der Version 1.1 zu überdenken und die neue Gruppe der *Character-Streams* einzuführen. Die Character-Streams verwenden grundsätzlich 16 Bit lange Unicode-Zeichen und arbeiten daher viel besser mit den String- und Zeichentypen von Java zusammen. Um zu den 8-Bit-Zeichensätzen in externen Dateien kompatibel zu bleiben, wurden explizite Brückenklassen eingeführt, die Character-Streams in Byte-Streams überführen und umgekehrt. Diese bieten nun auch die Möglichkeit der Anpassung an spezielle Zeichensätze und lokale Besonderheiten.

Hinweis

Die Namensgebung für Byte-Streams in Java folgt dem Prinzip, eine Klasse für den lesenden Zugriff als *InputStream* und für den schreibenden Zugriff als *OutputStream* zu bezeichnen. Die Character-Streams dagegen werden als *Reader* bei lesenden und als *Writer* bei schreibenden Zugriffen bezeichnet. Wir werden uns bei der folgenden Darstellung der Streams im wesentlichen auf die Reader- und Writer-Klassen beschränken. Sie stellen das modernere Stream-Konzept dar und bügeln einige der Schwächen von Byte-Streams des JDK 1.0 aus. Wenn also zukünftig von Streams die Rede ist, sollen immer die Character-Streams des JDK 1.1 gemeint sein. Bei den an einigen Stellen trotzdem erforderlichen Zugriffen auf Byte-Streams werden wir dies explizit erwähnen.

13.2 Ausgabe-Streams

- 13.2 Ausgabe-Streams
 - 13.2.1 Die abstrakte Klasse `Writer`
 - 13.2.2 Auswahl des Ausgabegerätes
 - `OutputStreamWriter` und `FileWriter`
 - `StringWriter` und `CharArrayWriter`
 - 13.2.3 Schachteln von Ausgabe-Streams
 - `BufferedWriter`
 - `PrintWriter`
 - `FilterWriter`

13.2.1 Die abstrakte Klasse `Writer`

Basis aller sequentiellen Ausgaben ist die abstrakte Klasse `Writer` des Paktes `java.io`, die eine Schnittstelle für stream-basierte Ausgaben zur Verfügung stellt:

```

java.io.Writer
protected Writer()

public void close()
public void flush()

public void write(int c)
public void write(char cbuf[])
abstract public void write(char cbuf[], int off, int len)
public void write(String str)
public void write(String str, int off, int len)

```

Neben einem parameterlosen Konstruktor definiert sie die Methoden `close` und `flush` und eine Reihe überladener `write`-Methoden. Die Bedeutung des Konstruktors liegt darin, den Ausgabestrom zu öffnen und für einen nachfolgenden Aufruf von `write` vorzubereiten, bevor er schließlich mit `close` wieder geschlossen wird. In der Grundform erwartet `write` einen einzigen `int`-Parameter und schreibt diesen als Byte in den Ausgabestrom. Daneben gibt es weitere Varianten, die ein Array von Bytes oder ein `String`-Objekt als Parameter erwarten und dieses durch wiederholten Aufruf der primitiven `write`-Methode ausgeben.

Von abgeleiteten Klassen wird erwartet, daß sie mindestens die Methoden `flush` und `close` Hinweis sowie `write(char, int, int)` überlagern. Aus Gründen der Performance ist es aber oftmals sinnvoll, auch die übrigen `write`-Methoden zu überlagern.

13.2.2 Auswahl des Ausgabegerätes

Als abstrakte Basisklasse kann `Writer` nicht instanziiert werden. Statt dessen gibt es eine Reihe konkreter Klassen, die aus `Writer` abgeleitet wurden, und deren Aufgabe es ist, die Verbindung zu einem konkreten Ausgabegerät herzustellen oder Filterfunktionen zu übernehmen. [Tabelle 13.1](#) gibt eine Übersicht dieser abgeleiteten Klassen.

Klasse	Bedeutung
OutputStreamWriter	Abstrakte Basisklasse für alle <code>Writer</code> , die einen Character-Stream in einen Byte-Stream umwandeln
FileWriter	Konkrete Ableitung von OutputStreamWriter zur Ausgabe in eine Datei
FilterWriter	Abstrakte Basisklasse für die Konstruktion von Ausgabefiltern
PrintWriter	Ausgabe aller Basistypen im Textformat
BufferedWriter	Writer zur Ausgabepufferung
StringWriter	Writer zur Ausgabe in einen <code>String</code>
CharArrayWriter	Writer zur Ausgabe in ein Zeichen-Array

PipedWriter	Writer zur Ausgabe in einen PipedReader
-----------------------------	---

Tabelle 13.1: Aus *Writer* abgeleitete Klassen

In den folgenden Unterabschnitten werden die Klassen [OutputStreamWriter](#), [FileWriter](#), [StringWriter](#) und [CharArrayWriter](#) erläutert. Die Klassen [BufferedWriter](#), [PrintWriter](#) und [FilterWriter](#) sind Thema des nächsten Abschnitts. Die Klasse [PipedWriter](#) wird hier nicht weiter behandelt.

OutputStreamWriter und FileWriter

Die Klasse [OutputStreamWriter](#) ist die abstrakte Basisklasse für alle *Writer*, die eine Konvertierung zwischen Character- und Byte-Streams vornehmen. Sie enthält ein Objekt des Typs [CharToByteConverter](#) (aus dem undokumentierten Paket [sun.io](#)), das die Konvertierung der Ausgabezeichen bei allen schreibenden Zugriffen vornimmt. Als abstrakte Basisklasse ist [OutputStreamWriter](#) für uns allerdings nicht so interessant wie die daraus abgeleitete Klasse [FileWriter](#), die die Ausgabe in eine Datei ermöglicht. Sie implementiert die abstrakten Eigenschaften von [Writer](#) und bietet vier zusätzliche Konstruktoren, die es erlauben, eine Datei zu öffnen:

```

public FileWriter(String fileName)
    throws IOException

public FileWriter(String fileName, boolean append)
    throws IOException

public FileWriter(File file)
    throws IOException

public FileWriter(FileDescriptor fd)

```

Am einfachsten kann eine Datei geöffnet werden, indem der Dateiname als *String*-Parameter *fileName* übergeben wird. Falls *fileName* eine bereits vorhandene Datei bezeichnet, wird sie geöffnet und ihr bisheriger Inhalt gelöscht, andernfalls wird eine neue Datei mit diesem Namen angelegt. Sukzessive Aufrufe von [write](#) schreiben weitere Bytes in diese Datei. Wird zusätzlich der Parameter *append* mit dem Wert *true* an den Konstruktor übergeben, so werden die Ausgabezeichen an die Datei angehängt, falls sie bereits existiert.

Das folgende Programm erstellt eine Datei *hallo.txt* und schreibt die Zeile »Hallo JAVA« in die Datei:

Beispiel

[Listing1301.java](#)

```

001 /* Listing1301.java */
002
003 import java.io.*;
004
005 public class Listing1301
006 {
007     public static void main(String[] args)
008     {
009         String hello = "Hallo JAVA\r\n";
010         FileWriter f1;
011
012         try {
013             f1 = new FileWriter("hallo.txt");
014             f1.write(hello);
015             f1.close();
016         } catch (IOException e) {
017             System.out.println("Fehler beim Erstellen der Datei");
018         }
019     }
020 }

```

Listing 13.1: Erstellen einer Datei

Fast alle Methoden der Klassen [OutputStreamWriter](#) und [FileWriter](#) deklarieren die Ausnahme [IOException](#), die als allgemeine Fehleranzeige im Paket [java.io](#) verwendet wird. [IOException](#) kann von einer Vielzahl von Methoden ausgelöst werden und hat dabei teilweise sehr unterschiedliche Bedeutungen. So bedeutet sie beim Aufruf des Konstruktors, daß es nicht möglich war, die Datei anzulegen, was wiederum eine ganze Reihe von Gründen haben kann. Beim Aufruf von [write](#) signalisiert sie einen Schreibfehler, und bei [close](#) zeigt sie einen nicht näher spezifizierten I/O-Fehler an.

In unserem Beispiel wurden alle Ausnahmen dieses Typs von einer einzigen *try-catch*-Anweisung behandelt. Falls eine differenziertere Fehlererkennung nötig ist, macht es Sinn, für die unterschiedlichen I/O-Operationen verschiedene *try-catch*-Anweisungen vorzusehen.

Die anderen Konstruktoren von [FileWriter](#) erwarten ein [File](#)-Objekt bzw. ein Objekt vom Typ [FileDescriptor](#). Während wir auf die Klasse [FileDescriptor](#) nicht näher eingehen werden, ist ein [File](#) die Repräsentation einer Datei im Kontext ihres Verzeichnisses. Wir werden weiter unten in [Abschnitt 13.5](#) darauf zurückkommen.

Hinweis

StringWriter und CharArrayWriter

Die Klassen [StringWriter](#) und [CharArrayWriter](#) sind ebenfalls aus [Writer](#) abgeleitet. Im Gegensatz zu [FileWriter](#) schreiben sie ihre Ausgabe jedoch nicht in eine Datei, sondern in einen dynamisch wachsenden [StringBuffer](#) bzw. in ein Zeichenarray.

[StringWriter](#) besitzt zwei Konstruktoren:

[java.io.StringWriter](#)

```
public StringWriter()  
  
public StringWriter(int initialSize)
```

Der parameterlose Konstruktor legt einen [StringWriter](#) mit der Standardgröße eines [StringBuffer](#)-Objekts an, während der zweite Konstruktor es erlaubt, die initiale Größe selbst festzulegen. Wie schon erwähnt, wächst der interne Puffer automatisch, wenn fortgesetzte Aufrufe von [write](#) dies erforderlich machen. Die schreibenden Zugriffe auf den Puffer erfolgen mit den von [Writer](#) bekannten [write](#)-Methoden.

Für den Zugriff auf den Inhalt des Puffers stehen die Methoden [getBuffer](#) und [toString](#) zur Verfügung:

[java.io.StringWriter](#)

```
public StringBuffer getBuffer()  
  
public String toString()
```

Die Methode [getBuffer](#) liefert den internen [StringBuffer](#), während [toString](#) den Puffer in einen [String](#) kopiert und diesen an den Aufrufer liefert.

Die Klasse [CharArrayWriter](#) arbeitet ähnlich wie [StringWriter](#), schreibt die Zeichen allerdings in ein Character-Array. Analog zu [StringWriter](#) wächst dieses automatisch bei fortgesetzten Aufrufen von [write](#). Die Konstruktoren haben dieselbe Struktur wie bei [StringWriter](#), und auch die dort verfügbaren [write](#)-Methoden stehen allesamt hier zur Verfügung:

[java.io.CharArrayWriter](#)

```
public CharArrayWriter()  
  
public CharArrayWriter(int initialSize)
```

Auf den aktuellen Inhalt des Arrays kann mit Hilfe der Methoden [toString](#) und [toCharArray](#) zugegriffen werden:

[java.io.CharArrayWriter](#)

```
public String toString()  
  
public char[] toCharArray()
```

Zusätzlich stehen die Methoden [reset](#) und [size](#) zur Verfügung, mit denen der interne Puffer geleert bzw. die aktuelle Größe des Zeichen-Arrays ermittelt werden kann. Durch Aufruf von [writeTo](#) kann des weiteren der komplette Inhalt des Arrays an einen anderen [Writer](#) übergeben und so beispielsweise mit einem einzigen Funktionsaufruf in eine Datei geschrieben werden:

[java.io.CharArrayWriter](#)

```
public void reset()  
  
public int size()  
  
public void writeTo(Writer out)  
    throws IOException
```

13.2.3 Schachteln von Ausgabe-Streams

Wie eingangs erwähnt, ist es in Java möglich, Streams zu schachteln. Dazu gibt es die aus [Writer](#) abgeleiteten Klassen [BufferedWriter](#), [PrintWriter](#) und [FilterWriter](#), deren wesentlicher Unterschied zu [Writer](#) darin besteht, daß sie als Membervariable einen zusätzlichen [Writer](#) besitzen, der im Konstruktor übergeben wird. Wenn nun die [write](#)-Methode eines derart geschachtelten Writers aufgerufen wird, gibt sie die Daten nicht direkt an den internen [Writer](#) weiter, sondern führt zuvor erst die erforderlichen Filterfunktionen aus. Damit lassen sich beliebige Filterfunktionen transparent realisieren.

BufferedWriter

Diese Klasse hat die Aufgabe, Stream-Ausgaben zu puffern. Dazu enthält sie einen internen Puffer, in dem die Ausgaben von `write` zwischengespeichert werden. Erst wenn der Puffer voll ist oder die Methode `flush` aufgerufen wird, werden alle gepufferten Ausgaben in den echten Stream geschrieben. Das Puffern der Ausgabe ist immer dann nützlich, wenn die Ausgabe in eine Datei geschrieben werden soll. Durch die Verringerung der `write`-Aufrufe reduziert sich die Anzahl der Zugriffe auf das externe Gerät, und die Performance wird erhöht.

`BufferedWriter` besitzt zwei Konstruktoren:

java.io.BufferedWriter

```
public BufferedWriter(Writer out)

public BufferedWriter(Writer out, int size)
```

In beiden Fällen wird ein bereits existierender `Writer` übergeben, an den die gepufferten Ausgaben weitergereicht werden. Falls die Größe des Puffers nicht explizit angegeben wird, legt `BufferedWriter` einen Standardpuffer an, dessen Größe für die meisten Zwecke ausreichend ist. Die `write`-Methoden von `BufferedWriter` entsprechen denen der Klasse `Writer`.

Zusätzlich gibt es eine Methode `newLine`, mit der eine Zeilenschaltung in den Stream geschrieben werden kann. Diese wird dem System-Property `line.separator` entnommen und entspricht damit den lokalen Konventionen.

Das nachfolgende Beispiel demonstriert die gepufferte Ausgabe einer Reihe von Textzeilen in eine Datei `buffer.txt`:

Beispiel

[Listing1302.java](#)

```
001 /* Listing1302.java */
002
003 import java.io.*;
004
005 public class Listing1302
006 {
007     public static void main(String args[])
008     {
009         Writer f1;
010         BufferedWriter f2;
011         String s;
012
013         try {
014             f1 = new FileWriter("buffer.txt");
015             f2 = new BufferedWriter(f1);
016             for (int i = 1; i <= 10000; ++i) {
017                 s = "Dies ist die " + i + ". Zeile";
018                 f2.write(s);
019                 f2.newLine();
020             }
021             f2.close();
022             f1.close();
023         } catch (IOException e) {
024             System.out.println("Fehler beim Erstellen der Datei");
025         }
026     }
027 }
```

Listing 13.2: Gepufferte Ausgabe in eine Datei

Dieses Beispiel erzeugt zunächst einen neuen `Writer` `f1`, um die Datei `buffer.txt` anzulegen. Dieser wird anschließend als Parameter an den `BufferedWriter` `f2` übergeben, der dann für den Aufruf der Ausgaberroutinen verwendet wird.

Die etwas umständliche Verwendung zweier Stream-Variablen lässt sich vereinfachen, indem die Konstruktoren direkt beim Aufruf geschachtelt werden. Das ist die unter Java übliche Methode, die auch zukünftig bevorzugt verwendet werden soll:

Tip

```

001 /* Listing1303.java */
002
003 import java.io.*;
004
005 public class Listing1303
006 {
007     public static void main(String args[])
008     {
009         BufferedWriter f;
010         String s;
011
012         try {
013             f = new BufferedWriter(
014                 new FileWriter("buffer.txt"));
015             for (int i = 1; i <= 10000; ++i) {
016                 s = "Dies ist die " + i + ". Zeile";
017                 f.write(s);
018                 f.newLine();
019             }
020             f.close();
021         } catch (IOException e) {
022             System.out.println("Fehler beim Erstellen der Datei");
023         }
024     }
025 }

```

Listing 13.3: Schachteln von Writer-Konstruktoren

PrintWriter

Die stream-basierten Ausgaberroutinen in anderen Programmiersprachen bieten meistens die Möglichkeit, alle primitiven Datentypen in textueller Form auszugeben. Java realisiert dieses Konzept über die Klasse [PrintWriter](#), die Ausgabemethoden für alle primitiven Datentypen und für Objekttypen zur Verfügung stellt. [PrintWriter](#) besitzt folgende Konstruktoren:

[java.io.PrintWriter](#)

```

public PrintWriter(Writer out)

public PrintWriter(Writer out, boolean autoflush)

```

Der erste Konstruktor instanziert ein [PrintWriter](#)-Objekt durch Übergabe eines [Writer](#)-Objekts, auf das die Ausgabe umgeleitet werden soll. Beim zweiten Konstruktor gibt zusätzlich der Parameter [autoflush](#) an, ob nach der Ausgabe einer Zeilenschaltung automatisch die Methode [flush](#) aufgerufen werden soll.

Die Ausgabe von primitiven Datentypen wird durch eine Reihe überladener Methoden mit dem Namen [print](#) realisiert. Zusätzlich gibt es alle Methoden in einer Variante [println](#), bei der automatisch an das Ende der Ausgabe eine Zeilenschaltung angehängt wird. [println](#) existiert darüber hinaus parameterlos, um lediglich eine einzelne Zeilenschaltung auszugeben. Damit stehen folgende [print](#)-Methoden zur Verfügung (analog für [println](#)):

[java.io.PrintWriter](#)

```

public void print(boolean b)
public void print(char c)
public void print(char s[])
public void print(double d)
public void print(float f)
public void print(int i)
public void print(long l)
public void print(Object obj)
public void print(String s)

```

Das folgende Beispiel berechnet die Zahlenfolge $1 + 1/2 + 1/4 + \dots$ und gibt die Folge der Summanden und die aktuelle Summe unter Verwendung mehrerer unterschiedlicher [print](#)-Routinen in die Datei `zwei.txt` aus:

[Listing1304.java](#)

```

001 /* Listing1304.java */
002
003 import java.io.*;
004
005 public class Listing1304
006 {
007     public static void main(String args[])
008     {
009         PrintWriter f;
010         double sum = 0.0;
011         int nenner;
012
013         try {
014             f = new PrintWriter(
015                 new BufferedWriter(
016                     new FileWriter("zwei.txt")));
017
018             for (nenner = 1; nenner <= 1024; nenner *= 2) {
019                 sum += 1.0 / nenner;
020                 f.print("Summand: 1/");
021                 f.print(nenner);
022                 f.print("    Summe: ");
023                 f.println(sum);
024             }
025             f.close();
026         } catch (IOException e) {
027             System.out.println("Fehler beim Erstellen der Datei");
028         }
029     }
030 }

```

Listing 13.4: Die Klasse `PrintWriter`

Das Programm verwendet Methoden zur Ausgabe von Strings, Ganz- und Fließkommazahlen. Nach Ende des Programms hat die Datei `zwei.txt` folgenden Inhalt:

```

Summand: 1/1    Summe: 1
Summand: 1/2    Summe: 1.5
Summand: 1/4    Summe: 1.75
Summand: 1/8    Summe: 1.875
Summand: 1/16   Summe: 1.9375
Summand: 1/32   Summe: 1.96875
Summand: 1/64   Summe: 1.98438
Summand: 1/128  Summe: 1.99219
Summand: 1/256  Summe: 1.99609
Summand: 1/512  Summe: 1.99805
Summand: 1/1024 Summe: 1.99902

```

Das vorliegende Beispiel realisiert sogar eine doppelte Schachtelung von [Writer](#)-Objekten. Das [PrintWriter](#)-Objekt schreibt in einen [BufferedWriter](#), der seinerseits in den [FileWriter](#) schreibt. Auf diese Weise werden Datentypen im ASCII-Format gepuffert in eine Textdatei geschrieben. Eine solche Schachtelung ist durchaus üblich in Java und kann in einer beliebigen Tiefe ausgeführt werden.

Tip

FilterWriter

Wie bereits mehrfach angedeutet, bietet die Architektur der [Writer](#)-Klassen die Möglichkeit, eigene Filter zu konstruieren. Dies kann beispielsweise durch Überlagern der Klasse [Writer](#) geschehen, so wie es etwa bei [PrintWriter](#) oder [BufferedWriter](#) realisiert wurde. Der offizielle Weg besteht allerdings darin, die abstrakte Klasse [FilterWriter](#) zu überlagern. [FilterWriter](#) besitzt ein internes [Writer](#)-Objekt `out`, das bei der Instanzierung an den Konstruktor übergeben wird. Zusätzlich überlagert es drei der vier [write](#)-Methoden, um die Ausgabe auf `out` umzuleiten. Die vierte [write](#)-Methode ([write\(String\)](#)) wird dagegen nicht überlagert, sondern ruft gemäß ihrer Implementierung in [Writer](#) die Variante [write\(String, int, int\)](#) auf.

Soll eine eigene Filterklasse konstruiert werden, so ist wie folgt vorzugehen:

- Die Klasse wird aus `FilterWriter` abgeleitet.
- Im Konstruktor wird der Superklassen-Konstruktor aufgerufen, um `out` zu initialisieren.
- Die drei `write`-Methoden werden separat überlagert. Dabei wird jeweils vor der Übergabe der Ausgabezeichen an die Superklassenmethode die eigene Filterfunktion ausgeführt.

Anschließend kann die neue Filterklasse wie gewohnt in einer Kette von geschachtelten `Writer`-Objekten verwendet werden.

Wir wollen uns die Konstruktion einer Filterklasse anhand der folgenden Beispielklasse `UpCaseWriter` ansehen, deren Aufgabe es ist, innerhalb eines Streams alle Zeichen in Großschrift zu konvertieren:

Beispiel

[Listing1305.java](#)

```
001 /* Listing1305.java */
002
003 import java.io.*;
004
005 class UpCaseWriter
006 extends FilterWriter
007 {
008     public UpCaseWriter(Writer out)
009     {
010         super(out);
011     }
012
013     public void write(int c)
014     throws IOException
015     {
016         super.write(Character.toUpperCase((char)c));
017     }
018
019     public void write(char cbuf[], int off, int len)
020     throws IOException
021     {
022         for (int i = 0; i < len; ++i) {
023             write(cbuf[off + i]);
024         }
025     }
026
027     public void write(String str, int off, int len)
028     throws IOException
029     {
030         write(str.toCharArray(), off, len);
031     }
032 }
033
034 public class Listing1305
035 {
036     public static void main(String args[])
037     {
038         PrintWriter f;
039         String s = "und dieser String auch";
040
041         try {
042             f = new PrintWriter(
043                 new UpCaseWriter(
044                     new FileWriter("upcase.txt")));
045             //Aufruf von außen
046             f.println("Diese Zeile wird schön groß geschrieben");
047             //Test von write(int)
048             f.write('a');
049             f.println();
050             //Test von write(String)
051             f.write(s);
052             f.println();
053             //Test von write(String, int, int)
054             f.write(s,0,17);
```

```

055         f.println();
056         //Test von write(char[], int, int)
057         f.write(s.toCharArray(),0,10);
058         f.println();
059         //---
060         f.close();
061     } catch (IOException e) {
062         System.out.println("Fehler beim Erstellen der Datei");
063     }
064 }
065 }

```

Listing 13.5: Konstruktion einer eigenen *FilterWriter*-Klasse

Im Konstruktor wird lediglich der Superklassen-Konstruktor aufgerufen, da keine weiteren Aufgaben zu erledigen sind. Die drei [write](#)-Methoden werden so überlagert, daß jeweils zunächst die Ausgabezeichen in Großschrift konvertiert werden und anschließend die passende Superklassenmethode aufgerufen wird, um die Daten an den internen [Writer out](#) zu übergeben.

Der Test von [UpCaseWriter](#) erfolgt mit der Klasse [Listing1305](#). Sie schachtelt einen [UpCaseWriter](#) innerhalb eines [FileWriter](#)- und eines [PrintWriter](#)-Objekts. Die verschiedenen Aufrufe der [write](#)- und [println](#)-Methoden rufen dann jede der vier unterschiedlichen [write](#)-Methoden des [UpCaseWriter](#)-Objekts mindestens einmal auf, um zu testen, ob die Konvertierung in jedem Fall vorgenommen wird. Die Ausgabe des Programms ist:

```

DIESE ZEILE WIRD SCHÖN GROß GESCHRIEBEN
A
UND DIESER STRING AUCH
UND DIESER STRING
UND DIESER

```

Wenn man sich die Implementierung der [write](#)-Methoden von [UpCaseWriter](#) genauer ansieht, könnte man auf die Idee kommen, daß sie wesentlich performanter implementiert werden könnten, wenn nicht alle Ausgaben einzeln an [write\(int\)](#) gesendet würden. So scheint es nahezuliegen, beispielsweise [write\(String, int, int\)](#) in der folgenden Form zu implementieren, denn die Methode [toUpperCase](#) existiert auch in der Klasse [String](#):

Warnung

```

super.write(str.toUpperCase(), off, len);

```

Die Sache hat nur leider den Haken, daß [String.toUpperCase](#) möglicherweise die Länge des übergebenen Strings verändert. So wird beispielsweise das "ß" in ein "SS" umgewandelt (an sich lobenswert!) und durch die unveränderlichen Konstanten [off](#) und [len](#) geht ein Zeichen verloren. Der hier beschrittene Workaround besteht darin, grundsätzlich die Methode [Character.toUpperCase](#) zu verwenden. Sie kann immer nur ein einzelnes Zeichen zurückgeben und läßt damit beispielsweise ein "ß" bei Umwandlung unangetastet.

Als interessante Erkenntnis am Rande lernen wir dabei, daß offensichtlich [String.toUpperCase](#) und [Character.toUpperCase](#) unterschiedlich implementiert sind. Ein Blick in den Quellcode der Klasse [String](#) bestätigt den Verdacht und zeigt zwei Sonderbehandlungen, eine davon für das "ß", wie folgender Auszug aus dem Quellcode des JDK 1.1 belegt (im JDK 1.2 sieht es ähnlich aus):

```

for (i = 0; i < len; ++i) {
    char ch = value[offset+i];
    if (ch == '\u00DF') { // sharp s
        result.append("SS");
        continue;
    }
    result.append(Character.toUpperCase(ch));
}

```

Im Umfeld dieser Methode sollte man also mit der nötigen Umsicht agieren. Das Gegenstück [lowerCase](#) hat diese Besonderheit übrigens nicht.

13.3 Eingabe-Streams

- 13.3 Eingabe-Streams
 - 13.3.1 Die abstrakte Klasse Reader
 - 13.3.2 Auswahl des Eingabegerätes
 - InputStreamReader und FileReader
 - StringReader und CharArrayReader
 - 13.3.3 Schachteln von Eingabe-Streams
 - BufferedReader
 - LineNumberReader
 - FilterReader und PushbackReader

13.3.1 Die abstrakte Klasse Reader

Basis aller sequentiellen Eingaben ist die abstrakte Klasse [Reader](#), die eine Schnittstelle für stream-basierte Eingaben zur Verfügung stellt:

```

public Reader()
public void close()
public void mark(int readAheadlimit)
public boolean markSupported()

public int read()
public int read(char cbuf[])
public int read(char cbuf[], int off, int len)

public long skip(long n)
public boolean ready()
public void reset()

```

Analog zu den [write](#)-Methoden von [Writer](#) stellt die Klasse [Reader](#) eine Reihe von [read](#)-Methoden zum Lesen von Daten zur Verfügung. Die parameterlose Variante liest das nächste Zeichen aus dem Eingabestrom und liefert es als [int](#), dessen Wert im Bereich von 0 bis 16383 liegt. Ein Rückgabewert von -1 zeigt das Ende des Eingabestroms an. Die beiden anderen Varianten von [read](#) übertragen eine Reihe von Zeichen in das als Parameter übergebene Array und liefern die Anzahl der tatsächlich gelesenen Zeichen als Rückgabewert. Auch hier wird -1 zurückgegeben, wenn das Ende des Streams erreicht ist.

Die Methode [ready](#) liefert [true](#), falls der nächste Aufruf von [read](#) erfolgen kann, ohne daß die Eingabe blockt, und [close](#) schließt den Eingabestrom. Mit Hilfe der Methoden [mark](#) und [reset](#) gibt es die Möglichkeit, eine bestimmte Position innerhalb des Eingabe-Streams zu markieren und zu einem späteren Zeitpunkt wieder anzuspringen. Zuvor muß allerdings durch einen Aufruf von [markSupported](#) überprüft werden, ob das Markieren überhaupt unterstützt wird. Ist dies nicht der Fall, würde ein Aufruf von [mark](#) oder [reset](#) eine Ausnahme erzeugen. Die Methode [mark](#) merkt sich die aktuelle Leseposition und spezifiziert, wie viele Bytes anschließend maximal gelesen werden können, bevor die Markierung ungültig wird. Ein Aufruf von [reset](#) setzt den Lesezeiger an die markierte Stelle zurück.

Mit der Methode [skip](#) ist es möglich, *n* Zeichen im Eingabestrom zu überspringen. Dabei kann es aus verschiedenen Gründen vorkommen, daß nicht exakt die angegebene Anzahl an Zeichen übersprungen wird (beispielsweise, wenn nicht mehr genügend Zeichen vorhanden sind). Der Rückgabewert von [skip](#) gibt die tatsächliche Anzahl an.

13.3.2 Auswahl des Eingabegerätes

Analog zur Klasse [Writer](#) dient auch bei der Klasse [Reader](#) die erste Ebene abgeleiteter Klassen vorwiegend dazu, die Art des Datenlieferanten zu bestimmen. [Tabelle 13.2](#) gibt eine Übersicht der aus [Reader](#) abgeleiteten Klassen.

Klasse	Bedeutung
InputStreamReader	Abstrakte Basisklasse für alle Reader, die einen Byte-Stream in einen Character-Stream umwandeln.

FileReader	Konkrete Ableitung von InputStreamReader zum Einlesen aus einer Datei.
FilterReader	Abstrakte Basisklasse für die Konstruktion von Eingabefiltern.
PushbackReader	Eingabefilter mit der Möglichkeit, Zeichen zurückzugeben.
BufferedReader	Reader zur Eingabepufferung und zum Lesen von kompletten Zeilen.
LineNumberReader	Ableitung aus BufferedReader mit der Fähigkeit, Zeilen zu zählen.
StringReader	Reader zum Einlesen von Zeichen aus einem String .
CharArrayReader	Reader zum Einlesen von Zeichen aus einem Zeichen-Array.
PipedReader	Reader zum Einlesen von Zeichen aus einem PipedWriter .

Tabelle 13.2: Aus Reader abgeleitete Klassen

In den folgenden Unterabschnitten werden die Klassen [InputStreamReader](#), [FileReader](#), [StringReader](#) und [CharArrayReader](#) erläutert. [FilterReader](#), [PushbackReader](#), [BufferedReader](#) und [LineNumberReader](#) werden in [Abschnitt 13.3.3](#) behandelt. Die Klasse [PipedReader](#) soll hier nicht erläutert werden.

InputStreamReader und FileReader

Die Klasse [InputStreamReader](#) ist die abstrakte Basisklasse für alle Reader, die eine Konvertierung zwischen Byte- und Character-Streams vornehmen. Sie enthält ein Objekt des Typs [ByteToCharConverter](#) (aus dem undokumentierten Paket [sun.io](#)), das die Konvertierung der Eingabezeichen bei allen lesenden Zugriffen vornimmt. Als abstrakte Basisklasse ist [InputStreamReader](#) für uns allerdings nicht so interessant wie die daraus abgeleitete Klasse [FileReader](#), die die Eingabe aus einer Datei ermöglicht. Sie implementiert die abstrakten Eigenschaften von [Reader](#) und bietet zusätzliche Konstruktoren, die es erlauben, eine Datei zu öffnen:

```

java.io.FileReader
public FileReader(String fileName)
    throws FileNotFoundException

public FileReader(File file)
    throws FileNotFoundException

public FileReader(FileDescriptor fd)
```

Bei der Übergabe der Zeichenkette `fileName` wird die Datei mit dem angegebenen Namen zum Lesen geöffnet. Falls sie nicht vorhanden ist, löst der Konstruktor eine Ausnahme des Typs [FileNotFoundException](#) aus. Die beiden anderen Konstruktoren erwarten ein [File](#)-Objekt, das eine zu öffnende Datei spezifiziert, oder ein [FileDescriptor](#)-Objekt, das eine bereits geöffnete Datei angibt.

Das folgende Beispiel demonstriert die Anwendung der Klasse [FileReader](#). Das Programm liest die Datei `config.sys` und gibt ihren Inhalt auf dem Bildschirm aus:

Beispiel

[Listing1306.java](#)

```

001 /* Listing1306.java */
002
003 import java.io.*;
004
005 public class Listing1306
006 {
007     public static void main(String args[])
008     {
009         FileReader f;
010         int c;
011
012         try {
013             f = new FileReader("c:\\config.sys");
014             while ((c = f.read()) != -1) {
015                 System.out.print((char)c);
016             }
017             f.close();
018         } catch (IOException e) {
019             System.out.println("Fehler beim Lesen der Datei");
020         }
021     }
022 }
```

Die Ausgabe des Programms ist:

```
DEVICE=c:\windows\himem.sys
DOS=HIGH,UMB
DEVICE=c:\windows\emm386.exe noems
SHELL=c:\command.com/e:1536/p
FILES=101
BUFFERS=5
DEVICEHIGH=c:\windows\command\ansi.sys
```

StringReader und CharArrayReader

Diese beide Klassen sind die Pendants zu `StringWriter` und `CharArrayWriter`. Sie erlauben das Lesen von Zeichen aus einem `String` bzw. einem Zeichen-Array. Die Schnittstelle der beiden Klassen ist identisch und unterscheidet sich von der Basisklasse `Reader` nur durch die geänderten Konstruktoren:

```
public StringReader(String s)
public CharArrayReader(char buf[])
public CharArrayReader(char buf[], int offset, int length)
```

Das folgende Programm zeigt die Verwendung der Klasse `StringReader` am Beispiel eines Programms, das einen `Reader` konstruiert, der den Satz liest, der hier an dieser Stelle steht:

```
001 /* Listing1307.java */
002
003 import java.io.*;
004
005 public class Listing1307
006 {
007     public static void main(String args[])
008     {
009         Reader f;
010         int c;
011         String s;
012
013         s = "Das folgende Programm zeigt die Verwendung\r\n";
014         s += "der Klasse StringReader am Beispiel eines\r\n";
015         s += "Programms, das einen Reader konstruiert, der\r\n";
016         s += "den Satz liest, der hier an dieser Stelle steht:\r\n";
017         try {
018             f = new StringReader(s);
019             while ((c = f.read()) != -1) {
020                 System.out.print((char)c);
021             }
022             f.close();
023         } catch (IOException e) {
024             System.out.println("Fehler beim Lesen des Strings");
025         }
026     }
027 }
```

Listing 13.7: Verwendung der Klasse `StringReader`

Die Ausgabe des Programms ist:

Das folgende Programm zeigt die Verwendung der Klasse `StringReader` am Beispiel eines Programms, das einen `Reader` konstruiert, der den Satz liest, der hier an dieser Stelle steht:

13.3.3 Schachteln von Eingabe-Streams

Das Konzept der geschachtelten Streams funktioniert bei der sequentiellen Eingabe genauso wie bei der Ausgabe. Mit den Klassen [BufferedReader](#), [LineNumberReader](#), [FilterReader](#) und [PushbackReader](#) stehen Klassen zur Verfügung, die im Konstruktor die Übergabe eines weiteren Readers erwarten und die Leseoperationen vor Ausführung der Filterfunktion an diesen [Reader](#) weiterleiten.

BufferedReader

Dieser Filter dient zur Pufferung von Eingaben und kann verwendet werden, um die Performance beim Lesen von externen Dateien zu erhöhen. Da nicht jedes Byte einzeln gelesen wird, verringert sich die Anzahl der Zugriffe auf den externen Datenträger, und die Lesegeschwindigkeit erhöht sich. Zusätzlich stellt [BufferedReader](#) die Methode [readLine](#) zur Verfügung, die eine komplette Textzeile liest und als [String](#) an den Aufrufer zurückgibt:

```
public String readLine()
    throws IOException
```

Eine Textzeile wird dabei durch die Zeichen '\n' oder '\r' oder durch die Folge "\r\n" begrenzt. Der Rückgabewert von [readLine](#) ist ein [String](#), der den Zeileninhalt ohne Begrenzungszeichen enthält bzw. [null](#), falls das Ende des Streams erreicht ist. [BufferedReader](#) besitzt zwei Konstruktoren:

```
public BufferedReader(Reader in)

public BufferedReader(Reader in, int sz)
```

Der erste Parameter *in* ist das [Reader](#)-Objekt, auf dem der [BufferedReader](#) aufgesetzt werden soll. Der optionale zweite Parameter *sz* gibt die Größe des internen Puffers an. Fehlt er, so wird eine für die meisten Situationen angemessene Standardeinstellung verwendet.

Das folgende Beispiel demonstriert den Einsatz der Eingabepufferung durch die Erweiterung von [Listing 13.6](#). Auch hier wird die Datei `config.sys` eingelesen und auf dem Bildschirm ausgegeben. Durch den Einsatz der Klasse [BufferedReader](#) versucht das Programm, die Performance beim Lesen der Datei zu erhöhen:

```
001 /* Listing1308.java */
002
003 import java.io.*;
004
005 public class Listing1308
006 {
007     public static void main(String args[])
008     {
009         BufferedReader f;
010         String line;
011
012         try {
013             f = new BufferedReader(
014                 new FileReader("c:\\config.sys"));
015             while ((line = f.readLine()) != null) {
016                 System.out.println(line);
017             }
018             f.close();
019         } catch (IOException e) {
020             System.out.println("Fehler beim Lesen der Datei");
021         }
022     }
023 }
```

Listing 13.8: Eingabepufferung beim Lesen aus Dateien

Zusätzlich wird die Eingabe nicht zeichen-, sondern mit Hilfe von [readLine](#) zeilenweise gelesen, was die Performance weiter erhöht. Die Ausgabe des Programms ist mit der von [Listing 13.6](#) identisch.

LineNumberReader

Diese Klasse ist eine Ableitung von [BufferedReader](#), die um die Fähigkeit erweitert wurde, die Anzahl der Eingabezeilen beim Einlesen zu zählen. Die Schnittstelle entspricht dabei exakt der von [BufferedReader](#), erweitert um die Methoden [getLineNumber](#) und

[setLineNumber](#):

[java.io.LineNumberReader](#)

```
public int getLineNumber()
```

```
public void setLineNumber(int lineNumber)
```

Mit [getLineNumber](#) wird der aktuelle Stand des Zeilenzählers abgefragt, mit [setLineNumber](#) kann er sogar verändert werden.

Das folgende Beispiel erweitert unsere bisherigen Programme zur Ausgabe der Datei `config.sys` in der Weise, daß nun jede einzelne Zeile mit vorangestellter Zeilennummer angezeigt wird. Dazu wird der [BufferedReader](#) durch einen [LineNumberReader](#) ersetzt und vor der Ausgabe jeder einzelnen Zeile zunächst die korrespondierende Zeilennummer ausgegeben:

Beispiel

[Listing1309.java](#)

```
001 /* Listing1309.java */
002
003 import java.io.*;
004
005 public class Listing1309
006 {
007     public static void main(String args[])
008     {
009         LineNumberReader f;
010         String line;
011
012         try {
013             f = new LineNumberReader(
014                 new FileReader("c:\\config.sys"));
015             while ((line = f.readLine()) != null) {
016                 System.out.print(f.getLineNumber() + ": ");
017                 System.out.println(line);
018             }
019             f.close();
020         } catch (IOException e) {
021             System.out.println("Fehler beim Lesen der Datei");
022         }
023     }
024 }
```

Listing 13.9: Die Klasse `LineNumberReader`

Die Ausgabe des Programms ist nun:

```
1: DEVICE=c:\windows\himem.sys
2: DOS=HIGH,UMB
3: DEVICE=c:\windows\emm386.exe noems
4: SHELL=c:\command.com/e:1536/p
5: FILES=101
6: BUFFERS=5
7: DEVICEHIGH=c:\windows\command\ansi.sys
```

FilterReader und PushbackReader

Das Schachteln von Eingabestreams funktioniert analog zum Schachteln von Ausgabestreams. Auch hier existiert eine abstrakte Basisklasse [FilterReader](#), die den eigentlichen [Reader](#) im Konstruktor übergeben bekommt und als Membervariable speichert. Bei der Konstruktion eigener Eingabefilter kann analog zur Konstruktion von Ausgabefiltern vorgegangen werden.

Das JDK 1.1 enthält einen vordefinierten Eingabefilter [PushbackReader](#), der aus [FilterReader](#) abgeleitet wurde. Ein [PushbackReader](#) erweitert die Klasse [FilterReader](#) um einen ein Byte großen *Pushbackbuffer*. Dieser erlaubt einer Anwendung, das zuletzt gelesene Zeichen wieder in den Eingabestrom zurückzuschieben. Der nächste Lesezugriff liest dann nicht das folgende Zeichen im Eingabestrom, sondern das gerade zurückgegebene Zeichen. Wahlweise kann beim Aufruf des Konstruktors sogar ein Pushbackpuffer mit einer von eins verschiedenen Größe angegeben werden:

[java.io.PushbackReader](#)

```
public PushbackReader(Reader in)
```

```
public PushbackReader(Reader in, int size)
```

Ein [PushbackReader](#) kann beispielsweise sinnvoll sein, wenn eine Methode das nächste Eingabezeichen kennen muß, um zu entscheiden, welche Aktion als nächstes auszuführen ist. Falls die Methode selbst nicht für die Behandlung des Eingabezeichens zuständig ist, kann sie das Zeichen an den Eingabestrom zurückgeben, und eine andere Methode kann mit der Bearbeitung beauftragt werden.

Die Rückgabe eines Zeichens wird mit Hilfe der Methode [unread](#) durchgeführt. Diese steht in verschiedenen Varianten zur Verfügung und kann zur Rückgabe eines einzelnen Zeichens oder mehrerer Zeichen verwendet werden:

java.io.PushbackReader

```
public void unread(int c)
    throws IOException

public void unread(char cbuf[], int off, int len)
    throws IOException

public void unread(char cbuf[])
    throws IOException
```

Hierbei muß das oder die zurückzugebenden Zeichen als Parameter [unread](#) übergeben werden.

13.4 Random-Access-Dateien

- [13.4 Random-Access-Dateien](#)
 - [13.4.1 Öffnen, Neuanlegen und Schließen](#)
 - [13.4.2 Positionierung des Dateizeigers](#)
 - [13.4.3 Lesezugriffe](#)
 - [13.4.4 Schreibzugriffe](#)

Neben der stream-basierten Ein- und Ausgabe bietet die Java-Klassenbibliothek auch die Möglichkeit des wahlfreien Zugriffs auf Dateien. Dabei wird eine Datei nicht als sequentielle Folge von Bytes angesehen, sondern als eine Art externes Array, das an beliebiger Stelle gelesen oder beschrieben werden kann.

Für den Zugriff auf Random-Access-Dateien stellt das Paket [java.io](#) die Klasse [RandomAccessFile](#) zur Verfügung. Anders als bei der stream-orientierten Ein-/Ausgabe kann dabei allerdings nur auf *Dateien* zugegriffen werden; die Verwendung von Arrays oder Pipes als Dateiersatz wird nicht unterstützt. Auch das durch die Streams realisierte Filterkonzept ist in Random-Access-Dateien nicht zu finden.

Die Klasse [RandomAccessFile](#) stellt Methoden zum Anlegen neuer Dateien und zum Öffnen vorhandener Dateien zur Verfügung. Für den lesenden oder schreibenden Zugriff auf die Datei gibt es ähnliche Methoden wie bei der sequentiellen Dateiverarbeitung. Zusätzlich kann der Satzzeiger wahlfrei positioniert werden, und es ist möglich, seine aktuelle Position abzufragen.

13.4.1 Öffnen, Neuanlegen und Schließen

Das Öffnen von Random-Access-Dateien erfolgt mit dem Konstruktor der Klasse [RandomAccessFile](#), der in zwei verschiedenen Varianten zur Verfügung steht:

[java.io.RandomAccessFile](#)

```
public RandomAccessFile(File file, String mode)
    throws IOException

public RandomAccessFile(String name, String mode)
    throws FileNotFoundException
```

Bei der Übergabe des [String](#)-Parameters `name` wird die Datei mit diesem Namen geöffnet und steht für nachfolgende Schreib- und Lesezugriffe zur Verfügung. Bei der Übergabe eines [File](#)-Objekts wird die durch dieses Objekt spezifizierte Datei geöffnet. Der zweite Parameter `mode` gibt die Art des Zugriffs auf die Datei an. Er kann entweder "r" sein, um die Datei nur zum Lesen zu öffnen, oder "rw", um sie zum Schreiben und Lesen zu öffnen. Ein reiner Schreibmodus, wie er beispielsweise unter UNIX möglich wäre, wird nicht unterstützt.

Während der Testphase des JDK 1.2 wurde die Deklaration der Konstruktoren teilweise dahingehend geändert, daß bei Zugriffsproblemen nicht mehr die Ausnahme [IOException](#), sondern [FileNotFoundException](#) ausgelöst wird. Gründe für eine der beiden Ausnahmen können sein, daß eine nicht vorhandene Datei zum Lesen geöffnet werden soll, daß der Dateiname ein *Verzeichnis* bezeichnet oder daß keine ausreichenden Zugriffsrechte auf die Datei zur Verfügung stehen.

JDK1.1/1.2

Leider gibt es in der Klasse [RandomAccessFile](#) keine explizite Differenzierung zwischen dem *Öffnen einer Datei zum Schreiben* und dem *Neuanlegen*. Hier gilt die implizite Regel, daß eine Datei neu angelegt wird, wenn sie beim Öffnen im Modus "w" nicht vorhanden ist. Existiert sie dagegen bereits, wird sie unverändert geöffnet, und es gibt keine Möglichkeit, ihren Inhalt zu löschen oder die Dateilänge auf einen bestimmten Wert zu setzen. Glücklicherweise bietet die Klasse [File](#) mit der Methode [delete](#) die Möglichkeit, eine Datei zu löschen und so das Neuanlegen über einen kleinen Umweg doch zu erreichen.

Warnung

Das Schließen einer Random-Access-Datei erfolgt wie bei Streams durch Aufruf der parameterlosen Methode [close](#). Diese leert zunächst alle internen Puffer und ruft dann die korrespondierende Betriebssystemfunktion zum Schließen der Datei auf.

13.4.2 Positionierung des Dateizeigers

Einer der Hauptunterschiede zwischen dem sequentiellen und dem wahlfreien Zugriff auf eine Datei besteht darin, daß beim wahlfreien Zugriff mit einem *expliziten* Satzzeiger gearbeitet wird. Jeder Schreib- und Lesezugriff erfolgt dabei an der Position, die durch den aktuellen Inhalt des Satzzeigers bestimmt wird, und positioniert den Zeiger um die Anzahl gelesener bzw. geschriebener Bytes weiter. Die Klasse [RandomAccessFile](#) stellt eine Reihe von Methoden zum Zugriff auf den Satzzeiger zur Verfügung:

```
public long getFilePointer()  
  
public void seek(long pos)  
  
public void skipBytes(int n)  
  
public long length()
```

[getFilePointer](#) liefert die aktuelle Position des Satzzeigers; das erste Byte einer Datei steht dabei an Position 0. Da der Rückgabewert vom Typ [long](#) ist, unterstützt Java den Zugriff auf Dateien, die größer als 2 GByte sind, sofern es das Betriebssystem zuläßt.

Das Positionieren des Satzzeigers erfolgt mit der Methode [seek](#), die den Satzzeiger an die durch [pos](#) angegebene Stelle positioniert. Anders als in C bezieht sich der Wert von [pos](#) dabei immer auf den Anfang der Datei. Das Positionieren relativ zur aktuellen Position kann mit der Methode [skipBytes](#) erledigt werden. Die neue Position wird dabei aus der aktuellen Position plus dem Inhalt des Parameters [n](#) berechnet. Auch negative Werte für [n](#) sind dabei erlaubt und bewirken eine Rückwärtsverschiebung des Satzzeigers.

13.4.3 Lesezugriffe

Für die lesenden Zugriffe auf eine Random-Access-Datei stehen die folgenden Methoden zur Verfügung:

```
public final boolean readBoolean()  
public final byte readByte()  
public final char readChar()  
public final double readDouble()  
public final float readFloat()  
public final int readInt()  
public final long readLong()  
public final short readShort()  
public final String readUTF()  
public final void readFully(byte b[])  
public final void readFully(byte b[], int off, int len)  
public final String readLine()  
public final int readUnsignedByte()  
public final int readUnsignedShort()
```

Sie lesen jeweils ein Element des angegebenen Typs, das in dem durch die korrespondierende [write...](#)-Methode vorgegebenen binären Format vorliegen muß. [readFully](#) kann dazu verwendet werden, beliebig viele Datenbytes ungeachtet ihres Datentyps einzulesen. [readLine](#) liest eine ganze Zeile Text aus der Eingabedatei und gibt sie als [String](#) an den Aufrufer zurück.

Darüber hinaus steht auch eine Reihe von [read](#)-Methoden zur Verfügung, die zum Einlesen eines einzelnen Bytes oder einer Menge von Bytes verwendet werden können:

```
public int read()  
public int read(byte b[])  
public int read(byte b[], int off, int len)
```

13.4.4 Schreibzugriffe

Die schreibenden Zugriffe erfolgen mit analogen Methoden:

```
public final void writeBoolean(boolean v)  
public final void writeByte(int v)  
public final void writeBytes(String s)  
public final void writeChar(int v)  
public final void writeChars(String s)  
public final void writeDouble(double v)  
public final void writeFloat(float v)  
public final void writeInt(int v)  
public final void writeLong(long v)  
public final void writeShort(int v)  
public final void writeUTF(String str)
```

Zusätzlich gibt es auch hier einige elementare [write](#)-Methoden :

```

public void write(int b)
public void write(byte b[])
public void write(byte b[], int off, int len)

```

Das folgende Listing zeigt die Verwendung der Klasse [RandomAccessFile](#) am Beispiel eines Programms, das die Signatur und Versionsnummer aus einem [.class](#)-File herausliest. Die Signatur einer Klassendatei ergibt das Wort »CAFEBABE«, wenn man die hexadezimale Darstellung der ersten vier Bytes ausgibt. In den nächsten beiden Bytes folgt die Minor-Versionsnummer und in den darauffolgenden zwei Bytes die Major-Versionsnummer.

Beispiel

Das nachfolgende Programm implementiert eine Klasse [ClassFileReader](#), die den Zugriff auf die Klassendatei ermöglicht. Der Konstruktor öffnet die Datei, und die Ausgabe der Signatur und Versionsinformation erfolgt mit Hilfe der Methoden [printSignature](#) und [printVersion](#). Das Einlesen der Signatur erfolgt durch Lesen der ersten 4 Bytes der Datei, die dann jeweils in High- und Lowbyte zerlegt und in ihre hexadezimale Darstellung umgewandelt werden. Bei der Verwendung der Methode [read](#) zum Einlesen der Bytes ist zu beachten, daß der Rückgabewert vom Typ [int](#) ist. Er darf auch nicht in ein [byte](#) konvertiert werden, weil es sonst einen Vorzeichenüberlauf geben würde. Das Einlesen der Versionsnummern erfolgt mit der Methode [readShort](#), die einen vorzeichenlosen 16-Bit-Wert aus der Datei liest. Auch hier ist der Rückgabewert vom Typ [int](#), um den gesamten Wertebereich von 0 bis 65535 darstellen zu können.

[Listing1310.java](#)

```

001 /* Listing1310.java */
002
003 import java.io.*;
004
005 class ClassFileReader
006 {
007     private RandomAccessFile f;
008
009     public ClassFileReader(String name)
010     throws IOException
011     {
012         if (!name.endsWith(".class")) {
013             name += ".class";
014         }
015         f = new RandomAccessFile(name, "r");
016     }
017
018     public void close()
019     {
020         if (f != null) {
021             try {
022                 f.close();
023             } catch (IOException e) {
024                 //nichts
025             }
026         }
027     }
028
029     public void printSignature()
030     throws IOException
031     {
032         String ret = "";
033         int b;
034
035         f.seek(0);
036         for (int i=0; i<4; ++i) {
037             b = f.read();
038             ret += (char)(b/16+'A'-10);
039             ret += (char)(b%16+'A'-10);
040         }
041         System.out.println(
042             "Signatur..... "+

```



```

043         ret
044     );
045 }
046
047 public void printVersion()
048 throws IOException
049 {
050     int minor, major;
051
052     f.seek(4);
053     minor = f.readShort();
054     major = f.readShort();
055     System.out.println(
056         "Version..... "+
057         major+"."+minor
058     );
059 }
060
061 }
062
063 public class Listing1310
064 {
065     public static void main(String[] args)
066     {
067         ClassFileReader f;
068
069         try {
070             f = new ClassFileReader("Listing1310");
071             f.printSignature();
072             f.printVersion();
073         } catch (IOException e) {
074             System.out.println(e.toString());
075         }
076     }
077 }

```

Listing 13.10: Lesen einer .class-Datei mit der Klasse RandomAccessFile

Die Ausgabe des Programms ist:

```

Signatur..... CAFEBADE
Version..... 45.3

```

13.5 Datei- und Verzeichnis-Handling

- [13.5 Datei- und Verzeichnis-Handling](#)
 - [13.5.1 Konstruktion eines File-Objektes](#)
 - [13.5.2 Zugriff auf Teile des Pfadnamens](#)
 - [13.5.3 Informationen über die Datei](#)
 - [13.5.4 Zugriff auf Verzeichnisse](#)
 - [13.5.5 Anlegen von Dateien](#)
 - [Temporäre Dateien](#)
 - [Lockdateien](#)

Im Paket `java.io` gibt es eine Klasse `File`, die als Abstraktion eines *Dateinamens* angesehen werden kann. `File` kann sowohl absolute als auch relative Namen unter UNIX und DOS/Windows (UNC und Laufwerksbuchstabe) repräsentieren und sowohl für eine Datei als auch für ein Verzeichnis stehen. Im Gegensatz zu den bisher besprochenen Klassen spielt der *Inhalt* einer Datei in der Klasse `File` keine Rolle. Statt dessen abstrahiert `File` den Namen und Zugriffspfad einer Datei und die Eigenschaften, die im Verzeichnis, in dem die Datei liegt, über sie gespeichert sind. Neben Dateien kann ein `File`-Objekt auch Verzeichnisse repräsentieren.

13.5.1 Konstruktion eines File-Objektes

Die Klasse `File` besitzt drei Konstruktoren:

```

public File(String pathname)
public File(String parent, String child)
public File(File parent, String child)

```

Wird lediglich der String `pathname` übergeben, so wird ein `File`-Objekt zu dem angegebenen Datei- oder Verzeichnisnamen konstruiert. Alternativ dazu kann der zweite Konstruktor verwendet werden, wenn Verzeichnis- und Dateiname getrennt übergeben werden sollen. Eine ähnliche Funktion übernimmt auch der dritte Konstruktor. Hier wird jedoch der übergebene Verzeichnisname als `File`-Objekt zur Verfügung gestellt.

Bei der Konstruktion von Datei- und Verzeichnisnamen unter MS-DOS ist zu beachten, daß der Separator (Backslash) gleichzeitig Escape-Zeichen für Strings ist und daher in Verzeichnis- oder Dateiliteralen doppelt angegeben werden muß (siehe [Abschnitt 4.2.2](#)).

Warnung

Beispiele für gültige Konstruktoraufrufe sind:

```

new File("TestFile.java");
new File("c:\\arc\\doku\\javacafe\\kap01.doc");
new File(".", "TestFile.java");
new File("c:\\config.sys");

```

13.5.2 Zugriff auf Teile des Pfadnamens

Nachdem ein `File`-Objekt konstruiert ist, können die Methoden zum Zugriff auf die einzelnen Bestandteile des Dateinamens aufgerufen werden:

```

public String getName()
public String getPath()
public String getAbsolutePath()
public String getParent()

```

`getName` liefert den Namen der Datei oder des Verzeichnisses; eventuelle Verzeichnisinformationen sind nicht darin enthalten. `getPath` liefert den kompletten Namen inklusive darin enthaltener Verzeichnisinformationen.

Mit [getAbsolutePath](#) kann der *absolute* Pfadname für das [File](#)-Objekt ermittelt werden. Wurde das [File](#)-Objekt mit Hilfe eines absoluten Pfadnamens konstruiert, liefert [getAbsolutePath](#) genau diesen Namen. Wurde es dagegen mit einem *relativen* Namen konstruiert, so stellt [getAbsolutePath](#) den Namen des aktuellen Verzeichnisses vor den Namen. Dabei werden allerdings die Namen `.` und `..` nicht interpretiert, so daß leicht Pfadangaben wie `C:\ARC\DOKU\javacafe\examples\.\TestFile.java` entstehen können, die in der Mitte einen oder mehrere Punkte enthalten.

Schließlich gibt es noch die Methode [getParent](#), die den Namen des *Vaterverzeichnisses* ermittelt. Repräsentiert das [File](#)-Objekt eine Datei, ist das der Name des Verzeichnisses, in dem die Datei liegt. Handelt es sich um ein Verzeichnis, wird der Name des darüber liegenden Verzeichnisses geliefert. Gibt es kein Vatterverzeichnis, liefert [getParent](#) `null` zurück.

13.5.3 Informationen über die Datei

Die Klasse [File](#) besitzt eine ganze Reihe von Methoden, um Informationen über die durch das [File](#)-Objekt bezeichnete Datei oder das Verzeichnis zu gewinnen:

[java.io.File](#)

```
public boolean exists()

public boolean canWrite()

public boolean canRead()

public boolean isHidden()

public boolean isFile()

public boolean isDirectory()

public boolean isAbsolute()

public long lastModified()

public long length()
```

Mit [exists](#) kann getestet werden, ob die Datei oder das Verzeichnis überhaupt existiert. Die Methoden [canWrite](#) und [canRead](#) ermitteln, ob ein lesender bzw. schreibender Zugriff möglich ist, mit [isHidden](#) kann festgestellt werden, ob die Datei versteckt ist. Mit [isFile](#) und [isDirectory](#) kann unterschieden werden, ob es sich um eine Datei oder ein Verzeichnis handelt. [isAbsolute](#) gibt an, ob das Objekt mit Hilfe einer absoluten Pfadbezeichnung konstruiert wurde.

[lastModified](#) liefert den Zeitpunkt der letzten Änderung der Datei in Millisekunden seit dem 1.1.1970. Dieser kann entweder direkt in Vergleichen verwendet werden, oder man benutzt ihn zur Konstruktion eines [Date](#)-Objekts. Das [Date](#)-Objekt kann beispielsweise an die Methode [setTime](#) der Klasse [GregorianCalendar](#) übergeben werden, um die einzelnen Uhrzeitkomponenten zu extrahieren.

13.5.4 Zugriff auf Verzeichnisse

Wurde ein [File](#)-Objekt für ein Verzeichnis konstruiert, so stehen weitere Methoden zur Verfügung, um auf die zusätzlichen Funktionen eines Verzeichnisses zuzugreifen. Mit Hilfe der Methode [list](#) ist es beispielsweise möglich, den Inhalt des Verzeichnisses auszulesen:

[java.io.File](#)

```
public String[] list()
```

[list](#) liefert ein Array von Strings, das für jeden gefundenen Verzeichniseintrag ein Element enthält. Die Liste enthält die Namen aller Dateien und Unterverzeichnisse mit Ausnahme von `.` und `..`. [list](#) gibt es noch in einer zweiten Variante, bei der die Auswahl der Verzeichniseinträge eingeschränkt werden kann. Dabei muß ein Objekt übergeben werden, das das Interface [FilenameFilter](#) implementiert. Dieses besitzt eine Methode [accept](#), die für jede gefundene Datei aufgerufen wird und entscheidet, ob sie in die Liste aufgenommen werden soll oder nicht.

Zusätzlich gibt es die statische Methode [listRoots](#), mit der eine Liste aller »Wurzeln« der verfügbaren Dateisysteme beschafft werden kann. Unter UNIX gibt es lediglich die Wurzel `/`, während es unter Windows zu jedem Laufwerksbuchstaben eine Wurzel gibt.

Neben dem Zugriff auf die Verzeichniseinträge gibt es in der Klasse [File](#) auch Methoden, um Dateien oder Verzeichnisse zu löschen oder umzubenennen und um Verzeichnisse neu anzulegen:

```

public boolean mkdir()

public boolean mkdirs()

public boolean renameTo(File dest)

public boolean delete()

```

Die Methode [delete](#) löscht die durch das [File](#)-Objekt bezeichnete Datei. Mit [renameTo](#) wird das [File](#)-Objekt in das als Parameter übergebene Objekt umbenannt. Durch Aufruf von [mkdir](#) wird das spezifizierte Verzeichnis angelegt. Mit [mkdirs](#) werden sogar alle Väterverzeichnis automatisch angelegt, wenn sie noch nicht existieren. Alle Methoden geben [true](#) zurück, wenn sie ihre Aufgabe erfolgreich ausführen konnten; andernfalls geben sie [false](#) zurück.

Das folgende Listing zeigt die Verwendung der Klasse [File](#) und den Aufruf verschiedener Methoden:

Beispiel

[TestFile.java](#)

```

001 /* TestFile.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 public class TestFile
007 {
008     public static void main(String[] args)
009     {
010         File fil = new File("TestFile.java");
011         TestFile.printFileInfo(fil);
012         fil = new File("..");
013         TestFile.printFileInfo(fil);
014     }
015
016     static void printFileInfo(File fil)
017     {
018         System.out.println("Name= "+fil.getName());
019         System.out.println("Path= "+fil.getPath());
020         System.out.println("AbsolutePath= "+fil.getAbsolutePath());
021         System.out.println("Parent= "+fil.getParent());
022         System.out.println("exists= "+fil.exists());
023         System.out.println("canWrite= "+fil.canWrite());
024         System.out.println("canRead= "+fil.canRead());
025         System.out.println("isFile= "+fil.isFile());
026         System.out.println("isDirectory= "+fil.isDirectory());
027         if (fil.isDirectory()) {
028             String fils[] = fil.list();
029             for (int i=0; i<fils.length; ++i) {
030                 System.out.println("  "+fils[i]);
031             }
032         }
033         System.out.println("isAbsolute= "+fil.isAbsolute());
034         System.out.println(
035             "lastModified= "+(new Date(fil.lastModified()))
036         );
037         System.out.println("length= "+fil.length());
038         System.out.println("");
039     }
040 }

```

Listing 13.11: Verwendung der Klasse File

Ein Aufruf des Programms liefert folgende Ausgabe:

```

Name= TestFile.java
Path= TestFile.java
AbsolutePath= C:\ARC\DOKU\java\examples\TestFile.java
Parent= null
exists= true

```

```

canWrite= true
canRead= true
isFile= true
isDirectory= false
isAbsolute= false
lastModified= Sun Jan 05 17:15:56 1997
length= 1242

Name= ..
Path= ..
AbsolutePath= C:\ARC\DOKU\java\examples\..
Parent= null
exists= true
canWrite= true
canRead= true
isFile= false
isDirectory= true
    makefile
    html.cfg
    ...
    jdbc.sgml
    tuning.sgml
    reflection.sgml
isAbsolute= false
lastModified= Wed Jul 22 16:55:32 GMT+02:00 1998
length= 0

```

13.5.5 Anlegen von Dateien

Das Anlegen von Dateien wurde ja bereits in früheren Abschnitten dieses Kapitels behandelt. Hier soll es noch einmal erwähnt werden, weil die Klasse [File](#) zusätzlich die Möglichkeit bietet, temporäre Dateien und Lockdateien anzulegen. Beide Varianten haben ihre Anwendungen, und wir wollen sie im folgenden kurz vorstellen.

Temporäre Dateien

Zum Anlegen von temporären Dateien stehen zwei Varianten der Methode [createTempFile](#) zur Verfügung:

[java.io.File](#)

```

public static File createTempFile(
    String prefix, String suffix, File dir
)
    throws IOException

public static File createTempFile(
    String prefix, String suffix
)
    throws IOException

```

In beiden Fällen ist es erforderlich, einen Präfix- und einen Suffix-String zu spezifizieren. Als Präfix sollte normalerweise ein kurzer String von drei oder vier Buchstaben verwendet werden, der den Typ der temporären Datei identifiziert. Der Suffix wird als Dateierweiterung verwendet, er könnte beispielsweise ".tmp" oder ".\$\$\$" sein. [createTempFile](#) füllt den Platz zwischen Präfix und Erweiterung mit einer Zeichenkette, so daß der resultierende Dateiname eindeutig ist. Wird kein weiterer Parameter angegeben, legt die Methode eine neue temporäre Datei in einem systemspezifischen Verzeichnis an (typischerweise "\tmp" oder "\temp"). Alternativ kann als drittes Argument ein [File](#)-Objekt übergeben werden, das ein alternatives Verzeichnis für die temporäre Datei angibt.

Das folgende Listing zeigt ein einfaches Beispiel für die Anwendung der Methode [createTempFile](#):

Beispiel

[Listing1312.java](#)

```
001 /* Listing1312.java */
002
003 import java.io.*;
004
005 public class Listing1312
006 {
007     public static void main(String args[])
008     {
009         try {
010             File tmp = File.createTempFile("xyz", ".tmp", null);
011         } catch (IOException e) {
012             System.out.println(e.toString());
013         }
014     }
015 }
```

Listing 13.12: Anlegen einer temporären Datei

Auf dem Testrechner wurden bei zweimaligem Aufruf des Programms im Verzeichnis "c:\tmp" die beiden folgenden Dateien angelegt:

xyz11626.tmp
xyz39639.tmp

Lockdateien

Ein häufiger Kritikpunkt der JDKs vor der Version 1.2 war, daß Java keine portable Möglichkeit zum Sperren von Dateien vorsah. Damit war es nicht (oder nur mit zusätzlichem Aufwand) möglich, Programme zu schreiben, die in einer Mehrbenutzerumgebung von unterschiedlichen Arbeitsplätzen gleichzeitig kontrolliert auf dieselben Dateien zugreifen.

JDK1.1/1.2

Mit den Methoden [createNewFile](#) und [deleteOnExit](#) bietet das JDK 1.2 nun eine rudimentäre Möglichkeit, diese Fähigkeiten zu realisieren:

[java.io.File](#)

```
public boolean createNewFile()
    throws IOException

public void deleteOnExit()
```

[createNewFile](#) erzeugt eine neue Datei aus dem Dateinamen des zugehörigen [File](#)-Objekts. Die Datei wird aber nur dann erzeugt, wenn sie bisher noch nicht vorhanden war. War sie dagegen bereits vorhanden, schlägt der Aufruf fehl und liefert [false](#) als Rückgabewert. Bei Erfolg wird die Datei angelegt und [true](#) zurückgegeben. Das JDK garantiert, daß die beiden erforderlichen Teiloperationen *Feststellen, ob die Datei bereits existiert* und *Anlegen einer neuen Datei* atomar ablaufen, also nicht unterbrechbar sind. Durch Aufruf von [deleteOnExit](#) kann sichergestellt werden, daß die Datei beim Beenden der VM in jedem Fall gelöscht wird, selbst wenn das Ende durch eine Ausnahme ausgelöst wurde.

Beide Methoden können nun auf unterschiedliche Weise zur Realisierung eines Locking-Systems verwendet werden. So kann beispielsweise der Name der Datei als zu sperrende Ressource und die Datei selbst als eigentliche Sperre angesehen werden. Dann würde es für jede zu sperrende Ressource eine eigene Sperrdatei geben (was bei einer großen Anzahl von potentiellen Ressourcen sehr viele einzelne Dateien bedeuten würde). Oder es ist denkbar, daß alle Sperrinformationen in einer einzigen Datei gehalten werden und nur der Zugriff auf diese Datei mit Hilfe einer Sperrdatei gesichert wird. Diese würde dann mit [createNewFile](#) in der Art eines Semaphors angelegt werden, um die Sperrdatei zu betreten, und nach Ende der Bearbeitung wieder entfernt werden.

13.6 Zusammenfassung

- [13.6 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Byte- und Character-Streams
- Die abstrakte Klasse [Writer](#)
- Die Auswahl des Ausgabegeräts und die Klassen [OutputStreamWriter](#), [FileWriter](#), [StringWriter](#) und [CharArrayWriter](#)
- Schachteln von Ausgabe-Streams und die Klassen [BufferedWriter](#), [PrintWriter](#) und [FilterWriter](#)
- Die abstrakte Klasse [Reader](#)
- Die Auswahl des Eingabegeräts und die Klassen [InputStreamReader](#), [FileReader](#), [StringReader](#) und [CharArrayReader](#)
- Schachteln von Eingabe-Streams und die Klassen [FilterReader](#), [PushbackReader](#), [BufferedReader](#) und [LineNumberReader](#)
- Random-Access-Dateien und die Klasse [RandomAccessFile](#)
- Datei- und Verzeichnis-Handling mit der Klasse [File](#)
- Temporäre Dateien und Locking

Kapitel 14

Grafikausgabe

- [14 Grafikausgabe](#)
 - [14.1 Das Abstract Windowing Toolkit](#)
 - [14.2 Grundlagen der Grafikausgabe](#)
 - [14.2.1 Anlegen eines Fensters](#)
 - [14.2.2 Die Methode paint](#)
 - [14.2.3 Das grafische Koordinatensystem](#)
 - [14.3 Elementare Grafikroutinen](#)
 - [14.3.1 Linie](#)
 - [14.3.2 Rechteck](#)
 - [14.3.3 Polygon](#)
 - [14.3.4 Kreis](#)
 - [14.3.5 Kreisbogen](#)
 - [14.4 Weiterführende Funktionen](#)
 - [14.4.1 Linien- oder Füllmodus](#)
 - [14.4.2 Kopieren und Löschen von Flächen](#)
 - [14.4.3 Die Clipping-Region](#)
 - [14.5 Zusammenfassung](#)

14.1 Das Abstract Windowing Toolkit

- [14.1 Das Abstract Windowing Toolkit](#)

Im Gegensatz zu den meisten anderen Programmiersprachen wurde Java von Anfang an mit dem Anspruch entwickelt, ein vielseitiges, aber einfach zu bedienendes System für die Gestaltung grafischer Oberflächen zur Verfügung zu stellen. Das Resultat dieser Bemühungen steht als Grafikbibliothek unter dem Namen *Abstract Windowing Toolkit (AWT)* zur Verfügung.

Die Fähigkeiten des AWT lassen sich grob in vier Gruppen unterteilen:

- Grafische Primitivoperationen zum Zeichnen von Linien oder Füllen von Flächen und zur Ausgabe von Text
- Methoden zur Steuerung des Programmablaufs auf der Basis von Nachrichten für Tastatur-, Maus- und Fensterereignisse
- Dialogelemente zur Kommunikation mit dem Anwender und Funktionen zum portablen Design von Dialogboxen
- Fortgeschrittenere Grafikfunktionen zur Darstellung und Manipulation von Bitmaps und zur Ausgabe von Sound

Da die grafischen Fähigkeiten Bestandteil der Sprache bzw. ihrer Klassenbibliothek sind, können sie als portabel angesehen werden. Unabhängig von der Zielplattform wird ein GUI-basiertes Programm auf allen verwendeten Systemen gleich oder zumindest ähnlich laufen.

Die Entwicklung von grafikorientierten Anwendungen im JDK 1.0 war zwar relativ einfach und erzeugte portable Programme mit grafischer Oberfläche, war aber durch eine Reihe von Fehlern und Restriktionen des AWT eingeschränkt. Vor allem bei der Erstellung großer GUI-Anwendungen wurden die Programme schnell unübersichtlich, und die Performance litt unter dem unzulänglichen Event-Modell der Version 1.0. Mit der Version 1.1 des JDK hat Sun das AWT massiv verändert. Tiefgreifende Bugfixings wurden vorgenommen, und eine Vielzahl von Methodennamen wurde geändert. Vor allem aber wurde das Event-Handling, also der Transfer von GUI-Ereignissen, komplett überarbeitet.

Hinweis

Das neue Schlagwort lautet *Delegation Based Event Handling* und meint die Fähigkeit des AWT, GUI-Ereignisse auch an Nicht-Komponenten weiterzuleiten und dort zu behandeln. Obwohl das Verständnis für diese neuen Techniken deutlich schwieriger zu erlangen ist als beim alten Event-Modell, lohnt sich der Aufwand. Die Entwicklung großer GUI-Anwendungen mit einer klaren Trennung zwischen Benutzeroberfläche und Applikationslogik wird so erst möglich gemacht. Wir werden in [Kapitel 18](#) detailliert auf das neue Paradigma eingehen. Wo erforderlich, werden einzelne Bestandteile bereits vorher informell eingeführt.

14.2 Grundlagen der Grafikausgabe

- [14.2 Grundlagen der Grafikausgabe](#)
 - [14.2.1 Anlegen eines Fensters](#)
 - [14.2.2 Die Methode paint](#)
 - [14.2.3 Das grafische Koordinatensystem](#)

14.2.1 Anlegen eines Fensters

Um die Grafikfähigkeiten von Java nutzen zu können, muß das Paket `java.awt` eingebunden werden. Dies geschieht zweckmäßigerweise mit Hilfe folgender Anweisung am Anfang der Klassendefinition:

```
import java.awt.*;
```

Danach stehen alle Klassen aus dem Paket `java.awt` zur Verfügung.

Zur Ausgabe von grafischen Elementen benötigt die Anwendung ein Fenster, auf das die Ausgabeoperationen angewendet werden können. Während bei der Programmierung eines Applets ein Standardfenster automatisch zur Verfügung gestellt wird, muß eine *Applikation* ihre Fenster selbst erzeugen. Da die Kommunikation mit einem Fenster über eine Reihe von Callback-Methoden abgewickelt wird, wird eine Fensterklasse in der Regel nicht einfach instanziiert. Statt dessen ist es meist erforderlich, eine eigene Klasse aus einer der vorhandenen abzuleiten und die benötigten Interfaces zu implementieren.

Zum Ableiten einer eigenen Fensterklasse wird in der Regel entweder die Klasse `Frame` oder die Klasse `Dialog` verwendet, die beide aus `Window` abgeleitet sind. Da `Dialog` vorwiegend dafür verwendet wird, Dialogboxen zu erstellen, die über darin enthaltene Komponenten mit dem Anwender kommunizieren, wollen wir ihre Verwendung bis zum [Kapitel 21](#) zurückstellen. Die wichtigste Klasse zur Ausgabe von Grafiken in Java-Applikationen ist also `Frame`.

Um ein einfaches Fenster zu erzeugen und auf dem Bildschirm anzuzeigen, muß ein neues Element der Klasse `Frame` erzeugt, auf die gewünschte Größe gebracht und durch Aufruf der Methode `setVisible` sichtbar gemacht werden:

Beispiel

[Listing1401.java](#)

```
001 /* Listing1401.java */
002
003 import java.awt.*;
004
005 class Listing1401
006 {
007     public static void main(String args[])
008     {
009         Frame wnd = new Frame("Listing1401");
010
011         wnd.setSize(400,300);
012         wnd.setVisible(true);
013     }
014 }
```

Listing 14.1: Ein einfaches Fenster erzeugen

Das Ausführen dieses Programms führt dazu, daß ein Fenster mit dem Titel »Listing1401« erzeugt und in der Größe 400*300 Pixel auf dem Bildschirm angezeigt wird.

Da wir noch keinen Code für die Behandlung von GUI-Events eingebaut haben, bietet das Fenster lediglich das von Windows her bekannte Standardverhalten. Es läßt sich verschieben und in der Größe verändern und besitzt eine Titel- und Menüleiste, die mit einem Systemmenü ausgestattet ist. Anders als in anderen grafikorientierten Systemen gibt es noch keine Funktionalität zum Beenden des Fensters. Das Beispielpogramm kann daher nur durch einen harten Abbruch seitens des Benutzers (z.B. durch Drücken von `[STRG]+[C]` in der DOS-Box, aus der das Fenster gestartet wurde) beendet werden. Wir werden im nächsten Beispiel Programmcode zum ordnungsgemäßen Schließen des Fensters einfügen.

Hinweis

14.2.2 Die Methode paint

Die Ausgabe in ein Fenster erfolgt durch Überlagern der Methode `paint`, die immer dann aufgerufen wird, wenn das Fenster ganz oder teilweise neu gezeichnet werden muß. Dies ist beispielsweise dann der Fall, wenn das Fenster zum ersten Mal angezeigt wird oder durch Benutzeraktionen ein Teil des Fensters sichtbar wird, der bisher verdeckt war. `paint` bekommt beim Aufruf eine Instanz der Klasse `Graphics` übergeben:

`java.awt.Component`

```
public void paint(Graphics g)
```

`Graphics` ist Javas Implementierung eines *Device-Kontexts* (auch *Grafikkontext* genannt) und stellt somit die Abstraktion eines universellen Ausgabegeräts für Grafik und Schrift dar. Die Klasse bietet Methoden zur Erzeugung von Linien-, Füll- und Textelementen. Darüber hinaus verwaltet `Graphics` die Zeichenfarbe, in der alle Ausgaben erfolgen, und einen Font, der zur Ausgabe von Schrift verwendet wird. Ein Device-Kontext kann daher als eine Art universelles Ausgabegerät angesehen werden, das elementare Funktionen zur Ausgabe von farbigen Grafik- und Schriftzeichen zur Verfügung stellt.

14.2.3 Das grafische Koordinatensystem

Die Ausgabe von Grafik basiert auf einem zweidimensionalen Koordinatensystem, dessen Ursprungspunkt (0,0) in der linken oberen Ecke liegt (siehe [Abbildung 14.1](#)). Positive *x*-Werte erstrecken sich nach rechts, positive *y*-Werte nach unten. Die Maßeinheit entspricht einem Bildschirmpixel und ist damit geräteabhängig.

In den meisten Fällen steht dem Programm nicht das gesamte Fenster zur Ausgabe zur Verfügung, sondern es gibt *Randelemente* wie Titelzeilen, Menüs oder Rahmen, die nicht überschrieben werden können. Mit der Methode `getInsets` kann die Größe dieser Randelemente ermittelt werden. Wir werden darauf in [Kapitel 15](#) noch einmal zurückkommen.

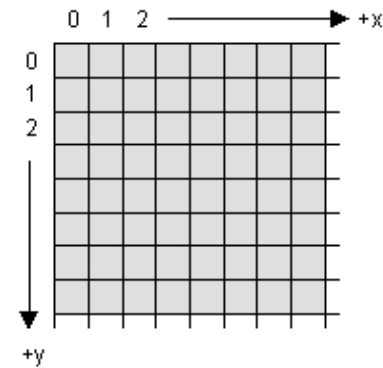


Abbildung 14.1: Das Koordinatensystem von Java

14.3 Elementare Grafikroutinen

- [14.3 Elementare Grafikroutinen](#)
 - [14.3.1 Linie](#)
 - [14.3.2 Rechteck](#)
 - [14.3.3 Polygon](#)
 - [14.3.4 Kreis](#)
 - [14.3.5 Kreisbogen](#)

Die Klasse [Graphics](#) stellt neben vielen anderen Funktionen auch eine Sammlung von linienbasierten Zeichenoperationen zur Verfügung. Diese sind zur Darstellung von einfachen Linien, Rechtecken oder Polygonen sowie von Kreisen, Ellipsen und Kreisabschnitten geeignet. Wir wollen im folgenden jede dieser Funktionsgruppen vorstellen und ihre Anwendung an einem Beispiel zeigen.

Um nicht jeweils eine komplette Klassendefinition angeben zu müssen, werden wir in den folgenden Beispielen jeweils nur die Implementierung der [paint](#)-Methode zeigen. Diese könnte dann beispielsweise in das folgende Programm eingebettet werden (auf der CD haben diese Dateien die Erweiterung [.inc](#)):

Hinweis

[GrafikBeispiel.java](#)

```

001 /* GrafikBeispiel.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class GrafikBeispiel
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         GrafikBeispiel wnd = new GrafikBeispiel();
012     }
013
014     public GrafikBeispiel()
015     {
016         super("GrafikBeispiel");
017         addWindowListener(
018             new WindowAdapter() {
019                 public void windowClosing(WindowEvent event)
020                 {
021                     System.exit(0);
022                 }
023             }
024         );
025         setBackground(Color.lightGray);
026         setSize(300,200);
027         setVisible(true);
028     }
029
030     public void paint(Graphics g)
031     {
032         //wird in den folgenden Beispielen überlagert
033     }
034 }

```

Listing 14.2: Rahmenprogramm für nachfolgende Beispiele

Da die [paint](#)-Methode in diesem Programm noch keine Ausgabeoperationen enthält, erzeugt das Programm lediglich ein leeres Fenster mit dem Titel »Grafikbeispiel«:

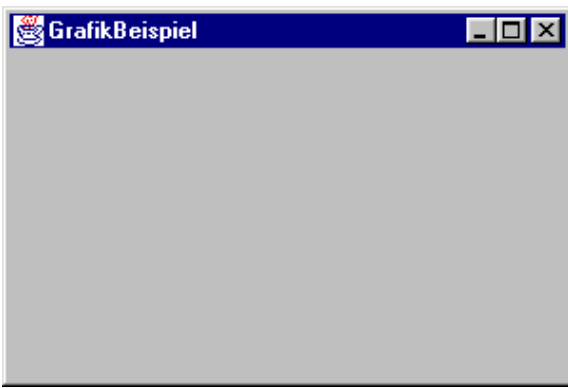


Abbildung 14.2: Ein einfaches Fenster

Das Beispielprogramm `GrafikBeispiel` ist so ungefähr der kleinstmögliche Rahmen, den man für ein AWT-basiertes Java-Programm vorgeben kann, wenn es ein einzelnes Fenster enthalten und beim Schließen desselben automatisch beendet werden soll. Es besteht aus folgenden Komponenten:

Hinweis

- Die Klasse `GrafikRahmen` ist aus `Frame` abgeleitet, um ein Top-Level-Window darzustellen.
- Sie enthält eine statische Methode `main`, die als Eintrittspunkt für das Programm dient und das Fensterobjekt erzeugt.
- Der Konstruktor der Klasse enthält drei Methodenaufrufe, `setBackground`, `setSize` und `setVisible`, um die Farbe des Fensterhintergrunds und seine Größe einzustellen und das Fenster auf dem Bildschirm sichtbar zu machen.
- Weiterhin enthält der Konstruktor einen Aufruf von `addWindowListener`, mit dem eine *anonyme Klasse*, die das Interface `WindowAdapter` implementiert, registriert wird. Diese überlagert die Methode `windowClosing` und sorgt dafür, daß beim Schließen des Fensters das Programm beendet wird. Wir werden auf die Details dieser Technik und die verschiedenen Varianten, auf Fensterereignisse zu reagieren, in [Kapitel 18](#) noch näher eingehen.
- Schließlich gibt es eine Methode `paint`, die hier noch leer ist, aber in den nachfolgenden Beispielen dieses Kapitels den Aufruf verschiedener Grafikroutinen zeigt.

14.3.1 Linie

[java.awt.Graphics](#)

```
public void drawLine(int x1, int y1, int x2, int y2)
```

Zieht eine Linie von der Position `(x1,y1)` zur Position `(x2,y2)`. Beide Punkte dürfen an beliebiger Stelle im Fenster liegen, das Einhalten einer bestimmten Reihenfolge ist nicht erforderlich. Teile der Ausgabe, die außerhalb des darstellbaren Bereichs liegen, werden, wie in grafikorientierten Systemen üblich, unterdrückt.

Das folgende Beispiel zeichnet eine Reihe von gleich hohen Linien, deren horizontaler Abstand durch einen Zufallszahlengenerator bestimmt wird. Das Ergebnis hat dadurch Ähnlichkeit mit einem Barcode (ist aber keiner):

Beispiel

[Linien.inc](#)

```
001 /* Linien.inc */
002
003 public void paint(Graphics g)
004 {
005     int i;
006     int x = 80;
007
008     for (i=0; i<60; ++i) {
009         g.drawLine(x,40,x,100);
010         x += 1+3*Math.random();
011     }
012 }
```

Listing 14.3: Ausgabe von Linien



Abbildung 14.3: Ausgabe von Linien

Die Methode [drawLine](#) zeichnet Linien grundsätzlich mit einer Dicke von einem Pixel. Die hier angezeigten unterschiedlich breiten Linien kommen dadurch zustande, daß zwei oder mehr Linien direkt nebeneinander ausgegeben werden. Das ist die einfachste Möglichkeit, Linien darzustellen, die dicker als 1 Pixel sind. Leider gibt es keine einfache Möglichkeit, gestrichelte oder gepunktete Linien zu zeichnen oder ein selbstdefiniertes Füllmuster zu verwenden.

Hinweis

14.3.2 Rechteck

[java.awt.Graphics](#)

```
public void drawRect(int x, int y, int width, int height)
```

Zeichnet ein Rechteck der Breite `width` und der Höhe `height`, dessen linke obere Ecke an der Position `(x,y)` liegt. Eine größere Breite dehnt das Rechteck nach rechts aus, eine größere Höhe nach unten.

Eine Variante von [drawRect](#) ist die Methode [drawRoundRect](#):

[java.awt.Graphics](#)

```
public void drawRoundRect(
    int x, int y,
    int width, int height,
    int arcWidth, int arcHeight
)
```

Gegenüber [drawRect](#) sind hier die Parameter `arcWidth` und `arcHeight` dazugekommen. Sie bestimmen den horizontalen und vertikalen Radius des Ellipsenabschnitts, der zur Darstellung der runden »Ecke« verwendet wird.

Das folgende Beispiel zeichnet eine Kette von nebeneinanderliegenden Rechtecken, deren Größe durch einen Zufallszahlengenerator bestimmt wird. Der Zufallszahlengenerator entscheidet auch, ob ein Rechteck an der Ober- oder Unterseite seines Vorgängers festgemacht wird:

Beispiel

[Rechtecke.inc](#)

```
001 /* Rechtecke.inc */
002
003 public void paint(Graphics g)
004 {
005     int x = 10, y = 80;
006     int sizex, sizey = 0;
007
008     while (x < 280 && y < 180) {
009         sizex = 4 + (int) (Math.random() * 9);
010         if (Math.random() > 0.5) {
011             y += sizey;
012             sizey = 4 + (int) (Math.random() * 6);
013         } else {
014             sizey = 4 + (int) (Math.random() * 6);
015             y -= sizey;
016         }
017         g.drawRect(x,y,sizex,sizey);
018         x += sizex;
019     }
020 }
```

Listing 14.4: Ausgabe von Rechtecken

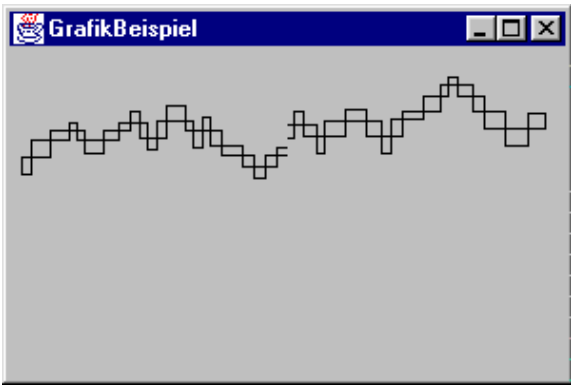


Abbildung 14.4: Ausgabe von Rechtecken

14.3.3 Polygon

Mit Hilfe der Methode [drawPolygon](#) ist es möglich, Linienzüge zu zeichnen, bei denen das Ende eines Elements mit dem Anfang des jeweils nächsten verbunden ist:

```
public void drawPolygon(int arx[], int ary[], int cnt)
```

[drawPolygon](#) erwartet drei Parameter. Der erste ist ein Array mit einer Liste der *x*-Koordinaten und der zweite ein Array mit einer Liste der *y*-Koordinaten. Beide Arrays müssen so synchronisiert sein, daß ein Paar von Werten an derselben Indexposition immer auch ein Koordinatenpaar ergibt. Die Anzahl der gültigen Koordinatenpaare wird durch den dritten Parameter festgelegt.

Im Gegensatz zum JDK 1.0 wird das Polygon nach Abschluß der Ausgabe automatisch geschlossen. Falls der erste und der letzte Punkt nicht identisch sind, werden diese durch eine zusätzliche Linie miteinander verbunden. Soll dagegen ein nichtgeschlossenes Polygon gezeichnet werden, so kann dazu die Methode [drawPolyline](#) verwendet werden:

Hinweis

```
public void drawPolyline(int arx[], int ary[], int cnt)
```

Eine zweite Variante, Polygone zu zeichnen, besteht darin, zunächst ein Objekt der Klasse [Polygon](#) zu konstruieren und dieses dann an [drawPolygon](#) zu übergeben. [Polygon](#) besitzt zwei Konstruktoren, von denen einer parameterlos ist und der andere dieselben Parameter wie die oben beschriebene Methode [drawPolygon](#) besitzt:

```
public void Polygon()
public void Polygon(int arx[], int ary[], int cnt)
```

Mit Hilfe der Methode [addPoint](#) kann ein Polygon um weitere Punkte erweitert werden. Schließlich kann das fertige Polygon an die Methode [drawPolygon](#) übergeben werden, die dann wie folgt aufzurufen ist:

```
public void drawPolygon(Polygon p)
```

Das folgende Beispiel gibt den Buchstaben »F« mit Hilfe eines geschlossenen Polygons aus:

Beispiel

```
001 /* Polygon.inc */
002
003 public void paint(Graphics g)
004 {
005     int arx[] = {50,50,120,120,80,80,100,100,80,80};
006     int ary[] = {170,40,40,70,70,100,100,130,130,170};
007
008     g.drawPolygon(arx,ary,arx.length);
009 }
```

Listing 14.5: Ausgabe eines Polygons

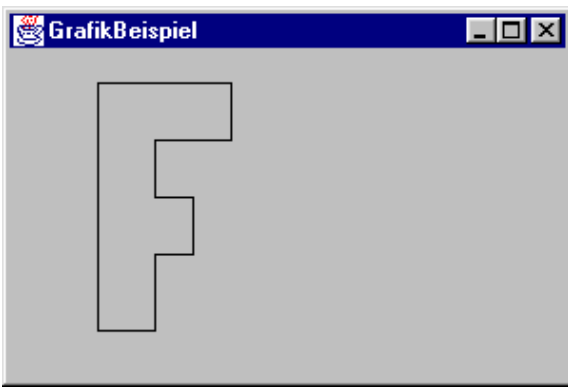


Abbildung 14.5: Ausgabe eines Polygons

An diesem Beispiel läßt sich ein potentielles Problem bei der Angabe von *x*- und *y*-Koordinaten zur Bestimmung einzelner Punkte in einem Fenster erkennen. Obwohl das Fenster durch Aufruf der Methode [setSize](#) eine Größe von 300*200 Pixeln hat, ist die untere Kante des »F« bereits sehr viel näher am unteren Rand des Fensters, als dessen *y*-Wert von 170 vermuten läßt, insbesondere, wenn man bedenkt, daß der obere Rand des Buchstabens 50 Pixel vom Fensterrand entfernt ist. Der Grund dafür ist, daß die Größenangabe für [setSize](#) die Größe des kompletten Fensters festlegt und damit auch den erforderlichen Platz für die Titelleiste und gegebenenfalls das Menü einschließt. Der zur Ausgabe zur Verfügung stehende Client-Bereich ist daher in aller Regel deutlich kleiner. Wir werden später noch lernen, wie die *exakte* Größe des Client-Bereichs bestimmt werden kann.

Warnung

14.3.4 Kreis

Die [Graphics](#)-Klasse von Java erlaubt sowohl das Zeichnen von Kreisen als auch von Ellipsen und Kreisabschnitten. Ein Kreis wird dabei als Verallgemeinerung einer Ellipse angesehen, und beide Objekte werden mit der Methode [drawOval](#) gezeichnet:

[java.awt.Graphics](#)

```
public void drawOval(int x, int y, int width, int height)
```

Anders als in anderen Grafiksystemen werden bei dieser Methode nicht der Mittelpunkt und der Radius des Kreises angegeben, sondern die übergebenen Parameter spezifizieren ein Rechteck der Größe *width* und *height*, dessen linke obere Ecke an der Position (*x*, *y*) liegt. Gezeichnet wird dann der größte Kreis, der vollständig in das Rechteck hineinpaßt.

Hinweis

Daß diese Vorgehensweise zwar untypisch ist, aber durchaus ihre Vorteile haben kann, zeigt das folgende Beispiel:

Beispiel

[Kreise.inc](#)

```
001 /* Kreise.inc */
002
003 public void paint(Graphics g)
004 {
005     int r = 8;
006     int i, j;
007     int x, y;
008
009     for (i=1; i<=10; ++i) {
010         x = 150 - r * i;
011         y = (int) (40 + (i - 1) * 1.7321 * r);
012         for (j=1; j<=i; ++j) {
013             g.drawOval(x,y,2*r,2*r);
014             x += 2 * r;
015         }
016     }
017 }
```

Listing 14.6: Ausgabe von Kreisen

Das Programm gibt dabei eine Pyramide von Kreisen aus. Da die an [drawOval](#) übergebenen Parameter bereits das umgebende Rechteck bezeichnen, kann die Figur wie eine Pyramide aus Rechtecken dargestellt werden:

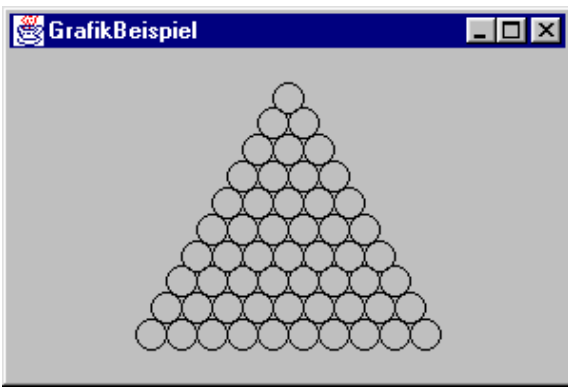


Abbildung 14.6: Ausgabe von Kreisen

14.3.5 Kreisbogen

Ein Kreisbogen ist ein zusammenhängender Abschnitt der Umfangslinie eines Kreises. Er kann mit der Methode [drawArc](#) gezeichnet werden:

```

public void drawArc(
    int x, int y, int width, int height,
    int startAngle, int arcAngle
)

```

Die ersten vier Parameter bezeichnen dabei den Kreis bzw. die Ellipse so, wie dies auch bei [drawOval](#) der Fall war. Mit [startAngle](#) wird der Winkel angegeben, an dem mit dem Kreisabschnitt begonnen werden soll, und [arcAngle](#) gibt den zu überdeckenden Bereich an. Dabei bezeichnet ein Winkel von 0 Grad die 3-Uhr-Position, und positive Winkel werden *entgegen* dem Uhrzeigersinn gemessen. Als Einheit wird *Grad* verwendet und nicht das sonst übliche Bogenmaß.

Das folgende Beispiel zeichnet durch wiederholten Aufruf der Methode [drawArc](#) eine Ellipse mit einer gestrichelten Umfangslinie. Ein Aufruf zeichnet dabei jeweils 4 Grad der Ellipse und läßt dann eine Lücke von 3 Grad, bevor das nächste Stück gezeichnet wird:

Beispiel

```

001 /* KreisBoegen.inc */
002
003 public void paint(Graphics g)
004 {
005     int line = 4;
006     int gap = 3;
007     int angle = 0;
008
009     while (angle < 360) {
010         g.drawArc(20,40,250,140,angle,line);
011         angle += gap + line;
012     }
013 }

```

[KreisBoegen.inc](#)

Listing 14.7: Ausgabe von Kreisbögen

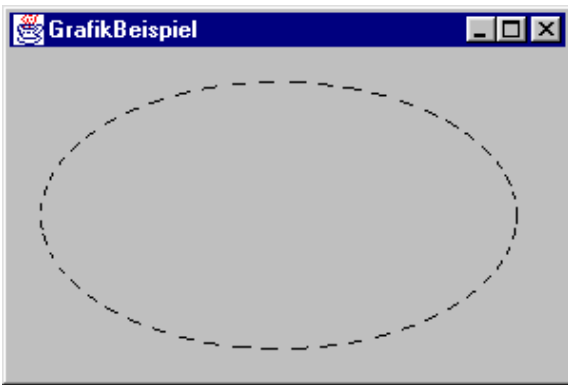


Abbildung 14.7: Ausgabe von Kreisbögen

14.4 Weiterführende Funktionen

- [14.4 Weiterführende Funktionen](#)
 - [14.4.1 Linien- oder Füllmodus](#)
 - [14.4.2 Kopieren und Löschen von Flächen](#)
 - [14.4.3 Die Clipping-Region](#)

14.4.1 Linien- oder Füllmodus

Mit Ausnahme von [drawLine](#) können alle vorgestellten Routinen entweder im *Linien-* oder im *Füllmodus* verwendet werden. Beginnt der Methodenname mit [draw](#), erfolgt die Ausgabe im Linienmodus. Es wird lediglich der Umriß der jeweiligen Figur gezeichnet. Beginnt der Methodenname mit [fill](#), erfolgt die Ausgabe im Füllmodus, und das Objekt wird mit ausgefüllter Fläche gezeichnet.

Nachfolgend noch einmal eine Zusammenstellung der verfügbaren Füllfunktionen:

```

public void fillRect(int x, int y, int w, int h)
public void fillRoundRect(
    int x, int y, int w, int h, int xr, int yr
)
public void fillPolygon(int arx[], int ary[], int cnt)
public void fillPolygon(Polygon p)
public void fillOval(int x, int y, int width, int height)
public void fillArc(
    int x, int y, int width, int height,
    int startAngle, int arcAngle
)

```

Das nachfolgende Beispiel verwendet die Methoden [fillRect](#), [fillPolygon](#), [fillOval](#) und [fillArc](#), um einen stilisierten »Java«-Schriftzug anzuzeigen:

Beispiel

```

001 /* GefuellteFlaechen.inc */
002
003 public void paint(Graphics g)
004 {
005     int arx[] = {150,175,200,150};
006     int ary[] = {100,150,100,100};
007
008     //---J
009     g.fillRect(70,40,20,80);
010     g.fillArc(30,90,60,60,225,180);
011     //---a
012     g.fillOval(100,100,40,50);
013     g.fillRect(120,100,20,50);
014     //---v
015     g.fillPolygon(arx,ary,arx.length);
016     //---a
017     g.fillOval(210,100,40,50);
018     g.fillRect(230,100,20,50);
019 }

```

Listing 14.8: Ausgabe von gefüllten Flächen

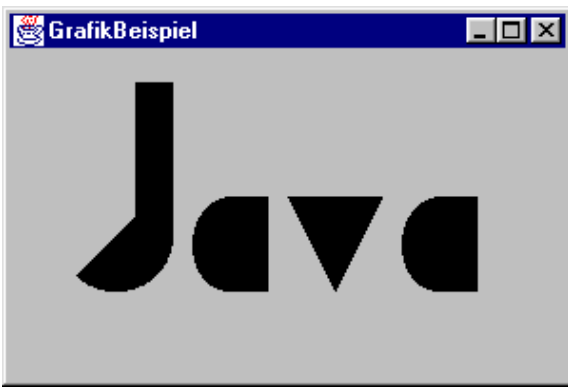


Abbildung 14.8: Ausgabe von gefüllten Flächen

Man kann sich die Wirkungsweise der verschiedenen Methodenaufrufe etwas deutlicher machen, wenn man sie nicht im Füll-, sondern im Linienmodus arbeiten läßt. Dies ist auch eine nützliche Vorgehensweise beim Erstellen von komplexeren Grafiken. In unserem Beispiel würde die Ausgabe des Programms dann so aussehen:

Tip

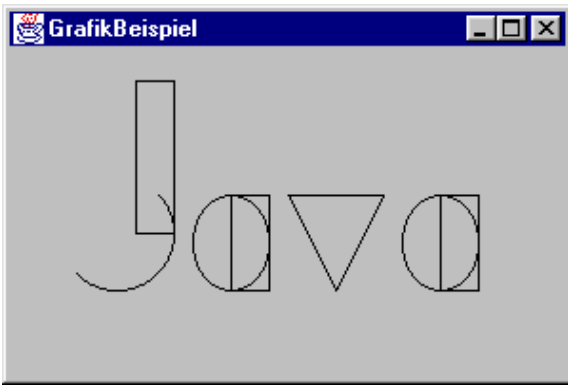


Abbildung 14.9: Ausgabe des Java-Logos als Liniengrafik

14.4.2 Kopieren und Löschen von Flächen

Die Klasse [Graphics](#) stellt auch einige Methoden zum Bearbeiten bereits gezeichneter Flächen zur Verfügung. Diese erlauben es beispielsweise, einen rechteckigen Ausschnitt des Ausgabefensters zu löschen oder zu kopieren:

```

public void clearRect(
    int x, int y,
    int width, int height
)

public void copyArea(
    int x, int y,
    int width, int height,
    int dx, int dy
)

```

[java.awt.Graphics](#)

Die Methode [clearRect](#) löscht das angegebene Rechteck, indem sie den Bereich mit der aktuellen Hintergrundfarbe ausfüllt. (x,y) bezeichnet die linke obere Ecke, $(width,height)$ die Breite und Höhe des Fensters.

Soll ein Teil des Fensters kopiert werden, kann dazu die Methode [copyArea](#) verwendet werden. Die ersten vier Parameter bezeichnen in der üblichen Weise den zu kopierenden Ausschnitt. (dx,dy) gibt die Entfernung des Zielrechtecks in x- und y-Richtung an. [copyArea](#) kopiert das ausgewählte Rechteck also immer an die Position $(x+dx, y+dy)$.

Das folgende Beispiel demonstriert die Anwendung von [copyArea](#) anhand eines Programms, das den Bildschirm (ähnlich dem Windows 95-Hintergrund »Labyrinth«) mit einem Muster füllt:

Beispiel

[Kopieren.inc](#)

```
001 /* Kopieren.inc */
002
003 public void paint(Graphics g)
004 {
005     int xorg = 4;
006     int yorg = 28;
007     int arx[] = {0,6,6,2,2,4,4,0,0};
008     int ary[] = {0,0,6,6,4,4,2,2,8};
009     for (int i = 0; i < arx.length; ++i) {
010         arx[i] += xorg;
011         ary[i] += yorg;
012     }
013     g.drawPolyline(arx,ary,arx.length);
014     for (int x = 0; x <= 300; x += 8) {
015         for (int y = 0; y <= 160; y += 8) {
016             if (x != 0 || y != 0) {
017                 g.copyArea(xorg,yorg,8,8,x,y);
018             }
019         }
020     }
021 }
```

Listing 14.9: Kopieren von Flächen mit copyArea

Hier wird zunächst ein einziges Exemplar des Musters in der Größe 8*8 Pixel in der linken oberen Ecke des Fensters gezeichnet. Alle weiteren Wiederholungen werden durch Kopieren hergestellt:

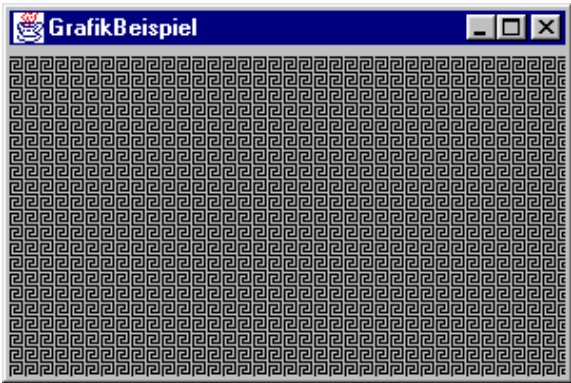


Abbildung 14.10: Kopieren von Grafiken

Wann die Anwendung der Methode [copyArea](#) in der Praxis sinnvoll ist, muß von Fall zu Fall entschieden werden. In unserem Beispiel wäre es sehr einfach gewesen, das Fenster auch ohne Kopieren mit dem Muster zu füllen, und der Overhead der Kopiermethode hätte vermieden werden können. Ist es dagegen sehr aufwendig, einen Teil des Fensters zu zeichnen, der an anderer Stelle repliziert werden soll, ist die Anwendung von [copyArea](#) sinnvoll.

Tip

14.4.3 Die Clipping-Region

Jeder Grafikkontext hat eine zugeordnete *Clipping-Region*, die dazu dient, die Ausgabe auf einen bestimmten Bereich einzugrenzen. So wird beispielsweise verhindert, daß wichtige Teile eines Fensters, wie z.B. der Rahmen oder die Menüzeile, von den Ausgabeoperationen des Programms überschrieben werden.

Auch in Java besitzt ein Fenster eine Clipping-Region, die an das übergebene [Graphics](#)-Objekt gebunden ist. Beim Aufruf der [paint](#)-Methode entspricht ihre Größe der zur Ausgabe zur Verfügung stehenden Fläche. Mit Hilfe der Methode [clipRect](#) kann die Clipping-Region nun sukzessive verkleinert werden:

[java.awt.Graphics](#)

```
public void clipRect(int x, int y, int width, int height)
```

Ein Aufruf dieser Methode begrenzt die Clipping-Region auf die Schnittmenge zwischen dem angegebenen Rechteck und ihrer vorherigen Größe. Da eine Schnittmenge niemals größer werden kann als eine der Mengen, aus denen sie gebildet wurde, kann `clipRect` also lediglich dazu verwendet werden, die Clipping-Region zu *verkleinern*.

Hinweis

Soll die Clipping-Region auf einen beliebigen Bereich innerhalb des aktuellen Fensters ausgedehnt werden (der dabei auch größer sein kann als die bisherige Clipping-Region), so kann dazu die Methode `setClip` verwendet werden, die in zwei verschiedenen Ausprägungen zur Verfügung steht:

`java.awt.Graphics`

```
public abstract void setClip(int x, int y, int width, int height)

public abstract void setClip(Shape clip)
```

Die erste Version übergibt ein Rechteck, das die Größe der gewünschten Clipping-Region in Client-Koordinaten angibt. Die zweite Variante erlaubt die Übergabe eines Objekts, welches das `Shape`-Interface implementiert. Das Interface besitzt derzeit jedoch nur die Methode `getBounds`, mit der das umschließende Rechteck ermittelt werden kann:

`java.awt.Shape`

```
public abstract Rectangle getBounds()
```

Da außer Rechtecken derzeit keine anders geformten `Shape`-Objekte zur Definition der Clipping-Region unterstützt werden, wollen wir hier nicht weiter auf die zweite Variante von `setClip` eingehen.

Soll die Ausdehnung der aktuellen Clipping-Region ermittelt werden, so kann dazu eine der Methoden `getClip` oder `getClipBounds` verwendet werden:

`java.awt.Graphics`

```
public abstract Shape getClip()

public abstract Rectangle getClipBounds()
```

Dabei liefert `getClip` ein `Shape`-Objekt, das derzeit denselben Einschränkungen unterliegt wie das beim Aufruf von `setClip` übergebene. `getClipBounds` liefert das kleinste Rechteck, das die aktuelle Clipping-Region vollständig umschließt.

Das folgende Beispiel wiederholt die Ausgabe des Java-Logos, grenzt aber vorher die Clipping-Region auf ein Fenster der Größe 150*80 Pixel ein, das seinen Ursprung an Position (50,50) hat:

Beispiel

`Clipping.inc`

```
001 /* Clipping.inc */
002
003 public void paint(Graphics g)
004 {
005     int arx[] = {150,175,200,150};
006     int ary[] = {100,150,100,100};
007
008     g.setClip(50,50,150,80);
009     //---J
010     g.fillRect(70,40,20,80);
011     g.fillArc(30,90,60,60,225,180);
012     //---a
013     g.fillOval(100,100,40,50);
014     g.fillRect(120,100,20,50);
015     //---v
016     g.fillPolygon(arx,ary,arx.length);
017     //---a
018     g.fillOval(210,100,40,50);
019     g.fillRect(230,100,20,50);
020 }
```

Listing 14.10: Verwendung der Clipping-Funktionen

Die Ausgabe des Programms erfolgt nun nur noch innerhalb der Clipping-Region:

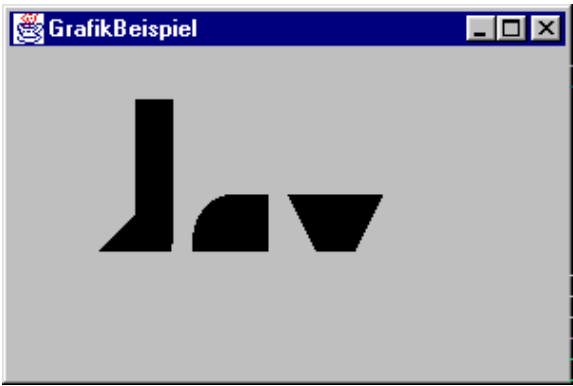


Abbildung 14.11: Verwendung der Clipping-Funktionen

14.5 Zusammenfassung

- [14.5 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Das Abstract Windowing Toolkit und das Paket [java.awt](#)
- Grundlagen der Grafikausgabe und Anlegen eines Fensters
- Die Methode [paint](#) und die Klasse [Graphics](#)
- Das grafische Koordinatensystem von Java
- Die Methoden [drawLine](#), [drawRect](#), [drawPolygon](#), [drawPolyline](#), [drawOval](#) und [drawArc](#) der Klasse [Graphics](#) zur Ausgabe von Linien, Rechtecken, Polygonen, Kreisen und Kreisbögen
- Die Bedeutung von Linien- und Füllmodus
- Die Methoden [fillRect](#), [fillRoundRect](#), [fillPolygon](#), [fillOval](#) und [fillArc](#) zum Erzeugen von gefüllten Flächen
- Kopieren und Löschen von Flächen mit den Methoden [clearRect](#) und [copyArea](#)
- Die Clipping-Region und die Methoden [clipRect](#), [setClip](#), [getClip](#) und [getBounds](#)

Kapitel 15

Textausgabe

- [15 Textausgabe](#)
 - [15.1 Ausgabefunktionen](#)
 - [15.2 Unterschiedliche Schriftarten](#)
 - [15.2.1 Font-Objekte](#)
 - [15.2.2 Standardschriftarten](#)
 - [15.3 Eigenschaften von Schriftarten](#)
 - [15.3.1 Font-Informationen](#)
 - [15.3.2 Font-Metriken](#)
 - [15.4 Drucken in Java](#)
 - [15.4.1 Grundlagen](#)
 - [15.4.2 Seitenweise Ausgabe](#)
 - [15.4.3 Plazierung des Codes zur Druckausgabe](#)
 - [15.5 Zusammenfassung](#)

15.1 Ausgabefunktionen

- 15.1 Ausgabefunktionen

Neben den Methoden zur Ausgabe von Linien- oder Flächengrafiken gibt es in Java die Möglichkeit, *Text* in einem Fenster auszugeben. Die dafür vorgesehenen Methoden [drawString](#), [drawChars](#) und [drawBytes](#) gehören ebenfalls zur Klasse [Graphics](#):

[java.awt.Graphics](#)

```
public void drawString(
    String str, int x, int y
)

public void drawChars(
    char data[], int offset, int length, int x, int y
)

public void drawBytes(
    byte data[], int offset, int length, int x, int y
)
```

Die einfachste Methode, Text auszugeben, besteht darin, [drawString](#) aufzurufen und dadurch den String *str* im Grafikfenster an der Position (*x,y*) auszugeben. Das Koordinatenpaar (*x,y*) bezeichnet dabei das linke Ende der *Basislinie* des ersten Zeichens in *str*. Die Bedeutung der Basislinie wird weiter unten bei der Beschreibung der Font-Metriken erläutert.

In dem folgenden Beispiel wird versucht, einen String, der die Größe der Client-Area angibt, zentriert im Ausgabefenster auszugeben (wir verwenden immer noch [Listing 14.2](#) aus [Kapitel 14](#)):

Beispiel

[Textausgabe.inc](#)

```
001 /* Textausgabe.inc */
002
003 public void paint(Graphics g)
004 {
005     int maxX=getSize().width-getInsets().left-getInsets().right;
006     int maxY=getSize().height-getInsets().top-getInsets().bottom;
007
008     g.drawString(
009         "Die Client-Area ist "+maxX+"*"+maxY+" Pixel groß",
010         getInsets().left + maxX/2,
011         getInsets().top + maxY/2
012     );
013 }
```

Listing 15.1: Einfache Textausgabe im Grafikfenster



Abbildung 15.1: Einfache Textausgabe

Man kann an diesem Beispiel gut sehen, wie die Größe der Client-Area ermittelt werden kann. Zunächst wird durch Aufruf von [getSize](#) die Größe des gesamten Fensters bestimmt. Anschließend werden die Teile des Fensters herausgerechnet, die nicht der Client-Area zur Verfügung stehen, nämlich der Rahmen und die Titelzeile. Auf diese Informationen kann mit Hilfe der Methode [getInsets](#) zugegriffen werden. Sie gibt die Abmessungen der Elemente an, die um die Client-Area herum platziert sind.

[Abbildung 15.2](#) erweitert das in [Kapitel 14](#) erläuterte Koordinatensystem des AWT um die erwähnten Randelemente. Dabei ist insbesondere zu erkennen, daß der Punkt (0,0) nicht den Ursprung der Client-Area bezeichnet, sondern die linke obere Ecke des Fensters. Der Ursprung der Client-Area hat die *x*-Koordinate `getInsets().left` und die *y*-Koordinate `getInsets().top`.

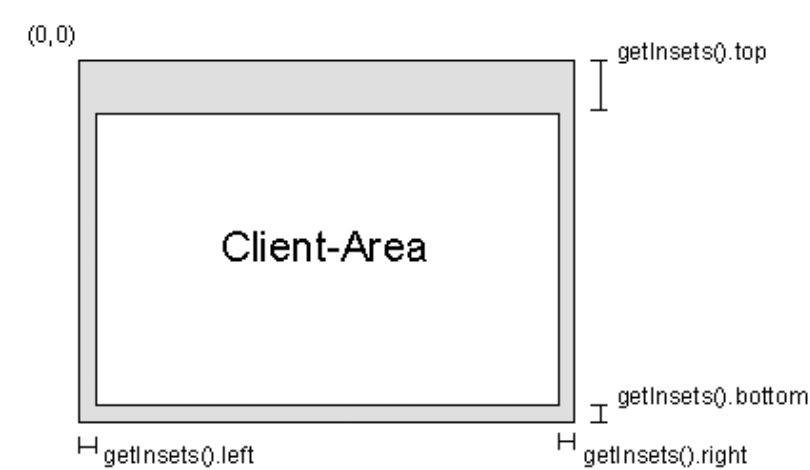


Abbildung 15.2: Die Randelemente eines Fensters

Leider hat das Programm einen kleinen Schönheitsfehler, denn der Mittelpunkt des Fensters wird zur Ausgabe des *ersten* Zeichens verwendet, obwohl dort eigentlich der Mittelpunkt des gesamten Strings liegen sollte. Wir werden dieses Beispiel weiter unten komplettieren, wenn uns mit Hilfe von *Font-Metriken* die Möglichkeit zur Verfügung steht, die Breite und Höhe eines Strings zu messen.

Hinweis

Die Methode `drawString` ist leider nicht besonders vielseitig, denn sie kann Text weder drehen noch clippen. Sie interpretiert auch keine eingebetteten Formatierungen, wie beispielsweise einen Zeilenumbruch oder einen Tabulator. Da es im Windows-API all diese und noch weitere Möglichkeiten gibt, zeigt sich hier wie auch an anderen Stellen des AWT, daß eine Java-Portierung in der Regel nicht alle Funktionen des darunterliegenden Betriebssystems zur Verfügung stellen kann. Das *2D-API* des JDK 1.2 stellt darüber hinaus viele zusätzliche Grafikfunktionen plattformübergreifend zur Verfügung.

Die Methoden `drawChars` und `drawBytes` sind leichte Variationen von `drawString`. Anstelle eines Strings erwarten sie ein Array von Zeichen bzw. Bytes als Quelle für den auszugebenden Text. Mit `offset` und `length` stehen zwei zusätzliche Parameter zur Verfügung, die zur Angabe der Startposition bzw. der Anzahl der auszugebenden Zeichen verwendet werden können.

15.2 Unterschiedliche Schriftarten

- 15.2 Unterschiedliche Schriftarten
 - 15.2.1 Font-Objekte
 - 15.2.2 Standardschriftarten

15.2.1 Font-Objekte

Führt man die Schriftausgabe wie bisher besprochen durch, werden die Texte in einem systemabhängigen Standard-Font ausgegeben. Soll ein anderer Font zur Textausgabe verwendet werden, so muß zunächst ein passendes Objekt der Klasse `Font` erzeugt und dem verwendeten `Graphics`-Objekt zugewiesen werden:

```
public void setFont(Font font)

public Font getFont()
```

Die Methode `setFont` wird dazu verwendet, das `Font`-Objekt `font` in den Grafikkontext einzutragen, und mit `getFont` kann der aktuelle Font abgefragt werden.

Das Erzeugen neuer `Font`-Objekte wird über die drei Parameter `name`, `style` und `size` des Konstruktors der Klasse `Font` gesteuert:

```
public Font(String name, int style, int size)
```

Der Parameter `name` gibt den Namen des gewünschten Fonts an. In allen Java-Systemen sollten die Namen `SansSerif` (früher `Helvetica`), `Serif` (früher `TimesRoman`) und `Monospaced` (früher `Courier`) unterstützt werden. Sie stehen für die systemspezifischen Proportionalzeichensätze der Familien `Helvetica` und `TimesRoman` bzw. für die nichtproportionale Familie `Courier`. Unter Windows werden diese Standardnamen auf die True-Type-Fonts `Arial`, `Times New Roman` und `Courier New` abgebildet.

Der zweite und dritte Parameter des Konstruktors sind beide vom Typ `int`. Ein beliebter Fehler besteht darin, beide zu verwechseln und so die Angaben für die Größe und die Textattribute zu vertauschen. Leider kann der Fehler vom Compiler nicht gefunden werden, sondern wird frühestens zur Laufzeit entdeckt. Selbst dann wird er leicht mit dem Fall verwechselt, daß die gewünschten Schriftarten bzw. Attribute auf dem Zielsystem nicht installiert sind. Beim Erzeugen von `Font`-Objekten ist also einige Vorsicht geboten.

Warnung

Schriften sind generell recht unportabel, deshalb ist bei ihrer Verwendung Vorsicht angebracht. Insbesondere bei der Verwendung von systemspezifischen Schriftarten kann es sein, daß der Font-Mapper eines anderen Systems eine völlig verkehrte Schrift auswählt und die Ausgabe des Programms dann unbrauchbar wird. Werden nur die genannten Standardschriften verwendet, so sollte die Schriftausgabe auf allen unterstützten Java-Systemen zumindest lesbar bleiben. Die alten Schriftnamen `Helvetica`, `TimesRoman` und `Courier` aus dem JDK 1.0 werden zwar noch unterstützt, sind aber als *deprecated* gekennzeichnet und sollten daher nicht mehr verwendet werden.

Warnung

Der Parameter `style` wird verwendet, um auszuwählen, ob ein Font in seiner Standardausprägung, fett oder kursiv angezeigt werden soll. Java stellt dafür die in [Tabelle 15.1](#) aufgeführten numerischen Konstanten zur Verfügung. Die Werte `BOLD` und `ITALIC` können auch gemeinsam verwendet werden, indem beide Konstanten addiert werden.

Name	Wert	Bedeutung
<code>Font.PLAIN</code>	0	Standard-Font
<code>Font.BOLD</code>	1	Fett
<code>Font.ITALIC</code>	2	Kursiv

Tabelle 15.1: Style-Parameter

Der dritte Parameter des Konstruktors gibt die Größe der gewünschten Schriftart in Punkt an. Übliche Punktgrößen für die Ausgabe von Text sind 10 oder 12 Punkt.

Das folgende Beispiel gibt die drei Standardschriften in 36 Punkt aus:

Beispiel

[Schriften.inc](#)

```
001 /* Schriften.inc */
002
003 public void paint(Graphics g)
004 {
005     Font font;
006     String arfonts[] = {"Serif","SansSerif","Monospaced"};
007
008     for (int i=0; i<arfonts.length; ++i) {
009         font = new Font(arfonts[i],Font.PLAIN,36);
010         g.setFont(font);
011         g.drawString(arfonts[i],10,30 + (i+1)*(36+5));
012     }
013 }
```

Listing 15.2: Ausgabe verschiedener Schriften



Abbildung 15.3: Ausgabe verschiedener Fonts

Die Abbildung von Schriftnamen im JDK auf die dazu passenden Schriften des Betriebssystems wird durch die Datei [font.properties](#) im `lib`-Verzeichnis des JDK gesteuert, die zur Laufzeit vom AWT interpretiert wird. Werden hier Anpassungen vorgenommen, so ändert sich die Darstellung der Standardschriften.

Hinweis

15.2.2 Standardschriftarten

Es gibt im AWT eine Klasse [Toolkit](#), die als Hilfsklasse bei der Abbildung der portablen AWT-Eigenschaften dient. [Toolkit](#) ist standardmäßig abstrakt, wird aber von jeder AWT-Implementierung konkretisiert und den Anwendungen über die Klassenmethode [getDefaultToolkit](#) zur Verfügung gestellt. Während die meisten Methoden von [Toolkit](#) nur für Implementatoren von AWT-Portierungen von Interesse sind, gibt es auch einige Methoden, die in der Anwendung sinnvoll verwendet werden können. Eine dieser Methoden ist [getFontList](#):

[java.awt.Toolkit](#)

```
public String[] getFontList()
```

Diese Methode liefert ein Array von Strings mit den Namen der Standardschriftarten, die in der vorliegenden AWT-Implementierung verfügbar sind. Die folgende [paint](#)-Methode listet alle verfügbaren Standardschriften auf dem Bildschirm auf:

Beispiel

[Standardschriften.inc](#)

```
001 /* Standardschriften.inc */
002
003 public void paint(Graphics g)
004 {
005     Font font;
006     String arfonts[] = Toolkit.getDefaultToolkit().getFontList();
007
008     for (int i=0; i<arfonts.length; ++i) {
009         font = new Font(arfonts[i],Font.PLAIN,36);
010         g.setFont(font);
011         g.drawString(arfonts[i],10,(i+1)*(36+5));
012     }
013 }
```

Listing 15.3: Auflistung aller Standardschriften

Im JDK 1.2 wurde die Methode `getFontList` als `deprecated` markiert. Sie wird ersetzt durch die Methode `getAvailableFontFamilyNames` der Klasse `GraphicsEnvironment`, die im JDK 1.2 dazu dient, die verfügbaren Grafikgeräte und -konfigurationen zu beschreiben. Mit der statischen Methode `getLocalGraphicsEnvironment` kann das aktuelle `GraphicsEnvironment`-Objekt beschafft werden.

JDK1.1/1.2

`java.awt.GraphicsEnvironment`

```
static GraphicsEnvironment getLocalGraphicsEnvironment()  
  
String[] getAvailableFontFamilyNames()
```

Das vorige Beispiel müßte also wie folgt geändert werden:

`Standardschriften12.inc`

```
001 /* Standardschriften12.inc */  
002  
003 public void paint(Graphics g)  
004 {  
005     Font font;  
006     GraphicsEnvironment ge =  
007         GraphicsEnvironment.getLocalGraphicsEnvironment();  
008     String arfonts[] = ge.getAvailableFontFamilyNames();  
009  
010     for (int i=0; i<arfonts.length; ++i) {  
011         font = new Font(arfonts[i],Font.PLAIN,36);  
012         g.setFont(font);  
013         g.drawString(arfonts[i],10,(i+1)*(36+5));  
014     }  
015 }
```

Listing 15.4: Auflistung aller Standardschriften im JDK 1.2

Die Ausgabe des Programms ist (man muß das Fenster etwas größer ziehen, damit alle Schriften angezeigt werden):



Abbildung 15.4: Liste der Standardschriften

Die letzte Schrift ist *ZapfDingbats*, eine Schrift mit vorwiegend grafischen Symbolen. Da ihr Zeichensatz anders organisiert ist als bei normalen Schriften, werden lediglich eckige Kästchen ausgegeben.

15.3 Eigenschaften von Schriftarten

- [15.3 Eigenschaften von Schriftarten](#)
 - [15.3.1 Font-Informationen](#)
 - [15.3.2 Font-Metriken](#)

15.3.1 Font-Informationen

Die Klasse [Font](#) besitzt Methoden, um Informationen über den aktuellen Font zu gewinnen:

[java.awt.Font](#)

```
public String getFamily()
public int getStyle()
public int getSize()
```

[getFamily](#) liefert den systemspezifischen Namen eines Fonts, [getStyle](#) den Style-Parameter und [getSize](#) die Größe des Fonts. Auf diese Weise läßt sich ein neues Font-Objekt als Variation eines bestehenden Fonts erzeugen. Das folgende Beispiel zeigt eine [paint](#)-Methode, die den zur Anzeige verwendeten Font bei jedem Aufruf um 1 Punkt vergrößert:

[SchriftGroesser.inc](#)

```
001 /* SchriftGroesser.inc */
002
003 public void paint(Graphics g)
004 {
005     Font font = getFont();
006
007     if (font.getSize() <= 64) {
008         setFont(
009             new Font(
010                 font.getFamily(),
011                 font.getStyle(),
012                 font.getSize() + 1
013             )
014         );
015     }
016     g.drawString("Hello, World",40,100);
017 }
```

Listing 15.5: Vergrößern der Schriftart

Dieses Beispiel hat eine Besonderheit, denn es werden nicht die Methoden [getFont](#) und [setFont](#) der Klasse [Graphics](#) aufgerufen, sondern diejenigen aus der Fensterklasse [Frame](#). Dadurch bleiben die Font-Informationen auch zwischen zwei Aufrufen von [paint](#) erhalten.

Hinweis

15.3.2 Font-Metriken

Oft benötigt man Informationen über die Größe eines einzelnen Zeichens oder eines kompletten Strings. Anders als in textbasierten Systemen ist die Größe von Schriftzeichen bei der Programmierung unter einer grafischen Oberfläche nämlich nicht konstant, sondern von der Art des Fonts, dem Ausgabegerät und vor allem von der Breite der Zeichen selbst abhängig.

Das Größenmodell von Java sieht für ein Textzeichen fünf unterschiedliche Maßzahlen vor (siehe [Abbildung 15.5](#)), die teilweise zu den im Schriftsatz verwendeten Begriffen kompatibel sind. Wichtigste Bezugsmarke für die verschiedenen Maße ist dabei die *Grundlinie* des Zeichens, die den oberen und unteren Teil voneinander trennt. Die *Unterlänge* gibt die Länge zwischen Grundlinie und unterer Begrenzung des Zeichens an. Die *Oberlänge* bezeichnet die Länge zwischen Grundlinie und oberem Rand des Zeichens. Die *Breite eines Zeichens* ist der Abstand vom linken Rand des Zeichens bis zum linken Rand des darauffolgenden Zeichens. Der *Zeilenabstand* ist der Abstand zwischen dem unteren Rand einer Zeile und dem oberen Rand der nächsten Zeile. Die *Höhe* ist die Summe aus Oberlänge, Unterlänge und Zeilenabstand.

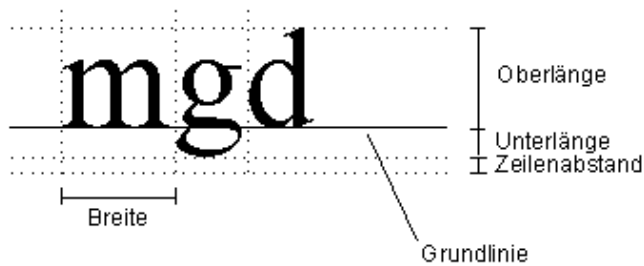


Abbildung 15.5: Größenmaßzahlen für Fonts in Java

Zur Bestimmung der Größeneigenschaften von Zeichen wird die Klasse [FontMetrics](#) verwendet. [FontMetrics](#) ist eine abstrakte Klasse, die nicht direkt instanziiert werden kann. Statt dessen wird sie durch Aufruf der Methode [getFontMetrics](#) aus dem Grafikkontext gewonnen:

```
public FontMetrics getFontMetrics(Font font)
    java.awt.Toolkit
```

```
public FontMetrics getFontMetrics()
```

Die parameterlose Variante dient dazu, Metriken zum aktuellen Font zu ermitteln, an die andere kann der zu untersuchende Font als Argument übergeben werden. Nun stehen Methoden zur Verfügung, die zur Bestimmung der genannten Eigenschaften aufgerufen werden können. Nachfolgend werden die wichtigsten von ihnen vorgestellt.

```
public int charWidth(char ch)
    java.awt.FontMetrics

public int stringWidth(String s)
```

Mit [charWidth](#) wird die Breite eines einzelnen Zeichens *ch* bestimmt, mit [stringWidth](#) die eines kompletten Strings. Der Rückgabewert dieser Methoden wird dabei stets in Bildschirmpixeln angegeben. Bei der Anwendung von [stringWidth](#) werden auch Unterschneidungen oder andere Sonderbehandlungen berücksichtigt, die bei der Ausgabe der Zeichenkette erfolgen würden.

```
public int getAscent()
    java.awt.FontMetrics

public int getDescent()

public int getHeight()

public int getLeading()
```

[getAscent](#) liefert die Oberlänge des Fonts, [getDescent](#) die Unterlänge, [getLeading](#) den Zeilenabstand und [getHeight](#) die Höhe. Obwohl diese Informationen für die meisten Zeichen des ausgewählten Fonts gültig sind, garantiert Java nicht, daß dies für alle Zeichen der Fall ist. Insbesondere kann es einzelne Zeichen geben, die eine größere Ober- oder Unterlänge haben. Zur Behandlung dieser Sonderfälle gibt es zusätzliche Methoden in [FontMetrics](#), die hier nicht näher behandelt werden sollen.

Das nachfolgende Beispiel zeigt eine [paint](#)-Methode, die einige Schriftzeichen in 72 Punkt Größe zusammen mit einem 10*10 Pixel großen Koordinatengitter ausgibt. Zusätzlich gibt das Programm die Font-Metriken aus:

Beispiel

```
001 /* Fontmetriken.inc */
002
003 public void paint(Graphics g)
004 {
005     Font font = new Font("TimesRoman",Font.PLAIN,72);
006
007     //---Linien
008     g.setColor(Color.blue);
009     for (int x = 10; x <= 260; x += 10) {
010         g.drawLine(x,30,x,130);
011     }
012     for (int y = 30; y <= 130; y += 10) {
013         g.drawLine(10,y,260,y);
014     }
015     //---Schrift
016     g.setColor(Color.black);
    Fontmetriken.inc
```

```

017 g.drawLine(0,100,270,100);
018 g.setFont(font);
019 g.drawString("mgdAW",10,100);
020 //---Font-Metriken
021 FontMetrics fm = getFontMetrics(font);
022 System.out.println("Oberlänge      = " + fm.getAscent());
023 System.out.println("Unterlänge     = " + fm.getDescent());
024 System.out.println("Höhe           = " + fm.getHeight());
025 System.out.println("Zeilenabstand = " + fm.getLeading());
026 System.out.println("---");
027 System.out.println("Breite(m)      = " + fm.charWidth('m'));
028 System.out.println("Breite(g)      = " + fm.charWidth('g'));
029 System.out.println("Breite(d)      = " + fm.charWidth('d'));
030 System.out.println("Breite(A)      = " + fm.charWidth('A'));
031 System.out.println("Breite(W)      = " + fm.charWidth('W'));
032 System.out.println("---");
033 }

```

Listing 15.6: Anzeige von Font-Metriken

Die Grafikausgabe des Programms ist wie folgt:

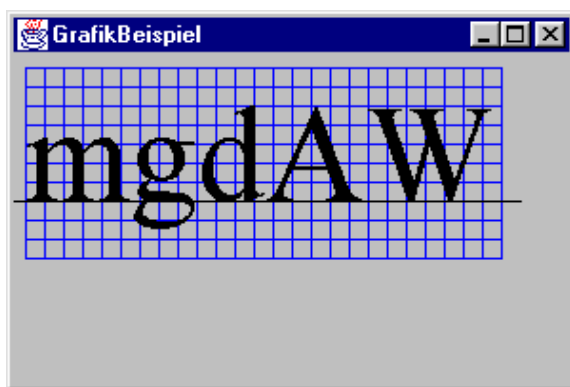


Abbildung 15.6: Anzeige von Font-Metriken

Zusätzlich werden die Font-Metriken in das Textfenster geschrieben:

```

Oberlänge      = 73
Unterlänge     = 16
Höhe           = 93
Zeilenabstand = 4
---
Breite(m)      = 55
Breite(g)      = 35
Breite(d)      = 36
Breite(A)      = 52
Breite(W)      = 68

```

Als weitere Anwendung der Font-Metriken wollen wir [Listing 15.1](#) wie versprochen komplettieren und den Text zentriert ausgeben:

[Zentriert.inc](#)

```

001 /* Zentriert.inc */
002
003 public void paint(Graphics g)
004 {
005     int maxX=getSize().width-getInsets().left-getInsets().right;
006     int maxY=getSize().height-getInsets().top-getInsets().bottom;
007     String s="Die Client-Area ist "+maxX+"*"+maxY+" Pixel groß";
008     FontMetrics fm = g.getFontMetrics();
009     int slen = fm.stringWidth(s);
010     g.drawString(
011         s,
012         getInsets().left + ((maxX - slen)/2),
013         getInsets().top + (maxY/2)
014     );

```

Beispiel

Die Programmausgabe ist:



Abbildung 15.7: Zentrierte Textausgabe

15.4 Drucken in Java

- [15.4 Drucken in Java](#)
 - [15.4.1 Grundlagen](#)
 - [15.4.2 Seitenweise Ausgabe](#)
 - [15.4.3 Plazierung des Codes zur Druckausgabe](#)

15.4.1 Grundlagen

Seit der Version 1.1 kann in Java gedruckt werden! Das ist deshalb bemerkenswert, weil es in den Vorgängerversionen keine Möglichkeit gab, dies zu tun. Im Laufe der Zeit entwickelten sich zwar einige inkompatible Erweiterungen, jedoch konnte sich keine von ihnen auf breiter Front durchsetzen.

Um so erfreulicher ist es, daß das Druck-API der Version 1.1 relativ übersichtlich gehalten wurde und leicht zu verstehen ist. Es gibt zwar einige Restriktionen und Besonderheiten, die beim Erstellen von Druckausgaben zu Problemen führen können, aber für einfache Ausdrücke von Text und Grafik ist die Schnittstelle recht gut geeignet.

Grundlage der Druckausgabe ist die Methode [getPrintJob](#) der Klasse [Toolkit](#):

```

public PrintJob getPrintJob(
    Frame frame,
    String jobtitle,
    Properties props
)

```

[java.awt.Toolkit](#)

Sie liefert ein Objekt des Typs [PrintJob](#), das zur Initialisierung eines Druckjobs verwendet werden kann. Ein Aufruf von [getPrintJob](#) führt gleichzeitig dazu, daß ein plattformspezifischer Druckdialog aufgerufen wird, der vom Anwender bestätigt werden muß. Bricht der Anwender den Druckdialog ab, liefert [getPrintJob](#) den Rückgabewert [null](#), andernfalls wird der Drucker initialisiert und die Ausgabe kann beginnen.

Die Klasse [PrintJob](#) stellt einige Methoden zur Verfügung, die für den Ausdruck benötigt werden:

```

public Graphics getGraphics()

public Dimension getPageDimension()

public int getPageResolution()

public abstract void end()

```

[java.awt.PrintJob](#)

Die wichtigste von ihnen ist [getGraphics](#). Sie liefert den Devicekontext zur Ausgabe auf den Drucker. Der Rückgabewert ist ein Objekt vom Typ [PrintGraphics](#), das aus [Graphics](#) abgeleitet ist und wie ein normaler Devicekontext verwendet werden kann, der beispielsweise an [paint](#) übergeben wird. Mit Hilfe des von [getGraphics](#) zurückgegebenen Devicekontexts können alle Grafik- und Textroutinen, die auch in [Graphics](#) zur Verfügung stehen, verwendet werden. Bezüglich der Verwendung von Farben gilt scheinbar, daß diese bei den linienbezogenen Ausgaberroutinen nicht unterstützt werden. Hier wird alles schwarz gezeichnet, was nicht den Farbwert (255, 255, 255) hat. Im Gegensatz dazu stellen die Füllfunktionen Farben (auf einem Schwarzweiß-Drucker) als Grauwerte dar. Dabei kann - je nach Druckertyp - auch Weiß eine Vollfarbe sein und dahinter liegende Objekte verdecken.

15.4.2 Seitenweise Ausgabe

Ein wichtiger Unterschied zu einem bildschirmbezogenen Devicekontext besteht darin, daß jeder Aufruf von [getGraphics](#) eine neue Druckseite beginnt. Die fertige Druckseite wird durch Aufruf von [dispose](#) an den Drucker geschickt. Für den Aufbau der Seite, das Ausgeben von Kopf- oder Fußzeilen, die Seitennumerierung und ähnliche Dinge ist die Anwendung selbst verantwortlich. Die Methode [end](#) ist aufzurufen, wenn der Druckjob beendet ist und alle Seiten ausgegeben wurden. Dadurch werden alle belegten Ressourcen freigegeben und die Druckerschnittstelle geschlossen.

Bei der Druckausgabe ist es wichtig zu wissen, wie groß die Abmessungen des Ausgabegeräts sind. Die hierzu angebotenen Methoden `getPageDimension` und `getPageResolution` sind in der aktuellen Version leider vollkommen unbrauchbar. `getPageResolution` liefert die tatsächliche Auflösung des Druckers in Pixel per Zoll (also z.B. 600 für einen Laserjet IV), während `getPageDimension` die Anzahl der Pixel liefert, die sich errechnet, wenn man ein Blatt Papier im US-Letter-Format (8,5 mal 11 Zoll) mit 72 dpi Auflösung darstellen würde. Leider erfolgt die Druckausgabe nicht mit 72 dpi, sondern in der aktuellen Bildschirmauflösung, wie sie von der Methode `getScreenResolution` der Klasse `Toolkit` geliefert wird. Da diese typischerweise bei 120 dpi liegt, füllen die von `getPageResolution` gelieferten Abmessungen nur etwa 60 % einer Seite.

Hinweis

Derzeit gibt es keine portable Lösung für dieses Problem. Ein Workaround besteht darin, die Papiergröße als fest anzunehmen (beispielsweise DIN A4 mit 21,0*29,7 cm), davon den nicht bedruckbaren Rand abzuziehen, das Ergebnis durch 2,54 (Anzahl cm je Zoll) zu teilen und mit der Auflösung von 120 dpi malzunehmen. Wir werden weiter unten ein Beispiel sehen, in dem diese Technik angewandt wird. Portabel ist sie allerdings nicht, denn das Programm muß Annahmen über die Papier- und Randgröße beisteuern. Es bleibt demnach zu hoffen, daß die nachfolgenden Versionen des JDK die Bestimmung der Abmessungen auf eine flexiblere Weise ermöglichen.

Durch die fixe Einstellung der Ausgabeauflösung ergibt sich ein weiteres Problem. So kann ein Drucker mit 600 dpi aus Java heraus nämlich nur mit der aktuellen Bildschirmauflösung (z.B. 120 dpi) angesteuert werden. Das bedeutet zwar nicht automatisch, daß Schriften oder schräge Linien mit Treppennustern dargestellt werden, denn sie werden meist als Vektorgrafiken an den Drucker übergeben. Allerdings können beispielsweise keine Pixelgrafiken in der aktuellen Druckerauflösung ausgegeben werden, denn die Positioniergenauigkeit eines einzelnen Pixels liegt bei 120 dpi. Eine Lösung für dieses Problem ist derzeit nicht bekannt.

Warnung

15.4.3 Plazierung des Codes zur Druckausgabe

Es gibt grundsätzlich zwei Möglichkeiten, die Druckausgabe im Programm zu plazieren. Einmal kann die normale `paint`-Methode dazu verwendet werden, sowohl Bildschirm- als auch Druckausgaben zu realisieren. Bei einem Aufruf der Methode `print` oder `printAll` der Klasse `Component` wird nämlich ein `PrintJob` erstellt, daraus der Grafikkontext beschafft und an `paint` übergeben:

`java.awt.Component`

```
public void print(Graphics g)
```

```
public void printAll(Graphics g)
```

Auf diese Weise kann bereits ohne zusätzliche Erweiterungen eine einfache Druckausgabe realisiert werden, die der Bildschirmausgabe relativ ähnlich sieht. Im Gegensatz zu `print` gibt `printAll` dabei nicht nur die aktuelle Komponente, sondern die komplette Container-Hierarchie eines komplexen Dialogs aus. Soll innerhalb von `paint` unterschieden werden, ob auf den Bildschirm oder den Drucker ausgegeben wird, so kann mit dem Ausdruck `g instanceof PrintGraphics` das übergebene `Graphics`-Objekt `g` auf Zugehörigkeit zur Klasse `PrintGraphics` getestet werden.

Die zweite Möglichkeit, die Druckausgabe zu plazieren, besteht darin, eine eigene Methode zu schreiben, die nur für die Ausgabe auf den Drucker verantwortlich ist. Diese könnte zunächst den `PrintJob` und das `PrintGraphics`-Objekt beschaffen und anschließend die Abmessungen der Ausgabefläche wie oben besprochen bestimmen. Die Methode wäre dann nicht darauf angewiesen, daß sie sowohl für die Bildschirm- als auch für die Druckausgabe vernünftige Resultate liefern muß, sondern könnte ihre Ausgaben für die Druckausgabe optimieren. Der Nachteil bei dieser Lösung ist allerdings, daß möglicherweise Code zum Erstellen der Ausgabe doppelt vorhanden ist und doppelt gepflegt werden muß.

Das nachfolgende Listing kombiniert beide Varianten und demonstriert dies am Ausdruck einer Testseite. Das Programm erstellt ein Hauptfenster und ruft zwei Sekunden später die Methode `printTestPage` zur Druckausgabe auf. Darin wird zunächst ein `PrintJob` erzeugt und dann gemäß dem oben beschriebenen Verfahren die Ausgabegröße ermittelt. Anschließend wird der Grafikkontext beschafft und ein Rahmen, einige Textzeilen mit Angaben zu den Metriken und eine Graustufenmatrix werden ausgegeben. Nach Ende der Druckausgabe wird die Seite mit `dispose` ausgegeben und der Druckjob mit `end` geschlossen. Der Code für die Darstellung der Graustufenmatrix wurde in der Methode `paintGrayBoxes` implementiert und wird von der Bildschirm- und Druckausgabe gemeinsam verwendet:

Beispiel

```

001 /* Listing1508.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1508
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1508 wnd = new Listing1508();
012     }
013
014     public Listing1508()
015     {
016         super("Listing1508");
017         addWindowListener(
018             new WindowAdapter() {
019                 public void windowClosing(WindowEvent event)
020                 {
021                     System.exit(0);
022                 }
023             }
024         );
025         setBackground(Color.lightGray);
026         setSize(400,400);
027         setVisible(true);
028         //Ausdruck in 2 Sekunden starten
029         try {
030             Thread.sleep(2000);
031         } catch (InterruptedException e) {
032             //nichts
033         }
034         printTestPage();
035     }
036
037     public void paint(Graphics g)
038     {
039         paintGrayBoxes(g, 40, 50);
040     }
041
042     public void printTestPage()
043     {
044         PrintJob pjob = getToolkit().getPrintJob(
045             this,
046             "Testseite",
047             null
048         );
049         if (pjob != null) {
050             //Metriken
051             int pres = pjob.getPageResolution();
052             int sres = getToolkit().getScreenResolution();
053             Dimension d2 = new Dimension(
054                 (int)((21.0 - 2.0) / 2.54) * sres,
055                 (int)((29.7 - 2.0) / 2.54) * sres
056             );
057             //Ausdruck beginnt
058             Graphics pg = pjob.getGraphics();
059             if (pg != null) {
060                 //Rahmen
061                 pg.drawRect(0, 0, d2.width, d2.height);
062                 //Text
063                 pg.setFont(new Font("TimesRoman", Font.PLAIN, 24));
064                 pg.drawString("Testseite", 40, 70);

```

```

065         pg.drawString(
066             "Druckerauflösung : " + pres + " dpi",
067             40,
068             100
069         );
070         pg.drawString(
071             "Bildschirmauflösung : " + sres + " dpi",
072             40,
073             130
074         );
075         pg.drawString(
076             "Seitengröße : " + d2.width + " * " + d2.height,
077             40,
078             160
079         );
080         //Graustufenkästchen
081         paintGrayBoxes(pg, 40, 200);
082         //Seite ausgeben
083         pg.dispose();
084     }
085     pjob.end();
086 }
087 }
088
089 private void paintGrayBoxes(Graphics g, int x, int y)
090 {
091     for (int i = 0; i < 16; ++i) {
092         for (int j = 0; j < 16; ++j) {
093             int level = 16 * i + j;
094             g.setColor(Color.black);
095             g.drawRect(x + 20 * j, y + 20 * i, 20, 20);
096             g.setColor(new Color(level, level, level));
097             g.fillRect(x + 1 + 20 * j, y + 1 + 20 * i, 19, 19);
098         }
099     }
100 }
101 }

```

Listing 15.8: Ausdruck einer Testseite

Die Bildschirmausgabe des Programms kann [Abbildung 15.8](#) entnommen werden. Die Druckausgabe sieht ähnlich aus, enthält aber zusätzlich noch einen Rahmen und die Textausgabe mit den Informationen zu den Druckmetriken.

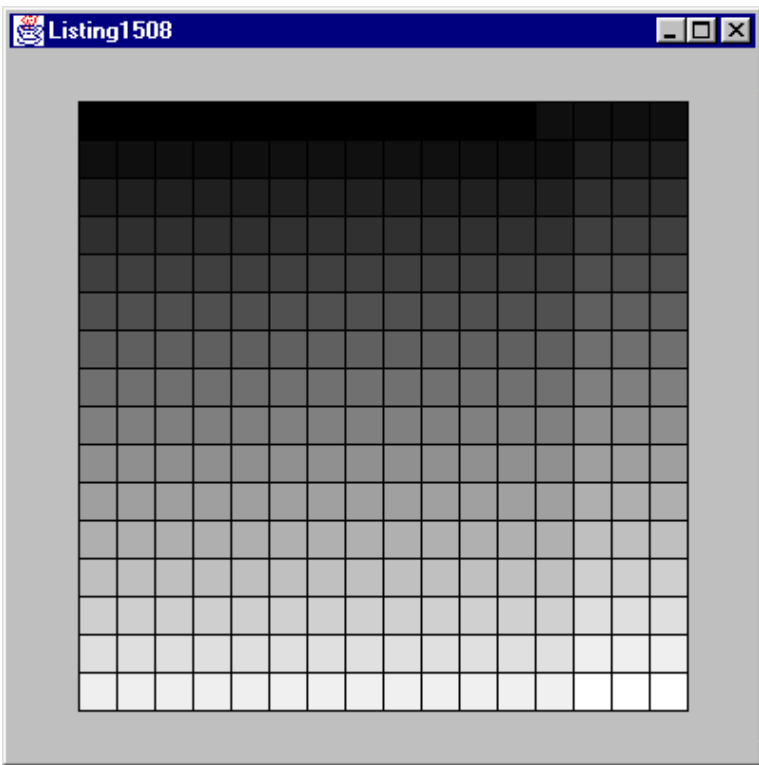


Abbildung 15.8: Das Programm zur Druckausgabe

Im JDK 1.2 wurde das Konzept der Druckausgabe erneut geändert. Anstelle der oben beschriebenen Klassen werden nun die Klassen und Interfaces aus dem Paket [java.awt.print](#) verwendet (beispielsweise [PrinterJob](#), [Printable](#) und [Pageable](#)). Leider war das neue Drucksystem mit der Ausgabe der Version 1.2 noch nicht stabil genug, um hier ausführlich beschrieben zu werden. Im Usenet wurden eine Vielzahl von Problemen und Schwierigkeiten mit unterschiedlichen Druckertypen und Betriebssystemen diskutiert. Wir werden die Neuerungen daher hier noch nicht behandeln, sondern sie zu gegebener Zeit in der Online-Version des Buchs aufgreifen.

JDK1.1/1.2

15.5 Zusammenfassung

- [15.5 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Elementare Textausgabe mit den Methoden [drawString](#), [drawChars](#) und [drawBytes](#)
- Die Größe der Client-Area und die Methoden [getSize](#) und [getInsets](#)
- Erzeugen unterschiedlicher Schriftarten mit Hilfe von [Font](#)-Objekten
- Die Standard-Fonts [SansSerif](#), [Serif](#) und [Monospaced](#)
- Ermitteln der auf einer Plattform verfügbaren Schriften
- Die Größenmaßzahlen einer Schrift
- Zugriff auf Font-Metriken mit der Klasse [FontMetrics](#)
- Erzeugen eines Druckjobs durch Aufruf von [getPrintJob](#)
- Der Device-Kontext [PrintGraphics](#) zur Ausgabe von Text und Grafiken auf einen Drucker
- Bestimmung der Auflösung und Abmessung einer Druckseite

Kapitel 16

Farben

- [16 Farben](#)
 - [16.1 Das Java-Farbmodell](#)
 - [16.2 Erzeugen von Farben](#)
 - [16.3 Verwenden von Farben](#)
 - [16.4 Systemfarben](#)
 - [16.5 Zusammenfassung](#)

16.1 Das Java-Farbmodell

- 16.1 Das Java-Farbmodell

Das Java-Farbmodell basiert auf dem *RGB-Farbmodell*. Dieses stellt Farben mit 24 Bit Tiefe dar und setzt jede Farbe aus einer Mischung der drei Grundfarben Rot, Grün und Blau zusammen. Da jede dieser Grundfarben einen Anteil von 0 bis 255 haben kann und so mit jeweils 8 Datenbits darstellbar ist, ergibt sich eine gesamte Farbtiefe von 24 Bit. Ein Wert von 0 für eine der Grundfarben bedeutet dabei, daß diese Grundfarbe nicht in das Ergebnis eingeht, ein Wert von 255 zeigt die maximale Intensität dieser Farbe an. RGB-Farben werden üblicherweise durch ganzzahlige Tripel (r,g,b) dargestellt, die den Anteil an der jeweiligen Grundfarbe in der Reihenfolge Rot, Grün und Blau darstellen. [Tabelle 16.1](#) listet einige gebräuchliche Farben und ihre RGB-Werte auf:

Farbe	Rot-Anteil	Grün-Anteil	Blau-Anteil
Weiß	255	255	255
Schwarz	0	0	0
Grau	127	127	127
Rot	255	0	0
Grün	0	255	0
Blau	0	0	255
Yellow	255	255	0
Magenta	255	0	255
Cyan	0	255	255

Tabelle 16.1: Gebräuchliche Farbwerte

Neben dem RGB-Farbmodell unterstützt Java auch das *HSB-Farbmodell*. Dieses stellt eine Farbe durch die drei Parameter *Farbton*, *Intensität* und *Helligkeit* dar. Java stellt einige Methoden zur Konvertierung zwischen RGB und HSB zur Verfügung, auf die wir aber nicht weiter eingehen werden.

Hinweis

16.2 Erzeugen von Farben

- 16.2 Erzeugen von Farben

Farben werden in Java durch die Klasse [Color](#) repräsentiert. Jedes [Color](#)-Objekt repräsentiert dabei eine Farbe, die durch ihre RGB-Werte eindeutig bestimmt ist. Farben können durch Instanzieren eines [Color](#)-Objekts und Übergabe des gewünschten RGB-Wertes an den Konstruktor erzeugt werden:

```
public Color(int r, int g, int b)
public Color(float r, float g, float b)
```

Der Konstruktor mit den [int](#)-Parametern erwartet dabei die Farbwerte als Ganzzahlen im Bereich von 0 bis 255. Alternativ dazu können die Farbanteile auch als Fließkommazahlen übergeben werden. In diesem Fall muß jeder der Werte im Bereich von 0.0 bis 1.0 liegen. 0.0 entspricht dem völligen Fehlen dieses Farbanteils und 1.0 der maximalen Intensität (entsprechend dem ganzzahligen Farbwert 255).

Alternativ stellt die Klasse [Color](#) ein Reihe von statischen [Color](#)-Objekten zur Verfügung, die direkt verwendet werden können:

```
public static Color white
public static Color lightGray
public static Color gray
public static Color darkGrey
public static Color black
public static Color red
public static Color blue
public static Color green
public static Color yellow
public static Color magenta
public static Color cyan
public static Color orange
public static Color pink
```

Um von einem bestehenden Farbobjekt die RGB-Werte zu ermitteln, stellt die Klasse [Color](#) die Methoden [getRed](#), [getGreen](#) und [getBlue](#) zur Verfügung:

```
public int getRed()
public int getGreen()
public int getBlue()
```

16.3 Verwenden von Farben

- 16.3 Verwenden von Farben

Um Farben bei der Ausgabe von Grafik oder Schrift zu verwenden, muß ein geeignetes [Color](#)-Objekt beschafft und dem [Graphics](#)-Objekt mit Hilfe der Methode [setColor](#) zugewiesen werden. Die Ausgaben erfolgen dann so lange in der neuen Farbe, bis durch Aufruf von [setColor](#) eine andere Farbe festgelegt wird. Mit Hilfe der Methode [getColor](#) kann die aktuelle Farbe ermittelt werden:

[java.awt.Graphics](#)

```
public void setColor(Color c)
```

```
public Color getColor()
```

Das folgende Beispiel zeigt die Verwendung von Farben und ihre Zusammensetzung aus den drei Grundfarben Rot, Grün und Blau. Es stellt das Prinzip der additiven Farbmischung mit Hilfe dreier überlappender Kreise dar:

Beispiel

[Listing1601.java](#)

```
001 /* Listing1601.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1601
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1601 wnd = new Listing1601();
012     }
013
014     public Listing1601()
015     {
016         super("Der Farbenkreis");
017         addWindowListener(
018             new WindowAdapter() {
019                 public void windowClosing(WindowEvent event)
020                 {
021                     System.exit(0);
022                 }
023             }
024         );
025         setSize(300,200);
026         setVisible(true);
027     }
028
029     public void paint(Graphics g)
030     {
031         int top  = getInsets().top;
032         int left = getInsets().left;
033         int maxX = getSize().width-left-getInsets().right;
034         int maxY = getSize().height-top-getInsets().bottom;
035         Color col;
036         int arx[]  = {130,160,190};
037         int ary[]  = {60,110,60};
038         int arr[]  = {50,50,50};
039         int arcol[] = {0,0,0};
040         boolean paintit;
041         int dx, dy;
042
043         for (int y = 0; y < maxY; ++y) {
044             for (int x = 0; x < maxX; ++x) {
```

```

045     paintit = false;
046     for (int i = 0; i < arcol.length; ++i) {
047         dx = x - arx[i];
048         dy = y - ary[i];
049         arcol[i] = 0;
050         if ((dx*dx+dy*dy) <= arr[i]*arr[i]) {
051             arcol[i] = 255;
052             paintit = true;
053         }
054     }
055     if (paintit) {
056         col = new Color(arcol[0],arcol[1],arcol[2]);
057         g.setColor(col);
058         g.drawLine(x+left,y+top,x+left+1,y+top+1);
059     }
060 }
061 }
062 }
063 }

```

Listing 16.1: Darstellung des Farbenkreises

Das Programm arbeitet in der Weise, daß für jeden einzelnen Punkt der Zeichenfläche berechnet wird, ob dieser in einem der drei Kreise liegt. Ist dies der Fall, so wird die zugehörige Farbkomponente auf 255 gesetzt, andernfalls auf 0. Die Bestimmung der Kreiszugehörigkeit erfolgt mit Hilfe des Satzes von Pythagoras, nach dem ein Punkt genau dann zu einem Kreis gehört, wenn die Summe der Quadrate der Abstände vom **x**- und **y**-Wert zum Mittelpunkt kleiner gleich dem Quadrat des Radius ist. Die drei Mittelpunkte werden in unserem Beispiel in den Arrays **arx** und **ary**, die Radien der Kreise in **arr** gespeichert. Die boolesche Variable **paintit** zeigt an, ob der Punkt in wenigstens einem der drei Kreise liegt und daher überhaupt eine Ausgabe erforderlich ist.

Hinweis

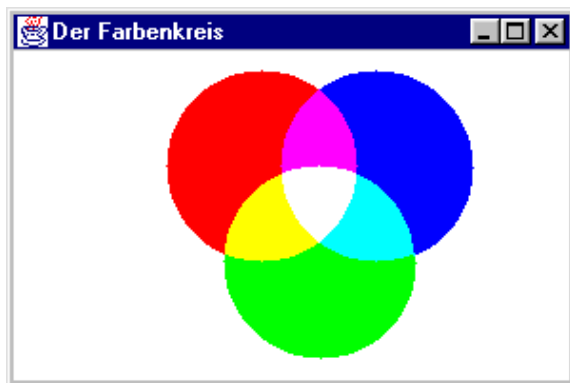


Abbildung 16.1: Der Farbenkreis

16.4 Systemfarben

- 16.4 Systemfarben

Um ihren Anwendungen ein einheitliches Look & Feel zu geben, definieren grafische Oberflächen in der Regel eine Reihe von Systemfarben. Diese können im Programm verwendet werden, um beispielsweise die Farbe des Hintergrunds oder die von Dialogelementen konsistent festzulegen. Während es in früheren Versionen von Java keine Möglichkeit gab, auf Systemfarben zuzugreifen, steht diese Möglichkeit mit der Klasse `SystemColor` im JDK 1.1 portabel zur Verfügung.

`SystemColor` ist aus `Color` abgeleitet und unterscheidet sich von ihr vor allem durch die Fähigkeit, die Farbe dynamisch zu ändern. Dieses Feature ist auf Systemen interessant, die eine Nachricht an die Anwendung senden, wenn sich eine Systemfarbe geändert hat. Wir wollen an dieser Stelle nicht näher darauf eingehen.

Die Klasse `SystemColor` stellt eine Reihe von vordefinierten Farben zur Verfügung, die zu den entsprechenden Systemfarben des Desktops korrespondieren. Da `SystemColor` aus `Color` abgeleitet ist, können diese leicht anstelle eines anwendungsspezifischen `Color`-Objekts verwendet werden, wenn eine einheitliche Farbgebung gewünscht ist. [Tabelle 16.2](#) gibt eine Übersicht dieser Farben.

Farbkonstante	Bedeutung
<code>SystemColor.desktop</code>	Hintergrundfarbe des Desktops
<code>SystemColor.activeCaption</code>	Hintergrundfarbe für die Titelleiste von selektierten Fenstern
<code>SystemColor.activeCaptionText</code>	Schriftfarbe für die Titelleiste von selektierten Fenstern
<code>SystemColor.activeCaptionBorder</code>	Rahmenfarbe für die Titelleiste von selektierten Fenstern
<code>SystemColor.inactiveCaption</code>	Hintergrundfarbe für die Titelleiste von nicht selektierten Fenstern
<code>SystemColor.inactiveCaptionText</code>	Schriftfarbe für die Titelleiste von nicht selektierten Fenstern
<code>SystemColor.inactiveCaptionBorder</code>	Rahmenfarbe für die Titelleiste von nicht selektierten Fenstern
<code>SystemColor.window</code>	Hintergrundfarbe für Fenster
<code>SystemColor.windowBorder</code>	Farbe für Fensterrahmen
<code>SystemColor.windowText</code>	Farbe für Text im Fenster
<code>SystemColor.menu</code>	Hintergrundfarbe für Menüs
<code>SystemColor.menuText</code>	Textfarbe für Menüs
<code>SystemColor.text</code>	Hintergrundfarbe für Textfelder
<code>SystemColor.textText</code>	Textfarbe für Textfelder
<code>SystemColor.textHighlight</code>	Hintergrundfarbe für hervorgehobenen Text
<code>SystemColor.textHighlightText</code>	Textfarbe für hervorgehobenen Text
<code>SystemColor.textInactiveText</code>	Textfarbe für inaktiven Text
<code>SystemColor.control</code>	Hintergrundfarbe für Dialogelemente
<code>SystemColor.controlText</code>	Textfarbe für Dialogelemente
<code>SystemColor.controlHighlight</code>	Farbe für hervorgehobene Dialogelemente
<code>SystemColor.controlLtHighlight</code>	Helle Farbe für hervorgehobene Dialogelemente
<code>SystemColor.controlShadow</code>	Farbe für den Schatten von Dialogelementen
<code>SystemColor.controlDkShadow</code>	Dunklere Farbe für den Schatten von Dialogelementen
<code>SystemColor.scrollbar</code>	Hintergrundfarbe für Schieberegler
<code>SystemColor.info</code>	Hintergrundfarbe für Hilfetext

<code>SystemColor.infoText</code>	Textfarbe für Hilfetext
-----------------------------------	-------------------------

Tabelle 16.2: Liste der vordefinierten Systemfarben

Das folgende Listing zeigt ein einfaches Programm, das den Text »Tag & Nacht« in den Systemfarben für normalen und hervorgehobenen Text ausgibt. Als Fensterhintergrund wird die Systemfarbe `desktop` verwendet. Der Text »Tag« wird mit Hilfe der Systemfarben `textText` und `text` in normaler Textfarbe auf normalem Texthintergrund ausgegeben. Der Text »Nacht« wird dagegen mit den Systemfarben `textHighlight` und `textHighlightText` invertiert dargestellt. Das dazwischen stehende »&« wird in Blau auf normalem Hintergrund ausgegeben.

Beispiel

[Listing1602.java](#)

```

001 /* Listing1602.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1602
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1602 wnd = new Listing1602();
012     }
013
014     public Listing1602()
015     {
016         super("Listing1602");
017         setBackground(SystemColor.desktop);
018         setSize(200,100);
019         setVisible(true);
020         //WindowListener
021         addWindowListener(
022             new WindowAdapter() {
023                 public void windowClosing(WindowEvent event)
024                 {
025                     System.exit(0);
026                 }
027             }
028         );
029     }
030
031     public void paint(Graphics g)
032     {
033         g.setFont(new Font("Serif",Font.PLAIN,36));
034         FontMetrics fm = g.getFontMetrics();
035         int sheight = fm.getHeight();
036         int curx = 10;
037         int cury = getInsets().top + 10;
038         //"Tag" in normaler Textfarbe
039         int swidth = fm.stringWidth("Tag");
040         g.setColor(SystemColor.text);
041         g.fillRect(curx,cury,swidth,sheight);
042         g.setColor(SystemColor.textText);
043         g.drawString("Tag",curx,cury+fm.getAscent());
044         //"&" in Blau auf normalem Hintergrund
045         curx += swidth + 5;
046         swidth = fm.stringWidth("&");
047         g.setColor(Color.blue);
048         g.drawString("&",curx,cury+fm.getAscent());
049         //"Nacht" in hervorgehobener Textfarbe
050         curx += swidth + 5;
051         swidth = fm.stringWidth("Nacht");
052         g.setColor(SystemColor.textHighlight);
053         g.fillRect(curx,cury,swidth,sheight);
054         g.setColor(SystemColor.textHighlightText);

```



```
055      g.drawString( "Nacht" ,curx,cury+fm.getAscent() );
056  }
057 }
```

Listing 16.2: Verwendung von Systemfarben

Abbildung 16.2 zeigt die Ausgabe des Programms:



Abbildung 16.2: Verwendung von Systemfarben

Ein weiteres Beispiel zur Verwendung der Systemfarben findet sich in [Kapitel 22](#). Das zu dem [Hinweis](#) Dialogelement [ScrollPane](#) vorgestellte Beispielprogramm zeigt alle Systemfarben zusammen mit ihrem Namen in einem verschiebbaren Fenster an.

16.5 Zusammenfassung

- [16.5 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Das RGB-Farbmodell
- Gebräuchliche Farbwerte und ihre RGB-Codes
- Erzeugen von Farben mit der Klasse [Color](#)
- Zugriff auf die Ausgabefarbe des Geräte-Kontexts durch Aufruf von [setColor](#) und [getColor](#)
- Verwendung der Systemfarben

Kapitel 17

Fenster

- [17 Fenster](#)
 - [17.1 Die verschiedenen Fensterklassen](#)
 - [17.2 Aufrufen und Schließen eines Fensters](#)
 - [17.3 Visuelle Eigenschaften](#)
 - [17.4 Anzeigezustand](#)
 - [17.5 Fensterelemente](#)
 - [17.5.1 Der Fenstertitel](#)
 - [17.5.2 Das Icon des Fensters](#)
 - [17.5.3 Der Mauscursor](#)
 - [17.5.4 Die Vorder- und Hintergrundfarbe](#)
 - [17.5.5 Der Standard-Font](#)
 - [17.6 Zusammenfassung](#)

17.1 Die verschiedenen Fensterklassen

- 17.1 Die verschiedenen Fensterklassen

Das Abstract Windowing Toolkit von Java enthält verschiedene Fensterklassen, die über eine gemeinsame Vererbungshierarchie miteinander in Verbindung stehen (siehe [Abbildung 17.1](#)). Oberste Fensterklasse ist [Component](#), daraus wurde [Container](#) abgeleitet. [Container](#) ist die Oberklasse der beiden Klassen [Window](#) und [Panel](#). Während [Window](#) sich in die Unterklassen [Frame](#) und [Dialog](#) verzweigt, wird aus [Panel](#) die Klasse [Applet](#) abgeleitet. Unterhalb von [Dialog](#) gibt es noch den Standard-File-Dialog in der Klasse [FileDialog](#).

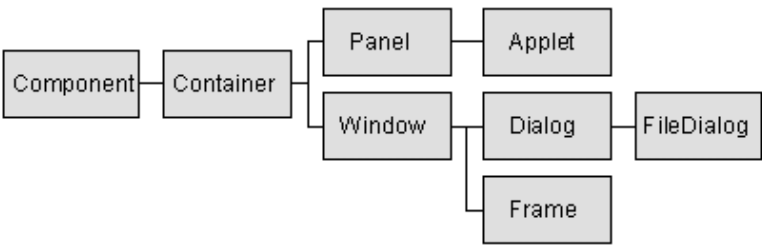


Abbildung 17.1: Hierarchie der Fensterklassen

[Component](#) ist eine abstrakte Klasse, deren Aufgabe darin besteht, ein Programmelement zu repräsentieren, das eine Größe und Position hat und das in der Lage ist, eine Vielzahl von Ereignissen zu senden und auf diese zu reagieren.

Die Klasse [Container](#) ist ebenfalls eine abstrakte Klasse. Sie ist dafür zuständig, innerhalb einer Komponente andere Komponenten aufnehmen zu können. [Container](#) stellt Methoden zur Verfügung, um Komponenten hinzuzufügen oder sie zu entfernen und realisiert in Zusammenarbeit mit den [LayoutManager](#)-Klassen die Positionierung und Anordnung der Komponenten.

[Panel](#) ist die einfachste konkrete Klasse mit den Eigenschaften von [Component](#) und [Container](#). Sie wird in der Praxis vor allem dazu verwendet, [Container](#) zu schachteln, um das Layout der Komponenten zu beeinflussen. Wir werden auf die Verwendung der Klasse [Panel](#) in [Kapitel 21](#) bei der Beschreibung von Benutzerdialogen zurückkommen.

Die für die Entwicklung von Applets wichtige Klasse [Applet](#) ist eine direkte Unterklasse von [Panel](#). Sie erweitert zwar die Funktionalität der Klasse [Panel](#) um Methoden, die für das Ausführen von Applets von Bedeutung sind, bleibt aber letztlich ein Programmelement, das eine Größe und Position hat, auf Ereignisse reagieren kann und in der Lage ist, weitere Komponenten aufzunehmen. Einerseits ist es bemerkenswert, daß eine Klasse wie [Applet](#), die für eine riesige Zahl von Anwendungen von elementarer Bedeutung ist, ganz unten in der Vererbungskette eines stark spezialisierten Zweigs der Klassenhierarchie steht. Andererseits liefert die Klasse [Applet](#) jedoch mit der vorstehenden Beschreibung eine nahezu perfekte Charakterisierung von Applets. Wir werden in [Kapitel 25](#) auf die Eigenschaften dieser Klasse näher eingehen.

Die Klasse [Window](#) abstrahiert ein Top-Level-Window ohne Rahmen, Titelleiste und Menü. Sie ist für Anwendungen geeignet, die ihre Rahmenelemente selbst zeichnen oder die volle Kontrolle über das gesamte Fenster benötigen.

Die Klasse [Frame](#) repräsentiert ein Top-Level-Window mit Rahmen, Titelleiste und optionalem Menü. Einem [Frame](#) kann ein Icon zugeordnet werden, das angezeigt wird, wenn das Fenster minimiert wird. Es kann eingestellt werden, ob das Fenster vom Anwender in der Größe verändert werden kann. Zusätzlich besteht die Möglichkeit, das Aussehen des Mauszeigers zu verändern.

Die zweite aus [Window](#) abgeleitete Klasse ist [Dialog](#). Sie ist dafür vorgesehen, modale oder nicht-modale Dialoge zu realisieren. Ein modaler Dialog ist ein Fenster, das immer im Vordergrund des Fensters bleibt, von dem es erzeugt wurde, und das alle übrigen Fensteraktivitäten und Ereignisse so lange blockiert, bis es geschlossen wird. Ein nicht-modaler Dialog kann mit anderen Fenstern koexistieren und erlaubt es, im aufrufenden Fenster weiterzuarbeiten.

Aufgrund von Bugs in der Windows-Portierung des AWT gab es bis zur Version 1.0.2 des AWT keine vernünftige Möglichkeit, *modale* Dialoge zu erzeugen. Dies war ein ziemlich schwerwiegendes Problem, denn modale Dialoge kommen in der Programmentwicklung sehr häufig vor. Sie werden immer dann benötigt, wenn das Programm auf die Eingabe eines Anwenders warten muß, bevor es fortfahren kann. Leider funktionierte gerade dieser Wartemechanismus nicht, und es wurden eine Reihe aufwendiger Workarounds veröffentlicht. Mit der Version 1.1 des JDK war es erstmals auch unter Windows 95 möglich, echte modale Dialoge in Java zu erstellen. Wir werden auf die Details in [Kapitel 21](#) zurückkommen.

Hinweis

Die unterste Klasse in der Hierarchie der Fenster ist [FileDialog](#). Sie stellt den Standard-Dateidialog des jeweiligen Betriebssystems zur

Verfügung. Dieser kann beim Laden oder Speichern einer Datei zur Eingabe oder Auswahl eines Dateinamens verwendet werden. Gegenüber den Möglichkeiten des Standard-Dateidialogs, wie er vom Windows-API zur Verfügung gestellt wird, sind die Fähigkeiten der Klasse [FileDialog](#) allerdings etwas eingeschränkt.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	<	>	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	----------------------	----------------------	--------------------------

17.2 Aufrufen und Schließen eines Fensters

- 17.2 Aufrufen und Schließen eines Fensters

Um ein Fenster auf dem Bildschirm anzuzeigen, muß zunächst die passende Fensterklasse instanziiert werden. Dafür kommen die Klassen [Window](#), [Frame](#), [Dialog](#), [Applet](#) und [FileDialog](#) in Frage. Da die beiden Dialogklassen und die Entwicklung von Applets später behandelt werden, verbleiben die Klassen [Window](#) und [Frame](#). Beide Klassen haben unterschiedliche Konstruktoren:

```

public Frame()
public Frame(String title)
public Window(Frame parent)

```

[java.awt.Frame](#)
[java.awt.Window](#)

Ein Objekt der Klasse [Frame](#) kann demnach parameterlos oder mit einem [String](#), der den Titel des Fensters angibt, erzeugt werden. Bei der Instanzierung eines [Window](#)-Objekts muß dagegen das Vaterfenster übergeben werden.

Nach der Instanzierung wird die Methode [setVisible](#) aufgerufen, um das Fenster anzuzeigen:

```

public void setVisible(boolean visible)

```

[java.awt.Component](#)

Der Parameter `visible` gibt an, ob das Fenster angezeigt oder geschlossen werden soll. Wird `true` übergeben, so wird es angezeigt, andernfalls geschlossen. Die Aufrufsyntax ist für [Frame](#) und [Window](#) identisch, denn [setVisible](#) wurde aus der gemeinsamen Oberklasse [Component](#) geerbt. Ein Aufruf von `setVisible(true)` legt die Fensterressource an, stattet das Fenster mit einer Reihe von Standardeigenschaften aus und zeigt es auf dem Bildschirm an.

Um ein Fenster zu schließen, sind die Methoden `setVisible(false)` und [dispose](#) aufzurufen:

```

public void dispose()

```

[java.awt.Window](#)

`setVisible(false)` macht das Fenster unsichtbar, und [dispose](#) gibt die zugeordneten Windows-Ressourcen frei und entfernt das Fenster aus der Owner-Child-Registrierung.

In der aktuellen Implementierung enthält [dispose](#) am Anfang einen Aufruf von `hide`, durch das seinerseits ein implizites `setVisible(false)` aufgerufen wird. Daher wäre es auch möglich, ein Fenster mit einem einzigen Aufruf von [dispose](#) zu schließen. Der zusätzliche Aufruf von `setVisible(false)` ist allerdings nicht schädlich und macht die Aufgabentrennung beider Methoden deutlich.

Hinweis

Das folgende Beispiel zeigt ein Programm, das einen [Frame](#) auf dem Bildschirm anzeigt, ihn nach ca. 3 Sekunden wieder entfernt und das Programm dann beendet:

Beispiel

[Listing1701.java](#)

```

001  /* Listing1701.java */
002
003  import java.awt.*;
004
005  public class Listing1701
006  {
007      public static void main(String[] args)
008      {
009          Frame frame = new Frame("Listing1701");
010          frame.setSize(300,200);
011          frame.setVisible(true);
012          try {
013              Thread.sleep(3000);
014          } catch (InterruptedException e) {
015              //nichts
016          }
017          frame.setVisible(false);
018          frame.dispose();

```

```
019      System.exit(0);
020  }
021 }
```

Listing 17.1: Anzeigen und Entfernen eines Frames

17.3 Visuelle Eigenschaften

- 17.3 Visuelle Eigenschaften

Ein Fenster besitzt eine Reihe von Eigenschaften, die sein Aussehen bestimmen. Dazu gehören die Art des Rahmens, die Position und Größe des Fensters und die Anordnung des Fensters in Relation zu anderen Fenstern auf dem Bildschirm.

Wie schon erwähnt, wird die Art des Rahmens durch die Klasse bestimmt, die zur Erzeugung eines Fensters verwendet wird. Während die Klasse [Window](#) ein Fenster ohne Rahmen darstellt, besitzt ein [Frame](#) einen Rahmen, eine Titelleiste und gegebenenfalls ein Menü.

Die Größe und Position eines Fensters können mit den Methoden [setSize](#), [setBounds](#) und [setLocation](#) bestimmt werden:

[java.awt.Component](#)

```
public void setSize(int width, int height)
```

```
public void setSize(Dimension d)
```

```
public void setBounds(int x, int y, int width, int height)
```

```
public void setBounds(Rectangle r)
```

```
public void setLocation(int x, int y)
```

```
public void setLocation(Point p)
```

[setSize](#) verändert die Größe des Fensters auf den Wert ([width,height](#)), und [setLocation](#) bewegt die linke obere Ecke an die Bildschirmposition ([x,y](#)). Die Methode [setBounds](#) kombiniert die Funktionen von [setSize](#) und [setLocation](#) und positioniert ein Fenster der Größe ([width,height](#)) an der Position ([x,y](#)).

Das folgende Beispiel stellt eine sehr einfache Form eines (manuell zu aktivierenden) »Bildschirmschoners« dar, der den gesamten Bildschirm dunkel schaltet und die Anmerkung »Bitte eine Taste drücken« in die linke obere Ecke schreibt. Zusätzlich gibt das Programm die aktuelle Auflösung des Bildschirms aus, nachdem sie mit der Methode [getScreenSize](#) der Klasse [Toolkit](#) ermittelt wurde. Nach Drücken einer Taste wird das Fenster geschlossen und das Programm beendet:

Beispiel

[Listing1702.java](#)

```
001 /* Listing1702.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1702
007 extends Window
008 {
009     public static void main(String[] args)
010     {
011         final Listing1702 wnd = new Listing1702();
012         wnd.setLocation(new Point(0,0));
013         wnd.setSize(wnd.getToolkit().getScreenSize());
014         wnd.setVisible(true);
015         wnd.requestFocus();
016         wnd.addKeyListener(
017             new KeyAdapter() {
018                 public void keyPressed(KeyEvent event)
019                 {
020                     wnd.setVisible(false);
021                     wnd.dispose();
022                     System.exit(0);
023                 }
024             }
025         );
```



```
026     }
027
028     public Listing1702()
029     {
030         super(new Frame());
031         setBackground(Color.black);
032     }
033
034     public void paint(Graphics g)
035     {
036         g.setColor(Color.red);
037         g.drawString(
038             "Bildschirmgröße ist "+
039             getSize().width+"*"+getSize().height,
040             10,
041             20
042         );
043         g.drawString("Bitte eine Taste drücken",10,40);
044     }
045 }
```

Listing 17.2: Ein einfacher Bildschirmschoner

Die Details des hier implementierten Event-Handlings mit Hilfe der durch [addKeyListener](#) eingefügten anonymen Klasse wollen wir auf [Kapitel 18](#) vertagen. Die Ausgabe des Programms (in verkleinerter Form) ist:

Hinweis



Abbildung 17.2: Ein einfacher Bildschirmschoner

17.4 Anzeigezustand

- 17.4 Anzeigezustand

Seit dem JDK 1.2 kann der Anzeigezustand eines Fensters geändert werden. Dazu gibt es in der Klasse `Frame` die Methoden `setState` und `getState`:

`java.awt.Frame`

```
public synchronized void setState(int state)
```

```
public synchronized int getState()
```

Mit `setState` kann der Anzeigezustand des Fensters zwischen »normal« und »als Symbol« umgeschaltet werden. Wird die Konstante `ICONIFIED` der Klasse `Frame` übergeben, erfolgt die Darstellung als Symbol, wird `NORMAL` übergeben, erfolgt die normale Darstellung. Mit `getState` kann der aktuelle Anzeigezustand abgefragt werden.

Das folgende Programm öffnet ein Fenster, stellt es nach zwei Sekunden als Symbol und nach weiteren zwei Sekunden wieder normal dar. Anschließend wird das Fenster geschlossen und das Programm beendet:

Beispiel

`Listing1703.java`

```
001 /* Listing1703.java */
002
003 import java.awt.*;
004
005 public class Listing1703
006 {
007     public static void main(String[] args)
008     {
009         Frame frame = new Frame("Listing1703");
010         frame.setSize(300,200);
011         frame.setVisible(true);
012         try {
013             Thread.sleep(2000);
014         } catch (InterruptedException e) {
015             //nichts
016         }
017         frame.setState(Frame.ICONIFIED);
018         try {
019             Thread.sleep(2000);
020         } catch (InterruptedException e) {
021             //nichts
022         }
023         frame.setState(Frame.NORMAL);
024         try {
025             Thread.sleep(2000);
026         } catch (InterruptedException e) {
027             //nichts
028         }
029         frame.setVisible(false);
030         frame.dispose();
031         System.exit(0);
032     }
033 }
```

Listing 17.3: Anzeigezustand eines Fensters umschalten

17.5 Fensterelemente

- [17.5 Fensterelemente](#)
 - [17.5.1 Der Fenstertitel](#)
 - [17.5.2 Das Icon des Fensters](#)
 - [17.5.3 Der Mauscursor](#)
 - [17.5.4 Die Vorder- und Hintergrundfarbe](#)
 - [17.5.5 Der Standard-Font](#)

Neben den visuellen Eigenschaften eines Fensters gibt es noch eine Reihe weiterer Elemente, die einem Fenster zugeordnet werden können. Hierzu zählen die *Titelleiste*, das *Menü*, ein *Icon*, ein *Mauscursor*, eine bestimmte *Vorder- und Hintergrundfarbe* und ein *Standard-Font* zur Ausgabe von Schriften. Bis auf die Definition und die Zuordnung von Menüs, die in [Kapitel 20](#) erläutert werden, soll der Umgang mit diesen Fensterelementen in den folgenden Unterabschnitten erklärt werden.

17.5.1 Der Fenstertitel

Die Titelleiste eines Fensters läßt sich in den Klassen [Frame](#) und [Dialog](#) mit Hilfe der Methode [setTitle](#) verändern:

```

public void setTitle(String title)
public String getTitle()

```

Ein Aufruf dieser Methode ändert die Beschriftung der Titelleiste in den als [String](#) übergebenen Parameter `title`. Mit der Methode [getTitle](#) kann die Titelleiste abgefragt werden.

17.5.2 Das Icon des Fensters

Wenn ein Fenster unter Windows minimiert wird, zeigt es ein Icon an. Mit einem Doppelklick auf das Icon kann die ursprüngliche Größe des Fensters wiederhergestellt werden. Mit Hilfe der Methode [setIconImage](#) der Klasse [Frame](#) kann dem Fenster ein Icon zugeordnet werden, das beim Minimieren angezeigt wird:

```

public void setIconImage(Image image)

```

Beim Design des Icons steht man nun vor dem Konflikt, entscheiden zu müssen, in welcher Größe das Icon entworfen werden soll. Ein Windows-Programm hat meist ein Haupticon in der Größe 32*32 Pixel und ein kleineres Icon mit 16*16 Pixeln. Beide werden an unterschiedlichen Stellen im Programm benötigt. Das JDK ist glücklicherweise in der Lage, die übergebenen Images so zu skalieren, daß sie die jeweils benötigte Größe annehmen. Die Ergebnisse sind im Falle des großen Icons durchaus brauchbar (so hat z.B. das in [Abbildung 17.3](#) gezeigte und in [Listing 17.4](#) verwendete Icon eine Originalgröße von 60*56 Pixeln), im Falle des kleinen Icons sind die Ergebnisse in Prä-1.2-JDKs jedoch nicht so befriedigend. Hier scheint die Skalierungsroutine lediglich ein schwarz-weißes Ergebnis zu erzeugen.



Abbildung 17.3: Das Beispiel-Icon

Leider unterstützen nicht alle Plattformen, auf denen Java läuft, die Darstellung eines Icons als Symbol für ein minimiertes Fenster. Daher kann [setIconImage](#) nicht als vollständig portabel angesehen werden.

17.5.3 Der Mauscursor

Zur Darstellung des Mausursors bietet die Klasse [Component](#) eine Methode [setCursor](#), mit der ein [Cursor](#)-Objekt vorgegeben werden kann:

```

public void setCursor(Cursor cursor)

```

Die Klasse [Cursor](#) besitzt einen Konstruktor, der als einziges Argument eine ganzzahlige Konstante erwartet, mit der der gewünschte Cursor aus der Liste der vordefinierten Cursortypen ausgewählt werden kann. [Tabelle 17.1](#) gibt eine Auswahl der verfügbaren Cursortypen an.

Das freie Erstellen benutzerdefinierter Cursortypen wird seit dem JDK 1.2 unterstützt. Dazu gibt es in der Klasse [Toolkit](#) die Methoden [createCustomCursor](#), [getBestCursorSize](#) und [getMaximumCursorColors](#). Wir wollen darauf nicht näher eingehen.

JDK1.1/1.2

Konstante aus Frame	Cursorform
Cursor.CROSSHAIR_CURSOR	Fadenkreuz
Cursor.DEFAULT_CURSOR	Standardpfeil
Cursor.MOVE_CURSOR	Vierfachpfeil
Cursor.TEXT_CURSOR	Senkrechter Strich
Cursor.WAIT_CURSOR	Eieruhr

Tabelle 17.1: Konstanten zur Cursorauswahl

17.5.4 Die Vorder- und Hintergrundfarbe

Die Hintergrundfarbe eines Fensters wird vor dem Aufruf von [paint](#) verwendet, um den Hintergrund des Fensters zu löschen. Das Löschen kann man sich so vorstellen, daß ein gefülltes Rechteck in der Größe der Client-Area in der aktuellen Hintergrundfarbe gezeichnet wird. Die Vordergrundfarbe dient zur Ausgabe von Grafik- und Textobjekten, wenn im Grafik-Kontext keine andere Farbe gesetzt wird. Wird die Einstellung nicht geändert, werden in beiden Fällen die unter Windows eingestellten Standardfarben verwendet. Mit Hilfe der Methoden [setForeground](#) und [setBackground](#) der Klasse [Component](#) können Vorder- und Hintergrundfarbe eines Fensters geändert werden:

[java.awt.Component](#)

```
public void setBackground(Color c)

public void setForeground(Color c)
```

17.5.5 Der Standard-Font

Ein Fenster hat einen Standard-Font, der zur Ausgabe von Schrift verwendet wird, wenn nicht im Grafik-Kontext ein anderer Font ausgewählt wird. Die Verwendung eines Standard-Fonts macht Sinn, wenn nur ein Font verwendet werden soll, um Text darzustellen. Dadurch ergeben sich Laufzeitvorteile gegenüber der separaten Auswahl eines Fonts bei jedem Aufruf von [paint](#). Der Standard-Font eines Fensters wird mit der Methode [setFont](#) der Klasse [Component](#) eingestellt:

[java.awt.Component](#)

```
public void setFont(Font f)
```

Das folgende Beispielprogramm erzeugt ein Fenster mit einer Titelleiste, gesetzter Vorder- und Hintergrundfarbe und der Eieruhr als Cursor. Zusätzlich besitzt das Programm das in [Abbildung 17.3](#) gezeigte Icon, das beim Minimieren des Programms und in der Taskleiste angezeigt wird.

Beispiel

[Listing1704.java](#)

```
001 /* Listing1704.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1704
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1704 wnd = new Listing1704();
012         wnd.setSize(300,200);
013         wnd.setLocation(50,50);
014         wnd.setVisible(true);
015     }
016
017     public Listing1704()
018     {
019         super("");
020         assignTitle();
021         assignIcon();
```

```

022     assignCursor();
023     assignColors();
024     assignFont();
025     addWindowListener(
026         new WindowAdapter() {
027             public void windowClosing(WindowEvent event)
028             {
029                 setVisible(false);
030                 dispose();
031                 System.exit(0);
032             }
033         }
034     );
035 }
036
037 private void assignTitle()
038 {
039     setTitle("Listing1704");
040 }
041
042 private void assignIcon()
043 {
044     Image img = getToolkit().getImage("testicon.gif");
045     MediaTracker mt = new MediaTracker(this);
046
047     mt.addImage(img, 0);
048     try {
049         //Warten, bis das Image vollständig geladen ist,
050         mt.waitForAll();
051     } catch (InterruptedException e) {
052         //nothing
053     }
054     setIconImage(img);
055 }
056
057 private void assignCursor()
058 {
059     setCursor(new Cursor(Cursor.WAIT_CURSOR));
060 }
061
062 private void assignColors()
063 {
064     setForeground(Color.white);
065     setBackground(Color.black);
066 }
067
068 private void assignFont()
069 {
070     setFont(new Font("Serif", Font.PLAIN, 28));
071 }
072
073 public void paint(Graphics g)
074 {
075     g.drawString("Test in Vordergrundfarbe",10,70);
076 }
077
078 }

```

Listing 17.4: Ein Programm mit veränderten Fensterelementen

Die Programmausgabe ist wie folgt:

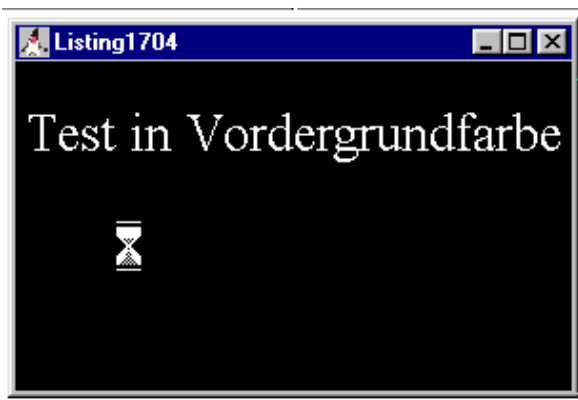


Abbildung 17.4: Ein Programm mit veränderten Fensterelementen

17.6 Zusammenfassung

- [17.6 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Hierarchie und Bedeutung der Fensterklassen [Component](#), [Container](#), [Panel](#), [Window](#), [Applet](#), [Dialog](#) und [Frame](#) in Java
- Anzeigen und Schließen eines Fensters mit den Methoden [setVisible](#) und [dispose](#)
- Die Methoden [setSize](#), [setBounds](#) und [setLocation](#) zum Zugriff auf Größe und Position eines Fensters
- Verändern des Anzeigezustands mit [setState](#)
- Die Methoden [setTitle](#), [getTitle](#), [setIconImage](#) und [setCursor](#)
- Verändern der Vorder- und Hintergrundfarbe des Fensters mit [setForeground](#) und [setBackground](#)
- Verändern des Standard-Fonts eines Fensters mit der Methode [setFont](#)

Kapitel 18

Event-Handling

- [18 Event-Handling](#)
 - [18.1 Das Event-Handling im JDK 1.1](#)
 - [18.1.1 Grundlagen](#)
 - [18.1.2 Ereignistypen](#)
 - [18.1.3 Ereignisempfänger](#)
 - [18.1.4 Ereignisquellen](#)
 - [18.1.5 Adapterklassen](#)
 - [18.1.6 Zusammenfassung](#)
 - [Focus-Ereignisse](#)
 - [Key-Ereignisse](#)
 - [Mouse-Ereignisse](#)
 - [MouseMotion-Ereignisse](#)
 - [Component-Ereignisse](#)
 - [Container-Ereignisse](#)
 - [Window-Ereignisse](#)
 - [Action-Ereignisse](#)
 - [Adjustment-Ereignisse](#)
 - [Item-Ereignisse](#)
 - [Text-Ereignisse](#)
 - [18.2 Entwurfsmuster für den Nachrichtenverkehr](#)
 - [18.2.1 Variante 1: Implementierung eines EventListener-Interfaces](#)
 - [18.2.2 Variante 2: Lokale und anonyme Klassen](#)
 - [Lokale Klassen](#)
 - [Anonyme Klassen](#)
 - [18.2.3 Variante 3: Trennung von GUI- und Anwendungscode](#)
 - [18.2.4 Variante 4: Überlagern der Event-Handler in den Komponenten](#)
 - [18.2.5 Ausblick](#)
 - [18.3 Zusammenfassung](#)

18.1 Das Event-Handling im JDK 1.1

- [18.1 Das Event-Handling im JDK 1.1](#)
 - [18.1.1 Grundlagen](#)
 - [18.1.2 Ereignistypen](#)
 - [18.1.3 Ereignisempfänger](#)
 - [18.1.4 Ereignisquellen](#)
 - [18.1.5 Adapterklassen](#)
 - [18.1.6 Zusammenfassung](#)
 - [Focus-Ereignisse](#)
 - [Key-Ereignisse](#)
 - [Mouse-Ereignisse](#)
 - [MouseMotion-Ereignisse](#)
 - [Component-Ereignisse](#)
 - [Container-Ereignisse](#)
 - [Window-Ereignisse](#)
 - [Action-Ereignisse](#)
 - [Adjustment-Ereignisse](#)
 - [Item-Ereignisse](#)
 - [Text-Ereignisse](#)

18.1.1 Grundlagen

Bei der Programmierung unter einer grafischen Oberfläche erfolgt die Kommunikation zwischen Betriebssystem und Anwendungsprogramm zu einem wesentlichen Teil durch das Versenden von Nachrichten. Die Anwendung wird dabei über alle Arten von Ereignissen und Zustandsänderungen vom Betriebssystem informiert. Dazu zählen beispielsweise Mausklicks, Bewegungen des Mauszeigers, Tastatureingaben oder Veränderungen an der Größe oder Lage des Fensters.

Bei der Verarbeitung des Nachrichtenverkehrs sind zwei verschiedene Arten von Objekten beteiligt. Die *Ereignisquellen (Event Sources)* sind die Auslöser der Nachrichten. Eine Ereignisquelle kann beispielsweise ein Button sein, der auf einen Mausklick reagiert, oder ein Fenster, das mitteilt, daß es über das Systemmenü geschlossen werden soll. Die Reaktion auf diese Nachrichten erfolgt in den speziellen *Ereignisempfängern* (den *EventListeners*); das sind Objekte, die das zum Ereignis passende Empfänger-Interface implementieren. Damit ein Ereignisempfänger die Nachrichten einer bestimmten Ereignisquelle erhält, muß er sich bei dieser registrieren.

Dieses Kommunikationsmodell nennt sich *Delegation Event Model* oder *Delegation Based Event Handling* und wurde mit der Version 1.1 des JDK eingeführt. Im Gegensatz zum alten Modell, bei dem jedes Ereignis die Verteilermethode `handleEvent` der Klasse `Component` durchlaufen mußte und von ihr an die verschiedenen Empfänger verteilt wurde, hat dieses neue Modell zwei wesentliche Vorteile:

- Es verringert den Nachrichtenverkehr, da nur noch die Ereignisse transportiert werden, für die es Empfänger gibt. Dadurch erhöht sich potentiell die Performance des Nachrichtentransports im AWT.
- Es erlaubt eine klare Trennung zwischen Programmcode zur Oberflächengestaltung und solchem zur Implementierung der Anwendungslogik. Es erleichtert dadurch die Erzeugung von robustem Code, der auch in großen Programmen den Entwurf sauber strukturierter Ereignishandler ermöglicht.

Diesen Vorteilen steht natürlich der etwas höhere Einarbeitungsaufwand gegenüber. Während man beispielsweise im AWT 1.0 einfach die `action`-Methode des Fensters überlagerte, um auf einen Buttonklick zu reagieren, muß man nun eine `EventListener`-Klasse schreiben, instanzieren und bei der Ereignisquelle registrieren.

Zudem werden wir feststellen, daß das neue Modell eine Vielzahl von unterschiedlichen Möglichkeiten impliziert, Ereignishandler zu implementieren. Diese sind je nach Anwendungsfall unterschiedlich gut oder schlecht geeignet, den jeweiligen Nachrichtenverkehr abzubilden. Wir werden uns in diesem Kapitel insbesondere mit folgenden Entwurfsmustern beschäftigen:

- Die Fensterklasse implementiert die erforderlichen `EventListener`-Interfaces, stellt die erforderlichen Callback-Methoden zur Verfügung und registriert sich selbst bei den Ereignisquellen.
- In der Fensterklasse werden lokale oder anonyme Klassen definiert, die einen `EventListener` implementieren oder sich aus einer

Adapterklasse ableiten (eine Adapterklasse implementiert ein Interface mit mehreren Methoden und erlaubt es somit abgeleiteten Klassen, nur noch die Methoden zu überlagern, die tatsächlich von Interesse sind).

- GUI-Code und Ereignisbehandlung werden vollkommen getrennt und auf unterschiedliche Klassen verteilt.
- In der Komponentenkasse werden die Methoden überlagert, die für das Empfangen und Verteilen der Nachrichten erforderlich sind.

Wir werden uns jede dieser Varianten in den nachfolgenden Abschnitten ansehen und ihre jeweiligen Einsatzmöglichkeiten anhand eines Beispiels aufzeigen. Zunächst sollen jedoch die verschiedenen Ereignistypen sowie die Ereignisempfänger und Ereignisquellen näher beleuchtet werden. Wir wollen dabei (etwas unscharf) die Begriffe Ereignis, Nachricht und Event in diesem Kapitel synonym verwenden, wenn wir innerhalb eines GUI-Programms die ausgetauschten Nachrichten oder ihre auslösenden Ereignisse meinen.

Das vorliegende Kapitel beschränkt sich auf das Erläutern der grundlegenden Eigenschaften des Event-Handlings im JDK 1.1. Die Details einzelner Ereignisarten werden in den nachfolgenden Kapiteln schrittweise erklärt und zusammen mit den zugehörigen Ereignisquellen vorgestellt.

Hinweis

18.1.2 Ereignistypen

Im Gegensatz zur Version 1.0 werden im JDK 1.1 die Ereignistypen nicht mehr durch eine einzige Klasse [Event](#) repräsentiert, sondern durch eine Hierarchie von Ereignisklassen, die aus der Klasse [java.util.EventObject](#) abgeleitet sind. Die Motivation der Java-Designer, diese Klasse in das Paket [java.util](#) zu legen, resultierte wohl aus der Überlegung, daß der Transfer von Nachrichten nicht allein auf den Oberflächenteil beschränkt sein muß, sondern auch zwischen anderen Elementen einer komplexen Anwendung sinnvoll sein kann. Die Klasse [java.util.EventObject](#) fungiert damit als allgemeine Oberklasse aller Arten von Ereignissen, die zwischen verschiedenen Programmteilen ausgetauscht werden können. Ihre einzige nennenswerte Fähigkeit besteht darin, das Objekt zu speichern, das die Nachricht ausgelöst hat, und durch Aufruf der Methode [getSource](#) anzugeben:

[java.util.EventObject](#)

```
public Object getSource()
```

Die Hierarchie der AWT-spezifischen Ereignisklassen beginnt eine Ebene tiefer mit der Klasse [AWTEvent](#), die aus [EventObject](#) abgeleitet wurde und sich im Paket [java.awt](#) befindet. [AWTEvent](#) ist Oberklasse aller Ereignisklassen des AWT. Diese befinden sich im Paket [java.awt.event](#), das damit praktisch in jede Klasse einzubinden ist, die sich mit dem Event-Handling von GUI-Anwendungen beschäftigt. [Abbildung 18.1](#) gibt einen Überblick über die Vererbungshierarchie der Ereignisklassen.

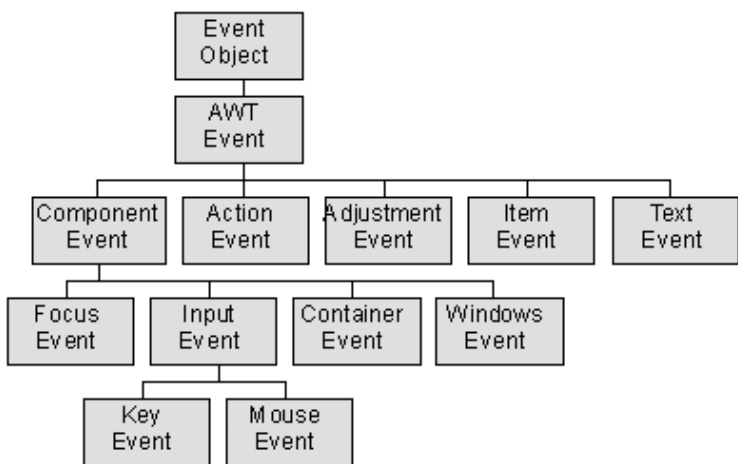


Abbildung 18.1: Die Hierarchie der Ereignisklassen

Die Dokumentation zum JDK unterteilt diese Klassen in zwei große Hierarchien. Unterhalb der Klasse [ComponentEvent](#) befinden sich alle *Low-Level-Ereignisse*. Sie sind für den Transfer von elementaren Nachrichten zuständig, die von Fenstern oder Dialogelementen stammen. Die übrigen Klassen [ActionEvent](#), [AdjustmentEvent](#), [ItemEvent](#) und [TextEvent](#) werden als *semantische Ereignisse* bezeichnet. Sie sind nicht an ein bestimmtes GUI-Element gebunden, sondern übermitteln höherwertige Ereignisse wie das Ausführen eines Kommandos oder die Änderung eines Zustands.

Ein Problem der AWT-Designer war es, einen guten Kompromiß zwischen der Größe und der Anzahl der zu implementierenden Ereignisklassen zu finden. Da der Ableitungsbaum sehr unübersichtlich geworden wäre, wenn für jedes Elementarereignis eine eigene Klasse implementiert worden wäre, hat man sich teilweise dazu entschlossen, mehrere unterschiedliche Ereignisse durch eine einzige Klasse zu realisieren. So ist beispielsweise die Klasse `MouseEvent` sowohl für Mausbewegungen als auch für alle Arten von Klick- oder Drag-Ereignissen zuständig. Mit Hilfe der Methode `getID` und der in den jeweiligen Eventklassen definierten symbolischen Konstanten kann das Programm herausfinden, um welche Art von Ereignis es sich handelt:

Hinweis

`java.awt.AWTEvent`

```
public int getID()
```

Falls das Programm eigene Eventklassen definieren will und Konstanten zur Vergabe der Event-Ids vergeben muß, sollten diese oberhalb der symbolischen Konstante `RESERVED_ID_MAX` liegen.

Die Eventklassen im JDK 1.1 enthalten keine frei zugänglichen Felder. Statt dessen sind alle Eigenschaften durch `set-/get`-Methoden gekapselt, die je nach Ereignisklasse unterschiedlich sind. So gibt es beispielsweise in `MouseEvent` die Methode `getPoint`, mit der die Mauszeigerposition abgefragt werden kann, in `KeyEvent` die Methode `getKeyChar` zur Bestimmung der gedrückten Taste und in der übergeordneten Klasse `InputEvent` die Methode `getModifiers`, mit der sowohl für Maus- als auch für Tastaturevents der Zustand der Sondertasten bestimmt werden kann. Wir werden bei der Besprechung der einzelnen Ereignisarten auf die Details eingehen.

Hinweis

18.1.3 Ereignisempfänger

Damit ein Objekt Nachrichten empfangen kann, muß es eine Reihe von Methoden implementieren, die von der Nachrichtenquelle, bei der es sich registriert hat, aufgerufen werden können. Um sicherzustellen, daß diese Methoden vorhanden sind, müssen die Ereignisempfänger bestimmte Interfaces implementieren, die aus der Klasse `EventListener` des Pakets `java.util` abgeleitet sind. Diese `EventListener`-Interfaces befinden sich im Paket `java.awt.events`.

Je Ereignisklasse gibt es ein `EventListener`-Interface. Es definiert eine separate Methode für jede Ereignisart dieser Ereignisklasse. So besitzt beispielsweise das Interface `MouseListener` die Methoden `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed` und `mouseReleased`, die bei Auftreten des jeweiligen Ereignisses aufgerufen werden. [Abbildung 18.2](#) gibt eine Übersicht über die Hierarchie der `EventListener`-Interfaces.

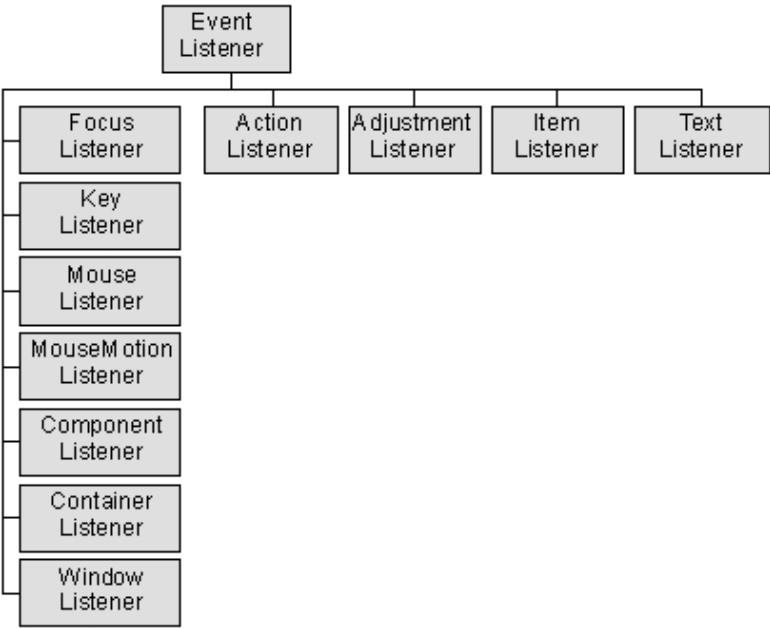


Abbildung 18.2: Die Hierarchie der EventListener-Interfaces

Jede der Methoden eines Listener-Interfaces enthält als einziges Argument ein Objekt vom zugehörigen Ereignistyp. Alle Methoden sind vom Typ `void`, erzeugen also keinen Rückgabewert. Dies steht in strengem Kontrast zu den Ereignishandlern des AWT 1.0, bei denen der Rückgabewert darüber entschieden hat, ob das Ereignis in der Vererbungshierarchie weiter nach oben gereicht werden sollte oder nicht. Im AWT 1.1 werden Ereignisse dagegen grundsätzlich nicht automatisch weitergereicht.

Hinweis

18.1.4 Ereignisquellen

Die Ereignisse stammen von den *Ereignisquellen*, also von Fenstern, Dialogelementen oder höheren Programmobjekten. Eine Ereignisquelle sendet aber nur dann Ereignisse an einen Ereignisempfänger, wenn dieser sich bei der Ereignisquelle *registriert* hat. Fehlt eine solche Registrierung, wird die Ereignisquelle keine Ereignisse senden und der Empfänger folglich auch keine erhalten.

Die Registrierung erfolgt mit speziellen Methoden, an die ein Objekt übergeben wird, das das jeweilige `EventListener`-Interface implementiert. So gibt es beispielsweise eine Methode `addMouseListener` in der Klasse `Component`, mit der ein Objekt, das das Interface `MouseListener` implementiert, sich für den Empfang von Mausereignissen bei der Komponente registrieren lassen kann. Nach erfolgter Registrierung wird bei jedem Mausereignis die entsprechende Methode `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed` oder `mouseReleased` aufgerufen.

Die Ereignisquellen unterstützen das Multicasting von Ereignissen. Dabei kann eine Ereignisquelle nicht nur einen einzelnen `EventListener` mit Nachrichten versorgen, sondern eine beliebige Anzahl von ihnen. Jeder Aufruf von `addXYZListener` registriert dabei einen weiteren Listener in der Ereignisquelle. Das AWT definiert nicht, in welcher Reihenfolge die verschiedenen Ereignisempfänger beim Auftreten eines Ereignisses aufgerufen werden und empfiehlt den Programmen, hierüber keine Annahmen zu treffen.

Hinweis

18.1.5 Adapterklassen

Eine *Adapterklasse* in Java ist eine Klasse, die ein vorgegebenes Interface mit leeren Methodenrumpfen implementiert. Adapterklassen können verwendet werden, wenn aus einem Interface lediglich ein Teil der Methoden benötigt wird, der Rest aber uninteressant ist. In diesem Fall leitet man einfach eine neue Klasse aus der Adapterklasse ab, anstatt das zugehörige Interface zu implementieren, und überlagert die benötigten Methoden. Alle übrigen Methoden des Interfaces werden von der Basisklasse zur Verfügung gestellt.

Zu jedem der Low-Level-Ereignisempfänger stellt das Paket `java.awt.event` eine passende Adapterklasse zur Verfügung. So gibt es die Adapterklassen `FocusAdapter`, `KeyAdapter`, `MouseAdapter`, `MouseMotionAdapter`, `ComponentAdapter`, `ContainerAdapter` und `WindowAdapter`. Sie implementieren die korrespondierenden Interfaces. Wir werden später bei der Beschreibung der lokalen und anonymen Klassen sehen, wie die Adapterklassen bei der Realisierung der Ereignisempfänger hilfreich sein können.

18.1.6 Zusammenfassung

Der folgende Überblick faßt die bisherigen Ausführungen zusammen und gibt eine Zusammenfassung aller Ereignisse nebst zugehörigen Ereignisempfängern und ihren Methoden. Außerdem sind die Ereignisquellen und die Methoden zur Registrierung der Ereignisse dargestellt. Jeder Ereignistyp wird dabei durch zwei Tabellen dargestellt. Die erste gibt die zugehörige Ereignisklasse, das Listener-Interface und den Namen der Methode zur Registrierung von Ereignisempfängern an. Sie listet außerdem die als Ereignisquelle in Frage kommenden Klassen auf. Die zweite Tabelle listet alle Methoden des zugehörigen Listener-Interfaces auf und beschreibt damit die zu diesem Ereignistyp gehörenden Elementarereignisse.

Genau genommen nimmt dieser Abschnitt Informationen vorweg, die in späteren Teilen des Buches konkretisiert werden. Obwohl zum jetzigen Zeitpunkt nicht alle Ereignisse in ihrer vollen Bedeutung dargestellt werden können, mag diese Übersicht für die weitere Arbeit mit dem AWT und zum Nachschlagen hilfreich sein.

Hinweis

Focus-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	<code>FocusEvent</code>
Listener-Interface	<code>FocusListener</code>
Registrierungsmethode	<code>addFocusListener</code>
Mögliche Ereignisquellen	<code>Component</code>

Tabelle 18.1: Focus-Ereignisse

Ereignismethode	Bedeutung
<code>focusGained</code>	Eine Komponente erhält den Focus.

focusLost	Eine Komponente verliert den Focus.
---------------------------	-------------------------------------

Tabelle 18.2: Methoden für Focus-Ereignisse

Key-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	KeyEvent
Listener-Interface	KeyListener
Registrierungsmethode	addKeyListener
Mögliche Ereignisquellen	Component

Tabelle 18.3: Key-Ereignisse

Ereignismethode	Bedeutung
keyPressed	Eine Taste wurde gedrückt.
keyReleased	Eine Taste wurde losgelassen.
keyTyped	Eine Taste wurde gedrückt und wieder losgelassen.

Tabelle 18.4: Methoden für Key-Ereignisse

Mouse-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	MouseEvent
Listener-Interface	MouseListener
Registrierungsmethode	addMouseListener
Mögliche Ereignisquellen	Component

Tabelle 18.5: Mouse-Ereignisse

Ereignismethode	Bedeutung
mouseClicked	Eine Maustaste wurde gedrückt und wieder losgelassen.
mouseEntered	Der Mauszeiger betritt die Komponente.
mouseExited	Der Mauszeiger verläßt die Komponente.
mousePressed	Eine Maustaste wurde gedrückt.
mouseReleased	Eine Maustaste wurde losgelassen.

Tabelle 18.6: Methoden für Mouse-Ereignisse

MouseMotion-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	MouseEvent
Listener-Interface	MouseMotionListener
Registrierungsmethode	addMouseMotionListener
Mögliche Ereignisquellen	Component

Tabelle 18.7: MouseMotion-Ereignisse

Ereignismethode	Bedeutung
mouseDragged	Die Maus wurde bei gedrückter Taste bewegt.
mouseMoved	Die Maus wurde bewegt, ohne daß eine Taste gedrückt wurde.

Tabelle 18.8: Methoden für MouseMotion-Ereignisse

Component-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ComponentEvent
Listener-Interface	ComponentListener
Registrierungsmethode	addComponentListener
Mögliche Ereignisquellen	Component

Tabelle 18.9: Komponenten-Ereignisse

Ereignismethode	Bedeutung
componentHidden	Eine Komponente wurde unsichtbar.
componentMoved	Eine Komponente wurde verschoben.
componentResized	Die Größe einer Komponente hat sich geändert.
componentShown	Eine Komponente wurde sichtbar.

Tabelle 18.10: Methoden für Komponenten-Ereignisse

Container-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ContainerEvent
Listener-Interface	ContainerListener
Registrierungsmethode	addContainerListener
Mögliche Ereignisquellen	Container

Tabelle 18.11: Container-Ereignisse

Ereignismethode	Bedeutung
componentAdded	Eine Komponente wurde hinzugefügt.
componentRemoved	Eine Komponente wurde entfernt.

Tabelle 18.12: Methoden für Container-Ereignisse

Window-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	WindowEvent
Listener-Interface	WindowListener
Registrierungsmethode	addWindowListener
Mögliche Ereignisquellen	Dialog , Frame

Tabelle 18.13: Window-Ereignisse

Ereignismethode	Bedeutung
windowActivated	Das Fenster wurde aktiviert.
windowClosed	Das Fenster wurde geschlossen.
windowClosing	Das Fenster wird geschlossen.
windowDeactivated	Das Fenster wurde deaktiviert.
windowDeiconified	Das Fenster wurde wiederhergestellt.
windowIconified	Das Fenster wurde auf Symbolgröße verkleinert.
windowOpened	Das Fenster wurde geöffnet.

Tabelle 18.14: Methoden für Window-Ereignisse

Action-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ActionEvent
Listener-Interface	ActionListener
Registrierungsmethode	addActionListener
Mögliche Ereignisquellen	Button , List , MenuItem , TextField

Tabelle 18.15: Action-Ereignisse

Ereignismethode	Bedeutung
actionPerformed	Eine Aktion wurde ausgelöst.

Tabelle 18.16: Methode für Action-Ereignisse

Adjustment-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	AdjustmentEvent
Listener-Interface	AdjustmentListener
Registrierungsmethode	addAdjustmentListener
Mögliche Ereignisquellen	Scrollbar

Tabelle 18.17: Adjustment-Ereignisse

Ereignismethode	Bedeutung
adjustmentValueChanged	Der Wert wurde verändert.

Tabelle 18.18: Methode für Adjustment-Ereignisse

Item-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	ItemEvent
Listener-Interface	ItemListener
Registrierungsmethode	addItemListener
Mögliche Ereignisquellen	Checkbox , Choice , List , CheckboxMenuItem

Tabelle 18.19: Item-Ereignisse

Ereignismethode	Bedeutung
itemStateChanged	Der Zustand hat sich verändert.

Tabelle 18.20: Methode für Item-Ereignisse

Text-Ereignisse

Eigenschaft	Klasse, Interface oder Methode
Ereignisklasse	TextEvent
Listener-Interface	TextListener
Registrierungsmethode	addTextListener
Mögliche Ereignisquellen	TextField , TextArea

Tabelle 18.21: Text-Ereignisse

Ereignismethode	Bedeutung
textValueChanged	Der Text wurde verändert.

Tabelle 18.22: Methode für Text-Ereignisse

18.2 Entwurfsmuster für den Nachrichtenverkehr

- [18.2 Entwurfsmuster für den Nachrichtenverkehr](#)
 - [18.2.1 Variante 1: Implementierung eines EventListener-Interfaces](#)
 - [18.2.2 Variante 2: Lokale und anonyme Klassen](#)
 - [Lokale Klassen](#)
 - [Anonyme Klassen](#)
 - [18.2.3 Variante 3: Trennung von GUI- und Anwendungscode](#)
 - [18.2.4 Variante 4: Überlagern der Event-Handler in den Komponenten](#)
 - [18.2.5 Ausblick](#)

Wir wollen uns in diesem Abschnitt damit beschäftigen, die oben erwähnten Entwurfsmuster für die Abwicklung des Nachrichtenverkehrs in Java-Programmen vorzustellen. Wie bereits erwähnt, hat jedes dieser Verfahren seine ganz spezifischen Vor- und Nachteile und ist für verschiedene Programmieraufgaben unterschiedlich gut geeignet.

Als Basis für unsere Experimente wollen wir ein einfaches Programm schreiben, das die folgenden Anforderungen erfüllt:

- Nach dem Start soll das Programm ein Fenster mit dem Titel »Nachrichtentransfer« auf dem Bildschirm anzeigen.
- Das Fenster soll einen grauen Hintergrund haben und in der Client-Area die Meldung »Zum Beenden bitte ESC drücken ...« anzeigen.
- Nach Drücken der Taste `[ESC]` soll das Fenster geschlossen und das Programm beendet werden. Andere Tastendrucke, Mausklicks oder ähnliche Ereignisse werden ignoriert.

Beispiel

Basis der Programme ist das folgende Listing:

[Listing1801.java](#)

```

001 /* Listing1801.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1801
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1801 wnd = new Listing1801();
012     }
013
014     public Listing1801()
015     {
016         super("Nachrichtentransfer");
017         setBackground(Color.lightGray);
018         setSize(300,200);
019         setLocation(200,100);
020         setVisible(true);
021     }
022
023     public void paint(Graphics g)
024     {
025         g.setFont(new Font("Serif",Font.PLAIN,18));
026         g.drawString("Zum Beenden bitte ESC drücken...",10,50);
027     }
028 }

```

Listing 18.1: Basisprogramm für den Nachrichtentransfer

Das Programm erfüllt die ersten der oben genannten Anforderungen, ist aber mangels Event-Handler noch nicht in der Lage, per `[ESC]` beendet zu werden. Um die Nachfolgeversionen vorzubereiten, haben wir bereits die Anweisung `import java.awt.event.*` eingefügt.

Hinweis

Die Ausgabe des Programms ist:



Abbildung 18.3: Das Programm für den Nachrichtentransfer

18.2.1 Variante 1: Implementierung eines EventListener-Interfaces

Bei der ersten Variante gibt es nur eine einzige Klasse, `Listing1802`. Sie ist einerseits eine Ableitung der Klasse `Frame`, um ein Fenster auf dem Bildschirm darzustellen und zu beschriften. Andererseits implementiert sie das Interface `KeyListener`, das die Methoden `keyPressed`, `keyReleased` und `keyTyped` definiert. Der eigentliche Code zur Reaktion auf die Taste `[ESC]` steckt in der Methode `keyPressed`, die immer dann aufgerufen wird, wenn eine Taste gedrückt wurde. Mit der Methode `getKeyCode` der Klasse `KeyEvent` wird auf den Code der gedrückten Taste zugegriffen und dieser mit der symbolischen Konstante `VK_ESCAPE` verglichen. Stimmen beide überein, wurde `[ESC]` gedrückt, und das Programm kann beendet werden.

[Listing1802.java](#)

```
001 /* Listing1802.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1802
007     extends Frame
008     implements KeyListener
009 {
010     public static void main(String[] args)
011     {
012         Listing1802 wnd = new Listing1802();
013     }
014
015     public Listing1802()
016     {
017         super("Nachrichtentransfer");
018         setBackground(Color.lightGray);
019         setSize(300,200);
020         setLocation(200,100);
021         setVisible(true);
022         addKeyListener(this);
023     }
024
025     public void paint(Graphics g)
026     {
027         g.setFont(new Font("Serif",Font.PLAIN,18));
028         g.drawString("Zum Beenden bitte ESC drücken...",10,50);
029     }
030
031     public void keyPressed(KeyEvent event)
032     {
033         if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
034             setVisible(false);
```

```

035         dispose();
036         System.exit(0);
037     }
038 }
039
040 public void keyReleased(KeyEvent event)
041 {
042 }
043
044 public void keyTyped(KeyEvent event)
045 {
046 }
047 }

```

Listing 18.2: Implementieren eines Listener-Interfaces

Die Verbindung zwischen der Ereignisquelle (in diesem Fall der Fensterklasse [Listing1802](#)) und dem Ereignisempfänger (ebenfalls die Klasse [Listing1802](#)) erfolgt über den Aufruf der Methode [addKeyListener](#) der Klasse [Frame](#). Alle Tastaturereignisse werden dadurch an die Fensterklasse selbst weitergeleitet und führen zum Aufruf der Methoden [keyPressed](#), [keyReleased](#) oder [keyTyped](#) des Interfaces [KeyListener](#). Diese Implementierung ist sehr naheliegend, denn sie ist einfach zu implementieren und erfordert keine weiteren Klassen.

Nachteilig ist dabei allerdings:

- Es besteht keine Trennung zwischen GUI-Code und Applikationslogik. Dies kann große Programme unübersichtlich und schwer wartbar machen.
- Für jeden Ereignistyp muß eine passende Listener-Klasse registriert werden. Da viele der [EventListener](#)-Interfaces mehr als eine Methode definieren, werden dadurch schnell viele leere Methodenrumpfe in der Fensterklasse zu finden sein. In diesem Beispiel sind es schon [keyReleased](#) und [keyTyped](#), bei zusätzlichen Interfaces würden schnell weitere hinzukommen.

Es bleibt festzuhalten, daß diese Technik bestenfalls für kleine Programme geeignet ist, die nur begrenzt erweitert werden müssen. Durch die Vielzahl leerer Methodenrumpfe können aber auch kleine Programme schnell unübersichtlich werden.

18.2.2 Variante 2: Lokale und anonyme Klassen

Die zweite Alternative ist eine bessere Lösung. Sie basiert auf der Verwendung *lokaler* bzw. *anonymer* Klassen und kommt ohne die Nachteile der vorigen Version aus. Sie ist das in der Dokumentation des JDK empfohlene Entwurfsmuster für das Event-Handling in kleinen Programmen oder bei Komponenten mit einfacher Nachrichtenstruktur. Vor ihrem Einsatz sollte man allerdings das Prinzip lokaler und anonymer Klassen kennenlernen, das mit dem JDK 1.1 in Java eingeführt wurde. Wir wollen es hier kurz vorstellen, uns dabei aber lediglich mit den Grundzügen dieser Technik beschäftigen, um den Einsatz für die Ereignisbehandlung aufzuzeigen. Ein weiteres Beispiel für die Verwendung lokaler Klassen findet sich in [Abschnitt 27.4](#).

Lokale Klassen

Bis zum JDK 1.0 wurden Klassen immer auf der Paketebene definiert, eine Schachtelung war nicht möglich. Seit JDK 1.1 gibt es die Möglichkeit, innerhalb einer bestehenden Klasse *X* eine neue Klasse *Y* zu definieren (im JDK wird das gesamte Konzept als *Inner Classes* bezeichnet). Diese Klasse unterliegt lexikalischen Sichtbarkeitsregeln und ist nur innerhalb von *X* sichtbar. Objektinstanzen von *Y* können damit auch nur aus *X* erzeugt werden. Anders herum (und das macht die lokalen Klassen für das Event-Handling interessant) kann *Y* auf alle Membervariablen von *X* zugreifen. Bei der Instanzierung wird (neben einem impliziten [this](#)-Zeiger) ein weiterer Verweis auf die erzeugende Instanz der umschließenden Klasse übergeben, der es ermöglicht, auf sie zuzugreifen.

Die Anwendung lokaler Klassen für die Ereignisbehandlung besteht darin, mit ihrer Hilfe die benötigten [EventListener](#) zu implementieren. Dazu wird in dem GUI-Objekt, das einen Event-Handler benötigt, eine lokale Klasse definiert und aus einer passenden *Adapterklasse* abgeleitet. Nun braucht nicht mehr das gesamte Interface implementiert zu werden (denn die Methodenrumpfe werden ja aus der Adapterklasse geerbt), sondern lediglich die tatsächlich benötigten Methoden. Da die lokale Klasse zudem auf die Membervariablen und Methoden der Klasse zugreifen kann, in der sie definiert wurde, lassen sich auf diese Weise sehr schnell die benötigten Ereignisempfänger zusammenbauen.

Das folgende Beispiel definiert eine lokale Klasse [MyKeyListener](#), die aus [KeyAdapter](#) abgeleitet wurde und auf diese Weise das [KeyListener](#)-Interface implementiert. Sie überlagert lediglich die Methode [keyPressed](#), um auf das Drücken einer Taste zu reagieren. Als lokale Klasse hat sie außerdem Zugriff auf die Methoden der umgebenden Klasse und kann somit durch Aufruf von [setVisible](#) und [dispose](#) das Fenster, in dem sie als Ereignisempfänger registriert wurde, schließen. Die Registrierung der lokalen Klasse erfolgt durch Aufruf von [addKeyListener](#), bei dem gleichzeitig eine Instanz der lokalen Klasse erzeugt wird. Als lokale Klasse ist [MyKeyListener](#) überall innerhalb von [Listing1803](#) sichtbar und kann an beliebiger Stelle instanziiert werden.

Beispiel

```

001 /* Listing1803.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1803
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1803 wnd = new Listing1803();
012     }
013
014     public Listing1803()
015     {
016         super("Nachrichtentransfer");
017         setBackground(Color.lightGray);
018         setSize(300,200);
019         setLocation(200,100);
020         setVisible(true);
021         addKeyListener(new MyKeyListener());
022     }
023
024     public void paint(Graphics g)
025     {
026         g.setFont(new Font("Serif",Font.PLAIN,18));
027         g.drawString("Zum Beenden bitte ESC drücken...",10,50);
028     }
029
030     class MyKeyListener
031     extends KeyAdapter
032     {
033         public void keyPressed(KeyEvent event)
034         {
035             if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
036                 setVisible(false);
037                 dispose();
038                 System.exit(0);
039             }
040         }
041     }
042 }

```

Listing 18.3: Verwendung lokaler Klassen

Der Vorteil dieser Vorgehensweise ist ganz offensichtlich: es werden keine unnützen Methodenrümpfe erzeugt, aber trotzdem verbleibt der Ereignisempfängercode wie im vorigen Beispiel innerhalb der Ereignisquelle. Dieses Verfahren ist also immer dann gut geeignet, wenn es von der Architektur oder der Komplexität der Ereignisbehandlung her sinnvoll ist, Quelle und Empfänger zusammenzufassen.

Anonyme Klassen

Eine Variante der lokalen Klassen sind die *anonymen Klassen*. Sie werden ebenfalls lokal zu einer anderen Klasse erzeugt, kommen aber ohne Klassennamen aus. Dazu werden sie bei der Übergabe eines Objektes an eine Methode oder als Rückgabewert einer Methode innerhalb einer einzigen Anweisung definiert und instanziiert. Damit eine anonyme Klasse überhaupt irgendeiner sinnvollen Aufgabe zugeführt werden kann, muß sie aus einer anderen Klasse abgeleitet sein oder ein bestehendes Interface implementieren.

Sowohl einfache lokale Klassen als auch anonyme lokale Klassen wurden realisiert, ohne daß eine Änderung der virtuellen Maschine erforderlich war. Statt dessen wurde die Implementierung dieses Konzepts vollständig dem Java-Compiler übertragen. Im Falle einer lokalen Klasse *Y* innerhalb einer Klasse *X* erzeugt der Compiler beim Übersetzen der Quelldatei Klassennamen der Art *X\$Y.class*. Im Falle einer anonymen Klasse innerhalb von *X* erzeugt er Code der Art *X\$1.class*, *X\$2.class* usw. Das Laufzeitsystem interpretiert die lokale Klasse wie eine nicht-lokale.

Hinweis

Das folgende Beispiel ist eine leichte Variation des vorigen Beispiels. Es zeigt die Verwendung einer anonymen Klasse, die aus [KeyAdapter](#) abgeleitet wurde, als Ereignisempfänger. Zum Instanzierungszeitpunkt erfolgt die Definition der überlagernden Methode [keyPressed](#), in der der Code zur Reaktion auf das Drücken der Taste `[ESC]` untergebracht wird.

[Listing1804.java](#)

```

001 /* Listing1804.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1804
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1804 wnd = new Listing1804();
012     }
013
014     public Listing1804()
015     {
016         super("Nachrichtentransfer");
017         setBackground(Color.lightGray);
018         setSize(300,200);
019         setLocation(200,100);
020         setVisible(true);
021         addKeyListener(
022             new KeyAdapter() {
023                 public void keyPressed(KeyEvent event)
024                 {
025                     if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
026                         setVisible(false);
027                         dispose();
028                         System.exit(0);
029                     }
030                 }
031             }
032         );
033     }
034
035     public void paint(Graphics g)
036     {
037         g.setFont(new Font("Serif",Font.PLAIN,18));
038         g.drawString("Zum Beenden bitte ESC drücken...",10,50);
039     }
040
041 }

```

Listing 18.4: Verwendung anonymer Klassen

Vorteilhaft bei dieser Vorgehensweise ist der verringerte Aufwand, denn es muß keine separate Klassendefinition angelegt werden. Statt dessen werden die wenigen Codezeilen, die zur Anpassung der Adapterklasse erforderlich sind, dort eingefügt, wo die Klasse instanziiert wird, nämlich beim Registrieren des Nachrichtenempfängers. Anonyme Klassen haben einen ähnlichen Einsatzbereich wie lokale, empfehlen sich aber vor allem, wenn sehr wenig Code für den Ereignisempfänger benötigt wird. Bei aufwendigeren Ereignisempfängern ist die explizite Definition einer benannten Klasse dagegen vorzuziehen.

18.2.3 Variante 3: Trennung von GUI- und Anwendungscode

Wir hatten am Anfang darauf hingewiesen, daß in größeren Programmen eine Trennung zwischen Programmcode, der für die Oberfläche zuständig ist, und solchem, der für die Anwendungslogik zuständig ist, wünschenswert wäre. Dadurch wird eine bessere Modularisierung des Programms erreicht, und der Austausch oder die Erweiterung von Teilen des Programms wird erleichtert.

Das Delegation Event Model wurde auch mit dem Designziel entworfen, eine solche Trennung **Hinweis** zu ermöglichen bzw. zu erleichtern. Der Grundgedanke dabei war es, auch Nicht-Komponenten die Reaktion auf GUI-Events zu ermöglichen. Dies wurde dadurch erreicht, daß jede Art von Objekt als Ereignisempfänger registriert werden kann, solange es die erforderlichen Listener-Interfaces implementiert. Damit ist es möglich, die Anwendungslogik vollkommen von der grafischen Oberfläche abzulösen und in Klassen zu verlagern, die eigens für diesen Zweck entworfen wurden.

Das nachfolgende Beispiel zeigt diese Vorgehensweise, indem es unser Beispielprogramm in die drei Klassen [Listing1805](#), [MainFrameCommand](#) und [MainFrameGUI](#) aufteilt. [Listing1805](#) enthält nur noch die [main](#)-Methode und dient lediglich dazu, die anderen beiden Klassen zu instanzieren. [MainFrameGUI](#) realisiert die GUI-Funktionalität und stellt das Fenster auf dem Bildschirm dar. [MainFrameCommand](#) spielt die Rolle des Kommandointerpreters, der immer dann aufgerufen wird, wenn im Fenster ein Tastaturereignis aufgetreten ist.

Die Verbindung zwischen beiden Klassen erfolgt durch Aufruf der Methode [addKeyListener](#) in [MainFrameGUI](#), an die das an den Konstruktor übergebene [MainFrameCommand](#)-Objekt weitergereicht wird. Dazu ist es erforderlich, daß das Hauptprogramm den Ereignisempfänger `cmd` zuerst instanziert, um ihn bei der Instanzierung des GUI-Objekts `gui` übergeben zu können.

Umgekehrt benötigt natürlich auch das Kommando-Objekt Kenntnis über das GUI-Objekt, denn es **Tip** soll ja das zugeordnete Fenster schließen und das Programm beenden. Der scheinbare Instanzierungskonflikt durch diese zirkuläre Beziehung ist aber in Wirklichkeit gar keiner, denn bei jedem Aufruf einer der Methoden von [MainFrameCommand](#) wird an das [KeyEvent](#)-Objekt der Auslöser der Nachricht übergeben, und das ist in diesem Fall stets das [MainFrameGUI](#)-Objekt `gui`. So kann innerhalb des Kommando-Objekts auf alle öffentlichen Methoden des GUI-Objekts zugegriffen werden.

[Listing1805.java](#)

```
001 /* Listing1805.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1805
007 {
008     public static void main(String[] args)
009     {
010         MainFrameCommand cmd = new MainFrameCommand();
011         MainFrameGUI      gui = new MainFrameGUI(cmd);
012     }
013 }
014
015 class MainFrameGUI
016 extends Frame
017 {
018     public MainFrameGUI(KeyListener cmd)
019     {
020         super("Nachrichtentransfer");
021         setBackground(Color.lightGray);
022         setSize(300,200);
023         setLocation(200,100);
024         setVisible(true);
025         addKeyListener(cmd);
026     }
027
028     public void paint(Graphics g)
029     {
030         g.setFont(new Font("Serif",Font.PLAIN,18));
031         g.drawString("Zum Beenden bitte ESC drücken...",10,50);
032     }
033 }
034
035 class MainFrameCommand
036 implements KeyListener
037 {
038     public void keyPressed(KeyEvent event)
039     {
040         Frame source = (Frame)event.getSource();
```

```

041         if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
042             source.setVisible(false);
043             source.dispose();
044             System.exit(0);
045         }
046     }
047
048     public void keyReleased(KeyEvent event)
049     {
050     }
051
052     public void keyTyped(KeyEvent event)
053     {
054     }
055 }

```

Listing 18.5: Trennung von GUI- und Anwendungslogik

Diese Designvariante ist vorwiegend für größere Programme geeignet, bei denen eine Trennung von Programmlogik und Oberfläche sinnvoll ist. Für sehr kleine Programme oder solche, die wenig Ereigniscode haben, sollte eher eine der vorherigen Varianten angewendet werden, wenn diese zu aufwendig ist.

Natürlich erhebt das vorliegende Beispielprogramm nicht den Anspruch, unverändert in ein sehr großes Programm übernommen zu werden. Es soll lediglich die Möglichkeit der Trennung von Programmlogik und Oberfläche in einem großen Programm mit Hilfe der durch das Event-Handling des JDK 1.1 vorgegebenen Möglichkeiten aufzeigen. Eine sinnvolle Erweiterung dieses Konzepts könnte darin bestehen, weitere Modularisierungen vorzunehmen (z.B. analog dem *MVC-Konzept* von Smalltalk, bei dem GUI-Anwendungen in *Model*-, *View*- und *Controller*-Layer aufgesplittet werden, oder auch durch Abtrennen spezialisierter Kommandoklassen). Empfehlenswert ist in diesem Zusammenhang die Lektüre der JDK-Dokumentation, die ein ähnliches Beispiel in leicht veränderter Form enthält.

18.2.4 Variante 4: Überlagern der Event-Handler in den Komponenten

Als letzte Möglichkeit, auf Nachrichten zu reagieren, soll das Überlagern der Event-Handler in den Ereignisquellen selbst aufgezeigt werden. Jede Ereignisquelle besitzt eine Reihe von Methoden, die für das Aufbereiten und Verteilen der Nachrichten zuständig sind. Soll eine Nachricht weitergereicht werden, so wird dazu zunächst innerhalb der Nachrichtenquelle die Methode [processEvent](#) aufgerufen. Diese verteilt die Nachricht anhand ihres Typs an spezialisierte Methoden, deren Name sich nach dem Typ der zugehörigen Ereignisklasse richtet. So ist beispielsweise die Methode [processActionEvent](#) für das Handling von Action-Events und [processMouseEvent](#) für das Handling von Mouse-Events zuständig:

```

java.awt.Component
protected void processEvent(AWTEvent e)

protected void processComponentEvent(ComponentEvent e)

protected void processFocusEvent(FocusEvent e)

...

```

Beide Methodenarten können in einer abgeleiteten Klasse überlagert werden, um die zugehörigen Ereignisempfänger zu implementieren. Wichtig ist dabei, daß in der abgeleiteten Klasse die gleichnamige Methode der Basisklasse aufgerufen wird, um das Standardverhalten sicherzustellen. Wichtig ist weiterhin, daß sowohl [processEvent](#) als auch [processActionEvent](#) usw. nur aufgerufen werden, wenn der entsprechende Ereignistyp für diese Ereignisquelle aktiviert wurde. Dies passiert in folgenden Fällen:

- Wenn ein passender Ereignisempfänger über die zugehörige [addEventListener](#)-Methode registriert wurde.
- Wenn der Ereignistyp explizit durch Aufruf der Methode [enableEvents](#) aktiviert wurde.

Warnung

Die Methode [enableEvents](#) erwartet als Argument eine Maske, die durch eine bitweise Oder-Verknüpfung der passenden Maskenkonstanten aus der Klasse [AWTEvent](#) zusammengesetzt werden kann:

```

java.awt.Component
protected final void enableEvents(long eventsToEnable)

```

Die verfügbaren Masken sind analog zu den Ereignistypen benannt und heißen [ACTION_EVENT_MASK](#), [ADJUSTMENT_EVENT_MASK](#), [COMPONENT_EVENT_MASK](#) usw.

Das folgende Beispiel überlagert die Methode `processKeyEvent` in der Klasse `Frame` (die sie aus `Component` geerbt hat). Durch Aufruf von `enableEvents` wird die Weiterleitung der Tastaturereignisse aktiviert, und das Programm zeigt dasselbe Verhalten wie die vorigen Programme.

[Listing1806.java](#)

```

001 /* Listing1806.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1806
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1806 wnd = new Listing1806();
012     }
013
014     public Listing1806()
015     {
016         super("Nachrichtentransfer");
017         setBackground(Color.lightGray);
018         setSize(300,200);
019         setLocation(200,100);
020         setVisible(true);
021         enableEvents(AWTEvent.KEY_EVENT_MASK);
022     }
023
024     public void paint(Graphics g)
025     {
026         g.setFont(new Font("Serif",Font.PLAIN,18));
027         g.drawString("Zum Beenden bitte ESC drücken...",10,50);
028     }
029
030     public void processKeyEvent(KeyEvent event)
031     {
032         if (event.getID() == KeyEvent.KEY_PRESSED) {
033             if (event.getKeyCode() == KeyEvent.VK_ESCAPE) {
034                 setVisible(false);
035                 dispose();
036                 System.exit(0);
037             }
038         }
039         super.processKeyEvent(event);
040     }
041 }

```

Listing 18.6: Überlagern der Komponenten-Event-Handler

Diese Art der Ereignisbehandlung ist nur sinnvoll, wenn Fensterklassen oder Dialogelemente überlagert werden und ihr Aussehen oder Verhalten signifikant verändert wird. Alternativ könnte natürlich auch in diesem Fall ein `EventListener` implementiert und die entsprechenden Methoden im Konstruktor der abgeleiteten Klasse registriert werden.

Das hier vorgestellte Verfahren umgeht das Delegation Event Model vollständig und hat damit **Hinweis** die gleichen inhärenten Nachteile wie das Event-Handling des alten JDK. Die Dokumentation zum JDK empfiehlt daher ausdrücklich, für alle »normalen« Anwendungsfälle das Delegation Event Model zu verwenden und die Anwendungen nach einem der in den ersten drei Beispielen genannten Entwurfsmuster zu implementieren.

18.2.5 Ausblick

Die hier vorgestellten Entwurfsmuster geben einen Überblick über die wichtigsten Designtechniken für das Event-Handling in Java-Programmen. Während die ersten beiden Beispiele für kleine bis mittelgroße Programme gut geeignet sind, kommen die Vorteile der in Variante 3 vorgestellten Trennung zwischen GUI-Code und Anwendungslogik vor allem bei größeren Programmen zum Tragen. Die vierte Variante ist vornehmlich für Spezialfälle geeignet und sollte entsprechend umsichtig eingesetzt werden.

Wir werden in den nachfolgenden Kapiteln vorwiegend die ersten beiden Varianten einsetzen. Wenn es darum geht, Ereignishandler für die Beispielprogramme zu implementieren, werden wir also entweder die erforderlichen Listener-Interfaces in der Fensterklasse selbst implementieren oder sie in lokalen oder anonymen Klassen unterbringen. Da das Event-Handling des JDK 1.1 eine Vielzahl von Designvarianten erlaubt, werden wir uns nicht immer sklavisch an die vorgestellten Entwurfsmuster halten, sondern teilweise leicht davon abweichen oder Mischformen verwenden. Dies ist beabsichtigt und soll den möglichen Formenreichtum demonstrieren. Wo nötig, werden wir auf spezielle Implementierungsdetails gesondert eingehen.

[Kapitel 19](#) widmet sich den wichtigsten Low-Level-Events und demonstriert den genauen Einsatz ihrer Listener- und Event-Methoden anhand vieler Beispiele. In späteren Kapiteln werden die meisten der High-Level-Events erläutert. Sie werden in der Regel dort eingeführt, wo ihr Einsatz durch das korrespondierende Dialogelement motiviert wird. So erläutert [Kapitel 20](#) in Zusammenhang mit der Vorstellung von Menüs die Action-Ereignisse, und in [Kapitel 22](#) werden Ereignisse erläutert, die von den dort vorgestellten Dialogelementen ausgelöst werden.

18.3 Zusammenfassung

- [18.3 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Grundlagen des *Delegation Based Event Handling* und des Nachrichtenversands zwischen Teilen einer Anwendung
- Die Klasse [Event](#) und die daraus abgeleiteten Ereignisklassen
- Die Klasse [EventListener](#) und die daraus abgeleiteten Ereignisempfängerklassen
- Registrierung von Ereignisempfängern
- Verwendung von Adapter-Klassen
- Überblick über die *Focus*-, *Key*-, *Mouse*-, *MouseMotion*-, *Component*-, *Container*-, *Window*-, *Action*-, *Adjustment*-, *Item*- und *Text*-Ereignisse
- Die Implementierung eines [EventListener](#)-Interfaces
- Lokale und anonyme Klassen und ihre Verwendung zur Implementierung von Ereignisempfängern
- Die Trennung von GUI- und Anwendungscode
- Das Überlagern der Event-Handler in den Komponentenklassen

Kapitel 19

Low-Level-Events

- [19 Low-Level-Events](#)
 - [19.1 Window-Events](#)
 - [19.2 Component-Events](#)
 - [19.3 Mouse-Events](#)
 - [19.4 MouseMotion-Events](#)
 - [19.5 Focus-Events](#)
 - [19.6 Key-Events](#)
 - [19.7 Zusammenfassung](#)

19.1 Window-Events

- 19.1 Window-Events

Ein Window-Event wird immer dann generiert, wenn sich am Status eines Fensters eine Änderung ergeben hat, die für das Anwendungsprogramm von Interesse sein könnte. So erlangt das Programm beispielsweise Kenntnis davon, wenn ein Fenster erstellt oder zerstört, aktiviert oder deaktiviert oder wenn es symbolisiert oder wiederhergestellt wird.

Ein Empfänger für Window-Events muß das Interface [WindowListener](#) implementieren und bekommt Events des Typs [WindowEvent](#) übergeben. [WindowEvent](#) erweitert die Klasse [ComponentEvent](#) und stellt neben [getID](#) und [getSource](#) die Methode [getWindow](#) zur Verfügung, mit der das Fenster ermittelt werden kann, das die Nachricht ausgelöst hat.

[java.awt.event.WindowEvent](#)

```
public Window getWindow()
```

Die Registrierung der Empfängerklasse erfolgt mit der Methode [addWindowListener](#), die in den Klassen [Dialog](#) und [Frame](#) zur Verfügung steht:

[java.awt.Frame](#)

```
public void addWindowListener(WindowListener l)
```

[Tabelle 19.1](#) gibt eine Übersicht der Methoden von [WindowListener](#) und erklärt ihre Bedeutung:

Ereignismethode	Bedeutung
windowActivated	Das Fenster wurde aktiviert. Diese Methode wird nach dem Erstellen des Fensters aufgerufen und wenn ein Fenster, das im Hintergrund stand, erneut in den Vordergrund gelangt.
windowClosed	Das Fenster wurde geschlossen.
windowClosing	Das Fenster soll geschlossen werden. Diese Methode wird aufgerufen, wenn der Anwender das Fenster über die Titelleiste, das Systemmenü oder die Tastenkombination <code>[ALT]+[F4]</code> schließen will. Es liegt in der Verantwortung der Anwendung, in diesem Fall Code zur Verfügung zu stellen, der das Fenster tatsächlich schließt. Standardmäßig reagiert das Programm nicht auf diese Benutzeraktionen.
windowDeactivated	Das Fenster wurde deaktiviert, also in den Hintergrund gestellt.
windowDeiconified	Das Fenster wurde wiederhergestellt, nachdem es zuvor auf Symbolgröße verkleinert wurde.
windowIconified	Das Fenster wurde auf Symbolgröße verkleinert.
windowOpened	Das Fenster wurde geöffnet.

Tabelle 19.1: Methoden von WindowListener

Das folgende Programm systematisiert die Implementierung der Methode [windowClosing](#), deren Verwendung bereits mehrfach gezeigt wurde. Dazu leitet es aus [Frame](#) eine neue Klasse [CloseableFrame](#) ab und registriert im Konstruktor einen anonymen [WindowAdapter](#), der [windowClosing](#) überlagert und das Fenster durch Aufrufe von `setVisible(false)` und `dispose()` schließt. Alle Ableitungen von [CloseableFrame](#) besitzen nun die Fähigkeit, vom Anwender per Systemmenü, per Schließen-Button oder durch Drücken von `[ALT]+[F4]` beendet zu werden. Die Klasse [Listing1901](#) demonstriert die Anwendung der neuen Klasse und erzeugt ein Fenster `wnd` aus [CloseableFrame](#). Um das Programm nach dem Schließen des Fensters zu beenden, registriert es bei der Fensterklasse zusätzlich einen anonymen [WindowAdapter](#), der bei Aufruf von [windowClosing](#) (also nach dem Schließen des Fensters) das Programm per Aufruf von `System.exit` beendet:

Beispiel

```

001 /* Listing1901.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class CloseableFrame
007 extends Frame
008 {
009     public CloseableFrame()
010     {
011         this("");
012     }
013
014     public CloseableFrame(String title)
015     {
016         super(title);
017         addWindowListener(
018             new WindowAdapter() {
019                 public void windowClosing(WindowEvent event)
020                 {
021                     setVisible(false);
022                     dispose();
023                 }
024             }
025         );
026     }
027 }
028
029 public class Listing1901
030 {
031     public static void main(String[] args)
032     {
033         CloseableFrame wnd = new CloseableFrame("Listing1901");
034         wnd.setBackground(Color.lightGray);
035         wnd.setSize(300,200);
036         wnd.setLocation(200,100);
037         wnd.setVisible(true);
038         wnd.addWindowListener(
039             new WindowAdapter() {
040                 public void windowClosed(WindowEvent event)
041                 {
042                     System.out.println("terminating program...");
043                     System.exit(0);
044                 }
045             }
046         );
047     }
048 }

```

Listing 19.1: Die Klasse CloseableFrame

19.2 Component-Events

- 19.2 Component-Events

Ein Component-Event wird generiert, wenn eine Komponente verschoben oder ihre Größe verändert wurde oder wenn sich der Anzeigezustand einer Komponente verändert hat. Da sowohl Fenster als auch alle Dialogelemente aus der Klasse `Component` abgeleitet sind, haben die hier erwähnten Ereignisse für nahezu alle GUI-Elemente des AWT Gültigkeit.

Ein Empfänger für Component-Events muß das Interface `ComponentListener` implementieren und bekommt Events des Typs `ComponentEvent` übergeben. `ComponentEvent` erweitert die Klasse `AWTEvent` und stellt neben `getID` und `getSource` die Methode `getComponent` zur Verfügung, mit der die Komponente ermittelt werden kann, die die Nachricht ausgelöst hat.

`java.awt.event.ComponentEvent`

```
public Component getComponent()
```

Die Registrierung der Empfängerklasse erfolgt mit der Methode `addComponentListener`, die in allen aus `Component` abgeleiteten Klassen zur Verfügung steht:

`java.awt.Component`

```
public void addComponentListener(ComponentListener l)
```

[Tabelle 19.2](#) gibt eine Übersicht der Methoden von `ComponentListener` und erklärt ihre Bedeutung:

Ereignismethode	Bedeutung
<code>componentShown</code>	Eine Komponente wurde sichtbar.
<code>componentHidden</code>	Eine Komponente wurde unsichtbar.
<code>componentMoved</code>	Eine Komponente wurde verschoben.
<code>componentResized</code>	Die Größe einer Komponente hat sich geändert.

Tabelle 19.2: Methoden von `ComponentListener`

Das folgende Programm demonstriert die Anwendung der Methoden `componentMoved` und `componentResized` am Beispiel eines Fensters `BirdsEyeFrame`, das schematisch sich selbst und seine Lage auf dem Desktop aus der Vogelperspektive darstellt. Wird das Fenster verschoben oder seine Größe geändert, so paßt es seine eigene Darstellung proportional an und zeichnet die Client-Area neu. Die Implementierung der `paint`-methode ermittelt dazu die Seitenverhältnisse von Fenster und Desktop und verwendet diese als Quotient zur Anzeige des Fensters im Fenster.

Um auf die Component-Events zu reagieren, registriert `BirdsEyeFrame` die Adapterklasse `ComponentRepaintAdapter`, die die Methoden `componentMoved` und `componentResized` implementiert. Sie werden immer dann aufgerufen, wenn das Fenster verschoben oder in der Größe geändert wurde und rufen `repaint` auf, um das Fenster neu zu zeichnen. Auf diese Weise werden alle Änderungen des Frames sofort in seiner eigenen Client-Area gespiegelt:

[Listing1902.java](#)

```
001 /* Listing1902.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class WindowClosingAdapter1902
007 extends WindowAdapter
008 {
009     public void windowClosing(WindowEvent event)
010     {
011         event.getWindow().setVisible(false);
012         event.getWindow().dispose();
013         System.exit(0);
014     }
015 }
016
017 class ComponentRepaintAdapter
```

Beispiel

```

018 extends ComponentAdapter
019 {
020     public void componentMoved(ComponentEvent event)
021     {
022         event.getComponent().repaint();
023     }
024
025     public void componentResized(ComponentEvent event)
026     {
027         event.getComponent().repaint();
028     }
029 }
030
031 class BirdsEyeFrame
032 extends Frame
033 {
034     public BirdsEyeFrame()
035     {
036         super("BirdsEyeFrame");
037         addWindowListener(new WindowClosingAdapter1902());
038         addComponentListener(new ComponentRepaintAdapter());
039         setBackground(Color.lightGray);
040     }
041
042     public void paint(Graphics g)
043     {
044         Dimension screensize = getToolkit().getScreenSize();
045         Dimension framesize = getSize();
046         double qx = framesize.width / (double)screensize.width;
047         double qy = framesize.height / (double)screensize.height;
048         g.setColor(Color.white);
049         g.fillRect(
050             (int)(qx * getLocation().x),
051             (int)(qy * getLocation().y),
052             (int)(qx * framesize.width),
053             (int)(qy * framesize.height)
054         );
055         g.setColor(Color.darkGray);
056         g.fillRect(
057             (int)(qx * getLocation().x),
058             (int)(qy * getLocation().y),
059             (int)(qx * framesize.width),
060             (int)(qy * getInsets().top)
061         );
062         g.drawRect(
063             (int)(qx * getLocation().x),
064             (int)(qy * getLocation().y),
065             (int)(qx * framesize.width),
066             (int)(qy * framesize.height)
067         );
068     }
069 }
070 }
071
072 public class Listing1902
073 {
074     public static void main(String[] args)
075     {
076         BirdsEyeFrame wnd = new BirdsEyeFrame();
077         wnd.setSize(300,200);
078         wnd.setLocation(200,100);
079         wnd.setVisible(true);
080     }
081 }

```

Listing 19.2: Das eigene Fenster aus der Vogelperspektive

Die Ausgabe des Programms ist:

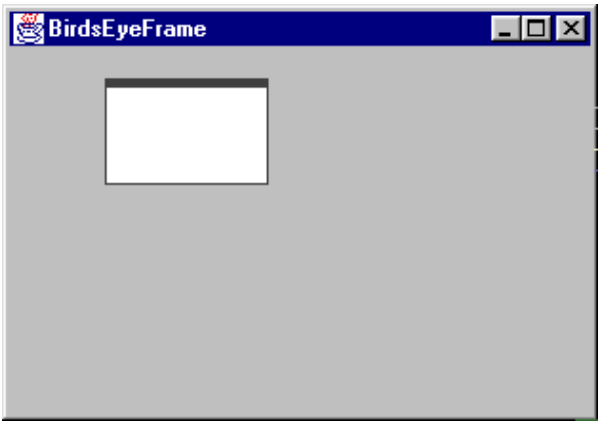


Abbildung 19.1: Das Fenster sieht sich selbst aus der Vogelperspektive

19.3 Mouse-Events

- 19.3 Mouse-Events

Ein Mouse-Event entsteht, wenn der Anwender innerhalb der Client-Area des Fensters eine der Maustasten drückt oder losläßt. Dabei reagiert das Programm sowohl auf Klicks der linken als auch - falls vorhanden - der rechten Maustaste und zeigt an, welche der Umschalttasten [STRG], [ALT], [UMSCHALT] oder [META] während des Mausklicks gedrückt waren. Des weiteren ist es möglich, zwischen einfachen und doppelten Mausklicks zu unterscheiden.

Ein Empfänger für Mouse-Events muß das Interface [MouseListener](#) implementieren und bekommt Events des Typs [MouseEvent](#) übergeben. [MouseEvent](#) erweitert die Klasse [InputEvent](#) und stellt neben [getID](#) und [getSource](#) eine Reihe zusätzlicher Methoden zur Verfügung, die wichtige Informationen liefern. Die Registrierung der Empfängerklasse erfolgt mit der Methode [addMouseListener](#), die in allen Klassen zur Verfügung steht, die aus [Component](#) abgeleitet wurden:

```

public void addMouseListener(MouseListener l)
```

[Tabelle 19.3](#) gibt eine Übersicht der Methoden von [MouseListener](#) und erklärt ihre Bedeutung:

Ereignismethode	Bedeutung
mousePressed	Eine Maustaste wurde gedrückt.
mouseReleased	Die gedrückte Maustaste wurde losgelassen.
mouseClicked	Eine Maustaste wurde gedrückt und wieder losgelassen. Diese Methode wird nach mouseReleased aufgerufen.
mouseEntered	Der Mauszeiger wurde in den Client-Bereich der auslösenden Komponente hineinbewegt.
mouseExited	Der Mauszeiger wurde aus dem Client-Bereich der auslösenden Komponente herausbewegt.

Tabelle 19.3: Methoden von MouseListener

Damit ein Programm auf Mausklicks angemessen reagieren kann, benötigt es zusätzliche Informationen, die es sich mit den Methoden der Klasse [MouseEvent](#) bzw. ihrer Vaterklasse [InputEvent](#) beschaffen kann. Zunächst kann mit [getX](#), [getY](#) und [getPosition](#) die Position des Mauszeigers zum Zeitpunkt des Ereignisses ermittelt werden:

Hinweis

```

public int getX()

public int getY()

public Point getPoint()
```

[getX](#) liefert die x- und [getY](#) die y-Koordinate des Punktes, an dem der Mauszeiger sich beim Auftreten des Ereignisses befindet. [getPoint](#) liefert dieselbe Information zusammengefaßt als [Point](#)-Objekt.

Die Koordinatenwerte werden relativ zum Ursprung der auslösenden Komponente angegeben. Bei einem Fenster des Typs [Frame](#) bedeutet dies, daß bei einem Klick in der oberen linken Ecke nicht (0,0) zurückgegeben wird, sondern ([getInsets\(\).left](#),[getInsets\(\).top](#)). Dies stimmt mit dem Ursprung des Koordinatensystems überein, das auch beim Aufruf der Grafikroutinen zugrundegelegt wird.

Tip

Des weiteren gibt es in [MouseEvent](#) die Methode [isPopupTrigger](#), mit der abgefragt werden kann, ob das Klickereignis den Aufruf eines Popup-Menüs anzeigen soll. Wir werden darauf in [Kapitel 20](#) zurückkommen.

Die Methode [getClickCount](#) liefert die Anzahl der Mausklicks, die dem aktuellen Ereignis zugeordnet ist:

```

public int getClickCount()
```

Normalerweise hat dieser Zähler den Wert 1, bei Mehrfachklicks gibt es aber auch [MouseEvents](#) mit dem Wert 2 oder (je nach Schnelligkeit der Klicks) noch höheren Werten. Die Abfrage von [getClickCount](#) kann dazu verwendet werden, auf Doppelklicks zu reagieren. Hierzu ist lediglich

beim Aufruf von `mousePressed` oder `mouseClicked` zu überprüfen, ob `getClickCount` den Wert 2 liefert. In diesem Fall handelt es sich um einen Doppelklick. Zu beachten ist, daß bei einem Doppelklick zwei aufeinanderfolgende `mousePressed-mouseReleased-mouseClicked`-Sequenzen gesendet werden. In der ersten hat der Zähler den Wert 1, um den ersten (einfachen) Mausklick zu signalisieren, und erst in der zweiten steht er auf 2. Vor einem Doppelklick wird also immer erst der zugehörige Einzelklick übertragen.

Eine alternative Methode, Doppelklicks zu erkennen, besteht darin, die Zeitspanne zwischen zwei Klicks und den Abstand der beiden Koordinatenpaare voneinander zu messen und bei hinreichend kleinen Differenzen einen Doppelklick anstelle zweier Einfachklicks zu melden. Dazu wird mit der Methode `getWhen` der Klasse `InputEvent` der Zeitpunkt (in Millisekunden) und mit `getPoint` oder `getX` und `getY` die Mausposition zweier aufeinanderfolgender Events bestimmt:

Hinweis

`java.awt.event.InputEvent`

```
public long getWhen()
```

Diese Vorgehensweise hat den Vorteil, daß die zeitliche und positionelle Toleranz für Doppelklicks selbst beeinflußt werden kann, was bei dem oben beschriebenen Standardverfahren nicht möglich ist.

Neben den bisher besprochenen Methoden stehen die aus `InputEvent` geerbten Methoden zur Verfügung. `InputEvent` ist die gemeinsame Basisklasse von `MouseEvent` und `KeyEvent` und stellt Methoden zur Verfügung, die generelle Informationen über den Zustand der Umschalttasten `[STRG]`, `[ALT]`, `[UMSCHALT]` oder `[META]` zum Zeitpunkt des Ereignisses liefern:

`java.awt.event.InputEvent`

```
public boolean isShiftDown()
```

```
public boolean isControlDown()
```

```
public boolean isMetaDown()
```

```
public boolean isAltDown()
```

Während die Tasten `[UMSCHALT]`, `[STRG]` und `[ALT]` erwartungsgemäß die zugehörigen Tasten erkennen, zeigt `isMetaDown` unter Windows 95 an, ob der Klick von der rechten oder der linken Maustaste stammt. Ist der Rückgabewert `true`, so wurde die rechte Maustaste gedrückt, andernfalls die linke.

Hinweis

Das folgende Programm ist ein einfaches Beispiel für die Reaktion auf Mausklicks. Es zeichnet an den Stellen, wo die Maustaste gedrückt wird, einen kleinen Smiley auf den Bildschirm:

Beispiel

`Listing1903.java`

```
001 /* Listing1903.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing1903
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing1903 wnd = new Listing1903();
012         wnd.setSize(300,200);
013         wnd.setLocation(200,100);
014         wnd.setVisible(true);
015     }
016
017     public Listing1903()
018     {
019         super("Listing1903");
020         addWindowListener(
021             new WindowAdapter() {
022                 public void windowClosing(WindowEvent event)
023                 {
024                     setVisible(false);
025                     dispose();
026                     System.exit(0);
027                 }
028             }
029         )
030     }
031 }
```

```

029     );
030     addMouseListener(new MyMouseListener());
031 }
032
033 class MyMouseListener
034 extends MouseAdapter
035 {
036     int cnt = 0;
037
038     public void mousePressed(MouseEvent event)
039     {
040         Graphics g = getGraphics();
041         int x = event.getX();
042         int y = event.getY();
043         if (event.getClickCount() == 1) { //Gesicht
044             ++cnt;
045             //Kopf und Augen
046             g.drawOval(x-10,y-10,20,20);
047             g.fillRect(x-6,y-5,4,5);
048             g.fillRect(x+3,y-5,4,5);
049             //Mund
050             if (event.isMetaDown()) { //grimmig
051                 g.drawLine(x-5,y+7,x+5,y+7);
052             } else { //lächeln
053                 g.drawArc(x-7,y-7,14,14,225,100);
054             }
055             //Zähler
056             g.drawString(""+cnt,x+10,y-10);
057         } else if (event.getClickCount() == 2) { //Brille
058             g.drawLine(x-9,y-3,x+9,y-3);
059         }
060     }
061 }
062 }

```

Listing 19.3: Reaktion auf Mausklicks

Beim Drücken der linken Maustaste lächelt das Gesicht, bei der rechten sieht es grimmig aus. Bei einem Doppelklick bekommt der Smiley zusätzlich eine Brille aufgesetzt. Die Ausgabe des Programms sieht beispielsweise so aus:

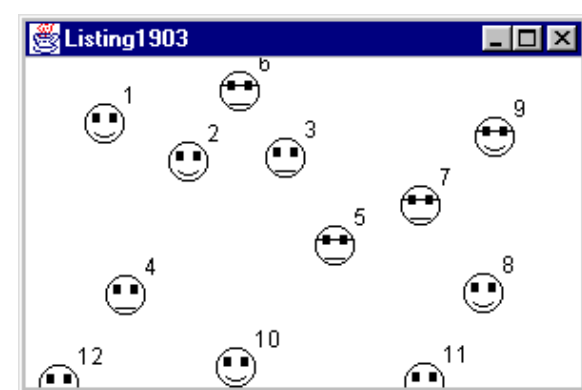


Abbildung 19.2: Die Ausgabe des Mausklick-Programms

Erwähnenswert ist die Tatsache, daß das Programm ohne [paint](#)-Methode auskommt. Tatsächlich beschafft die Methode [mousePressed](#) sich die zum Zeichnen erforderliche Instanz der Klasse [Graphics](#) durch Aufruf von [getGraphics](#) selbst. Die Methode [getGraphics](#) gibt immer den Grafikkontext des aktuellen Fensterobjekts zurück. Das gelieferte [Graphics](#)-Objekt kann auf dieselbe Weise verwendet werden wie das als Parameter an [paint](#) übergebene.

Hinweis

Leider hat das Beispielprogramm eine gravierende konzeptionelle Schwäche, denn die gezeichneten Smileys sind nicht *dauerhaft*. Wird das Fenster in den Hintergrund gestellt, nachdem einige Mausklicks durchgeführt wurden, sind bei erneuter Anzeige des Fensters die zuvor gezeichneten Smileys wieder verschwunden. Der Grund dafür ist, daß wir in unserem Beispiel nicht die [paint](#)-Methode überlagert haben, sondern die Ausgaben als direkte Reaktion auf ein Mausereignis erledigten. Da Windows nicht dafür sorgt, den Inhalt eines zuvor verdeckten Fensters zu rekonstruieren, ist

die Anwendung selbst dafür verantwortlich, dies in ihrer [paint](#)-Methode zu tun. Eine bessere Implementierung dieses Programms würde also darin bestehen, die Koordinaten und Eigenschaften aller gezeichneten Objekte zu speichern und sie beim Aufruf von [paint](#) später selbst zu rekonstruieren. Wir werden diese Vorgehensweise in einem späteren Beispiel genauer kennenlernen.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	≤	≤	≥	≥
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	-------------------	-------------------	-------------------	-------------------

19.4 MouseMotion-Events

- 19.4 MouseMotion-Events

Im JDK 1.1 gibt es eine klare Trennung zwischen *Mouse*-Events und *MouseMotion*-Events. Während die Mouse-Events für Mausklicks und das Betreten oder Verlassen der Komponente zuständig sind, geben MouseMotion-Events Auskunft über die *Bewegung* des Mauszeigers. Neben der verbesserten Modularisierung (sehr viele Programme wollen lediglich von Mausklicks, nicht aber von Mausbewegungen unterrichtet werden) wurde die Trennung vor allem deshalb vorgenommen, um Performance-Verbesserungen zu erzielen. Durch die Entkopplung der Ereignishandler für Mausbewegungen wird die Anzahl der Events, mit denen die meisten Event-Handler beschossen werden, im Vergleich zum JDK 1.0 drastisch reduziert.

Ein Empfänger für MouseMotion-Events muß das Interface [MouseMotionListener](#) implementieren. Er wird mit der Methode [addMouseMotionListener](#) registriert, die in allen Objekten der Klasse [Component](#) oder daraus abgeleiteten Klassen zur Verfügung steht.

```

java.awt.Component
public void addMouseMotionListener(MouseListener l)

```

Im Gegensatz zur bisherigen Systematik bekommen die Methoden von [MouseMotionListener](#) allerdings keine Events des Typs [MouseMotionEvent](#) übergeben (die gibt es nämlich nicht), sondern solche des Typs [MouseEvent](#). Damit stehen dieselben Methoden wie bei Mouse-Events zur Verfügung.

Warnung

Das Interface [MouseMotionListener](#) definiert die Methoden [mouseMoved](#) und [mouseDragged](#):

```

java.awt.event.MouseMotionListener
public abstract void mouseMoved(MouseEvent e)

public abstract void mouseDragged(MouseEvent e)

```

[mouseMoved](#) wird aufgerufen, wenn die Maus bewegt wird, ohne daß dabei eine der Maustasten gedrückt ist. [mouseDragged](#) wird dagegen aufgerufen, wenn die Maus bei gedrückter linker oder rechter Maustaste bewegt wird.

Das folgende Listing zeigt den Einsatz von [mouseDragged](#) am Beispiel eines Programms, mit dem Rechtecke gezeichnet werden können. Das Drücken der linken Maustaste legt den Anfangspunkt des Rechtecks fest, und durch Ziehen der Maus wird seine Größe bestimmt. Nach dem Loslassen der Maustaste wird das Rechteck in die Liste der gezeichneten Objekte eingetragen und beim nächsten Aufruf von [paint](#) gezeichnet.

Beispiel

[Listing1904.java](#)

```

001 /* Listing1904.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005 import java.util.*;
006
007 public class Listing1904
008 extends Frame
009 {
010     private Vector drawlist;
011     private Rectangle actrect;
012
013     public static void main(String[] args)
014     {
015         Listing1904 wnd = new Listing1904();
016         wnd.setLocation(200,200);
017         wnd.setSize(400,300);
018         wnd.setVisible(true);
019     }
020
021     public Listing1904()
022     {
023         super("Listing1904");
024         drawlist = new Vector();
025         actrect = new Rectangle(0,0,0,0);

```

```

026     addWindowListener(new MyWindowListener());
027     addMouseListener(new MyMouseListener());
028     addMouseMotionListener(new MyMouseMotionListener());
029 }
030
031 public void paint(Graphics g)
032 {
033     Rectangle r;
034     Enumeration e;
035
036     for (e = drawlist.elements(); e.hasMoreElements(); ) {
037         r = (Rectangle)e.nextElement();
038         g.drawRect(r.x, r.y, r.width, r.height);
039     }
040     if (actrect.x > 0 || actrect.y > 0) {
041         g.drawRect(
042             actrect.x,
043             actrect.y,
044             actrect.width,
045             actrect.height
046         );
047     }
048 }
049
050 class MyMouseListener
051 extends MouseAdapter
052 {
053     public void mousePressed(MouseEvent event)
054     {
055         actrect = new Rectangle(event.getX(),event.getY(),0,0);
056     }
057
058     public void mouseReleased(MouseEvent event)
059     {
060         if (actrect.width > 0 || actrect.height > 0) {
061             drawlist.addElement(actrect);
062         }
063         repaint();
064     }
065 }
066
067 class MyMouseMotionListener
068 extends MouseMotionAdapter
069 {
070     public void mouseDragged(MouseEvent event)
071     {
072         int x = event.getX();
073         int y = event.getY();
074         if (x > actrect.x && y > actrect.y) {
075             actrect.width = x - actrect.x;
076             actrect.height = y - actrect.y;
077         }
078         repaint();
079     }
080 }
081
082 class MyWindowListener
083 extends WindowAdapter
084 {
085     public void windowClosing(WindowEvent event)
086     {
087         setVisible(false);
088         dispose();
089         System.exit(0);
090     }

```

```
091     }  
092 }
```

Listing 19.4: Zeichnen von Rechtecken durch Ziehen der Maus

Eine Beispielsitzung mit dem Programm könnte folgendes Ergebnis liefern:

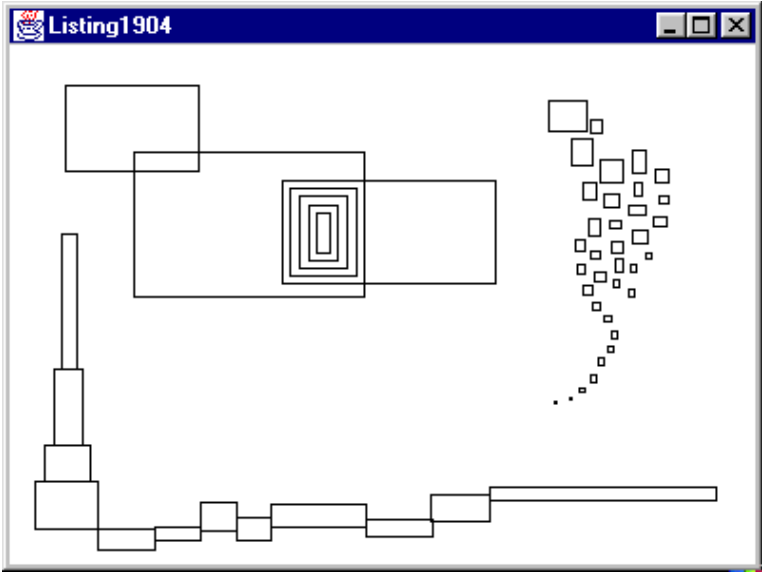


Abbildung 19.3: Die Ausgabe des Mausbewegungsprogramms

Interessant ist hier das Zusammenspiel zwischen `paint` und den Methoden, die die Maus-Events behandeln. Beim Aufruf von `paint` werden zunächst alle Rechtecke gezeichnet, die sich in der Liste `drawlist` befinden. Anschließend überprüft `paint`, ob das aktuelle Element (dieses wurde beim Mausklick angelegt) eine Länge oder Breite größer Null hat und zeichnet es gegebenenfalls. Dies ist genau dann der Fall, wenn der Anwender die Maustaste gedrückt und die Maus vom Ursprung nach rechts unten bewegt hat. Beim Loslassen der Maustaste wird das aktuelle Element in die Liste der Rechtecke eingetragen und steht so beim nächsten `paint` zur Verfügung.

Hinweis

19.5 Focus-Events

- 19.5 Focus-Events

Der *Fokus* zeigt an, welches Fenster die Tastatureingaben erhält. Sind mehrere Fenster gleichzeitig geöffnet, so kann immer nur eines von ihnen den Fokus beanspruchen. Sind auf einem aktiven Fenster mehrere Dialogelemente aktiv, so kann ebenfalls nur eines davon den Fokus erhalten, denn jedes Dialogelement wird ebenfalls durch ein (meist unsichtbares) Fenster dargestellt.

Ein Empfänger für Focus-Events muß das Interface [FocusListener](#) implementieren und bekommt Events des Typs [FocusEvent](#) übergeben. [FocusEvent](#) erweitert die Klasse [ComponentEvent](#) und stellt neben [getID](#) und [getSource](#) die Methode [isTemporary](#) zur Verfügung, die anzeigt, ob der Fokuswechsel temporär oder permanent ist:

```

                                java.awt.event.FocusEvent

public boolean isTemporary()
```

Die Registrierung von Focus-Events erfolgt mit der Methode [addFocusListener](#), die auf allen Objekten des Typs [Component](#) oder daraus abgeleiteten Objekten zur Verfügung steht:

```

                                java.awt.Component

public void addFocusListener(FocusListener l)
```

Das Interface [FocusListener](#) enthält lediglich zwei unterschiedliche Methoden:

```

                                java.awt.event.FocusListener

public abstract void focusGained(FocusEvent e)

public abstract void focusLost(FocusEvent e)
```

[focusGained](#) wird aufgerufen, wenn die Komponente den Fokus erhält, und [focusLost](#), wenn sie ihn wieder abgibt. Eine Komponente erhält den Fokus beispielsweise, wenn sie mit der Maus angeklickt oder (unter Windows) mit Hilfe der Tasten [\[TAB\]](#) oder [\[UMSCHALT\]+\[TAB\]](#) oder über einen Beschleuniger angesprungen wurde. Außerdem gibt es die Möglichkeit, den Fokus programmgesteuert zu verändern. Dazu gibt es die Methode [requestFocus](#) der Klasse [Component](#), mit der eine Komponente den Fokus für sich selbst beanspruchen bzw. ihn einer anderen Komponente zuweisen kann:

```

                                java.awt.Component

public void requestFocus()
```

Der explizite Umgang mit Focus-Events ist etwa bei Dialogelementen sinnvoll. Dadurch kann ein Edit-Control beispielsweise feststellen, daß das nächste Dialogelement angesprungen werden soll und ein eventuell geänderter Text zuvor gespeichert werden muß. Focus-Events sind oft schwierig zu debuggen, denn durch den Wechsel zwischen Debug-Window und Anwendung werden meist zusätzliche Fokuswechsel ausgelöst, die den Fehler verschleiern oder die Fehlerstelle unerreichbar machen. Wird innerhalb von Fokus-Handlern die Methode [requestFocus](#) aufgerufen, kann es zudem leicht zu Endlosschleifen kommen.

Hinweis

Das folgende Programm zeigt die Anwendung eines [FocusListener](#)-Interfaces zur Implementierung der Methoden [focusGained](#) und [focusLost](#) auf einem [Frame](#)-Objekt. Wenn das Fenster den Eingabefokus erhält, wird der Hintergrund gelb und gibt die Meldung »Fokus erhalten« aus. Verliert das Fenster den Fokus, so wird der Hintergrund grau und die Meldung »Kein Fokus« wird angezeigt. Die Registrierung des Fokusempfänger-Objekts erfolgt durch Aufruf von [addFocusListener\(this\)](#), bei dem das Fensterobjekt sich selbst als Empfänger registriert:

Beispiel

[Listing1905.java](#)

```

001 /* Listing1905.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class WindowClosingAdapter1905
007 extends WindowAdapter
008 {
009     public void windowClosing(WindowEvent event)
010     {
011         event.getWindow().setVisible(false);
012         event.getWindow().dispose();
013     }
014 }
```



```

013         System.exit(0);
014     }
015 }
016
017 public class Listing1905
018 extends Frame
019 implements FocusListener
020 {
021     boolean havefocus = false;
022
023     public static void main(String[] args)
024     {
025         Listing1905 wnd = new Listing1905();
026     }
027
028     public Listing1905()
029     {
030         super("Listing1905");
031         addFocusListener(this);
032         addWindowListener(new WindowClosingAdapter1905());
033         setBackground(Color.lightGray);
034         setSize(300,200);
035         setLocation(200,100);
036         setVisible(true);
037     }
038
039     public void paint(Graphics g)
040     {
041         if (havefocus) {
042             g.setColor(Color.black);
043             g.drawString("Fokus erhalten",10,50);
044         } else {
045             g.setColor(Color.darkGray);
046             g.drawString("Kein Fokus",10,50);
047         }
048     }
049
050     public void focusGained(FocusEvent event)
051     {
052         havefocus = true;
053         setBackground(Color.yellow);
054         repaint();
055     }
056
057     public void focusLost(FocusEvent event)
058     {
059         havefocus = false;
060         setBackground(Color.lightGray);
061         repaint();
062     }
063 }

```

Listing 19.5: Behandlung von Fokus-Ereignissen

Das Programm verwendet zwei unterschiedliche Techniken, um Ereignisempfänger zu registrieren. Für die Focus-Events implementiert es das [FocusListener](#)-Interface und registriert sich bei sich selbst. Dies ist eine vernünftige Vorgehensweise, da sämtliche Methoden des Interfaces benötigt werden. Das [windowClosing](#)-Event wird dagegen von der externen Klasse [WindowClosingAdapter1905](#) zur Verfügung gestellt, die sich aus [WindowAdapter](#) ableitet und die Methode [windowClosing](#) mit der Funktionalität zum Schließen des Fensters und zum Beenden des Programms belegt. Diese Technik demonstriert beispielhaft die Entwicklung spezialisierter Adapterklassen, die mehrfach verwendet werden können.

Hinweis

Die Ausgabe des Programms (nachdem es den Fokus erhalten hat) ist:

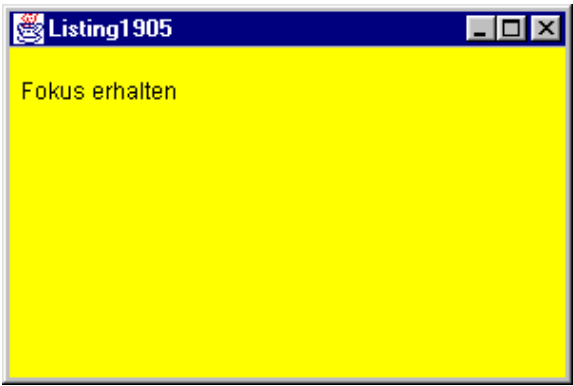


Abbildung 19.4: Programm nach Erhalt des Eingabefokus

19.6 Key-Events

- 19.6 Key-Events

Unter Windows werden alle Tastatureingaben an die fokussierte Komponente gesendet. Ein Empfänger für Key-Events muß das Interface [KeyListener](#) implementieren und bekommt Events des Typs [KeyEvent](#) übergeben. [KeyEvent](#) erweitert die Klasse [InputEvent](#), die ihrerseits aus [ComponentEvent](#) abgeleitet ist, und stellt neben [getID](#) und [getSource](#) eine ganze Reihe von Methoden zur Verfügung, mit denen die Erkennung und Bearbeitung der Tastencodes vereinfacht wird.

Die Registrierung von Key-Events erfolgt mit der Methode [addKeyListener](#), die auf allen Objekten des Typs [Component](#) oder daraus abgeleiteten Klassen zur Verfügung steht:

```

public void addKeyListener(KeyListener l)
```

java.awt.Component

Das Interface [KeyListener](#) definiert drei unterschiedliche Methoden:

```

public abstract void keyPressed(KeyEvent e)

public abstract void keyReleased(KeyEvent e)

public abstract void keyTyped(KeyEvent e)
```

java.awt.event.KeyListener

Um die Funktionsweise dieser Methoden im Zusammenspiel mit den Methoden der Klasse [KeyEvent](#) besser verstehen zu können, wollen wir zwischen *Zeichentasten* und *Funktionstasten* unterscheiden. Zeichentasten sind dabei solche Tasten, mit denen Buchstaben, Ziffern oder sonstige gültige Unicode-Zeichen eingegeben werden, wie z.B. `[a]`, `[A]`, `[B]`, `[1]`, `[2]`, `[%]`, `[+]`, aber auch `[ESC]`, `[LEER]` oder `[TAB]`. Zu den Funktionstasten gehören beispielsweise `[F1]`, `[F2]`, `[POS1]` oder `[CURSORLINKS]`, aber auch die Umschalttasten `[STRG]`, `[ALT]` und `[UMSCHALT]`.

Hinweis

Die Methode [keyTyped](#) wird immer dann aufgerufen, wenn eine Zeichentaste gedrückt wurde. Beim Drücken einer Funktionstaste wird sie dagegen nicht aufgerufen. Im Gegensatz dazu wird [keyPressed](#) bei jedem Tastendruck aufgerufen, unabhängig davon, ob es sich um eine Zeichentaste oder eine Funktionstaste handelt. Mit [keyPressed](#) können sogar in beschränktem Umfang die Feststelltasten wie `[NUMLOCK]` oder `[CAPSLOCK]` erkannt werden. Beide Methoden erhalten auch Tastatur-Repeats, werden also bei längerem Festhalten einer Taste wiederholt aufgerufen. Die Methode [keyReleased](#) wird aufgerufen, wenn eine gedrückte Taste losgelassen wurde, unabhängig davon, ob es sich um eine Zeichen- oder Funktionstaste handelt.

Um beim Auftreten eines Tastatur-Events zu erkennen, welche Taste gedrückt wurde, stellt die Klasse [KeyEvent](#) die Methoden [getKeyCode](#) und [getKeyChar](#) und zusätzlich die aus [InputEvent](#) geerbten Methoden [isShiftDown](#), [isControlDown](#), [isMetaDown](#) und [isAltDown](#) zur Verfügung:

```

public int getKeyCode()

public char getKeyChar()

public boolean isShiftDown()

public boolean isControlDown()

public boolean isMetaDown()

public boolean isAltDown()
```

java.awt.event.KeyEvent

[getKeyChar](#) liefert das Zeichen, das der gedrückten Zeichentaste entspricht, also ein 'a', wenn die Taste `[A]` gedrückt wurde, und ein 'A', wenn die Tastenkombination `[UMSCHALT]+[A]` gedrückt wurde. [getKeyCode](#) liefert dagegen *virtuelle Tastencodes*, die in [KeyEvent](#) als symbolische Konstanten definiert wurden. Hier wird beim Drücken der Taste `[A]` immer der Code `VK_A` geliefert, unabhängig davon, ob `[UMSCHALT]` gedrückt wurde oder nicht. [Tabelle 19.4](#) gibt eine Übersicht der wichtigsten virtuellen Keycodes der Klasse [KeyEvent](#).

Symbolischer Name	Bedeutung
<code>VK_0</code> ... <code>VK_9</code>	<code>[0]</code> ... <code>[9]</code>

VK_A ... VK_Z	[A] ... [Z]
VK_ENTER	[ENTER]
VK_SPACE	[LEER]
VK_TAB	[TAB]
VK_ESCAPE	[ESC]
VK_BACK_SPACE	[RÜCK]
VK_F1 ... VK_F12	Die Funktionstasten [F1] ... [F12]
VK_HOME, VK_END	[HOME], [END]
VK_INSERT, VK_DELETE	[EINFG], [ENTF]
VK_PAGE_UP, VK_PAGE_DOWN	[BILDHOCH], [BILDRUNTER]
VK_DOWN, VK_UP	[CURSORHOCH], [CURSORRUNTER]
VK_LEFT, VK_RIGHT	[CURSORLINKS], [CURSORRECHTS]

Tabelle 19.4: Virtuelle Key-Codes

Am einfachsten ist es, innerhalb von `keyTyped` mit `getKeyChar` die Zeichentasten abzufragen. Dabei liefert `getKeyChar` stets den ASCII-Code der gedrückten Zeichentaste, Funktionstasten werden nicht übertragen. Der Rückgabewert von `getKeyCode` ist in diesem Fall immer `KeyEvent.VK_UNDEFINED`. Sollen dagegen auch Funktionstasten abgefragt werden, muß die Methode `keyPressed` überlagert werden. Hier ist etwas Vorsicht geboten, denn es wird auf *alle* Tastendrücke reagiert, und sowohl `getKeyCode` als auch `getKeyChar` liefern Werte zurück. Die Unterscheidung von Zeichen- und Funktionstasten kann in diesem Fall mit Hilfe von `getKeyChar` vorgenommen werden, deren Rückgabewert die Konstante `KeyEvent.CHAR_UNDEFINED` ist, wenn eine Funktionstaste gedrückt wurde.

Die `is`-Methoden sind bereits bekannt, mit ihnen können die Umschalttasten abgefragt werden. Das ist beispielsweise sinnvoll, um bei einer Funktionstaste herauszufinden, ob sie mit gedrückter Umschalttaste ausgelöst wurde oder nicht. Allerdings sind die Umschalttasten im Event-Modell des JDK 1.1 keine *Tottasten*, sondern liefern selbst ein Key-Event und lösen die Methode `keyPressed` aus.

Insgesamt ist das Handling von Tastatur-Events nicht ganz trivial und erfordert ein wenig Aufwand bei der Unterscheidung von Zeichen-, Funktions-, Umschalt- oder Feststelltasten. [Tabelle 19.5](#) faßt die bisherigen Ausführungen noch einmal zusammen. Die erste Zeile zeigt das Verhalten beim Aufruf der Listener-Methode `keyTyped` an, die zweite beim Aufruf von `keyPressed`. Die *erste* Spalte liefert dazu den Rückgabewert von `getKeyCode`, die zweite den von `getKeyChar`. Jedes Element beschreibt in der oberen Hälfte den Rückgabewert beim Drücken einer Zeichentaste und in der unteren den beim Drücken einer Funktionstaste.

	getKeyCode	getKeyChar
keyTyped	Zeichentaste: <code>VK_UNDEFINED</code> Funktionstaste: --	Zeichentaste: Taste als <code>char</code> Funktionstaste: --
keyPressed	Zeichentaste: <code>VK_...</code> Funktionstaste: <code>VK_...</code>	Zeichentaste: Taste als <code>char</code> Funktionstaste: <code>CHAR_UNDEFINED</code>

Tabelle 19.5: Rückgabecodes bei Tastaturereignissen

Das folgende Beispiel demonstriert die Abfrage der Tastaturereignisse. Es implementiert `keyPressed`, um die Funktionstasten [F1] bis [F3] und den Status der Umschalttasten abzufragen. Jeder Tastendruck wird in einen String übersetzt, in `msg1` gespeichert und durch Aufruf von `repaint` auf dem Bildschirm angezeigt. Nach dem Loslassen der Taste wird die Anzeige wieder vom Bildschirm entfernt. Weiterhin wurde `keyTyped` überlagert, um die Zeichentasten abzufragen. Jeder Tastendruck wird in `msg2` gespeichert und ebenfalls auf dem Bildschirm angezeigt. Im Gegensatz zu den Funktionstasten bleibt die Ausgabe auch erhalten, wenn die Taste losgelassen wird. Bei jedem weiteren Tastendruck wird sie um ein Zeichen ergänzt. Zusätzlich werden die einzelnen Ereignisse auf der Konsole dokumentiert.

Beispiel

```
001 /* Listing1906.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class WindowClosingAdapter
007 extends WindowAdapter
008 {
009     public void windowClosing(WindowEvent event)
010     {
011         event.getWindow().setVisible(false);
012         event.getWindow().dispose();
013         System.exit(0);
014     }
015 }
016
017 public class Listing1906
018 extends Frame
019 implements KeyListener
020 {
021     String msg1 = "";
022     String msg2 = "";
023
024     public static void main(String[] args)
025     {
026         Listing1906 wnd = new Listing1906();
027     }
028
029     public Listing1906()
030     {
031         super("Listing1906");
032         addKeyListener(this);
033         addWindowListener(new WindowClosingAdapter());
034         setBackground(Color.lightGray);
035         setSize(300,200);
036         setLocation(200,100);
037         setVisible(true);
038     }
039
040     public void paint(Graphics g)
041     {
042         if (msg1.length() > 0) {
043             draw3DRect(g,20,50,250,30);
044             g.setColor(Color.black);
045             g.drawString(msg1,30,70);
046         }
047         if (msg2.length() > 0) {
048             draw3DRect(g,20,100,250,30);
049             g.setColor(Color.black);
050             g.drawString(msg2,30,120);
051         }
052     }
053
054     void draw3DRect(Graphics g,int x,int y,int width,int height)
055     {
056         g.setColor(Color.darkGray);
057         g.drawLine(x,y,x,y+height);
058         g.drawLine(x,y,x+width,y);
059         g.setColor(Color.white);
060         g.drawLine(x+width,y+height,x,y+height);
061         g.drawLine(x+width,y+height,x+width,y);
062     }
063
064     public void keyPressed(KeyEvent event)
```

```

065 {
066     msg1 = "";
067     System.out.println(
068         "key pressed: " +
069         "key char = " + event.getKeyChar() + "    " +
070         "key code = " + event.getKeyCode()
071     );
072     if (event.getKeyChar() == KeyEvent.CHAR_UNDEFINED) {
073         int key = event.getKeyCode();
074         //Funktionstaste abfragen
075         if (key == KeyEvent.VK_F1) {
076             msg1 = "F1";
077         } else if (key == KeyEvent.VK_F2) {
078             msg1 = "F2";
079         } else if (key == KeyEvent.VK_F3) {
080             msg1 = "F3";
081         }
082         //Modifier abfragen
083         if (msg1.length() > 0) {
084             if (event.isAltDown()) {
085                 msg1 = "ALT + " + msg1;
086             }
087             if (event.isControlDown()) {
088                 msg1 = "STRG + " + msg1;
089             }
090             if (event.isShiftDown()) {
091                 msg1 = "UMSCHALT + " + msg1;
092             }
093         }
094     }
095     repaint();
096 }
097
098 public void keyReleased(KeyEvent event)
099 {
100     System.out.println("key released");
101     msg1 = "";
102     repaint();
103 }
104
105 public void keyTyped(KeyEvent event)
106 {
107     char key = event.getKeyChar();
108     System.out.println("key typed: " + key);
109     if (key == KeyEvent.VK_BACK_SPACE) {
110         if (msg2.length() > 0) {
111             msg2 = msg2.substring(0,msg2.length() - 1);
112         }
113     } else if (key >= KeyEvent.VK_SPACE) {
114         if (msg2.length() < 40) {
115             msg2 += event.getKeyChar();
116         }
117     }
118     repaint();
119 }
120 }

```

Listing 19.6: Reaktion auf Tastaturereignisse

Eine beispielhafte Ausgabe des Programms ist:

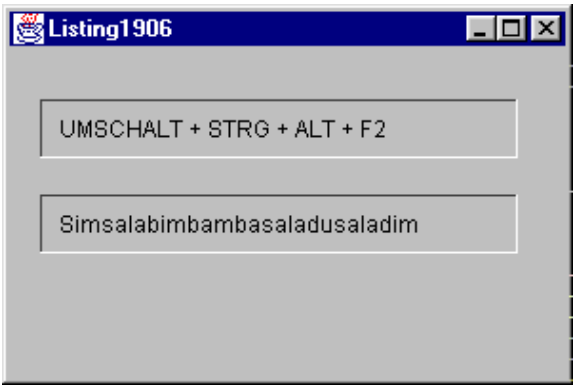


Abbildung 19.5: Darstellung von Tastaturereignissen

19.7 Zusammenfassung

- [19.7 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Das Interface [WindowListener](#), Events des Typs [WindowEvent](#) und die Methode [addWindowListener](#)
- Das Interface [ComponentListener](#), Events des Typs [ComponentEvent](#) und die Methode [addComponentListener](#)
- Das Interface [MouseListener](#), Events des Typs [MouseEvent](#) und die Methode [addMouseListener](#)
- Das Interface [MouseMotionListener](#) und die Methode [addMouseMotionListener](#)
- Das Interface [FocusListener](#), Events des Typs [FocusEvent](#) und die Methode [addFocusListener](#)
- Das Interface [KeyListener](#), Events des Typs [KeyEvent](#) und die Methode [addKeyListener](#)
- Die virtuellen Tastencodes und die zugehörigen symbolischen Konstanten

Kapitel 20

Menüs

- [20 Menüs](#)
 - [20.1 Grundlagen](#)
 - [20.2 Menüleiste](#)
 - [20.3 Menüs](#)
 - [20.4 Menüeinträge](#)
 - [20.4.1 Einfache Menüeinträge](#)
 - [20.4.2 CheckboxMenuItem](#)
 - [20.4.3 Beschleunigertasten](#)
 - [20.4.4 Untermenüs](#)
 - [20.5 Action-Events](#)
 - [20.6 Kontextmenüs](#)
 - [20.7 Datenaustausch mit der Zwischenablage](#)
 - [20.7.1 Überblick](#)
 - [20.7.2 Kommunikation mit der Zwischenablage](#)
 - [20.8 Zusammenfassung](#)

20.1 Grundlagen

- 20.1 Grundlagen

Eine der Möglichkeiten eines Programms, mit dem Anwender zu interagieren, besteht darin, *Menüs* zur Verfügung zu stellen. Aufgabe des Programms ist es dabei, den Aufbau und die visuellen Eigenschaften der Menüs festzulegen. Der Anwender wählt Menüpunkte aus und löst dadurch eine Nachricht aus, auf die das Programm entsprechend reagiert.

Während die Definition der Menüstruktur in den meisten auf Windows basierten Entwicklungssystemen in eine *Ressourcendatei* ausgelagert wird, erfolgt sie in Java innerhalb des Programms. Diese Vorgehensweise hat nicht nur den Vorteil, portabel zu sein, sondern bietet auch die Möglichkeit, bei der Gestaltung der Menüs die Vorteile der objektorientierten Programmierung zu nutzen. So ist es in Java ohne weiteres möglich, Menüs zu definieren, die allgemeine Eigenschaften an spezialisierte Unterklassen vererben. In größeren Programmen kann dies eine große Hilfe bei der Pflege der Menüstruktur und der Erhaltung der Konsistenz der Menüeinträge sein.

In Java wird die Konstruktion von Menüs durch eine Reihe speziell dafür vorgesehener Klassen unterstützt. Die Klasse [MenuBar](#) stellt die *Menüzeile* eines Fensters dar, die Klasse [Menu](#) ein einzelnes der darin enthaltenen *Menüs*, und die Klassen [MenuItem](#) und [CheckboxMenuItem](#) bilden die vom Anwender auswählbaren Einträge innerhalb der Menüs.

Wird ein Menüeintrag ausgewählt, löst dies im Programm eine entsprechende Nachricht aus. Das Programm wird üblicherweise auf diese Nachricht reagieren und die vom Anwender ausgewählte Funktion ausführen. Welche Nachrichten beim Auswählen eines Menüpunktes ausgelöst werden und wie das Programm darauf reagieren kann, werden wir uns im nächsten Abschnitt ansehen. Zunächst wollen wir uns auf das *Erstellen* der Menüs konzentrieren.

Hinweis

20.2 Menüleiste

- [20.2 Menüleiste](#)

Eine *Menüleiste* stellt das Hauptmenü eines Fensters dar. Sie befindet sich unterhalb der Titelleiste am oberen Rand des Fensters und zeigt die Namen der darin enthaltenen Menüs an. Eine Menüleiste wird durch Instanzieren der Klasse [MenuBar](#) erzeugt:

```
public MenuBar()  
    java.awt.MenuBar
```

Der parameterlose Konstruktor erzeugt eine leere Menüleiste, in die dann durch Aufruf der Methode [add](#) Menüs eingefügt werden können:

```
public void add(Menu m)  
    java.awt.MenuBar
```

Zum Entfernen eines bestehenden Menüs kann die Methode [remove](#) verwendet werden. Zur Auswahl des zu entfernenden Menüs kann dabei entweder der Index des zu entfernenden Menüs oder das Menü selbst als Parameter übergeben werden:

```
public void remove(int index)  
  
public void remove(MenuComponent m)  
    java.awt.MenuBar
```

Mit [getMenu](#) stellt die Klasse [MenuBar](#) eine Methode zum Zugriff auf ein beliebiges Menü der Menüleiste zur Verfügung. [getMenu](#) liefert dabei das Menüobjekt, das sich an der Position mit dem angegebenen Index befindet:

```
public Menu getMenu(int index)  
    java.awt.MenuBar
```

Um eine Menüleiste an ein Fenster zu binden, besitzt die Klasse [Frame](#) eine Methode [setMenuBar](#):

```
public void setMenuBar(MenuBar mb)  
    java.awt.Frame
```

Durch Aufruf dieser Methode wird die angegebene Menüleiste im Fenster angezeigt, und beim Auswählen eines Menüpunkts werden Nachrichten ausgelöst und an das Fenster gesendet. Die Fensterklasse kann diese Nachrichten durch das Registrieren eines Objekts vom Typ [ActionListener](#) bearbeiten. Wir gehen darauf weiter unten genauer ein.

20.3 Menüs

- [20.3 Menüs](#)

Die *Menüs* bilden die Bestandteile einer Menüleiste. Sie werden in Java durch Instanzen der Klasse [Menu](#) repräsentiert. Im einfachsten Fall erwartet der Konstruktor von [Menu](#) einen [String](#)-Parameter, der den Namen des Menüs angibt:

```
public Menu(String label)
```

java.awt.Menu

Dieser Name wird verwendet, um das Menü in der Menüleiste zu verankern. Ähnlich wie bei der Menüleiste stehen auch bei einem Menü die Methoden [add](#) und [remove](#) zur Verfügung. Im Gegensatz zu [MenuBar](#) bearbeiten sie allerdings keine Menüs, sondern Menüeinträge:

```
public void add(MenuItem mi)

public void add(String label)

public void remove(int index)

public void remove(MenuComponent item)
```

java.awt.Menu

Die Methode [remove](#) kann entweder mit einem numerischen Index oder mit einem [MenuItem](#) als Parameter aufgerufen werden, um den zu entfernenden Menüeintrag zu identifizieren. Wird ein numerischer Index verwendet, so beginnt die Zählung des ersten Elements bei 0.

[add](#) steht ebenfalls in zwei Varianten zur Verfügung. Bei der ersten muß ein Objekt der Klasse [MenuItem](#) übergeben werden. Die zweite erwartet lediglich einen [String](#), der den Menünamen bezeichnet. Sie generiert automatisch eine entsprechende Instanz der Klasse [MenuItem](#).

Neben Menüeinträgen, die ein Ereignis auslösen, können mit den Methoden [addSeparator](#) und [insertSeparator](#) auch *Separatoren* eingefügt werden. Ein Separator wird als waagerechter Strich angezeigt, der dazu dient, Menüeinträge optisch voneinander zu trennen. Für den Nachrichtenfluß oder die Funktionalität eines Menüs hat ein Separator keine Bedeutung. Während [addSeparator](#) den Separator hinter dem zuletzt eingefügten Menüeintrag einfügt, kann bei [insertSeparator](#) die Einfügeposition frei angegeben werden:

```
public void addSeparator()

public void insertSeparator(int index)
```

java.awt.Menu

Mit [getItem](#) kann schließlich auf einen beliebigen Menüeintrag zugegriffen werden, und die Methode [getItemCount](#) liefert die Anzahl der Einträge des Menüs:

```
public MenuItem getItem(int index)

public int getItemCount()
```

java.awt.Menu

20.4 Menüeinträge

- [20.4 Menüeinträge](#)
 - [20.4.1 Einfache Menüeinträge](#)
 - [20.4.2 CheckboxMenuItem](#)
 - [20.4.3 Beschleunigertasten](#)
 - [20.4.4 Untermenüs](#)

20.4.1 Einfache Menüeinträge

Die *Menüeinträge* sind die elementaren Bestandteile eines Menüs. Sie besitzen einen Text, mit dem sie dem Anwender die dahinterstehende Funktion anzeigen. Wenn der zugehörige Menüpunkt aufgerufen wird, sendet das Programm eine Nachricht an das zugehörige Fenster, die dann zum Aufruf der entsprechenden Methode führt.

Menüeinträge werden in Java mit der Klasse [MenuItem](#) erzeugt. Ihr Konstruktor erwartet als Parameter einen [String](#), der den Namen des Menüeintrags angibt:

```

java.awt.MenuItem
public MenuItem(String label)

```

Auch nach der Konstruktion eines Menüeintrags ist ein Zugriff auf seinen Namen möglich. Mit der Methode [getLabel](#) kann der Name des Menüeintrags abgefragt und mit [setLabel](#) sogar verändert werden:

```

java.awt.MenuItem
public String getLabel()

public void setLabel(String label)

```

Neben einem Namen besitzt ein Menüeintrag eine interne Zustandsvariable, die anzeigt, ob er *aktiv* ist oder nicht. Nur ein aktiver Eintrag kann vom Anwender ausgewählt werden und so eine Nachricht auslösen. Ein inaktiver Eintrag dagegen wird im Menü grau dargestellt, kann vom Anwender nicht mehr ausgewählt werden und daher auch keine Nachricht mehr auslösen.

Nach dem Aufruf des Konstruktors ist ein Menüeintrag zunächst aktiviert. Er kann durch Aufruf von [setEnabled\(false\)](#) deaktiviert und mit [setEnabled\(true\)](#) aktiviert werden. Durch Aufruf von [isEnabled](#) kann der aktuelle Zustand abgefragt werden:

```

java.awt.MenuItem
public void setEnabled(boolean b)

public boolean isEnabled()

```

20.4.2 CheckboxMenuItem

Neben der Klasse [MenuItem](#) gibt es mit der Klasse [CheckboxMenuItem](#) eine zweite Klasse zum Erzeugen von Menüeinträgen. [CheckboxMenuItem](#) ist aus [MenuItem](#) abgeleitet und bietet als zusätzliches Feature eine interne Zustandsvariable, die zwischen [true](#) und [false](#) umgeschaltet werden kann. Die visuelle Darstellung der Zustandsvariablen erfolgt durch Anfügen oder Entfernen eines Häkchens neben dem Menüeintrag. Der Nutzen der Klasse [CheckboxMenuItem](#) besteht darin, daß eine logische Programmvariable durch Auswählen des Menüpunkts abwechselnd an- und ausgeschaltet werden kann.

Die Instanzierung eines [CheckboxMenuItem](#) erfolgt wie bei einem [MenuItem](#). Zusätzlich stehen die beiden Methoden [setState](#) und [getState](#) zum Setzen und Abfragen des Zustands zur Verfügung:

```

java.awt.CheckboxMenuItem
public void setState(boolean state)

public boolean getState()

```

Das folgende Programm stellt alle bisher erwähnten Eigenschaften in einem Beispiel dar. Es leitet dazu die Klasse `MainMenu1` aus `MenuBar` ab und erzeugt im Konstruktor die Menüs und Menüeinträge. Gegenüber der einfachen Instanzierung von `MenuBar` bietet die Ableitung den Vorteil, daß die neue Klasse Methoden zur Verfügung stellen kann, die zum Zugriff auf Menüs oder Menüeinträge verwendet werden können.

[Listing2001.java](#)

```

001 /* Listing2001.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class MainMenu1
007 extends MenuBar
008 {
009     private MenuItem miRueck;
010     private CheckboxMenuItem miFarbe;
011
012     public MainMenu1()
013     {
014         Menu m;
015
016         //Datei
017         m = new Menu("Datei");
018         m.add(new MenuItem("Neu"));
019         m.add(new MenuItem("Laden"));
020         m.add(new MenuItem("Speichern"));
021         m.addSeparator();
022         m.add(new MenuItem("Beenden"));
023         add(m);
024
025         //Bearbeiten
026         m = new Menu("Bearbeiten");
027         m.add((miRueck = new MenuItem("Rueckgaengig")));
028         m.addSeparator();
029         m.add(new MenuItem("Ausschneiden"));
030         m.add(new MenuItem("Kopieren"));
031         m.add(new MenuItem("Einfuegen"));
032         m.add(new MenuItem("Loeschen"));
033         add(m);
034
035         //Optionen
036         m = new Menu("Optionen");
037         m.add(new MenuItem("Einstellungen"));
038         m.add((miFarbe = new CheckboxMenuItem("Farbe")));
039         add(m);
040
041         //Rueckgaengig deaktivieren
042         enableRueckgaengig(false);
043
044         //Farbe anschalten
045         setFarbe(true);
046     }
047
048     public void enableRueckgaengig(boolean ena)
049     {
050         if (ena) {
051             miRueck.setEnabled(true);
052         } else {
053             miRueck.setEnabled(false);
054         }
055     }
056
057     public void setFarbe(boolean on)
058     {
059         miFarbe.setState(on);
060     }
061 }
062
063 public class Listing2001

```

```

060 extends Frame
061 {
062     public static void main(String[] args)
063     {
064         Listing2001 wnd = new Listing2001();
065     }
066
067     public Listing2001()
068     {
069         super("Listing2001");
070         setLocation(100,100);
071         setSize(400,300);
072         setMenuBar(new MainMenu1());
073         setVisible(true);
074         addWindowListener(
075             new WindowAdapter() {
076                 public void windowClosing(WindowEvent event)
077                 {
078                     setVisible(false);
079                     dispose();
080                     System.exit(0);
081                 }
082             }
083         );
084     }
085 }

```

Listing 20.1: Erzeugen von Menüs

Das Programm erzeugt eine Menüleiste mit den drei Einträgen »Datei«, »Bearbeiten« und »Optionen«, die in [Abbildung 20.1](#) dargestellt werden:

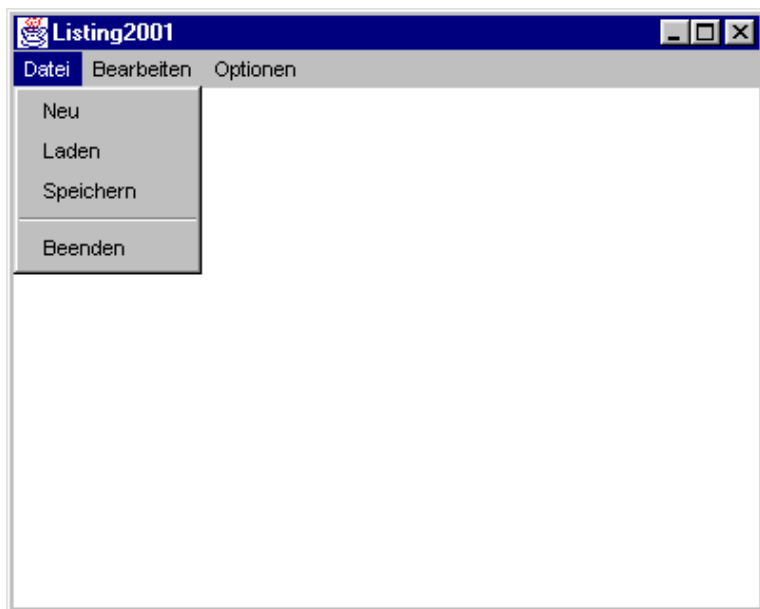


Abbildung 20.1: Erzeugen von Menüs

20.4.3 Beschleunigertasten

In den meisten Programmen lassen sich Menüs nicht nur mit der Maus bedienen, sondern über *Beschleunigertasten* auch mit der Tastatur. Im JDK 1.0 konnten Beschleunigertasten unter Windows 95 ganz einfach dadurch eingefügt werden, daß an beliebiger Stelle im Menütext das Zeichen »&« eingefügt und so die nachfolgende Taste als Beschleuniger definiert wurde. Dies war natürlich nicht portabel, funktionierte nur unter Windows und wurde folglich als Bug angesehen und eliminiert.

Warnung

Das JDK 1.1 implementiert nun ein eigenes Beschleunigerkonzept, das über Plattformgrenzen hinweg funktioniert. Dazu wurde die Klasse [MenuShortcut](#) eingeführt, mit deren Hilfe Beschleunigertasten definiert und an einzelne Menüeinträge angehängt werden können. Eine Beschleunigertaste ist dabei immer ein einzelnes Zeichen der Tastatur, das zusammen mit der *systemspezifischen Umschalttaste für Beschleuniger* ([**STRG**] unter Windows und Motif, [**COMMAND**] unter MAC-OS) gedrückt werden muß, um den Menüeintrag aufzurufen.

Um einen Beschleuniger zu definieren, muß zunächst eine Instanz der Klasse [MenuShortcut](#) erzeugt werden:

```
public MenuShortcut(int key)
public MenuShortcut(int key, boolean useShiftModifier)
```

Der erste Konstruktor erwartet den virtuellen Tastencode der gewünschten Beschleunigertaste (siehe [Kapitel 19](#)). Für einfache alphanumerische Zeichen kann hier auch das Zeichen selbst übergeben werden. Die Übergabe einer Funktionstaste ist leider nicht ohne weiteres möglich, denn deren virtuelle Tastencodes überschneiden sich mit den ASCII-Codes der Kleinbuchstaben. Funktionstasten können daher nur dann als Beschleuniger verwendet werden, wenn ihre virtuellen Tastencodes die Umwandlung in Großbuchstaben unverändert überstehen (z.B. [VK_DELETE](#)). Der zweite Konstruktor erlaubt zusätzlich die Übergabe eines booleschen Parameters `useShiftModifier`, der dafür sorgt, daß der Beschleuniger nur dann greift, wenn neben der systemspezifischen Umschalttaste für Beschleuniger zusätzlich die Taste `[UMSCHALT]` gedrückt wird.

Um einen Beschleuniger an einen Menüpunkt zu binden, ist das [MenuShortcut](#)-Objekt als zweites Argument an den Konstruktor von [MenuItem](#) zu übergeben:

```
public MenuItem(String label, MenuShortcut s)
```

Alternativ kann auch die Methode [setShortCut](#) aufgerufen werden:

```
public void setShortcut(MenuShortcut s)
```

Durch Aufruf von [deleteShortCut](#) kann der einem Menüeintrag zugeordnete Beschleuniger gelöscht werden:

```
public void deleteShortcut()
```

Aufgrund eines Bugs im AWT (der auch im JDK 1.2 noch enthalten ist) muß nach der Definition eines Beschleunigers zusätzlich die Methode [setActionCommand](#) aufgerufen werden, um den [String](#), der beim Auslösen des Beschleunigers an den ActionListener gesendet werden soll, festzulegen:

```
public void setActionCommand(String command)
```

Ohne diesen Aufruf würde ein `null`-Objekt gesendet werden. Eine beispielhafte Aufrufsequenz zur Erzeugung eines Menüeintrags mit Beschleuniger sieht damit so aus:

```
001 Menu m;
002 MenuItem mi;
003 MenuShortcut ms;
004
005 //Datei
006 m = new Menu("Datei");
007
008 ms = new MenuShortcut(KeyEvent.VK_N);
009 mi = new MenuItem("Neu",ms);
010 mi.setActionCommand("Neu");
011 mi.addActionListener(listener);
012 m.add(mi);
```

Listing 20.2: Erzeugen eines Menüeintrags mit Beschleuniger

Hier wird der Menüeintrag »Neu« wie im vorigen Beispiel generiert und mit der Beschleunigertaste `[STRG]+[N]` ausgestattet.

Das folgende Beispiel zeigt eine Menüleiste mit zwei Menüs »Datei« und »Bearbeiten«, bei denen alle Menüeinträge mit Beschleunigern ausgestattet wurden:

Beispiel


```

001 /* MainMenu2.inc */
002
003 class MainMenu2
004 extends MenuBar
005 {
006     public MainMenu2()
007     {
008         Menu m;
009         MenuItem mi;
010         MenuShortcut ms;
011
012         //Datei
013         m = new Menu("Datei");
014
015         ms = new MenuShortcut(KeyEvent.VK_N);
016         mi = new MenuItem("Neu",ms);
017         mi.setActionCommand("Neu");
018         m.add(mi);
019
020         ms = new MenuShortcut(KeyEvent.VK_L);
021         mi = new MenuItem("Laden",ms);
022         mi.setActionCommand("Laden");
023         m.add(mi);
024
025         ms = new MenuShortcut(KeyEvent.VK_S);
026         mi = new MenuItem("Speichern",ms);
027         mi.setActionCommand("Speichern");
028         m.add(mi);
029
030         ms = new MenuShortcut(KeyEvent.VK_E);
031         mi = new MenuItem("Beenden",ms);
032         mi.setActionCommand("Beenden");
033         m.add(mi);
034         add(m);
035
036         //Bearbeiten
037         m = new Menu("Bearbeiten");
038
039         ms = new MenuShortcut(KeyEvent.VK_X);
040         mi = new MenuItem("Ausschneiden",ms);
041         mi.setActionCommand("Ausschneiden");
042         m.add(mi);
043
044         ms = new MenuShortcut(KeyEvent.VK_C);
045         mi = new MenuItem("Kopieren",ms);
046         mi.setActionCommand("Kopieren");
047         m.add(mi);
048
049         ms = new MenuShortcut(KeyEvent.VK_V);
050         mi = new MenuItem("Einfügen",ms);
051         mi.setActionCommand("Einfügen");
052         m.add(mi);
053         add(m);
054     }
055 }

```

Listing 20.3: Menüleisten mit zwei Menüs und Beschleunigertasten

Wir werden später weiter unten eine Methode vorstellen, die den Aufwand für das Erzeugen und Einfügen von Beschleunigern vermindert.

Die im JDK 1.1 eingeführten Beschleuniger haben Vor- und Nachteile. Ihr Vorteil ist, daß sie einfach zu erzeugen sind und über Plattformgrenzen hinweg funktionieren. Der Nachteil ist allerdings ihre eingeschränkte Funktionalität und die Unterschiede im Look & Feel gegenüber den speziellen Beschleunigern des jeweiligen Betriebssystems. So gibt es unter Windows beispielsweise keine Beschleuniger mehr in der Menüleiste ([**ALT**]+Buchstabe), und auch Menüeinträge können nicht mehr mit [**ALT**]+Tastenkürzel aufgerufen werden (sie zeigen auch keinen unterstrichenen Buchstaben mehr an).

Außerdem wird eine Beschleunigtaste zwangsweise an die systemspezifische Umschalttaste gebunden. Es ist damit nicht möglich, einfache Tasten wie `[EINFG]` oder `[ENTF]` als Beschleuniger zu definieren. Des weiteren lassen sich wegen der unglücklichen Umwandlung des virtuellen Tastencodes in Großbuchstaben viele Funktionstasten nicht als Beschleuniger verwenden. Dies sind sicherlich gravierende Restriktionen, die die Bedienung nicht unerheblich einschränken. Es bleibt zu hoffen, daß die nächste Version des AWT hier Verbesserungen bringt und eine umfassendere Menge der plattformspezifischen Features portabel zur Verfügung stellt.

20.4.4 Untermenüs

Menüs lassen sich auf einfache Art und Weise schachteln. Dazu ist beim Aufruf der `add`-Methode anstelle einer Instanz der Klasse `MenuItem` ein Objekt der Klasse `Menu` zu übergeben, das das gewünschte Untermenü repräsentiert. Das folgende Beispiel erweitert das Menü »Optionen« der Klasse `MainMenu1` um den Menüeintrag »Schriftart«, der auf ein Untermenü mit den verfügbaren Schriftarten verzweigt (der Code zur Erzeugung des Untermenüs steht in den Zeilen [034](#) bis [034](#)):

[MainMenu3.inc](#)

```
001 /* MainMenu3.inc */
002
003 class MainMenu3
004 extends MenuBar
005 {
006     private MenuItem miRueck;
007     private CheckboxMenuItem miFarbe;
008
009     public MainMenu3()
010     {
011         Menu m;
012
013         //Datei
014         m = new Menu("Datei");
015         m.add(new MenuItem("Neu"));
016         m.add(new MenuItem("Laden"));
017         m.add(new MenuItem("Speichern"));
018         m.addSeparator();
019         m.add(new MenuItem("Beenden"));
020         add(m);
021         //Bearbeiten
022         m = new Menu("Bearbeiten");
023         m.add((miRueck = new MenuItem("Rueckgaengig")));
024         m.addSeparator();
025         m.add(new MenuItem("Ausschneiden"));
026         m.add(new MenuItem("Kopieren"));
027         m.add(new MenuItem("Einfuegen"));
028         m.add(new MenuItem("Loeschen"));
029         add(m);
030         //Optionen
031         m = new Menu("Optionen");
032         m.add(new MenuItem("Einstellungen"));
033
034         //Untermenü Schriftart
035         Menu m1 = new Menu("Schriftart");
036         m1.add(new MenuItem("Arial"));
037         m1.add(new MenuItem("TimesRoman"));
038         m1.add(new MenuItem("Courier"));
039         m1.add(new MenuItem("System"));
040         m.add(m1);
041         //Ende Untermenü Schriftart
042
043         m.add((miFarbe = new CheckboxMenuItem("Farbe")));
044         add(m);
045         //Rueckgaengig deaktivieren
046         enableRueckgaengig(false);
047         //Farbe anschalten
048         setFarbe(true);
049     }
050
051     public void enableRueckgaengig(boolean ena)
052     {
```

```
053         if (ena) {
054             miRueck.setEnabled(true);
055         } else {
056             miRueck.setEnabled(false);
057         }
058     }
059
060     public void setFarbe(boolean on)
061     {
062         miFarbe.setState(on);
063     }
064 }
```

Listing 20.4: Geschachtelte Menüs

Ein Aufruf des Untermenüs wird folgendermaßen dargestellt:

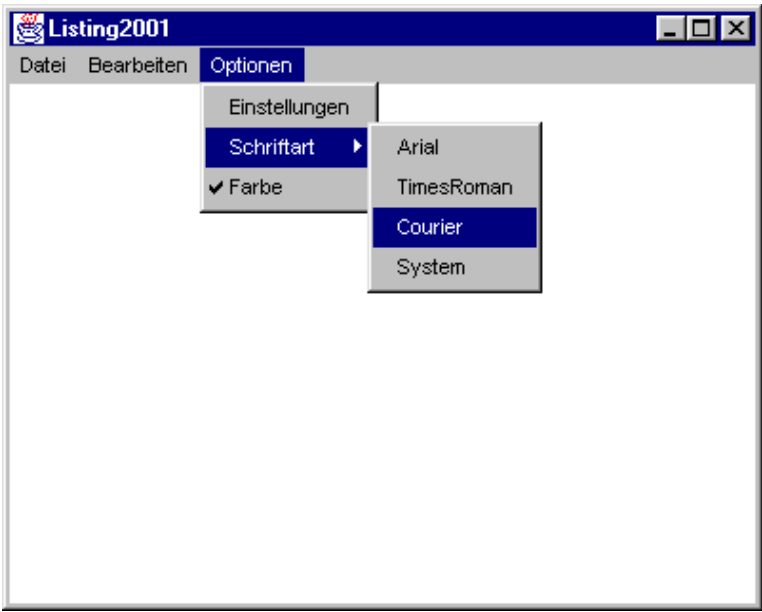


Abbildung 20.2: Geschachtelte Menüs

20.5 Action-Events

- 20.5 Action-Events

Ein Action-Event wird generiert, wenn der Anwender einen Menüpunkt selektiert und ausgewählt hat. Das Programm kann den auslösenden Menüeintrag bestimmen und so geeignet darauf reagieren. Action-Events werden von der Klasse `MenuItem` und drei weiteren Klassen (auf die wir in den folgenden Kapiteln zurückkommen) gesendet:

- `MenuItem` sendet ein `ActionEvent`, wenn der Menüpunkt aufgerufen wurde.
- Ein `Button` sendet ein `ActionEvent`, nachdem er vom Anwender gedrückt wurde.
- Ein Objekt vom Typ `List` sendet ein `ActionEvent` nach einem Doppelclick des Anwenders.
- Ein `TextField` sendet ein `ActionEvent`, wenn der Anwender die Taste `[ENTER]` gedrückt hat.

Ein Empfänger für Action-Events muß das Interface `ActionListener` implementieren und bekommt Events des Typs `ActionEvent` übergeben. `ActionEvent` erweitert die Klasse `AWTEvent` und stellt neben `getID` und `getSource` vor allem die Methode `getActionCommand` zur Verfügung, mit der die verschiedenen Ereignisquellen unterschieden werden können:

`java.awt.event.ActionEvent`

```
public String getActionCommand()
```

Wird das Action-Event durch ein `MenuItem` ausgelöst, liefert dessen Methode `getActionCommand` die Bezeichnung des Menüpunktes, wie sie an den Konstruktor übergeben wurde. Durch einen expliziten Aufruf von `setActionCommand` kann dieser String auch unabhängig von der Beschriftung des Menüpunkts geändert werden. Ein Aufruf von `getActionCommand` liefert den aktuellen Inhalt des Menüpunkts bzw. seine Beschriftung, falls `setActionCommand` noch nicht aufgerufen wurde:

Hinweis

`java.awt.MenuItem`

```
public void setActionCommand(String command)
```

```
public String getActionCommand()
```

Die Registrierung der Empfängerklasse erfolgt mit der Methode `addActionListener`, die in den Klassen `MenuItem`, `Button`, `List` und `TextField` zur Verfügung steht:

`java.awt.Button`

```
public void addActionListener(ActionListener l)
```

Das Interface `ActionListener` stellt lediglich die Methode `actionPerformed` zur Verfügung, die beim Aufruf ein `ActionEvent` übergeben bekommt:

`java.awt.event.ActionListener`

```
public abstract void actionPerformed(ActionEvent e)
```

Üblicherweise wird in `actionPerformed` zunächst durch Aufruf von `getActionCommand` und/oder `getSource` die Quelle des Action-Events ermittelt, bevor der Code folgt, der die Reaktion auf das Ereignis implementiert.

Tip

Das folgende Programm zeigt die Reaktion auf Action-Events. Das Programm öffnet ein Fenster, das mit Hilfe von Menüeinträgen auf dem Bildschirm verschoben oder in der Größe verändert werden kann:

Beispiel

`Listing2005.java`

```
001 /* Listing2005.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 import java.awt.*;
007 import java.awt.event.*;
008
009 class MainMenu4
010 extends MenuBar
011 {
012     private MenuItem miRueck;
```

```

013 private CheckboxMenuItem miFarbe;
014
015 private static void
016 addNewMenuItem(Menu menu, String name, ActionListener listener)
017 {
018     int pos = name.indexOf('&');
019     MenuShortcut shortcut = null;
020     MenuItem mi;
021     if (pos != -1) {
022         if (pos < name.length() - 1) {
023             char c = name.charAt(pos + 1);
024             shortcut=new MenuShortcut(Character.toLowerCase(c));
025             name=name.substring(0,pos)+name.substring(pos + 1);
026         }
027     }
028     if (shortcut != null) {
029         mi = new MenuItem(name, shortcut);
030     } else {
031         mi = new MenuItem(name);
032     }
033     mi.setActionCommand(name);
034     mi.addActionListener(listener);
035     menu.add(mi);
036 }
037
038 public MainMenu4(ActionListener listener)
039 {
040     Menu m;
041
042     //Menü "Größe"
043     m = new Menu("Größe");
044     addNewMenuItem(m, "&Größer", listener);
045     addNewMenuItem(m, "&Kleiner", listener);
046     m.addSeparator();
047     addNewMenuItem(m, "B&eenden", listener);
048     add(m);
049
050     //Menü "Position"
051     m = new Menu("Position");
052     addNewMenuItem(m, "&Links", listener);
053     addNewMenuItem(m, "&Rechts", listener);
054     addNewMenuItem(m, "&Oben", listener);
055     addNewMenuItem(m, "&Unten", listener);
056     add(m);
057 }
058 }
059
060 public class Listing2005
061 extends Frame
062 implements ActionListener
063 {
064     public static void main(String[] args)
065     {
066         Listing2005 wnd = new Listing2005();
067     }
068
069     public Listing2005()
070     {
071         super("Listing2005");
072         setLocation(100,100);
073         setSize(300,200);
074         setMenuBar(new MainMenu4(this));
075         setVisible(true);
076         addWindowListener(
077             new WindowAdapter() {

```

```

078         public void windowClosing(WindowEvent event)
079         {
080             setVisible(false);
081             dispose();
082             System.exit(0);
083         }
084     }
085 );
086 }
087
088 public void paint(Graphics g)
089 {
090     Insets in = getInsets();
091     Dimension d = getSize();
092     g.setColor(Color.red);
093     g.drawLine(
094         in.left, in.top,
095         d.width - in.right, d.height - in.bottom
096     );
097     g.drawLine(
098         in.left, d.height - in.bottom,
099         d.width - in.right, in.top
100     );
101 }
102
103 public void actionPerformed(ActionEvent event)
104 {
105     String cmd = event.getActionCommand();
106     if (cmd.equals("Größer")) {
107         Dimension d = getSize();
108         d.height *= 1.05;
109         d.width *= 1.05;
110         setSize(d);
111     } else if (cmd.equals("Kleiner")) {
112         Dimension d = getSize();
113         d.height *= 0.95;
114         d.width *= 0.95;
115         setSize(d);
116     } else if (cmd.equals("Beenden")) {
117         setVisible(false);
118         dispose();
119         System.exit(0);
120     } else if (cmd.equals("Links")) {
121         setLocation(getLocation().x-10, getLocation().y);
122     } else if (cmd.equals("Rechts")) {
123         setLocation(getLocation().x+10, getLocation().y);
124     } else if (cmd.equals("Oben")) {
125         setLocation(getLocation().x, getLocation().y-10);
126     } else if (cmd.equals("Unten")) {
127         setLocation(getLocation().x, getLocation().y+10);
128     }
129 }
130 }

```

Listing 20.5: Reaktion auf Action-Events aus einem Menü

Das Programm besitzt eine Klasse `MainMenu4`, in der das Menü definiert wird. Um die Definition der Menüeinträge zu vereinfachen, wurde die Methode `addNewMenuItem` implementiert, die einen neuen Menüeintrag erzeugt, ggfs. mit einem Beschleuniger versieht, den `ActionListener` registriert und schließlich an das übergebene Menü anhängt. Der erste Parameter von `addNewMenuItem` ist das Menü, für das ein Menüeintrag erstellt werden soll. Der zweite Parameter ist die Bezeichnung des Menüeintrags. Ist darin ein »&« enthalten, so wird dieses als Präfix für die Beschleunigertaste angesehen und der nachfolgende Buchstabe als Beschleuniger registriert. Anschließend wird das »&« entfernt. Als drittes Argument wird der `ActionListener` übergeben, der beim Menüeintrag registriert werden soll.

Wir verwenden in diesem Beispiel lediglich einen einzigen ActionListener, der bei allen Menüeinträgen registriert wird. Über den `this`-Zeiger wird das Fenster an den Konstruktor von `MainMenu4` übergeben und von dort an `addNewMenuItem` weitergegeben. Voraussetzung dafür ist, daß das Fenster das Interface `ActionListener` implementiert und die Methode `actionPerformed` zur Verfügung stellt.

Hinweis

Die Unterscheidung der verschiedenen Ereignisquellen wird innerhalb von `actionPerformed` durch Aufruf von `getActionCommand` erledigt. Deren Rückgabewert wird abgefragt, um das passende Kommando auszuführen. Die Größenänderung erfolgt durch Aufruf von `getSize` und `setSize`, die Positionierung mit `getLocation` und `setLocation`. Zur Kontrolle zeichnet das Programm in `paint` zwei rote Diagonalen über die volle Länge der Client-Area:

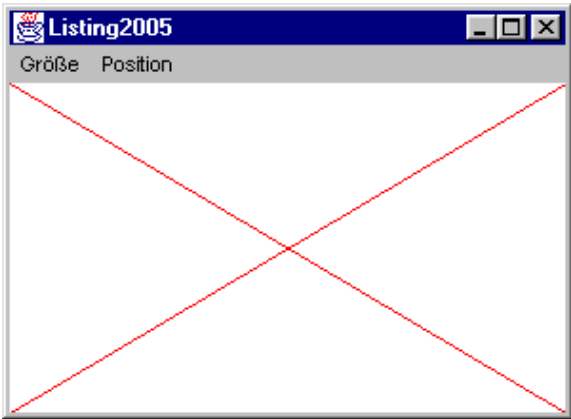


Abbildung 20.3: Ein Programm, das auf Action-Events reagiert

20.6 Kontextmenüs

- 20.6 Kontextmenüs

Ein wichtiger Bestandteil grafischer Oberflächen, der in den letzten Jahren verstärkt in die Programme Einzug gehalten hat, sind die *Kontext-* oder *Popup-Menüs*. Sie liefern auf rechten Mausklick ein Menü mit den wichtigsten Aktionen, die innerhalb der ausgewählten Komponente zur Verfügung stehen. Auf diese Weise steht eine Vielzahl von Funktionen dort zur Verfügung, wo sie benötigt wird, anstatt umständlich aus dem Hauptmenü des Programms ausgewählt oder per Kommando eingegeben werden zu müssen.

Auch im JDK 1.1 gibt es Kontextmenüs, die so funktionieren. Sie werden durch die Klasse `PopupMenu`, die aus `Menu` abgeleitet ist, implementiert. `PopupMenu` ist genauso zu bedienen wie `Menu` und wird daher vor allem mittels geeigneter Aufrufe von `add` mit Menüeinträgen bestückt. Diesen kann ein `ActionListener` zugeordnet werden, der bei Auslösen des Menüpunkts aufgerufen wird. Gegenüber der Klasse `Menu` besitzt `PopupMenu` eine zusätzliche Methode `show`, mit der das Kontextmenü angezeigt wird:

```

                                java.awt.PopupMenu
public void show(Component origin, int x, int y)
```

Der erste Parameter `origin` ist die Komponente, an die das Kontextmenü gebunden wird. Diese Komponente hat lediglich administrative Aufgaben, spielt aber beim Aufruf des Kontextmenüs keine Rolle. Die Argumente `x` und `y` geben die Position des Kontextmenüs relativ zum Ursprung von `origin` an.

Um ein Kontextmenü aufzurufen, sind mehrere Dinge zu tun. Zunächst muß das instanzierte Kontextmenü durch Aufruf von `add` an die Komponente gebunden werden, die auf Mausereignisse für den Aufruf reagieren soll. Dies kann beispielsweise das Fenster sein, in dem die Komponente untergebracht ist, oder die Komponente selbst. Anschließend muß in der Komponente durch Aufruf von `enableEvents` die Behandlung von Maus-Events freigeschaltet werden. Drittens muß die Methode `processMouseEvent` überlagert werden, und es muß bei jedem Mausereignis mit `isPopupTrigger` abgefragt werden, ob es sich um das Ereignis zum Aufruf des Kontextmenüs handelte. In diesem Fall kann das Kontextmenü durch Aufruf von `show` angezeigt werden.

Kontextmenüs sind von Plattform zu Plattform leicht unterschiedlich implementiert, und insbesondere die Art des Aufrufs unterscheidet sich voneinander (zweite oder dritte Maustaste, Aufruf beim Drücken oder Loslassen usw.). Um das Look & Feel des jeweiligen Systems beizubehalten, sollte `processMouseEvent` überlagert werden, um bei *jeder* Art von Mausereignis feststellen zu können, ob der PopupMenu-Trigger ausgelöst wurde. Der einfache Aufruf von `show` aus einem `mousePressed`- oder `mouseReleased`-Ereignis heraus ist nicht portabel und sollte daher vermieden werden.

Hinweis

Das folgende Beispiel zeigt ein Programm mit einem Kontextmenü, das die Punkte »Rückgängig«, »Ausschneiden«, »Kopieren« und »Einfügen« enthält. Das Kontextmenü wird an das Hauptfenster gehängt und von Mausereignissen dieses Fensters aufgerufen. Beim Auslösen einer Option des Kontextmenüs wird eine entsprechende Meldung auf die Systemkonsole geschrieben.

Beispiel

```

                                Listing2006.java
001  /* Listing2006.java */
002
003  import java.awt.*;
004  import java.awt.event.*;
005
006  class MyPopupMenu
007  extends PopupMenu
008  {
009      public MyPopupMenu(ActionListener listener)
010      {
011          MenuItem mi;
012
013          mi = new MenuItem("Rueckgaengig");
014          mi.addActionListener(listener);
015          add(mi);
016
017          addSeparator();
018
019          mi = new MenuItem("Ausschneiden");
020          mi.addActionListener(listener);
```



```

021         add(mi);
022
023         mi = new MenuItem("Kopieren");
024         mi.addActionListener(listener);
025         add(mi);
026
027         mi = new MenuItem("Einfuegen");
028         mi.addActionListener(listener);
029         add(mi);
030     }
031 }
032
033 public class Listing2006
034 extends Frame
035 implements ActionListener
036 {
037     MyPopupMenu popup;
038
039     public static void main(String[] args)
040     {
041         Listing2006 wnd = new Listing2006();
042     }
043
044     public Listing2006()
045     {
046         super("Listing2006");
047         setLocation(100,100);
048         setSize(300,200);
049         setVisible(true);
050         addWindowListener(
051             new WindowAdapter() {
052                 public void windowClosing(WindowEvent event)
053                 {
054                     setVisible(false);
055                     dispose();
056                     System.exit(0);
057                 }
058             }
059         );
060         //Kontextmenü erzeugen und aktivieren
061         popup = new MyPopupMenu(this);
062         add(popup);
063         enableEvents(AWTEvent.MOUSE_EVENT_MASK);
064     }
065
066     public void processMouseEvent(MouseEvent event)
067     {
068         if (event.isPopupTrigger()) {
069             popup.show(
070                 event.getComponent(),
071                 event.getX(),
072                 event.getY()
073             );
074         }
075         super.processMouseEvent(event);
076     }
077
078     public void actionPerformed(ActionEvent event)
079     {
080         System.out.println(event.getActionCommand());
081     }
082 }

```

Listing 20.6: Einbinden eines Kontextmenüs

[Abbildung 20.4](#) zeigt den Aufruf des Kontextmenüs:

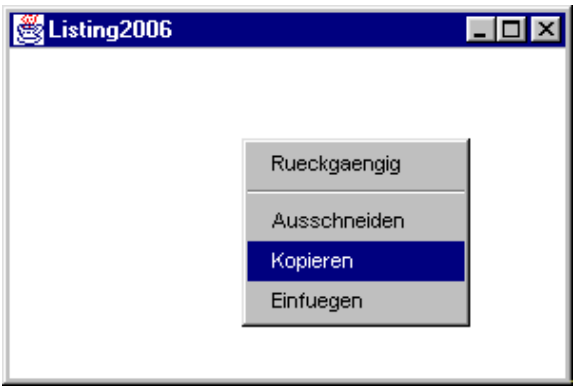


Abbildung 20.4: Aufruf eines Kontextmenüs

20.7 Datenaustausch mit der Zwischenablage

- [20.7 Datenaustausch mit der Zwischenablage](#)
 - [20.7.1 Überblick](#)
 - [20.7.2 Kommunikation mit der Zwischenablage](#)

20.7.1 Überblick

Die Zwischenablage ist in grafischen Oberflächen eines der wichtigsten Hilfsmittel, um Daten zwischen Dialogelementen oder über Anwendungsgrenzen hinweg auszutauschen. Seit dem JDK 1.1 gibt es ein allgemeines API für den Datenaustausch, das Funktionen für den Datenaustausch mit der Zwischenablage zur Verfügung stellt. In zukünftigen Versionen wird es auch für die Unterstützung des Drag & Drop von Bedeutung sein.

Die zugehörigen Klassen und Interfaces befinden sich im Paket [java.awt.datatransfer](#). Die für die Kommunikation mit der Zwischenablage interessanten Bestandteile dieses APIs sind:

- Das Interface [Transferable](#), das die Schnittstelle von Objekten festlegt, die mit der Zwischenablage ausgetauscht werden können.
- Die Definition der Datentypen, die ausgetauscht werden können. Diese werden in Java als *DataFlavors* bezeichnet und basieren auf der *MIME-Spezifikation (Multi-purpose Internet Mail Extensions)*, die in RFC 1521 und 1522 beschrieben wird.
- Die Klasse [Clipboard](#), mit der die Zwischenablage implementiert wird. Neben der systemweiten Zwischenablage können auch benutzerdefinierte Zwischenablagen verwendet werden.

20.7.2 Kommunikation mit der Zwischenablage

Wir wollen uns nun ansehen, welche Schritte erforderlich sind, um einen [String](#) in die Zwischenablage zu kopieren. Zunächst muß ein Objekt erzeugt werden, das das Interface [Transferable](#) implementiert. [Transferable](#) spezifiziert die Schnittstelle für den Transportmechanismus von Daten, die über die Zwischenablage ausgetauscht werden können. Es verwaltet eine Liste der in Frage kommenden Datentypen, die es mit [getTransferDataFlavors](#) auf Anfrage zur Verfügung stellt. Mit [isDataFlavorSupported](#) kann festgestellt werden, ob ein bestimmter Datentyp unterstützt wird, und [getTransferData](#) liefert die zu übertragenden Daten:

```

public DataFlavor[] getTransferDataFlavors()

public boolean isDataFlavorSupported(DataFlavor flavor)

public Object getTransferData(DataFlavor flavor)
    throws UnsupportedFlavorException, IOException
    
```

Die Initialisierung eines [Transferable](#)-Objekts ist nicht Bestandteil der Schnittstelle, sondern muß von den implementierenden Klassen in Eigenregie - beispielsweise bei der Instanzierung - vorgenommen werden.

Hinweis

Im AWT gibt es eine vordefinierte Klasse [StringSelection](#), die das Interface [Transferable](#) implementiert. Sie ist in der Lage, Strings auszutauschen und unterstützt die aus [DataFlavor](#) abgeleiteten Datentypen [plainTextFlavor](#) und [stringFlavor](#). Beide liefern die Daten als Unicode-kodierte Zeichenkette. Während [plainTextFlavor](#) sein Ergebnis als [InputStream](#) zur Verfügung stellt und den MIME-Typ `text/plain` repräsentiert, liefert [stringFlavor](#) einen String und repräsentiert den MIME-Typ `application/x-java-serialized-object`.

Ein [StringSelection](#)-Objekt wird initialisiert, indem die zu übertragende Zeichenkette an den Konstruktor übergeben wird. Anschließend kann es an die Zwischenablage übergeben werden, die die Daten durch Aufruf von [getTransferData](#) übernimmt. Jeder Aufruf von [getTransferData](#) muß in eine `try-catch`-Anweisung eingebunden werden und folgende Fehler abfangen:

- Das [Transferable](#)-Objekt ist nicht in der Lage, die Daten in dem gewünschten Format zu liefern.
- Da der Rückgabewert vom Typ [Object](#) ist, muß er in der Regel in den tatsächlich erforderlichen Typ konvertiert werden. Hierbei kann eine Ausnahme des Typs [ClassCastException](#) auftreten.

Bevor die Zwischenablage die Daten aus dem [Transferable](#)-Objekt entnehmen kann, muß dieses natürlich erst einmal an die Zwischenablage übergeben werden. Eine Zwischenablage ist immer eine Instanz der Klasse [Clipboard](#) oder einer ihrer Unterklassen. Zwar ist es möglich, anwendungsspezifische Zwischenablagen anzulegen, wir wollen uns aber nur mit der systemweit gültigen Zwischenablage des Betriebssystems beschäftigen.

Im [Toolkit](#) gibt es eine Methode [getSystemClipboard](#), mit der ein Objekt für die systemweite Zwischenablage beschafft werden kann:

```
public Clipboard getSystemClipboard()  
// java.awt.Toolkit
```

Sie stellt im wesentlichen einen Konstruktor und drei Methoden zur Verfügung:

```
public String getName()  
  
public Transferable getContents(Object requestor)  
  
public void setContents(  
    Transferable contents, ClipboardOwner owner  
)
```

Mit [getName](#) kann der Name der Zwischenablage ermittelt werden, [getContents](#) liefert den Inhalt der Zwischenablage, und mit [setContents](#) kann der Zwischenablage ein neues Objekt zugewiesen werden. Ein Aufruf von [getContents](#) liefert null, wenn die Zwischenablage leer ist. Der Rückgabewert ist ein [Transferable](#)-Objekt, dessen Daten mit [getTransferData](#) abgefragt werden können. Beim Aufruf von [getContents](#) muß zusätzlich ein Objekt [requestor](#) übergeben werden, das derzeit keine Funktion hat.

Ein Objekt, das den Inhalt der Zwischenablage ändern will, tut dies über den Aufruf der Methode [setContents](#). Als erstes Argument ist ein [Transferable](#)-Objekt zu übergeben, das die Daten enthält. Als zweites muß ein Objekt übergeben werden, das das Interface [ClipboardOwner](#) implementiert. Da die Zwischenablage von verschiedenen Objekten verwendet wird, ist es unter Umständen wichtig zu wissen, wann die übergebenen Daten verworfen und durch ein neues Objekt ersetzt werden. Dazu definiert [ClipboardOwner](#) die Methode [lostOwnership](#), die aufgerufen wird, wenn der Inhalt der Zwischenablage verändert wird:

```
public void lostOwnership(  
    Clipboard clipboard, Transferable contents  
)  
// java.awt.datatransfer.ClipboardOwner
```

Nach diesen Vorüberlegungen wollen wir uns ein Beispiel ansehen. Dazu soll die Methode [actionPerformed](#) des vorigen Beispiels erweitert werden, um die beiden Menüeinträge »Kopieren« und »Einfügen« mit Funktionalität zum Datenaustausch auszustatten. Zusätzlich implementiert das Beispielprogramm das Interface [ClipboardOwner](#) und definiert dazu die Methode [lostOwnership](#):

Beispiel

```
001 /* clpbrd.inc */  
002  
003 public void actionPerformed(ActionEvent event)  
004 {  
005     Clipboard clip = getToolkit().getSystemClipboard();  
006     String cmd = event.getActionCommand();  
007     if (cmd.equals("Kopieren")) {  
008         String s = "Es ist " + System.currentTimeMillis() + " Uhr";  
009         StringSelection cont = new StringSelection(s);  
010         clip.setContents(cont, this);  
011     } else if (cmd.equals("Einfuegen")) {  
012         Transferable cont = clip.getContents(this);  
013         if (cont == null) {  
014             System.out.println("Zwischenablage ist leer");  
015         } else {  
016             try {  
017                 String s = (String) cont.getTransferData(  
018                     DataFlavor.stringFlavor  
019                 );  
020                 System.out.println(s);  
021             } catch (Exception e) {  
022                 System.out.println(  
023                     "Zwischenablage enthält keinen Text"  
024                 );  
025             }  
026         }  
027     }  
028 }  
029  
030 public void lostOwnership(Clipboard clip, Transferable cont)
```

[clpbrd.inc](#)

```
031 {
032     System.out.println("Inhalt der Zwischenablage ersetzt");
033 }
```

Listing 20.7: Kommunikation mit der Zwischenablage

20.8 Zusammenfassung

- [20.8 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Definition einer Menüleiste mit Hilfe der Klasse [MenuBar](#)
- Definition von Menüs mit Hilfe der Klasse [Menu](#)
- Einfügen eines Separators in ein Menü
- Definition eines Menüeintrags mit Hilfe der Klasse [MenuItem](#)
- Verwendung der Klasse [CheckboxMenuItem](#) zum Erzeugen von Menüeinträgen mit einer umschaltbaren Zustandsvariable
- Zuordnen von Beschleunigertasten und die Klasse [MenuShortcut](#)
- Schachteln von Menüs
- Auslösen von Action-Events als Reaktion auf Menüereignisse
- Erzeugen und Einbinden von Kontextmenüs mit Hilfe der Klasse [PopupMenu](#)
- Das Datenaustausch-API des JDK 1.1 und das Paket [java.awt.datatransfer](#)
- Datenaustausch mit der Zwischenablage durch Implementierung des Interfaces [Transferable](#) und Kommunikation mit einem [Clipboard](#)-Objekt

Kapitel 21

Dialoge

- [21 Dialoge](#)
 - [21.1 Erstellen eines Dialogs](#)
 - [21.1.1 Anlegen eines Dialogfensters](#)
 - [21.1.2 Zuordnen eines Layoutmanagers](#)
 - [21.1.3 Einfügen von Dialogelementen](#)
 - [21.1.4 Anzeigen des Dialogfensters](#)
 - [21.2 Die Layoutmanager](#)
 - [21.2.1 FlowLayout](#)
 - [21.2.2 GridLayout](#)
 - [Die Größe der Komponenten](#)
 - [21.2.3 BorderLayout](#)
 - [21.2.4 GridBagLayout](#)
 - [21.2.5 NULL-Layout](#)
 - [21.2.6 Schachteln von Layoutmanagern](#)
 - [21.3 Modale Dialoge](#)
 - [21.4 Zusammenfassung](#)

21.1 Erstellen eines Dialogs

- [21.1 Erstellen eines Dialogs](#)
 - [21.1.1 Anlegen eines Dialogfensters](#)
 - [21.1.2 Zuordnen eines Layoutmanagers](#)
 - [21.1.3 Einfügen von Dialogelementen](#)
 - [21.1.4 Anzeigen des Dialogfensters](#)

Das Erstellen eines Dialogs erfolgt in vier Schritten:

- Anlegen eines Fensters
- Zuordnen eines Layoutmanagers
- Einfügen von Dialogelementen
- Anzeigen des Fensters

21.1.1 Anlegen eines Dialogfensters

Das Anlegen eines Fensters zur Aufnahme von Dialogelementen erfolgt genauso wie das Anlegen eines normalen Fensters. Üblicherweise wird dazu eine eigene Fensterklasse abgeleitet, um die Steuerung des Dialogs zu kapseln.

Da Java prinzipiell keinen Unterschied zwischen Fenstern zur Ausgabe eines Dialogs und solchen zur Anzeige von Grafiken macht, ist es möglich, ein Dialogfenster wahlweise aus [Frame](#) oder [Dialog](#) abzuleiten. Die Klasse [Dialog](#) erlaubt es, das Verändern der Fenstergröße durch den Anwender zu unterbinden und bietet die Möglichkeit, den Dialog *modal* zu machen. Dadurch wird die Interaktion des Anwenders mit *anderen* Fenstern der Anwendung bis zum Schließen des Dialogfensters blockiert. Im Gegensatz zu [Frame](#) fehlt jedoch die Möglichkeit, eine Menüleiste zu erzeugen oder dem Fenster ein Icon zuzuordnen. Wir werden in den nachfolgenden Beispielen meist die Klasse [Frame](#) verwenden, um Dialoge zu erzeugen. Die Klasse [Dialog](#) werden wir am Ende dieses Kapitels vorstellen. Dabei werden wir insbesondere das Erzeugen modaler Dialoge und die Rückgabe von Ergebniswerten aufzeigen.

Hinweis

21.1.2 Zuordnen eines Layoutmanagers

Wie bereits erwähnt, sind die Layoutmanager in Java für die Anordnung der Dialogelemente im Fenster verantwortlich. Jeder Layoutmanager verfolgt dabei eine eigene Strategie, Elemente zu platzieren und in der Größe so anzupassen, daß sie aus seiner Sicht optimal präsentiert werden.

Die Zuordnung eines Layoutmanagers zu einem Fenster wird in der Klasse [Container](#) realisiert. [Container](#) ist direkt aus [Component](#) abgeleitet, und beide zusammen bilden das Gerüst für alle anderen Fensterklassen. Die Klasse [Container](#) stellt eine Methode [setLayout](#) zur Verfügung, mit der der gewünschte Layoutmanager dem Fenster zugeordnet werden kann:

```

public void setLayout(LayoutManager mgr)

```

[java.awt.Container](#)

Java stellt standardmäßig die fünf Layoutmanager [FlowLayout](#), [GridLayout](#), [BorderLayout](#), [CardLayout](#) und [GridBagLayout](#) zur Verfügung. Der einfachste Layoutmanager ist [FlowLayout](#), er positioniert die Dialogelemente zeilenweise hintereinander. Paßt ein Element nicht mehr in die aktuelle Zeile, so wird es in der nächsten plaziert usw. Die genaue Funktionsweise der Layoutmanager wird weiter unten in diesem Kapitel vorgestellt.

21.1.3 Einfügen von Dialogelementen

Das Einfügen von Dialogelementen in das Fenster erfolgt mit der Methode [add](#) der Klasse [Container](#):

```

public Component add(Component comp)

public Component add(Component comp, int pos)

public Component add(String name, Component comp)

```

[java.awt.Container](#)

Bei der ersten Variante wird lediglich die einzufügende Komponente übergeben und vom Layoutmanager an der dafür vorgesehenen Position untergebracht. Die zweite Variante erlaubt das Einfügen der aktuellen Komponente an beliebiger Stelle in der Liste der Komponenten.

Die dritte Variante erwartet zusätzlich einen `String`-Parameter, der bei bestimmten Layoutmanagern weitere Informationen zur Positionierung der Komponente angibt. Wird beispielsweise die Klasse `BorderLayout` zur Anordnung der Dialogelemente verwendet, kann hier einer der Werte `South`, `North`, `West`, `East` oder `Center` übergeben werden, um anzuzeigen, an welcher Stelle des Fensters das Element plaziert werden soll.

Sollen Komponenten, die bereits an das Fenster übergeben wurden, wieder daraus entfernt werden, so kann dazu die Methode `remove` verwendet werden:

java.awt.Container

```
public void remove(Component comp)
```

Als Parameter ist dabei das zu löschende Objekt zu übergeben.

21.1.4 Anzeigen des Dialogfensters

Wurden alle Komponenten an den Container übergeben, kann der Dialog formatiert und durch einen Aufruf von `setVisible` angezeigt werden. Zweckmäßigerweise sollte vorher die Methode `pack` der Klasse `Window` aufgerufen werden, um die Größe des Fensters an den zur Darstellung der Dialogelemente erforderlichen Platz anzupassen:

java.awt.Window

```
public void pack()
```

Wir wollen uns ein einfaches Beispiel ansehen, das diese vier Schritte demonstriert:

Listing2101.java

Beispiel

```
001 /* Listing2101.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2101
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2101 wnd = new Listing2101();
012         wnd.setVisible(true);
013     }
014
015     public Listing2101()
016     {
017         super("Dialogtest");
018         //WindowListener hinzufügen
019         addWindowListener(
020             new WindowAdapter() {
021                 public void windowClosing(WindowEvent event)
022                 {
023                     setVisible(false);
024                     dispose();
025                     System.exit(0);
026                 }
027             }
028         );
029         //Layout setzen und Komponenten hinzufügen
030         setLayout(new FlowLayout());
031         add(new Button("Abbruch"));
032         add(new Button("OK"));
033         pack();
034     }
035 }
```

Listing 21.1: Ein Dialog mit zwei Buttons

Das Programm erzeugt ein kleines Fenster, das nur die beiden Buttons enthält. Da wir für das Fenster keine Größe angegeben haben, sondern diese durch Aufruf von `pack` automatisch berechnen lassen, ist das Fenster gerade so groß, daß beide Buttons darin Platz finden:

Hinweis



Abbildung 21.1: Ein Dialog mit zwei Buttons

21.2 Die Layoutmanager

- [21.2 Die Layoutmanager](#)
 - [21.2.1 FlowLayout](#)
 - [21.2.2 GridLayout](#)
 - [Die Größe der Komponenten](#)
 - [21.2.3 BorderLayout](#)
 - [21.2.4 GridBagLayout](#)
 - [21.2.5 NULL-Layout](#)
 - [21.2.6 Schachteln von Layoutmanagern](#)

In vielen grafischen Oberflächen wird die Anordnung der Elemente eines Dialoges durch Angabe *absoluter Koordinaten* vorgenommen. Dabei wird für jede Komponente manuell oder mit Hilfe eines Ressourcen-Editors pixelgenau festgelegt, an welcher Stelle im Dialog sie zu erscheinen hat.

Da Java-Programme auf vielen unterschiedlichen Plattformen mit unterschiedlichen Ausgabegeräten laufen sollen, war eine solche Vorgehensweise für die Designer des AWT nicht akzeptabel. Sie wählten statt dessen den Umweg über einen Layoutmanager, der für die Anordnung der Dialogelemente verantwortlich ist. Um einen Layoutmanager verwenden zu können, wird dieser dem Fenster vor der Übergabe der Dialogelemente mit der Methode [setLayout](#) zugeordnet. Er ordnet dann die per [add](#) übergebenen Elemente auf dem Fenster an.

Jeder Layoutmanager implementiert seine eigene Logik bezüglich der optimalen Anordnung der Komponenten:

- Das [FlowLayout](#) ordnet Dialogelemente nebeneinander in einer Zeile an. Wenn keine weiteren Elemente in die Zeile passen, wird mit der nächsten Zeile fortgefahren.
- Das [GridLayout](#) ordnet die Dialogelemente in einem rechteckigen Gitter an, dessen Zeilen- und Spaltenzahl beim Erstellen des Layoutmanagers angegeben wird.
- Das [BorderLayout](#) verteilt die Dialogelemente nach Vorgabe des Programms auf die vier Randbereiche und den Mittelbereich des Fensters.
- Das [CardLayout](#) ist in der Lage, mehrere Unterdialoge in einem Fenster unterzubringen und jeweils einen davon auf Anforderung des Programms anzuzeigen.
- Das [GridBagLayout](#) ist ein komplexer Layoutmanager, der die Fähigkeiten von [GridLayout](#) erweitert und es ermöglicht, mit Hilfe von Bedingungsobjekten sehr komplexe Layouts zu erzeugen.

Neben den gestalterischen Fähigkeiten eines Layoutmanagers bestimmt in der Regel die Reihenfolge der Aufrufe der [add](#)-Methode des Fensters die tatsächliche Anordnung der Komponenten auf dem Bildschirm. Wenn nicht - wie es z.B. beim [BorderLayout](#) möglich ist - zusätzliche Positionierungsinformationen an das Fenster übergeben werden, ordnet der jeweilige Layoutmanager die Komponenten in der Reihenfolge ihres Eintreffens an.

Um komplexere Layouts realisieren zu können, als die Layoutmanager sie in ihren jeweiligen Grundausrägungen bieten, gibt es die Möglichkeit, Layoutmanager zu schachteln. Auf diese Weise kann auch ohne Vorgabe fester Koordinaten fast jede gewünschte Komponentenanzordnung realisiert werden. Sollte auch diese Variante nicht genau genug sein, so bietet sich schließlich durch Verwendung eines *Null-Layouts* die Möglichkeit an, Komponenten durch Vorgabe fester Koordinaten zu platzieren.

21.2.1 FlowLayout

Die Klasse [FlowLayout](#) stellt den einfachsten Layoutmanager dar. Alle Elemente werden so lange nacheinander in einer Zeile angeordnet, bis kein Platz mehr vorhanden ist und in der nächsten Zeile fortgefahren wird.

Das [FlowLayout](#) wird einem Fenster durch folgende Anweisung zugeordnet:

```
setLayout(new FlowLayout());
```

Neben dem parameterlosen gibt es weitere Konstruktoren, die die Möglichkeit bieten, die Anordnung der Elemente und ihre Abstände voneinander vorzugeben:

```
public FlowLayout(int align)
public FlowLayout(int align, int hgap, int vgap)
```

[java.awt.FlowLayout](#)

Der Parameter `align` gibt an, ob die Elemente einer Zeile linksbündig, rechtsbündig oder zentriert angeordnet werden. Hierzu stehen die Konstanten `FlowLayout.CENTER`, `FlowLayout.LEFT` und `FlowLayout.RIGHT` zur Verfügung, als Voreinstellung wird `FlowLayout.CENTER` verwendet. Mit `hgap` und `vgap` können die horizontalen und vertikalen Abstände zwischen den Komponenten vorgegeben werden. Die Voreinstellung ist 5.

Das folgende Listing zeigt die Verwendung der Klasse `FlowLayout` am Beispiel von fünf Buttons, die in einem Grafikfenster angezeigt werden. Anstelle der normalen zentrierten Anordnung werden die Elemente linksbündig angeordnet, und der Abstand zwischen den Buttons beträgt 20 Pixel:

Beispiel

[Listing2102.java](#)

```
001 /* Listing2102.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2102
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2102 wnd = new Listing2102();
012         wnd.setVisible(true);
013     }
014
015     public Listing2102()
016     {
017         super("Test FlowLayout");
018         //WindowListener hinzufügen
019         addWindowListener(
020             new WindowAdapter() {
021                 public void windowClosing(WindowEvent event)
022                 {
023                     setVisible(false);
024                     dispose();
025                     System.exit(0);
026                 }
027             }
028         );
029         //Layout setzen und Komponenten hinzufügen
030         setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));
031         add(new Button("Button 1"));
032         add(new Button("Button 2"));
033         add(new Button("Button 3"));
034         add(new Button("Button 4"));
035         add(new Button("Button 5"));
036         pack();
037     }
038 }
```

Listing 21.2: Die Klasse `FlowLayout`

Das Programm erzeugt folgendes Fenster:



Abbildung 21.2: Verwendung der Klasse `FlowLayout`

21.2.2 GridLayout

Ein `GridLayout` bietet eine größere Kontrolle über die Anordnung der Elemente als ein `FlowLayout`. Hier werden die Komponenten nicht einfach nacheinander auf dem Bildschirm positioniert, sondern innerhalb eines rechteckigen Gitters angeordnet, dessen Elemente eine feste Größe haben.

Das Programm übergibt dazu beim Aufruf des Konstruktors zwei Parameter, `rows` und `columns`, mit denen die vertikale und horizontale Anzahl

an Elementen festgelegt wird:

[java.awt.GridLayout](#)

```
public void GridLayout(int rows, int columns)
```

Beim Aufruf von [add](#) werden die Komponenten dann nacheinander in die einzelnen Zellen der Gittermatrix gelegt. Analog zum [FlowLayout](#) wird dabei zunächst die erste Zeile von links nach rechts gefüllt, dann die zweite usw.

Ähnlich wie beim [FlowLayout](#) steht ein zusätzlicher Konstruktor zur Verfügung, der es erlaubt, die Größe der horizontalen und vertikalen Lücken zu verändern:

[java.awt.GridLayout](#)

```
public void GridLayout(int rows, int columns, int hgap, int vgap)
```

Die größere Kontrolle des Programms über das Layout besteht nun darin, daß der Umbruch in die nächste Zeile genau vorhergesagt werden kann. Anders als beim [FlowLayout](#) erfolgt dieser nicht erst dann, wenn keine weiteren Elemente in die Zeile passen, sondern wenn dort so viele Elemente plaziert wurden, wie das Programm vorgegeben hat.

Das folgende Beispiel zeigt ein Fenster mit einem Gitter der Größe 4*2 Elemente, das insgesamt sieben Buttons anzeigt:

Beispiel

[Listing2103.java](#)

```
001 /* Listing2103.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2103
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2103 wnd = new Listing2103();
012         wnd.setVisible(true);
013     }
014
015     public Listing2103()
016     {
017         super("Test GridLayout");
018         //WindowListener hinzufügen
019         addWindowListener(
020             new WindowAdapter() {
021                 public void windowClosing(WindowEvent event)
022                 {
023                     setVisible(false);
024                     dispose();
025                     System.exit(0);
026                 }
027             }
028         );
029         //Layout setzen und Komponenten hinzufügen
030         setLayout(new GridLayout(4,2));
031         add(new Button("Button 1"));
032         add(new Button("Button 2"));
033         add(new Button("Button 3"));
034         add(new Button("Button 4"));
035         add(new Button("Button 5"));
036         add(new Button("Button 6"));
037         add(new Button("Button 7"));
038         pack();
039     }
040 }
```

Listing 21.3: Die Klasse GridLayout

Die Ausgabe des Programms ist:

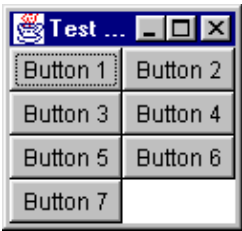


Abbildung 21.3: Verwendung der Klasse `GridLayout`

Die Größe der Komponenten

Wenn wir die Größe des Fensters nicht durch einen Aufruf von `pack`, sondern manuell festgelegt hätten, wäre an dieser Stelle ein wichtiger Unterschied zwischen den beiden bisher vorgestellten Layoutmanagern deutlich geworden. [Abbildung 21.4](#) zeigt das veränderte [Listing 21.2](#), bei dem die Größe des Fensters durch Aufruf von `setSize(500,200);` auf 500*200 Pixel festgelegt und der Aufruf von `pack` entfernt wurde:

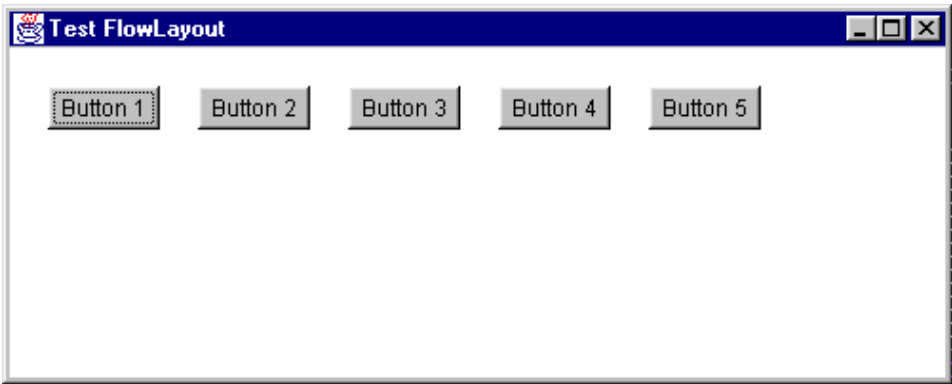


Abbildung 21.4: Das `FlowLayout` in einem größeren Fenster

Hier verhält sich das Programm noch so, wie wir es erwarten würden, denn die Buttons haben ihre Größe behalten, während das Fenster größer geworden ist. Anders sieht es dagegen aus, wenn wir die Fenstergröße in [Listing 21.3](#) ebenfalls auf 500*200 Pixel fixieren:

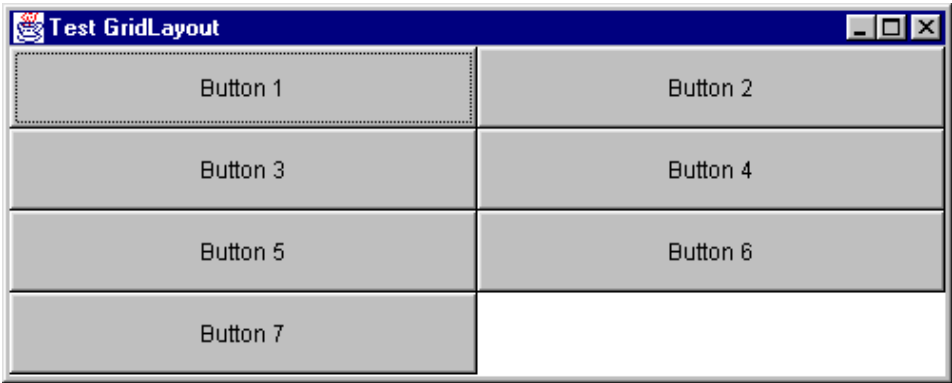


Abbildung 21.5: Das `GridLayout` in einem größeren Fenster

Nun werden die Buttons plötzlich sehr viel größer als ursprünglich angezeigt, obwohl sie eigentlich weniger Platz in Anspruch nehmen würden. Der Unterschied besteht darin, daß ein `FlowLayout` die gewünschte Größe eines Dialogelements verwendet, um seine Ausmaße zu bestimmen. Das `GridLayout` dagegen ignoriert die gewünschte Größe und dimensioniert die Dialogelemente fest in der Größe eines Gitterelements. Ein `LayoutManager` hat also offensichtlich die Freiheit zu entscheiden, ob und in welcher Weise er die Größen von Fenster und Dialogelementen den aktuellen Erfordernissen anpaßt.

Wir werden am Ende dieses Abschnitts auch Möglichkeiten kennenlernen, durch Schachteln von Layoutmanagern beide Möglichkeiten zu kombinieren: Das feste Gitterraster bleibt erhalten, aber die einzelnen Komponenten innerhalb des Gitters werden in ihrer gewünschten Größe angezeigt. Am Ende von [Kapitel 22](#) werden wir zusätzlich kurz auf die verschiedenen Größen von Dialogelementen zurückkommen, wenn wir zeigen, wie selbstdefinierte Komponenten erzeugt werden.

Hinweis

21.2.3 BorderLayout

Das [BorderLayout](#) verfolgt einen anderen Ansatz als die beiden vorigen Layoutmanager, denn die Positionierung der Komponenten wird nicht mehr primär durch die Reihenfolge der Aufrufe von [add](#) bestimmt. Statt dessen teilt das [BorderLayout](#) den Bildschirm in fünf Bereiche auf, und zwar in die vier Ränder und das Zentrum. Durch Angabe eines *Himmelsrichtungs-Strings* wird beim Aufruf von [add](#) angegeben, auf welchem dieser Bereiche die Komponente platziert werden soll:

- "South": unterer Rand
- "North": oberer Rand
- "East": rechter Rand
- "West": linker Rand
- "Center": Mitte

Zur Übergabe dieses Parameters gibt es die Methode [add](#) der Klasse [Container](#) in einer Variante mit einem [String](#) als ersten Parameter:

[java.awt.Container](#)

```
public void add(String name, Component comp)
```

Das folgende Beispiel zeigt die Anordnung von fünf Buttons in einem [BorderLayout](#). Jeweils einer der Buttons wird auf die vier Randbereiche verteilt, und der fünfte Button steht in der Mitte. Die Größe des Fensters wird fest auf 300*200 Pixel eingestellt:

Beispiel

[Listing2104.java](#)

```
001 /* Listing2104.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2104
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2104 wnd = new Listing2104();
012         wnd.setVisible(true);
013     }
014
015     public Listing2104()
016     {
017         super("Test BorderLayout");
018         //WindowListener hinzufügen
019         addWindowListener(
020             new WindowAdapter() {
021                 public void windowClosing(WindowEvent event)
022                 {
023                     setVisible(false);
024                     dispose();
025                     System.exit(0);
026                 }
027             }
028         );
029         //Layout setzen und Komponenten hinzufügen
030         setSize(300,200);
031         setLayout(new BorderLayout());
032         add("North", new Button("Button 1"));
033         add("South", new Button("Button 2"));
034         add("West", new Button("Button 3"));
035         add("East", new Button("Button 4"));
036         add("Center",new Button("Button 5"));
037     }
038 }
```

Listing 21.4: Das BorderLayout

Die Ausgabe des Programms ist:

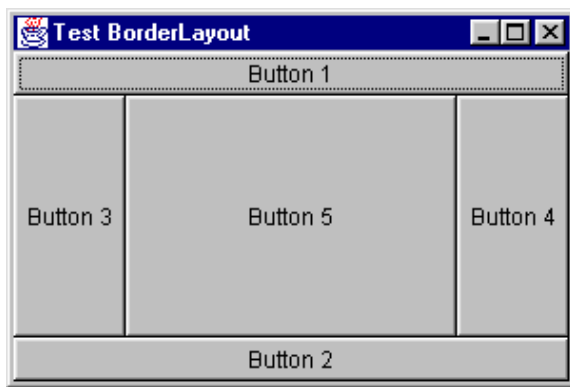


Abbildung 21.6: Verwendung der Klasse BorderLayout

Bezüglich der Skalierung der Komponenten verfolgt [BorderLayout](#) einen Mittelweg zwischen [FlowLayout](#) und [GridLayout](#). Während [FlowLayout](#) die Komponenten immer in ihrer gewünschten Größe beläßt und [GridLayout](#) sie immer skaliert, ist dies bei [BorderLayout](#) von verschiedenen Faktoren abhängig:

- Nord- und Südelement behalten ihre gewünschte Höhe, werden aber auf die volle Fensterbreite skaliert.
- Ost- und Westelement behalten ihre gewünschte Breite, werden aber in der Höhe so skaliert, daß sie genau zwischen Nord- und Südelement passen.
- Das Mittelelement wird in der Höhe und Breite so angepaßt, daß es den verbleibenden freien Raum einnimmt.

Auch beim [BorderLayout](#) kann die Größe der Lücken zwischen den Elementen an den Konstruktor übergeben werden:

[java.awt.Borderlayout](#)

```
public BorderLayout(int hgap, int vgap)
```

Wenn das vorige Beispiel ein [BorderLayout](#) mit einer vertikalen und horizontalen Lücke von 10 Pixeln definieren würde, wäre die Ausgabe des Programms wie folgt:

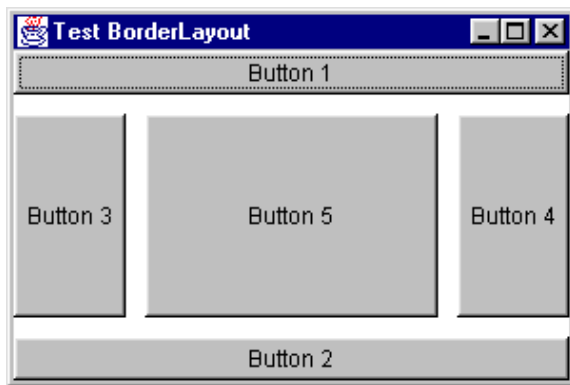


Abbildung 21.7: Ein BorderLayout mit Lücken

21.2.4 GridBagLayout

Das [GridBagLayout](#) ist der aufwendigste Layoutmanager in Java. Er erlaubt eine sehr flexible Gestaltung der Oberfläche und bietet viele Möglichkeiten, die in den anderen Layoutmanagern nicht zu finden sind. Der Preis dafür ist eine etwas kompliziertere Bedienung und ein höherer Einarbeitungsaufwand. Wir wollen in diesem Abschnitt die Klasse [GridBagLayout](#) vorstellen und ihre grundlegenden Anwendungsmöglichkeiten besprechen.

Um mit einem [GridBagLayout](#) zu arbeiten, ist wie folgt vorzugehen:

- Zunächst ist ein Objekt des Typs [GridBagLayout](#) zu instanzieren und durch Aufruf von [setLayout](#) dem Fenster zuzuweisen.
- Für jedes einzufügende Dialogelement ist nun ein Objekt des Typs [GridBagConstraints](#) anzulegen, um über dessen Membervariablen die spezifischen Layouteigenschaften des Dialogelements zu beschreiben.
- Sowohl das Dialogelement als auch das Eigenschaftenobjekt werden durch Aufruf der Methode [setConstraints](#) an den Layoutmanager übergeben.
- Schließlich wird das Dialogelement mit [add](#) an das Fenster übergeben.

Das könnte beispielsweise so aussehen


```

001 ...
002 GridBagLayout gbl = new GridBagLayout();
003 GridBagConstraints gbc = new GridBagConstraints();
004 setLayout(gbl);
005 List list = new List();
006 gbc.gridx = 0;
007 gbc.gridy = 0;
008 gbc.gridwidth = 1;
009 gbc.gridheight = 1;
010 gbc.weightx = 100;
011 gbc.weighty = 100;
012 gbc.fill = GridBagConstraints.BOTH;
013 gbl.setConstraints(list, gbc);
014 add(list);
015 ...

```

Listing 21.5: Umgang mit GridBagLayout und GridBagConstraints

Das Geheimnis der korrekten Verwendung eines [GridBagLayout](#) liegt also in der richtigen Konfiguration der Membervariablen der [GridBagConstraints](#)-Objekte. Es stehen folgende zur Verfügung:

[java.awt.GridBagConstraints](#)

```

public int gridx
public int gridy

public int gridwidth
public int gridheight

public int anchor

public int fill

public double weightx
public double weighty

public Insets insets

public int ipadx
public int ipady

```

Ihre Bedeutung ist:

- Ein [GridBagLayout](#) unterteilt das Fenster ähnlich wie das [GridLayout](#) in ein rechteckiges Gitter von Zellen, in denen die Dialogelemente platziert werden. Im Gegensatz zum [GridLayout](#) können die Zellen allerdings unterschiedliche Größen haben, und die Dialogelemente können auf verschiedene Weise innerhalb der Zellen platziert werden. Zudem kann ein Dialogelement sich in beiden Richtungen über mehr als eine einzige Zelle erstrecken.
- Der Parameter [gridx](#) gibt an, in welcher Spalte des logischen Gitters der linke Rand des Dialogelements liegen soll. Die erste Spalte hat den Wert 0. [gridy](#) gibt analog die logische Zeile des oberen Rands des Dialogelements an. Beide Werte werden nicht etwa in Pixeln angegeben, sondern bezeichnen die Zellen des logischen Gitternetzes. (0, 0) ist die linke obere Ecke.
- Die Parameter [gridwidth](#) und [gridheight](#) geben die horizontale und vertikale Ausdehnung des Dialogelements an. Soll ein Dialogelement beispielsweise zwei Zellen breit sein, so ist für [gridwidth](#) der Wert 2 anzugeben.
- Mit dem Parameter [fill](#) wird definiert, wie sich die Abmessungen des Dialogelements verändern, wenn die Größe des Fensters verändert wird. Wird hier [GridBagConstraints.NONE](#) angegeben, bleibt das Element immer in seiner ursprünglichen Größe. Wird einer der Werte [GridBagConstraints.HORIZONTAL](#), [GridBagConstraints.VERTICAL](#) oder [GridBagConstraints.BOTH](#) angegeben, skaliert der Layoutmanager das Dialogelement automatisch in horizontaler, vertikaler oder in beiden Richtungen.
- Der Parameter [anchor](#) bestimmt, an welcher Kante der Zelle das Dialogelement festgemacht wird, wenn nach dem Skalieren noch Platz in der Zelle verbleibt. Hier können die folgenden Konstanten angegeben werden:
 - [GridBagConstraints.CENTER](#)
 - [GridBagConstraints.NORTH](#)
 - [GridBagConstraints.NORTHEAST](#)
 - [GridBagConstraints.EAST](#)
 - [GridBagConstraints.SOUTHEAST](#)

- [GridBagConstraints.SOUTH](#)
- [GridBagConstraints.SOUTHWEST](#)
- [GridBagConstraints.WEST](#)
- [GridBagConstraints.NORTHWEST](#)
- Die Parameter [weightx](#) und [weighty](#) sind etwas trickreich. Sie bestimmen das Verhältnis, in dem überschüssiger Platz auf die Zellen einer Zeile bzw. einer Spalte verteilt wird. Ist der Wert 0, bekommt die Zelle nichts von eventuell überschüssigem Platz ab. Bedeutsam ist dabei nur das relative Verhältnis, der absolute Wert dieses Parameters spielt keine Rolle. Die Werte sollten allerdings nicht negativ sein.
- Mit dem Parameter [insets](#) kann ein Rand gesetzt werden. Er wird um das Dialogelement herum auf jeden Fall freigelassen.
- Mit [ipadx](#) und [ipady](#) kann ein konstanter Wert zur minimalen Breite und Höhe der Komponente hinzugefügt werden.

Um die Anwendung der Parameter zu veranschaulichen, wollen wir uns ein Beispiel ansehen. Das zu erstellende Programm soll ein Fenster mit sechs Dialogelementen entsprechend [Abbildung 21.8](#) aufbauen. Die Liste auf der linken Seite soll beim Skalieren in beide Richtungen vergrößert werden. Die Textfelder sollen dagegen lediglich in der Breite wachsen, ihre ursprüngliche Höhe aber beibehalten. Bei vertikaler Vergrößerung des Fensters sollen sie untereinander stehen bleiben. Die Beschriftungen sollen ihre anfängliche Größe beibehalten und - unabhängig von der Fenstergröße - direkt vor den Textfeldern stehen bleiben. Auch der Button soll seine ursprüngliche Größe beibehalten, wenn das Fenster skaliert wird. Zudem soll er immer in der rechten unteren Ecke stehen. Zwischen den Dialogelementen soll ein Standardabstand von zwei Pixeln eingehalten werden.

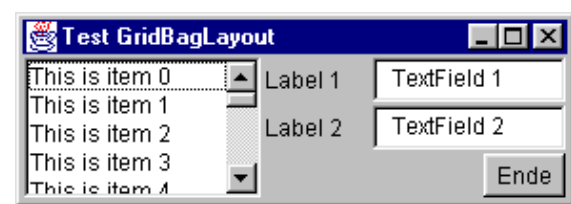


Abbildung 21.8: Beispiel für GridBagLayout

Wir konstruieren das [GridBagLayout](#) als Zelle der Größe drei mal drei gemäß [Abbildung 21.9](#):

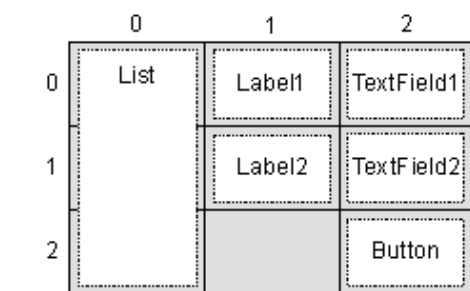


Abbildung 21.9: Zellschema für GridBagLayout-Beispiel

Die Listbox beginnt also in Zelle (0, 0) und erstreckt sich über eine Zelle in horizontaler und drei Zellen in vertikaler Richtung. Die beiden Labels liegen in Spalte 1 und den Zeilen 0 bzw. 1. Die Textfelder liegen eine Spalte rechts daneben. Textfelder und Beschriftungen sind genau eine Zelle breit und eine Zelle hoch. Der Button liegt in Zelle (2, 2). Die übrigen Eigenschaften werden entsprechend unserer Beschreibung so zugewiesen, daß die vorgegebenen Anforderungen erfüllt werden:

[Listing2106.java](#)

```
001 /* Listing2106.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2106
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2106 wnd = new Listing2106();
012         wnd.setVisible(true);
013     }
014
015     public Listing2106()
016     {
017         super("Test GridBagLayout");
```

```

018 setBackground(Color.lightGray);
019 //WindowListener hinzufügen
020 addWindowListener(
021     new WindowAdapter() {
022         public void windowClosing(WindowEvent event)
023         {
024             setVisible(false);
025             dispose();
026             System.exit(0);
027         }
028     }
029 );
030 //Layout setzen und Komponenten hinzufügen
031 GridBagLayout gbl = new GridBagLayout();
032 GridBagConstraints gbc;
033 setLayout(gbl);
034
035 //List hinzufügen
036 List list = new List();
037 for (int i = 0; i < 20; ++i) {
038     list.add("This is item " + i);
039 }
040 gbc = makegbc(0, 0, 1, 3);
041 gbc.weightx = 100;
042 gbc.weighty = 100;
043 gbc.fill = GridBagConstraints.BOTH;
044 gbl.setConstraints(list, gbc);
045 add(list);
046 //Zwei Labels und zwei Textfelder
047 for (int i = 0; i < 2; ++i) {
048     //Label
049     gbc = makegbc(1, i, 1, 1);
050     gbc.fill = GridBagConstraints.NONE;
051     Label label = new Label("Label " + (i + 1));
052     gbl.setConstraints(label, gbc);
053     add(label);
054     //Textfeld
055     gbc = makegbc(2, i, 1, 1);
056     gbc.weightx = 100;
057     gbc.fill = GridBagConstraints.HORIZONTAL;
058     TextField field = new TextField("TextField " + (i + 1));
059     gbl.setConstraints(field, gbc);
060     add(field);
061 }
062 //Ende-Button
063 Button button = new Button("Ende");
064 gbc = makegbc(2, 2, 0, 0);
065 gbc.fill = GridBagConstraints.NONE;
066 gbc.anchor = GridBagConstraints.SOUTHEAST;
067 gbl.setConstraints(button, gbc);
068 add(button);
069 //Dialogelemente layouten
070 pack();
071 }
072
073 private GridBagConstraints makegbc(
074     int x, int y, int width, int height)
075 {
076     GridBagConstraints gbc = new GridBagConstraints();
077     gbc.gridx = x;
078     gbc.gridy = y;
079     gbc.gridwidth = width;
080     gbc.gridheight = height;
081     gbc.insets = new Insets(1, 1, 1, 1);
082     return gbc;

```

```
083 }
084 }
```

Listing 21.6: Beispiel für GridBagLayout

Das Programm besitzt eine kleine Hilfsmethode `makegbc`, mit der auf einfache Weise ein neues `GridBagConstraints`-Objekt für einen vorgegebenen Bereich von Zellen erzeugt werden kann. Die übrigen Membervariablen werden im Konstruktor der Fensterklasse zugewiesen. [Abbildung 21.10](#) zeigt das Fenster nach dem Skalieren in x- und y-Richtung.

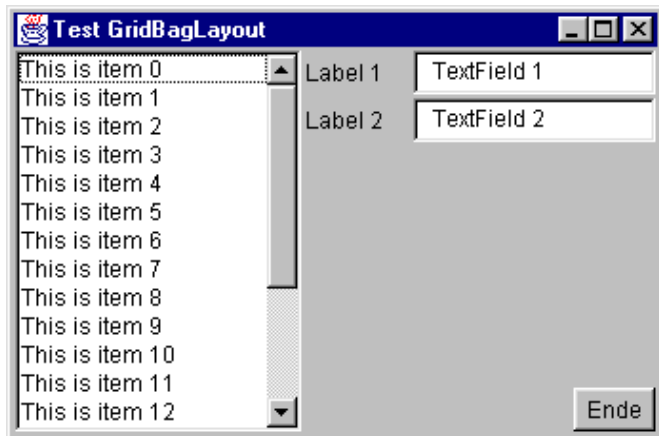


Abbildung 21.10: Das GridBagLayout-Beispiel nach dem Skalieren

21.2.5 NULL-Layout

Ein *Null-Layout* wird erzeugt, indem die Methode `setLayout` mit dem Argument `null` aufgerufen wird. In diesem Fall verwendet das Fenster keinen Layoutmanager, sondern überläßt die Positionierung der Komponenten der Anwendung. Für jedes einzufügende Dialogelement sind dann drei Schritte auszuführen:

- Das Dialogelement wird erzeugt.
- Seine Größe und Position werden festgelegt.
- Das Dialogelement wird an das Fenster übergeben.

Der erste und letzte Schritt unterscheiden sich nicht von unseren bisherigen Versuchen, bei denen wir vordefinierte Layoutmanager verwendet haben. Im zweiten Schritt ist allerdings etwas Handarbeit notwendig. Größe und Position des Dialogelements können mit den Methoden `move` und `setSize` oder - noch einfacher - mit `setBounds` festgelegt werden:

[java.awt.Component](#)

```
public void setBounds(
    int x, int y, int width, int height
)
```

Der Punkt `(x,y)` bestimmt die neue Anfangsposition des Dialogelements und `(width, height)` seine Größe.

Das folgende Listing demonstriert diese Vorgehensweise am Beispiel einer treppenartigen Anordnung von fünf Buttons:

Beispiel

[Listing2107.java](#)

```
001 /* Listing2107.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2107
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2107 wnd = new Listing2107();
012         wnd.setVisible(true);
013     }
014
015     public Listing2107()
016     {
017         super("Dialogelemente ohne Layoutmanager");
018         //WindowListener hinzufügen
```

```

019     addWindowListener(
020         new WindowAdapter() {
021             public void windowClosing(WindowEvent event)
022             {
023                 setVisible(false);
024                 dispose();
025                 System.exit(0);
026             }
027         }
028     );
029     //Layout setzen und Komponenten hinzufügen
030     setSize(300,250);
031     setLayout(null);
032     for (int i = 0; i < 5; ++i) {
033         Button button = new Button("Button"+(i+1));
034         button.setBounds(10+i*35,40+i*35,100,30);
035         add(button);
036     }
037
038 }
039 }

```

Listing 21.7: Anordnen von Dialogelementen ohne Layoutmanager

Die Ausgabe des Programms ist:

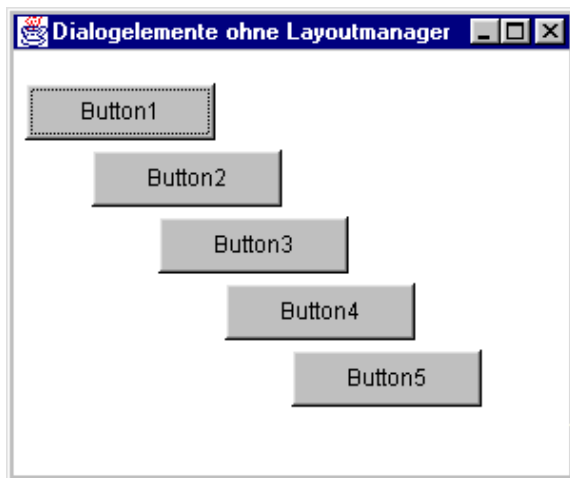


Abbildung 21.11: Verwendung des Null-Layouts

21.2.6 Schachteln von Layoutmanagern

Eines der Schlüsselkonzepte zur Realisierung komplexer, portabler Dialoge ist die Fähigkeit, Layoutmanager schachteln zu können. Dazu wird an der Stelle, die ein Sublayout erhalten soll, einfach ein Objekt der Klasse [Panel](#) eingefügt, das einen eigenen Layoutmanager erhält. Dieses [Panel](#) kann mit Dialogelementen bestückt werden, die entsprechend dem zugeordneten Unterlayout formatiert werden.

Wenn wir an die in [Abbildung 17.1](#) vorgestellte Klassenhierarchie innerhalb des AWT denken, werden wir uns daran erinnern, daß ein [Panel](#) die einfachste konkrete Containerklasse ist. Sie erbt alle Eigenschaften von [Container](#) und bietet damit die Möglichkeit, Komponenten aufzunehmen und mit Hilfe eines Layoutmanagers auf dem Bildschirm anzuordnen.

Das folgende Beispiel zeigt einen Dialog, der ein [GridLayout](#) der Größe 1*2 Elemente verwendet. Innerhalb des ersten Elements wird ein [Panel](#) mit einem [GridLayout](#) der Größe 3*1 Elemente verwendet, innerhalb des zweiten Elements ein [BorderLayout](#):

Beispiel

```

001 /* Listing2108.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2108
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2108 wnd = new Listing2108();
012         wnd.setVisible(true);
013     }
014
015     public Listing2108()
016     {
017         super("Geschachtelte Layoutmanager");
018         //WindowListener hinzufügen
019         addWindowListener(
020             new WindowAdapter() {
021                 public void windowClosing(WindowEvent event)
022                 {
023                     setVisible(false);
024                     dispose();
025                     System.exit(0);
026                 }
027             }
028         );
029         //Layout setzen und Komponenten hinzufügen
030         //Panel 1
031         Panel panel1 = new Panel();
032         panel1.setLayout(new GridLayout(3,1));
033         panel1.add(new Button("Button1"));
034         panel1.add(new Button("Button2"));
035         panel1.add(new Button("Button3"));
036         //Panel 2
037         Panel panel2 = new Panel();
038         panel2.setLayout(new BorderLayout());
039         panel2.add("North", new Button("Button4"));
040         panel2.add("South", new Button("Button5"));
041         panel2.add("West", new Button("Button6"));
042         panel2.add("East", new Button("Button7"));
043         panel2.add("Center", new Button("Button8"));
044         //Hauptfenster
045         setLayout(new GridLayout(1,2));
046         add(panel1);
047         add(panel2);
048         pack();
049     }
050 }

```

Listing 21.8: Schachteln von Layoutmanagern

Die Ausgabe des Programms ist:

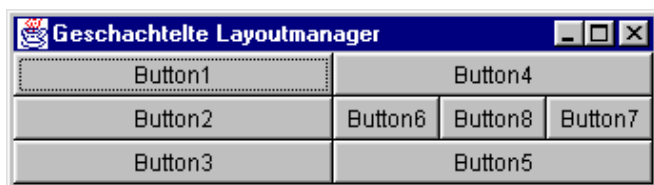


Abbildung 21.12: Verwendung eines geschachtelten Layouts

Geschachtelte Layouts können auch dann sinnvoll verwendet werden, wenn innerhalb der Bestandteile eines `BorderLayouts` mehrere Komponenten untergebracht werden sollen oder wenn die Komponenten eines *skalierenden* Layoutmanagers ihre ursprüngliche Größe beibehalten sollen.

Tip

Das folgende Beispiel demonstriert beide Anwendungen anhand eines `BorderLayouts`, das im South-Element zwei rechtsbündig angeordnete Buttons in Originalgröße und im Center-Element sechs innerhalb eines `GridLayouts` der Größe 3*2 Elemente angeordnete, skalierte Buttons enthält:

Beispiel

[Listing2109.java](#)

```
001 /* Listing2109.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2109
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2109 wnd = new Listing2109();
012         wnd.setVisible(true);
013     }
014
015     public Listing2109()
016     {
017         super("Geschachtelte Layoutmanager, Teil II");
018         //WindowListener hinzufügen
019         addWindowListener(
020             new WindowAdapter() {
021                 public void windowClosing(WindowEvent event)
022                 {
023                     setVisible(false);
024                     dispose();
025                     System.exit(0);
026                 }
027             }
028         );
029         //Layout setzen und Komponenten hinzufügen
030         setSize(300,200);
031         //Panel 1
032         Panel panel1 = new Panel();
033         panel1.setLayout(new GridLayout(3,2));
034         panel1.add(new Button("Button1"));
035         panel1.add(new Button("Button2"));
036         panel1.add(new Button("Button3"));
037         panel1.add(new Button("Button4"));
038         panel1.add(new Button("Button5"));
039         panel1.add(new Button("Button6"));
040         //Panel 2
041         Panel panel2 = new Panel();
042         panel2.setLayout(new FlowLayout(FlowLayout.RIGHT));
043         panel2.add(new Button("Abbruch"));
044         panel2.add(new Button("OK"));
045         //Hauptfenster
046         setLayout(new BorderLayout());
047         add("Center",panel1);
048         add("South", panel2);
049     }
050 }
```

Listing 21.9: Eine weitere Anwendung für geschachtelte Layoutmanager

Die Ausgabe des Programms ist:

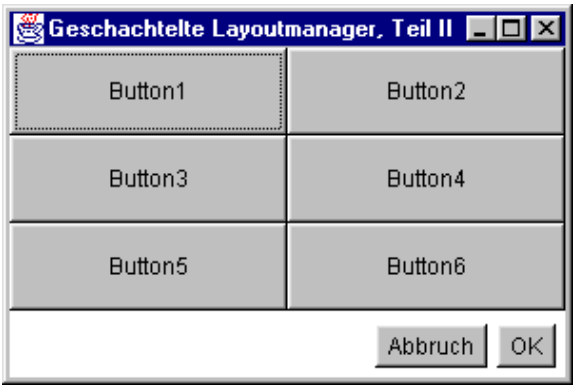


Abbildung 21.13: Ein weiteres Beispiel für geschachtelte Layouts

21.3 Modale Dialoge

- [21.3 Modale Dialoge](#)

Modale Dialoge sind solche, die alle Benutzereingaben des Programmes beanspruchen und andere Fenster erst dann wieder zum Zuge kommen lassen, wenn das Dialogfenster geschlossen wird. Eine wichtige Eigenschaft modaler Dialoge ist es, daß im Programm der Aufruf zur Anzeige des Dialogs so lange blockiert, bis der Dialog beendet ist. Auf diese Weise kann an einer bestimmten Stelle im Programm auf eine Eingabe gewartet werden und erst dann mit der Bearbeitung fortgefahren werden, wenn die Eingabe erfolgt ist.

Im AWT des JDK 1.0 gab es einen schwerwiegenden Fehler, durch den das Erzeugen *modaler* Dialoge unmöglich gemacht wurde. Der Fehler hielt sich bis zur Version 1.0.2 und wurde erst mit Erscheinen des JDK 1.1 behoben. Da viele Anwenderdialoge von Natur aus modalen Charakter haben, wurde dieser Fehler von vielen als schwerwiegend angesehen, und die Java-Gemeinde entwickelte eine Reihe von Workarounds, die aber allesamt nicht voll zufriedenstellen konnten. Glücklicherweise wurden die Probleme mit dem JDK 1.1 behoben, und wir wollen in diesem Kapitel aufzeigen, wie modale Dialoge in Java erzeugt werden können.

Ein modaler Dialog muß immer aus der Klasse [Dialog](#) abgeleitet werden. Nur sie bietet die Möglichkeit, an den Konstruktor einen booleschen Wert zu übergeben, der festlegt, daß die übrigen Fenster der Anwendung während der Anzeige des Dialogs suspendiert werden. [Dialog](#) besitzt folgende Konstruktoren:

```

public Dialog(Frame owner)
public Dialog(Frame owner, boolean modal)
public Dialog(Frame owner, String title)
public Dialog(Frame owner, String title, boolean modal)
public Dialog(Dialog owner)
public Dialog(Dialog owner, String title)
public Dialog(Dialog owner, String title, boolean modal)

```

Als erstes Argument muß in jedem Fall ein [Frame](#)- oder [Dialog](#)-Objekt als Vaterfenster übergeben werden (bis zur Version 1.1 waren nur [Frame](#)-Objekte erlaubt). Mit `title` kann der Inhalt der Titelzeile vorgegeben werden, und der Parameter `modal` entscheidet, ob der Dialog modal dargestellt wird oder nicht.

[Dialog](#) bietet die Methoden [isModal](#) und [setModal](#), mit denen auf die Modalität des Dialogs zugegriffen werden kann:

```

public boolean isModal()
public void setModal(boolean b)

```

Der Rückgabewert von [isModal](#) ist `true`, falls der Dialog modal ist. Andernfalls ist er `false`.

Die Methode [setModal](#), deren Fähigkeit darin besteht, einen bestehenden Dialog zwischen den Zuständen modal und nicht-modal umzuschalten, ist mit Vorsicht zu genießen. Unter Windows 95 hat ein Aufruf im JDK 1.1 mitunter dazu geführt, daß beim Schließen des Fensters die Nachrichtenschleife des Aufrufers deaktiviert blieb und das Programm sich aufhängte. Am besten, man entscheidet schon im Konstruktor, ob der Dialog modal sein soll oder nicht.

Warnung

Ein zusätzliches Feature der Klasse [Dialog](#) besteht darin, die Veränderbarkeit der Größe eines Fensters durch den Anwender zu unterbinden:

```

public void setResizable(boolean resizable)
public boolean isResizable()

```

Nach einem Aufruf von [setResizable](#) mit `false` als Argument kann die Größe des Fensters nicht mehr vom Anwender verändert werden. Nach einem Aufruf von [setResizable\(true\)](#) ist dies wieder möglich. Mit [isResizable](#) kann der aktuelle Zustand dieses Schalters abgefragt werden.

Auch die Methode `setResizable` ist nicht ganz unkritisch, denn sie hat auf einigen UNIX-Systemen zu Problemen geführt. Sowohl unter SUN Solaris als auch unter LINUX kam es im JDK 1.1 zu Hängern bei Programmen, die diese Methode benutzten. Falls sich die nachfolgenden Beispielprogramme auf Ihrem System nicht so verhalten wie angegeben (typischerweise wird der Dialog gar nicht erst angezeigt), sollten Sie versuchsweise den Aufruf von `setResizable` auskommentieren.

Warnung

Das folgende Beispiel demonstriert die Anwendung der Klasse `Dialog`. Es zeigt einen `Frame`, aus dem per Buttonklick ein modaler Dialog aufgerufen werden kann. Der Dialog fragt den Anwender, ob die Anwendung beendet werden soll und wartet, bis einer der Buttons »Ja« oder »Nein« gedrückt wurde. In diesem Fall wird ein boolescher Rückgabewert generiert, der nach dem Schließen des Dialogs an das Vaterfenster zurückgegeben wird. Falls der Rückgabewert `true` (entsprechend dem Button »Ja«) ist, wird die Anwendung beendet, andernfalls läuft sie weiter:

Beispiel

[Listing2110.java](#)

```
001 /* Listing2110.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class YesNoDialog
007 extends Dialog
008 implements ActionListener
009 {
010     boolean result;
011
012     public YesNoDialog(Frame owner, String msg)
013     {
014         super(owner, "Ja-/Nein-Auswahl", true);
015         //Fenster
016         setBackground(Color.lightGray);
017         setLayout(new BorderLayout());
018         setResizable(false); //Hinweis im Text beachten
019         Point parloc = owner.getLocation();
020         setLocation(parloc.x + 30, parloc.y + 30);
021         //Message
022         add("Center", new Label(msg));
023         //Buttons
024         Panel panel = new Panel();
025         panel.setLayout(new FlowLayout(FlowLayout.CENTER));
026         Button button = new Button("Ja");
027         button.addActionListener(this);
028         panel.add(button);
029         button = new Button("Nein");
030         button.addActionListener(this);
031         panel.add(button);
032         add("South", panel);
033         pack();
034     }
035
036     public void actionPerformed(ActionEvent event)
037     {
038         result = event.getActionCommand().equals("Ja");
039         setVisible(false);
040         dispose();
041     }
042
043     public boolean getResult()
044     {
045         return result;
046     }
047 }
048
049 public class Listing2110
050 extends Frame
051 implements ActionListener
```

```

052 {
053     public static void main(String[] args)
054     {
055         Listing2110 wnd = new Listing2110();
056         wnd.setVisible(true);
057     }
058
059     public Listing2110()
060     {
061         super("Modale Dialoge");
062         setLayout(new FlowLayout());
063         setBackground(Color.lightGray);
064         Button button = new Button("Ende");
065         button.addActionListener(this);
066         add(button);
067         setLocation(100,100);
068         setSize(300,200);
069         setVisible(true);
070     }
071
072     public void actionPerformed(ActionEvent event)
073     {
074         String cmd = event.getActionCommand();
075         if (cmd.equals("Ende")) {
076             YesNoDialog dlg;
077             dlg = new YesNoDialog(
078                 this,
079                 "Wollen Sie das Programm wirklich beenden?"
080             );
081             dlg.setVisible(true);
082             //Auf das Schließen des Dialogs warten...
083             if (dlg.getResult()) {
084                 setVisible(false);
085                 dispose();
086                 System.exit(0);
087             }
088         }
089     }
090 }

```

Listing 21.10: Konstruktion modaler Dialoge

Um den Dialog relativ zur Position seines Vaterfensters anzuzeigen, wird dessen Position durch Aufruf von `getLocation` ermittelt. Der Ursprung des Dialogfensters wird dann 30 Pixel weiter nach unten bzw. nach rechts gelegt.

Hinweis

Nach dem Start zeigt das Programm zunächst das in [Abbildung 21.14](#) dargestellte Fenster an:

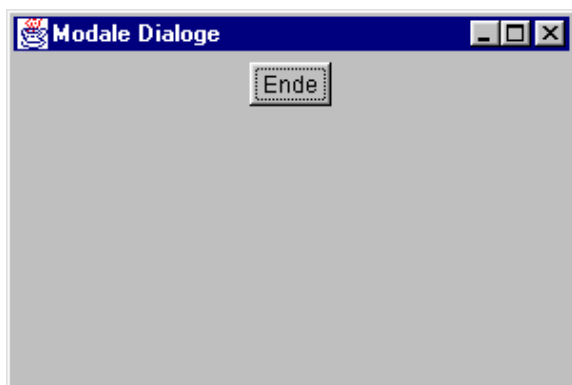


Abbildung 21.14: Das Vaterfenster für den modalen Dialog

Nach dem Klick auf »Ende« wird `actionPerformed` aufgerufen und läuft bis zum Aufruf des Dialogs (siehe [Abbildung 21.15](#)). Die Anweisung `dlg.setVisible(true);` ruft den Dialog auf und blockiert dann. Sie terminiert erst, wenn der Dialog durch Drücken eines Buttons beendet wurde. In diesem Fall wird mit `getResult` der Rückgabewert abgefragt und das Programm beendet, wenn dieser `true` ist. Die Methode `getResult` liefert den Inhalt der privaten Instanzvariablen `result`, die beim Auftreten eines Action-Events durch einen der beiden Buttons

gesetzt wird. Zwar ist das Dialogfenster beim Aufruf von `getResult` bereits zerstört, aber die Objektvariable `dlg` existiert noch und `getResult` kann aufgerufen werden.

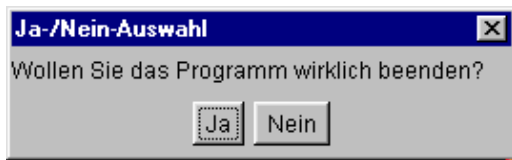


Abbildung 21.15: Ein einfacher Ja-/Nein-Dialog

Bevor wir das Kapitel beenden, wollen wir dieses Programm ein wenig erweitern und ein weiteres Beispiel für die Anwendung modaler Dialoge geben. Das folgende Programm implementiert eine Klasse `ModalDialog`, die aus `Dialog` abgeleitet ist. Mit ihrer Hilfe können modale Dialoge der obigen Art konstruiert werden, bei denen die Ausstattung mit Buttons variabel ist und zum Zeitpunkt der Instanzierung festgelegt werden kann:

Beispiel

```
public ModalDialog(
    Frame owner,
    String title,
    String msg,
    String buttons
);
```

Der Konstruktor erwartet neben dem Vater-`Frame` drei weitere Parameter, `title`, `msg` und `buttons`. In `title` wird der Inhalt der Titelzeile übergeben, und `msg` spezifiziert den Text innerhalb des Dialogs. Der Parameter `buttons` erwartet eine Liste von `Button`-Bezeichnern, die zur Festlegung der Anzahl und Beschriftung der Buttons dienen. Die einzelnen Elemente sind durch Kommata zu trennen und werden innerhalb des Dialogs mit einem `StringTokenizer` zerlegt. Der in `getResult` gelieferte Rückgabewert des Dialogs ist - anders als im vorigen Beispiel - ein `String` und entspricht der Beschriftung des zum Schließen verwendeten Buttons.

`ModalDialog` enthält einige statische Methoden, mit denen Dialoge mit einer festen Buttonstruktur einfach erzeugt werden können. So erzeugt `OKButton` lediglich einen Button mit der Beschriftung »OK«, `YesNoDialog` zwei Buttons, »Ja« und »Nein«, und `YesNoCancelDialog` erzeugt einen Dialog mit drei Buttons, »Ja«, »Nein« und »Abbruch«. Das folgende Listing zeigt die Klasse `ModalDialog` und ein Rahmenprogramm zum Testen:

Tip

[Listing2111.java](#)

```
001 /* Listing2111.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005 import java.util.*;
006
007 class ModalDialog
008 extends Dialog
009 implements ActionListener
010 {
011     String result;
012
013     public static String OKDialog(Frame owner, String msg)
014     {
015         ModalDialog dlg;
016         dlg = new ModalDialog(owner, "Nachricht", msg, "OK");
017         dlg.setVisible(true);
018         return dlg.getResult();
019     }
020
021     public static String YesNoDialog(Frame owner, String msg)
022     {
023         ModalDialog dlg;
024         dlg = new ModalDialog(owner, "Frage", msg, "Ja,Nein");
025         dlg.setVisible(true);
026         return dlg.getResult();
027     }
028
029     public static String YesNoCancelDialog(Frame owner, String msg)
030     {
031         ModalDialog dlg;
```

```

032     dlg = new ModalDialog(owner, "Frage", msg, "Ja, Nein, Abbruch");
033     dlg.setVisible(true);
034     return dlg.getResult();
035 }
036
037 public ModalDialog(
038     Frame owner,
039     String title,
040     String msg,
041     String buttons
042 )
043 {
044     super(owner, title, true);
045     //Fenster
046     setBackground(Color.lightGray);
047     setLayout(new BorderLayout());
048     setResizable(false);
049     Point parloc = owner.getLocation();
050     setLocation(parloc.x + 30, parloc.y + 30);
051     //Message
052     add("Center", new Label(msg));
053     //Buttons
054     Panel panel = new Panel();
055     panel.setLayout(new FlowLayout(FlowLayout.CENTER));
056     StringTokenizer strtok = new StringTokenizer(buttons, ",");
057     while (strtok.hasMoreTokens()) {
058         Button button = new Button(strtok.nextToken());
059         button.addActionListener(this);
060         panel.add(button);
061     }
062     add("South", panel);
063     pack();
064 }
065
066 public void actionPerformed(ActionEvent event)
067 {
068     result = event.getActionCommand();
069     setVisible(false);
070     dispose();
071 }
072
073 public String getResult()
074 {
075     return result;
076 }
077 }
078
079 public class Listing2111
080 extends Frame
081 implements ActionListener
082 {
083     public static void main(String[] args)
084     {
085         Listing2111 wnd = new Listing2111();
086         wnd.setVisible(true);
087     }
088
089     public Listing2111()
090     {
091         super("Drei modale Standarddialoge");
092         setLayout(new FlowLayout());
093         setBackground(Color.lightGray);
094         Button button = new Button("OKDialog");
095         button.addActionListener(this);
096         add(button);

```

```

097     button = new Button("YesNoDialog");
098     button.addActionListener(this);
099     add(button);
100     button = new Button("YesNoCancelDialog");
101     button.addActionListener(this);
102     add(button);
103     setLocation(100,100);
104     setSize(400,200);
105     setVisible(true);
106 }
107
108 public void actionPerformed(ActionEvent event)
109 {
110     String cmd = event.getActionCommand();
111     if (cmd.equals("OKDialog")) {
112         ModalDialog.OKDialog(this,"Dream, dream, dream, ...");
113     } else if (cmd.equals("YesNoDialog")) {
114         String ret = ModalDialog.YesNoDialog(
115             this,
116             "Programm beenden?"
117         );
118         if (ret.equals("Ja")) {
119             setVisible(false);
120             dispose();
121             System.exit(0);
122         }
123     } else if (cmd.equals("YesNoCancelDialog")) {
124         String msg = "Verzeichnis erstellen?";
125         String ret = ModalDialog.YesNoCancelDialog(this,msg);
126         ModalDialog.OKDialog(this,"Rückgabe: " + ret);
127     }
128 }
129 }

```

Listing 21.11: Drei modale Standarddialoge

Die drei Dialoge, die durch Aufrufe von `OKDialog`, `YesNoDialog` und `YesNoCancelDialog` generiert werden, sind in den folgenden Abbildungen [21.16](#) bis [21.18](#) dargestellt:

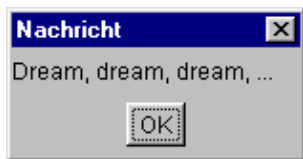


Abbildung 21.16: Ein Aufruf von `OKDialog`

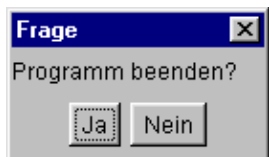


Abbildung 21.17: Ein Aufruf von `YesNoDialog`

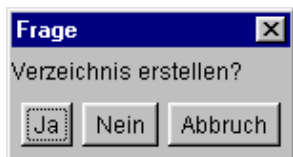


Abbildung 21.18: Ein Aufruf von `YesNoCancelDialog`

21.4 Zusammenfassung

- [21.4 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Verwendung der Klassen [Frame](#) und [Dialog](#) zum Erstellen eines Dialogfensters
- Aufruf von [setLayout](#), um einer Fensterklasse einen Layout-Manager zuzuordnen
- Einfügen einer Dialogkomponente durch Aufruf von [add](#)
- Funktionsweise und Verwendung der Layout-Manager [FlowLayout](#), [GridLayout](#) und [BorderLayout](#)
- Verwendung eines Null-Layouts zur absoluten Positionierung von Dialogelementen
- Strategien zur Berechnung der Komponentengröße bei den verschiedenen Layout-Managern
- Verwendung der Klasse [Panel](#) zum Schachteln von Layouts innerhalb eines Dialogs
- Erzeugen modaler Dialoge und die Rückgabe von Ergebniswerten an den Aufrufer

Kapitel 22

Vordefinierte Dialogelemente

- [22 Vordefinierte Dialogelemente](#)
 - [22.1 Rahmenprogramm](#)
 - [22.2 Label](#)
 - [22.3 Button](#)
 - [22.4 Checkbox](#)
 - [22.5 CheckboxGroup](#)
 - [22.6 TextField](#)
 - [22.7 TextArea](#)
 - [22.8 Choice](#)
 - [22.9 List](#)
 - [22.10 Scrollbar](#)
 - [22.11 ScrollPane](#)
 - [22.12 Zusammenfassung](#)

22.1 Rahmenprogramm

- [22.1 Rahmenprogramm](#)

Nachdem in [Kapitel 21](#) die grundsätzliche Vorgehensweise beim Anlegen eines Dialogs besprochen wurde, behandelt dieses Kapitel nun die verschiedenen Dialogelemente. Zu jedem Dialogelement werden die Konstruktoren und die verschiedenen Methoden zur Steuerung seines Verhaltens vorgestellt. Falls ein Dialogelement auch Nachrichten sendet, werden diese erläutert und ihre Anwendung demonstriert.

Jedes Dialogelement wird in Java durch eine eigene Klasse repräsentiert, die dessen Verhalten und Eigenschaften kapselt. Zur Aufnahme eines neuen Dialogelements in einen Dialog wird eine neue Instanz der gewünschten Klasse angelegt und das resultierende Objekt mit [add](#) in den Dialog eingefügt. Die Methoden des Dialogelements können dann zu einem beliebigen Zeitpunkt aufgerufen werden, um dessen Verhalten zu beeinflussen. Alle Dialogelemente sind aus der Klasse [Component](#) abgeleitet. Sie verfügen über die grundlegenden Eigenschaften eines Fensters, besitzen eine Größe und Position und sind in der Lage, Nachrichten zu empfangen und zu bearbeiten.

Damit nicht jedesmal ein komplettes Programm abgedruckt werden muß, wollen wir die Beispiele in ein vordefiniertes Rahmen-Programm einbetten. Dieses erzeugt ein Fenster mit zwei Buttons, die zum Aufrufen des Dialogs bzw. zum Beenden des Programms verwendet werden können:

Hinweis



Abbildung 22.1: Das Beispielprogramm zum Aufruf der Beispieldialoge

Der nach Drücken des »Dialog«-Buttons aufgerufene Dialog besitzt ein [BorderLayout](#), das in der South-Komponente einen [Button](#) zum Beenden des Dialogs enthält. In der Center-Komponente wird ein [Panel](#) platziert, das an die Methode [customizeLayout](#) weitergegeben wird, die darin die Beispielkomponenten platziert. In der Basisversion ist diese Methode leer, und der Dialog hat folgendes Aussehen:



Abbildung 22.2: Der noch leere Beispieldialog

Die Beispielprogramme können dann dem übergebenen Panel innerhalb von [customizeLayout](#) einen beliebigen Layoutmanager zuordnen und eine beliebige Auswahl von Komponenten darauf plazieren. Der Ende-Button bleibt auf jeden Fall erhalten und kann zum Beenden des Dialogs verwendet werden:

[DialogBeispiel.java](#)

```

001 /* DialogBeispiel.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class MyDialog
007 extends Dialog
008 implements ActionListener
009 {
010     public MyDialog(Frame parent)
011     {
012         super(parent,"MyDialog",true);
013         Point parloc = parent.getLocation();

```

```

014     setBounds(parloc.x + 30, parloc.y + 30,400,300);
015     setBackground(Color.lightGray);
016     setLayout(new BorderLayout());
017     //Panel
018     Panel panel = new Panel();
019     customizeLayout(panel);
020     add("Center",panel);
021     //Ende-Button
022     Button button = new Button("Ende");
023     button.addActionListener(this);
024     add("South", button);
025     //Window-Listener
026     addWindowListener(
027         new WindowAdapter() {
028             public void windowClosing(WindowEvent event)
029             {
030                 endDialog();
031             }
032         }
033     );
034     pack();
035 }
036
037 private void customizeLayout(Panel panel)
038 {
039     //Beispielcode hier
040 }
041
042 public void actionPerformed(ActionEvent event)
043 {
044     if (event.getActionCommand().equals("Ende")) {
045         endDialog();
046     }
047 }
048
049 void endDialog()
050 {
051     setVisible(false);
052     dispose();
053     ((Window)getParent()).toFront();
054     getParent().requestFocus();
055 }
056 }
057
058 public class DialogBeispiel
059 extends Frame
060 implements ActionListener
061 {
062     public static void main(String[] args)
063     {
064         DialogBeispiel wnd = new DialogBeispiel();
065         wnd.setSize(300,200);
066         wnd.setVisible(true);
067     }
068
069     public DialogBeispiel()
070     {
071         super("DialogBeispiel");
072         setBackground(Color.lightGray);
073         setLayout(new FlowLayout());
074         //Dialog-Button
075         Button button = new Button("Dialog");
076         button.addActionListener(this);
077         add(button);
078         //Ende-Button

```

```

079     button = new Button("Ende");
080     button.addActionListener(this);
081     add(button);
082     //Window-Listener
083     addWindowListener(
084         new WindowAdapter() {
085             public void windowClosing(WindowEvent event)
086             {
087                 setVisible(false);
088                 dispose();
089                 System.exit(0);
090             }
091         }
092     );
093 }
094
095 public void actionPerformed(ActionEvent event)
096 {
097     String cmd = event.getActionCommand();
098     if (cmd.equals("Dialog")) {
099         MyDialog dlg = new MyDialog(this);
100         dlg.setVisible(true);
101     } else if (cmd.equals("Ende")) {
102         setVisible(false);
103         dispose();
104         System.exit(0);
105     }
106 }
107 }

```

Listing 22.1: Rahmenprogramm für Dialogelemente

22.2 Label

- 22.2 Label

Ein [Label](#) dient zur Beschriftung von Dialogboxen. Es enthält eine Zeile Text, die auf dem Bildschirm angezeigt wird und vom Programm verändert werden kann.

```

public Label()

public Label(String text)

public Label(String text, int align)
```

Der parameterlose Konstruktor erzeugt ein [Label](#) ohne Text. Wird der Parameter [text](#) übergeben, verwendet das Label diesen als Beschriftung. Der Parameter [align](#) bestimmt die Ausrichtung des Texts. Hier kann eine der Konstanten [Label.LEFT](#), [Label.RIGHT](#) oder [Label.CENTER](#) übergeben werden.

```

public void setText(String text)

public String getText()

public void setAlignment(int align)

public int getAlignment()
```

[setText](#) und [getText](#) erlauben den Zugriff auf den Text des Labels und [setAlignment](#) und [getAlignment](#) auf die Ausrichtung des Texts. Die Parameter dieser Methoden haben dieselbe Bedeutung wie im Konstruktor.

```

001 /* Label.inc */
002
003 private void customizeLayout(Panel panel)
004 {
005     panel.setLayout(new GridLayout(4,1));
006     panel.add(new Label("Default"));
007     panel.add(new Label("Links",Label.LEFT));
008     panel.add(new Label("Zentriert",Label.CENTER));
009     panel.add(new Label("Rechts",Label.RIGHT));
010 }
```

Listing 22.2: Verwendung von Label-Komponenten



Abbildung 22.3: Ein Dialog mit Label-Komponenten

22.3 Button

- [22.3 Button](#)

Ein [Button](#) ist eine beschriftete Schaltfläche, die dazu verwendet wird, auf Knopfdruck des Anwenders Aktionen in der Fensterklasse auszulösen.

```
public Button()  
  
public Button(String label)
```

Der parameterlose Konstruktor erzeugt einen [Button](#) ohne Text. Üblicherweise wird ein [Button](#) gleich bei der Konstruktion beschriftet, was durch Übergabe eines [String](#)-Objekts erreicht werden kann.

```
public void setLabel(String label)  
  
public String getLabel()
```

[setLabel](#) und [getLabel](#) erlauben den Zugriff auf die Beschriftung des Buttons.

Wird ein [Button](#) gedrückt, so sendet er ein Action-Event an seine Ereignisempfänger. Diese müssen das Interface [ActionListener](#) implementieren und sich durch Aufruf von [addActionListener](#) registrieren:

```
public void addActionListener(ActionListener l)  
  
public void removeActionListener(ActionListener l)
```

Das Action-Event führt im Ereignisempfänger zum Aufruf der Methode [actionPerformed](#), die ein [ActionEvent](#) übergeben bekommt. Dieses besitzt die Methode [getActionCommand](#), mit der die Beschriftung des Buttons abgefragt werden kann. Soll das Action-Kommando nicht mit der Beschriftung identisch sein, kann es in der Buttonklasse durch Aufruf von [setActionCommand](#) geändert werden:

```
public void setActionCommand(String command)
```

In [Kapitel 21](#) findet sich eine ganze Reihe von Beispielen für die Anwendung von Buttons. Wir wollen daher an dieser Stelle auf ein weiteres Beispiel verzichten.

Hinweis

22.4 Checkbox

- 22.4 Checkbox

Eine [Checkbox](#) ist ein Eingabeelement, das eine Zustandsvariable besitzt, die zwischen den Werten [true](#) und [false](#) umgeschaltet werden kann. Der aktuelle Zustand wird durch ein Häkchen oder Kreuz in der [Checkbox](#) angezeigt.

```

public Checkbox()
public Checkbox(String label)
public Checkbox(String label, boolean state)
public Checkbox(String label, CheckboxGroup cbg, boolean state)
public Checkbox(String label, boolean state, CheckboxGroup cbg)

```

Der parameterlose Konstruktor erzeugt eine [Checkbox](#) ohne Beschriftung mit dem Anfangszustand [false](#). Wird der String [label](#) angegeben, bekommt die [Checkbox](#) die angegebene Beschriftung. Durch Übergabe des booleschen Parameters [state](#) ist es möglich, den anfänglichen Zustand der [Checkbox](#) vorzugeben. Der in den letzten beiden Konstruktoren verwendete Parameter [cbg](#) dient zur Gruppierung von *Radiobuttons* und sollte bei einer [Checkbox](#) stets [null](#) sein.

```

public String getLabel()
public void setLabel(String label)
public boolean getState()
public void setState(boolean state)

```

Die Methoden ermöglichen den Zugriff auf die Beschriftung und den aktuellen Zustand der [Checkbox](#).

Oft repräsentiert eine [Checkbox](#) einen logischen Schalter, dessen Zustand abgefragt wird, nachdem der Dialog beendet wurde. Zusätzlich kann aber auch unmittelbar auf eine Zustandsänderung reagiert werden, denn eine [Checkbox](#) generiert ein Item-Event, wenn die Markierung vom Anwender gesetzt oder zurückgenommen wird. Um auf dieses Event zu reagieren, ist durch Aufruf von [addItemListener](#) ein Objekt, das das Interface [ItemListener](#) implementiert, bei der [Checkbox](#) zu registrieren:

```

public void addItemListener(ItemListener l)

```

Ein Item-Event wird immer dann gesendet, wenn sich der Zustand einer Checkbox verändert hat. In diesem Fall wird im Ereignisempfänger die Methode [itemStateChanged](#) mit einem Argument vom Typ [ItemEvent](#) aufgerufen:

```

public abstract void itemStateChanged(ItemEvent e)

```

Das [ItemEvent](#) stellt die Methode [getItemSelectable](#) zur Verfügung, mit der ermittelt werden kann, durch welche [Checkbox](#) das Ereignis ausgelöst wurde:

```

public ItemSelectable getItemSelectable()

```

Der Rückgabewert kann dann in ein Objekt des Typs [Checkbox](#) konvertiert werden. Durch Aufruf von [getState](#) kann der aktuelle Zustand bestimmt werden.

```

001 /* Checkbox1.inc */
002
003 private void customizeLayout(Panels panel)
004 {
005     panel.setLayout(new GridLayout(3,1));
006     Checkbox cb = new Checkbox("Checkbox 1");
007     cb.addItemListener(this);
008     panel.add(cb);
009     cb = new Checkbox("Checkbox 2", true);
010     cb.addItemListener(this);
011     panel.add(cb);
012     cb = new Checkbox("Checkbox 3", false);
013     cb.addItemListener(this);
014     panel.add(cb);
015 }

```

Listing 22.3: Verwendung von Checkbox-Komponenten



Abbildung 22.4: Ein Dialog mit Checkbox-Komponenten

Das folgende Listing zeigt eine beispielhafte Implementierung der Methode `itemStateChanged`, die in der Klasse `MyDialog` beim Auftreten eines Item-Events aufgerufen wird (zusätzlich muß `MyDialog` das Interface `ItemListener` implementieren):

[Checkbox2.inc](#)

```

001 /* Checkbox2.inc */
002
003 public void itemStateChanged(ItemEvent event)
004 {
005     Checkbox cb = (Checkbox) event.getItemSelectable();
006     System.out.println(cb.getLabel() + ": " + cb.getState());
007 }

```

Listing 22.4: Behandlung von Item-Events

22.5 CheckboxGroup

- 22.5 CheckboxGroup

Eine [CheckboxGroup](#) ist die Java-Variante einer Gruppe von *Radiobuttons*, also einer Kollektion von Buttons, von denen immer genau einer aktiviert ist. Wird ein anderer Button aktiviert, so ändert er seinen internen Status auf [true](#) und der zuvor gesetzte wird [false](#).

Eine [CheckboxGroup](#) ist nichts anderes als eine [Checkbox](#), deren [CheckboxGroup](#)-Parameter gesetzt ist.

```

public Checkbox(String label, CheckboxGroup cbg, boolean state)
public Checkbox(String label, boolean state, CheckboxGroup cbg)

```

Anders als bei einer gewöhnlichen [Checkbox](#) stehen jetzt nur noch zwei Konstruktoren zur Verfügung. An diese werden die Beschriftung, der Anfangszustand und ein Objekt der Klasse [CheckboxGroup](#) übergeben.

Die [CheckboxGroup](#) sorgt dabei für den Gruppierungseffekt und die Logik beim Umschalten der Radiobuttons. Für jede zusammenhängende Gruppe von Radiobuttons ist daher ein eigenes Objekt der Klasse [CheckboxGroup](#) zu instanzieren und an den Konstruktor zu übergeben.

Die [CheckboxGroup](#) bietet alle Methoden, die auch bei einer [Checkbox](#) verfügbar sind. [setState](#) funktioniert allerdings nur dann, wenn als Parameter [true](#) übergeben wird. In diesem Fall wird der Zustand der zugehörigen [Checkbox](#) gesetzt und der aller anderen Checkboxes derselben Gruppe zurückgenommen. Bei der Übergabe von [false](#) an [setState](#) passiert gar nichts. Zusätzlich gibt es zwei Methoden [getCheckboxGroup](#) und [setCheckboxGroup](#), die den Zugriff auf die zugeordnete [CheckboxGroup](#) erlauben:

```

public void setCheckboxGroup(CheckboxGroup cbg)
public CheckboxGroup getCheckboxGroup()

```

Mit [setCheckboxGroup](#) kann die Zuordnung der Checkboxes zu einer [CheckboxGroup](#) auch nach der Konstruktion geändert werden. Das beim Aufruf von [getCheckboxGroup](#) zurückgegebene Objekt kann dazu verwendet werden, mit Hilfe der Methoden [getCurrent](#) und [setCurrent](#) der Klasse [CheckboxGroup](#) auf das selektierte Element der [CheckboxGroup](#) zuzugreifen:

```

public Checkbox getCurrent()
public void setCurrent(Checkbox box)

```

[getCurrent](#) liefert die [Checkbox](#), deren Zustand [true](#) ist. Um mit [setCurrent](#) eine [Checkbox](#) zu aktivieren, muß diese als Parameter übergeben werden. Alle anderen Checkboxes derselben Gruppe bekommen dann den Zustand [false](#).

Die einzelnen Checkboxes einer [CheckboxGroup](#) senden ein ITEM-Event, wenn die [Checkbox](#) selektiert wird. Dagegen wird kein Ereignis gesendet, wenn die [Checkbox](#) deselektiert wird oder wenn ihr Zustand nicht vom Anwender, sondern vom Programm verändert wird.

Das folgende Beispiel zeigt die Definition von zwei Objekten des Typs [CheckboxGroup](#) mit jeweils drei Elementen. In jeder der beiden Gruppen ist immer genau ein Element markiert, das unabhängig von den Elementen der anderen Gruppe ausgewählt werden kann:

Beispiel

```

001 /* CheckboxGroup.inc */
002
003 private void customizeLayout(Panel panel)
004 {
005     panel.setLayout(new GridLayout(3,2));
006     CheckboxGroup cbg1 = new CheckboxGroup();
007     CheckboxGroup cbg2 = new CheckboxGroup();
008     panel.add(new Checkbox("rot",cbg1,true));
009     panel.add(new Checkbox("eckig",cbg2,true));
010     panel.add(new Checkbox("blau",cbg1,false));
011     panel.add(new Checkbox("rund",cbg2,false));
012     panel.add(new Checkbox("gelb",cbg1,false));
013     panel.add(new Checkbox("schief",cbg2,false));
014 }

```

[CheckboxGroup.inc](#)



Abbildung 22.5: Ein Dialog mit CheckboxGroup-Elementen

22.6 TextField

- [22.6 TextField](#)

Ein [TextField](#) dient zur Darstellung und zur Eingabe von Text. Sowohl der Anwender als auch das Programm können den dargestellten Text auslesen und verändern.

```

public TextField()
public TextField(int width)
public TextField(String text)
public TextField(String text, int width)

```

Der parameterlose Konstruktor erzeugt ein leeres Textfeld, in das der Anwender beliebig viel Text eingeben kann. Wird mit `width` die Breite des Textfeldes vorgegeben, so beschränkt sich die Anzahl der auf dem Bildschirm angezeigten Zeichen auf den angegebenen Wert. Die Anzahl *einzugebender* Zeichen wird allerdings nicht begrenzt. Ist der Text länger, scrollt er innerhalb des Textfeldes. Mit dem Parameter `text` kann eine Zeichenkette vorgegeben werden, die beim Aufruf des Textfeldes vorgelegt wird.

Die Klasse `TextField` ist aus [TextComponent](#) abgeleitet. Sie besitzt eine ganze Reihe von Methoden, mit denen es möglich ist, auf den Text zuzugreifen oder die Eigenschaften des Textfeldes zu verändern. Die Methoden [getText](#) und [setText](#) werden zum Lesen und Verändern des Textes verwendet:

```

public String getText()
public void setText(String text)

```

Mit [getColumns](#) kann die Anzahl der darstellbaren Zeichen des Textfeldes abgefragt und mit [setColumns](#) verändert werden:

```

public int getColumns()
public void setColumns(int columns)

```

Zwei weitere Methoden stehen für das Markieren von Text zur Verfügung:

```

public void selectAll()
public void select(int first, int last)

```

[selectAll](#) markiert den kompletten Text und [select](#) den Bereich von `first` bis `last` (die Zählung beginnt bei 0). Mit den beiden Methoden [getSelectionStart](#) und [getSelectionEnd](#) kann die aktuelle Selektierung abgefragt werden, [getSelectedText](#) liefert den selektierten Text:

```

public int getSelectionStart()
public int getSelectionEnd()
public String getSelectedText()

```

Auf die aktuelle Cursorposition innerhalb des Textes kann mit den Methoden [getCaretPosition](#) und [setCaretPosition](#) zugegriffen werden:

```

public int getCaretPosition()
public void setCaretPosition(int position)

```

Des weiteren kann man verhindern, daß der Text geändert wird, und es besteht die Möglichkeit, verdeckte Eingaben (etwa für Paßwörter) zu realisieren:

```

public void setEditable(boolean allowed)

public boolean isEditable()

public void setEchoCharacter(char c)

public char getEchoChar()

```

Durch einen Aufruf von [setEditable](#) mit [false](#) als Parameter werden weitere Eingaben unterbunden. Der aktuelle Status kann mit [isEditable](#) abgefragt werden. Mit Hilfe von [setEchoCharacter](#) kann ein Zeichen übergeben werden, das bei jedem Tastendruck anstelle des vom Anwender eingegebenen Zeichens ausgegeben wird. Durch Übergabe eines '*' kann beispielsweise die verdeckte Eingabe eines Paßworts realisiert werden.

Ein Textfeld generiert ein Action-Event, wenn der Anwender innerhalb des Textfeldes die ENTER-Taste drückt. In diesem Fall liefert die Methode [getActionCommand](#) des Action-Events den Inhalt des Textfeldes. Eine Methode [setActionCommand](#), die es wie bei Buttons erlaubt, den Rückgabewert von [getActionCommand](#) zu verändern, gibt es bei Textfeldern nicht. Um die Action-Events mehrerer Textfelder, die einen gemeinsamen Empfänger haben, unterscheiden zu können, kann mit [getSource](#) die Ereignisquelle abgefragt werden. Wird ein gemeinsamer Empfänger für alle Action-Events verwendet, so kann das von [getSource](#) gelieferte Objekt mit dem Operator [instanceof](#) daraufhin untersucht werden, ob es sich um ein [TextField](#) oder einen [Button](#) handelt. Für das nachfolgende Beispiel könnte das etwa so aussehen:

[TextField1.inc](#)

```

001 /* TextField1.inc */
002
003 public void actionPerformed(ActionEvent event)
004 {
005     Object obj = event.getSource();
006     if (obj instanceof TextField) {
007         System.out.println(
008             "ButtonAction: "+event.getActionCommand()
009         );
010     } else if (obj instanceof Button) {
011         if (event.getActionCommand().equals("Ende")) {
012             endDialog();
013         }
014     }
015 }

```

Listing 22.6: Verwendung von Textfeldern

Neben dem Action-Ereignis generiert ein Textfeld bei jeder Textänderung ein *Text-Ereignis*. Ein Empfänger für Text-Ereignisse kann mit der Methode [addTextListener](#) von [TextField](#) registriert werden, die als Argument ein Objekt erwartet, das das Interface [TextListener](#) implementiert. Beim Auftreten eines Text-Ereignisses wird im [TextListener](#) die Methode [textValueChanged](#) mit einem [TextEvent](#) als Parameter aufgerufen. [TextEvent](#) ist aus [AWTEvent](#) abgeleitet und erbt die Methoden [getID](#) und [getSource](#), stellt darüber hinaus aber keine eigenen Methoden zur Verfügung. Typischerweise wird innerhalb von [textValueChanged](#) mit [getSource](#) das zugehörige [TextField](#) beschafft und mit [getText](#) auf seinen Inhalt zugegriffen:

[TextField2.inc](#)

```

001 /* TextField2.inc */
002
003 public void textValueChanged(TextEvent event)
004 {
005     TextField tf = (TextField)event.getSource();
006     System.out.println("textValueChanged: "+tf.getText());
007 }

```

Listing 22.7: Behandlung von Text-Events

Das folgende Beispiel zeigt die Anwendung von Textfeldern und die Registrierung von Empfängern für Text-Events und Action-Events. Das Beispiel demonstriert auch, wie man Textfelder und Beschriftungen kombinieren kann. Innerhalb des Dialogs werden in einem [FlowLayout](#) zwei Panels nebeneinander angeordnet. Das erste Panel enthält ein [GridLayout](#) mit drei Zeilen Beschriftung und das zweite ein [GridLayout](#) mit drei Textfeldern. Da die Höhe der Elemente im [GridLayout](#) in beiden Panels identisch ist, stehen Beschriftung und Text jeweils auf der gleichen Höhe nebeneinander.

[TextField3.inc](#)

```
001 /* TextField3.inc */
002
003 private void customizeLayout(Panel panel)
004 {
005     panel.setLayout(new FlowLayout(FlowLayout.LEFT));
006     Panel labelPanel = new Panel();
007     labelPanel.setLayout(new GridLayout(3,1));
008     labelPanel.add(new Label("Name",Label.LEFT));
009     labelPanel.add(new Label("Vorname",Label.LEFT));
010     labelPanel.add(new Label("Ort",Label.LEFT));
011     Panel editPanel = new Panel();
012     editPanel.setLayout(new GridLayout(3,1));
013
014     //Dieses Textfeld sendet Action- und Text-Ereignisse
015     TextField tf = new TextField("Meier",20);
016     tf.addActionListener(this);
017     tf.addTextListener(this);
018     editPanel.add(tf);
019
020     //Diese Textfelder senden keine Ereignisse
021     editPanel.add(new TextField("Luise",20));
022     editPanel.add(new TextField("Hamburg",20));
023     panel.add(labelPanel);
024     panel.add(editPanel);
025 }
```

Listing 22.8: Textfelder mit Beschriftung



Abbildung 22.6: Ein Dialog mit beschrifteten Textfeldern

22.7 TextArea

- [22.7 TextArea](#)

Im Gegensatz zur Klasse [TextField](#), bei der der Text auf eine einzige Zeile beschränkt ist, können mit der Klasse [TextArea](#) mehrzeilige Textfelder erzeugt werden. Zusätzlich kann der Text in alle Richtungen scrollen, so daß auch größere Texte bearbeitet werden können.

[java.awt.TextArea](#)

```
public TextArea()

public TextArea(int rows, int cols)

public TextArea(String text)

public TextArea(String text, int rows, int cols)

public TextArea(String text, int rows, int cols, int scroll)
```

Der parameterlose Konstruktor erzeugt ein leeres [TextArea](#)-Objekt in einer vom System vorgegebenen Größe. Werden die Parameter [rows](#) und [cols](#) vorgegeben, legen sie die Anzahl der sichtbaren Zeilen und Spalten fest. Mit dem Parameter [text](#) kann die zu editierende Zeichenkette übergeben werden. Mit Hilfe des zusätzlichen Parameters [scroll](#) kann die Ausstattung der [TextArea](#) mit Schiebereglern festgelegt werden. Dazu stellt [TextArea](#) die Konstanten [SCROLLBARS_BOTH](#), [SCROLLBARS_VERTICAL_ONLY](#), [SCROLLBARS_HORIZONTAL_ONLY](#) und [SCROLLBARS_NONE](#) zur Verfügung, die als Argument übergeben werden können.

Ebenso wie [TextField](#) ist auch [TextArea](#) aus der Klasse [TextComponent](#) abgeleitet und besitzt mit Ausnahme von [setEchoCharacter](#) und [getEchoCharacter](#) alle Methoden, die bereits bei [TextField](#) vorgestellt wurden. Zusätzlich gibt es einige Methoden, mit denen Teile des Textes verändert werden können:

[java.awt.TextArea](#)

```
public void insert(String str, int pos)

public void append(String str)

public void replaceRange(String text, int start, int end)
```

Mit [insert](#) wird die Zeichenkette [str](#) an der Position [pos](#) eingefügt, und der dahinter stehende Text wird entsprechend nach hinten geschoben. [append](#) hängt den als Argument übergebenen Text hinten an den bestehenden Text an, und [replaceRange](#) ersetzt den Text zwischen [start](#) und [end](#) durch die übergebene Zeichenkette [text](#).

Ein Objekt der Klasse [TextArea](#) sendet Text-Events, so wie es bei [TextField](#) beschrieben wurde. Action-Events werden dagegen nicht gesendet, denn die ENTER-Taste erzeugt eine Zeilenschaltung. Ein zum folgenden Beispiel passender Event-Handler könnte so realisiert werden:

[TextArea2.inc](#)

```
001 /* TextArea2.inc */
002
003 public void textValueChanged(TextEvent event)
004 {
005     TextArea tf = (TextArea)event.getSource();
006     System.out.println("textValueChanged: "+tf.getText());
007 }
```

Listing 22.9: Behandlung von Text-Events bei der Komponente TextArea

Das Beispielprogramm zum Testen der [TextArea](#)-Komponente sieht so aus:

Beispiel

[TextArea1.inc](#)

```
001 /* TextArea1.inc */
002
003 private void customizeLayout(Panels panel)
004 {
005     panel.setLayout(new FlowLayout(FlowLayout.LEFT));
006     TextArea ta = new TextArea(10,40);
007     ta.addTextListener(this);
008     panel.add(ta);
009 }
```

Listing 22.10: Verwendung einer TextArea-Komponente

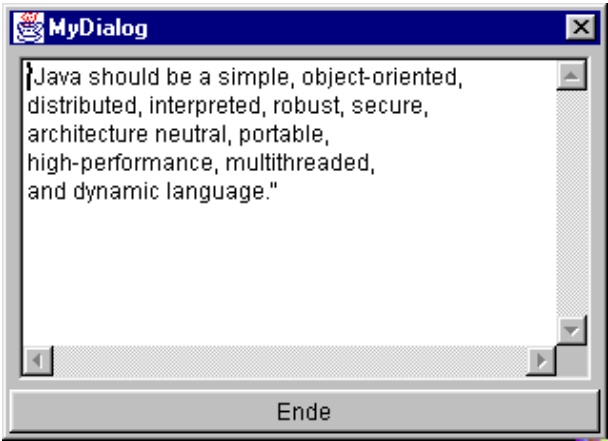


Abbildung 22.7: Ein Dialog mit einem TextArea-Objekt

22.8 Choice

- [22.8 Choice](#)

Ein [Choice](#)-Menü ist ein aufklappbares Textfeld, dessen Inhalt aus einer vom Programm vorgegebenen Liste ausgewählt werden kann. Dialogelemente des Typs [Choice](#) entsprechen den unter Windows üblichen *Drop-Down-Listboxen*, einer eingeschränkten Form der *Comboboxen*, bei denen das zugehörige Editfeld nicht separat geändert werden kann.

[java.awt.Choice](#)

```
public Choice()
```

Der parameterlose Konstruktor erstellt ein [Choice](#)-Objekt mit einer leeren Liste.

Eine der wichtigsten Methoden der Klasse [Choice](#) ist [add](#):

[java.awt.Choice](#)

```
public void add(String item)
```

Jeder Aufruf von [add](#) hängt das übergebene Element [item](#) an das Ende der Liste an. Die Elemente werden dabei in der Reihenfolge der Aufrufe von [add](#) eingefügt. Eine automatische Sortierung wurde in Java nicht vorgesehen.

Analog zur Methode [add](#) gibt es eine bedeutungsgleiche Methode [addItem](#). Während in

JDK1.1/1.2

 der Klasse [Choice](#) beide Methoden verwendet werden können, wurde die Methode [addItem](#) in der Klasse [List](#) mit dem JDK 1.2 als [deprecated](#) markiert, sollte also nicht mehr verwendet werden.

Für den Zugriff auf die Elemente der Combobox stehen die Methoden [getItemCount](#), [getSelectedIndex](#), [getItem](#) und [getSelectedItem](#) zur Verfügung:

[java.awt.Choice](#)

```
public int getSelectedIndex()
public String getItem(int n)
public String getSelectedItem()
public int getItemCount()
```

[getItemCount](#) liefert die Anzahl der Elemente, die sich gegenwärtig in der Liste befinden. Mit [getItem](#) kann auf das *n*-te Element der Liste über seinen Index zugegriffen werden; die Zählung beginnt bei 0.

[getSelectedIndex](#) liefert den Index des selektierten Elements, so daß durch Übergabe dieses Wertes an [getItem](#) das aktuelle Element beschafft werden kann. Alternativ dazu kann [getSelectedItem](#) aufgerufen werden, um ohne explizite Kenntnis der internen Nummer das selektierte Element zu beschaffen.

Ebenso wie der Anwender kann auch das Programm selbst ein Element aus der Liste selektieren:

[java.awt.Choice](#)

```
public select(int)
public select(String)
```

Hierbei wird an [select](#) wahlweise ein [int](#) oder ein [String](#) übergeben. Im ersten Fall wird das Element über seinen numerischen Index selektiert, im zweiten direkt durch Suche des angegebenen Wertes.

Ebenso wie eine [Checkbox](#) sendet auch ein [Choice](#)-Element Item-Ereignisse, wenn ein Element selektiert wurde. Um auf dieses Event zu reagieren, ist durch Aufruf von [addItemListener](#) ein [ItemListener](#) zu registrieren:

[java.awt.Choice](#)

```
public void addItemListener(ItemListener l)
```

Ein Item-Event wird immer dann gesendet, wenn ein anderes Element ausgewählt wurde. In diesem Fall wird im Ereignisempfänger die Methode [itemStateChanged](#) mit einem Argument vom Typ [ItemEvent](#) aufgerufen:

[java.awt.event.ItemEvent](#)

```
public abstract void itemStateChanged(ItemEvent e)
```

Das [ItemEvent](#) stellt die Methode [getItemSelectable](#) zur Verfügung, mit der ermittelt werden kann, durch welches [Choice](#)-Element das Ereignis ausgelöst wurde. Zusätzlich gibt es die Methode [getItem](#), die das ausgewählte Element als [String](#) zur Verfügung stellt:

```
public ItemSelectable getItemSelectable()

public Object getItem()
```

Damit kann offensichtlich auf zwei unterschiedliche Arten auf die Änderung der Auswahl reagiert werden. Einerseits kann mit [getItemSelectable](#) ein [Choice](#)-Element beschafft und mit [getSelectedItem](#) das selektierte Element ermittelt werden. Andererseits kann [getItem](#) aufgerufen und das Ergebnis in einen [String](#) umgewandelt werden. Das folgende Listing demonstriert dies beispielhaft:

```
001 /* Choice2.inc */
002
003 public void itemStateChanged(ItemEvent event)
004 {
005     Choice choice = (Choice) event.getItemSelectable();
006     String str1 = choice.getSelectedItem();
007     String str2 = (String) event.getItem();
008     System.out.println("choice.getSelectedItem: " + str1);
009     System.out.println("event.getItem:         " + str2);
010 }
```

Listing 22.11: Behandlung der Ereignisse einer Choice-Komponente

Das Beispielprogramm zum Testen der [Choice](#)-Komponente sieht so aus:

Beispiel

Choice1.inc

```
001 /* Choice1.inc */
002
003 private void customizeLayout(Panels panel)
004 {
005     panel.setLayout(new FlowLayout());
006     Choice choice = new Choice();
007     choice.addItemListener(this);
008     choice.add("rot");
009     choice.add("grün");
010     choice.add("gelb");
011     choice.add("blau");
012     choice.add("rosa");
013     choice.add("lila");
014     panel.add(choice);
015 }
```

Listing 22.12: Verwendung einer Choice-Komponente

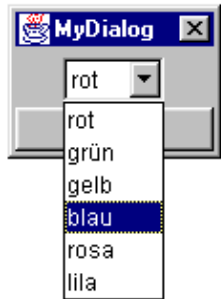


Abbildung 22.8: Ein Dialog mit einer Choice-Komponente

22.9 List

- 22.9 List

Eine [List](#) ist eine listenartige Darstellung von Werten, aus denen der Anwender einen oder mehrere auswählen kann. Anders als ein [Choice](#)-Element ist ein Element der Klasse [List](#) ständig in voller Größe auf dem Bildschirm sichtbar. Unter Windows werden Objekte der Klasse [List](#) durch *Listboxen* dargestellt.

[java.awt.List](#)

```
public List()
```

```
public List(int size)
```

```
public List(int size, boolean multiselect)
```

Der parameterlose Konstruktor legt eine leere Listbox an, deren dargestellte Größe vom Layoutmanager begrenzt wird. Der Parameter [size](#) legt die Anzahl der angezeigten Zeilen fest. Dabei können auch mehr Elemente in der Listbox enthalten sein, als auf einmal angezeigt werden können. [List](#)-Dialogelemente können wahlweise die *Mehrfachselektion* von Elementen zulassen. Wird bei der Konstruktion der Listbox der Parameter [multiselect](#) auf [true](#) gesetzt, kann der Anwender nicht nur ein Element, sondern auch mehrere Elemente selektieren.

Das Dialogelement [List](#) bietet nahezu dieselbe Funktionalität wie [Choice](#). Zusätzlich stehen die Methoden [getSelectedIndexes](#) und [getSelectedItems](#) zur Verfügung, um bei einer Mehrfachselektion auf die selektierten Elemente zugreifen zu können:

[java.awt.List](#)

```
public int[] getSelectedIndexes()
```

```
public String[] getSelectedItems()
```

[getSelectedIndexes](#) liefert ein Array mit den Indexpositionen aller selektierten Elemente, und [getSelectedItems](#) liefert eine Liste der Werte selbst.

Mit den Methoden [select](#) und [deselect](#) lassen sich einzelne Elemente selektieren oder deselektieren:

[java.awt.List](#)

```
public void select(int index)
```

```
public void deselect(int index)
```

Im Gegensatz zu [Choice](#) ist es bei einer Komponente des Typs [List](#) auch möglich, Elemente zu entfernen oder ihren Wert zu ändern:

[java.awt.List](#)

```
public void delItem(int index)
```

```
public void remove(int index)
```

```
public void replaceItem(String newValue, int index)
```

Sowohl [delItem](#) als auch [remove](#) löschen das Element an der Position [index](#), und [replaceItem](#) ersetzt das Element an der Position [index](#) durch den neuen Wert [newValue](#).

Eine [List](#) sendet sowohl Item-Ereignisse als auch Action-Ereignisse. Ein Action-Ereignis wird dann generiert, wenn ein Listenelement per Doppelklick ausgewählt wurde. Das Drücken der ENTER-Taste löst - wenigstens in der aktuellen Windows-Implementierung - *kein* Action-Ereignis aus. Ein Item-Ereignis wird ausgelöst, nachdem in der Liste ein Element ausgewählt wurde. Bezüglich der Implementierung von Event-Handleern für diese Ereignisse verweisen wir auf die Beschreibungen der Klassen [Choice](#) und [TextField](#) weiter oben.

Falls ein Item-Ereignis ausgelöst wurde, verhält sich die Methode [getItemSelectable](#) des [ItemEvent](#) analog zu [Choice](#) und liefert das [List](#)-Objekt, das das Ereignis ausgelöst hat. Der Rückgabewert von [getItem](#) ist dagegen anders zu interpretieren, denn er liefert nicht das ausgewählte Element als [String](#), sondern dessen Position als [Integer](#)-Objekt. Dieses kann mit [intValue](#) in ein [int](#) konvertiert werden, das die Position des ausgewählten Elements liefert. Beispielhafte Implementierungen von [actionPerformed](#) und [itemStateChanged](#) könnten folgendermaßen aussehen:

Warnung

```

001 /* List2.inc */
002
003 public void itemStateChanged(ItemEvent event)
004 {
005     List list = (List) event.getItemSelectable();
006     String str1 = list.getSelectedItemAt();
007     int pos = ((Integer) event.getItem()).intValue();
008     System.out.println("list.getSelectedItemAt: " + str1);
009     System.out.println("event.getItem:         " + pos);
010 }
011
012 public void actionPerformed(ActionEvent event)
013 {
014     Object obj = event.getSource();
015     if (obj instanceof List) {
016         System.out.println("ListAction: "+event.getActionCommand());
017     } else if (obj instanceof Button) {
018         if (event.getActionCommand().equals("Ende")) {
019             endDialog();
020         }
021     }
022 }

```

Listing 22.13: Behandlung der Ereignisse einer List-Komponente

Das Beispielprogramm für die Verwendung der [List](#)-Komponente sieht so aus:

Beispiel

[List1.inc](#)

```

001 /* List1.inc */
002
003 private void customizeLayout(Panels panel)
004 {
005     panel.setLayout(new FlowLayout(FlowLayout.LEFT));
006     List list = new List(6,false);
007     list.addActionListener(this);
008     list.addItemListener(this);
009     list.add("Äpfel");
010     list.add("Birnen");
011     list.add("Bananen");
012     list.add("Pfirsiche");
013     list.add("Kirschen");
014     list.add("Kiwis");
015     list.add("Ananas");
016     list.add("Erdbeeren");
017     list.add("Blaubeeren");
018     list.add("Mandarinen");
019     panel.add(list);
020     list.select(1);
021 }

```

Listing 22.14: Verwendung einer List-Komponente



Abbildung 22.9: Ein Dialog mit einer Listbox

22.10 Scrollbar

- [22.10 Scrollbar](#)

Ein [Scrollbar](#) ist ein Schieberegler, der zur quasianalogen Anzeige und Eingabe eines Wertes aus einem vorgegebenen Wertebereich verwendet werden kann. Der Schieberegler kann entweder horizontal oder vertikal angeordnet werden und besitzt einen Schieber, dessen Größe veränderlich ist. Der interne Wert eines Schiebereglers und die Anzeigeposition seines Schiebers sind untrennbar miteinander verbunden. Ändert der Anwender die Position des Schiebers, ändert sich automatisch auch sein interner Wert. Wird vom Programm der Wert verändert, führt dies auch zu einer Repositionierung des Schiebers.

```

public Scrollbar()

public Scrollbar(int orientation)

public Scrollbar(
    int orientation, int value, int visible,
    int minimum, int maximum
)

```

Der parameterlose Konstruktor erzeugt einen vertikalen Schieberegler. Mit dem Parameter [orientation](#) kann die Orientierung festgelegt werden. Hier kann eine der Konstanten [Scrollbar.HORIZONTAL](#) oder [Scrollbar.VERTICAL](#) angegeben werden.

Der dritte Konstruktor erlaubt die Angabe weiterer Eigenschaften. [minimum](#) und [maximum](#) spezifizieren die Grenzen des repräsentierten Wertebereichs. Die untere Grenze liegt bei einem vertikalen Schieberegler oben und bei einem horizontalen Schieberegler auf der linken Seite. Mit [value](#) kann der Anfangswert des Schiebers festgelegt werden.

Der Parameter [visible](#) dient dazu, die *Seitengröße* des Schiebers zu bestimmen. Diese muß kleiner als der Wertebereich des Schiebereglers sein. Die Seitengröße bestimmt einerseits die visuelle Größe des Schiebers und andererseits die Größe der Veränderung des Wertes, wenn der Anwender auf die Schaltfläche zwischen Schieber und Button des Schiebereglers klickt.

Die Methoden der Klasse [Scrollbar](#) realisieren den Zugriff auf die Attribute des Schiebereglers. Die meisten von ihnen sind im Interface [Adjustable](#) definiert, das von [Scrollbar](#) implementiert wird. Mit [getValue](#) und [setValue](#) wird auf den aktuellen Wert des Schiebers zugegriffen, mit [getMinimum](#) und [getMaximum](#) auf die Grenzen des Wertebereichs und mit [getVisibleAmount](#) auf die Größe des Schiebers. Zusätzlich kann mit [getUnitIncrement](#) und [setUnitIncrement](#) sowie mit [getBlockIncrement](#) und [setBlockIncrement](#) auf die Parameter zugegriffen werden, die die Stärke der Veränderung des Wertes beim Klicken auf die Buttons bzw. die Schaltfläche zwischen Schieber und Buttons bestimmen.

```

public int getValue()

public void setValue(int value)

public int getMinimum()

public int getMaximum()

public int getVisible()

public int getUnitIncrement()

public void setUnitIncrement(int l)

public int getBlockIncrement()

public void setBlockIncrement(int l)

```

Ein [Scrollbar](#) sendet *Adjustment-Ereignisse* an seine Ereignisempfänger. Diese müssen das Interface [AdjustmentListener](#) implementieren und sich durch Aufruf von [addAdjustmentListener](#) registrieren:

```

public void addAdjustmentListener(AdjustmentListener l)

```

Das Adjustment-Ereignis führt im Ereignisempfänger zum Aufruf der Methode `adjustmentValueChanged`, die ein `AdjustmentEvent` übergeben bekommt:

```
java.awt.event AdjustmentListener
public abstract void adjustmentValueChanged(AdjustmentEvent e)
```

Dieses besitzt die Methoden `getAdjustable`, mit der der auslösende Scrollbar bestimmt werden kann, und `getValue`, mit der der aktuelle Wert des Schiebereglers bestimmt werden kann. Zusätzlich gibt es die Methode `getAdjustmentType`, die Auskunft darüber gibt, welche Benutzeraktion zur Auslösung des Ereignisses führte. [Tabelle 22.1](#) listet die möglichen Konstanten auf, die von `getAdjustmentType` zurückgegeben werden können, und beschreibt ihre Bedeutung.

Konstante	Bedeutung
<code>AdjustmentEvent.UNIT_INCREMENT</code>	Der Wert wurde durch Klicken eines Buttons um eine Einheit erhöht.
<code>AdjustmentEvent.UNIT_DECREMENT</code>	Der Wert wurde durch Klicken eines Buttons um eine Einheit vermindert.
<code>AdjustmentEvent.BLOCK_INCREMENT</code>	Der Wert wurde durch Klicken der Schaltfläche zwischen Button und Schieber um eine Seite erhöht.
<code>AdjustmentEvent.BLOCK_DECREMENT</code>	Der Wert wurde durch Klicken der Schaltfläche zwischen Button und Schieber um eine Seite vermindert.
<code>AdjustmentEvent.TRACK</code>	Der Wert wurde durch Ziehen des Schiebers verändert.

Tabelle 22.1: Konstanten für Schieberegler-Ereignisse

Eine beispielhafte Implementierung von `adjustmentValueChanged` könnte etwa so aussehen:

Beispiel

[Scrollbar.inc](#)

```
001 /* Scrollbar.inc */
002
003 public void adjustmentValueChanged(AdjustmentEvent event)
004 {
005     Adjustable sb = event.getAdjustable();
006     if (sb.getOrientation() == Scrollbar.HORIZONTAL) {
007         System.out.print("Horizontal: ");
008     } else {
009         System.out.print("Vertikal: ");
010     }
011     switch (event.getAdjustmentType()) {
012     case AdjustmentEvent.UNIT_INCREMENT:
013         System.out.println("AdjustmentEvent.UNIT_INCREMENT");
014         break;
015     case AdjustmentEvent.UNIT_DECREMENT:
016         System.out.println("AdjustmentEvent.UNIT_DECREMENT");
017         break;
018     case AdjustmentEvent.BLOCK_DECREMENT:
019         System.out.println("AdjustmentEvent.BLOCK_DECREMENT");
020         break;
021     case AdjustmentEvent.BLOCK_INCREMENT:
022         System.out.println("AdjustmentEvent.BLOCK_INCREMENT");
023         break;
024     case AdjustmentEvent.TRACK:
025         System.out.println("AdjustmentEvent.TRACK");
026         break;
027     }
028     System.out.println(" value: " + event.getValue());
029 }
030
031 private void customizeLayout(Panels panel)
032 {
033     panel.setLayout(new BorderLayout());
034     Scrollbar hsb=new Scrollbar(Scrollbar.HORIZONTAL,1,10,1,100);
035     hsb.addAdjustmentListener(this);
036     panel.add("South",hsb);
037     Scrollbar vsb=new Scrollbar(Scrollbar.VERTICAL, 1,10,1,100);
038     vsb.addAdjustmentListener(this);
```

```
039     panel.add( "East", vsb );
040 }
```

Listing 22.15: Verwendung von Scrollbars



Abbildung 22.10: Ein Dialog mit zwei Schiebereglern

22.11 ScrollPane

- 22.11 ScrollPane

Ein sehr nützliches Dialogelement, das mit der Version 1.1 des JDK eingeführt wurde, ist [ScrollPane](#), ein [Container](#) für automatisches horizontales und vertikales Scrolling. Ein [ScrollPane](#) ist von der Funktion her einem [Panel](#) ähnlich und kann wie jedes andere Dialogelement innerhalb eines Fensters verwendet werden. [ScrollPane](#) unterscheidet sich allerdings durch zwei wichtige Eigenschaften von einem gewöhnlichen [Panel](#):

- Es kann *genau ein* Dialogelement aufnehmen und benötigt keinen expliziten Layoutmanager.
- Es ist in der Lage, eine *virtuelle Ausgabefläche* zu verwalten, die größer ist als die auf dem Bildschirm zur Verfügung stehende.

Die innerhalb von [ScrollPane](#) angezeigte Komponente arbeitet dabei immer mit der virtuellen Ausgabefläche und merkt nichts von eventuellen Größenbeschränkungen auf dem Bildschirm. Falls die benötigte Ausgabefläche größer ist als die anzeigbare, blendet [ScrollPane](#) automatisch die erforderlichen Schieberegler ein, um das Dialogelement horizontal und vertikal verschieben zu können.

Hinweis

Zur Instanziierung eines [ScrollPane](#) stehen zwei Konstruktoren zur Verfügung:

[java.awt.ScrollPane](#)

```
public ScrollPane()  
  
public ScrollPane(int scrollbarDisplayPolicy)
```

Der Parameter `scrollbarDisplayPolicy` definiert die Strategie zur Anzeige der Schieberegler entsprechend den in [Tabelle 22.2](#) aufgelisteten Konstanten.

Konstante	Bedeutung
ScrollPane.SCROLLBARS_AS_NEEDED	Die Schieberegler werden genau dann angezeigt, wenn es erforderlich ist, wenn also mehr Platz benötigt wird, als zur Anzeige zur Verfügung steht.
ScrollPane.SCROLLBARS_ALWAYS	Die Schieberegler werden immer angezeigt.
ScrollPane.SCROLLBARS_NEVER	Die Schieberegler werden nie angezeigt, und der Bildschirmausschnitt kann nur vom Programm aus verschoben werden.

Tabelle 22.2: Konstanten zur Anzeige der Schieberegler in [ScrollPane](#)

Nach der Konstruktion des [ScrollPane](#) wird die aus [Container](#) geerbte Methode [add](#) aufgerufen, um eine Komponente hinzuzufügen. Im Gegensatz zu anderen [Containers](#) sollte [add](#) hier lediglich einmal aufgerufen werden, denn [ScrollPane](#) kann nur ein einziges Element aufnehmen. Soll ein komplexer Dialog mit vielen Elementen dargestellt werden, so müssen diese in ein [Panel](#) verpackt und dann gemeinsam an [add](#) übergeben werden.

Nach dem Einfügen einer Komponente kann die Methode [setSize](#) aufgerufen werden, um die sichtbare Größe des [ScrollPane](#) festzulegen. Die sichtbare Größe ist die Größe, in der das [ScrollPane](#) dem Fenster erscheint, in das es eingebettet wurde. Der dadurch definierte Ausschnitt aus dem virtuellen Ausgabebereich wird als *Viewport* bezeichnet.

Die Größe des virtuellen Ausgabebereichs wird dagegen durch das mit [add](#) eingefügte Element selbst festgelegt. Es überlagert dazu [getPreferredSize](#) und gibt so die gewünschten Abmessungen an den Aufrufer zurück. Die Methode [getPreferredSize](#) spielt eine wichtige Rolle bei der automatischen Anordnung von Dialogelementen mit Hilfe eines Layoutmanagers. Sie wird aufgerufen, wenn ein [Container](#) wissen will, wieviel Platz eine Komponente innerhalb eines Bildschirmlayouts belegt. Wir werden in [Kapitel 23](#) noch einmal auf [getPreferredSize](#) zurückkommen. [Abbildung 22.11](#) faßt die bisherigen Ausführungen zusammen.

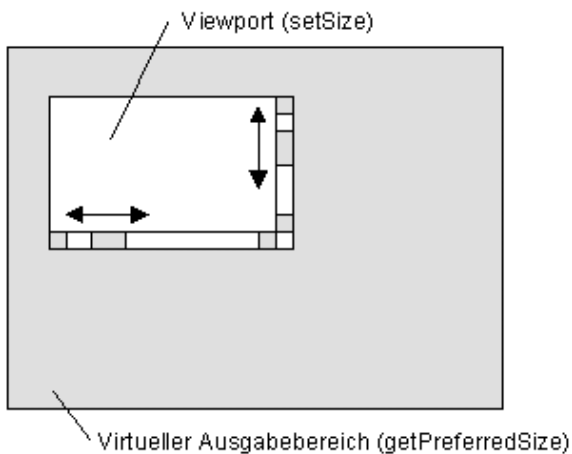


Abbildung 22.11: ViewPort und virtueller Ausgabebereich beim ScrollPane

Die Klasse [ScrollPane](#) stellt einige Methoden zur Verfügung, mit denen die Darstellung und Reaktion der Schieberegler beeinflusst werden kann:

[java.awt.ScrollPane](#)

```
public Adjustable getHAdjustable()
```

```
public Adjustable getVAdjustable()
```

Mit [getHAdjustable](#) wird ein Objekt beschafft, das den Zugriff auf den horizontalen Schieberegler ermöglicht, und mit [getVAdjustable](#) eines für den vertikalen Schieberegler. Das Interface [Adjustable](#) wurde bereits beim [Scrollbar](#) vorgestellt, es abstrahiert den Zugriff auf einen Schieberegler. Da die meisten Eigenschaften der Schieberegler bereits von [ScrollPane](#) voreingestellt werden, sollte sich das eigene Programm darauf beschränken, durch Aufruf von [setUnitIncrement](#) die Schrittweite der Schieberegler einzustellen.

Des weiteren gibt es einige Methoden für den Zugriff auf den Viewport und seine Position innerhalb des virtuellen Ausgabebereichs:

[java.awt.ScrollPane](#)

```
public Dimension getViewPortSize()
```

```
public void setScrollPosition(int x, int y)
```

```
public Point getScrollPosition()
```

[getViewPortSize](#) liefert die aktuelle Größe des Viewports. Mit [getScrollPosition](#) kann die Position desjenigen Punktes der virtuellen Ausgabefläche ermittelt werden, der gerade in der linken oberen Ecke des Viewports angezeigt wird. Mit [setScrollPosition](#) kann der Viewport vom Programm verschoben werden. Die übergebenen Werte müssen dabei zwischen 0,0 und der jeweils maximalen Größe des virtuellen Ausgabebereichs minus der Größe des Viewports liegen.

Wir wollen ein einfaches Beispiel konstruieren, das die Anwendung von [ScrollPane](#) demonstriert. Dazu soll ein Programm geschrieben werden, das eine Liste aller Systemfarben mit ihrer Bezeichnung und einer Farbprobe auf dem Bildschirm ausgibt. Da es insgesamt 26 verschiedene Systemfarben gibt, soll das Programm jeweils nur einen kleinen Ausschnitt darstellen, der vom Anwender verschoben werden kann.

Beispiel

Zur Erstellung des Programms gehen wir in drei Schritten vor:

- Wir definieren zunächst eine Hilfsklasse [NamedSystemColors](#), die alle verfügbaren Systemfarben sortiert und einen einfachen Zugriff auf ihre Namen und das zugehörige Farbobjekt erlaubt.
- Anschließend definieren wir eine Klasse [SystemColorViewer](#), die aus [Canvas](#) abgeleitet wird und dazu dient, die Systemfarben nacheinander auf dem Bildschirm auszugeben. Wir überlagern die Methode [getPreferredSize](#), um die zur Ausgabe aller Farben erforderliche Größe der Komponente bekannt zu machen. Außerdem überlagern wir [paint](#), um die eigentliche Ausgabe zu realisieren. Dabei reserviert [getPreferredSize](#) ausreichend Platz, um jede Farbe in einer eigenen Ausgabezeile darstellen zu können. Innerhalb von [paint](#) wird lediglich jedes einzelne Farbobjekt aus [NamedSystemColors](#) beschafft und dazu die Farbbox und der zugehörige Text ausgegeben. In [Kapitel 23](#) werden wir noch einmal genauer auf die Verwendung von [Canvas](#) zur Erzeugung eigener Komponenten zurückkommen.
- Nach diesen Vorarbeiten ist die Verwendung von [ScrollPane](#) ganz einfach. Wir legen dazu einen [Frame](#) an, der das Hauptfenster unserer Applikation wird, und instanzieren das [ScrollPane](#)-Objekt. Mit [add](#) wird eine Instanz von [SystemColorViewer](#) übergeben, anschließend die Schrittweite der Schieberegler angepaßt und schließlich mit [getSize](#) die Größe des sichtbaren Ausschnitts festgelegt. Nachdem [ScrollPane](#) an den [Frame](#) übergeben wurde, ist das Programm einsatzbereit.

Der nachfolgende Quellcode enthält alle drei Klassen und zeigt, wie sie zusammenarbeiten:

```

001 /* Listing2216.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class NamedSystemColors
007 {
008     String      names[];
009     SystemColor colors[];
010
011     public NamedSystemColors()
012     {
013         names  = new String[SystemColor.NUM_COLORS];
014         colors = new SystemColor[SystemColor.NUM_COLORS];
015         names [ 0] = "desktop";
016         colors[ 0] = SystemColor.desktop;
017         names [ 1]= "activeCaption";
018         colors[ 1] = SystemColor.activeCaption;
019         names [ 2] = "activeCaptionText";
020         colors[ 2] = SystemColor.activeCaptionText;
021         names [ 3] = "activeCaptionBorder";
022         colors[ 3] = SystemColor.activeCaptionBorder;
023         names [ 4] = "inactiveCaption";
024         colors[ 4] = SystemColor.inactiveCaption;
025         names [ 5] = "inactiveCaptionText";
026         colors[ 5] = SystemColor.inactiveCaptionText;
027         names [ 6] = "inactiveCaptionBorder";
028         colors[ 6] = SystemColor.inactiveCaptionBorder;
029         names [ 7] = "window";
030         colors[ 7] = SystemColor.window;
031         names [ 8] = "windowBorder";
032         colors[ 8] = SystemColor.windowBorder;
033         names [ 9] = "windowText";
034         colors[ 9] = SystemColor.windowText;
035         names [10] = "menu";
036         colors[10] = SystemColor.menu;
037         names [11] = "menuText";
038         colors[11] = SystemColor.menuText;
039         names [12] = "text";
040         colors[12] = SystemColor.text;
041         names [13] = "textText";
042         colors[13] = SystemColor.textText;
043         names [14] = "textHighlight";
044         colors[14] = SystemColor.textHighlight;
045         names [15] = "textHighlightText";
046         colors[15] = SystemColor.textHighlightText;
047         names [16] = "textInactiveText";
048         colors[16] = SystemColor.textInactiveText;
049         names [17] = "control";
050         colors[17] = SystemColor.control;
051         names [18] = "controlText";
052         colors[18] = SystemColor.controlText;
053         names [19] = "controlHighlight";
054         colors[19] = SystemColor.controlHighlight;
055         names [20] = "controlLtHighlight";
056         colors[20] = SystemColor.controlLtHighlight;
057         names [21] = "controlShadow";
058         colors[21] = SystemColor.controlShadow;
059         names [22] = "controlDkShadow";
060         colors[22] = SystemColor.controlDkShadow;
061         names [23] = "scrollbar";
062         colors[23] = SystemColor.scrollbar;
063         names [24] = "info";
064         colors[24] = SystemColor.info;

```



```

065     names [25] = "infoText";
066     colors[25] = SystemColor.infoText;
067 }
068
069 public int getSize()
070 {
071     return SystemColor.NUM_COLORS;
072 }
073
074 public String getName(int i)
075 {
076     return names[i];
077 }
078
079 public SystemColor getColor(int i)
080 {
081     return colors[i];
082 }
083 }
084
085 class SystemColorViewer
086 extends Canvas
087 {
088     NamedSystemColors colors;
089
090     public SystemColorViewer()
091     {
092         colors = new NamedSystemColors();
093     }
094
095     public Dimension getPreferredSize()
096     {
097         return new Dimension(150,16 + colors.getSize() * 20);
098     }
099
100     public void paint(Graphics g)
101     {
102         for (int i = 0; i < colors.getSize(); ++i) {
103             //Rahmen für Farbbox
104             g.setColor(Color.black);
105             g.drawRect(10,16+20*i,16,16);
106             //Farbbox
107             g.setColor(colors.getColor(i));
108             g.fillRect(11,17+20*i,15,15);
109             //Bezeichnung
110             g.setColor(Color.black);
111             g.drawString(colors.getName(i),30,30+20*i);
112         }
113     }
114 }
115
116 public class Listing2216
117 extends Frame
118 {
119     public static void main(String[] args)
120     {
121         Listing2216 wnd = new Listing2216();
122         wnd.setLocation(200,100);
123         wnd.setVisible(true);
124     }
125
126     public Listing2216()
127     {
128         super("Listing2216");
129         setBackground(Color.lightGray);

```

```
130 //ScrollPane
131 ScrollPane sc = new ScrollPane(
132     ScrollPane.SCROLLBARS_AS_NEEDED
133 );
134 sc.add(new SystemColorViewer());
135 sc.getVAdjustable().setUnitIncrement(1);
136 sc.getHAdjustable().setUnitIncrement(1);
137 sc.setSize(200,200);
138 add(sc);
139 //Window-Listener
140 addWindowListener(
141     new WindowAdapter() {
142         public void windowClosing(WindowEvent event)
143         {
144             setVisible(false);
145             dispose();
146             System.exit(0);
147         }
148     }
149 );
150 //Dialogelement anordnen
151 pack();
152 }
153 }
```

Listing 22.16: Verwendung der Klasse ScrollPane

Die `paint`-Methode von `SystemColorViewer` ist etwas ineffizient, denn sie gibt bei jedem Aufruf den gesamten Inhalt der virtuellen Ausgabefläche komplett aus. Eine bessere Implementierung würde zuvor die Position und Größe der Clipping-Region abfragen und nur diesen Bereich aktualisieren. Die Clipping-Region enthält dabei einen Bereich, dessen Größe und Lage exakt dem aktuellen Viewport entspricht.

Tip

Ein Probelauf des Programms ergibt folgende Ausgabe:

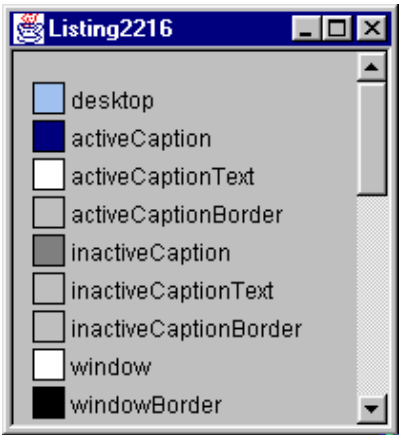


Abbildung 22.12: Verwendung von ScrollPane

22.12 Zusammenfassung

- [22.12 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Eigenschaften der Klasse [Component](#) als Basisklasse aller Dialogelemente
- Die Klasse [Label](#) zur Beschriftung eines Dialogs
- Erzeugen einer Schaltfläche mit der Klasse [Button](#)
- Erzeugen eines Ankreuzfeldes mit der Klasse [Checkbox](#)
- Zusammenfassen von Ankreuzfeldern mit Hilfe der Klasse [CheckboxGroup](#)
- Erzeugen eines einzelnen Textfeldes mit der Klasse [TextField](#)
- Erzeugen eines mehrzeiligen Textfeldes mit der Klasse [TextArea](#)
- Erzeugen von Kombinationsfeldern mit der Klasse [Choice](#)
- Erzeugen von Listefeldern mit der Klasse [List](#)
- Erzeugen von horizontalen und vertikalen Schiebereglern mit der Klasse [Scrollbar](#)
- Realisierung einer virtuellen Ausgabefläche mit automatischem Scrolling mit Hilfe der Klasse [ScrollPane](#)

Kapitel 23

Eigene Dialogelemente

- [23 Eigene Dialogelemente](#)
 - [23.1 Die Klasse Canvas](#)
 - [23.2 Entwicklung einer 7-Segment-Anzeige](#)
 - [23.2.1 Anforderungen](#)
 - [23.2.2 Bildschirmanzeige](#)
 - [23.2.3 Ereignisbehandlung](#)
 - [23.3 Einbinden der Komponente](#)
 - [23.4 Zusammenfassung](#)

23.1 Die Klasse Canvas

- 23.1 Die Klasse Canvas

Bisher haben wir uns nur mit *vordefinierten* Dialogelementen beschäftigt. Java bietet aber auch die Möglichkeit, Dialogelemente selbst zu definieren und wie eine vordefinierte Komponente zu verwenden. In der *Java-Beans-Spezifikation* wurden die dazu erforderlichen Techniken beschrieben und die zugrundeliegenden Konzepte definiert. Wir wollen an dieser Stelle allerdings nicht so weit ins Detail gehen, sondern uns lediglich mit den Grundlagen der Entwicklung eigener Dialogelemente auf der Basis der Klasse [Canvas](#) befassen, wie sie mit der Klasse [SystemColorViewer](#) in [Abschnitt 22.11](#) angedeutet werden.

Ein [Canvas](#) ist ein frei definierbares Dialogelement, das in der Grundversion praktisch keinerlei Funktionalität zur Verfügung stellt. Damit ein [Canvas](#) etwas Sinnvolles tun kann, muß daraus eine eigene Klasse abgeleitet werden, und in dieser müssen die Methode [paint](#) und die Methoden zur Reaktion auf Nachrichten überlagert werden.

```
public Canvas()
```

java.awt.Canvas

Durch Überlagern der [paint](#)-Methode sorgt eine [Canvas](#)-Komponente für die Darstellung auf dem Bildschirm:

```
public void paint(Graphics g)
```

java.awt.Canvas

Die Standardversion von [paint](#) zeichnet nur die Ausgabefläche in der aktuellen Hintergrundfarbe. Eine überlagernde Version kann hier natürlich ein beliebig komplexes Darstellungsverhalten realisieren. Der Punkt (0,0) des übergebenen [Graphics](#)-Objektes entspricht dabei der linken oberen Ecke des Ausgabebereichs.

Da die Klasse [Canvas](#) aus [Component](#) abgeleitet ist, bekommt ein [Canvas](#)-Objekt alle Ereignisse zugestellt, die auch an eine Komponente gehen. Hierzu zählen Tastatur-, Maus-, Mausbewegungs-, Fokus- und Komponentenergebnisse. Die Implementierung der Ereignishandler erfolgt zweckmäßigerweise so, wie es im vierten Entwurfsmuster in [Kapitel 18](#) vorgestellt wurde.

23.2 Entwicklung einer 7-Segment-Anzeige

- 23.2 Entwicklung einer 7-Segment-Anzeige
 - 23.2.1 Anforderungen
 - 23.2.2 Bildschirmanzeige
 - 23.2.3 Ereignisbehandlung

23.2.1 Anforderungen

In diesem Abschnitt wollen wir uns ein konkretes Beispiel zur Komponentenentwicklung ansehen. Dazu soll eine einstellige 7-Segment-Anzeige entwickelt werden, die in der Lage ist, die Ziffern 0 bis 9 darzustellen. Des weiteren sollen folgende Eigenschaften realisiert werden:

- Die Anzeige soll in der Größe skalierbar sein und sich zu allen Layoutmanagern kompatibel verhalten.
- Die 7-Segment-Anzeige soll auf Mausklicks reagieren. Nach einem Mausklick soll sie den Fokus erhalten. Nach Drücken der linken Maustaste soll der Anzeigewert herunter- und nach Drücken der rechten Maustaste heraufgezählt werden. Wird während des Mausklicks die `[UMSCHALT]`-Taste gedrückt, so soll lediglich der Fokus zugewiesen werden, der Wert aber unverändert bleiben.
- Der Anzeigewert soll durch Drücken der Tasten `[+]` und `[-]` herauf- bzw. heruntergezählt werden. Wird eine Zifferntaste gedrückt, so soll die Anzeige auf diesen Wert gesetzt werden.
- Eine Anzeige, die den Fokus hat, soll visuell von einer ohne Fokus unterscheidbar sein. Der Fokus soll weiterhin - wie unter Windows üblich - mit Hilfe der Tasten `[TAB]` und `[UMSCHALT]+[TAB]` von einem Dialogelement zum nächsten weitergereicht werden können.

23.2.2 Bildschirmanzeige

Die Architektur unserer Anzeigekomponente ist denkbar einfach. Wir definieren dazu eine neue Klasse `Segment7`, die aus `Canvas` abgeleitet wird. `Segment7` besitzt eine Membervariable `digit`, die den aktuellen Anzeigewert speichert. Dieser kann mit den öffentlichen Methoden `getValue` und `setValue` abgefragt bzw. gesetzt werden. Die Klasse bekommt zwei Konstruktoren, die es erlauben, den Anzeigewert wahlweise bei der Instanzierung zu setzen oder die Voreinstellung 0 zu verwenden.

`Segment7` überlagert die Methoden `getPreferredSize` und `getMinimumSize` der Klasse `Component`, um den Layoutmanagern die gewünschte Größe mitzuteilen:

```

public Dimension getPreferredSize()
public Dimension getMinimumSize()
    
```

Beide Methoden liefern ein Objekt der Klasse `Dimension`, also ein rechteckiges Element mit einer Höhe und Breite. `getPreferredSize` teilt dem Layoutmanager mit, welches die *gewünschte* Größe der Komponente ist, und `getMinimumSize` gibt an, welches die kleinste akzeptable Größe ist. Der Layoutmanager `FlowLayout` beispielsweise verwendet `getPreferredSize`, um die Größe der Komponente zu bestimmen. `GridLayout` gibt die Größe selbst vor und paßt sie an die Gitterelemente an. Durch Aufruf von `pack` kann allerdings auch `GridLayout` dazu veranlaßt werden, die gewünschte Größe der Komponenten abzufragen und zur Dimensionierung des Fensters (und damit letztlich zur Dimensionierung der Einzelkomponenten) zu verwenden. Seit dem JDK 1.1 gibt es noch eine dritte Methode `getMaximumSize`, mit der die Komponente ihre maximale Größe mitteilen kann. Sie wird in unserem Beispiel nicht benötigt.

Zur Darstellung der Leuchtdiodenanzeige auf dem Bildschirm wird die Methode `paint` überlagert; in ihr befindet sich die Logik zur Darstellung der sieben Segmente (siehe [Abbildung 23.1](#)). In `Segment7` wurden dazu drei Arrays `digits`, `polysx` und `polysy` definiert, die die Belegung der Segmente für jede einzelne Ziffer darstellen und die Eckpunkte jedes einzelnen Segments vorgeben.

`paint` verwendet das Array `digits`, um herauszufinden, welche Segmente zur Darstellung der aktuellen Ziffer verwendet werden. Für jedes der beteiligten Segmente wird dann aus den Arrays `polysx` und `polysy` das passende Polygon gebildet und mit `fillPolygon` angezeigt. Als interne Berechnungseinheit werden zwei Parameter `dx` und `dy` verwendet, die beim Aufruf von `paint` aus dem für die Komponente verfügbaren Platz berechnet werden.

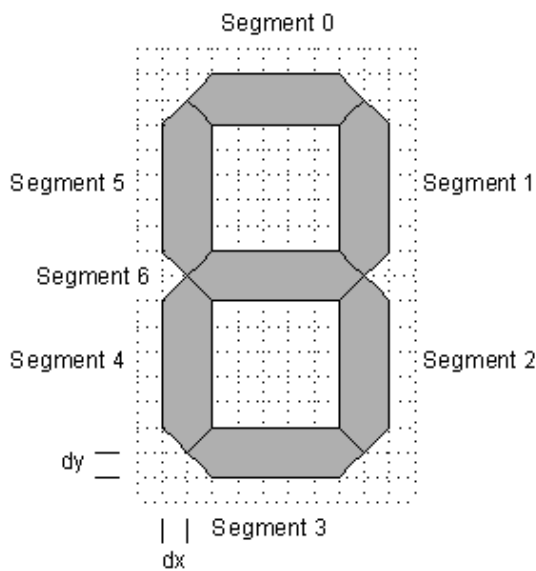


Abbildung 23.1: Der Aufbau der 7-Segment-Anzeige

23.2.3 Ereignisbehandlung

Wie in [Kapitel 18](#) erwähnt, erfolgt die Ereignisbehandlung in selbstdefinierten Komponenten üblicherweise auf der Basis des vierten vorgestellten Architekturmodells. Bei diesem werden die Ereignisse nicht durch registrierte Listener-Klassen bearbeitet, sondern durch Überlagern der Methoden `process...Event` der Klasse `Component`.

Damit die Ereignisse tatsächlich an diese Methoden weitergegeben werden, müssen sie zuvor durch Aufruf von `enableEvents` und Übergabe der zugehörigen Ereignismaske aktiviert werden. Da wir Component-, Focus-, Key- und Mouse-Ereignisse behandeln wollen, rufen wir `enableEvents` mit den Konstanten `AWTEvent.COMPONENT_EVENT_MASK`, `AWTEvent.FOCUS_EVENT_MASK`, `AWTEvent.MOUSE_EVENT_MASK` und `AWTEvent.KEY_EVENT_MASK` auf. Beim Überlagern dieser Methoden sollte in jedem Fall der entsprechende Ereignishandler der Superklasse aufgerufen werden, um die korrekte Standard-Ereignisbehandlung sicherzustellen.

Die Reaktion auf Mausklicks wird durch Überlagern der Methode `processMouseEvent` realisiert. Hier wird zunächst überprüft, ob es sich um ein `MOUSE_PRESSED`-Ereignis handelt, also ob eine der Maustasten gedrückt wurde. Ist dies der Fall, wird dem Anzeigeelement durch Aufruf von `requestFocus` der Eingabefokus zugewiesen.

Anschließend wird überprüft, ob die `[UMSCHALT]`-Taste gedrückt wurde. Ist das der Fall, wird die Ereignisbehandlung beendet, andernfalls wird der Anzeigewert hoch- bzw. heruntergezählt, je nachdem, ob die rechte oder linke Maustaste gedrückt wurde. Am Ende von `processMouseEvent` wird in jedem Fall `super.processMouseEvent` aufgerufen, um sicherzustellen, daß die normale Ereignisbehandlung aufgerufen wird.

Ein selbstdefiniertes Dialogelement bekommt nicht automatisch den Fokus zugewiesen, wenn mit der Maus darauf geklickt wird. Statt dessen muß es selbst auf Mausklicks reagieren und sich - wie oben beschrieben - durch Aufruf von `requestFocus` selbst den Fokus zuweisen. Bei jeder Fokusänderung wird ein Focus-Event ausgelöst, das wir durch Überlagern der Methode `processFocusEvent` bearbeiten. Hier unterscheiden wir zunächst, ob es sich um ein `FOCUS_GAINED`- oder `FOCUS_LOST`-Ereignis handelt und setzen eine interne Statusvariable `hasfocus` entsprechend. Diese wird nach dem anschließenden Aufruf von `repaint` verwendet, um in `paint` durch Modifikation der Anzeigefarbe ein visuelles Feedback zu geben. Hat ein Element den Fokus, so ist die Farbe der Anzeigesegmente gelb, andernfalls rot.

Seit dem JDK 1.1 gibt es einen Mechanismus, der es erlaubt, mit den Tasten `[TAB]` und `[UMSCHALT]+[TAB]` zwischen den Eingabefeldern eines Dialogs zu wechseln. Genauer gesagt wird dadurch der Fokus an das nächste Element weiter- bzw. zum vorigen zurückgegeben. Da diese Vorgehensweise nicht bei jedem Dialogelement sinnvoll ist, kann das Dialogelement sie durch Überlagern der Methode `isFocusTraversable` selbst bestimmen. Liefert `isFocusTraversable` den Rückgabewert `true`, so nimmt das Objekt an der `[TAB]`-Behandlung teil, andernfalls nicht. Die Klasse `Segment7` überlagert `isFocusTraversable` und gibt `true` zurück. So kann mit `[TAB]` und `[UMSCHALT]+[TAB]` wie besprochen zwischen den 7-Segment-Anzeigen gewechselt werden.

Ein Dialogelement enthält nur dann Tastatureingaben, wenn es den Fokus hat. Durch den zuvor beschriebenen Mechanismus des Aufrufs von `requestFocus` stellen wir sicher, daß nach einem Mausklick bzw. nach dem Wechsel des Fokus mit `[TAB]` und `[UMSCHALT]+[TAB]` Tastaturereignisse an das Element gesendet werden. Diese werden durch Überlagern der Methode `processKeyEvent` behandelt. Wir überprüfen darin zunächst, ob das Ereignis vom Typ `KEY_PRESSED` ist und besorgen dann mit `getKeyChar` den Wert der gedrückten Taste. Ist er '+', so wird der Anzeigewert um 1 erhöht, bei '-' entsprechend verringert. Wurde eine der Zifferntasten gedrückt, so erhält das Anzeigeelement diesen Wert. Anschließend wird durch Aufruf von `repaint` die Anzeige neu gezeichnet.

Ein Component-Ereignis brauchen wir in unserem Beispiel nur, damit wir dem Dialogelement unmittelbar nach der Anzeige auf dem Bildschirm den

Fokus zuweisen können. Dazu überlagern wir die Methode [processComponentEvent](#) und überprüfen, ob das Ereignis vom Type [COMPONENT_SHOWN](#) ist. In diesem Fall wird [requestFocus](#) aufgerufen, andernfalls passiert nichts.

Damit ist die Konstruktion der Komponente auch schon abgeschlossen. Durch die Definition von [getPreferredSize](#) und [getMinimumSize](#) und die automatische Skalierung in der [paint](#)-Methode verhält sich unsere neue Komponente so, wie es die Layoutmanager von ihr erwarten. Daher kann sie wie eine vordefinierte Komponente verwendet werden. Hier ist der Quellcode von [Segment7](#):

[Segment7.java](#)

```
001 /* Segment7.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class Segment7
007 extends Canvas
008 {
009     private int digit;
010     private boolean hasfocus;
011     private int polysx[][] = {
012         { 1, 2, 8, 9, 8, 2},      //Segment 0
013         { 9,10,10, 9, 8, 8},      //Segment 1
014         { 9,10,10, 9, 8, 8},      //Segment 2
015         { 1, 2, 8, 9, 8, 2},      //Segment 3
016         { 1, 2, 2, 1, 0, 0},      //Segment 4
017         { 1, 2, 2, 1, 0, 0},      //Segment 5
018         { 1, 2, 8, 9, 8, 2},      //Segment 6
019     };
020     private int polysy[][] = {
021         { 1, 0, 0, 1, 2, 2},      //Segment 0
022         { 1, 2, 8, 9, 8, 2},      //Segment 1
023         { 9,10,16,17,16,10},      //Segment 2
024         {17,16,16,17,18,18},      //Segment 3
025         { 9,10,16,17,16,10},      //Segment 4
026         { 1, 2, 8, 9, 8, 2},      //Segment 5
027         { 9, 8, 8, 9,10,10},      //Segment 6
028     };
029     private int digits[][] = {
030         {1,1,1,1,1,1,0},          //Ziffer 0
031         {0,1,1,0,0,0,0},          //Ziffer 1
032         {1,1,0,1,1,0,1},          //Ziffer 2
033         {1,1,1,1,0,0,1},          //Ziffer 3
034         {0,1,1,0,0,1,1},          //Ziffer 4
035         {1,0,1,1,0,1,1},          //Ziffer 5
036         {1,0,1,1,1,1,1},          //Ziffer 6
037         {1,1,1,0,0,0,0},          //Ziffer 7
038         {1,1,1,1,1,1,1},          //Ziffer 8
039         {1,1,1,1,0,1,1},          //Ziffer 9
040     };
041
042     public Segment7()
043     {
044         this(0);
045     }
046
047     public Segment7(int digit)
048     {
049         super();
050         this.digit = digit;
051         this.hasfocus = false;
052         enableEvents(AWTEvent.COMPONENT_EVENT_MASK);
053         enableEvents(AWTEvent.FOCUS_EVENT_MASK);
054         enableEvents(AWTEvent.MOUSE_EVENT_MASK);
055         enableEvents(AWTEvent.KEY_EVENT_MASK);
056     }
057
058     public Dimension getPreferredSize()
```



```

059     {
060         return new Dimension(5*10,5*18);
061     }
062
063     public Dimension getMinimumSize()
064     {
065         return new Dimension(1*10,1*18);
066     }
067
068     public boolean isFocusTraversable()
069     {
070         return true;
071     }
072
073     public void paint(Graphics g)
074     {
075         Color darkred  = new Color(127,0,0);
076         Color lightred = new Color(255,0,0);
077         Color yellow   = new Color(255,255,0);
078         //dx und dy berechnen
079         int dx = getSize().width / 10;
080         int dy = getSize().height / 18;
081         //Hintergrund
082         g.setColor(darkred);
083         g.fillRect(0,0,getSize().width,getSize().height);
084         //Segmente
085         if (hasfocus) {
086             g.setColor(yellow);
087         } else {
088             g.setColor(lightred);
089         }
090         for (int i=0; i < 7; ++i) { //alle Segmente
091             if (digits[digit][i] == 1) {
092                 Polygon poly = new Polygon();
093                 for (int j = 0; j < 6; ++j) { //alle Eckpunkte
094                     poly.addPoint(dx*polysx[i][j],dy*polysy[i][j]);
095                 }
096                 g.fillPolygon(poly);
097             }
098         }
099         //Trennlinien
100         g.setColor(darkred);
101         g.drawLine(0,0,dx*10,dy*10);
102         g.drawLine(0,8*dy,10*dx,18*dy);
103         g.drawLine(0,10*dy,10*dx,0);
104         g.drawLine(0,18*dy,10*dx,8*dy);
105     }
106
107     public int getValue()
108     {
109         return digit;
110     }
111
112     public void setValue(int value)
113     {
114         digit = value % 10;
115     }
116
117     protected void processComponentEvent(ComponentEvent event)
118     {
119         if (event.getID() == ComponentEvent.COMPONENT_SHOWN) {
120             requestFocus();
121         }
122         super.processComponentEvent(event);
123     }

```

```

124
125 protected void processFocusEvent(FocusEvent event)
126 {
127     if (event.getID() == FocusEvent.FOCUS_GAINED) {
128         hasfocus = true;
129         repaint();
130     } else if (event.getID() == FocusEvent.FOCUS_LOST) {
131         hasfocus = false;
132         repaint();
133     }
134     super.processFocusEvent(event);
135 }
136
137 protected void processMouseEvent(MouseEvent event)
138 {
139     if (event.getID() == MouseEvent.MOUSE_PRESSED) {
140         requestFocus();
141         if (!event.isShiftDown()) {
142             if (event.isMetaDown()) {
143                 setValue(getValue() + 1); //increment by 1
144             } else {
145                 setValue(getValue() + 9); //decrement by 1
146             }
147         }
148         repaint();
149     }
150     super.processMouseEvent(event);
151 }
152
153 protected void processKeyEvent(KeyEvent event)
154 {
155     if (event.getID() == KeyEvent.KEY_PRESSED) {
156         char key = event.getKeyChar();
157         if (key >= '0' && key <= '9') {
158             setValue(key - '0');
159             repaint();
160         } else if (key == '+') {
161             setValue(getValue() + 1); //increment by 1
162             repaint();
163         } else if (key == '-') {
164             setValue(getValue() + 9); //decrement by 1
165             repaint();
166         }
167     }
168     super.processKeyEvent(event);
169 }
170 }

```

Listing 23.1: Eine 7-Segment-Anzeige

23.3 Einbinden der Komponente

- [23.3 Einbinden der Komponente](#)

Auch die Einbindung dieser neuen Komponente ist sehr einfach und kann in Anlehnung an die vorherigen Beispiele durch Aufruf von [add](#) in einem [Container](#) erfolgen:

[Listing2302.java](#)

```

001 /* Listing2302.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 class MyDialog2302
007 extends Dialog
008 implements ActionListener
009 {
010     public MyDialog2302(Frame parent)
011     {
012         super(parent, "MyDialog2302", true);
013         setBounds(100, 100, 400, 300);
014         setBackground(Color.lightGray);
015         setLayout(new BorderLayout());
016         Panel panel = new Panel();
017         customizeLayout(panel);
018         add("Center", panel);
019         //Ende-Button
020         Button button = new Button("Ende");
021         button.addActionListener(this);
022         add("South", button);
023         pack();
024         //Window-Ereignisse
025         addWindowListener(
026             new WindowAdapter() {
027                 public void windowClosing(WindowEvent event)
028                 {
029                     endDialog();
030                 }
031             }
032         );
033     }
034
035     private void customizeLayout(Panel panel)
036     {
037         panel.setLayout(new FlowLayout());
038         panel.add(new Segment7(0));
039         panel.add(new Segment7(1));
040         panel.add(new Segment7(2));
041         panel.add(new Segment7(3));
042         panel.add(new Segment7(4));
043         panel.add(new Segment7(5));
044         panel.add(new Segment7(6));
045         panel.add(new Segment7(7));
046         panel.add(new Segment7(8));
047         panel.add(new Segment7(9));
048     }
049
050     public void actionPerformed(ActionEvent event)
051     {
052         String cmd = event.getActionCommand();

```

```

053     if (cmd.equals("Ende")) {
054         endDialog();
055     }
056 }
057
058 void endDialog()
059 {
060     setVisible(false);
061     dispose();
062     ((Window)getParent()).toFront();
063     getParent().requestFocus();
064 }
065 }
066
067 public class Listing2302
068 extends Frame
069 implements ActionListener
070 {
071     public static void main(String[] args)
072     {
073         Listing2302 wnd = new Listing2302();
074         wnd.setSize(300,200);
075         wnd.setVisible(true);
076     }
077
078     public Listing2302()
079     {
080         super("Listing2302");
081         setBackground(Color.lightGray);
082         setLayout(new FlowLayout());
083         //Dialog-Button
084         Button button = new Button("Dialog");
085         button.addActionListener(this);
086         add(button);
087         //Ende-Button
088         button = new Button("Ende");
089         button.addActionListener(this);
090         add(button);
091         //Window-Ereignisse
092         addWindowListener(
093             new WindowAdapter() {
094                 public void windowClosing(WindowEvent event)
095                 {
096                     setVisible(false);
097                     dispose();
098                     System.exit(0);
099                 }
100             }
101         );
102     }
103
104     public void actionPerformed(ActionEvent event)
105     {
106         String cmd = event.getActionCommand();
107         if (cmd.equals("Dialog")) {
108             MyDialog2302 dlg = new MyDialog2302(this);
109             dlg.setVisible(true);
110         } else if (cmd.equals("Ende")) {
111             setVisible(false);
112             dispose();
113             System.exit(0);
114         }
115     }
116 }

```

Listing 23.2: Einbinden der 7-Segment-Anzeige

Das Ergebnis kann sich sehen lassen:



Abbildung 23.2: Ein Beispiel für die Anwendung der 7-Segment-Anzeige

Wir wollen nun die Entwicklung von Dialogen abschließen und uns in [Kapitel 24](#) der Einbindung von Bildern und der Entwicklung von Animationen zuwenden.

23.4 Zusammenfassung

- [23.4 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Bedeutung der Klasse [Canvas](#) zur Realisierung benutzerdefinierter Komponenten
- Die Architektur einer Komponente zur Darstellung einer 7-Segment-Anzeige
- Die Implementierung der Methoden [getPreferredSize](#) und [getMinimumSize](#)
- Überlagern von [paint](#) zur Realisierung der Bildschirmanzeige
- Überlagern der Methoden [processXYZEvent](#) zur Reaktion auf GUI-Ereignisse und Aktivieren des Ereignisversands mit [enableEvents](#)
- Die Besonderheiten bei der Bearbeitung der Focus-Events
- Einbindung einer benutzerdefinierten Komponente in einen Dialog

Kapitel 24

Bitmaps und Animation

- [24 Bitmaps und Animation](#)
 - [24.1 Bitmaps](#)
 - [24.1.1 Laden und Anzeigen einer Bitmap](#)
 - [Die Klasse MediaTracker](#)
 - [24.1.2 Entwicklung einer eigenen Bitmap-Komponente](#)
 - [24.2 Animation](#)
 - [24.2.1 Prinzipielle Vorgehensweise](#)
 - [Die repaint-Schleife](#)
 - [Verwendung von Threads](#)
 - [24.2.2 Abspielen einer Folge von Bitmaps](#)
 - [24.2.3 Animation mit Grafikprimitiven](#)
 - [24.2.4 Reduktion des Bildschirmflackerns](#)
 - [Bildschirm nicht löschen](#)
 - [Nur den wirklich benötigten Teil des Bildschirms löschen](#)
 - [Doppelpufferung](#)
 - [24.3 Zusammenfassung](#)

24.1 Bitmaps

- [24.1 Bitmaps](#)
 - [24.1.1 Laden und Anzeigen einer Bitmap](#)
 - [Die Klasse MediaTracker](#)
 - [24.1.2 Entwicklung einer eigenen Bitmap-Komponente](#)

24.1.1 Laden und Anzeigen einer Bitmap

Das Anzeigen einer Bitmap kann in zwei Schritte unterteilt werden:

- das Laden der Bitmap von einem externen Speichermedium oder aus dem Netz
- die eigentliche Ausgabe auf den Bildschirm

Das Laden erfolgt mit der Methode [getImage](#), die eine Instanz der Klasse [Image](#) zurückgibt. Das [Image](#)-Objekt kann dann mit der Methode [drawImage](#) der Klasse [Graphics](#) angezeigt werden.

[getImage](#) gibt es in verschiedenen Varianten, die sich dadurch unterscheiden, aus welcher Quelle sie die Bitmap-Daten laden. In einer Java-Applikation wird in der Regel die Methode [getImage](#) aus der Klasse [Toolkit](#) verwendet. Sie erwartet den Namen einer lokalen Datei als Parameter:

```

java.awt.Toolkit
public Image getImage(String filename)

```

[getImage](#) versteht in der aktuellen Version des AWT die beiden Bitmap-Typen [gif](#) und [jpeg](#). Andere Grafikformate, wie etwa das unter Windows gebräuchliche [bmp](#)-Format, werden nicht unterstützt, sondern müssen bei Bedarf konvertiert werden.

Neben den [getImage](#)-Methoden gibt es seit dem JDK 1.2 auch zwei Methoden mit dem JDK1.1/1.2 Namen [createImage](#) in der Klasse [Toolkit](#):

```

java.awt.Toolkit
public abstract Image createImage(String filename)

public abstract Image createImage(URL url)

```

Sie laden ein Image bei jeder Verwendung neu und führen (im Gegensatz zu [getImage](#)) kein Caching des Bildes durch. Die JDK-Dokumentation empfiehlt sie gegenüber [getImage](#), weil bei deren Verwendung Speicherlecks durch das unbegrenzte Zwischenspeichern der Bilddaten entstehen können.

Das [Toolkit](#) für die aktuelle Umgebung kann mit der Methode [getToolkit](#) der Klasse [Component](#) beschafft werden:

```

java.awt.Component
public Toolkit getToolkit()

```

Der gesamte Code zum Laden einer Bitmap [duke.gif](#) sieht daher so aus:

```

001 Image img;
002 img = getToolkit().getImage("duke.gif");

```

Listing 24.1: Laden einer Bitmap-Datei

Um das [Image](#) anzuzeigen, kann die Methode [drawImage](#) der Klasse [Graphics](#) aufgerufen werden:

```

java.awt.Graphics
public boolean drawImage(
    Image img, int x, int y, ImageObserver observer
)

```


[drawImage](#) gibt es in unterschiedlichen Ausprägungen. Die hier vorgestellte Variante erwartet das anzuzeigende [Image](#)-Objekt und die Position (x,y) , an der die linke obere Ecke der Bitmap platziert werden soll. Das zusätzlich angegebene Objekt [observer](#) dient zur Übergabe eines [ImageObserver](#)-Objektes, mit dem der Ladezustand der Bitmaps überwacht werden kann. Hier kann der [this](#)-Zeiger, also eine Referenz auf das Fensterobjekt, übergeben werden. Eine weitere Variante von [drawImage](#), die ein Bild sogar skalieren und spiegeln kann, wurde in [Kapitel 2](#) vorgestellt. Wir wollen hier nicht weiter auf Details eingehen.

Hinweis

Das folgende Listing ist ein einfaches Beispiel für das Laden und Anzeigen der Bitmap [duke.gif](#). Alle erforderlichen Aktionen erfolgen innerhalb von [paint](#):

Beispiel

```
001 public void paint(Graphics g)
002 {
003     Image img;
004     img = getToolkit().getImage("duke.gif");
005     g.drawImage(img,40,40,this);
006 }
```

Listing 24.2: Laden und Anzeigen einer Bitmap

Die Ausgabe des Programms ist:

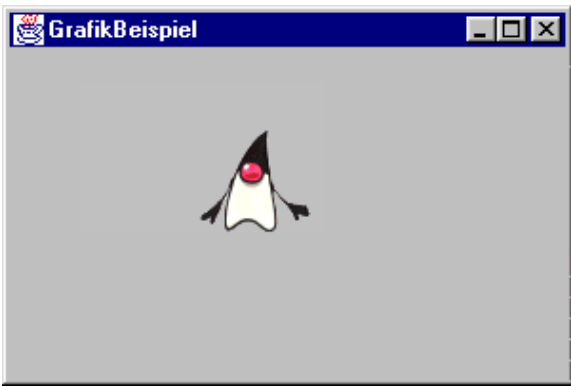


Abbildung 24.1: Laden und Anzeigen einer Bitmap

Die gewählte Vorgehensweise ist nicht besonders effizient, denn die Bitmap wird bei jedem Aufruf von [paint](#) neu geladen. Besser ist es, die benötigten Bitmaps einmal zu laden und dann im Speicher zu halten. Obwohl man vermuten könnte, daß dies die Ladezeit des Fensters unannehmbar verlängern würde, ist der Konstruktor der Klasse eine gute Stelle dafür. Der Aufruf von [getImage](#) lädt die Bitmap nämlich noch nicht, sondern bereitet das Laden nur vor. Der eigentliche Ladevorgang erfolgt erst, wenn die Bitmap beim Aufruf von [drawImage](#) tatsächlich benötigt wird.

Tip

Die Klasse MediaTracker

Manchmal kann es sinnvoll sein, den tatsächlichen Ladevorgang des Bildes abzuwarten, bevor im Programm fortgefahren wird. Wird zum Beispiel die Größe der Bitmap benötigt, um sie korrekt auf dem Bildschirm anordnen oder skalieren zu können, muß das Programm warten, bis das [Image](#) vollständig erzeugt ist.

Für diese Zwecke steht die Klasse [MediaTracker](#) zur Verfügung, die das Laden eines oder mehrerer Bilder überwacht. Dazu wird zunächst eine Instanz der Klasse angelegt:

[java.awt.MediaTracker](#)

```
public MediaTracker(Component comp)
```

Als Komponente wird der [this](#)-Zeiger des aktuellen Fensters übergeben. Anschließend werden durch Aufruf von [addImage](#) alle Bilder, deren Ladevorgang überwacht werden soll, an den [MediaTracker](#) übergeben:

[java.awt.MediaTracker](#)

```
public void addImage(Image img, int id)
```

Der zusätzlich übergebene Parameter [id](#) kann dazu verwendet werden, einen Namen zu vergeben, unter dem auf das [Image](#) zugegriffen werden kann. Zusätzlich bestimmt er die Reihenfolge, in der die Images geladen werden. Bitmaps mit kleineren Werten werden zuerst geladen.

Der [MediaTracker](#) bietet eine Reihe von Methoden, um den Ladezustand der Bilder zu überwachen. Wir wollen hier nur die Methode

[waitForAll](#) betrachten. Sie wartet, bis alle Images vollständig geladen sind:

[java.awt.MediaTracker](#)

```
public void waitForAll()  
    throws InterruptedException
```

Nach Abschluß des Ladevorgangs sendet [waitForAll](#) eine Ausnahme des Typs [InterruptedException](#).

Das vollständige Beispielprogramm zur Anzeige von `duke.gif` sieht nun so aus:

Beispiel

[Listing2403.java](#)

```
001 /* Listing2403.java */  
002  
003 import java.awt.*;  
004 import java.awt.event.*;  
005  
006 public class Listing2403  
007 extends Frame  
008 {  
009     private Image img;  
010  
011     public static void main(String[] args)  
012     {  
013         Listing2403 wnd = new Listing2403();  
014     }  
015  
016     public Listing2403()  
017     {  
018         super("Listing2403");  
019         setBackground(Color.lightGray);  
020         setSize(250,150);  
021         setVisible(true);  
022         //WindowListener  
023         addWindowListener(  
024             new WindowAdapter() {  
025                 public void windowClosing(WindowEvent event)  
026                 {  
027                     System.exit(0);  
028                 }  
029             }  
030         );  
031         //Bild laden  
032         img = getToolkit().getImage("duke.gif");  
033         MediaTracker mt = new MediaTracker(this);  
034         mt.addImage(img, 0);  
035         try {  
036             //Warten, bis das Image vollständig geladen ist,  
037             mt.waitForAll();  
038         } catch (InterruptedException e) {  
039             //nothing  
040         }  
041         repaint();  
042     }  
043  
044     public void paint(Graphics g)  
045     {  
046         if (img != null) {  
047             g.drawImage(img,40,40,this);  
048         }  
049     }  
050 }
```

Listing 24.3: Programm zum Laden und Anzeigen einer Bitmap

24.1.2 Entwicklung einer eigenen Bitmap-Komponente

Ein schönes Beispiel für die Verwendung von Bitmaps ist die Konstruktion einer Komponente [BitmapComponent](#), die in Dialogboxen zur Anzeige von Bitmaps verwendet werden kann. Die Verwendung soll dabei so einfach wie möglich sein, d.h., außer der Übergabe des Dateinamens an den Konstruktor soll kein zusätzlicher Aufwand entstehen.

Zur Konstruktion der Komponente gehen wir in folgenden Schritten vor:

- Ableiten einer neuen Klasse [BitmapComponent](#) aus [Canvas](#).
- Überlagern des Konstruktors, um dort die Bitmap zu laden.
- Überlagern von [paint](#), um das Image auf dem Bildschirm auszugeben.
- Überlagern von [getPreferredSize](#) und [getMinimumSize](#), um dem Layoutmanager die Größe der Komponente mitzuteilen.

Nach diesen Ausführungen ist die Implementierung einfach:

[BitmapComponent.java](#)

```
001 /* BitmapComponent.java */
002
003 import java.awt.*;
004
005 class BitmapComponent
006 extends Canvas
007 {
008     private Image img;
009
010     public BitmapComponent(String fname)
011     {
012         img = getToolkit().getImage(fname);
013         MediaTracker mt = new MediaTracker(this);
014
015         mt.addImage(img, 0);
016         try {
017             //Warten, bis das Image vollständig geladen ist,
018             //damit getWidth() und getHeight() funktionieren
019             mt.waitForAll();
020         } catch (InterruptedException e) {
021             //nothing
022         }
023     }
024
025     public void paint(Graphics g)
026     {
027         g.drawImage(img,1,1,this);
028     }
029
030     public Dimension getPreferredSize()
031     {
032         return new Dimension(
033             img.getWidth(this),
034             img.getHeight(this)
035         );
036     }
037
038     public Dimension getMinimumSize()
039     {
040         return new Dimension(
041             img.getWidth(this),
042             img.getHeight(this)
043         );
044     }
045 }
```

Listing 24.4: Eine Komponente zum Anzeigen einer Bitmap

Bei diesem Beispiel ist die Verwendung des [MediaTrackers](#) obligatorisch. Andernfalls könnte es passieren, daß zum Zeitpunkt des Aufrufs von [getPreferredSize](#) oder [getMinimumSize](#) die Bitmap noch nicht geladen ist und die Methoden [getWidth](#) und [getHeight](#) 0 zurückgeben. Dies würde dann dazu führen, daß der Layoutmanager den benötigten Platz fehlerhaft berechnet und die Komponente falsch oder gar nicht angezeigt wird.

Warnung

Die Einbindung von [BitmapComponent](#) in einen Dialog erfolgt analog zur Einbindung jeder anderen Komponente durch Aufruf der Methode [add](#) der Klasse [Component](#). Das folgende Listing gibt ein Beispiel für die Einbindung der neuen Komponente:

Beispiel

[Listing2405.java](#)

```
001 /* Listing2405.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2405
007 extends Frame
008 {
009     public static void main(String[] args)
010     {
011         Listing2405 wnd = new Listing2405();
012     }
013
014     public Listing2405()
015     {
016         super("Listing2405");
017         setBackground(Color.lightGray);
018         setSize(250,150);
019         setVisible(true);
020         //Hinzufügen der Komponenten
021         setLayout(new GridLayout(2,2));
022         add(new BitmapComponent("duke.gif"));
023         add(new BitmapComponent("duke.gif"));
024         add(new BitmapComponent("duke.gif"));
025         add(new BitmapComponent("duke.gif"));
026         pack();
027         //WindowListener
028         addWindowListener(
029             new WindowAdapter() {
030                 public void windowClosing(WindowEvent event)
031                 {
032                     System.exit(0);
033                 }
034             }
035         );
036     }
037 }
```

Listing 24.5: Verwenden der Bitmap-Komponente

Die Ausgabe des Programms ist:

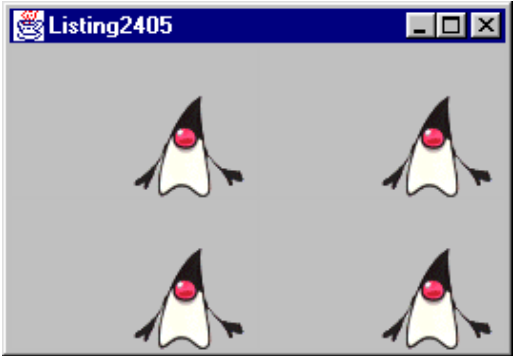


Abbildung 24.2: Verwendung von BitmapComponent

24.2 Animation

- [24.2 Animation](#)
 - [24.2.1 Prinzipielle Vorgehensweise](#)
 - [Die repaint-Schleife](#)
 - [Verwendung von Threads](#)
 - [24.2.2 Abspielen einer Folge von Bitmaps](#)
 - [24.2.3 Animation mit Grafikprimitiven](#)
 - [24.2.4 Reduktion des Bildschirmflackerns](#)
 - [Bildschirm nicht löschen](#)
 - [Nur den wirklich benötigten Teil des Bildschirms löschen](#)
 - [Doppelpufferung](#)

24.2.1 Prinzipielle Vorgehensweise

Das Darstellen einer Animation auf dem Bildschirm ist im Prinzip nichts anderes als die schnell aufeinanderfolgende Anzeige einer Sequenz von Einzelbildern. Die Bildfolge erscheint dem menschlichen Auge aufgrund seiner Trägheit als zusammenhängende Bewegung.

Obwohl die prinzipielle Vorgehensweise damit klar umrissen ist, steckt die Tücke bei der Darstellung von animierten Bildsequenzen im Detail. Zu den Problemen, die in diesem Zusammenhang zu lösen sind, gehören:

- Für die Anzeige der Einzelbilder muß das richtige *Timing* gewählt werden. Werden zu wenig Bilder je Zeiteinheit angezeigt, erscheint das Ergebnis ruckelig. Werden zu viele gezeigt, kann es sein, daß [paint](#)-Events verlorengehen und Teile des Bildes nicht korrekt angezeigt werden.
- Wird zur Anzeige der Einzelbilder der normale [paint](#)-/[repaint](#)-Mechanismus von Java verwendet, wird die Darstellung durch ein starkes *Flackern* gestört. Wir werden eine Reihe von Techniken kennenlernen, mit denen dieses Flackern verhindert werden kann.
- Die Darstellung der Einzelbilder darf nicht die *Interaktivität* des Programms beeinträchtigen. Vor allem darf die Nachrichtenschleife nicht blockiert werden. Wir werden lernen, wie man diese Probleme mit Hilfe separater Threads umgehen kann.

All dies sind Standardprobleme, die vom Programmierer bei der Entwicklung von Animationen zu lösen sind. Wir werden feststellen, daß Java dafür durchweg brauchbare Lösungen zu bieten hat und die Programmierung kleiner Animationen recht einfach zu realisieren ist.

Die repaint-Schleife

Das Grundprinzip einer Animation besteht darin, in einer Schleife die Methode [repaint](#) wiederholt aufzurufen. Ein Aufruf von [repaint](#) führt dazu, daß die [paint](#)-Methode aufgerufen wird, und innerhalb von [paint](#) generiert die Anwendung dann die für das aktuelle Einzelbild benötigte Bildschirmausgabe.

[paint](#) muß sich also merken (oder mitgeteilt bekommen), welches Bild bei welchem Aufruf erzeugt werden soll. Typischerweise wird dazu ein Schleifenzähler verwendet, der das gerade anzuzeigende Bild bezeichnet. Nach dem Ausführen der Ausgabeanweisungen terminiert [paint](#), und der Aufrufer wartet eine bestimmte Zeitspanne. Dann zählt er den Bildzähler hoch und führt den nächsten Aufruf von [repaint](#) durch. Dies setzt sich so lange fort, bis die Animation beendet ist oder das Programm abgebrochen wird.

Das folgende Listing stellt eines der einfachsten Beispiele für eine Grafikanimation dar:

Beispiel

[Listing2406.java](#)

```

001 /* Listing2406.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2406
007 extends Frame
008 {
009     int cnt = 0;
010
011     public static void main(String[] args)
012     {

```

```

013 Listing2406 wnd = new Listing2406();
014 wnd.setSize(250,150);
015 wnd.setVisible(true);
016 wnd.startAnimation();
017 }
018
019 public Listing2406()
020 {
021     super("Listing2406");
022     setBackground(Color.lightGray);
023     //WindowListener
024     addWindowListener(
025         new WindowAdapter() {
026             public void windowClosing(WindowEvent event)
027             {
028                 System.exit(0);
029             }
030         }
031     );
032 }
033
034 public void startAnimation()
035 {
036     while (true) {
037         repaint();
038     }
039 }
040
041 public void paint(Graphics g)
042 {
043     ++cnt;
044     g.drawString("Counter = "+cnt,10,50);
045     try {
046         Thread.sleep(1000);
047     } catch (InterruptedException e) {
048     }
049 }
050 }

```

Listing 24.6: Ein animierter Zähler

Das Programm öffnet ein Fenster und zählt in Sekundenabständen einen Zähler um eins hoch:

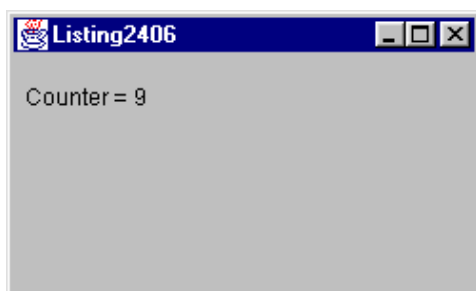


Abbildung 24.3: Ein animierter Zähler

Leider hat das Programm einen entscheidenden Nachteil. Die Animation selbst funktioniert zwar wunderbar, aber das Programm reagiert nur noch sehr schleppend auf Windows-Nachrichten. Wir wollen zunächst dieses Problem abstellen und uns ansehen, wie man die [repaint](#)-Schleife in einem eigenen [Thread](#) laufen läßt.

Warnung

Verwendung von Threads

Um die vorherige Version des Programms zu verbessern, sollte die [repaint](#)-Schleife in einem eigenen [Thread](#) laufen. Zusätzlich ist es erforderlich, die Zeitverzögerung aus [paint](#) herauszunehmen und statt dessen in die [repaint](#)-Schleife zu verlagern. So bekommt der Haupt-Thread des Animationsprogramms genügend Zeit, die Bildschirmausgabe durchzuführen, und kann andere [Events](#) bearbeiten. Daß in einem anderen [Thread](#) eine Endlosschleife läuft, merkt er nur noch daran, daß in regelmäßigen Abständen [repaint](#)-Ereignisse eintreffen.

Um das Programm auf die Verwendung mehrerer Threads umzustellen, sollte die Fensterklasse das Interface [Runnable](#) implementieren und eine Instanzvariable vom Typ [Thread](#) anlegen. Dann wird die Methode [startAnimation](#) so modifiziert, daß sie den neuen [Thread](#) instanziert und startet. Die eigentliche [repaint](#)-Schleife wird in die Methode [run](#) verlagert. Schließlich sollte beim Beenden des Programms auch der laufende [Thread](#) beendet werden. Hier ist die modifizierte Fassung:

[Listing2407.java](#)

```
001 /* Listing2407.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2407
007 extends Frame
008 implements Runnable
009 {
010     int cnt = 0;
011
012     public static void main(String[] args)
013     {
014         Listing2407 wnd = new Listing2407();
015         wnd.setSize(250,150);
016         wnd.setVisible(true);
017         wnd.startAnimation();
018     }
019
020     public Listing2407()
021     {
022         super("Listing2407");
023         setBackground(Color.lightGray);
024         //WindowListener
025         addWindowListener(
026             new WindowAdapter() {
027                 public void windowClosing(WindowEvent event)
028                 {
029                     System.exit(0);
030                 }
031             }
032         );
033     }
034
035     public void startAnimation()
036     {
037         Thread th = new Thread(this);
038         th.start();
039     }
040
041     public void run()
042     {
043         while (true) {
044             repaint();
045             try {
046                 Thread.sleep(1000);
047             } catch (InterruptedException e) {
048                 //nichts
049             }
050         }
051     }
052
053     public void paint(Graphics g)
054     {
055         ++cnt;
056         g.drawString("Counter = "+cnt,10,50);
057     }
058 }
```

Listing 24.7: Verwendung von Threads zur Animation

Das so modifizierte Programm erzeugt dieselbe Ausgabe wie das vorige, ist aber in der Lage, in der gewohnten Weise auf Ereignisse zu reagieren. Selbst wenn die Verzögerungsschleife ganz entfernt und der Hauptprozeß so pausenlos mit [repaint](#)-Anforderungen bombardiert würde, könnte das Programm noch normal beendet werden.

24.2.2 Abspielen einer Folge von Bitmaps

Eine der einfachsten und am häufigsten verwendeten Möglichkeiten, eine Animation zu erzeugen, besteht darin, die zur Darstellung erforderliche Folge von Bitmaps aus einer Reihe von Bilddateien zu laden. Jedem Einzelbild wird dabei ein [Image](#)-Objekt zugeordnet, das vor dem Start der Animation geladen wird. Alle Images liegen in einem Array oder einem anderen Container und werden in der [repaint](#)-Schleife nacheinander angezeigt.

Das folgende Programm speichert die 30 anzuzeigenden Einzelbilder in einem Array `arImg`, das nach dem Start des Programms komplett geladen wird. Da dieser Vorgang einige Sekunden dauern kann, zeigt das Programm den Ladefortschritt auf dem Bildschirm an:

Beispiel

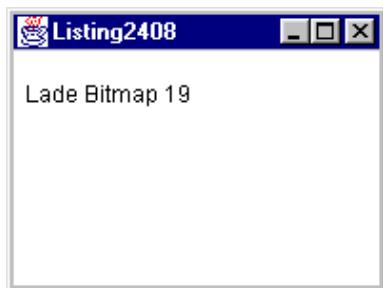


Abbildung 24.4: Die Ausgabe während des Ladevorgangs

Erst nach dem vollständigen Abschluß des Ladevorgangs, der mit einem [MediaTracker](#) überwacht wird, beginnt die eigentliche Animation. Die ganzzahlige Instanzvariable `actimage` dient als Zähler für die Bildfolge und wird nacheinander von 0 bis 29 hochgezählt, um dann wieder bei 0 zu beginnen. Nach jedem Einzelbild wartet das Programm 50 Millisekunden und führt dann den nächsten Aufruf von [repaint](#) durch:

[Listing2408.java](#)

```
001 /* Listing2408.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2408
007 extends Frame
008 implements Runnable
009 {
010     Thread th;
011     Image arImg[];
012     int actimage;
013
014     public static void main(String[] args)
015     {
016         Listing2408 wnd = new Listing2408();
017         wnd.setSize(200,150);
018         wnd.setVisible(true);
019         wnd.startAnimation();
020     }
021
022     public Listing2408()
023     {
024         super("Listing2408");
025         //WindowListener
026         addWindowListener(
027             new WindowAdapter() {
028                 public void windowClosing(WindowEvent event)
029                 {
030                     System.exit(0);
031                 }
032             }
033         );
034     }
035 }
```

```

034     }
035
036     public void startAnimation()
037     {
038         th = new Thread(this);
039         actimage = -1;
040         th.start();
041     }
042
043     public void run()
044     {
045         //Bilder laden
046         arImg = new Image[30];
047         MediaTracker mt = new MediaTracker(this);
048         Toolkit tk = getToolkit();
049         for (int i = 1; i <= 30; ++i) {
050             arImg[i-1] = tk.getImage("images/jana"+i+".gif");
051             mt.addImage(arImg[i-1], 0);
052             actimage = -i;
053             repaint();
054             try {
055                 mt.waitForAll();
056             } catch (InterruptedException e) {
057                 //nothing
058             }
059         }
060         //Animation beginnen
061         actimage = 0;
062         while (true) {
063             repaint();
064             actimage = (actimage + 1) % 30;
065             try {
066                 Thread.sleep(50);
067             } catch (InterruptedException e) {
068                 //nichts
069             }
070         }
071     }
072
073     public void paint(Graphics g)
074     {
075         if (actimage < 0) {
076             g.drawString("Lade Bitmap "+(-actimage),10,50);
077         } else {
078             g.drawImage(arImg[actimage],10,30,this);
079         }
080     }
081 }

```

Listing 24.8: Abspielen einer Folge von Bitmaps

Das vorliegende Beispiel verwendet die Bilddateien `jana1.gif` bis `jana30.gif`. Sie zeigen die verschiedenen Phasen des in Schreibschrift geschriebenen Namens »Jana«. Alternativ kann aber auch jede andere Sequenz von Bilddateien verwendet werden. Die folgenden Abbildungen zeigen einige Schnappschüsse der Programmausgabe:

Hinweis



Abbildung 24.5: Animation eines Schriftzugs, Schnappschuß 1



Abbildung 24.6: Animation eines Schriftzugs, Schnappschuß 2



Abbildung 24.7: Animation eines Schriftzugs, Schnappschuß 3

24.2.3 Animation mit Grafikprimitiven

Alternativ zur Anzeige von Bilddateien kann jedes Einzelbild der Animation natürlich auch mit den Ausgabeprimitiven der Klasse `Graphics` erzeugt werden. Dies hat den Vorteil, daß der Anwender nicht auf das Laden der Bilder warten muß. Außerdem ist das Verfahren flexibler als der bitmap-basierte Ansatz. Der Nachteil ist natürlich, daß die Grafikoperationen zeitaufwendiger sind und eine zügige Bildfolge bei komplexen Sequenzen schwieriger zu erzielen ist.

Als Beispiel für diese Art von Animation wollen wir uns die Aufgabe stellen, eine aus rechteckigen Kästchen bestehende bunte Schlange über den Bildschirm laufen zu lassen. Sie soll an den Bildschirmrändern automatisch umkehren und auch innerhalb des Ausgabefensters von Zeit zu Zeit ihre Richtung wechseln.

Das folgende Programm stellt die Schlange als `Vector` von Objekten des Typs `ColorRectangle` dar. `ColorRectangle` ist aus `Rectangle` abgeleitet und besitzt zusätzlich eine Membervariable zur Darstellung der Farbe des Rechtecks.

Dieses Beispiel folgt dem allgemeinen Architekturschema für Animationen, das wir auch in den letzten Beispielen verwendet haben. Der erste Schritt innerhalb von `run` besteht darin, die Schlange zu konstruieren. Dazu wird eine Folge von Objekten der Klasse `ColorRectangle` konstruiert, und ab Position `(100,100)` werden die Objekte horizontal nebeneinander angeordnet. Die Farben werden dabei so vergeben, daß die Schlange in fließenden Übergängen von rot bis blau dargestellt wird. Alle Elemente werden in dem `Vector snake` gespeichert.

Nachdem die Schlange konstruiert wurde, beginnt die Animation. Dazu wird die aktuelle Schlange angezeigt, eine Weile pausiert und dann durch Aufruf der Methode `moveSnake` die nächste Position der Schlange berechnet. `moveSnake` ist relativ aufwendig, denn hier liegt der Löwenanteil der »Intelligenz« der Animation. Die Richtung der Bewegung der Schlange wird durch die Variablen `dx` und `dy` getrennt für die *x*- und *y*-Richtung bestimmt. Steht hier der Wert `-1`, bewegt sich die Schlange im nächsten Schritt um die Breite eines Rechtecks in Richtung kleinerer Koordinaten. Bei `1` vergrößert sie die Koordinate entsprechend, und wenn der Wert `0` enthalten ist, verändert sich der zugehörige Koordinatenwert im nächsten Schritt gar nicht.

`dx` und `dy` werden entweder dann verändert, wenn die Schlange an einem der vier Bildschirmränder angekommen ist und umkehren muß oder (im Mittel bei jedem zehnten Schritt) auch auf freier Strecke. Nachdem auf diese Weise die neue Richtung bestimmt wurde, wird das erste Element der Schlange auf die neue Position bewegt. Alle anderen Elemente der Schlange bekommen dann die Position zugewiesen, die zuvor ihr Vorgänger hatte.

Eine alternative Art, die Schlange neu zu berechnen, würde darin bestehen, lediglich ein neues erstes Element zu generieren, an vorderster Stelle in den `Vector` einzufügen und das letzte Element zu löschen. Dies hätte allerdings den Nachteil, daß die Farbinformationen von vorne nach hinten durchgereicht würden und so jedes Element seine Farbe ständig ändern würde. Dieses (sehr viel performantere) Verfahren könnte verwendet werden, wenn alle Elemente der Schlange dieselbe Farbe hätten.

```
001  /* Listing2409.java */
002
003  import java.awt.*;
004  import java.awt.event.*;
005  import java.util.*;
006
007  class ColorRectangle
008  extends Rectangle
009  {
010      public Color color;
011  }
012
013  public class Listing2409
014  extends Frame
015  implements Runnable
016  {
017      //Konstanten
018      private static final int    SIZERECT    = 7;
019      private static final int    SLEEP       = 40;
020      private static final int    NUMELEMENTS = 20;
021      private static final Color  BGCOLOR     = Color.black;
022
023      //Instanzvariablen
024      private Thread th;
025      private Vector snake;
026      private int dx;
027      private int dy;
028
029      public static void main(String args[])
030      {
031          Listing2409 frame = new Listing2409();
032          frame.setSize(200,150);
033          frame.setVisible(true);
034          frame.startAnimation();
035      }
036
037      public Listing2409()
038      {
039          super("Listing2409");
040          setBackground(BGCOLOR);
041          //WindowListener
042          addWindowListener(
043              new WindowAdapter() {
044                  public void windowClosing(WindowEvent event)
045                  {
046                      System.exit(0);
047                  }
048              }
049          );
050          snake = new Vector();
051      }
052
053      public void startAnimation()
054      {
055          th = new Thread(this);
056          th.start();
057      }
058
059      public void run()
060      {
061          //Schlange konstruieren
062          ColorRectangle cr;
063          int x = 100;
```

```

064     int y = 100;
065     for (int i=0; i < NUMELEMENTS; ++i) {
066         cr = new ColorRectangle();
067         cr.x = x;
068         cr.y = y;
069         cr.width = SIZERECT;
070         cr.height = SIZERECT;
071         x += SIZERECT;
072         cr.color = new Color(
073             i*(256/NUMELEMENTS),
074             0,
075             240-i*(256/NUMELEMENTS)
076         );
077         snake.addElement(cr);
078     }
079
080     //Vorzugsrichtung festlegen
081     dx = -1;
082     dy = -1;
083
084     //Schlange laufen lassen
085     while (true) {
086         repaint();
087         try {
088             Thread.sleep(SLEEP);
089         } catch (InterruptedException e){
090             //nichts
091         }
092         moveSnake();
093     }
094 }
095
096 public void moveSnake()
097 {
098     Dimension size = getSize();
099     int sizex = size.width-getInsets().left-getInsets().right;
100     int sizey = size.height-getInsets().top-getInsets().bottom;
101     ColorRectangle cr = (ColorRectangle)snake.firstElement();
102     boolean lBorder = false;
103     int xalt, yalt;
104     int xtmp, ytmp;
105
106     //Kopf der Schlange neu berechnen
107     if (cr.x <= 1) {
108         dx = 1;
109         lBorder = true;
110     }
111     if (cr.x + cr.width >= sizex) {
112         dx = -1;
113         lBorder = true;
114     }
115     if (cr.y <= 1) {
116         dy = 1;
117         lBorder = true;
118     }
119     if (cr.y + cr.height >= sizey) {
120         dy = -1;
121         lBorder = true;
122     }
123     if (! lBorder) {
124         if (rand(10) == 0) {
125             if (rand(2) == 0) {
126                 switch (rand(5)) {
127                     case 0: case 1:
128                         dx = -1;

```

```

129         break;
130     case 2:
131         dx = 0;
132         break;
133     case 3: case 4:
134         dx = 1;
135         break;
136     }
137 } else {
138     switch (rand(5)) {
139     case 0: case 1:
140         dy = -1;
141         break;
142     case 2:
143         dy = 0;
144         break;
145     case 3: case 4:
146         dy = 1;
147         break;
148     }
149 }
150 }
151 }
152 xalt = cr.x + SIZERECT * dx;
153 yalt = cr.y + SIZERECT * dy;
154 //Rest der Schlange hinterherziehen
155 Enumeration e = snake.elements();
156 while (e.hasMoreElements()) {
157     cr = (ColorRectangle)e.nextElement();
158     xtmp = cr.x;
159     ytmp = cr.y;
160     cr.x = xalt;
161     cr.y = yalt;
162     xalt = xtmp;
163     yalt = ytmp;
164 }
165 }
166
167 public void paint(Graphics g)
168 {
169     ColorRectangle cr;
170     Enumeration e = snake.elements();
171     int inleft = getInsets().left;
172     int intop = getInsets().top;
173     while (e.hasMoreElements()) {
174         cr = (ColorRectangle)e.nextElement();
175         g.setColor(cr.color);
176         g.fillRect(cr.x+inleft,cr.y+intop,cr.width,cr.height);
177     }
178 }
179
180 private int rand(int limit)
181 {
182     return (int)(Math.random() * limit);
183 }
184 }

```

Listing 24.9: Die animierte Schlange

Die Schlange kann in einem beliebig kleinen oder großen Fenster laufen. Hier sind ein paar Beispiele für die Ausgabe des Programms, nachdem das Fenster in der Größe verändert wurde:

Hinweis

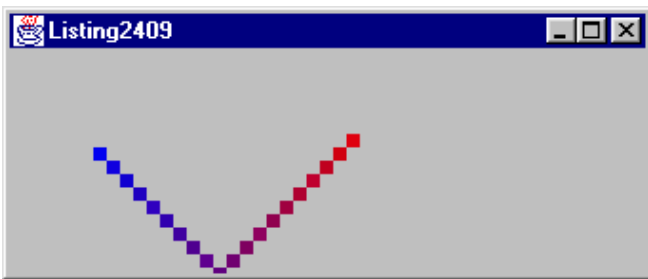


Abbildung 24.8: Die animierte Schlange, Schnappschuß 1



Abbildung 24.9: Die animierte Schlange, Schnappschuß 2

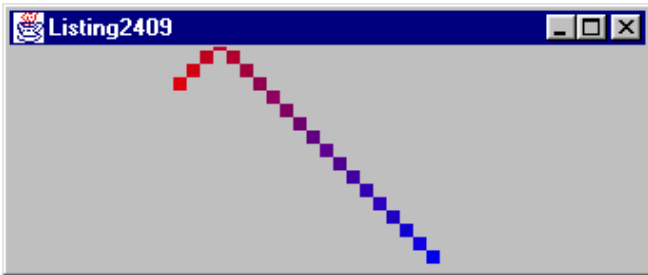


Abbildung 24.10: Die animierte Schlange, Schnappschuß 3

24.2.4 Reduktion des Bildschirmflackerns

Alle bisher entwickelten Animationen zeigen während der Ausführung ein ausgeprägtes Flackern, das umso stärker ist, je später ein Bildanteil innerhalb eines Animationsschrittes angezeigt wird. Der Grund für dieses Flackern liegt darin, daß vor jedem Aufruf von `paint` zunächst das Fenster gelöscht wird und dadurch unmittelbar vor der Ausgabe des nächsten Bildes ganz kurz ein vollständig leerer Hintergrund erscheint.

Leider besteht die Lösung für dieses Problem nicht einfach darin, das Löschen zu unterdrücken. Bei einer animierten Bewegung beispielsweise ist es erforderlich, all die Bestandteile des vorigen Bildes zu löschen, die im aktuellen Bild nicht mehr oder an einer anderen Stelle angezeigt werden.

Auch wenn `paint` deshalb aufgerufen wird, weil ein bisher verdeckter Bildausschnitt wieder sichtbar wird, muß natürlich der entsprechende Bildausschnitt zunächst gelöscht werden, um die Bestandteile des anderen Fensters zu entfernen. Im Grunde ist es also eine ganz vernünftige Vorgehensweise, das Fenster vor jedem Aufruf von `paint` zu löschen.

Das Flackern kann nun auf unterschiedliche Weise unterdrückt werden. Die drei gebräuchlichsten Methoden sind folgende:

- den Bildschirm gar nicht zu löschen (was - wie oben erwähnt - problematisch ist)
- nur den wirklich benötigten Teil des Bildschirms zu löschen
- das Verfahren der Doppelpufferung anzuwenden

Jedes dieser Verfahren hat Vor- und Nachteile und kann in verschiedenen Situationen unterschiedlich gut angewendet werden. Wir werden sie in den folgenden Unterabschnitten kurz vorstellen und ein Beispiel für ihre Anwendung geben. Es gibt noch einige zusätzliche Möglichkeiten, das Flackern zu unterdrücken oder einzuschränken, wie beispielsweise das Clipping der Ausgabe auf den tatsächlich veränderten Bereich, aber darauf wollen wir hier nicht näher eingehen.

Bildschirm nicht löschen

Den Bildschirm überhaupt nicht zu löschen, um das Flackern zu unterdrücken, ist nur bei nicht bewegten Animationen möglich. Wir wollen uns als Beispiel für ein Programm, das hierfür geeignet ist, das folgende Lauflicht ansehen:

```

001 /* Listing2410.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2410
007 extends Frame
008 implements Runnable
009 {
010     //Konstanten
011     private static final int NUMLEDS = 20;
012     private static final int SLEEP = 60;
013     private static final int LEDSIZE = 10;
014     private static final Color ONCOLOR = new Color(255,0,0);
015     private static final Color OFFCOLOR = new Color(100,0,0);
016
017     //Instanzvariablen
018     private Thread th;
019     private int switched;
020     private int dx;
021
022     public static void main(String args[])
023     {
024         Listing2410 frame = new Listing2410();
025         frame.setSize(270,150);
026         frame.setVisible(true);
027         frame.startAnimation();
028     }
029
030     public Listing2410()
031     {
032         super("Listing2410");
033         setBackground(Color.lightGray);
034         //WindowListener
035         addWindowListener(
036             new WindowAdapter() {
037                 public void windowClosing(WindowEvent event)
038                 {
039                     setVisible(false);
040                     dispose();
041                     System.exit(0);
042                 }
043             }
044         );
045     }
046
047     public void startAnimation()
048     {
049         th = new Thread(this);
050         th.start();
051     }
052
053     public void run()
054     {
055         switched = -1;
056         dx = 1;
057         while (true) {
058             repaint();
059             try {
060                 Thread.sleep(SLEEP);
061             } catch (InterruptedException e){
062                 //nichts
063             }
064             switched += dx;

```



```

065         if (switched < 0 || switched > NUMLEDS- 1) {
066             dx = -dx;
067             switched += 2*dx;
068         }
069     }
070 }
071
072 public void paint(Graphics g)
073 {
074     for (int i = 0; i < NUMLEDS; ++i) {
075         g.setColor(i == switched ? ONCOLOR : OFFCOLOR);
076         g.fillOval(10+i*(LEDSIZE+2),80,LEDSIZE,LEDSIZE);
077     }
078 }
079 }

```

Listing 24.10: Bildschirmflackern reduzieren bei stehenden Animationen

Das Programm zeigt eine Kette von 20 Leuchtdioden, die nacheinander an- und ausgeschaltet werden und dadurch ein Lauflicht simulieren, das zwischen linkem und rechtem Rand hin- und herläuft:

Hinweis

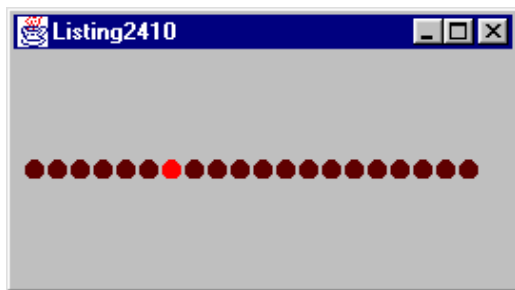


Abbildung 24.11: Die Lauflicht-Animation

Wie kann nun aber das Löschen verhindert werden? Die Lösung basiert auf der Tatsache, daß bei einem Aufruf von [repaint](#) nicht gleich [paint](#), sondern zunächst die Methode [update](#) aufgerufen wird. In der Standardversion der Klasse [Component](#) könnte [update](#) etwa so implementiert sein:

```

001 public void update(Graphics g)
002 {
003     g.setColor(getBackground());
004     g.fillRect(0, 0, width, height);
005     g.setColor(getForeground());
006     paint(g);
007 }

```

Listing 24.11: Standard-Implementierung von update

Zunächst wird die aktuelle Hintergrundfarbe ausgewählt, um in dieser Farbe ein ausgefülltes Rechteck in der Größe des Bildschirms zu zeichnen. Erst nach diesem Löschvorgang wird die Vordergrundfarbe gesetzt und [paint](#) aufgerufen.

Da in Java alle Methodenaufrufe dynamisch gebunden werden, kann das Löschen dadurch verhindert werden, daß [update](#) durch eine eigene Version überlagert wird, die den Hintergrund unverändert läßt. Durch einfaches Hinzufügen der folgenden drei Zeilen kann das Flackern des Lauflichts vollkommen unterdrückt werden:

[update1.inc](#)

```

001 /* update1.inc */
002
003 public void update(Graphics g)
004 {
005     paint(g);
006 }

```

Listing 24.12: Modifizierte Version von update

Nur den wirklich benötigten Teil des Bildschirms löschen

Wie schon erwähnt, kann auf das Löschen des Bildschirms nur dann komplett verzichtet werden, wenn die Animation keine Bewegung enthält. Ist sie dagegen bewegt, kann es sinnvoll sein, nur die Teile des Bildes zu löschen, die beim aktuellen Animationsschritt leer sind, im vorigen Schritt aber Grafikelemente enthielten.

Um welche Teile der Grafik es sich dabei handelt, ist natürlich von der Art der Animation abhängig. Zudem muß jeder Animationsschritt Informationen über den vorigen Schritt haben, um die richtigen Stellen löschen zu können. Ein Beispiel, bei dem diese Technik gut angewendet werden kann, ist die bunte Schlange aus dem Abschnitt »Animation mit Grafikprimitiven«.

Da die Schlange bei jedem Schritt einen neuen Kopf bekommt und alle anderen Elemente die Plätze ihres jeweiligen Vorgängers einnehmen, bleibt als einziges wirklich zu löschendes Element das letzte Element der Schlange aus dem vorherigen Animationsschritt übrig. Dessen Position könnte man sich bei jedem Schritt merken und im nächsten Schritt in der Hintergrundfarbe neu zeichnen.

Noch einfacher geht es, indem man an die Schlange einfach ein zusätzliches unsichtbares Element anhängt. Wird nämlich das letzte Element grundsätzlich in der Hintergrundfarbe dargestellt, hinterläßt es keine Spuren auf dem Bildschirm und braucht damit auch nicht explizit gelöscht zu werden! Wir brauchen also nur hinter die `for-next`-Schleife zur Konstruktion der Schlange ein weiteres, unsichtbares Element an den `snake-Vector` anzuhängen (in [Listing 24.13](#) in den Zeilen [025](#) bis [031](#) eingefügt):

[Schlange2.inc](#)

```
001 /* Schlange2.inc */
002
003 public void run()
004 {
005     //Schlange konstruieren
006     ColorRectangle cr;
007     int x = 100;
008     int y = 100;
009     for (int i=0; i < NUMELEMENTS; ++i) {
010         cr = new ColorRectangle();
011         cr.x = x;
012         cr.y = y;
013         cr.width = SIZERECT;
014         cr.height = SIZERECT;
015         x += SIZERECT;
016         cr.color = new Color(
017             i*(256/NUMELEMENTS),
018             0,
019             240-i*(256/NUMELEMENTS)
020         );
021         snake.addElement(cr);
022     }
023
024     //Löschelement anhängen
025     cr = new ColorRectangle();
026     cr.x = x;
027     cr.y = y;
028     cr.width = SIZERECT;
029     cr.height = SIZERECT;
030     cr.color = BGCOLOR;
031     snake.addElement(cr);
032
033     //Vorzugsrichtung festlegen
034     dx = -1;
035     dy = -1;
036
037     //Schlange laufen lassen
038     while (true) {
039         repaint();
040         try {
041             Thread.sleep(SLEEP);
042         } catch (InterruptedException e){
043             //nichts
044         }
045         moveSnake();
046     }
047 }
```

Listing 24.13: Modifizierte Schlangenanimation

Wird nun zusätzlich die Methode `update` überlagert, wie es auch im vorigen Abschnitt getan wurde, läuft die Schlange vollkommen flackerfrei.

Doppelpufferung

Das Doppelpuffern bietet sich immer dann an, wenn die beiden vorigen Methoden versagen. Das kann beispielsweise dann der Fall sein, wenn es bei einer bewegten Animation zu aufwendig ist, nur den nicht mehr benötigten Teil der Bildschirmausgabe zu löschen, oder wenn der aktuelle Animationsschritt keine Informationen darüber besitzt, welcher Teil zu löschen ist.

Beim Doppelpuffern wird bei jedem Animationsschritt zunächst die gesamte Bildschirmausgabe in ein *Offscreen-Image* geschrieben. Erst wenn alle Ausgabeoperationen abgeschlossen sind, wird dieses Offscreen-Image auf die Fensteroberfläche kopiert. Im Detail sind dazu folgende Schritte erforderlich:

- Das Fensterobjekt beschafft sich durch Aufruf von `createImage` ein Offscreen-Image und speichert es in einer Instanzvariablen.
- Durch Aufruf von `getGraphics` wird ein Grafikkontext zu diesem Image beschafft.
- Alle Bildschirmausgaben (inklusive Löschen des Bildschirms) gehen zunächst auf den Offscreen-Grafikkontext.
- Wenn alle Ausgabeoperationen abgeschlossen sind, wird das Offscreen-Image mit `drawImage` in das Ausgabefenster kopiert.

Durch diese Vorgehensweise wird erreicht, daß das Bild komplett aufgebaut ist, bevor es angezeigt wird. Da beim anschließenden Kopieren die neuen Pixel direkt über die alten kopiert werden, erscheinen dem Betrachter nur die Teile des Bildes verändert, die auch tatsächlich geändert wurden. Ein Flackern, das entsteht, weil Flächen für einen kurzen Zeitraum gelöscht und dann wieder gefüllt werden, kann nicht mehr auftreten.

Die Anwendung des Doppelpufferns ist nicht immer sinnvoll. Sollte eine der anderen Methoden mit vertretbarem Aufwand implementiert werden können, kann es sinnvoller sein, diese zu verwenden. Nachteilig sind vor allem der Speicherbedarf für die Konstruktion des Offscreen-Images und die Verzögerungen durch das doppelte Schreiben der Bilddaten. Hier muß im Einzelfall entschieden werden, welche Variante zum Einsatz kommen soll. In vielen Fällen allerdings können die genannten Nachteile vernachlässigt werden, und die Doppelpufferung ist ein probates Mittel, um das Bildschirmflackern zu verhindern.

Tip

Das folgende Programm ist ein Beispiel für die Anwendung des Doppelpufferns bei der Ausgabe einer bewegten Animation. Wir wollen uns dafür die Aufgabe stellen, eine große Scheibe über den Bildschirm laufen zu lassen, über deren Rand zwei stilisierte »Ameisen« mit unterschiedlicher Geschwindigkeit in entgegengesetzte Richtungen laufen.

Beispiel

Das folgende Programm löst diese Aufgabe. Dabei folgt die Animation unserem bekannten Architekturschema für bewegte Grafik und braucht hier nicht weiter erklärt zu werden. Um das Flackern zu verhindern, deklarieren wir zwei Instanzvariablen, `dbImage` und `dbGraphics`:

```
private Image dbImage;  
private Graphics dbGraphics;
```

Glücklicherweise können die zum Doppelpuffern erforderlichen Schritte gekapselt werden, wenn man die Methode `update` geeignet überlagert:

Hinweis

[update2.inc](#)

```
001 /* update2.inc */  
002  
003 public void update(Graphics g)  
004 {  
005     //Double-Buffer initialisieren  
006     if (dbImage == null) {  
007         dbImage = createImage(  
008             this.getSize().width,  
009             this.getSize().height  
010         );  
011         dbGraphics = dbImage.getGraphics();  
012     }  
013     //Hintergrund löschen  
014     dbGraphics.setColor(getBackground());  
015     dbGraphics.fillRect(  
016         0,  
017         0,  
018         this.getSize().width,  
019         this.getSize().height  
020     );  
021     //Vordergrund zeichnen  
022     dbGraphics.setColor(getForeground());  
023     paint(dbGraphics);  
024     //Offscreen anzeigen  
025     g.drawImage(dbImage, 0, 0, this);  
026 }
```

Falls nicht schon geschehen, werden hier zunächst die beiden Variablen `dbImage` und `dbGraphics` initialisiert. Anschließend wird der Hintergrund gelöscht, wie es auch in der Standardversion von `update` der Fall ist. Im Gegensatz zu dieser erfolgt das Löschen aber auf dem Offscreen-Image und ist somit für den Anwender nicht zu sehen. Nun wird `paint` aufgerufen und bekommt anstelle des normalen den Offscreen-Grafikkontext übergeben. Ohne selbst etwas davon zu wissen, sendet `paint` damit alle seine Grafikbefehle auf das Offscreen-Image. Nachdem `paint` beendet wurde, wird durch Aufruf von `drawImage` das Offscreen-Image auf dem Bildschirm angezeigt.

Hier ist der komplette Quellcode des Programms:

[Listing2415.java](#)

```
001 /* Listing2415.java */
002
003 import java.awt.*;
004 import java.awt.event.*;
005
006 public class Listing2415
007 extends Frame
008 implements Runnable
009 {
010     private Thread th;
011     private int actx;
012     private int dx;
013     private int actarc1;
014     private int actarc2;
015     private Image dbImage;
016     private Graphics dbGraphics;
017
018     public static void main(String[] args)
019     {
020         Listing2415 frame = new Listing2415();
021         frame.setSize(210,170);
022         frame.setVisible(true);
023         frame.startAnimation();
024     }
025
026     public Listing2415()
027     {
028         super("Listing2415");
029         //WindowListener
030         addWindowListener(
031             new WindowAdapter() {
032                 public void windowClosing(WindowEvent event)
033                 {
034                     setVisible(false);
035                     dispose();
036                     System.exit(0);
037                 }
038             }
039         );
040     }
041
042     public void startAnimation()
043     {
044         Thread th = new Thread(this);
045         th.start();
046     }
047
048     public void run()
049     {
050         actx = 0;
051         dx = 1;
052         actarc1 = 0;
053         actarc2 = 0;
054         while (true) {
055             repaint();
```

```

056         actx += dx;
057         if (actx < 0 || actx > 100) {
058             dx = -dx;
059             actx += 2*dx;
060         }
061         actarc1 = (actarc1 + 1) % 360;
062         actarc2 = (actarc2 + 2) % 360;
063         try {
064             Thread.sleep(40);
065         } catch (InterruptedException e) {
066             //nichts
067         }
068     }
069 }
070
071 public void update(Graphics g)
072 {
073     //Double-Buffer initialisieren
074     if (dbImage == null) {
075         dbImage = createImage(
076             this.getSize().width,
077             this.getSize().height
078         );
079         dbGraphics = dbImage.getGraphics();
080     }
081     //Hintergrund löschen
082     dbGraphics.setColor(getBackground());
083     dbGraphics.fillRect(
084         0,
085         0,
086         this.getSize().width,
087         this.getSize().height
088     );
089     //Vordergrund zeichnen
090     dbGraphics.setColor(getForeground());
091     paint(dbGraphics);
092     //Offscreen anzeigen
093     g.drawImage(dbImage,0,0,this);
094 }
095
096 public void paint(Graphics g)
097 {
098     int xoffs = getInsets().left;
099     int yoffs = getInsets().top;
100     g.setColor(Color.lightGray);
101     g.fillOval(xoffs+actx,yoffs+20,100,100);
102     g.setColor(Color.red);
103     g.drawArc(xoffs+actx,yoffs+20,100,100,actarc1,10);
104     g.drawArc(xoffs+actx-1,yoffs+19,102,102,actarc1,10);
105     g.setColor(Color.blue);
106     g.drawArc(xoffs+actx,yoffs+20,100,100,360-actarc2,10);
107     g.drawArc(xoffs+actx-1,yoffs+19,102,102,360-actarc2,10);
108 }
109 }

```

Listing 24.15: Animation mit Doppelpufferung

Ein Schnappschuß des laufenden Programms sieht so aus (die beiden »Ameisen« sind in der Abbildung etwas schwer zu erkennen, im laufenden Programm sieht man sie besser):

Hinweis

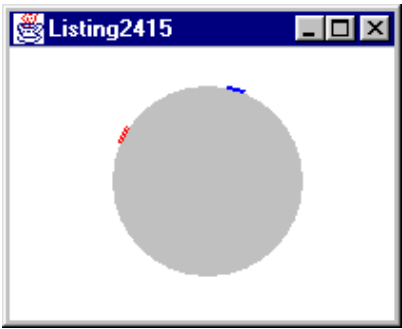


Abbildung 24.12: Eine Animation mit Doppelpufferung

Durch die Kapselung des Doppelpufferns können Programme sogar nachträglich flackerfrei gemacht werden, ohne daß in den eigentlichen Ausgaberoutinen irgend etwas geändert werden müßte. Man könnte beispielsweise aus `Frame` eine neue Klasse `DoubleBufferFrame` ableiten, die die beiden privaten Membervariablen `dbImage` und `dbGraphics` besitzt und `update` in der beschriebenen Weise implementiert. Alle Klassen, die dann von `DoubleBufferFrame` anstelle von `Frame` abgeleitet werden, unterstützen das Doppelpuffern ihrer Grafikausgaben automatisch.

24.3 Zusammenfassung

- [24.3 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Klasse [Image](#) als Repräsentation einer Bitmap im laufenden Programm
- Laden und Anzeigen einer Bitmap mit [getImage](#) und [drawImage](#)
- Die Verwendung der Klasse [MediaTracker](#) zum synchronen Laden von Bildern
- Entwurf einer eigenen Komponente zur Anzeige von Bitmaps
- Prinzipielle Vorgehensweise bei der Animation von Grafiken
- Verwendung eines Threads zur Entkoppelung der Animation vom Hauptprogramm
- Laden und animierte Darstellung einer Folge von Bitmaps
- Erstellen von Animationen unter Verwendung der grafischen Primitivoperationen
- Begründung des Bildschirmflackerns bei Animationen und Vorstellung verschiedener Techniken zur Verminderung des Flackerns
- Verwendung eines Offscreen-Images zur Implementierung der Doppelpufferung

Kapitel 25

Applets

- [25 Applets](#)
 - [25.1 Die Architektur eines Applets](#)
 - [25.1.1 Grundlagen](#)
 - [25.1.2 Die Klasse java.awt.Applet](#)
 - [25.1.3 Initialisierung und Endebehandlung](#)
 - [Instanzierung des Applets](#)
 - [Initialisierung des Applets](#)
 - [Starten des Applets](#)
 - [Stoppen des Applets](#)
 - [Zerstören des Applets](#)
 - [25.1.4 Weitere Methoden der Klasse Applet](#)
 - [Methoden zum Nachrichtentransfer](#)
 - [showStatus](#)
 - [getParameterInfo](#)
 - [getAppletInfo](#)
 - [25.2 Einbinden eines Applets](#)
 - [25.2.1 Das APPLET-Tag](#)
 - [25.2.2 Die Parameter des Applet-Tags](#)
 - [25.2.3 Parameterübergabe an Applets](#)
 - [25.3 Die Ausgabe von Sound](#)
 - [25.3.1 Soundausgabe in Applets](#)
 - [25.3.2 Soundausgabe in Applikationen](#)
 - [25.4 Verweise auf andere Seiten](#)
 - [25.4.1 Die Klasse URL](#)
 - [25.4.2 Der Applet-Kontext](#)
 - [25.4.3 Die Methode showDocument](#)
 - [25.5 Animation in Applets](#)
 - [25.6 Zusammenfassung](#)

25.1 Die Architektur eines Applets

- [25.1 Die Architektur eines Applets](#)
 - [25.1.1 Grundlagen](#)
 - [25.1.2 Die Klasse java.awt.Applet](#)
 - [25.1.3 Initialisierung und Endebehandlung](#)
 - [Instanziierung des Applets](#)
 - [Initialisierung des Applets](#)
 - [Starten des Applets](#)
 - [Stoppen des Applets](#)
 - [Zerstören des Applets](#)
 - [25.1.4 Weitere Methoden der Klasse Applet](#)
 - [Methoden zum Nachrichtentransfer](#)
 - [showStatus](#)
 - [getParameterInfo](#)
 - [getAppletInfo](#)

25.1.1 Grundlagen

Für viele Leser, die Java lernen wollen, steht die Entwicklung von Applets im Vordergrund und ist der Hauptgrund für die Beschäftigung mit der Sprache. In letzter Zeit ist allerdings ein Trend zu beobachten, bei dem Java zunehmend auch als Sprache für die Anwendungsentwicklung an Bedeutung gewinnt. Daß man mit Java auch Web-Pages verschönern kann, ist dabei ein angenehmer Nebeneffekt.

Tatsächlich unterscheiden sich Applets und Applikationen gar nicht so stark voneinander, wie man vermuten könnte. Bis auf wenige Ausnahmen werden sie mit denselben Werkzeugen und Techniken konstruiert. Vereinfacht kann man sagen, daß Java-Applikationen eigenständige Programme sind, die zur Ausführung den Stand-Alone-Java-Interpreter benötigen, während Java-Applets aus HTML-Seiten heraus aufgerufen werden und zur Ausführung einen Web-Browser benötigen.

Die wichtigsten Unterschiede kann man in einer kurzen Liste zusammenfassen:

- Das Hauptprogramm eines Applets wird immer aus der Klasse [Applet](#) abgeleitet. Bei einer Applikation ist es prinzipiell gleichgültig, woraus die Hauptklasse abgeleitet wird.
- Eine Applikation wird gestartet, indem vom Java-Interpreter die Klassenmethode [main](#) aufgerufen wird. Das Starten eines Applets wird dadurch erreicht, daß der Web-Browser die Applet-Klasse instanziert und die Methoden [init](#) und [start](#) aufruft.
- Aus Sicherheitsgründen darf ein Applet in der Regel weder auf Dateien des lokalen Rechners zugreifen noch externe Programme auf diesem starten. Eine Ausnahme bilden *signierte Applets*. Für eine Applikation gelten diese Beschränkungen nicht.
- Ein Applet arbeitet immer grafik- und ereignisorientiert. Bei einer Applikation dagegen ist es möglich, auf die Verwendung des AWT zu verzichten und alle Ein-/Ausgaben textorientiert zu erledigen.
- Applets bieten einige zusätzliche Möglichkeiten im Bereich des Benutzerschnittstellen-Designs, die so bei Applikationen nicht ohne weiteres zu finden sind. Die Ausgabe von Sound beispielsweise ist standardmäßig auf Applets beschränkt.

Dieses Kapitel erklärt die Grundlagen der Applet-Programmierung und erläutert die Einbindung von Applets in HTML-Dokumente. Es baut dabei auf vielen der in den Kapiteln [14](#) bis [24](#) des Buches vermittelten AWT-Features auf, ohne deren Kenntnis die Applet-Programmierung kaum möglich wäre.

25.1.2 Die Klasse java.awt.Applet

Wie schon erwähnt, wird das Hauptprogramm eines Applets aus der Klasse [Applet](#) des Pakets [java.applet](#) abgeleitet. [Applet](#) ist nun aber keine der abstrakten Basisklassen der Java-Klassenbibliothek, wie man es vielleicht erwarten würde, sondern eine konkrete Grafikklasse am Ende der Klassenhierarchie. [Applet](#) ist aus der Klasse [Panel](#) abgeleitet, diese aus [Container](#) und [Container](#) aus [Component](#). [Abbildung 25.1](#) stellt die Ableitungshierarchie schematisch dar:



Abbildung 25.1: Ableitungsbaum der Applet-Klasse

Ein Applet ist also ein rechteckiges Bildelement, das eine Größe und Position hat, Ereignisse empfangen kann und in der Lage ist, grafische Ausgaben vorzunehmen. Tatsächlich kommt dies dem Anspruch eines Applets, ein eigenständiges Programm zu sein, das innerhalb eines fensterartigen Ausschnitts in einem grafikfähigen Web-Browser läuft, sehr entgegen. Durch die Vererbungshierarchie erbt ein Applet bereits von seinen Vaterklassen einen großen Teil der Fähigkeiten, die es zur Ausführung benötigt. Die über die Fähigkeiten von [Container](#), [Component](#) und [Panel](#) hinaus erforderliche Funktionalität, die benötigt wird, um ein Objekt der Klasse [Applet](#) als Hauptmodul eines eigenständigen Programms laufen zu lassen, wird von der Klasse [Applet](#) selbst zur Verfügung gestellt.

25.1.3 Initialisierung und Endebehandlung

Die Kommunikation zwischen einem Applet und seinem Browser läuft auf verschiedenen Ebenen ab. Nach dem Laden wird das Applet zunächst *instanziert* und dann *initialisiert*. Anschließend wird es gestartet, erhält GUI-Events und wird irgendwann wieder gestoppt. Schließlich wird das Applet vom Browser nicht mehr benötigt und zerstört.

Zu jedem dieser Schritte gibt es eine korrespondierende Methode, die vor dem entsprechenden Statuswechsel aufgerufen wird. Die aus [Applet](#) abgeleitete Klasse ist dafür verantwortlich, diese Methoden zu überlagern und mit der erforderlichen Funktionalität auszustatten.

Instanziierung des Applets

Bevor ein Applet aktiv werden kann, muß der Browser zunächst ein Objekt der abgeleiteten [Applet](#)-Klasse instanzieren. Hierzu ruft er den parameterlosen Default-Konstruktor der Klasse auf:

```
public Applet()
    java.applet.Applet
```

Üblicherweise wird dieser in der abgeleiteten Klasse nicht überlagert, sondern von [Applet](#) geerbt. Notwendige Initialisierungen von Membervariablen werden später erledigt.

Initialisierung des Applets

Nach der Instanziierung ruft der Browser die Methode [init](#) auf, um dem Applet die Möglichkeit zu geben, Initialisierungen vorzunehmen:

```
public void init()
    java.applet.Applet
```

[init](#) wird während der Lebensdauer eines Applets *genau einmal* aufgerufen, nachdem die Klassendatei geladen und das Applet instanziiert wurde. Innerhalb von [init](#) können Membervariablen initialisiert, Images oder Fonts geladen oder Parameter ausgewertet werden.

Hinweis

Starten des Applets

Nachdem die Initialisierung abgeschlossen ist, wird die Methode [start](#) aufgerufen, um die Ausführung des Applets zu starten:

```
public void start()
    java.applet.Applet
```

Im Gegensatz zur Initialisierung kann das Starten eines Applets mehrfach erfolgen. Wenn der Browser eine andere Web-Seite lädt, wird das Applet nicht komplett zerstört, sondern lediglich gestoppt (siehe nächsten Abschnitt). Bei erneutem Aufruf der Seite wird es dann wieder gestartet und die Methode [start](#) erneut aufgerufen.

Hinweis

Stoppen des Applets

Durch Aufrufen der Methode [stop](#) zeigt der Browser dem Applet an, daß es gestoppt werden soll:

```
public void stop()
    java.applet.Applet
```

Wie erwähnt, geschieht dies immer dann, wenn eine andere Seite geladen wird. Während der Lebensdauer eines Applets können die Methoden [start](#) und [stop](#) also mehrfach aufgerufen werden. Keinesfalls darf ein Applet also innerhalb von [stop](#) irgendwelche endgültigen Aufräumarbeiten durchführen und Ressourcen entfernen, die es bei einem nachträglichen Neustart wieder benötigen würde.

Zerstören des Applets

Wenn ein Applet ganz bestimmt nicht mehr gebraucht wird (z.B. weil der Browser beendet wird), ruft der Browser die Methode `destroy` auf:

```
public void destroy()
```

Diese kann überlagert werden, um Aufräumarbeiten zu erledigen, die erforderlich sind, wenn das Applet nicht mehr verwendet wird. Eine typische Anwendung von `destroy` besteht beispielsweise darin, einen Thread zu zerstören, der bei der Initialisierung eines Applets angelegt wurde.

25.1.4 Weitere Methoden der Klasse Applet

Methoden zum Nachrichtentransfer

Neben diesen vier speziellen Methoden kann ein Applet alle Nachrichten erhalten, die an ein `Component`-Objekt versendet werden. Hierzu zählen Mouse-, MotionEvent-, Key-, Focus- und Component-Events ebenso wie die Aufforderung an das Applet, seine Client-Area neu zu zeichnen. Bezüglich Reaktion auf die Events und die Registrierung und Programmierung geeigneter Listener-Klassen verhält sich ein Applet genauso wie jedes andere Fenster. In [Kapitel 18](#) und [Kapitel 19](#) wurden die möglichen Ereignisse und die Reaktion des Programms darauf vorgestellt.

showStatus

Mit der Methode `showStatus` kann das Applet einen Text in die Statuszeile des HTML-Browsers schreiben, der das Applet ausführt:

```
public void showStatus(String msg)
```

Das folgende Beispiel zeigt ein sehr einfaches Applet, das den Text »Hello, world« auf dem Bildschirm ausgibt und in die Statuszeile des Browsers schreibt:

Beispiel

[Listing2501.java](#)

```
001 /* Listing2501.java */
002
003 import java.awt.*;
004 import java.applet.*;
005
006 public class Listing2501
007 extends Applet
008 {
009     public void paint(Graphics g)
010     {
011         showStatus("Hello, world");
012         g.drawString("Hello, world",10,50);
013     }
014 }
```

Listing 25.1: Ein einfaches Applet

getParameterInfo

Die Klasse `Applet` besitzt eine Methode `getParameterInfo`, die in abgeleiteten Klassen überlagert werden sollte:

```
public String[][] getParameterInfo()
```

`getParameterInfo` kann vom Browser aufgerufen werden, um Informationen über die vom Applet akzeptierten Parameter zu erhalten. Der Rückgabewert von `getParameterInfo` ist ein zweidimensionales Array von Strings. Jedes Element des Arrays beschreibt einen Parameter des Applets durch ein Subarray mit drei Elementen. Das erste Element gibt den Namen des Parameters an, das zweite seinen Typ und das dritte eine textuelle Beschreibung des Parameters. Die Informationen sollten so gestaltet sein, daß sie für *menschliche* Benutzer verständlich sind; eine programmgesteuerte Verwendung ist eigentlich nicht vorgesehen.

Das nachfolgende Listing zeigt eine beispielhafte Implementierung der Methode `getParameterInfo` für das Applet in [Listing 25.5](#):

Beispiel

```
001 public String[][] getParameterInfo()
002 {
003     String ret = {
004         {"redwidth","int","Breite eines roten Segments"},
005         {"whitewidth","int","Breite eines weissen Segments"}
006     };
007     return ret;
008 }
```

Listing 25.2: Verwendung von `getParameterInfo`

getAppletInfo

Ähnlich der Methode [getParameterInfo](#) gibt es eine Methode [getAppletInfo](#), mit der die Anwendung Informationen über das Applet zur Verfügung stellen sollte:

java.applet.Applet

```
public String getAppletInfo()
```

Die Sprachspezifikation gibt an, daß hier ein String zurückgegeben werden sollte, der Angaben zum Applet selbst, zur aktuellen Version und zum Autor des Applets macht. Eine beispielhafte Implementierung könnte so aussehen:

```
001 public String getAppletInfo()  
002 {  
003     return "AppletBeispiel Ver. 1.0 (C) 1997-99 Guido Krueger";  
004 }
```

Listing 25.3: Die Methode getAppletInfo

25.2 Einbinden eines Applets

- 25.2 Einbinden eines Applets
 - 25.2.1 Das APPLET-Tag
 - 25.2.2 Die Parameter des Applet-Tags
 - 25.2.3 Parameterübergabe an Applets

25.2.1 Das APPLET-Tag

Das Einbinden eines Applets in ein HTML-Dokument erfolgt unter Verwendung des APPLET-Tags, es wird also durch `<APPLET>` eingeleitet und durch `</APPLET>` beendet. Zwischen den beiden Marken kann ein Text stehen, der angezeigt wird, wenn das Applet nicht aufgerufen werden kann. Ein applet-fähiger Browser ignoriert den Text. Beispiel:

```

001 <APPLET CODE="Hello.class" WIDTH=300 HEIGHT=200>
002 Hier steht das Applet Hello
003 </APPLET>

```

Listing 25.4: Das APPLET-Tag

Ein Applet-Tag wird wie normaler Text in die Browser-Ausgabe eingebunden. Das Applet belegt soviel Platz auf dem Bildschirm, wie durch die Größenangaben `WIDTH` und `HEIGHT` reserviert wurde. Soll das Applet in einer eigenen Zeile stehen, müssen separate Zeilenschaltungen in den HTML-Code eingebaut werden (beispielsweise `<p>` oder `
`), oder es muß ein Tag verwendet werden, dessen Ausgabe in einer eigenen Zeile steht (z.B. `<h1>` bis `<h6>`).

Neben dem Ersatztext, der zwischen dem Beginn- und Ende-Tag steht, besitzt ein Applet-Tag weitere Parameter:

- Eine Reihe von Parametern, die innerhalb von `<APPLET>` untergebracht werden und die Größe und Anordnung des Applets bestimmen. `CODE`, `WIDTH` und `HEIGHT` müssen dabei in jedem Fall angegeben werden, außerdem gibt es noch einige optionale Parameter.
- Eine Liste von PARAM-Tags, die zwischen `<APPLET>` und `</APPLET>` stehen und zur Parameterübergabe an das Applet verwendet werden.

Zwischen beiden Parameterarten besteht ein grundsätzlicher Unterschied. Während die Parameter der ersten Gruppe (also `CODE`, `WIDTH` und `HEIGHT`) vom Browser interpretiert werden, um die visuelle Darstellung des Applets zu steuern, werden die Parameter der zweiten Gruppe an das Applet weitergegeben. Der Browser übernimmt bei ihnen nur die Aufbereitung und die Übergabe an das Applet, führt aber selbst keine Interpretation der Parameter aus.

25.2.2 Die Parameter des Applet-Tags

Der wichtigste Parameter des Applet-Tags heißt `CODE` und gibt den Namen der Applet-Klasse an. Bei der Angabe des `CODE`-Parameters sind einige Dinge zu beachten:

- Der Klassenname sollte inklusive der Extension `.class` angegeben werden.
- Groß- und Kleinschreibung müssen eingehalten werden.
- Die Klassendatei muß im aktuellen bzw. angegebenen Verzeichnis zu finden sein.

Alternativ kann die Klassendatei auch in einem der Verzeichnisse liegen, die in der Umgebungsvariablen `CLASSPATH` angegeben wurden. `CLASSPATH` enthält eine Liste von durch Kommata getrennten Verzeichnissen, die in der Reihenfolge ihres Auftretens durchsucht werden. `CLASSPATH` spielt außerdem beim Aufruf des Compilers eine Rolle: Sie dient dazu, die importierten Pakete zu suchen.

Das Applet-Tag hat zwei weitere nichtoptionale Parameter `WIDTH` und `HEIGHT`, die die Höhe und Breite des für das Applet reservierten Bildschirmausschnitts angeben. Innerhalb des Applets steht ein Rechteck dieser Größe als Ausgabefläche zur Verfügung.

Das Applet-Tag besitzt weitere optionale Parameter. Diese dienen zur Konfiguration des Applets und zur Beeinflussung der Darstellung des Applets und des umgebenden Textes. [Tabelle 25.1](#) gibt einen Überblick über die verfügbaren Parameter:

Parameter	Bedeutung
<code>CODEBASE</code>	Hier kann ein alternatives Verzeichnis für das Laden der Klassendateien angegeben werden. Fehlt diese Angabe, wird das Dokumentenverzeichnis genommen.

ARCHIVE	Angabe eines JAR-Archivs, aus dem die Klassendateien und sonstigen Ressourcen des Applets geladen werden sollen. Ein Beispiel zur Verwendung des ARCHIV-Parameters ist in Abschnitt 26.2.2 bei der Vorstellung von jar zu finden.
OBJECT	Name einer Datei, die den serialisierten Inhalt des Applets enthält.
ALT	Alternativer Text für solche Browser, die zwar das Applet-Tag verstehen, aber Java nicht unterstützen.
NAME	Eindeutiger Name für das Applet. Er kann zur Unterscheidung mehrerer, miteinander kommunizierender Applets auf einer Web-Seite verwendet werden.
ALIGN	Vertikale Anordnung des Applets in einer Textzeile. Hier kann einer der Werte left , right , top , texttop , middle , absmiddle , baseline , bottom oder absbottom angegeben werden.
VSPACE	Rand über und unter dem Applet.
HSPACE	Rand links und rechts vom Applet.

Tabelle 25.1: Optionale Parameter des APPLET-Tags

25.2.3 Parameterübergabe an Applets

Neben den Parametern des Applet-Tags gibt es die Möglichkeit, Parameter an das Applet selbst zu übergeben. Dazu kann innerhalb von `<APPLET>` und `</APPLET>` das optionale Tag `<PARAM>` verwendet werden. Jedes PARAM-Tag besitzt die beiden Parameter `name` und `value`, die den Namen und den Wert des zu übergebenden Parameters angeben.

Innerhalb des Applets können die Parameter mit der Methode `getParameter` der Klasse `Applet` abgefragt werden:

```
public String getParameter(String name)
```

Für jeden angegebenen Parameter liefert `getParameter` den zugehörigen Wert als `String`. Numerische Parameter müssen vor der weiteren Verwendung also erst konvertiert werden. Wird der angegebene Parameter nicht gefunden, gibt die Methode `null` zurück.

Das folgende Listing demonstriert den Einsatz von `getParameter` am Beispiel eines Applets, das eine rot-weiße Schranke zeichnet. Das Applet erwartet zwei Parameter `redwidth` und `whitewidth`, die die Breite des roten und weißen Abschnitts angeben. Diese werden in der `init`-Methode gelesen und dem Array `dx` zugewiesen. In `paint` wird dieses Array dann verwendet, um abwechselnd weiße und rote Parallelogramme der gewünschten Größe auszugeben. Insgesamt entsteht dadurch der Eindruck einer rot-weißen Schranke:

Beispiel

[Schranke.java](#)

```
001 /* Schranke.java */
002
003 import java.awt.*;
004 import java.applet.*;
005
006 public class Schranke
007 extends Applet
008 {
009     private int dx[];
010     private Color color[];
011
012     public void init()
013     {
014         String tmp;
015
016         dx = new int[2];
017         try {
018             dx[0] = Integer.parseInt(
019                 getParameter("redwidth")
020             );
021             dx[1] = Integer.parseInt(
022                 getParameter("whitewidth")
023             );
024         } catch (NumberFormatException e) {
025             dx[0] = 10;
```

```

026         dx[1] = 10;
027     }
028     color = new Color[2];
029     color[0] = Color.red;
030     color[1] = Color.white;
031 }
032
033 public void paint(Graphics g)
034 {
035     int maxX = getSize().width;
036     int maxY = getSize().height;
037     int x = 0;
038     int flg = 0;
039     Polygon p;
040     while (x <= maxX+maxY/2) {
041         p = new Polygon();
042         p.addPoint(x,0);
043         p.addPoint(x+dx[flg],0);
044         p.addPoint(x+dx[flg]-maxY/2,maxY);
045         p.addPoint(x-maxY/2,maxY);
046         p.addPoint(x,0);
047         g.setColor(color[flg]);
048         g.fillPolygon(p);
049         x += dx[flg];
050         flg = (flg==0) ? 1 : 0;
051     }
052 }
053 }

```

Listing 25.5: Ein parametrisiertes Applet

Das folgende HTML-Dokument zeigt die Einbindung eines Schranken-Applets mit einer Höhe von 10 Pixeln und einer Breite von 400 Pixeln. Die roten Felder der Schranke sind 10 und die weißen 7 Pixel breit:

[Schranke.html](#)

```

001 <html>
002 <head>
003 <title>Schranke</title>
004 </head>
005 <body>
006 <h1>Schranke</h1>
007 <applet code=Schranke.class width=400 height=10>
008 <param name="redwidth" value=10>
009 <param name="whitewidth" value=7>
010 Hier steht das Applet Schranke.class
011 </applet>
012 </body>
013 </html>

```

Listing 25.6: Die HTML-Datei zum Schranken-Applet

Beim Aufruf mit dem Internet Explorer sieht die Ausgabe des Applets so aus:

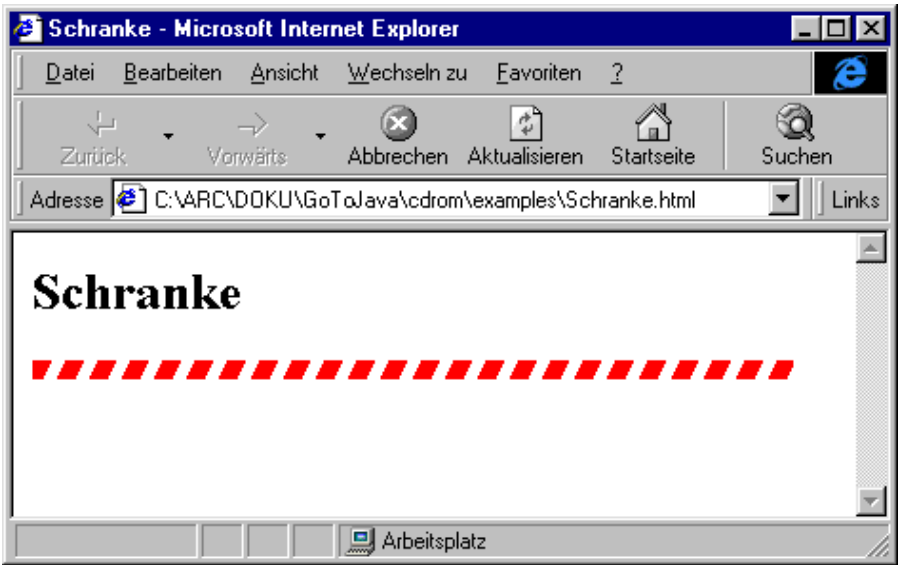


Abbildung 25.2: Darstellung des Schranken-Applets im Internet Explorer

Im Gegensatz zu einer Applikation wird ein Applet nicht direkt mit dem Java-Interpreter `java.exe` aufgerufen. Statt dessen wird es in eine HTML-Datei eingebunden und indirekt über den *Appletviewer* oder einen Web-Browser aufgerufen, der die HTML-Datei lädt. Unser Programm kann beispielsweise mit dem folgenden Kommando gestartet werden:

Hinweis

```
appletviewer Schranke.html
```

Auch die »echten« Web-Browser können meist mit einer Datei als Argument aufgerufen werden. Alternativ kann die HTML-Datei natürlich auch direkt aus dem laufenden Browser geladen werden. Der Applet-Viewer ist kein vollwertiger Browser, sondern extrahiert lediglich die `APPLET`-Tags und ihre Parameter, um die in der HTML-Datei angegebenen Applets zu starten.

25.3 Die Ausgabe von Sound

- [25.3 Die Ausgabe von Sound](#)
 - [25.3.1 Soundausgabe in Applets](#)
 - [25.3.2 Soundausgabe in Applikationen](#)

25.3.1 Soundausgabe in Applets

Das JDK bietet auch einige Möglichkeiten, Sound auszugeben. Hierbei muß klar zwischen dem JDK 1.2 und seinen Vorgängern unterschieden werden. Während das JDK 1.2 die Soundausgabe sowohl Applikationen als auch Applets ermöglicht, war diese vorher nur für Applets möglich. Dabei war die Ausgabe auf gesampelte Sounds beschränkt, die im AU-Format vorliegen mußten. Das AU-Format stammt aus der Sun-Welt und legt ein Sample im Format 8 Bit Mono, Sampling-Rate 8 kHz, µ-law-Kompression ab. Seit dem JDK 1.2 werden dagegen auch die Sample-Formate WAV und AIFF sowie die Midi-Formate *Typ 0* und *Typ 1* und RMF unterstützt. Zudem gibt es einige Shareware- oder Freeware-Tools, die zwischen verschiedenen Formaten konvertieren können (z.B. CoolEdit oder GoldWave).

Die Ausgabe von Sound ist denkbar einfach und kann auf zwei unterschiedliche Arten erfolgen. Zum einen stellt die Klasse [Applet](#) die Methode [play](#) zur Verfügung, mit der eine Sound-Datei geladen und abgespielt werden kann:

```

public void play(URL url)
public void play(URL url, String name)

```

Hierbei kann entweder der *URL* einer Sound-Datei (siehe nächster Abschnitt) oder die Kombination von Verzeichnis-URL und Dateinamen angegeben werden. Üblicherweise wird zur Übergabe des Verzeichnis-URLs eine der [Applet](#)-Methoden [getCodeBase](#) oder [getDocumentBase](#) verwendet. Diese liefern einen URL des Verzeichnisses, aus dem das Applet gestartet wurde bzw. in dem die aktuelle HTML-Seite liegt:

```

public URL getCodeBase()
public URL getDocumentBase()

```

Der Nachteil dieser Vorgehensweise ist, daß die Sound-Datei bei jedem Aufruf neu geladen werden muß. In der zweiten Variante wird zunächst durch einen Aufruf von [getAudioClip](#) ein Objekt der Klasse [AudioClip](#) beschafft, das dann beliebig oft abgespielt werden kann:

```

public getAudioClip(URL url, String name)

```

[AudioClip](#) stellt die drei Methoden [play](#), [loop](#) und [stop](#) zur Verfügung:

```

public void play()
public void loop()
public void stop()

```

[play](#) startet die zuvor geladene Sound-Datei und spielt sie genau einmal ab. [loop](#) startet sie ebenfalls, spielt den Sound in einer Endlosschleife aber immer wieder ab. Durch Aufruf von [stop](#) kann diese Schleife beendet werden. Es ist auch möglich, mehr als einen Sound gleichzeitig abzuspielen. So kann beispielsweise eine Hintergrundmelodie in einer Schleife immer wieder abgespielt werden, ohne daß die Ausgabe von zusätzlichen Vordergrund-Sounds beeinträchtigt würde.

Das folgende Beispiel ist eine neue Variante des »Hello, World«-Programms. Anstatt der textuellen Ausgabe stellt das Applet zwei Buttons zur Verfügung, mit denen die Worte »Hello« und »World« abgespielt werden können:

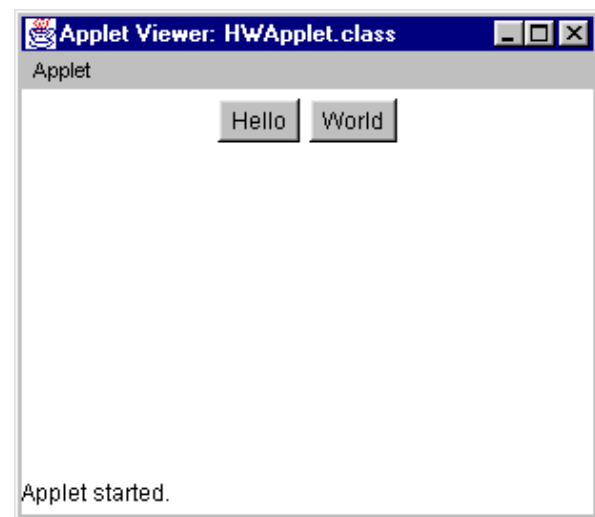


Abbildung 25.3: Das sprechende »Hello, World«-Programm

Hier ist der Sourcecode des Programms:

[HWApplet.java](#)

```

001  /* HWApplet.java */
002
003  import java.awt.*;
004  import java.awt.event.*;
005  import java.applet.*;
006
007  public class HWApplet
008  extends Applet
009  implements ActionListener
010  {
011      Button    hello;
012      Button    world;
013      AudioClip helloClip;
014      AudioClip worldClip;
015
016      public void init()
017      {
018          super.init();
019          setLayout(new FlowLayout());
020          hello = new Button("Hello");
021          hello.addActionListener(this);
022          add(hello);
023          world = new Button("World");
024          world.addActionListener(this);
025          add(world);
026          helloClip = getAudioClip(getCodeBase(),"hello.au");
027          worldClip = getAudioClip(getCodeBase(),"world.au");
028      }
029
030      public void actionPerformed(ActionEvent event)
031      {
032          String cmd = event.getActionCommand();
033          if (cmd.equals("Hello")) {
034              helloClip.play();
035          } else if (cmd.equals("World")) {
036              worldClip.play();
037          }
038      }
039  }

```

Listing 25.7: Das sprechende »Hello, World«

Eine HTML-Datei `HWApplet.html` zum Aufruf dieses Applets findet sich in [Abschnitt 26.2.2](#). Dort wird die Einbindung in Zusammenhang mit der Verwendung von jar-Dateien erläutert.

Hinweis

25.3.2 Soundausgabe in Applikationen

Seit dem JDK 1.2 kann nicht nur in Applets, sondern auch in Applikationen Sound ausgegeben werden. Dazu bietet die Klasse [Applet](#) eine statische Methode [newAudioClip](#):

JDK1.1/1.2

```
public static AudioClip newAudioClip(URL url)
```

Da es sich um eine Klassenmethode handelt, kann sie auch außerhalb eines Applets aufgerufen werden. Das folgende Beispiel zeigt ein einfaches Programm, das in der Kommandozeile den URL einer Sounddatei erwartet und diese dann maximal 10 Sekunden lang abspielt:

[PlaySound.java](#)

```
001 /* PlaySound.java */
002
003 import java.net.*;
004 import java.applet.*;
005
006 public class PlaySound
007 {
008     public static void main(String args[])
009     {
010         if (args.length >= 1) {
011             try {
012                 URL url = new URL(args[0]);
013                 AudioClip clip = Applet.newAudioClip(url);
014                 clip.play();
015                 try {
016                     Thread.sleep(10000);
017                 } catch (InterruptedException e) {
018                 }
019             } catch (MalformedURLException e) {
020                 System.out.println(e.toString());
021             }
022         }
023     }
024 }
```

Listing 25.8: Soundausgabe aus einer Applikation

Das Programm kann beispielsweise dazu verwendet werden, einige der Standard-Sounddateien unter Windows abzuspielen:

```
java PlaySound file:///c:/windows/media/passport.mid
java PlaySound file:///c:/windows/media/dermic~1.wav
```

25.4 Verweise auf andere Seiten

- [25.4 Verweise auf andere Seiten](#)
 - [25.4.1 Die Klasse URL](#)
 - [25.4.2 Der Applet-Kontext](#)
 - [25.4.3 Die Methode showDocument](#)

25.4.1 Die Klasse URL

Das Konzept der *URLs* ist eines der wichtigsten im ganzen Web. Ein URL (*Uniform Resource Locator*) ist ein universelles Hilfsmittel zur Darstellung von Internet-Adressen. Hinter jedem Link, der sich auf einem HTML-Dokument verbirgt, steht ein URL, der beim Aufrufen des Links angesprungen wird.

Der Aufbau von URLs ist in seiner Vielfältigkeit recht komplex und ermöglicht es, sehr unterschiedliche Typen von Adressen zu verwalten. Ein URL-Link kann beispielsweise verwendet werden, um eine andere Web-Seite aufzurufen, eine Datei zu laden oder elektronische Post an einen anderen Anwender zu senden. Wir wollen hier nur auf die URLs eingehen, die man zur Darstellung von Web-Seiten-Adressen verwendet. Eine exakte Beschreibung des URL-Konzepts befindet sich in RFC 1630.

Ein Uniform Resource Locator besteht aus mehreren Teilen. Als erstes wird ein *Dienst* angegeben, der beschreibt, auf welche Art von Service die Adresse verweist. Typische Dienste sind *html* für Web-Seiten, *ftp* für einen File-Download oder *mailto*, um eine Mail zu versenden.

Bei einer HTML-Adresse besteht der URL aus drei weiteren Teilen:

- Dem *Host-Namen* (z.B. [java.sun.com](#)), der vom Dienst durch die Zeichenkette "://" getrennt wird.
- Einer optionalen *Port-Nummer* (z.B. 80 für einen *http*-Dämon), die vom Host-Namen durch einen ":" abgetrennt wird.
- Einem Dateinamen (z.B. *bugsandfeatures.html*), der vom Host-Namen bzw. der Port-Nummer durch "/" abgetrennt wird. Diesem Dateinamen kann auch die Rolle eines Kommandos oder Parameterstrings zukommen, beispielsweise, wenn ein *CGI-Script* aufgerufen wird.

Gültige URLs sind also [http://java.sun.com/bugsandfeatures.html](#) oder auch [http://www.yahoo.com](#). Die meisten Browser und Server sind in der Lage, fehlende Bestandteile eines URLs weitgehend zu ergänzen. Fehlt beispielsweise die Dateierweiterung, wird *.html* angehängt. Bezeichnet der URL lediglich ein Verzeichnis, wird ein Standard-Dateiname wie beispielsweise *index.html* oder *default.htm* angehängt.

Beispiel

Java implementiert das Konzept eines Uniform Resource Locators durch eine eigene Klasse *URL*, die sich im Paket [java.net](#) befindet. Diese dient dazu, die Syntax von URLs zu kapseln und die einzelnen Bestandteile voneinander zu trennen. Die Klasse *URL* besitzt vier Konstruktoren, die es ermöglichen, eine Adresse auf verschiedene Art zusammenzubauen. Wir werden hier nur die Variante verwenden, die einen *String* als Argument akzeptiert:

[java.net.URL](#)

```
public URL(String url)
    throws MalformedURLException
```

Bei der Anwendung dieses Konstruktors muß ein syntaktisch einwandfreier URL als *String* übergeben werden. Enthält der String einen Syntaxfehler, löst der Konstruktor eine Ausnahme des Typs *MalformedURLException* aus.

An dieser Stelle wird natürlich noch nicht überprüft, ob die angegebene Adresse *wirklich existiert*. Dazu wäre eine funktionsfähige Netzwerkverbindung nötig, und es würde ein in der Regel nicht akzeptabler Aufwand bei der Konstruktion von URL-Objekten entstehen. Die im Konstruktor durchgeführten Überprüfungen sind lediglich *syntaktischer* Natur, um sicherzustellen, daß ein URL ein gültiges Adressenformat hat. Nachdem ein gültiges *URL*-Objekt erzeugt wurde, kann es zur Adressenübergabe verwendet werden.

Weitere Hinweise zur Anwendung der Klasse *URL* finden sich in [Abschnitt 32.4](#).

Hinweis

25.4.2 Der Applet-Kontext

Ein anderes Konzept, das bei der Programmierung von Links eine Rolle spielt, ist das des *Applet-Kontexts*. Hierunter versteht Java *das Programm*, das das aktuelle Applet ausführt. Dies ist in der Regel der Browser, der das Applet geladen hat; während der Entwicklung und Testphase des Programms kann es natürlich auch der *Appletviewer* sein.

Mit Hilfe der Methode [getAppletContext](#) kann ein Objekt des Typs *AppletContext* beschafft werden:

[java.applet.Applet](#)

```
public AppletContext getAppletContext()
```

In der Klasse [AppletContext](#) gibt es eine Reihe von Methoden, die dem Applet Funktionalitäten des Browsers zur Verfügung stellen:

- [getApplets](#) liefert eine Liste aller laufenden Applets
- [getApplet](#) liefert ein einzelnes Applet, wenn sein Name bekannt ist
- [showDocument](#) erlaubt das Laden eines anderen HTML-Dokuments

Wir wollen auf die ersten beiden Methoden nicht weiter eingehen, sondern uns hier lediglich mit [showDocument](#) beschäftigen.

25.4.3 Die Methode showDocument

Die Methode [showDocument](#) kann dazu verwendet werden, eine andere Web-Seite zu laden. Dies führt dazu, daß das aktive Applet angehalten wird und die Kontrolle über die Web-Seite verliert. Befindet sich auf der neuen Web-Seite ein anderes Applet, so wird dieses geladen bzw. - falls es bereits geladen war - reaktiviert.

Die Methode [showDocument](#) steht in zwei unterschiedlichen Varianten zur Verfügung:

[java.applet.AppletContext](#)

```
public void showDocument(URL url)

public void showDocument(URL url, String name)
```

In seiner einfachsten Form erwartet [showDocument](#) lediglich einen einzigen Parameter, nämlich die Adresse der Zielseite. Hier muß ein gültiges [URL](#)-Objekt angegeben werden, also eine komplette Adresse mit Dienst, Host-Name, Verzeichnis und Datei. In der zweiten Variante kann zusätzlich ein *target* angegeben werden, also das Sprungziel innerhalb der ausgewählten Datei. Hier können die Standard-HTML-Targets *_self*, *_parent*, *_top* und *_blank* sowie alle benutzerdefinierten Sprungmarken verwendet werden.

Soll statt einer absoluten eine relative Adresse angegeben werden , also beispielsweise eine andere Web-Seite aus demselben Verzeichnis oder aus einem seiner Unterverzeichnisse, muß das [URL](#)-Objekt mit einem erweiterten Konstruktor instanziiert werden. Zusätzlich zur Angabe des relativen URL-Strings ist als erstes Argument ein *Kontext-URL* anzugeben, der das Basisverzeichnis für das zweite Argument vorgibt. Mit Hilfe der Methoden [getCodeBase](#) und [getDocumentBase](#) der Klasse [Applet](#) kann beispielsweise das Stammverzeichnis des Applets bzw. der HTML-Seite ermittelt werden. Als zweites Argument muß dann lediglich der Dateiname relativ zu diesem Verzeichnis angegeben werden.

Tip

Das folgende Applet demonstriert die Anwendung von [showDocument](#). Es erstellt eine Reihe von Buttons und zeigt sie auf dem Bildschirm an. Wird einer der Buttons gedrückt, verzweigt das Programm auf die durch den Button spezifizierte Seite. Während der Initialisierung sucht das Applet nach Parametern mit den Namen [button1](#), [button2](#) usw. Jeder dieser Parameter sollte den Namen des Buttons und die Zieladresse enthalten. Die beiden Teile müssen durch ein Komma getrennt sein. Eine Besonderheit besteht darin, daß auch Adressen angegeben werden können, die mit *=* anfangen. In diesem Fall wird das *=* entfernt und der interne URL nicht absolut, sondern unter Verwendung des erweiterten Konstruktors relativ zur aktuellen Web-Seite angelegt.

Beispiel

Um die Buttons sowohl auf dem Bildschirm anzeigen zu können als auch den zugehörigen URL zu speichern, wurde eine neue Klasse [URLButton](#) definiert. [URLButton](#) erweitert die Klasse [Button](#) um die Fähigkeit, einen URL mitzuspeichern und bei Bedarf abzufragen.

Hier ist das Listing:

[URLLaden.java](#)

```
001 /* URLLaden.java */
002
003 import java.applet.*;
004 import java.awt.*;
005 import java.awt.event.*;
006 import java.util.*;
007 import java.net.*;
008
009 class URLButton
010 extends Button
011 {
012     private URL url;
013
014     public URLButton(String label, URL url)
015     {
016         super(label);
017         this.url = url;
```

```

018     }
019
020     public URL getURL()
021     {
022         return url;
023     }
024 }
025
026 public class URLLaden
027 extends Applet
028 implements ActionListener
029 {
030     Vector buttons;
031
032     public void init()
033     {
034         super.init();
035         setLayout(new FlowLayout());
036         addNotify();
037         buttons = new Vector();
038         for (int i=1; ; ++i) {
039             String s = getParameter("button"+i);
040             if (s == null) {
041                 break;
042             }
043             try {
044                 StringTokenizer st = new StringTokenizer(s,",");
045                 String label = st.nextToken();
046                 String urlstring = st.nextToken();
047                 URL url;
048                 if (urlstring.charAt(0) == '=') {
049                     urlstring = urlstring.substring(1);
050                     url = new URL(getDocumentBase(),urlstring);
051                 } else {
052                     url = new URL(urlstring);
053                 }
054                 URLButton button = new URLButton(label,url);
055                 button.addActionListener(this);
056                 add(button);
057                 buttons.addElement(button);
058             } catch (NoSuchElementException e) {
059                 System.out.println("Button"+i+": "+e.toString());
060                 break;
061             } catch (MalformedURLException e) {
062                 System.out.println("Button"+i+": "+e.toString());
063                 break;
064             }
065         }
066     }
067
068     public void actionPerformed(ActionEvent event)
069     {
070         URLButton source = (URLButton)event.getSource();
071         Enumeration en = buttons.elements();
072         while (en.hasMoreElements()) {
073             URLButton button = (URLButton)en.nextElement();
074             if (button == source) {
075                 System.out.println(
076                     "showDocument("+button.getURL().toString()+")"
077                 );
078                 getAppletContext().showDocument(button.getURL());
079             }
080         }
081     }
082 }

```

Um die wichtigsten Fälle zu demonstrieren, wurde das Applet mit der folgenden HTML-Datei gestartet:

[URLLaden.html](#)

```
001 <html>
002 <head>
003 <title>URLLaden</title>
004 </head>
005 <body>
006 <h1>URLLaden</h1>
007 <applet code=URLLaden.class width=400 height=200>
008 <param
009     name="button1"
010     value="JAVA-Home,http://java.sun.com/">
011 <param
012     name="button2"
013     value="Guido-Home,http://www.gkrueger.com">
014 <param
015     name="button3"
016     value="Schranke,=Schranke.html">
017 <param
018     name="button4"
019     value="Jana30,=images/jana30.gif">
020 Hier steht das Applet URLLaden.class
021 </applet>
022 </body>
023 </html>
```

Listing 25.10: Die HTML-Datei zum Laden der Webseiten

Hier werden vier verschiedene Buttons definiert:

Hinweis

- Der Button *JAVA-Home* verzweigt auf die Java-Seite von SUN.
- Der Button *Guido-Home* verzweigt auf die Homepage des Autors.
- Der Button *Schranke* lädt das Beispiel [Schranke](#), das sich in demselben Verzeichnis wie die aktuelle HTML-Seite befindet.
- Der Button *Jana30* schließlich demonstriert zwei weitere Features. Erstens, daß es bei den meisten Browsern auch möglich ist, Grafikdateien (beispielsweise im GIF-Format) zu laden. Zweitens, wie man mit Unterverzeichnissen umgeht (Pfadtrennzeichen in UNIX-Manier).

Die Ausgabe des Programms im Internet Explorer sieht folgendermaßen aus:



Abbildung 25.4: Darstellung von URLLaden im Internet Explorer

25.5 Animation in Applets

- [25.5 Animation in Applets](#)

Animation ist in Applets ebenso möglich wie in Applikationen. Alle Techniken, die in [Kapitel 24](#) erklärt wurden, sind grundsätzlich auch auf Applets anwendbar. Aufgrund der Tatsache, daß Applets in einem Browser laufen und über eine Netzwerk-Fernverbindung mit Daten versorgt werden müssen, sollten folgende Besonderheiten beachtet werden:

- Ein animiertes Applet sollte in einem eigenen Thread laufen. Andernfalls würde möglicherweise der Browser lahmgelegt werden und nicht mehr auf Benutzereingaben reagieren.
- Die Animation in Applets kann durch Sound unterstützt werden.
- Das Laden von Sound- und Image-Dateien kann unter Umständen sehr lange dauern und liegt in der Regel um Größenordnungen über den Ladezeiten im lokalen Betrieb. Müssen während der Applet-Initialisierung größere Dateien geladen werden, kann das Starten des Applets für Anwender mit langsamen Netzwerkverbindungen schwierig werden.

Wir wollen uns diese Regeln zu eigen machen und in diesem Abschnitt ein einfaches animiertes Applet entwickeln. Das Programm soll die Skyline einer Großstadt bei Nacht darstellen. Dabei gehen in den Wolkenkratzern die Lichter an und aus, auf einigen Dächern gibt es rote Blinklichter, und von Zeit zu Zeit schlägt der Blitz mit Krachen in einen der Wolkenkratzer ein. (Diese Aufgabenstellung erinnert nicht ganz zu Unrecht an einen bekannten Bildschirmschoner.) [Abbildung 25.5](#) zeigt einen Schnappschuß des laufenden Programms.

Beispiel

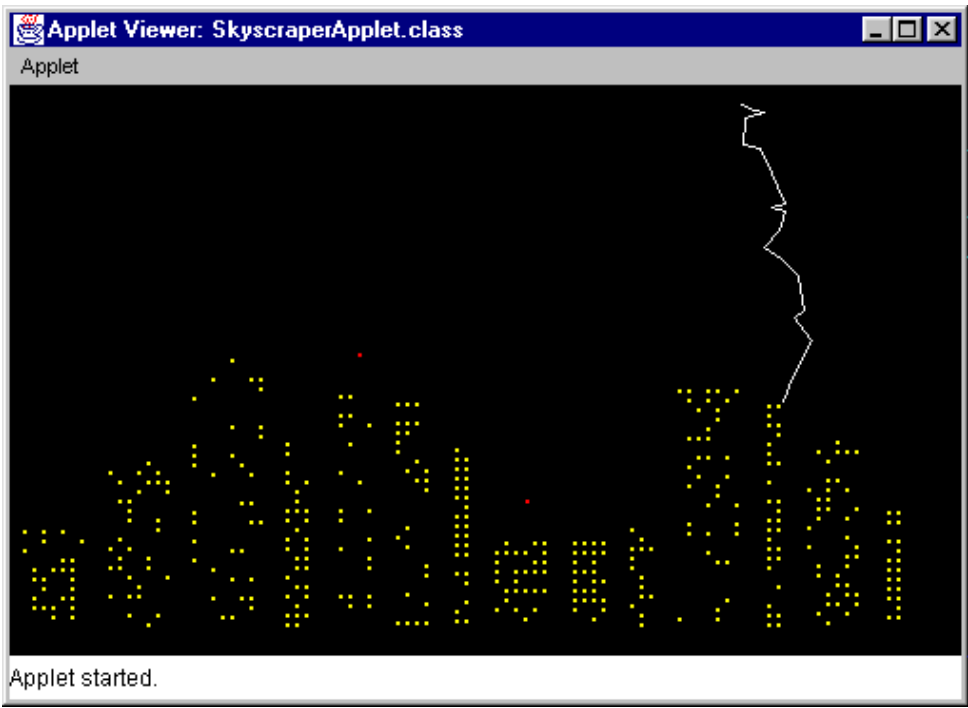


Abbildung 25.5: Das Wolkenkratzer-Beispielprogramm

Das Programm implementiert zwei Klassen, [Skyscraper](#) und [SkyscraperApplet](#). [Skyscraper](#) repräsentiert einen Wolkenkratzer, der die Membervariablen *x*- und *y*-Position, *Höhe*, *Breite* und *Anzahl der Fenster* in *x*- und *y*-Richtung besitzt. Zusätzlich kann ein [Skyscraper](#)-Objekt auf Simulations-Events reagieren, die durch den Aufruf der Methode [LightEvent](#) ausgelöst werden. In diesem Fall wird das Licht in einem der Fenster an- oder ausgeschaltet oder das rote Blinklicht auf dem Dach getriggert.

Das eigentliche Applet wird durch die Klasse [SkyscraperApplet](#) realisiert. In der [init](#)-Methode wird zunächst eine Reihe von [Skyscraper](#)-Objekten erzeugt und im [Vector c](#) abgelegt. Zusätzlich werden die Parameter [DELAY](#), [FLASH](#) und [THUNDER](#) eingelesen, die zur Steuerung der Animationsverzögerung, der Blitzwahrscheinlichkeit und der Sound-Datei für den Donner dienen. In der Methode [start](#) wird ein Thread erzeugt, so daß die eigentliche Animation mit der [repaint](#)-Schleife in [run](#) abläuft. Um das Bildschirmflackern zu verringern, wird [update](#) überlagert, wie in [Kapitel 24](#) erläutert. In [paint](#) wird per Zufallszahlengenerator eines der in [v](#) gespeicherten [Skyscraper](#)-Objekte ausgewählt und dessen [LigthEvent](#)-Methode aufgerufen, um ein Beleuchtungsereignis zu simulieren.

Manchmal wird auch noch die Methode [Lightning](#) aufgerufen, um einen Blitzeinschlag darzustellen. Ein Blitz wird dabei durch einen Streckenzug vom oberen Bildrand bis zur Dachspitze eines Hochhauses dargestellt. Dieser Streckenzug wird für einen kurzen Augenblick in weißer Farbe auf den Bildschirm gezeichnet und anschließend durch erneutes Zeichnen in schwarzer Farbe wieder entfernt. Um ein realistisches Flackern zu erreichen, wird dieser Vorgang noch einmal wiederholt. Unmittelbar vor der Darstellung des Blitzes wird das zuvor geladene Donnergeräusch abgespielt.

Hier ist der Quellcode des Applets:

[SkyscraperApplet.java](#)

```
001 /* SkyscraperApplet.java */
002
003 import java.awt.*;
004 import java.util.*;
005 import java.applet.*;
006
007 class Skyscraper
008 {
009     public int x;
010     public int y;
011     public int width;
012     public int height;
013     int      wndcntx;
014     int      wndcnty;
015     boolean  blinkon = false;
016
017     Skyscraper(int x, int y)
018     {
019         this.x = x;
020         this.y = y;
021         this.width = (int)(30*(0.5+Math.random()));
022         this.height = (int)(100*(0.5+Math.random()));
023         wndcntx = (width-4)/5;
024         wndcnty = (height-4)/5;
025     }
026
027     void LightEvent(Graphics g)
028     {
029         double rnd = Math.random();
030         int xwnd = (int)(Math.random()*wndcntx);
031         int ywnd = (int)(Math.random()*wndcnty);
032         if (blinkon) {
033             g.setColor(Color.black);
034             g.fillRect(x+width/2,y-height-20,2,2);
035             blinkon = false;
036         }
037         if (rnd >= 0.9) {
038             blinkon = true;
039             g.setColor(Color.red);
040             g.fillRect(x+width/2,y-height-20,2,2);
041         } else if (rnd >= 0.7) {
042             g.setColor(Color.black);
043             g.fillRect(x+2+xwnd*5,y-height+2+ywnd*5,2,2);
044         } else {
045             g.setColor(Color.yellow);
046             g.fillRect(x+2+xwnd*5,y-height+2+ywnd*5,2,2);
047         }
048     }
049 }
050
051 public class SkyscraperApplet
052 extends Applet
053 implements Runnable
054 {
055     //Membervariablen
056     Thread th;
057     Vector v = new Vector();
```

```
058 AudioClip thunder;
059 boolean running;
060
061 //Parameter
062 int DELAY;
063 float FLASH;
064 String THUNDER;
065
066 public void init()
067 {
068     Skyscraper house;
069     int x = 5;
070
071     //Häuser erzeugen
072     while (this.getSize().width-x-1 >= 30) {
073         house = new Skyscraper(x,this.getSize().height-10);
074         v.addElement(house);
075         x += house.width + 5;
076     }
077     setBackground(Color.black);
078
079     //Parameter einlesen
080     try {
081         DELAY = Integer.parseInt(getParameter("delay"));
082     } catch (NumberFormatException e) {
083         DELAY = 75;
084     }
085     try {
086         FLASH = (new Float(getParameter("flash"))).floatValue();
087     } catch (NumberFormatException e) {
088         FLASH = 0.01F;
089     }
090     THUNDER = getParameter("thunder");
091     if (THUNDER != null) {
092         thunder = getAudioClip(getCodeBase(),THUNDER);
093     }
094     System.out.println("DELAY = "+DELAY);
095     System.out.println("FLASH = "+FLASH);
096     System.out.println("THUNDER = "+THUNDER);
097 }
098
099 public void start()
100 {
101     if (th == null) {
102         running = true;
103         th = new Thread(this);
104         th.start();
105     }
106 }
107
108 public void stop()
109 {
110     if (th != null) {
111         running = false;
112         th = null;
113     }
114 }
115
116 public void run()
117 {
118     while (running) {
119         repaint();
120         try {
121             Thread.sleep(DELAY);
122         } catch (InterruptedException e) {
```

```

123         //nothing
124     }
125 }
126 }
127
128 public void update(Graphics g)
129 {
130     paint(g);
131 }
132
133 public void paint(Graphics g)
134 {
135     int i;
136     Skyscraper house;
137
138     i = (int)Math.floor(Math.random()*v.size());
139     house = (Skyscraper)v.elementAt(i);
140     house.LightEvent(g);
141     if (Math.random() < FLASH) {
142         Lightning(g,house.x+10,house.y-house.height);
143     }
144 }
145
146 public void Lightning(Graphics g, int x, int y)
147 {
148     Vector poly = new Vector();
149     int dx, dy, i, polysize;
150
151     thunder.play();
152     //Blitzpolygon berechnen
153     poly.addElement(new Point(x,y));
154     polysize = 1;
155     while (y > 10) {
156         dx = 10 - (int)(Math.floor(Math.random()*20));
157         dy = - (int)(Math.floor(Math.random()*20));
158         x += dx;
159         y += dy;
160         poly.addElement(new Point(x,y));
161         ++polysize;
162     }
163     //Blitzvector in Koordinaten-Arrays umwandeln
164     int xpoints[] = new int[poly.size()];
165     int ypoints[] = new int[poly.size()];
166     for (i = 0; i < polysize; ++i) {
167         Point p = (Point)poly.elementAt(i);
168         xpoints[i] = p.x;
169         ypoints[i] = p.y;
170     }
171     //Blitz zeichnen
172     for (i = 0; i <= 1; ++i) {
173         g.setColor(Color.white);
174         g.drawPolyline(xpoints, ypoints, polysize);
175         try {
176             Thread.sleep(20);
177         } catch (InterruptedException e) {}
178         g.setColor(Color.black);
179         g.drawPolyline(xpoints, ypoints, polysize);
180         try {
181             Thread.sleep(20);
182         } catch (InterruptedException e) {}
183     }
184 }
185 }

```

Listing 25.11: Das Wolkenkratzer-Applet

Zum Aufruf des Applets kann beispielsweise folgende HTML-Datei verwendet werden:

[SkyscraperApplet.html](#)

```
001 <html>
002 <head>
003 <title>Skyscraper</title>
004 </head>
005 <body>
006 <h1>Skyscraper</h1>
007 <applet code=SkyscraperApplet.class width=500 height=300>
008 <param name="delay" value=75>
009 <param name="flash" value=0.01>
010 <param name="thunder" value="thunder.au">
011 Hier steht das Applet Skyscraper.class
012 </applet>
013 </body>
014 </html>
```

Listing 25.12: Die HTML-Datei zum Aufrufen des Wolkenkratzer-Applets

25.6 Zusammenfassung

- [25.6 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die Architektur eines Applets
- Unterschiede zwischen Applets und Applikationen
- Die Klasse [Applet](#)
- Zustandssteuerung eines Applets durch Aufruf der Methoden [init](#), [start](#), [stop](#) und [destroy](#)
- Die Methoden [showStatus](#), [getAppletInfo](#) und [getParameterInfo](#)
- Einbinden eines Applets in eine HTML-Datei
- Das APPLET-Tag und seine Parameter
- Übergabe von Parametern an ein Applet und die Methode [getParameter](#)
- Ausgabe von Sound
- Die Methoden [getDocumentBase](#) und [getCodeBase](#)
- Das Konzept der Uniform Resource Locators, die Klasse [URL](#) und der Aufruf eines anderen Dokuments mit [showDocument](#)
- Der Applet-Kontext eines Applets
- Animation in Applets

Kapitel 26

Die Werkzeuge des JDK

- [26 Die Werkzeuge des JDK](#)
 - [26.1 Entwicklungswerkzeuge](#)
 - [26.1.1 javac - Der Compiler](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Optionen](#)
 - [26.1.2 java - Der Interpreter](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Optionen](#)
 - [26.1.3 appletviewer - Der Appletviewer](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Optionen](#)
 - [26.1.4 jdb - Der Debugger](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Vorbereitungen](#)
 - [26.2 Werkzeuge zur Dokumentation und Archivierung](#)
 - [26.2.1 javadoc - Der Dokumentationsgenerator](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Dokumentationskommentare](#)
 - [Aufruf von javadoc](#)
 - [Optionen](#)
 - [26.2.2 jar - Das Archivierungswerkzeug](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Kommandos](#)
 - [Verwendung von jar-Dateien in Applets](#)
 - [26.2.3 javap - Der Disassembler](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Optionen](#)
 - [26.3 Zusammenfassung](#)

26.1 Entwicklungswerkzeuge

- [26.1 Entwicklungswerkzeuge](#)
 - [26.1.1 javac - Der Compiler](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Optionen](#)
 - [26.1.2 java - Der Interpreter](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Optionen](#)
 - [26.1.3 appletviewer - Der Appletviewer](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Optionen](#)
 - [26.1.4 jdb - Der Debugger](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Vorbereitungen](#)

In diesem Kapitel wollen wir die wichtigsten Werkzeuge des Java Development Kit vorstellen. Alle Programme arbeiten kommandozeilenorientiert und werden aus einer DOS-Box oder einer Shell heraus aufgerufen. Wir stellen die Aufrufsyntax und die wichtigsten Optionen der folgenden Programme vor:

- [javac](#) - Der Compiler
- [java](#) - Der Interpreter
- [appletviewer](#) - Der Applet-Interpreter
- [jdb](#) - Der Debugger
- [javadoc](#) - Der Dokumentationsgenerator
- [jar](#) - Das Archivierungswerkzeug
- [javap](#) - Der Disassembler

Das JDK stellt darüber hinaus weitere Werkzeuge zur Verfügung, auf die wir hier nicht näher eingehen wollen. Dazu zählen beispielsweise [javah](#), [rmix](#) oder [keytool](#). Sie sind Bestandteil spezieller APIs und werden in der JDK-Dokumentation ausführlich beschrieben.

26.1.1 javac - Der Compiler

Aufruf

```
javac [ options ] [@filelist | { filename.java }]
```

Beschreibung

Der Compiler [javac](#) übersetzt Sourcecode in Bytecode. Zu jeder Klasse, die innerhalb einer [.java](#)-Datei definiert wurde, wird eine eigene [.class](#)-Datei erzeugt. Der Compiler wacht automatisch über die Abhängigkeiten zwischen den Quelldateien. Wird beispielsweise in einer Klasse [X](#) in der Datei [X.java](#) eine Klasse [Y](#), die in [Y.java](#) liegt, verwendet, so wird [Y.java](#) automatisch mit übersetzt, wenn es erforderlich ist. Anstelle der Liste von Dateinamen kann nach einem @ auch eine Textdatei angegeben werden, in der die zu übersetzenden Dateien durch Whitespaces getrennt angegeben werden.

Optionen

Option	Bedeutung
<code>-classpath path</code>	Gibt die Liste der Pfade zur Suche von Klassendateien an. Dieser Schalter übersteuert den Wert einer eventuell gesetzten Umgebungsvariable CLASSPATH . Bezüglich der Auswirkungen des Klassenpfades in den unterschiedlichen JDK-Versionen lesen Sie bitte Abschnitt 8.2.2 .
<code>-g</code>	Aktiviert die Erzeugung von Debug-Informationen. Dieser Schalter sollte aktiviert werden, bevor ein Programm debugt wird.
<code>-nowarn</code>	Deaktiviert die Ausgabe von Warnungen.
<code>-O</code>	Schaltet den Code-Optimierer an. Dessen Fähigkeiten sind auch in aktuellen Versionen des JDK 1.2 noch nicht sehr ausgeprägt und beschränken sich vorwiegend auf das Expandieren von Methodenaufrufen.
<code>-verbose</code>	Aktiviert die Ausgabe von Meldungen über geladene Quell- und Klassendateien während der Übersetzung.
<code>-depend</code>	Normalerweise wird eine Quelldatei neu kompiliert, wenn das Datum der letzten Änderung nach dem Änderungsdatum der Klassendatei liegt oder wenn die Klassendatei ganz fehlt. Mit Hilfe dieses Schalters wird diese Entscheidung auf der Basis von Headerinformationen in der Klassendatei auf eine zuverlässigere Weise getroffen. Der Übersetzungsvorgang wird aber unter Umständen sehr viel langsamer.
<code>-deprecation</code>	Sorgt dafür, daß bei jedem Aufruf einer als deprecated markierten Methode (@deprecated -Marke im Dokumentationskommentar) zusätzliche Informationen über mögliche Workarounds ausgegeben werden. Alle Methoden aus älteren JDKs, die im aktuellen JDK nicht mehr verwendet werden sollen, werden mit diesem Flag markiert.

Tabelle 26.1: Optionen von `javac`

26.1.2 java - Der Interpreter

Aufruf

```
java [ options ] classname [{ args }]
javaw [ options ] classname [{ args }]
```

Beschreibung

Der Interpreter [java](#) dient dazu, kompilierte Java-Programme auszuführen, die als Bytecode in [.class](#)-Dateien vorliegen. [javaw](#) erfüllt denselben Zweck, erzeugt aber kein Terminalfenster beim Start des Programms und erlaubt nicht die Verwendung der Standard-Streams [System.in](#), [System.out](#) und [System.err](#). Beim Aufruf beider Programme wird der Name einer Klassendatei erwartet (ohne die Erweiterung [.class](#)). Damit sie ausgeführt werden kann, muß sie eine Klassenmethode [main](#) mit folgender Signatur enthalten:

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

Alle Argumente, die nach dem Namen der Klassendatei an [java](#) übergeben werden, stehen nach dem Aufruf von [main](#) in [args](#) zur Verfügung. Der Java-Interpreter wird nach dem Rücksprung aus [main](#) beendet, wenn keine eigenen Threads erzeugt wurden. Falls weitere Threads erzeugt wurden, wird er beendet, nachdem der letzte Vordergrund-Thread beendet wurde.

Da während der Ausführung eines Java-Programms meist weitere Klassendateien benötigt werden, muß der Interpreter wissen, wo diese zu finden sind. Standardmäßig sucht er dabei im systemspezifischen Installationsverzeichnis und im aktuellen Verzeichnis. Die Suchstrategie kann durch Setzen der Umgebungsvariable [CLASSPATH](#) oder mit Hilfe der Option `-classpath` verändert werden. Sollen nur die Standardbibliotheken des JDK verwendet werden, sind weder [CLASSPATH](#) noch `-classpath` erforderlich. Weitere Informationen zu [CLASSPATH](#)-Einstellungen finden Sie in [Abschnitt 8.2.2](#).

In der Windows-Version des JDK 1.2 ist ein *Just-In-Time-Compiler* (kurz *JIT*) enthalten, der standardmäßig aktiviert ist. Der JIT übersetzt zur Laufzeit des Programmes häufig benötigte Bytecodes in Maschinencode und beschleunigt so die weitere Ausführung des Programms. Soll der JIT deaktiviert werden, kann die Umgebungsvariable `JAVA_COMPILER` oder die Systemeigenschaft `java.compiler` auf den Wert `NONE` gesetzt werden.

JDK1.1/1.2

Optionen

Option	Bedeutung
<code>-classpath path</code>	Gibt die Liste der Pfade zur Suche von Klassendateien an. Alternativ kann auch der Schalter <code>-cp</code> verwendet werden.
<code>-prof</code>	Aktiviert in Prä-1.2-Versionen des JDK den <i>Profiler</i> im Interpreter, der Informationen über das Laufzeitverhalten der Anwendung in die Datei <code>java.prof</code> schreibt. Ab dem JDK 1.2 wird der Profiler mit der Option <code>-Xprof</code> aktiviert. Genaue Informationen zur Verwendung des Profilers sind in Abschnitt 29.3 zu finden.
<code>-version</code>	Ausgabe der Versionsnummer.
<code>-help</code>	Ausgabe eines kurzen Hilfetextes.
<code>-verbose</code>	Gibt bei jedem Laden einer Klasse eine Meldung auf der Console aus.
<code>-verbosegc</code>	Veranlaßt den Garbage Collector bei jedem Aufruf zur Ausgabe einer Nachricht.
<code>-DpropertyName=value</code>	Weist dem Property <code>propertyName</code> den Wert <code>value</code> zu.
<code>-Xms n</code>	Spezifiziert die Größe des beim Start allozierten Speichers. <i>n</i> ist dabei eine Ganzzahl mit einer der Erweiterungen »k« oder »m«. Die Buchstaben stehen für die Größenordnungen <i>kilo</i> und <i>mega</i> . Die Standardeinstellung ist <i>1m</i> .
<code>-Xmx n</code>	Spezifiziert die Größe des maximal allozierbaren Speichers. <i>n</i> ist dabei eine Ganzzahl mit einer der Erweiterungen »k« oder »m«. Die Standardeinstellung ist <i>16m</i> .

Tabelle 26.2: Optionen des Java-Interpreters

In den 1.1er-Versionen des JDK gab es ein Programm `jre`, das dazu diente, den Interpreter des Laufzeitsystems zu starten (*Java Runtime Environment*). Dieses Programm ist in der aktuellen JDK-Version nicht mehr vorhanden, sondern wird durch das Programm `java` ersetzt. Auch der in früheren Versionen vorhandene debugging-fähige Interpreter `java_g` existiert seit dem JDK 1.2 nicht mehr.

JDK1.1/1.2

26.1.3 appletviewer - Der Appletviewer

Aufruf

```
appletviewer [ options ] { url }
```

Beschreibung

`appletviewer` ist ein Hilfsprogramm zum Ausführen von Applets. Er interpretiert dazu die als Argument übergebenen (und auf HTML-Dokumente zeigenden) URLs und öffnet für jedes darin gefundene `APPLET`-Tag ein Fenster, in dem das zugehörige Applet gestartet wird. Alle übrigen HTML-Befehle werden ignoriert. Details zur Struktur der HTML-Dateien und zur Ausführung von Applets finden sich in [Kapitel 25](#).

Optionen

Option	Bedeutung
<code>-debug</code>	Startet den Appletviewer unter Kontrolle des Debuggers und erlaubt so die Fehlersuche in Applets.

Tabelle 26.3: Optionen von appletviewer

26.1.4 jdb - Der Debugger

Aufruf

jdb [classfile]

Beschreibung

[jdb](#) ist der Debugger des JDK. Er bietet die Möglichkeit, Programme kontrolliert ablaufen zu lassen und dabei Breakpoints zu setzen, im Einzelschrittmodus den nächsten Befehl ausführen zu lassen oder den Inhalt von Variablen oder Objekten zu inspizieren. Man sollte allerdings nicht zu große Erwartungen an [jdb](#) stellen, denn das Programm ist ein Kommandozeilendebugger in schönster UNIX-Tradition. Mit den grafischen Debuggern der integrierten Java-Entwicklungsumgebungen hat er nur wenig gemeinsam. Leider ist er nicht nur umständlicher zu bedienen, sondern läßt (insbesondere in Prä-1.2-JDKs) auch einige wichtige Features moderner Debugger vermissen.

Vorbereitungen

Damit ein Programm debugt werden kann, sollte es mit der Option `-g` übersetzt werden. Dadurch werden symbolische Informationen in die Klassendatei geschrieben, die der Debugger zur Interpretation von lokalen Variablen benötigt. Beim Aufruf des Debuggers kann die zu untersuchende Klassendatei als Argument angegeben werden:

jdb classfile

Damit [jdb](#) überhaupt startet, muß ein laufender TCP/IP-Stack vorhanden sein. Dies ist für Anwender unangenehm, die nur eine Wählverbindung zu ihrem Provider haben, denn bei jedem Starten des Debuggers die Verbindung aufzubauen ist nicht nur umständlich, sondern auch kostspielig. Die übliche Empfehlung im Usenet lautet in diesem Fall, eine Datei [hosts](#) anzulegen und den folgenden Eintrag einzufügen:

Hinweis

127.0.0.1 localhost

Unter Windows 95 muß die Datei im Windows-Installationsverzeichnis (typischerweise `c:\windows`) angelegt werden, meist gibt es dort schon eine Kopiervorlage `hosts.sam`, die verwendet werden kann. In aktuellen JDKs kann auf das Anlegen der Datei unter Umständen verzichtet werden. Falls der Debugger nicht starten will, kann es sinnvoll sein, zusätzlich die *DNS-Konfiguration* zu deaktivieren. Dazu ist in der Systemsteuerung im Bereich »Netzwerk« das TCP/IP-Protokoll auszuwählen und nach Klick auf »Eigenschaften« im Registerblatt »DNS-Konfiguration« der Button »DNS deaktivieren« auszuwählen.

Bei dieser Anpassung ist allerdings Vorsicht geboten, denn Veränderungen an den Netzwerkeinstellungen können die Netzwerkanbindung des Rechners unbrauchbar machen. Es empfiehlt sich in jedem Fall, vor der Veränderung der Parameter die alte Konfiguration zu notieren, um sie nötigenfalls wieder restaurieren zu können. Veränderungen sollte sowieso nur derjenige vornehmen, der genau weiß, was er tut.

Warnung

Nachdem der Debugger gestartet wurde, meldet er sich mit seiner Kommandozeile und ist bereit, Befehle entgegenzunehmen. Die wichtigsten von ihnen werden in [Tabelle 26.4](#) vorgestellt und kurz erläutert.

Kommando	Bedeutung
<code>?</code>	Liefert eine Übersicht aller Kommandos. Alternativ kann auch das Kommando help verwendet werden.
<code>!!</code>	Wiederholt das letzte Kommando.
<code>load classname</code>	Lädt eine Klassendatei in den Debugger. Falls jdb bereits mit einer Klassendatei als Argument aufgerufen wurde, muß dieses Kommando nicht mehr aufgerufen werden.
<code>run</code>	Startet das geladene Programm. Falls die Klassendatei mit <code>load</code> geladen wurde, müssen zusätzlich der Klassenname und ggfs. weitere Parameter des Programms übergeben werden. Nach Ausführung dieses Programms läuft das Programm bis zum nächsten Breakpoint oder bis es mit dem <code>suspend</code> -Kommando angehalten wird.
<code>quit</code>	Beendet den Debugger. Alternativ kann auch das Kommando <code>exit</code> verwendet werden.
<code>stop in</code>	Setzt einen Breakpoint auf den Anfang einer Methode. Das Kommando erwartet den Namen der Klasse, einen Punkt und den Namen der Methode als Argument. Beispiel: <code>stop in JDBBeispiel.actionPerformed</code>

<code>stop at</code>	Setzt einen Breakpoint auf eine vorgegebene Zeile. Dazu müssen der Name der Klasse, ein Doppelpunkt und die Zeilennummer innerhalb der Quelldatei angegeben werden. Beispiel: <code>stop at JDBBeispiel:48</code>
<code>clear</code>	Löscht einen Breakpoint. Als Argument müssen der Klassenname, gefolgt von einem Doppelpunkt und der Zeilennummer, in der sich der Breakpoint befindet, angegeben werden. Es gibt keine Möglichkeit, eine Liste der aktuellen Breakpoints anzeigen zu lassen.
<code>list</code>	Zeigt den aktuellen Ausschnitt des Quelltextes an, an dem das Programm angehalten wurde. Die als nächstes auszuführende Zeile wird durch einen Pfeil markiert. Alternativ kann auch der Name einer Methode oder eine Zeilennummer an das Kommando übergeben werden.
<code>step</code>	Nachdem ein Programm angehalten wurde, kann es mit diesem Kommando im Einzelschrittmodus fortgeführt werden. Befindet sich an der Aufrufstelle ein Methodenaufruf, springt das Kommando in die Methode hinein und bleibt bei der ersten ausführbaren Anweisung stehen.
<code>next</code>	Wie <code>step</code> , springt dabei aber nicht in einen Methodenaufruf hinein, sondern führt die Methode als Ganzes aus und bleibt beim nächsten Kommando nach dem Methodenaufruf stehen. Dieses Kommando steht erst ab dem JDK 1.2 zur Verfügung.
<code>step up</code>	Führt die aktuelle Methode bis zum Ende aus.
<code>cont</code>	Führt das Programm nach einem Breakpoint fort. Es läuft dann bis zum nächsten Breakpoint oder bis es mit dem <code>suspend</code> -Kommando angehalten wird.
<code>print</code>	Zeigt den Inhalt der Variablen an, die als Argument übergeben wurde. <code>print</code> verwendet dazu die Methode <code>toString</code> , die in allen Objektvariablen implementiert ist. Damit eine Variable angezeigt werden kann, muß sie an der Aufrufstelle sichtbar sein. Es gibt leider keine Möglichkeit, den Inhalt einer Variablen aus dem Debugger heraus zu verändern.
<code>dump</code>	Zeigt den Inhalt der Objektvariable, die als Argument übergeben wurde, inklusive aller Membervariablen an. Damit eine Variable angezeigt werden kann, muß sie an der Aufrufstelle sichtbar sein.
<code>locals</code>	Gibt eine Liste aller lokalen Variablen und ihrer aktuellen Inhalte aus.
<code>where</code>	Zeigt einen Stacktrace an.

Tabelle 26.4: Kommandos von `jdb`

26.2 Werkzeuge zur Dokumentation und Archivierung

- [26.2 Werkzeuge zur Dokumentation und Archivierung](#)
 - [26.2.1 javadoc - Der Dokumentationsgenerator](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Dokumentationskommentare](#)
 - [Aufruf von javadoc](#)
 - [Optionen](#)
 - [26.2.2 jar - Das Archivierungswerkzeug](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Kommandos](#)
 - [Verwendung von jar-Dateien in Applets](#)
 - [26.2.3 javap - Der Disassembler](#)
 - [Aufruf](#)
 - [Beschreibung](#)
 - [Optionen](#)

26.2.1 javadoc - Der Dokumentationsgenerator

Aufruf

```
javadoc [ options ] { package | sourcefile }
```

Beschreibung

[javadoc](#) ist ein Programm, das aus Java-Quelltexten Dokumentationen im HTML-Format erstellt. Dazu verwendet es die öffentlichen Klassen-, Interface- und Methodendeklarationen und fügt zusätzliche Informationen aus eventuell vorhandenen Dokumentationskommentaren hinzu. Zu jeder Klassendatei `xyz.java` wird eine HTML-Seite `xyz.html` generiert, die über verschiedene Querverweise mit den anderen Seiten desselben Projekts in Verbindung steht. Zusätzlich generiert [javadoc](#) diverse Index- und Hilfsdateien, die das Navigieren in den Dokumentationsdateien erleichtern.

Dokumentationskommentare

Bereits ohne zusätzliche Informationen erstellt [javadoc](#) aus dem Quelltext eine brauchbare Beschreibung aller Klassen und Interfaces. Durch das Einfügen von Dokumentationskommentaren kann die Ausgabe zusätzlich bereichert werden. Ein Dokumentationskommentar beginnt mit `/**` und endet mit `*/` und ähnelt damit einem gewöhnlichen Kommentar. Er muß im Quelltext immer unmittelbar vor dem zu dokumentierenden Item platziert werden (einer Klassendefinition, einer Methode oder einer Instanzvariable). Er kann aus mehreren Zeilen bestehen. Die erste Zeile des Kommentars wird später als Kurzbeschreibung verwendet.

Zur Erhöhung der Übersichtlichkeit darf am Anfang jeder Zeile ein Sternchen stehen, es wird später ignoriert. Innerhalb der Dokumentationskommentare dürfen neben normalem Text auch HTML-Tags vorkommen. Sie werden unverändert in die Dokumentation übernommen und erlauben es damit, bereits im Quelltext die Formatierung der späteren Dokumentation vorzugeben. Die Tags `<h1>` und `<h2>` sollten möglichst nicht verwendet werden, da sie von [javadoc](#) selbst zur Strukturierung der Ausgabe verwendet werden.

Hinweis

[javadoc](#) erkennt des weiteren *markierte Absätze* innerhalb von Dokumentationskommentaren. Die Markierung muß mit dem Zeichen `@` beginnen und - abgesehen von Leerzeichen - am Anfang der Zeile stehen. Jede Markierung leitet einen eigenen Abschnitt innerhalb der Beschreibung ein, alle Markierungen eines Typs müssen hintereinander stehen. [Tabelle 26.5](#) gibt eine Übersicht der wichtigsten Markierungen und beschreibt, wie sie verwendet werden.

Markierung und Parameter	Dokumentation	Verwendung in
@author <i>name</i>	Erzeugt einen Autoreneintrag.	Klasse, Interface
@version <i>version</i>	Erzeugt einen Versionseintrag. Darf höchstens einmal je Klasse oder Interface verwendet werden.	Klasse, Interface
@since <i>jdk-version</i>	Beschreibt, seit wann das beschriebene Feature existiert.	Klasse, Interface
@see <i>reference</i>	Erzeugt einen Querverweis auf eine andere Klasse, Methode oder einen beliebigen anderen Teil der Dokumentation. Gültige Verweise sind: <ul style="list-style-type: none"> • @see java.util.Vector • @see Vector • @see Vector#addElement • @see Spez 	Klasse, Interface, Instanzvariable, Methode
@param <i>name</i> <i>description</i>	Parameterbeschreibung einer Methode.	Methode
@return <i>description</i>	Beschreibung des Rückgabewerts einer Methode.	Methode
@exception <i>classname</i> <i>description</i>	Beschreibung einer Ausnahme, die von dieser Methode ausgelöst wird.	Methode
@deprecated <i>description</i>	Markiert eine veraltete Methode, die zukünftig nicht mehr verwendet werden sollte.	Methode

Tabelle 26.5: Markierungen in Dokumentationskommentaren

Aufruf von javadoc

Um [javadoc](#) aufzurufen, sollte zunächst in das Verzeichnis gewechselt werden, in dem sich die zu dokumentierenden Quelldateien befinden. Anschließend kann durch Eingabe des folgenden Kommandos die Erzeugung der Dokumentationen für alle Quelldateien gestartet werden:

```
javadoc *.java
```

Das Programm erzeugt eine Reihe von HTML-Dateien, die die zu den jeweiligen Quellen korrespondierenden Dokumentationen enthalten. Zusätzlich werden eine Reihe von Hilfsdateien zur Darstellung und Indexierung der Dokumentationsdateien erstellt.

Alternativ zum Aufruf mit einer Reihe von Quelldateien kann [javadoc](#) auch mit Paketnamen als Argument aufgerufen werden. Wenn der Klassenpfad korrekt gesetzt ist, spielt es dann keine Rolle mehr, aus welchem Verzeichnis das Programm gestartet wird, denn die Klassendateien werden automatisch korrekt gefunden. Wenn nicht per Schalter **-d** etwas anderes angegeben wurde, erzeugt [javadoc](#) die Dokumentationsdateien im aktuellen Verzeichnis.

In den JDKs 1.0 und 1.1 erzeugt [javadoc](#) zwei unterschiedliche Arten von Standardverweisen. Bei der ersten Art werden Grafiken eingebunden, um Überschriften für die verschiedenen Dokumentationsabschnitte zu generieren. Damit diese im Browser korrekt angezeigt werden, muß ein Unterverzeichnis [images](#) angelegt und die erforderlichen Grafikdateien dorthin kopiert werden (beispielsweise aus `\jdk1.1.2\docs\api\images`). Ab dem JDK 1.2 werden dagegen keine Grafikdateien mehr benötigt.

JDK1.1/1.2

Die zweite Art von Verweisen ist die auf Klassen oder Methoden der Java-Klassenbibliothek (z.B. `java.lang.String`). Im JDK 1.1 geht [javadoc](#) davon aus, daß sich die Dokumentationsdateien zu allen externen Klassen und Methoden im selben Verzeichnis wie die zu erstellenden Dokumentationsdateien befinden. Damit also externe Verweise funktionieren, müßte man die HTML-Files der Originaldokumentation des JDK in sein eigenes Dokumentationsverzeichnis kopieren, was sicherlich nicht immer praktikabel ist.

Im JDK 1.2 wurde die Option **-link** eingeführt, mit der ein Pfad auf die Dokumentation der Standardklassen angegeben werden kann. Der Pfad muß als URL angegeben werden und das Verzeichnis beschreiben, in dem die Datei [package-list](#) der Dokumentationsdateien liegt. In der Dokumentation zu [javadoc](#) gibt SUN folgendes Beispiel an:

JDK1.1/1.2

```
javadoc -link http://java.sun.com/products/jdk/1.2/docs/api ...
```

Dadurch wird bei Standard-Klassennamen auf die auf dem SUN-Server liegende Originaldokumentation verwiesen. Soll dagegen auf eine lokal installierte Dokumentation verwiesen werden, kann auch ein **file**-URL angegeben werden. Liegt die Dokumentation des JDK beispielsweise im Verzeichnis `c:\jdk1.2\docs` (und somit die API-Dokumentation im Verzeichnis `c:\jdk1.2\docs\api`), kann [javadoc](#) wie folgt

aufgerufen werden:

```
javadoc -link file:///c:/jdk1.2/docs/api ...
```

Optionen

Option	Bedeutung
-classpath path	Gibt die Liste der Pfade zur Suche von Klassendateien an
-public	Nur Elemente des Typs <code>public</code> werden dokumentiert
-protected	Elemente des Typs <code>public</code> und <code>protected</code> werden dokumentiert (das ist die Voreinstellung)
-package	Elemente des Typs <code>package</code> , <code>public</code> und <code>protected</code> werden dokumentiert
-private	Alle Elemente werden dokumentiert
-version	Versionseintrag generieren
-author	Autoreneintrag generieren
-sourcepath path	Pfad mit den Quelldateien
-d directory	Verzeichnis, in dem die generierten Dokumentationsdateien abgelegt werden. Standardmäßig werden sie im aktuellen Verzeichnis angelegt.
-verbose	Ausgabe zusätzlicher Meldungen während der Dokumentationserstellung

Tabelle 26.6: Einige Optionen von javadoc

Neben den hier erwähnten Schaltern kennt `javadoc` (insbesondere im JDK 1.2) eine ganze Reihe zusätzlicher Optionen, mit denen die Codeerzeugung beeinflusst werden kann. Bemerkenswert ist dabei auf jeden Fall das Konzept der *Doclets*, mit denen das Verhalten von `javadoc` und das Aussehen der generierten Dokumentationsdateien weitgehend verändert werden kann. Doclets sind Zusatzmodule für `javadoc`, deren Aufgabe es ist, auf der Basis des Doclet-APIs und der geparsten Quelltexte die Ausgabedateien zu erzeugen. Ob der generierte Code dabei im HTML-, RTF- oder einem anderen Format erzeugt wird, spielt keine Rolle. Das mit dem JDK 1.2 ausgelieferte Standard-Doclet erzeugt die Dokumentationsdateien im HTML-Format.

JDK1.1/1.2

26.2.2 jar - Das Archivierungswerkzeug

Aufruf

```
jar [ commands ] archive { input-file }
```

Beschreibung

`jar` ist ein Archivierungswerkzeug, das Dateien und komplette Unterverzeichnisse komprimieren und in eine gemeinsame Archivdatei packen kann. Es verwendet ein Kompressionsformat, das den diversen `zip`-/`unzip`-Programmen ähnelt, und wird analog dem UNIX-Tool `tar` bedient. Ein Vorteil von `jar` ist seine Portabilität, die sowohl für das erzeugte Dateiformat als auch für das (in Java geschriebene) Programm selbst gilt.

Wichtigster Einsatzzweck von `jar` ist es, alle zu einem Applet gehörenden Dateien (`.class`-, Image-, Sound-Dateien usw.) in einer einzigen Datei zusammenzufassen. Dadurch müssen Web-Browser nicht für jede einzelne Datei, die in einem Applet benötigt wird, eine eigene GET-Transaktion absetzen, sondern können alle erforderlichen Files in einem Schritt laden. Die Ladezeit von Applets wird dadurch drastisch verringert, insbesondere, wenn viele kleine Dateien benötigt werden.

Kommandos

Im Gegensatz zu den übrigen Programmen, die in diesem Kapitel vorgestellt wurden, kennt `jar` keine Optionsparameter, sondern erwartet *Kommandos* an ihrer Stelle. Ein Kommando besteht aus einem Buchstaben, der ohne den Präfix `-` angegeben wird. Sollen mehrere Kommandos kombiniert werden, so werden die zugehörigen Buchstaben ohne Lücken direkt hintereinander geschrieben. Diese abweichende Syntax stammt von dem Kommando `tar`, das auf UNIX-Rechnern zur Archivierung von Dateien eingesetzt wird. [Tabelle 26.7](#) gibt eine Übersicht der verfügbaren Kommandos.

Kommando	Bedeutung
c	Erzeugt eine neue Archivdatei (<i>create</i>). Kann nicht zusammen mit t oder x verwendet werden.

t	Gibt das Inhaltsverzeichnis der Archivdatei aus (<i>table of contents</i>). Kann nicht zusammen mit c oder x verwendet werden.
x file	Extrahiert eine oder mehrere Dateien aus dem Archiv (<i>extract</i>). Kann nicht zusammen mit c oder t verwendet werden.
u	Fügt die angegebenen Dateien in die bestehende Archivdatei ein.
f	Gibt an, daß der nächste Parameter der Name der Archivdatei ist. Wird das Kommando f nicht angegeben, verwendet jar statt dessen die Standardein- und ausgabe.
v	Gibt zusätzliche Informationen aus (<i>verbose</i>). Kann zusätzlich zu einem der anderen Kommandos verwendet werden.
0	Die Dateien werden ohne Kompression gespeichert.

Tabelle 26.7: Kommandos von jar

Sollen beispielsweise alle `.java`-Dateien des aktuellen Verzeichnisses in ein Archiv mit der Bezeichnung `xyz.jar` gepackt werden, so kann dazu folgendes Kommando verwendet werden:

```
jar cf xyz.jar *.java
```

Das Inhaltsverzeichnis des Archivs kann folgendermaßen abgerufen werden:

```
jar tf xyz.jar
```

Etwas ausführlicher geht es mit:

```
jar tvf xyz.jar
```

Um die Datei `Test.java` aus dem Archiv zu extrahieren, kann das folgende Kommando verwendet werden (das natürlich auch ohne den Zusatz `v` funktioniert):

```
jar xvf xyz.jar Test.java
```

Verwendung von jar-Dateien in Applets

Die Verwendung von `jar`-Dateien in Applets erfolgt mit Hilfe des `ARCHIVE`-Parameters des `APPLET`-Tags (siehe [Kapitel 25](#)). Soll beispielsweise das »Hello, World«-Programm `HWApplet.java` aus [Listing 25.7](#) aus einem `jar`-Archiv `hello.jar` ausgeführt werden, so ist in den folgenden Schritten vorzugehen.

Zunächst werden die Dateien `HWApplet.class`, `hello.au` und `world.au` in ein `jar`-Archiv gepackt:

```
jar cvf hello.jar HWApplet.class hello.au world.au
```

Anschließend wird die HTML-Datei `HWApplet.html` zum Aufruf des Applets erstellt:

[HWApplet.html](#)

```
001 <html>
002 <head>
003 <title>HWApplet</title>
004 </head>
005 <body>
006 <h1>HWApplet</h1>
007 <applet
008     code=HWApplet.class
009     archive=hello.jar
010     width=300
011     height=200>
012 Hier steht das Applet HWApplet.class
013 </applet>
014 </body>
015 </html>
```

Listing 26.1: HTML mit ARCHIVE-Tag

Nun kann das Applet wie bisher gestartet werden, benötigt aber zum Laden aller Dateien nur noch eine einzige HTTP-Transaktion.

Leider unterstützen noch nicht alle Browser das [jar](#)-Format, so daß seine Verwendung zum heutigen Zeitpunkt überlegt sein will. Für die nahe Zukunft ist es aber ein wichtiger Schritt zur Verbesserung der Ladezeiten von Applets.

Hinweis

26.2.3 javap - Der Disassembler

Aufruf

```
javap [ options ] classname
```

Beschreibung

Der Disassembler [javap](#) liest den übersetzten Code einer Klasse und gibt Informationen darüber auf der Standardausgabe aus. Dabei können entweder nur Informationen über Variablen und Methoden oder der komplette Bytecode der Klasse ausgegeben werden. [javap](#) ist nicht in der Lage, den Java-Quellcode einer Klassendatei wieder herzustellen. Beim Aufruf ist der Name der Klasse ohne die Erweiterung [.class](#) anzugeben, also beispielsweise:

```
javap -c java.lang.String
```

Optionen

Option	Bedeutung
-classpath path	Gibt die Liste der Pfade zur Suche von Klassendateien an.
-public	Nur die Klassenelemente des Typs public werden angezeigt.
-protected	Nur die Klassenelemente des Typs public und protected werden angezeigt.
-package	Die Klassenelemente des Typs public , protected und die Elemene mit Paketsichtbarkeit werden angezeigt. Das ist die Voreinstellung.
-private	Alle Klassenelemente werden angezeigt.
-c	Disassemblieren des Codes.
-s	Ausgabe der Methodensignaturen.
-l	Ausgabe von Zeilennummern.

Tabelle 26.8: Optionen von javap

26.3 Zusammenfassung

- [26.3 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Aufruf des Java-Compilers [javac](#) zum Übersetzen von [.java](#)- in [.class](#)-Dateien
- Aufruf des Java-Interpreters [java](#) zum Ausführen von Java-Applikationen
- Die Bedeutung der [CLASSPATH](#)-Umgebungsvariable
- Aufruf des Applet-Viewers [appletviewer](#) zum Ausführen von Java-Applets
- Vorbereiten eines Programms zum Debuggen und Aufruf des Debuggers [jdb](#)
- Die wichtigsten Kommandos des Debuggers
- Erzeugen von Quelltextdokumentationen mit Hilfe von [javadoc](#)
- Die Struktur von Dokumentationskommentaren und die Markierungen [@author](#), [@version](#), [@since](#), [@see](#), [@param](#), [@return](#), [@exception](#) und [@deprecated](#)
- Die Bedienung des Archivierungswerkzeugs [jar](#)
- Verwendung von [jar](#)-Dateien in Applets
- Aufruf des Disassemblers [javap](#)

Kapitel 27

Collections

- [27 Collections](#)
 - [27.1 Grundlagen und Konzepte](#)
 - [27.2 Die Collection des Typs List](#)
 - [27.2.1 Abstrakte Eigenschaften](#)
 - [27.2.2 Implementierungen](#)
 - [27.3 Iteratoren](#)
 - [27.3.1 Das Interface Iterator](#)
 - [27.3.2 Das Interface ListIterator](#)
 - [27.4 Eine eigene Queue-Klasse](#)
 - [27.4.1 Anforderungen](#)
 - [27.4.2 Implementierung](#)
 - [27.5 Die Collection des Typs Set](#)
 - [27.5.1 Abstrakte Eigenschaften](#)
 - [27.5.2 Implementierungen](#)
 - [27.6 Die Collection des Typs Map](#)
 - [27.6.1 Abstrakte Eigenschaften](#)
 - [27.6.2 Implementierungen](#)
 - [27.7 Sortierte Collections](#)
 - [27.7.1 Comparable und Comparator](#)
 - [27.7.2 SortedSet und TreeSet](#)
 - [27.7.3 SortedMap und TreeMap](#)
 - [27.8 Die Klasse Collections](#)
 - [27.8.1 Sortieren und Suchen](#)
 - [27.8.2 Synchronisieren von Collections](#)
 - [27.8.3 Erzeugen unveränderlicher Collections](#)
 - [27.9 Zusammenfassung](#)

27.1 Grundlagen und Konzepte

- 27.1 Grundlagen und Konzepte

Als *Collections* bezeichnet man Datenstrukturen, die dazu dienen, *Mengen von Daten* aufzunehmen und zu verarbeiten. Die Daten werden in einer geeigneten Form abgelegt, und der Zugriff ist nur mit Hilfe vorgegebener Funktionen erlaubt. Typische Collections sind Stacks, Queues, Deques, Priority Queues, Listen oder Trees. Collections gibt es in großer Anzahl und diversen Implementierungsvarianten; jeder angehende Informatiker kann ein Lied davon singen.

In Java existieren seit der Version 1.0 die Collections [Vector](#), [Stack](#), [Hashtable](#) und [BitSet](#). Obwohl sie prinzipiell durchaus ihren Zweck erfüllen, gibt es bereits seit geraumer Zeit Kritik am Collections-Konzept des JDK. Zunächst wird die geringe Vielseitigkeit kritisiert, etwa im Vergleich zum mächtigen Collection-Konzept der Sprache SmallTalk. Zudem gelten die JDK-1.0-Collections als nicht gerade performant. Fast alle wichtigen Methoden sind [synchronized](#), und als generische Speicher von Elementen des Typs [Object](#) ist bei jedem Zugriff ein Typecast bzw. eine Typüberprüfung notwendig. Zudem gibt es einige Detailschwächen, wie zum Beispiel bei der Benennung der Methodennamen des [Enumeration](#)-Interfaces oder bei der Implementierung der Klasse [Stack](#), die aus [Vector](#) abgeleitet wurde und damit weit über ihr eigentliches Ziel hinausschießt.

Diese Kritik wurde von den Java-Designern zum Anlaß genommen, das Collection-Konzept im Rahmen der Version 1.2 neu zu überdenken. Herausgekommen ist dabei eine Sammlung von gut 20 Klassen und Interfaces im Paket [java.util](#), die das Collections-Framework des JDK 1.2 bilden. Wer bei dieser Zahl erschreckt, sei getröstet. Letztlich werden im Wesentlichen die drei Grundformen [Set](#), [List](#) und [Map](#) realisiert. Die große Anzahl ergibt sich aus der Bereitstellung verschiedener Implementierungsvarianten, Interfaces und einiger abstrakter Basisklassen, mit denen die Implementierung eigener Collections vereinfacht werden kann. Wir wollen uns zunächst mit der grundlegenden Arbeitsweise der Collection-Typen vertraut machen:

- Eine [List](#) ist eine beliebig große Liste von Elementen beliebigen Typs, auf die sowohl wahlfrei als auch sequentiell zugegriffen werden kann.
- Ein [Set](#) ist eine (doublettenlose) Menge von Elementen, auf die mit typischen Mengenoperationen zugegriffen werden kann.
- Eine [Map](#) ist ein Abbildung von Elementen eines Typs auf Elemente eines anderen Typs, also eine Menge zusammengehöriger Paare von Objekten.

Jede dieser Grundformen ist als Interface unter dem oben angegebenen Namen implementiert. Zudem gibt es jeweils eine oder mehrere konkrete Implementierungen. Sie unterscheiden sich in den verwendeten Datenstrukturen und Algorithmen und damit in ihrer Eignung für verschiedene Anwendungsfälle. Weiterhin gibt es eine abstrakte Implementierung des Interfaces, mit dessen Hilfe das Erstellen eigener Collections erleichtert wird.

Im Gegensatz zu den 1.1-Klassen sind die Collections des JDK 1.2 aus Performancegründen durchgängig unsynchronisiert. Soll also von mehr als einem Thread gleichzeitig auf eine Collection zugegriffen werden (Collections sind häufig Kommunikationsmittel zwischen gekoppelten Threads), so ist unbedingt darauf zu achten, die Zugriffe selbst zu synchronisieren. Andernfalls können leicht Programmfehler und Dateninkonsistenzen entstehen.

JDK1.1/1.2

Die Namensgebung der Interfaces und Klassen folgt einem einfachen Schema. Das Interface hat immer den allgemein verwendeten Namen der zu implementierenden Collection, also beispielsweise [List](#), [Set](#) oder [Map](#). Jede Implementierungsvariante stellt vor den Namen dann eine spezifische Bezeichnung, die einen Hinweis auf die Art der verwendeten Datenstrukturen und Algorithmen geben soll. So gibt es für das Interface [List](#) beispielsweise die Implementierungsvarianten [LinkedList](#) und [ArrayList](#) sowie die abstrakte Implementierung [AbstractList](#). Wenn man dieses Schema einmal begriffen hat, verliert der »Collection-Zoo« viel von seinem Schrecken.

Die Interfaces spielen eine wichtige Rolle, denn sie beschreiben bereits recht detailliert die Eigenschaften der folgenden Implementierungen. Dabei wurde im JDK 1.2 eine weitreichende Design-Entscheidung getroffen. Um nicht für jede denkbare Collection-Klasse ein eigenes Interface definieren zu müssen, wurde ein Basisinterface [Collection](#) geschaffen, aus dem die meisten Interfaces abgeleitet wurden. Es faßt die wesentlichen Eigenschaften einer großen Menge unterschiedlicher Collections zusammen:

```

int size()
boolean isEmpty()
boolean contains(Object o)
Iterator iterator()
Object[] toArray()
Object[] toArray(Object a[])
boolean add(Object o)
boolean remove(Object o)
boolean containsAll(Collection c)
boolean addAll(Collection c)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
void clear()
boolean equals(Object o)
int hashCode()

```

Zusätzlich fordert die JDK-1.2-Spezifikation für jede Collection-Klasse zwei Konstruktoren (was ja leider nicht im Rahmen der Interface-Definition sichergestellt werden kann). Ein parameterloser Konstruktor dient dazu, eine leere Collection anzulegen. Ein weiterer, der ein einzelnes [Collection](#)-Argument besitzt, dient dazu, eine neue Collection anzulegen und mit allen Elementen aus der als Argument übergebenen Collection zu füllen. Die Interfaces [List](#) und [Set](#) sind direkt aus [Collection](#) abgeleitet, [SortedSet](#) entstammt ihr (als Nachfolger von [Set](#)) mittelbar. Lediglich die Interfaces [Map](#) und das daraus abgeleitete Interface [SortedMap](#) wurden nicht aus [Collection](#) abgeleitet.

Der Vorteil dieses Designs ist natürlich, daß eine flexible Schnittstelle für den Umgang mit *Mengen von Objekten* zur Verfügung steht. Wenn eine Methode einen Rückgabewert vom Typ [Collection](#) besitzt, können die Aufrufer dieser Methode auf die zurückgegebenen Elemente - unabhängig vom Typ der tatsächlich zurückgegebenen Collection-Klasse - einheitlich zugreifen. Selbst wenn eine andere *Implementierung* gewählt wird, ändert sich für den Aufrufer nichts, solange das zurückgegebene Objekt das [Collection](#)-Interface implementiert.

Soweit zur Theorie. Der designerische Nachteil dieses Ansatzes besteht darin, daß längst nicht alle tatsächlichen Collections eine Schnittstelle besitzen, wie sie in [Collection](#) definiert wurde. Während diese für eine Liste noch passen mag, besitzt eine Queue oder ein Stack gänzlich anders arbeitende Zugriffsroutinen. Dieser Konflikt wurde dadurch zu lösen versucht, daß die Methoden *zum Ändern* der Collection *optional* sind. Während also [contains](#), [containsAll](#), [equals](#), [hashCode](#), [isEmpty](#), [size](#) und [toArray](#) obligatorisch sind, müssen die übrigen Methoden in einer konkreten Implementierung nicht unbedingt zur Verfügung gestellt werden, sondern können weggelassen, ersetzt oder ergänzt werden.

Aus [Abschnitt 7.7](#) wissen wir allerdings, daß *alle* definierten Methoden eines Interfaces in einer konkreten Implementierung zur Verfügung gestellt werden müssen. Davon sind natürlich auch die Collection-Klassen nicht ausgenommen. Wenn sich diese entschließen, eine optionale Methode nicht zu realisieren, so muß diese zwar implementiert werden, löst aber beim Aufruf eine Exception des Typs [UnsupportedOperationException](#) aus. Die tatsächliche Schnittstelle einer konkreten Collection-Klasse kann also nicht dem Interface entnommen werden, das sie implementiert, sondern erfordert zusätzlich die schriftliche Dokumentation der Klasse.

Über diesen Designansatz kann man sich streiten. Da sowieso nur die nicht-verändernden Methoden nicht-optional sind, hätte man ebenso gut diese allein in ein generisches Collection-Interface stecken können und von allen Collection-Klassen implementieren lassen können. Zusätzliche Methoden wären dann in abgeleiteten Interfaces passend zur jeweiligen Collection-Klasse zu definieren.

27.2 Die Collection des Typs List

- [27.2 Die Collection des Typs List](#)
 - [27.2.1 Abstrakte Eigenschaften](#)
 - [27.2.2 Implementierungen](#)

27.2.1 Abstrakte Eigenschaften

Eine Collection vom Typ [List](#) ist eine geordnete Menge von Objekten, auf die entweder sequentiell oder über ihren Index (ihre Position in der Liste) zugegriffen werden kann. Wie bei Arrays hat das erste Element den Index 0 und das letzte den Index `size() - 1`. Es ist möglich, an einer beliebigen Stelle der Liste ein Element einzufügen oder zu löschen. Die weiter hinten stehenden Elemente werden dann entsprechend nach rechts bzw. links verschoben. Des weiteren gibt es Methoden, um Elemente in der Liste zu suchen.

Das Interface [List](#) ist direkt aus [Collection](#) abgeleitet und erbt somit dessen Methoden. Zusätzlich gibt es einige neue Methoden, die zum wahlfreien Zugriff auf die Elemente benötigt werden. Um Elemente in die Liste einzufügen, können die Methoden [add](#) und [addAll](#) verwendet werden:

```

java.util.List
void add(int index, Object element)
boolean add(Object o)

boolean addAll(Collection c)
boolean addAll(int index, Collection c)
    
```

Mit [add](#) wird ein einfaches Element in die Liste eingefügt. Wenn die Methode mit einem einzelnen [Object](#) als Parameter aufgerufen wird, hängt sie das Element an das Ende der Liste an. Wird zusätzlich der Index angegeben, so wird das Element an der spezifizierten Position eingefügt und alle übrigen Elemente um eine Position nach rechts geschoben. Mit [addAll](#) kann eine komplette Collection in die Liste eingefügt werden. Auch hier können die Elemente wahlweise an das Ende angehängt oder an einer beliebigen Stelle in der Liste eingefügt werden.

Der Rückgabewert von [add](#) ist [true](#), wenn die Liste durch den Aufruf von [add](#) verändert, also das Element hinzugefügt wurde. Er ist [false](#), wenn die Liste nicht verändert wurde. Das kann beispielsweise dann der Fall sein, wenn die Liste keine Doubletten erlaubt und ein bereits vorhandenes Element noch einmal eingefügt werden soll. Konnte das Element dagegen aus einem anderen Grund nicht eingefügt werden, wird eine Ausnahme des Typs [UnsupportedOperationException](#), [ClassCastException](#) oder [IllegalArgumentException](#) ausgelöst.

Hinweis

Das Löschen von Elementen kann mit den Methoden [remove](#), [removeAll](#) und [retainAll](#) erfolgen:

```

java.util.List
Object remove(int index)
boolean remove(Object o)

boolean removeAll(Collection c)
boolean retainAll(Collection c)
    
```

An [remove](#) kann dabei wahlweise der Index des zu löschenden Objekts oder das Objekt selbst übergeben werden. Mit [removeAll](#) werden alle Elemente gelöscht, die auch in der als Argument übergebenen Collection enthalten sind, und [retainAll](#) löscht alle Elemente außer den in der Argument-Collection enthaltenen.

27.2.2 Implementierungen

Das Interface [List](#) wird im JDK 1.2 von verschiedenen Klassen implementiert:

- Die Klasse [AbstractList](#) ist eine abstrakte Basisklasse, bei der alle optionalen Methoden die Ausnahme [UnsupportedOperationException](#) auslösen und diverse obligatorische Methoden als [abstract](#) deklariert wurden. Sie dient als Basisklasse für eigene [List](#)-Implementierungen.
- Die Klasse [LinkedList](#) realisiert eine Liste, deren Elemente als doppelt verkettete lineare Liste gehalten werden. Ihre Einfüge- und Löschooperationen sind im Prinzip (viele Elemente vorausgesetzt) performanter als die der [ArrayList](#). Der wahlfreie Zugriff ist dagegen normalerweise langsamer.

- Die Klasse [ArrayList](#) implementiert die Liste als Array von Elementen, das bei Bedarf vergrößert wird. Hier ist der wahlfreie Zugriff schneller, aber bei großen Elementzahlen kann das Einfügen und Löschen länger dauern als bei einer [LinkedList](#).
- Aus Gründen der Vereinheitlichung implementiert im JDK 1.2 auch die Klasse [Vector](#) das [List](#)-Interface. Neben den bereits in [Abschnitt 12.1](#) erwähnten Methoden besitzt ein 1.2-[Vector](#) also auch die entsprechenden Methoden des [List](#)-Interfaces.

Soll im eigenen Programm eine Liste verwendet werden, stellt sich die Frage, welche der genannten Implementierungen dafür am besten geeignet ist. Während die Klasse [AbstractList](#) nur als Basisklasse eigener Listenklassen sinnvoll verwendet werden kann, ist die Entscheidung für eine der drei übrigen Klassen von den Spezifika der jeweiligen Anwendung abhängig. Bleibt die Liste klein, wird hauptsächlich wahlfrei darauf zugegriffen; überwiegen die lesenden die schreibenden Zugriffe deutlich, so liefert die [ArrayList](#) die besten Ergebnisse. Ist die Liste dagegen sehr groß und werden häufig Einfügungen und Löschungen vorgenommen, ist wahrscheinlich die [LinkedList](#) die bessere Wahl. Wird von mehreren Threads gleichzeitig auf die Liste zugegriffen, kann die Klasse [Vector](#) verwendet werden, denn ihre Methoden sind bereits weitgehend als [synchronized](#) deklariert. Weitere Untersuchungen zur Performance der Listentypen sind in [Abschnitt 29.2.3](#) zu finden.

Das folgende Beispiel zeigt das Anlegen und Bearbeiten zweier unterschiedlicher Listen:

Beispiel

[Listing2701.java](#)

```
001 /* Listing2701.java */
002
003 import java.util.*;
004
005 public class Listing2701
006 {
007     static void fillList(List list)
008     {
009         for (int i = 0; i < 10; ++i) {
010             list.add("" + i);
011         }
012         list.remove(3);
013         list.remove("5");
014     }
015
016     static void printList(List list)
017     {
018         for (int i = 0; i < list.size(); ++i) {
019             System.out.println((String)list.get(i));
020         }
021         System.out.println("---");
022     }
023
024     public static void main(String args[])
025     {
026         //Erzeugen der LinkedList
027         LinkedList list1 = new LinkedList();
028         fillList(list1);
029         printList(list1);
030         //Erzeugen der ArrayList
031         ArrayList list2 = new ArrayList();
032         fillList(list2);
033         printList(list2);
034         //Test von removeAll
035         list2.remove("0");
036         list1.removeAll(list2);
037         printList(list1);
038     }
039 }
```

Listing 27.1: Anlegen und Bearbeiten zweier Listen

Hierbei wird zunächst je eine Liste des Typs [LinkedList](#) und [ArrayList](#) angelegt; beide werden durch Aufruf von `fillList` identisch gefüllt. Der Parameter `list` hat den Typ [List](#) und akzeptiert damit beliebige Objekte, die dieses Interface implementieren. Tatsächlich spielt es für `fillList` keine Rolle, welche konkrete Listenklasse verwendet wurde, und die nachfolgenden Ausgabemethoden geben jeweils genau dasselbe aus:

```
0
1
2
```

4
6
7
8
9

0
1
2
4
6
7
8
9

0

Anschließend wird aus `list2` das Element "0" entfernt und dann aus `list1` alle Elemente gelöscht, die noch in `list2` enthalten sind. Nach dem Aufruf von `removeAll` verbleibt also nur noch das (zu diesem Zeitpunkt nicht mehr in `list2` enthaltene) Element "0" in `list1` und wird durch den folgenden Aufruf von `printList` ausgegeben.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

27.3 Iteratoren

- 27.3 Iteratoren
 - 27.3.1 Das Interface Iterator
 - 27.3.2 Das Interface ListIterator

27.3.1 Das Interface Iterator

Auf die Collections der Prä-1.2-JDKs konnte mit Hilfe des [Enumeration](#)-Interfaces und seinen beiden Methoden [hasMoreElements](#) und [nextElement](#) zugegriffen werden. Da Objekte, die dem Durchlaufen von Collections dienen, überall in der Informatik als *Iteratoren* bezeichnet werden, wurde die Namensgebung des Interfaces und seiner Methoden vielfach kritisiert. Die Designer der Collections der Version 1.2 haben sich nun dem allgemeinen Sprachgebrauch angepaßt und das Interface zum Durchlaufen der Elemente einer Collection als [Iterator](#) bezeichnet und mit folgenden Methoden ausgestattet:

```
boolean hasNext()  
  
Object next()  
  
void remove()
```

[hasNext](#) gibt genau dann [true](#) zurück, wenn der Iterator mindestens ein weiteres Element enthält. [next](#) liefert das nächste Element bzw. löst eine Ausnahme des Typs [NoSuchElementException](#) aus, wenn es keine weiteren Elemente gibt. Wie beim [Enumeration](#)-Interface ist der Rückgabewert als [Object](#) deklariert und muß daher auf das passende Object gecastet werden. Als neues Feature (gegenüber einer [Enumeration](#)) bietet ein [Iterator](#) die Möglichkeit, die Collection während der Abfrage zu ändern, indem das zuletzt geholt Element mit der Methode [remove](#) gelöscht wird. Bei allen Collections, die das Interface [Collection](#) implementieren, kann ein Iterator zum Durchlaufen aller Elemente mit der Methode [iterator](#) beschafft werden.

Dies ist auch gleichzeitig die einzige erlaubte Möglichkeit, die Collection während der Verwendung eines Iterators zu ändern. Alle direkt ändernden Zugriffe auf die Collection machen das weitere Verhalten des Iterators undefiniert.

Wir wollen uns die Benutzung eines Iterators an einem Beispiel ansehen:

Listing2702.java

```
001 /* Listing2702.java */  
002  
003 import java.util.*;  
004  
005 public class Listing2702  
006 {  
007     public static void main(String args[])  
008     {  
009         //Füllen der Liste  
010         ArrayList list = new ArrayList();  
011         for (int i = 1; i <= 20; ++i) {  
012             list.add("" + i);  
013         }  
014         //Löschen von Elementen über Iterator  
015         Iterator it = list.iterator();  
016         while (it.hasNext()) {  
017             String s = (String) it.next();  
018             if (s.startsWith("1")) {  
019                 it.remove();  
020             }  
021         }  
022         //Ausgeben der verbleibenden Elemente  
023         it = list.iterator();  
024         while (it.hasNext()) {  
025             System.out.println((String) it.next());  
026         }  
027     }  
028 }
```

Beispiel

```
027     }
028 }
```

Listing 27.2: Zugriff auf eine Collection mit einem Iterator

Das Programm erzeugt zunächst eine Liste mit den Elementen "1", "2", ..., "20". Anschließend wird durch Aufruf von [iterator](#) ein Iterator über alle Elemente der Liste beschafft. Mit seiner Hilfe werden alle Elemente der Liste durchlaufen und diejenigen, die mit "1" anfangen, gelöscht. Der zweite Durchlauf durch die Liste zeigt dann nur noch die übriggebliebenen Elemente an:

```
2
3
4
5
6
7
8
9
20
```

27.3.2 Das Interface ListIterator

Neben dem [Iterator](#)-Interface gibt es das daraus abgeleitete Interface [ListIterator](#). Es steht nur bei Collections des Typs [List](#) (und daraus abgeleiteten Klassen) zur Verfügung und bietet zusätzlich die Möglichkeit, die Liste in beiden Richtungen zu durchlaufen, auf den Index des nächsten oder vorigen Elements zuzugreifen, das aktuelle Element zu verändern und ein neues Element hinzuzufügen:

```
boolean hasPrevious()
Object previous()

int nextIndex()
int previousIndex()

void add(Object o)
void set(Object o)
```

Mit [hasPrevious](#) kann bestimmt werden, ob es *vor* der aktuellen Position ein weiteres Element gibt; der Zugriff darauf würde mit [previous](#) erfolgen. Die Methoden [nextIndex](#) und [previousIndex](#) liefern den Index des nächsten bzw. vorigen Elements des Iterators. Wird [previousIndex](#) am Anfang des Iterators aufgerufen, ist sein Rückgabewert -1. Wird [nextIndex](#) am Ende aufgerufen, liefert es [size\(\)](#) als Rückgabewert. Mit [add](#) kann ein neues Element an der Stelle in die Liste eingefügt werden, die unmittelbar vor dem nächsten Element des Iterators liegt. [set](#) erlaubt es, das durch den letzten Aufruf von [next](#) bzw. [previous](#) beschaffte Element zu ersetzen.

Ebenso wie beim Interface [Collection](#) sind die *ändernden* Methoden der Iteratoren optional. Falls ein Iterator eine dieser Methoden nicht zur Verfügung stellen will, löst er bei ihrem Aufruf eine Ausnahme des Typs [UnsupportedOperationException](#) aus. Das gilt für die Methoden [add](#), [set](#) und [remove](#).

Hinweis

27.4 Eine eigene Queue-Klasse

- [27.4 Eine eigene Queue-Klasse](#)
 - [27.4.1 Anforderungen](#)
 - [27.4.2 Implementierung](#)

27.4.1 Anforderungen

Nachdem wir gezeigt haben, wie *vordefinierte* Collections verwendet werden, wollen wir uns nun ansehen, wie man eigene Collections erstellen kann. Wir wollen dazu die Datenstruktur einer *Queue* implementieren, die folgende Eigenschaften hat:

- Zu Beginn ist die Queue leer.
- Durch Aufruf von `add` wird ein neues Element am Ende der Queue angefügt.
- Ein Aufruf von `retrieve` gibt das erste Element der Queue zurück und entfernt es aus der Queue. Besitzt die Queue keine Elemente, wird eine Ausnahme des Typs `NoSuchElementException` ausgelöst.
- Die Methode `size` liefert die Anzahl der Elemente in der Queue.
- Mit `clear` werden alle Elemente aus der Queue entfernt.
- Die Methode `iterator` liefert einen Iterator für alle Elemente der Queue. Beim Aufruf von `remove` wird eine Ausnahme des Typs `UnsupportedOperationException` ausgelöst.

Die Elemente sollen als einfach verkettete Liste von Elementen gehalten werden. Jedes Element wird in einer Instanz der lokalen Klasse `ElementWrapper` gehalten, die neben dem Element selbst auch den Zeiger auf sein Nachfolgeelement verwaltet. Um (ganz nebenbei) den Umgang mit Listenstrukturen mit Zeigern zu zeigen, wollen wir die Daten nicht einer Collection des Typs `LinkedList` halten, sondern die Zeigeroperationen selbst implementieren.

27.4.2 Implementierung

Zunächst definieren wir ein Interface `Queue`, in dem wir die abstrakten Eigenschaften unserer Queue-Klasse spezifizieren:

[Queue.java](#)

```

001  /* Queue.java */
002
003  import java.util.*;
004
005  /**
006   * Das Interface der Queue-Collection.
007   */
008  public interface Queue
009  {
010      /**
011       * Fügt das Element o am Ende der Queue an. Falls
012       * keine Ausnahme ausgelöst wurde, gibt die Methode
013       * true zurück.
014       */
015      public boolean add(Object o);
016
017      /**
018       * Liefert das erste Element der Queue und entfernt es
019       * daraus. Falls die Queue leer ist, wird eine Ausnahme
020       * des Typs NoSuchElementException ausgelöst.
021       */
022      public Object retrieve()
023      throws NoSuchElementException;
024
025      /**
026       * Liefert die Anzahl der Elemente der Queue.
027       */
028      public int size();
029  }
```

```

030  /**
031   * Entfernt alle Elemente aus der Queue.
032   */
033  public void clear();
034
035  /**
036   * Liefert einen Iterator über alle Elemente der Queue.
037   */
038  public Iterator iterator();
039  }

```

Listing 27.3: Das Interface Queue

Dieses Interface kann nun verwendet werden, um konkrete Ausprägungen einer Queue zu implementieren. Wir wollen die Queue als verkettete Liste realisieren und definieren dazu die Klasse `LinkedListQueue`:

[LinkedListQueue.java](#)

```

001  /* LinkedListQueue.java */
002
003  import java.util.*;
004
005  /**
006   * Die folgende Klasse realisiert eine Queue-Collection
007   * auf der Basis einer einfach verketteten linearen Liste.
008   * Die LinkedListQueue kann im Prinzip beliebig viele Elemente
009   * aufnehmen. Die Laufzeiten der Einfüge- und Löschmethoden
010   * sind O(1). Der von iterator() gelieferte Iterator
011   * besitzt KEINE Implementierung der Methode remove().
012   */
013  public class LinkedListQueue
014  implements Queue
015  {
016      protected ElementWrapper first;
017      protected ElementWrapper last;
018      protected int count;
019
020      public LinkedListQueue()
021      {
022          first = last = null;
023          count = 0;
024      }
025
026      public boolean add(Object o)
027      {
028          if (count == 0) {
029              //insert first element
030              first = new ElementWrapper();
031              last = first;
032              count = 1;
033          } else {
034              //insert element into non-empty queue
035              last.next = new ElementWrapper();
036              last = last.next;
037              ++count;
038          }
039          last.element = o;
040          last.next = null;
041          return true;
042      }
043
044      public Object retrieve()
045      throws NoSuchElementException
046      {
047          if (count <= 0) {
048              throw new NoSuchElementException();
049          }
050          ElementWrapper ret = first;

```

```

051     --count;
052     first = first.next;
053     if (first == null) {
054         last = null;
055         count = 0;
056     }
057     return ret.element;
058 }
059
060 public int size()
061 {
062     return count;
063 }
064
065 public void clear()
066 {
067     while (first != null) {
068         ElementWrapper tmp = first;
069         first = first.next;
070         tmp.next = null;
071     }
072     first = last = null;
073     count = 0;
074 }
075
076 public Iterator iterator()
077 {
078     return new Iterator()
079     {
080         ElementWrapper tmp = first;
081
082         public boolean hasNext()
083         {
084             return tmp != null;
085         }
086
087         public Object next()
088         {
089             if (tmp == null) {
090                 throw new NoSuchElementException();
091             }
092             Object ret = tmp.element;
093             tmp = tmp.next;
094             return ret;
095         }
096
097         public void remove()
098         {
099             throw new UnsupportedOperationException();
100         }
101     };
102 }
103
104 /**
105  * Lokale Wrapperklasse für die Queue-Elemente.
106  */
107 class ElementWrapper
108 {
109     public Object element;
110     public ElementWrapper next;
111 }
112 }

```

Listing 27.4: Die Klasse `LinkedList`

Die einzelnen Elemente der Queue werden in Objekten der Wrapperklasse `ElementWrapper` gehalten. Die beiden `ElementWrapper`-Zeiger

`first` und `next` zeigen auf das erste bzw. letzte Element der Liste und werden zum Einfügen und Löschen von Elementen gebraucht. Der Zähler `count` hält die Anzahl der Elemente der Queue und wurde eingeführt, um die Methode `size` performant implementieren zu können. Die Methoden `add` und `retrieve` führen die erforderlichen Zeigeroperationen aus, um Elemente einzufügen bzw. zu entfernen, sie sollen hier nicht näher erläutert werden. In `clear` werden alle Elemente durchlaufen und ihr jeweiliger Nachfolgezeiger explizit auf `null` gesetzt, um dem Garbage Collector die Arbeit zu erleichtern.

Interessanter ist die Implementierung von `iterator`. Hier wird eine anonyme lokale Klasse definiert und zurückgegeben, die das Interface `Iterator` implementiert. Sie besitzt eine Membervariable `tmp`, mit dem die Elemente der Queue nacheinander durchlaufen werden. Wichtig ist dabei, daß `tmp` auf den *Originaldaten* der Liste arbeitet; es ist nicht nötig, Elemente oder Zeiger zu kopieren (dieses Prinzip ist auch der Schlüssel zur Implementierung der Methode `remove`, die wir hier nicht realisiert haben). `hasNext` muß also lediglich prüfen, ob `tmp` der Leerzeiger ist, was sowohl bei leerer Queue als auch nach Ende des Durchlaufs der Fall ist. `next` führt diese Prüfung ebenfalls aus, liefert gegebenenfalls das im Wrapperobjekt enthaltene Element zurück und stellt `tmp` auf das nächste Element.

Die hier gezeigte Anwendung von anonymen lokalen Klassen ist durchaus üblich und wird meist auch von den JDK-Klassen zur Implementierung der Methode `iterator` verwendet. Prägen Sie sich diese Vorgehensweise ein, denn sie zeigt ein wichtiges und häufig angewendetes Designprinzip in Java. Weitere Hinweise zu anonymen und lokalen Klassen sind in [Abschnitt 18.2.2](#) zu finden.

Hinweis

Abschließend wollen wir uns noch ein Beispiel für die Anwendung unserer Queue ansehen:

Beispiel

[Listing2705.java](#)

```
001 /* Listing2705.java */
002
003 import java.util.*;
004
005 public class Listing2705
006 {
007     public static void main(String args[])
008     {
009         //Erzeugen der Queue
010         LinkedList q = new LinkedList();
011         for (int i = 1; i <= 20; ++i) {
012             if (i % 4 == 0) {
013                 System.out.println(
014                     "Lösche: " + (String)q.retrieve()
015                 );
016             } else {
017                 q.add("" + i);
018             }
019         }
020         //Ausgeben aller Elemente
021         Iterator it = q.iterator();
022         while (it.hasNext()) {
023             System.out.println((String)it.next());
024         }
025     }
026 }
```

Listing 27.5: Anwendung der Queue-Klasse

Das Programm füllt die Queue, indem eine Schleife von 1 bis 20 durchlaufen wird. Ist der Schleifenzähler durch vier teilbar, wird das vorderste Element der Queue entfernt, andernfalls wird der Zähler in einen `String` konvertiert und an das Ende der Queue angehängt. Nach Ende der Schleife enthält die Queue also die Strings "1", "2", ..., "20", sofern sie nicht durch vier teilbar sind, und abzüglich der ersten fünf Elemente, die durch den Aufruf von `retrieve` entfernt wurden. Die Ausgabe des Programms ist demnach:

```
Lösche: 1
Lösche: 2
Lösche: 3
Lösche: 5
Lösche: 6
7
9
10
11
13
```

14
15
17
18
19

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	≤	≤	≥	≥
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	-------------------	-------------------	-------------------	-------------------

27.5 Die Collection des Typs Set

- [27.5 Die Collection des Typs Set](#)
 - [27.5.1 Abstrakte Eigenschaften](#)
 - [27.5.2 Implementierungen](#)

27.5.1 Abstrakte Eigenschaften

Ein [Set](#) ähnelt der [List](#), erlaubt aber im Gegensatz zu dieser keine doppelten Elemente. Genauer gesagt, in einem [Set](#) darf es zu keinem Zeitpunkt zwei Elemente `x` und `y` geben, für die `x.equals(y)` wahr ist. Ein [Set](#) entspricht damit im mathematischen Sinn einer *Menge*, in der ja ebenfalls jedes Element nur einmal vorkommen kann. Ein [Set](#) hat allerdings keine spezielle Schnittstelle, sondern erbt seine Methoden von der Basisklasse [Collection](#).

Die Unterschiede zu [List](#) treten zutage, wenn man versucht, mit `add` ein Element einzufügen, das bereits vorhanden ist (bei dem also die `equals`-Methode wie oben beschrieben `true` ergibt). In diesem Fall wird es nicht erneut eingefügt, sondern `add` gibt `false` zurück. Auch für die Methoden `addAll` und den Konstruktor `Set(Collection c)` gilt, daß sie kein Element doppelt einfügen und damit insgesamt die Integritätsbedingung einer Menge erhalten.

Ein weiterer Unterschied zu [List](#) ist, daß ein [Set](#) keinen [ListIterator](#), sondern lediglich einen einfachen [Iterator](#) erzeugen kann. Dessen Elemente haben keine definierte Reihenfolge. Sie kann sich durch wiederholtes Einfügen von Elementen im Laufe der Zeit sogar ändern.

Besondere Vorsicht ist geboten, wenn ein [Set](#) dazu benutzt wird, *veränderliche Objekte* (*mutable objects*) zu speichern. Wird ein Objekt, das in einem [Set](#) gespeichert ist, so verändert, daß der `equals`-Vergleich mit einem anderen Element danach einen anderen Wert ergeben könnte, so gilt der komplette [Set](#) als undefiniert und darf nicht mehr benutzt werden.

Warnung

Zentrale Ursache für dieses Problem ist die Tatsache, daß Objektvariablen intern als *Zeiger* auf die zugehörigen Objekte dargestellt werden. Auf ein Objekt, das in einem [Set](#) enthalten ist, kann somit auch von außen zugegriffen werden, wenn nach dem Einfügen des Objekts noch ein Verweis darauf zur Verfügung steht. Bei den in Java vordefinierten "kleinen" Klassen wie [String](#), [Integer](#) usw. ist das kein Problem, denn Objekte dieses Typs können nach ihrer Konstruktion nicht mehr verändert werden (*immutable objects*). Bei veränderlichen Objekten könnten dagegen im [Set](#) enthaltene Elemente unbemerkt verändert werden und dessen Integrität verletzt werden.

27.5.2 Implementierungen

Das [Set](#)-Interface wird im JDK nur von den Klassen [HashSet](#) und [AbstractSet](#) implementiert. Die abstrakte Implementierung dient lediglich als Basisklasse für eigene Ableitungen. In der voll funktionsfähigen [HashSet](#)-Implementierung werden die Elemente intern in einer [HashMap](#) gespeichert. Vor jedem Einfügen wird geprüft, ob das einzufügende Element bereits in der [HashMap](#) enthalten ist. Die Performance der Einfüge-, Löscho- und Zugriffsfunktionen ist im Mittel *konstant*. Neben den oben erwähnten Standardkonstruktoren besitzt die Klasse [HashSet](#) zwei weitere Konstruktoren, deren Argumente direkt an den Konstruktor der [HashMap](#) weitergereicht werden:

[java.util.HashSet](#)

```
HashSet(int initialCapacity)
HashSet(int initialCapacity, float loadFactor)
```


Wir wollen uns die Anwendung eines [HashSet](#) am Beispiel der Generierung eines Lottotips ansehen. Ein ähnliches Beispiel gab es bereits in [Listing 12.6](#), dort wurde ein [BitSet](#) verwendet, um doppelte Zahlen zu verhindern.

[Listing2706.java](#)

```
001 /* Listing2706.java */
002
003 import java.util.*;
004
005 public class Listing2706
006 {
007     public static void main(String args[])
008     {
009         HashSet set = new HashSet(10);
010         int doubletten = 0;
011         //Lottozahlen erzeugen
012         while (set.size() < 6) {
013             int num = (int)(Math.random() * 49) + 1;
014             if (!set.add(new Integer(num))) {
015                 ++doubletten;
016             }
017         }
018         //Lottozahlen ausgeben
019         Iterator it = set.iterator();
020         while (it.hasNext()) {
021             System.out.println(((Integer)it.next()).toString());
022         }
023         System.out.println("Ignorierte Doubletten: " + doubletten);
024     }
025 }
```

Listing 27.6: Generierung eines Lottotips mit HashSet

In diesem Listing wird ein [HashSet](#) so lange mit Zufallszahlen zwischen 1 und 49 bestückt, bis die Anzahl der Elemente 6 ist. Da in einen [Set](#) keine Elemente eingefügt werden können, die bereits darin enthalten sind, ist dafür gesorgt, daß jede Zahl nur einmal im Tip auftaucht. Der Zähler `doubletten` wird durch den Rückgabewert `false` von `add` getriggert. Er zählt, wie oft eine Zahl eingefügt werden sollte, die bereits enthalten war. Die Ausgabe des Programms könnte beispielsweise so aussehen:

```
29
17
16
35
3
30
Ignorierte Doubletten: 2
```

27.6 Die Collection des Typs Map

- [27.6 Die Collection des Typs Map](#)
 - [27.6.1 Abstrakte Eigenschaften](#)
 - [27.6.2 Implementierungen](#)

27.6.1 Abstrakte Eigenschaften

Eine Collection des Typs [Map](#) realisiert einen assoziativen Speicher, der Schlüssel auf Werte abbildet. Sowohl Schlüssel als auch Werte sind Objekte eines beliebigen Typs. Je Schlüssel gibt es entweder keinen oder genau einen Eintrag in der Collection. Soll ein Schlüssel-Wert-Paar eingefügt werden, dessen Schlüssel bereits existiert, wird dieses nicht neu eingefügt. Es wird lediglich dem vorhandenen Schlüssel der neue Wert zugeordnet. Der Wert wird also praktisch *ausgetauscht*.

Das Interface [Map](#) ist nicht von [Collection](#) abgeleitet, sondern eigenständig. Es definiert folgende Methoden:

```

int size()
boolean isEmpty()
boolean containsKey(Object key)
boolean containsValue(Object value)
Object get(Object key)
Object put(Object key, Object value)
Object remove(Object key)
void putAll(Map t)
void clear()
public Set keySet()
public Collection values()
public Set entrySet()
boolean equals(Object o)
int hashCode()

```

Die Methoden [size](#), [isEmpty](#), [remove](#), [clear](#), [equals](#) und [hashCode](#) sind mit den gleichnamigen Methoden des [Collection](#)-Interfaces identisch und brauchen daher nicht noch einmal erklärt zu werden.

Mit Hilfe von [put](#) wird ein neues Schlüssel-Wert-Paar eingefügt bzw. dem bereits vorhandenen Schlüssel ein neuer Wert zugeordnet. Die Methode [putAll](#) macht das für alle Paare der als Argument übergebenen [Map](#). Auch hier gilt die Regel, daß ein Schlüssel, der bereits vorhanden ist, nicht neu eingefügt wird. Lediglich sein zugeordneter Wert wird ausgetauscht. Mit der Methode [get](#) kann der Wert zu einem Schlüssel beschafft werden. Da der Rückgabewert vom Typ [Object](#) ist, muß er auf den erwarteten Wert gecastet werden.

Falls der angegebene Schlüssel nicht in der [Map](#) enthalten ist, wird [null](#) zurückgegeben. Hier

Hinweis

 ist Vorsicht geboten, denn ein [null](#)-Wert wird auch dann zurückgegeben, wenn die [Map](#) das Einfügen von [null](#)-Werten erlaubt und ein solcher diesem Schlüssel explizit zugeordnet wurde. Um diese beiden Fälle zu unterscheiden, kann die Methode [containsKey](#) verwendet werden. Sie gibt genau dann [true](#) zurück, wenn der gewünschte Schlüssel in der [Map](#) enthalten ist. Andernfalls liefert sie [false](#). Analog dazu kann mit [containsValue](#) festgestellt werden, ob ein bestimmter Wert mindestens einmal in der [Map](#) enthalten ist.

Im Vergleich zum [Collection](#)-Interface fällt auf, daß eine [Map](#) keine Methode [iterator](#) besitzt. Statt dessen kann sie drei unterschiedliche Collections erzeugen, die dann natürlich ihrerseits dazu verwendet werden können, einen Iterator zu liefern. Diese Collections werden als *Views*, also als *Sichten* auf die Collection bezeichnet:

- Die Methode [keySet](#) liefert die Menge der Schlüssel. Da per Definition keine doppelten Schlüssel in einer [Map](#) auftauchen, ist diese Collection vom Typ [Set](#).
- Die Methode [values](#) liefert die Menge der Werte der [Map](#). Da Werte sehr wohl doppelt enthalten sein können, ist der Rückgabewert lediglich vom Typ [Collection](#), wird also typischerweise eine Liste oder eine anonyme Klasse mit entsprechenden Eigenschaften sein.
- Die Methode [entrySet](#) liefert eine Menge von Schlüssel-Wert-Paaren. Jedes Element dieser Menge ist vom Typ [Map.Entry](#), d.h., es implementiert das lokale Interface [Entry](#) des Interfaces [Map](#). Dieses stellt u.a. die Methoden [getKey](#) und [getValue](#) zur Verfügung, um auf die beiden Komponenten des Paares zuzugreifen.

Neben den im Interface definierten Methoden sollte eine konkrete [Map](#)-Implementierung zwei Konstruktoren zur Verfügung stellen. Ein leerer Konstruktor dient dazu, eine leere [Map](#) zu erzeugen. Ein zweiter Konstruktor, der ein einzelnes [Map](#)-Argument erwartet, erzeugt eine neue [Map](#) mit denselben Schlüssel-Wert-Paaren wie die als Argument übergebene [Map](#).

27.6.2 Implementierungen

Das JDK stellt mehrere Implementierungen des [Map](#)-Interfaces zur Verfügung:

- Mit [AbstractMap](#) steht eine abstrakte Basisklasse für eigene Ableitungen zur Verfügung.
- Die Klasse [HashMap](#) implementiert das Interface auf der Basis einer *Hashtabelle*. Dabei wird ein Speicher fester Größe angelegt, und die Schlüssel werden mit Hilfe der *Hash-Funktion*, die den Speicherort direkt aus dem Schlüssel berechnet, möglichst gleichmäßig auf die verfügbaren Speicherplätze abgebildet. Da die Anzahl der potentiellen Schlüssel meist wesentlich größer als die Anzahl der verfügbaren Speicherplätze ist, können beim Einfügen Kollisionen auftreten, die mit geeigneten Mitteln behandelt werden müssen (bei der [HashMap](#) werden alle kollidierenden Elemente in einer verketteten Liste gehalten).
- Die altbekannte Klasse [Hashtable](#) implementiert seit dem JDK 1.2 ebenfalls das [Map](#)-Interface. Ihre Arbeitsweise entspricht im Prinzip der von [HashMap](#), allerdings mit dem Unterschied, daß ihre Methoden synchronisiert sind und daß es nicht erlaubt ist, [null](#)-Werte einzufügen (die [HashMap](#) läßt dies zu).

Weitere Details zur Klasse [Hashtable](#) finden sich in [Abschnitt 12.3](#). Dort gibt es auch ein kleines Anwendungsbeispiel, das wir hier in einer für die Klasse [HashMap](#) angepaßten Form zeigen wollen:

[Listing2707.java](#)

```
001 /* Listing2707.java */
002
003 import java.util.*;
004
005 public class Listing2707
006 {
007     public static void main(String[] args)
008     {
009         HashMap h = new HashMap();
010
011         //Pflege der Aliase
012         h.put("Fritz","f.mueller@test.de");
013         h.put("Franz","fk@b-blabla.com");
014         h.put("Paula","user0125@mail.uofm.edu");
015         h.put("Lissa","lb3@gateway.fhdto.northsurf.dk");
016
017         //Ausgabe
018         Iterator it = h.entrySet().iterator();
019         while (it.hasNext()) {
020             Map.Entry entry = (Map.Entry)it.next();
021             System.out.println(
022                 (String)entry.getKey() + " --> " +
023                 (String)entry.getValue()
024             );
025         }
026     }
027 }
```

Listing 27.7: Anwendung der Klasse *HashMap*

Der wichtigste Unterschied liegt in der Ausgabe der Ergebnisse. Während bei der [Hashtable](#) eine [Enumeration](#) verwendet wurde, müssen wir hier durch Aufruf von [entrySet](#) zunächst eine Menge der Schlüssel-Wert-Paare beschaffen. Diese liefert dann einen Iterator, der für jedes Element ein Objekt des Typs [Map.Entry](#) zurückgibt. Dessen Methoden [getKey](#) und [getValue](#) liefern den Schlüssel bzw. Wert des jeweiligen Eintrags. Alternativ hätten wir auch mit [keySet](#) die Menge der Schlüssel durchlaufen und mit [get](#) auf den jeweils aktuellen Wert zugreifen können. Dazu müßten die Zeilen [018](#) bis [025](#) im vorigen Listing durch folgenden Code ersetzt werden:

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

Die Ausgabe des Programms unterscheidet sich nur in der Sortierung von der von [Listing 12.3](#):

```
Franz --> fk@b-blabla.com
Fritz --> f.mueller@test.de
Paula --> user0125@mail.uofm.edu
Lissa --> lb3@gateway.fhdto.northsurf.dk
```

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

27.7 Sortierte Collections

- [27.7 Sortierte Collections](#)
 - [27.7.1 Comparable und Comparator](#)
 - [27.7.2 SortedSet und TreeSet](#)
 - [27.7.3 SortedMap und TreeMap](#)

27.7.1 Comparable und Comparator

Die bisher vorgestellten [Set](#)- und [Map](#)-Collections waren unsortiert, d.h., ihre Iteratoren haben die Elemente in keiner bestimmten Reihenfolge zurückgegeben. Im Gegensatz dazu gibt ein [List](#)-Iterator die Elemente in der Reihenfolge ihrer Indexnummern zurück. Im JDK gibt es nun auch die Möglichkeit, die Elemente eines [Set](#) oder einer [Map](#) zu sortieren. Dabei kann entweder die *natürliche Ordnung* der Elemente verwendet werden, oder die Elemente können mit Hilfe eines expliziten Vergleichsobjekts sortiert werden.

Bei der natürlichen Ordnung muß sichergestellt sein, daß alle Elemente der Collection eine [compareTo](#)-Methode besitzen und je zwei beliebige Elemente miteinander verglichen werden können, ohne daß es zu einem Typfehler kommt. Dazu müssen die Elemente das Interface [Comparable](#) aus dem Paket [java.lang](#) implementieren:

[java.lang.Comparable](#)

```
public int compareTo(Object o)
```

[Comparable](#) besitzt lediglich eine einzige Methode [compareTo](#), die aufgerufen wird, um das aktuelle Element mit einem anderen zu vergleichen.

- [compareTo](#) muß einen Wert kleiner 0 zurückgeben, wenn das aktuelle Element *vor* dem zu vergleichenden liegt.
- [compareTo](#) muß einen Wert größer 0 zurückgeben, wenn das aktuelle Element *hinter* dem zu vergleichenden liegt.
- [compareTo](#) muß 0 zurückgeben, wenn das aktuelle Element und das zu vergleichende *gleich* sind.

Während in älteren JDKs bereits einige Klassen sporadisch eine [compareTo](#)-Methode besaßen, wird seit dem JDK 1.2 das [Comparable](#)-Interface bereits von vielen der eingebauten Klassen implementiert, etwa von [String](#), [Character](#), [Double](#) usw. Die natürliche Ordnung einer Menge von Elementen ergibt sich nun, indem man alle Elemente paarweise miteinander vergleicht und dabei jeweils das kleinere vor das größere Element stellt.

JDK1.1/1.2

Die zweite Möglichkeit, eine Menge von Elementen zu sortieren, besteht darin, an den Konstruktor der Collection-Klasse ein Objekt des Typs [Comparator](#) zu übergeben. [Comparator](#) ist ein Interface, das lediglich eine einzige Methode [compare](#) definiert:

[java.util.Comparator](#)

```
public int compare(Object o1, Object o2)
```

Das übergebene [Comparator](#)-Objekt übernimmt die Aufgabe einer »Vergleichsfunktion«, deren Methode [compare](#) immer dann aufgerufen wird, wenn bestimmt werden muß, in welcher Reihenfolge zwei beliebige Elemente zueinander stehen.

27.7.2 SortedSet und TreeSet

Zur Realisierung von sortierten Mengen wurde aus [Set](#) das Interface [SortedSet](#) abgeleitet. Es erweitert das Basisinterface um einige interessante Methoden:

[java.util.SortedSet](#)

```
Object first()
Object last()
```

```
SortedSet headSet(Object toElement)
SortedSet subSet(Object fromElement, Object toElement)
SortedSet tailSet(Object fromElement)
```

Mit [first](#) und [last](#) kann das (gemäß der Sortierordnung) erste bzw. letzte Element der Menge beschafft werden. Die übrigen Methoden dienen dazu, aus der Originalmenge Teilmengen auf der Basis der Sortierung der Elemente zu konstruieren:

- [headSet](#) liefert alle Elemente, die echt kleiner als das als Argument übergebene Argument sind.
- [tailSet](#) liefert alle Elemente, die größer oder gleich dem als Argument übergebenen Element sind.

- [subSet](#) liefert alle Elemente, die größer oder gleich [fromElement](#) und kleiner als [toElement](#) sind.

Neben dem hier beschriebenen Interface fordert die Dokumentation zu [SortedSet](#) eine Reihe von Konstruktoren:

- Ein parameterloser Konstruktor erzeugt eine leere Menge, deren (zukünftige) Elemente bezüglich ihrer natürlichen Ordnung sortiert werden.
- Ein Konstruktor mit einem Argument des Typs [Comparator](#) erzeugt eine leere Menge, deren (zukünftige) Elemente bezüglich der durch den [Comparator](#) vorgegebenen Ordnung sortiert werden.
- Ein Konstruktor mit einem Argument vom Typ [Collection](#) erzeugt eine Menge, die alle eindeutigen Elemente der als Argument übergebenen Collection in ihrer natürlichen Ordnung enthält.
- Ein Konstruktor mit einem Argument des Typs [SortedSet](#) erzeugt eine Menge mit denselben Elementen und derselben Sortierung wie die als Argument übergebene Menge.

Die einzige Klasse im JDK 1.2, die das Interface [SortedSet](#) implementiert, ist [TreeSet](#). Sie implementiert die sortierte Menge mit Hilfe der Klasse [TreeMap](#). Diese verwendet einen *Red-Black-Tree* als Datenstruktur, also einen Baum, der durch eine imaginäre Rot-Schwarz-Färbung seiner Knoten und spezielle Einfüge- und Löschfunktionen davor geschützt wird, im Extremfall zu einer linearen Liste zu entarten. Alle Basisoperationen können in logarithmischer Zeit bezüglich der Anzahl der Elemente des Baums ausgeführt werden und sind damit auch bei großen Elementzahlen recht schnell.

Für uns interessanter ist die Fähigkeit, einen sortierten Iterator zu erzeugen. Wir wollen uns dazu ein Beispiel ansehen. Das folgende Programm erzeugt eine sortierte Menge und fügt einige Strings unsortiert ein. Anschließend werden die Strings mit Hilfe eines Iterators ausgegeben:

[Listing2708.java](#)

```
001 /* Listing2708.java */
002
003 import java.util.*;
004
005 public class Listing2708
006 {
007     public static void main(String[] args)
008     {
009         //Konstruieren des Sets
010         TreeSet s = new TreeSet();
011         s.add("Kiwi");
012         s.add("Kirsche");
013         s.add("Ananas");
014         s.add("Zitrone");
015         s.add("Grapefruit");
016         s.add("Banane");
017         //Sortierte Ausgabe
018         Iterator it = s.iterator();
019         while (it.hasNext()) {
020             System.out.println((String)it.next());
021         }
022     }
023 }
```

Listing 27.8: Die Klasse TreeSet

Die Ausgabe des Programms ist:

```
Ananas
Banane
Grapefruit
Kirsche
Kiwi
Zitrone
```

Der Iterator hat die Elemente also in alphabetischer Reihenfolge ausgegeben. Der Grund dafür ist, daß die Klasse [String](#) im JDK 1.2 das [Comparable](#)-Interface implementiert und eine Methode [compareTo](#) zur Verfügung stellt, mit der die Zeichenketten in lexikographischer Ordnung angeordnet werden. Sollen die Elemente unserer Menge dagegen rückwärts sortiert werden, ist die vorhandene [compareTo](#)-Methode dazu nicht geeignet. Statt dessen wird nun einfach ein [Comparator](#)-Objekt an den Konstruktor übergeben, dessen [compare](#)-Methode so implementiert wurde, daß zwei zu vergleichende Strings genau dann als aufsteigend beurteilt werden, wenn sie gemäß ihrer lexikographischen Ordnung absteigend sind. Das folgende Listing zeigt dies am Beispiel der Klasse [ReverseStringComparator](#):

```

001 /* Listing2709.java */
002
003 import java.util.*;
004
005 public class Listing2709
006 {
007     public static void main(String[] args)
008     {
009         //Konstruieren des Sets
010         TreeSet s = new TreeSet(new ReverseStringComparator());
011         s.add("Kiwi");
012         s.add("Kirsche");
013         s.add("Ananas");
014         s.add("Zitrone");
015         s.add("Grapefruit");
016         s.add("Banane");
017         //Rückwärts sortierte Ausgabe
018         Iterator it = s.iterator();
019         while (it.hasNext()) {
020             System.out.println((String)it.next());
021         }
022     }
023 }
024
025 class ReverseStringComparator
026 implements Comparator
027 {
028     public int compare(Object o1, Object o2)
029     {
030         return ((String)o2).compareTo((String)o1);
031     }
032 }

```

Listing 27.9: Rückwärts sortieren mit einem Comparator

Das Programm gibt nun die Elemente in umgekehrter Reihenfolge aus:

```

Zitrone
Kiwi
Kirsche
Grapefruit
Banane
Ananas

```

Mit Hilfe eines Comparators kann eine beliebige Sortierung der Elemente eines [SortedSet](#) erreicht werden. Wird ein [Comparator](#) an den Konstruktor übergeben, so wird die [compareTo](#)-Methode überhaupt nicht mehr verwendet, sondern die Sortierung erfolgt ausschließlich mit Hilfe der Methode [compare](#) des [Comparator](#)-Objekts. So können beispielsweise auch Elemente in einem [SortedSet](#) gespeichert werden, die das [Comparable](#)-Interface nicht implementieren.

Hinweis

27.7.3 SortedMap und TreeMap

Neben einem sortierten [Set](#) gibt es auch eine sortierte [Map](#). Das Interface [SortedMap](#) ist analog zu [SortedSet](#) aufgebaut und enthält folgende Methoden:

[java.util.SortedMap](#)

```

Object first()
Object last()

SortedMap headMap(Object toElement)
SortedMap subMap(Object fromElement, Object toElement)
SortedMap tailMap(Object fromElement)

```

Als konkrete Implementierung von [SortedMap](#) gibt es die Klasse [TreeMap](#), die analog zu [TreeSet](#) arbeitet. Die Methoden [keySet](#) und [entrySet](#) liefern Collections, deren Iteratoren ihre Elemente in aufsteigender Reihenfolge abliefern. Auch bei einer [SortedMap](#) kann wahlweise

mit der natürlichen Ordnung der Schlüssel gearbeitet werden oder durch Übergabe eines [Comparator](#)-Objekts an den Konstruktor eine andere Sortierfolge erzwungen werden.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

27.8 Die Klasse Collections

- [27.8 Die Klasse Collections](#)
 - [27.8.1 Sortieren und Suchen](#)
 - [27.8.2 Synchronisieren von Collections](#)
 - [27.8.3 Erzeugen unveränderlicher Collections](#)

Im Paket `java.util` gibt es eine Klasse `Collections` (man achte auf das »s« am Ende), die eine große Anzahl statischer Methoden zur Manipulation und Verarbeitung von Collections enthält. Darunter finden sich Methoden zum Durchsuchen, Sortieren, Kopieren und Synchronisieren von Collections sowie solche zur Extraktion von Elementen mit bestimmten Eigenschaften. Wir wollen uns hier nur einige der interessanten Methoden dieser Klasse ansehen und verweisen für weitere Informationen auf die JDK-Dokumentation.

27.8.1 Sortieren und Suchen

Die Klasse `Collections` enthält zwei Methoden `sort`:

[java.util.Collections](#)

```
static void sort(List list)
static void sort(List list, Comparator c)
```

Mit Hilfe von `sort` können beliebige Listen sortiert werden. Als Argument wird die Liste und wahlweise ein `Comparator` übergeben. Fehlt der `Comparator`, wird die Liste in ihrer natürlichen Ordnung sortiert. Dazu müssen alle Elemente das `Comparable`-Interface implementieren und ohne Typfehler paarweise miteinander vergleichbar sein. Gemäß JDK-Dokumentation verwendet diese Methode ein modifiziertes Mergesort, das auch im Worst-Case eine Laufzeit von $n \cdot \log(n)$ hat (auch bei der Klasse `LinkedList`) und damit auch für große Listen geeignet sein sollte.

Wir wollen als Beispiel noch einmal [Listing 27.8](#) aufgreifen und zeigen, wie man die unsortierten Elemente einer Liste mit Hilfe der Methode `sort` sortieren kann:

Beispiel

[Listing2710.java](#)

```
001 /* Listing2710.java */
002
003 import java.util.*;
004
005 public class Listing2710
006 {
007     public static void main(String[] args)
008     {
009         //Konstruieren des Sets
010         List l = new ArrayList();
011         l.add("Kiwi");
012         l.add("Kirsche");
013         l.add("Ananas");
014         l.add("Zitrone");
015         l.add("Grapefruit");
016         l.add("Banane");
017         //Unsortierte Ausgabe
018         Iterator it = l.iterator();
019         while (it.hasNext()) {
020             System.out.println((String)it.next());
021         }
022         System.out.println("---");
023         //Sortierte Ausgabe
024         Collections.sort(l);
025         it = l.iterator();
026         while (it.hasNext()) {
027             System.out.println((String)it.next());
028         }
029     }
030 }
```

Listing 27.10: Sortieren einer Liste

Die Ausgabe des Programms lautet:

```
Kiwi
Kirsche
Ananas
Zitrone
Grapefruit
Banane
---
Ananas
Banane
Grapefruit
Kirsche
Kiwi
Zitrone
```

Muß in einer großen Liste wiederholt gesucht werden, macht es Sinn, diese einmal zu sortieren und anschließend eine *binäre Suche* zu verwenden. Dabei wird das gewünschte Element durch eine Intervallschachtelung mit fortgesetzter Halbierung der Intervallgröße immer weiter eingegrenzt, und das gesuchte Element ist nach spätestens log(n) Schritten gefunden. Die binäre Suche wird mit Hilfe der Methoden [binarySearch](#) realisiert:

```
static int binarySearch(List list, Object key)
static int binarySearch(List list, Object key, Comparator c)
```

Auch hier gibt es wieder eine Variante, die gemäß der natürlichen Ordnung vorgeht, und eine zweite, die einen expliziten [Comparator](#) erfordert.

27.8.2 Synchronisieren von Collections

Wir haben bereits mehrfach erwähnt, daß die neuen Collections des JDK 1.2 nicht thread-sicher sind und wir aus Performancegründen auf den Gebrauch des Schlüsselworts [synchronized](#) weitgehend verzichtet haben. Damit in einer Umgebung, bei der von mehr als einem Thread auf eine Collection zugegriffen werden kann, nicht alle Manipulationen vom Aufrufer synchronisiert werden müssen, gibt es einige Methoden, die eine unsynchronisierte Collection in eine synchronisierte verwandeln:

```
static Collection synchronizedCollection(Collection c)
static List synchronizedList(List list)
static Map synchronizedMap(Map m)
static Set synchronizedSet(Set s)
static SortedMap synchronizedSortedMap(SortedMap m)
static SortedSet synchronizedSortedSet(SortedSet s)
```

Die Methoden erzeugen jeweils aus der als Argument übergebenen Collection eine synchronisierte Variante und geben diese an den Aufrufer zurück. Erreicht wird dies, indem eine neue Collection desselben Typs erzeugt wird, deren sämtliche Methoden synchronisiert sind. Wird eine ihrer Methoden aufgerufen, leitet sie den Aufruf innerhalb eines [synchronized](#)-Blocks einfach an die als Membervariable gehaltene Original-Collection weiter (dieses Designpattern bezeichnet man auch als *Delegate*).

27.8.3 Erzeugen unveränderlicher Collections

Analog zum Erzeugen von synchronisierten Collections gibt es einige Methoden, mit denen aus gewöhnlichen Collections *unveränderliche* Collections erzeugt werden können:

```
static Collection unmodifiableCollection(Collection c)
static List unmodifiableList(List list)
static Map unmodifiableMap(Map m)
static Set unmodifiableSet(Set s)
static SortedMap unmodifiableSortedMap(SortedMap m)
static SortedSet unmodifiableSortedSet(SortedSet s)
```

Auch hier wird jeweils eine Wrapper-Klasse erzeugt, deren Methodenaufrufe an die Original-Collection delegiert werden. Alle Methoden, mit denen die Collection verändert werden könnte, werden so implementiert, daß sie beim Aufruf eine Ausnahme des Typs [UnsupportedOperationException](#) auslösen.

27.9 Zusammenfassung

- [27.9 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Grundlagen der Collection-Klassen des JDK 1.2
- Das Interface [Collection](#)
- Das Interface [List](#) und die Implementierungen [LinkedList](#) und [ArrayList](#)
- Die Iteratoren [Iterator](#) und [ListIterator](#)
- Erstellen eigener Collection-Klassen am Beispiel einer Queue
- Die Collections des Typs [Set](#) und [Map](#) und ihre Implementierungen
- Sortierte Collections und die Bedeutung der Klassen [Comparable](#) und [Comparator](#)
- Die Hilfsklasse [Collections](#) und ihre Methoden zum Suchen, Sortieren und Kopieren von Collections

Kapitel 28

Serialisierung

- [28 Serialisierung](#)
 - [28.1 Grundlagen](#)
 - [28.1.1 Begriffsbestimmung](#)
 - [28.1.2 Schreiben von Objekten](#)
 - [28.1.3 Lesen von Objekten](#)
 - [28.2 Weitere Aspekte der Serialisierung](#)
 - [28.2.1 Versionierung](#)
 - [28.2.2 Nicht-serialisierte Membervariablen](#)
 - [28.2.3 Objektreferenzen](#)
 - [28.2.4 Serialisieren von Collections](#)
 - [28.3 Anwendungen](#)
 - [28.3.1 Ein einfacher Objektspeicher](#)
 - [28.3.2 Kopieren von Objekten](#)
 - [28.4 Zusammenfassung](#)

28.1 Grundlagen

- [28.1 Grundlagen](#)
 - [28.1.1 Begriffsbestimmung](#)
 - [28.1.2 Schreiben von Objekten](#)
 - [28.1.3 Lesen von Objekten](#)

28.1.1 Begriffsbestimmung

Unter *Serialisierung* wollen wir die Fähigkeit verstehen, ein Objekt, das im Hauptspeicher der Anwendung existiert, in ein Format zu konvertieren, das es erlaubt, das Objekt in eine Datei zu schreiben oder über eine Netzwerkverbindung zu transportieren. Dabei wollen wir natürlich auch den umgekehrten Weg einschließen, also das Rekonstruieren eines in serialisierter Form vorliegenden Objekts in das interne Format der laufenden Java-Maschine.

Serialisierung wird häufig mit dem Begriff *Persistenz* gleichgesetzt, vor allem in objektorientierten Programmiersprachen. Das ist nur bedingt richtig, denn *Persistenz* bezeichnet genau genommen das *dauerhafte* Speichern von Daten auf einem externen Datenträger, so daß sie auch nach dem Beenden des Programms erhalten bleiben. Obwohl die persistente Speicherung von Objekten sicherlich eine der Hauptanwendungen der Serialisierung ist, ist sie nicht ihre einzige. Wir werden später Anwendungen sehen, bei der die Serialisierung von Objekten nicht zum Zweck ihrer persistenten Speicherung genutzt werden.

Hinweis

28.1.2 Schreiben von Objekten

Während es vor dem JDK 1.1 keine einheitliche Möglichkeit gab, Objekte zu serialisieren, gibt es seither im Paket [java.io](#) die Klasse [ObjectOutputStream](#), mit der das sehr einfach zu realisieren ist. [ObjectOutputStream](#) besitzt einen Konstruktor, der einen [OutputStream](#) als Argument erwartet:

[java.io.ObjectOutputStream](#)

```
public ObjectOutputStream(OutputStream out)
    throws IOException
```

Der an den Konstruktor übergebene [OutputStream](#) dient als Ziel der Ausgabe. Hier kann ein beliebiges Objekt der Klasse [OutputStream](#) oder einer daraus abgeleiteten Klasse übergeben werden. Typischerweise wird ein [FileOutputStream](#) verwendet, um die serialisierten Daten in eine Datei zu schreiben.

[ObjectOutputStream](#) besitzt sowohl Methoden, um primitive Typen zu serialisieren, als auch die wichtige Methode [writeObject](#), mit der ein komplettes Objekt serialisiert werden kann:

[java.io.ObjectOutputStream](#)

```
public final void writeObject(Object obj)
    throws IOException
public void writeBoolean(boolean data)
    throws IOException
public void writeByte(int data)
    throws IOException
public void writeShort(int data)
    throws IOException
public void writeChar(int data)
    throws IOException
public void writeInt(int data)
    throws IOException
public void writeLong(long data)
    throws IOException
public void writeFloat(float data)
    throws IOException
public void writeDouble(double data)
    throws IOException
public void writeBytes(String data)
    throws IOException
```

```

public void writeChars(String data)
    throws IOException
public void writeUTF(String data)
    throws IOException

```

Während die Methoden zum Schreiben der primitiven Typen ähnlich funktionieren wie die gleichnamigen Methoden der Klasse [RandomAccessFile](#) (siehe [Abschnitt 13.4.4](#)), ist die Funktionsweise von [writeObject](#) wesentlich komplexer. [writeObject](#) schreibt folgende Daten in den [OutputStream](#):

- Die Klasse des als Argument übergebenen Objekts
- Die Signatur der Klasse
- Alle *nicht-statischen, nicht-transienten* Membervariablen des übergebenen Objekts inkl. der aus allen Vaterklassen geerbten Membervariablen

Insbesondere der letzte Punkt verdient dabei besondere Beachtung. Die Methode [writeObject](#) durchsucht also das übergebene Objekt nach Membervariablen und überprüft deren Attribute. Ist eine Membervariable vom Typ [static](#), wird es nicht serialisiert, denn es gehört nicht zum Objekt, sondern zur Klasse des Objekts. Weiterhin werden alle Membervariablen ignoriert, die mit dem Schlüsselwort [transient](#) deklariert wurden. Auf diese Weise kann das Objekt Membervariablen definieren, die aufgrund ihrer Natur nicht serialisiert werden sollen oder dürfen. Wichtig ist weiterhin, daß ein Objekt nur dann mit [writeObject](#) serialisiert werden kann, wenn es das Interface [Serializable](#) implementiert.

Aufwendiger als auf den ersten Blick ersichtlich ist das Serialisieren von Objekten vor allem aus zwei Gründen:

- Erstens muß zur Laufzeit ermittelt werden, welche Membervariablen das zu serialisierende Objekt besitzt (bzw. geerbt hat) und welche von ihnen serialisiert werden sollen. Dazu wird das *Reflection*-API verwendet (siehe [Kapitel 31](#)).
- Zweitens kann eine Membervariable natürlich selbst ein Objekt sein, das seinerseits serialisiert werden muß. Insbesondere muß [writeObject](#) dabei korrekt mit zyklischen Verweisen umgehen und dafür sorgen, daß Objektreferenzen erhalten bleiben (siehe [Abschnitt 28.2.3](#)).

Wir wollen uns zunächst ein Beispiel ansehen. Dazu konstruieren wir eine einfache Klasse [Time](#), die eine Uhrzeit, bestehend aus Stunden und Minuten, kapselt:

[Time.java](#)

```

001 /* Time.java */
002
003 import java.io.*;
004
005 public class Time
006 implements Serializable
007 {
008     private int hour;
009     private int minute;
010
011     public Time(int hour, int minute)
012     {
013         this.hour = hour;
014         this.minute = minute;
015     }
016
017     public String toString()
018     {
019         return hour + ":" + minute;
020     }
021 }

```

Listing 28.1: Eine serialisierbare Uhrzeitklasse

[Time](#) besitzt einen öffentlichen Konstruktor und eine [toString](#)-Methode zur Ausgabe der Uhrzeit. Die Membervariablen [hour](#) und [minute](#) wurden als [private](#) deklariert und sind nach außen nicht sichtbar. Die Sichtbarkeit einer Membervariable hat keinen Einfluß darauf, ob es von [writeObject](#) serialisiert wird oder nicht. Mit Hilfe eines Objekts vom Typ [ObjectOutputStream](#) kann ein [Time](#)-Objekt serialisiert werden:

```

001 /* Listing2802.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 public class Listing2802
007 {
008     public static void main(String args[])
009     {
010         try {
011             FileOutputStream fs = new FileOutputStream("test1.ser");
012             ObjectOutputStream os = new ObjectOutputStream(fs);
013             Time time = new Time(10,20);
014             os.writeObject(time);
015             os.close();
016         } catch (IOException e) {
017             System.err.println(e.toString());
018         }
019     }
020 }

```

Listing 28.2: Serialisieren eines Time-Objekts

Wir konstruieren zunächst einen [FileOutputStream](#), der das serialisierte Objekt in die Datei "test1.ser" schreiben soll. Anschließend erzeugen wir einen [ObjectOutputStream](#) durch Übergabe des [FileOutputStream](#) an dessen Konstruktor. Nun wird ein [Time](#)-Objekt für die Uhrzeit 10:20 konstruiert und mit [writeObject](#) serialisiert. Nach dem Schließen des Streams steht das serialisierte Objekt in "test1.ser".

Wichtig an der Deklaration von [Time](#) ist das Implementieren des [Serializable](#)-Interfaces. Zwar definiert [Serializable](#) keine Methoden, [writeObject](#) testet jedoch, ob das zu serialisierende Objekt dieses Interface implementiert. Ist das nicht der Fall, wird eine Ausnahme des Typs [NotSerializableException](#) ausgelöst.

Hinweis

Ein [ObjectOutputStream](#) kann nicht nur ein Objekt serialisieren, sondern beliebig viele, sie werden nacheinander in den zugrundeliegenden [OutputStream](#) geschrieben. Das folgende Programm zeigt, wie zunächst ein [int](#), dann ein [String](#) und schließlich zwei [Time](#)-Objekte serialisiert werden:

```

001 /* Listing2803.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 public class Listing2803
007 {
008     public static void main(String args[])
009     {
010         try {
011             FileOutputStream fs = new FileOutputStream("test2.ser");
012             ObjectOutputStream os = new ObjectOutputStream(fs);
013             os.writeInt(123);
014             os.writeObject("Hallo");
015             os.writeObject(new Time(10, 30));
016             os.writeObject(new Time(11, 25));
017             os.close();
018         } catch (IOException e) {
019             System.err.println(e.toString());
020         }
021     }
022 }

```

Listing 28.3: Serialisieren mehrerer Objekte

Da ein [int](#) ein primitiver Typ ist, muß er mit [writeInt](#) serialisiert werden. Bei den übrigen Aufrufen kann [writeObject](#) verwendet werden, denn alle übergebenen Argumente sind Objekte.

Es gibt keine verbindlichen Konventionen für die Benennung von Dateien mit serialisierten Objekten. Die in den Beispielen verwendete Erweiterung ".ser" ist allerdings recht häufig zu finden, ebenso wie Dateierweiterungen des Typs ".dat". Wenn eine Anwendung viele unterschiedliche Dateien mit serialisierten Objekten hält, kann es auch sinnvoll sein, die Namen nach dem Typ der serialisierten Objekte zu vergeben.

Hinweis

28.1.3 Lesen von Objekten

Nachdem ein Objekt serialisiert wurde, kann es mit Hilfe der Klasse `ObjectInputStream` wieder rekonstruiert werden. Analog zu `ObjectOutputStream` gibt es Methoden zum Wiedereinlesen von primitiven Typen und eine Methode `readObject`, mit der ein serialisiertes Objekt wieder hergestellt werden kann:

`java.io.ObjectInputStream`

```
public final Object readObject()
    throws OptionalDataException,
           ClassNotFoundException,
           IOException
public boolean readBoolean()
    throws IOException
public byte readByte()
    throws IOException
public short readShort()
    throws IOException
public char readChar()
    throws IOException
public int readInt()
    throws IOException
public long readLong()
    throws IOException
public float readFloat()
    throws IOException
public double readDouble()
    throws IOException
public String readUTF()
    throws IOException
```

Zudem besitzt die Klasse `ObjectInputStream` einen Konstruktor, der einen `InputStream` als Argument erwartet, der zum Einlesen der serialisierten Objekte verwendet wird:

`java.io.ObjectInputStream`

```
public ObjectInputStream(InputStream in)
```

Das *Deserialisieren* eines Objektes kann man sich stark vereinfacht aus den folgenden beiden Schritten bestehend vorstellen:

- Zunächst wird ein neues Objekt des zu deserialisierenden Typs angelegt, und die Membervariablen werden mit Default-Werten vorbelegt. Zudem wird der Default-Konstruktor der ersten nicht-serialisierbaren Superklasse aufgerufen.
- Anschließend werden die serialisierten Daten gelesen und den entsprechenden Membervariablen des angelegten Objekts zugewiesen.

Das erzeugte Objekt hat anschließend dieselbe Struktur und denselben Zustand, den das serialisierte Objekt hatte (abgesehen von den nicht serialisierten Membervariablen des Typs `static` oder `transient`). Da der Rückgabewert von `readObject` vom Typ `Object` ist, muß das erzeugte Objekt in den tatsächlichen Typ (oder eine seiner Oberklassen) umgewandelt werden. Das folgende Programm zeigt das Deserialisieren am Beispiel des in [Listing 28.2](#) serialisierten und in die Datei "test1.ser" geschriebenen `Time`-Objekts:

`Listing2804.java`

```
001 /* Listing2804.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 public class Listing2804
007 {
008     public static void main(String args[])
009     {
010         try {
011             FileInputStream fs = new FileInputStream("test1.ser");
012             ObjectInputStream is = new ObjectInputStream(fs);
013             Time time = (Time)is.readObject();
014             System.out.println(time.toString());
```



```

015     is.close();
016 } catch (ClassNotFoundException e) {
017     System.err.println(e.toString());
018 } catch (IOException e) {
019     System.err.println(e.toString());
020 }
021 }
022 }

```

Listing 28.4: Deserialisieren eines Time-Objekts

Hier wird zunächst ein [FileInputStream](#) für die Datei "test1.ser" geöffnet und an den Konstruktor des [ObjectInputStream](#)-Objekts `is` übergeben. Alle lesenden Aufrufe von `is` beschaffen ihre Daten damit aus "test1.ser". Jeder Aufruf von [readObject](#) liest immer das nächste gespeicherte Objekt aus dem Eingabestream. Das Programm zum Deserialisieren muß also genau wissen, welche Objekttypen in welcher Reihenfolge serialisiert wurden, um sie erfolgreich deserialisieren zu können. In unserem Beispiel ist die Entscheidung einfach, denn in der Eingabedatei steht nur ein einziges [Time](#)-Objekt. [readObject](#) deserialisiert es und liefert ein neu erzeugtes [Time](#)-Objekt, dessen Membervariablen mit den Werten aus dem serialisierten Objekt belegt werden. Die Ausgabe des Programms ist demnach:

10:20

Es ist wichtig zu verstehen, daß beim Deserialisieren nicht der Konstruktor des erzeugten Objekts aufgerufen wird. Lediglich bei einer serialisierbaren Klasse, die in ihrer Vererbungshierarchie Superklassen hat, die nicht das Interface [Serializable](#) implementieren, wird der parameterlose Konstruktor der nächsthöheren nicht-serialisierbaren Vaterklasse aufgerufen. Da die aus der nicht-serialisierbaren Vaterklasse geerbten Membervariablen nicht serialisiert werden, soll auf diese Weise sichergestellt sein, daß sie wenigstens sinnvoll initialisiert werden.

Warnung

Auch eventuell vorhandene Initialisierungen einzelner Membervariablen werden nicht ausgeführt. Wir könnten beispielsweise die [Time](#)-Klasse aus [Listing 28.1](#) um eine Membervariable `seconds` erweitern:

```
private transient int seconds = 11;
```

Dann wäre zwar bei allen mit `new` konstruierten Objekten der Sekundenwert mit 11 vorbelegt. Bei Objekten, die durch Deserialisieren erzeugt wurden, bleibt er aber 0 (das ist der Standardwert eines [int](#), siehe [Tabelle 4.1](#)), denn der Initialisierungscode wird in diesem Fall nicht ausgeführt.

Beim Deserialisieren von Objekten können einige Fehler passieren. Damit ein Aufruf von [readObject](#) erfolgreich ist, müssen mehrere Kriterien erfüllt sein:

- Das nächste Element des Eingabestreams ist tatsächlich ein *Objekt* (kein primitiver Typ).
- Das Objekt muß sich vollständig und fehlerfrei aus der Eingabedatei lesen lassen.
- Es muß eine Konvertierung auf den gewünschten Typ erlauben, also entweder zu derselben oder einer daraus abgeleiteten Klasse gehören.
- Der Bytecode für die Klasse des zu deserialisierenden Objekts muß vorhanden sein. Er wird beim Serialisieren nicht mitgespeichert, sondern muß dem Empfängerprogramm wie üblich als kompilierter Bytecode zur Verfügung stehen.
- Die Klasseninformation des serialisierten Objekts und die im deserialisierenden Programm als Bytecode vorhandene Klasse müssen zueinander kompatibel sein. Wir werden auf diesen Aspekt in [Abschnitt 28.2.1](#) detailliert eingehen.

Soll beispielsweise die in [Listing 28.3](#) erzeugte Datei "test2.ser" deserialisiert werden, so müssen die Aufrufe der `read`-Methoden in Typ und Reihenfolge denen des serialisierenden Programms entsprechen:

[Listing2805.java](#)

```

001 /* Listing2805.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 public class Listing2805
007 {
008     public static void main(String args[])
009     {
010         try {
011             FileInputStream fs = new FileInputStream("test2.ser");
012             ObjectInputStream is = new ObjectInputStream(fs);
013             System.out.println("" + is.readInt());

```

```
014      System.out.println((String)is.readObject());
015      Time time = (Time)is.readObject();
016      System.out.println(time.toString());
017      time = (Time)is.readObject();
018      System.out.println(time.toString());
019      is.close();
020  } catch (ClassNotFoundException e) {
021      System.err.println(e.toString());
022  } catch (IOException e) {
023      System.err.println(e.toString());
024  }
025  }
026 }
```

Listing 28.5: Deserialisieren mehrerer Elemente

Das Programm rekonstruiert alle serialisierten Elemente aus "test2.ser". Seine Ausgabe ist:

```
123
Hallo
10:30
11:25
```

28.2 Weitere Aspekte der Serialisierung

- [28.2 Weitere Aspekte der Serialisierung](#)
 - [28.2.1 Versionierung](#)
 - [28.2.2 Nicht-serialisierte Membervariablen](#)
 - [28.2.3 Objektreferenzen](#)
 - [28.2.4 Serialisieren von Collections](#)

Mit den Grundlagen aus dem vorigen Abschnitt sind bereits die wichtigsten Prinzipien der Serialisierung in Java erklärt. Beeindruckend ist dabei einerseits, wie das Konzept in die Klassenbibliothek eingebunden wurde. [ObjectOutputStream](#) und [ObjectInputStream](#) passen in natürlicher Weise in die Stream-Hierarchie und zeigen, wie man Streams konstruiert, die *strukturierte* Daten verarbeiten. Andererseits ist es eine große Hilfe, daß Objekte ohne größere Änderungen serialisiert werden können. Es ist lediglich erforderlich, das [Serializable](#)-Interface zu implementieren, um ein einfaches Objekt persistent machen zu können.

Dennoch ist das API leistungsfähig genug, auch komplexe Klassen serialisierbar zu machen. Wir wollen in diesem Abschnitt weiterführende Aspekte betrachten, die im Rahmen dieser Einführung noch verständlich sind. Daneben gibt es weitere Möglichkeiten, mit denen das Serialisieren und Deserialisieren von Klassen komplett an die speziellen Anforderungen einer Applikation angepaßt werden kann. Auf diese Details wollen wir hier aber nicht eingehen. Als vertiefende Lektüre empfiehlt sich die »Java Object Serialization Specification«, die Bestandteil der Online-Dokumentation des JDK 1.2 ist.

28.2.1 Versionierung

Applikationen, in denen Code und Daten getrennt gehalten werden, haben grundsätzlich mit dem Problem der Inkonsistenz beider Bestandteile zu kämpfen. Wie kann sichergestellt werden, daß die Struktur der zu verarbeitenden Daten tatsächlich den vom Programm erwarteten Strukturen entspricht? Dieses Problem gibt es bei praktisch allen Datenbankanwendungen, und es tritt immer dann verstärkt auf, wenn Code und Datenstruktur getrennt geändert werden. Auch durch das Serialisieren von Objekten haben wir das Problem, denn die Datei mit den serialisierten Objekten enthält nur die *Daten*, der zugehörige *Code* kommt dagegen aus dem `.class`-File.

Das Serialisierungs-API versucht diesem Problem mit einem *Versionierungsmechanismus* zu begegnen. Dazu enthält das Interface [Serializable](#) eine `long`-Konstante `serialVersionUID`, in der eine Versionskennung zur Klasse gespeichert wird. Sie wird beim Aufruf von `writeObject` automatisch berechnet und stellt einen Hashcode über die wichtigsten Eigenschaften der Klasse dar. So gehen beispielsweise Name und Signatur der Klasse, implementierte Interfaces sowie Methoden und Konstruktoren in die Berechnung ein. Selbst triviale Änderungen wie das Umbenennen oder Hinzufügen einer öffentlichen Methode verändern die `serialVersionUID`.

Die `serialVersionUID` einer Klasse kann mit Hilfe des Werkzeugs `serialver` ermittelt werden. Dieses einfache Programm wird zusammen mit dem Namen der Klasse in der Kommandozeile aufgerufen und liefert die Versionsnummer als Ausgabe. Alternativ kann es auch mit dem Argument `-show` aufgerufen werden. Es hat dann eine einfache Oberfläche, in der der Name der Klasse interaktiv eingegeben werden kann (siehe [Abbildung 28.1](#)).

Tip

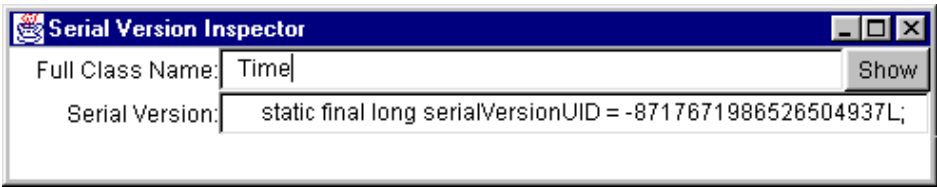


Abbildung 28.1: Das Programm serialver

Beim Serialisieren eines Objektes wird auch die `serialVersionUID` der zugehörigen Klasse mit in die Ausgabedatei geschrieben. Soll das Objekt später deserialisiert werden, so wird die in der Datei gespeicherte `serialVersionUID` mit der aktuellen `serialVersionUID` des geladenen `.class`-Files verglichen. Stimmen beide nicht überein, so gibt es eine Ausnahme des Typs [InvalidClassException](#), und der Deserialisierungsvorgang bricht ab.

Diese Art der Versionierung ist zwar recht sicher, aber auch sehr rigoros. Schon eine kleine Änderung an der Klasse macht die serialisierten Objekte unbrauchbar, weil sie sich nicht mehr deserialisieren lassen. Die in [Listing 28.1](#) vorgestellte Klasse `Time` hat die `serialVersionUID` `-8717671986526504937L`. Wird beispielsweise eine neue Methode `public void test()` hinzugefügt (die für das Deserialisieren eigentlich völlig bedeutungslos ist), ändert sich die `serialVersionUID` auf `9202005869290334574L`, und weder die Datei `"test1.ser"` noch `"test2.ser"` lassen sich zukünftig deserialisieren.

Anstatt die `serialVersionUID` automatisch berechnen zu lassen, kann sie von der zu serialisierenden Klasse auch fest vorgegeben werden. Dazu wird einfach eine Konstante `static final long serialVersionUID` definiert und mit einem vorgegebenen Wert belegt (der zum Beispiel mit Hilfe von `serialver` ermittelt wird). In diesem Fall wird die `serialVersionUID` beim Aufruf von `writeObject` nicht neu berechnet, sondern es wird der vorgegebene Wert verwendet. Läßt man diese Konstante unverändert, können beliebige Änderungen der Klasse durchgeführt werden, ohne daß `readObject` beim Deserialisieren mit einer Ausnahme abbricht. Die `Time`-Klasse aus [Listing 28.1](#) hätte dann folgendes Aussehen:

```
001 import java.io.*;
002
003 public class Time
004 implements Serializable
005 {
006     static final long serialVersionUID = -8717671986526504937L;
007
008     private int hour;
009     private int minute;
010
011     public Time(int hour, int minute)
012     {
013         this.hour = hour;
014         this.minute = minute;
015     }
016
017     public String toString()
018     {
019         return hour + ":" + minute;
020     }
021 }
```

Listing 28.6: Die Uhrzeitklasse mit `serialVersionUID`

Jetzt muß die Anwendung natürlich selbst darauf achten, daß die durchgeführten Änderungen kompatibel sind, daß also durch das Laden der Daten aus dem älteren Objekt keine Inkonsistenzen verursacht werden. Dabei mögen folgende Regeln als Anhaltspunkte dienen:

- Das Hinzufügen oder Entfernen von Methoden ist meist unkritisch.
- Das Entfernen von Membervariablen ist meist unkritisch.
- Beim Hinzufügen neuer Membervariablen muß beachtet werden, daß diese nach dem Deserialisieren uninitialized sind.
- Problematisch ist es meist, Membervariablen umzubenennen, sie auf `transient` oder `static` (oder umgekehrt) zu ändern, die Klasse von `Serializable` auf `Externalizable` (oder umgekehrt) zu ändern oder den Klassennamen zu ändern.

Solange die Änderungen kompatibel bleiben, ist also durch eine feste `serialVersionUID` sichergestellt, daß serialisierte Objekte lesbar und deserialisierbar bleiben. Sind die Änderungen dagegen inkompatibel, sollte die Konstante entsprechend geändert werden, und die serialisierten Daten dürfen nicht mehr verwendet werden (bzw. müssen vor der weiteren Verwendung konvertiert werden).

28.2.2 Nicht-serialisierte Membervariablen

Mitunter besitzt eine Klasse Membervariablen, die nicht serialisiert werden sollen. Typische Beispiele sind Variablen, deren Wert sich während des Programmlaufs dynamisch ergibt, oder solche, die nur der temporären Kommunikation zwischen zwei oder mehr Methoden dienen. Auch Daten, die nur im Kontext der laufenden Anwendung Sinn machen, wie beispielsweise Filehandles, Sockets oder GUI-Ressourcen, sollten nicht serialisiert werden; sie »verfallen« mit dem Ende des Programms.

Membervariablen, die nicht serialisiert werden sollen, können mit dem Attribut `transient` versehen werden. Dadurch werden sie beim Schreiben des Objekts mit `writeObject` ignoriert und gelangen nicht in die Ausgabedatei. Beim Deserialisieren werden die transienten Objekte lediglich mit dem typspezifischen Standardwert belegt.

28.2.3 Objektreferenzen

Eine wichtige Eigenschaft des Serialisierungs-APIs im JDK besteht darin, daß auch *Referenzen* automatisch gesichert und rekonstruiert werden. Besitzt ein Objekt selbst Strings, Arrays oder andere Objekte als Membervariablen, so werden diese ebenso wie die primitiven Typen serialisiert und deserialisiert. Da eine Objektvariable lediglich einen *Verweis* auf das im Hauptspeicher allozierte Objekt darstellt, ist es wichtig, daß diese Verweise auch nach dem Serialisieren/Deserialisieren erhalten bleiben. Insbesondere darf ein Objekt auch dann nur einmal angelegt werden, wenn darauf von mehr als einer Variable verwiesen wird. Auch nach dem Deserialisieren darf das Objekt nur einmal vorhanden sein, und die verschiedenen Objektvariablen müssen auf dieses Objekt zeigen.

Der `ObjectOutputStream` hält zu diesem Zweck eine Hashtabelle, in der alle bereits serialisierten Objekte verzeichnet werden. Bei jedem Aufruf von `writeObject` wird zunächst in der Tabelle nachgesehen, ob das Objekt bereits serialisiert wurde. Ist das der Fall, wird in der

Ausgabedatei lediglich ein Verweis auf das Objekt gespeichert. Andernfalls wird das Objekt serialisiert und in der Hashtabelle eingetragen. Beim Deserialisieren eines Verweises wird dieser durch einen Objektverweis auf das zuvor deserialisierte Objekt ersetzt. Auf diese Weise werden Objekte nur einmal gespeichert, die Objektreferenzen werden konserviert, und das Problem von Endlosschleifen durch zyklische Referenzen ist ebenfalls gelöst.

Das folgende Programm zeigt das Speichern von Verweisen am Beispiel eines Graphen, der Eltern-Kind-Beziehungen darstellt. Zunächst benötigen wir dazu eine Klasse [Person](#), die den Namen und die Eltern einer Person speichern kann. Jeder Elternteil wird dabei durch einen Verweis auf eine weitere Person dargestellt:

[Person.java](#)

```
001 import java.io.*;
002
003 public class Person
004 implements Serializable
005 {
006     public String name;
007     public Person mother;
008     public Person father;
009
010     public Person(String name)
011     {
012         this.name = name;
013     }
014 }
```

Listing 28.7: Die Klasse Person

Der Einfachheit halber wurden alle Membervariablen als [public](#) deklariert. Wir wollen nun ein Programm erstellen, das den folgenden Eltern-Kind-Graph aufbaut:

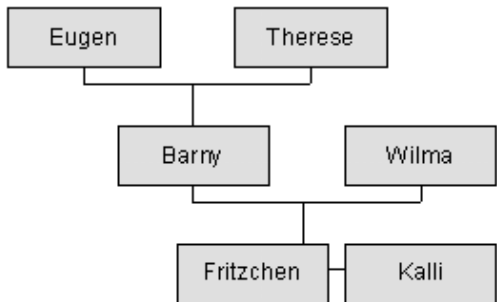


Abbildung 28.2: Eltern-Kind-Graph für Serialisierungsbeispiel

Das Programm soll den Graph dann in eine Datei "test3.ser" serialisieren und anschließend durch Deserialisieren wieder rekonstruieren. Wir wollen dann überprüfen, ob alle Verweise wiederhergestellt wurden und ob die Objekteindeutigkeit gewahrt wurde.

[Listing2808.java](#)

```
001 /* Listing2808.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 public class Listing2808
007 {
008     public static void main(String args[])
009     {
010         //Erzeugen der Familie
011         Person opa = new Person("Eugen");
012         Person oma = new Person("Therese");
013         Person vater = new Person("Barry");
014         Person mutter = new Person("Wilma");
015         Person kind1 = new Person("Fritzchen");
016         Person kind2 = new Person("Kalli");
017         vater.father = opa;
018         vater.mother = oma;
019         kind1.father = kind2.father = vater;
020         kind1.mother = kind2.mother = mutter;
021     }
```

```

022 //Serialisieren der Familie
023 try {
024     FileOutputStream fs = new FileOutputStream("test3.ser");
025     ObjectOutputStream os = new ObjectOutputStream(fs);
026     os.writeObject(kind1);
027     os.writeObject(kind2);
028     os.close();
029 } catch (IOException e) {
030     System.err.println(e.toString());
031 }
032
033 //Rekonstruieren der Familie
034 kind1 = kind2 = null;
035 try {
036     FileInputStream fs = new FileInputStream("test3.ser");
037     ObjectInputStream is = new ObjectInputStream(fs);
038     kind1 = (Person)is.readObject();
039     kind2 = (Person)is.readObject();
040     //Überprüfen der Objekte
041     System.out.println(kind1.name);
042     System.out.println(kind2.name);
043     System.out.println(kind1.father.name);
044     System.out.println(kind1.mother.name);
045     System.out.println(kind2.father.name);
046     System.out.println(kind2.mother.name);
047     System.out.println(kind1.father.father.name);
048     System.out.println(kind1.father.mother.name);
049     //Name des Vaters ändern
050     kind1.father.name = "Fred";
051     //Erneutes Überprüfen der Objekte
052     System.out.println("---");
053     System.out.println(kind1.name);
054     System.out.println(kind2.name);
055     System.out.println(kind1.father.name);
056     System.out.println(kind1.mother.name);
057     System.out.println(kind2.father.name);
058     System.out.println(kind2.mother.name);
059     System.out.println(kind1.father.father.name);
060     System.out.println(kind1.father.mother.name);
061     is.close();
062 } catch (ClassNotFoundException e) {
063     System.err.println(e.toString());
064 } catch (IOException e) {
065     System.err.println(e.toString());
066 }
067 }
068 }

```

Listing 28.8: Serialisieren von Objekten und Referenzen

Das Programm erzeugt in den Zeilen [011](#) bis [020](#) zunächst den in [Abbildung 28.2](#) abgebildeten Verwandtschaftsgraph und serialisiert ihn anschließend in die Datei "test3.ser". Bemerkenswert ist hier vor allem, daß wir lediglich die beiden Kinder `kind1` und `kind2` explizit serialisieren. Da alle anderen Objekte über Verweise von den Kindern aus zu erreichen sind, ist es nicht nötig, diese separat mit `writeObject` zu speichern.

In [Zeile 034](#) setzen wir die beiden Kindvariablen auf `null`, um zu beweisen, daß sie ausschließlich durch das nachfolgende Deserialisieren korrekt gesetzt werden. Nun werden `kind1` und `kind2` deserialisiert, und in den Zeilen [041](#) bis [048](#) wird der komplette Verwandtschaftsgraph ausgegeben. An der Ausgabe des Programms können wir erkennen, daß tatsächlich alle Objekte rekonstruiert und die Verweise darauf korrekt gesetzt wurden:

```

Fritzchen
Kalli
Barny
Wilma
Barny
Wilma
Eugen
Therese

```

Fritzchen
Kalli
Fred
Wilma
Fred
Wilma
Eugen
Therese

Der zweite Block von Ausgabeanweisungen (in den Zeilen [052](#) bis [060](#)) zeigt, daß auch die Objekteindeutigkeit gewahrt wurde. Dazu haben wir nämlich in [Zeile 050](#) den Namen des Vaterobjekts von `kind1` auf "Fred" geändert. Wie im zweiten Teil der Ausgabe des Programms zu erkennen ist, wurde damit auch der Name des Vaters des zweiten Kindes auf "Fred" geändert, und wir können sicher sein, daß es sich um ein und dasselbe Objekt handelt.

Obwohl (oder gerade weil) das Serialisieren von Objektgraphen in aller Regel sehr bequem und vollautomatisch abläuft, seien an dieser Stelle einige Warnungen ausgesprochen:

Warnung

- Einerseits kann es passieren, daß mehr Objekte als erwartet serialisiert werden. Insbesondere bei komplexen Objektbeziehungen kann es sein, daß an dem zu serialisierenden Objekt indirekt viele weitere Objekte hängen und beim Serialisieren wesentlich mehr Objekte gespeichert werden, als erwartet wurde. Das kostet unnötig Zeit und Speicher.
- Durch das Zwischenspeichern der bereits serialisierten Objekte in [ObjectOutputStream](#) werden viele Verweise auf Objekte gehalten, die sonst möglicherweise für das Programm unerreichbar wären. Da der Garbage Collector diese Objekte nicht freigibt, kann es beim Serialisieren einer großen Anzahl von Objekten zu Speicherproblemen kommen. Mit Hilfe der Methode [reset](#) kann der [ObjectOutputStream](#) in den Anfangszustand versetzt; alle bereits bekannten Objektreferenzen werden »vergessen«. Wird ein bereits serialisiertes Objekt danach noch einmal gespeichert, wird kein Verweis, sondern das Objekt selbst noch einmal geschrieben.
- Wenn ein bereits serialisiertes Objekt *verändert* und anschließend erneut serialisiert wird, bleibt die Veränderung beim Deserialisieren unsichtbar, denn in der Ausgabedatei wird lediglich ein Verweis auf das Originalobjekt gespeichert.

28.2.4 Serialisieren von Collections

Neben selbstgeschriebenen Klassen sind auch viele der Standardklassen des JDK serialisierbar, insbesondere die meisten Collection-Klassen. Um beispielsweise alle Daten eines Vektors oder einer Hashtable persistent zu speichern, genügt ein einfaches Serialisieren nach obigem Muster. Voraussetzung ist allerdings, daß auch die Elemente der Collection serialisierbar sind, andernfalls gibt es eine [NotSerializableException](#). Auch die Wrapperklassen zu den Basistypen (siehe [Abschnitt 7.8.1](#)) sind standardmäßig serialisierbar und können damit problemlos als Objekte in serialisierbaren Collections verwendet werden. Im nächsten Abschnitt stellen wir eine kleine Anwendung für das Serialisieren von Hashtabellen vor.

28.3 Anwendungen

- [28.3 Anwendungen](#)
 - [28.3.1 Ein einfacher Objektspeicher](#)
 - [28.3.2 Kopieren von Objekten](#)

28.3.1 Ein einfacher Objektspeicher

Die folgende Klasse `TrivialObjectStore` stellt ein einfaches Beispiel für einen persistenten Objektspeicher dar. Sie besitzt eine Methode `putObject`, mit der beliebige `String-Object`-Paare angelegt und später mit `getObject` wieder abgerufen werden können. Durch Aufruf von `save` kann der komplette Objektspeicher serialisiert werden, ein Aufruf von `load` lädt ihn von der Festplatte. Die Klasse `TrivialObjectStore` verwendet eine `Hashtable` zur Ablage der `String-Object`-Paare. Diese implementiert bereits standardmäßig das Interface `Serializable` und kann daher sehr einfach auf einem externen Datenträger gespeichert oder von dort geladen werden.

[TrivialObjectStore.java](#)

```

001 /* TrivialObjectStore.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 /**
007  * Trivialer Objektspeicher, der Mengen von Name-Objekt-
008  * Paaren aufnehmen und persistent speichern kann.
009  */
010 public class TrivialObjectStore
011 {
012     //Instance variables
013     private String fname;
014     private Hashtable objects;
015
016     /**
017      * Erzeugt einen neuen Objektspeicher mit dem angegebenen
018      * Namen (die Erweiterung ".tos" ("trivial object store")
019      * wird ggfs. automatisch angehängt.
020      */
021     public TrivialObjectStore(String fname)
022     {
023         this.fname = fname;
024         if (!fname.endsWith(".tos")) {
025             this.fname += ".tos";
026         }
027         this.objects = new Hashtable(50);
028     }
029
030     /**
031      * Sichert den Objektspeicher unter dem im Konstruktor
032      * angegebenen Namen.
033      */
034     public void save()
035     throws IOException
036     {
037         FileOutputStream fs = new FileOutputStream(fname);
038         ObjectOutputStream os = new ObjectOutputStream(fs);
039         os.writeObject(objects);
040         os.close();
041     }
042
043     /**
044      * Lädt den Objektspeicher mit dem im Konstruktor

```



```

045     * angegebenen Namen.
046     */
047     public void load()
048     throws ClassNotFoundException, IOException
049     {
050         FileInputStream fs = new FileInputStream(fname);
051         ObjectInputStream is = new ObjectInputStream(fs);
052         objects = (Hashtable)is.readObject();
053         is.close();
054     }
055
056     /**
057     * Fügt ein Objekt in den Objektspeicher ein.
058     */
059     public void putObject(String name, Object object)
060     {
061         objects.put(name, object);
062     }
063
064     /**
065     * Liest das Objekt mit dem angegebenen Namen aus dem
066     * Objektspeicher. Ist es nicht vorhanden, wird null
067     * zurückgegeben.
068     */
069     public Object getObject(String name)
070     {
071         return objects.get(name);
072     }
073
074     /**
075     * Liefert eine Aufzählung aller gespeicherten Namen.
076     */
077     public Enumeration getAllNames()
078     {
079         return objects.keys();
080     }
081 }

```

Listing 28.9: Ein einfacher Objektspeicher

Objekte der Klasse `TrivialObjectStore` können nun verwendet werden, um beliebige serialisierbare Objekte unter Zuordnung eines Namens auf einem externen Datenträger zu speichern. Das folgende Listing zeigt dies am Beispiel eines fiktiven »Tamagotchi-Shops«, dessen Eigenschaften *Name*, *Besitzer* und *Liste der Produkte* im Objektspeicher abgelegt, in die Datei "shop.tos" geschrieben und anschließend wieder ausgelesen werden:

[Listing2810.java](#)

```

001 /* Listing2810.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 public class Listing2810
007 {
008     public static void main(String args[])
009     {
010         //Erzeugen und Speichern des Objektspeichers
011         TrivialObjectStore tos = new TrivialObjectStore("shop");
012         tos.putObject("name", "Tami-Shop Norderelbe");
013         tos.putObject("besitzer", "Meier, Fridolin");
014         Vector products = new Vector(10);
015         products.addElement("Dinky Dino");
016         products.addElement("96er Classic");
017         products.addElement("Black Frog");
018         products.addElement("SmartGotchi");
019         products.addElement("Pretty Dolly");
020         tos.putObject("produkte", products);
021         try {
022             tos.save();

```

```

023     } catch (IOException e) {
024         System.err.println(e.toString());
025     }
026
027     //Einlesen des Objektspeichers
028     TrivialObjectStore tos2 = new TrivialObjectStore("shop");
029     try {
030         tos2.load();
031         Enumeration names = tos2.getAllNames();
032         while (names.hasMoreElements()) {
033             String name = (String)names.nextElement();
034             Object obj = tos2.getObject(name);
035             System.out.print(name + ": ");
036             System.out.println(obj.getClass().toString());
037             if (obj instanceof Collection) {
038                 Iterator it = ((Collection)obj).iterator();
039                 while (it.hasNext()) {
040                     System.out.println("    " + it.next().toString());
041                 }
042             } else {
043                 System.out.println("    " + obj.toString());
044             }
045         }
046     } catch (IOException e) {
047         System.err.println(e.toString());
048     } catch (ClassNotFoundException e) {
049         System.err.println(e.toString());
050     }
051 }
052 }

```

Listing 28.10: Beispielanwendung für den einfachen Objektspeicher

Hier wird zunächst ein neuer Objektspeicher `tos` erstellt und mit den Objekten aus dem Tamagotchi-Shop gefüllt. Neben zwei Strings `name` und `besitzer` wird dabei unter der Bezeichnung `produkte` eine weitere Collection, der `Vector` mit den Produkten, eingefügt. Das durch Aufruf von `save` ausgelöste Serialisieren der `Hashtable` bewirkt, daß alle darin gespeicherten Elemente serialisiert werden, sofern sie das Interface `Serializable` implementieren.

Ab [Zeile 027](#) wird dann der Objektspeicher wieder eingelesen, in diesem Fall in die Variable `tos2`. Mit `getAllNames` beschafft das Programm zunächst eine `Enumeration` über alle Objektnamen und durchläuft sie elementweise. Zu jedem Namen wird mit `getElement` das zugehörige Element geholt, und sein Name und der Name der zugehörigen Klasse werden ausgegeben ([Zeile 036](#)). Anschließend wird überprüft, ob das gefundene Objekt das Interface `Collection` implementiert und ggfs. über alle darin enthaltenen Elemente iteriert. Andernfalls wird das Objekt direkt mit `toString` ausgegeben.

Die Ausgabe des Programms ist:

```

produkte: class java.util.Vector
  Dinky Dino
  96er Classic
  Black Frog
  SmartGotchi
  Pretty Dolly
besitzer: class java.lang.String
  Meier, Fridolin
name: class java.lang.String
  Tami-Shop Norderelbe

```

Die Klasse `TrivialObjectStore` verdeutlicht *eine mögliche* Vorgehensweise bei der persistenten Speicherung von Objekten. Für einen echten Praxiseinsatz (etwa in der Anwendungsentwicklung) fehlen aber noch ein paar wichtige Eigenschaften:

- Anstatt den Objektspeicher immer *komplett* zu laden und zu speichern, sollte es möglich sein, einzelne Elemente zu speichern, zu laden und zu löschen.
- Der Objektspeicher sollte mehrbenutzerfähig sein und Transaktions- und Recovery-Logik mitbringen.
- Die Suche nach Objekten sollte durch Indexdateien beschleunigt werden können.

Hinweis

Leider ist die Implementierung dieser Features nicht trivial. Ein gutes Beispiel für die Implementierung des ersten und dritten Punkts findet sich in "Java Algorithms" von Scott Robert Ladd. Der Autor zeigt zunächst, wie man Objekte auf der Basis von Random-Access-Dateien (anstelle der üblichen Streams) serialisiert. Anschließend zeigt er die Verwendung von Indexdateien am Beispiel der Implementierung von B-Trees.

28.3.2 Kopieren von Objekten

Eine auf den ersten Blick überraschende Anwendung der Serialisierung besteht darin, Objekte zu kopieren. Es sei noch einmal daran erinnert, daß die Zuweisung eines Objektes an eine Objektvariable lediglich eine Zeigeroperation war; daß also immer nur ein Verweis geändert wurde. Soll ein komplexes Objekt kopiert werden, wird dazu üblicherweise das Interface [Cloneable](#) implementiert und eine Methode [clone](#) zur Verfügung gestellt, die den eigentlichen Kopiervorgang vornimmt. Sollte ein Objekt kopiert werden, das [Cloneable](#) nicht implementiert, blieb bisher nur der umständliche Weg über das manuelle Kopieren aller Membervariablen. Das ist insbesondere dann mühsam und fehlerträchtig, wenn das zu kopierende Objekt Unterobjekte enthält, die ihrerseits kopiert werden müssen (*deep copy* anstatt *shallow copy*).

Der Schlüssel zum Kopieren von Objekten mit Hilfe der Serialisierung liegt darin, anstelle der üblichen *dateibasierten* Streamklassen solche zu verwenden, die ihre Daten im Hauptspeicher halten. Am besten sind dazu [ByteArrayOutputStream](#) und [ByteArrayInputStream](#) geeignet. Sie sind integraler Bestandteil der [OutputStream](#)- und [InputStream](#)-Hierarchien und man kann die Daten problemlos von einem zum anderen übergeben. Das folgende Programm implementiert eine Methode [seriaClone\(\)](#), die ein beliebiges Objekt als Argument erwartet und in einen [ByteArrayOutputStream](#) serialisiert. Das resultierende Byte-Array wird dann zur Konstruktion eines [ByteArrayInputStream](#) verwendet, dort deserialisiert und als Objektkopie an den Aufrufer zurückgegeben:

[Listing2811.java](#)

```
001 /* Listing2811.java */
002
003 import java.io.*;
004 import java.util.*;
005
006 public class Listing2811
007 {
008     public static Object seriaClone(Object o)
009     throws IOException, ClassNotFoundException
010     {
011         //Serialisieren des Objekts
012         ByteArrayOutputStream out = new ByteArrayOutputStream();
013         ObjectOutputStream os = new ObjectOutputStream(out);
014         os.writeObject(o);
015         os.flush();
016         //Deserialisieren des Objekts
017         ByteArrayInputStream in = new ByteArrayInputStream(
018             out.toByteArray()
019         );
020         ObjectInputStream is = new ObjectInputStream(in);
021         Object ret = is.readObject();
022         is.close();
023         os.close();
024         return ret;
025     }
026
027     public static void main(String args[])
028     {
029         try {
030             //Erzeugen des Buchobjekts
031             Book book = new Book();
032             book.author = "Peitgen, Heinz-Otto";
033             String s[] = {"Jürgens, Hartmut", "Saupe, Dietmar"};
034             book.coAuthors = s;
035             book.title = "Bausteine des Chaos";
036             book.publisher = "rororo science";
037             book.pubyear = 1998;
038             book.pages = 514;
039             book.isbn = "3-499-60250-4";
040             book.reflist = new Vector();
041             book.reflist.addElement("The World of MC Escher");
042             book.reflist.addElement(
043                 "Die fraktale Geometrie der Natur"
044             );
045             book.reflist.addElement("Gödel, Escher, Bach");
```

```

046     System.out.println(book.toString());
047     //Erzeugen und Verändern der Kopie
048     Book copy = (Book)seriaClone(book);
049     copy.title += " - Fraktale";
050     copy.reflist.addElement("Fractal Creations");
051     //Ausgeben von Original und Kopie
052     System.out.print(book.toString());
053     System.out.println("---");
054     System.out.print(copy.toString());
055 } catch (IOException e) {
056     System.err.println(e.toString());
057 } catch (ClassNotFoundException e) {
058     System.err.println(e.toString());
059 }
060 }
061 }
062
063 class Book
064 implements Serializable
065 {
066     public String author;
067     public String coAuthors[];
068     public String title;
069     public String publisher;
070     public int    pubyear;
071     public int    pages;
072     public String isbn;
073     public Vector reflist;
074
075     public String toString()
076     {
077         String NL = System.getProperty("line.separator");
078         StringBuffer ret = new StringBuffer(200);
079         ret.append(author + NL);
080         for (int i = 0; i < coAuthors.length; ++i) {
081             ret.append(coAuthors[i] + NL);
082         }
083         ret.append("\\" + title + "\\" + NL);
084         ret.append(publisher + " " + pubyear + NL);
085         ret.append(pages + " pages" + NL);
086         ret.append(isbn + NL);
087         Enumeration e = reflist.elements();
088         while (e.hasMoreElements()) {
089             ret.append(" " + (String)e.nextElement() + NL);
090         }
091         return ret.toString();
092     }
093 }

```

Listing 28.11: Kopieren von Objekten durch Serialisierung

Das Programm verwendet zum Testen ein Objekt der Klasse `Book`, das mit den Daten eines Buchtitels initialisiert wird. Anschließend wird mit `seriaClone` eine Kopie hergestellt und der Variable `copy` zugewiesen. Um zu verdeutlichen, daß wirklich eine Kopie hergestellt wurde, modifizieren wir nun einige Angaben der Kopie und geben anschließend beide Objekte aus:

```

Peitgen, Heinz-Otto
Jürgens, Hartmut
Saupe, Dietmar
"Bausteine des Chaos"
rororo science 1998
514 pages
3-499-60250-4
    The World of MC Escher
    Die fraktale Geometrie der Natur
    Gödel, Escher, Bach
---
```

Peitgen, Heinz-Otto
Jürgens, Hartmut
Saupe, Dietmar
"Bausteine des Chaos - Fraktale"
rororo science 1998
514 pages
3-499-60250-4
The World of MC Escher
Die fraktale Geometrie der Natur
Gödel, Escher, Bach
Fractal Creations

An der Programmausgabe kann man erkennen, daß das Objekt tatsächlich ordnungsgemäß kopiert wurde. Auch alle Unterobjekte wurden kopiert und konnten anschließend unabhängig voneinander geändert werden. Ohne Serialisierung wäre der manuelle Aufwand um ein Vielfaches größer gewesen. Das Verfahren findet dort seine Grenzen, wo die zu kopierenden Objekte nicht serialisierbar sind oder nicht-serialisierbare Unterobjekte enthalten. Zudem muß im echten Einsatz das Laufzeitverhalten überprüft werden, denn der Vorgang des Serialisierens/Deserialisierens ist um ein Vielfaches langsamer als das direkte Kopieren der Objektattribute.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

28.4 Zusammenfassung

- [28.4 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Grundlegende Konzepte der Serialisierung in Java
- Lesen und Schreiben von Objekten mit den Klassen [ObjectOutputStream](#) und [ObjectInputStream](#)
- Das Interface [Serializable](#)
- Versionierung und die Membervariable [serialVersionUID](#)
- Serialisieren von Objektreferenzen
- Serialisieren von Collections
- Konstruktion eines einfachen persistenten Objektspeichers
- Verwendung der Serialisierung zum Kopieren von Objekten

Kapitel 29

Performance-Tuning

- [29 Performance-Tuning](#)
 - [29.1 Einleitung](#)
 - [29.2 Tuning-Tips](#)
 - [29.2.1 String und StringBuffer](#)
 - [String-Verkettung](#)
 - [Einfügen und Löschen in Strings](#)
 - [Die Methode toString der Klasse StringBuffer](#)
 - [Die Unveränderlichkeit von String-Objekten](#)
 - [Durchlaufen von Zeichenketten](#)
 - [29.2.2 Methodenaufrufe](#)
 - [29.2.3 Vektoren und Listen](#)
 - [29.2.4 Dateizugriffe](#)
 - [Schreiben von Streams](#)
 - [Lesen von Streams](#)
 - [RandomAccess-Dateien](#)
 - [29.3 Einsatz eines Profilers](#)
 - [29.3.1 Erzeugen der Profiling-Informationen](#)
 - [29.3.2 Auswerten der Profiling-Informationen](#)
 - [29.3.3 Ausblick](#)
 - [29.4 Zusammenfassung](#)

29.1 Einleitung

- 29.1 Einleitung

Java gilt gemeinhin als Sprache, die mit Performance-Problemen zu kämpfen hat. Nicht nur die Ablaufgeschwindigkeit des Compilers und anderer Entwicklungswerkzeuge, sondern vor allem die der eigenen Programme läßt oft zu wünschen übrig. Aufgrund der Plattformunabhängigkeit des vom Compiler generierten Bytecodes kann dieser normalerweise nicht direkt auf dem jeweiligen Betriebssystem ausgeführt werden. Er verwendet statt dessen einen eigenen Interpreter, die *Virtuelle Maschine* (kurz: VM), zur Ausführung der erstellten Programme.

Interpretierter Code wird naturgemäß langsamer ausgeführt als kompilierter, selbst wenn er in Form von Bytecodes vorliegt. Zwar ist es prinzipiell möglich, auch Java-Programme in Native-Code zu übersetzen (unter Windows sind dazu beispielsweise *Visual Cafe* von Symantec oder *SuperCede* von Asymetrix in der Lage), aber dann ist es mit der Plattformunabhängigkeit aus, und das fertige Programm läuft nur noch auf einem Betriebssystem. Während das für *Applikationen* in bestimmten Fällen akzeptabel sein mag, verbietet sich diese Vorgehensweise für *Applets*, die im Internet auf vielen verschiedenen Browsern und Betriebssystemen laufen müssen, von selbst. Zudem konterkarieren native-kompilierte Programme die Grundidee der plattformübergreifenden Binärkompatibilität, die eine der herausragenden Eigenschaften von Java ist.

Eine Alternativlösung bieten *Just-In-Time-Compiler* (kurz: *JIT*), deren Entwicklung in großen Schritten voranschreitet. Ein JIT ist ein Programm, das den Bytecode während der Ausführung in Maschinencode der aktuellen Plattform übersetzt und so beim nächsten Mal wesentlich schneller ausführen kann. Vorteilhaft ist dabei, daß die Klassendateien mit dem Bytecode unverändert ausgeliefert werden können und das Programm seinen plattformübergreifenden Charakter erhält. Nur der Just-In-Time-Compiler ist plattformspezifisch und an ein bestimmtes Betriebssystem gebunden. Nahezu alle Hersteller kommerzieller Java-Produkte haben mittlerweile einen JIT als festen Bestandteil ihres Java-Entwicklungssystems eingebunden. Auch SUN liefert seit dem JDK 1.1.6 den Just-In-Time-Compiler von Symantec als festen Bestandteil des JDK aus.

Leider ist auch ein Just-In-Time-Compiler kein Allheilmittel gegen Performanceprobleme. Zwar ist er in der Lage, bestimmte Codeteile so stark zu beschleunigen, daß ihre Ablaufgeschwindigkeit der von kompiliertem C-Code nahekommt. Andererseits gibt es nach wie vor genügend Möglichkeiten, Programme zu schreiben, die *inhärent langsamen* Code enthalten, der auch von einem Just-In-Time-Compiler nicht entscheidend verbessert werden kann. Zudem entsteht durch den Einsatz des JIT ein gewisser Overhead, der möglicherweise einen Netto-Performancegewinn verhindert, denn das Kompilieren des Bytecodes kostet Zeit und zusätzlichen Speicher.

Des weiteren ist zu bedenken, daß zur Laufzeit eine Vielzahl von Checks durchgeführt werden müssen, die die Ablaufgeschwindigkeit von Java-Programmen vermindert:

- Array- und String-Zugriffe werden auf Bereichsüberschreitungen geprüft.
- Zeiger werden vor der Dereferenzierung gegen `null` gecheckt.
- Zuweisungen von Objektinstanzen werden auf korrekte Typisierung geprüft.
- Es gibt Checks zu vielen arithmetischen Operationen (Überläufe, Teilen durch Null usw.).

Am besten ist es daher, bereits während der Entwicklung der Programme auf die Ablaufgeschwindigkeit des erzeugten Codes zu achten. Wir wollen uns in diesem Kapitel einige typische Java-Konstrukte ansehen, die bei unachtsamer Verwendung zu Performance-Problemen führen können. Gleichzeitig wollen wir Möglichkeiten aufzeigen, wie man mit alternativem Code den Engpaß umgehen und die Ablaufgeschwindigkeit des Programms verbessern kann. Wenn man diese Regeln beachtet, ist es durchaus möglich, in Java größere Programme zu schreiben, deren Laufzeitverhalten auf aktuellen Rechnern durchaus akzeptabel ist.

Wir wollen uns in diesem Kapitel nicht mit grundlegenden Techniken der Codeoptimierung beschäftigen. Auch wenn sie zeitweilig kurz angedeutet werden, können diese Themen besser in Büchern über Programmiersprachen, Algorithmen oder Optimierungstechniken für Compiler nachgelesen werden. Auch Tips & Tricks, die in aller Regel nur marginale Verbesserungen bringen, oder langsamer Code, für den keine einfach anzuwendenden Alternativen bekannt sind, sollen hier nicht behandelt werden. Statt dessen wollen wir uns auf einige große Themenkomplexe konzentrieren, die leicht umzusetzen sind und in der Praxis schnell zu Verbesserungen führen.

Hinweis

29.2 Tuning-Tips

- [29.2 Tuning-Tips](#)
 - [29.2.1 String und StringBuffer](#)
 - [String-Verkettung](#)
 - [Einfügen und Löschen in Strings](#)
 - [Die Methode toString der Klasse StringBuffer](#)
 - [Die Unveränderlichkeit von String-Objekten](#)
 - [Durchlaufen von Zeichenketten](#)
 - [29.2.2 Methodenaufrufe](#)
 - [29.2.3 Vektoren und Listen](#)
 - [29.2.4 Dateizugriffe](#)
 - [Schreiben von Streams](#)
 - [Lesen von Streams](#)
 - [RandomAccess-Dateien](#)

29.2.1 String und StringBuffer

String-Verkettung

In Java gibt es zwei unterschiedliche Klassen [String](#) und [StringBuffer](#) zur Verarbeitung von Zeichenketten, deren prinzipielle Eigenschaften in [Kapitel 11](#) erläutert wurden. Java-Anfänger verwenden meist hauptsächlich die Klasse [String](#), denn sie stellt die meisten Methoden zur Zeichenkettenextraktion und -verarbeitung zur Verfügung und bietet mit dem `+`-Operator eine bequeme Möglichkeit, Zeichenketten miteinander zu verketteten.

Daß diese Bequemlichkeit ihren Preis hat, zeigt folgender Programmausschnitt:

```
001 String s;
002 s = "";
003 for (int i = 0; i < 20000; ++i) {
004     s += "x";
005 }
```

Listing 29.1: Langsame String-Verkettung

Das Programmfragment hat die Aufgabe, einen String zu erstellen, der aus 20000 aneinandergereihten "x" besteht. Das ist zwar nicht sehr praxisnah, illustriert aber die häufig vorkommende Verwendung des `+=`-Operators auf Strings. Der obige Code ist sehr ineffizient, denn er läuft langsam und belastet das Laufzeitsystem durch 60000 temporäre Objekte, die alloziert und vom Garbage Collector wieder freigegeben werden müssen. Der Compiler übersetzt das Programmfragment etwa so:

```
001 String s;
002 s = "";
003 for (int i = 0; i < 20000; ++i) {
004     s = new StringBuffer(s).append("x").toString();
005 }
```

Listing 29.2: Wie der Java-Compiler String-Verkettungen übersetzt

Dieser Code ist in mehrfacher Hinsicht unglücklich. Pro Schleifendurchlauf wird ein temporäres [StringBuffer](#)-Objekt alloziert und mit dem zuvor erzeugten String initialisiert. Der Konstruktor von [StringBuffer](#) erzeugt ein internes Array (also eine weitere Objektinstanz), um die Zeichenkette zu speichern. Immerhin ist dieses Array 16 Byte größer als eigentlich erforderlich, so daß der nachfolgende Aufruf von [append](#) das Array nicht neu allozieren und die Zeichen umkopieren muß. Schließlich wird durch den Aufruf von [toString](#) ein neues [String](#)-Objekt erzeugt und `s` zugewiesen. Auf diese Weise werden pro Schleifendurchlauf drei temporäre Objekte erzeugt, und der Code ist durch das wiederholte Kopieren der Zeichen im Konstruktor von [StringBuffer](#) sehr ineffizient.

Eine eminente Verbesserung ergibt sich, wenn die Klasse [StringBuffer](#) und ihre Methode [append](#) direkt verwendet werden:

```

001 String s;
002 StringBuffer sb = new StringBuffer(1000);
003 for (int i = 0; i < 20000; ++i) {
004     sb.append("x");
005 }
006 s = sb.toString();

```

Listing 29.3: Performante String-Verkettungen mit `StringBuffer.append`

Hier wird zunächst ein [StringBuffer](#) erzeugt und mit einem 1000 Zeichen großen Puffer versehen. Da die [StringBuffer](#)-Klasse sich die Länge der gespeicherten Zeichenkette merkt, kann der Aufruf `append("x")` meist in konstanter Laufzeit erfolgen. Dabei ist ein Umkopieren nur dann erforderlich, wenn der interne Puffer nicht mehr genügend Platz bietet, um die an [append](#) übergebenen Daten zu übernehmen. In diesem Fall wird ein größeres Array alloziert und der Inhalt des bisherigen Puffers umkopiert. In der Summe ist die letzte Version etwa um den Faktor 10 schneller als die ersten beiden und erzeugt 60000 temporäre Objekte weniger.

Interessant ist dabei der Umfang der Puffervergrößerung, den das [StringBuffer](#)-Objekt vornimmt, denn er bestimmt, wann bei fortgesetztem Aufruf von [append](#) das nächste Mal umkopiert werden muß. Anders als beispielsweise bei der Klasse [Vector](#), die einen veränderbaren *Ladefaktor* besitzt, *verdoppelt* sich die Größe eines [StringBuffer](#)-Objekts bei jeder Kapazitätserweiterung. Dadurch wird zwar möglicherweise mehr Speicher als nötig alloziert, aber die Anzahl der Kopiervorgänge wächst höchstens logarithmisch mit der Gesamtmenge der eingefügten Daten. In unserem Beispiel kann der interne Puffer zunächst 1000 Zeichen aufnehmen, wird beim nächsten Überlauf auf etwa 2000 Zeichen vergrößert, dann auf 4000, 8000, 16000 und schließlich auf 32000 Zeichen. Hätten wir die initiale Größe auf 20000 Zeichen gesetzt, wäre sogar überhaupt kein Kopiervorgang erforderlich geworden und das Programm hätte 12000 Zeichen weniger alloziert.

Bei der Verwendung der Operatoren `+` und `+=` auf [String](#)-Objekten sollte man zusätzlich bedenken, daß deren Laufzeit nicht konstant ist (bzw. ausschließlich von der Länge des anzuhängenden Strings abhängt). Tatsächlich hängt sie auch stark von der Länge des Strings ab, an den angehängt werden soll, denn die Laufzeit eines Kopiervorgangs wächst nun einmal proportional zur Länge des zu kopierenden Objekts. Damit wächst das Laufzeitverhalten der Schleife in [Listing 29.1](#) nicht linear, sondern annähernd quadratisch. Es verschlechtert sich also mit zunehmender Länge der Schleife überproportional.

Warnung

Einfügen und Löschen in Strings

Ein immer noch deutlicher, wenn auch nicht ganz so drastischer Vorteil bei der Verwendung von [StringBuffer](#) ergibt sich beim Einfügen von Zeichen *am vorderen Ende* des Strings:

```

001 String s;
002 s = "";
003 for (int i = 0; i < 10000; ++i) {
004     s = "x" + s;
005 }

```

Listing 29.4: Langsames Einfügen in einen String

In diesem Beispiel wird wiederholt ein Zeichen vorne in den String eingefügt. Der Compiler wandelt das Programm auch hier in wiederholte Aufrufe von [StringBuffer](#)-Methoden um, wobei unnötig viele Zwischenobjekte entstehen und unnötig oft kopiert werden muß. Eine bessere Lösung kann man auch hier durch direkte Verwendung eines [StringBuffer](#)-Objekts erzielen:

```

001 String s;
002 StringBuffer sb = new StringBuffer(1000);
003 for (int i = 0; i < 10000; ++i) {
004     sb.insert(0, "x");
005 }
006 s = sb.toString();

```

Listing 29.5: Schnelles Einfügen in einen String

Im Test war die Laufzeit dieser Variante etwa um den Faktor vier besser als die der ersten Version. Außerdem wird nicht ein einziges temporäres Objekt erzeugt, so daß zusätzlich das Memory-Subsystem und der Garbage Collector entlastet werden.

Seit JDK 1.2 gibt es in der Klasse [StringBuffer](#) eine Methode [delete](#), mit der ein Teil der Zeichenkette gelöscht werden kann. Dadurch können beispielsweise Programmteile der folgenden Art beschleunigt werden:

```
String sub1 = s.substring(0, 1000) + s.substring(2000);
```

Anstatt hier die ersten 1000 Zeichen mit allen Zeichen ab Position 2000 zu verbinden, kann unter Verwendung eines [StringBuffers](#) auch direkt das gewünschte Stück gelöscht werden:

```
String sub2 = sb.delete(1000, 2000).toString();
```

Die Methode toString der Klasse StringBuffer

Den vorangegangenen Abschnitten kann man entnehmen, daß die Verwendung der Klasse [StringBuffer](#) meist dann sinnvoll ist, wenn die Zeichenkette zunächst aus vielen kleinen Teilen *aufgebaut* werden soll oder wenn sie sich häufig ändert. Ist der String dagegen fertig konstruiert oder muß auf einen vorhandenen String lesend zugegriffen werden, geht dies im allgemeinen mit den vielseitigeren Methoden der Klasse [String](#) besser. Um einen [String](#) in einen [StringBuffer](#) zu konvertieren, wird die Methode [toString](#) aufgerufen, die durch einen kleinen Trick sehr effizient arbeitet. Anstatt beim Aufruf von [toString](#) einen Kopiervorgang zu starten, teilen sich [String](#)- und [StringBuffer](#)-Objekt nach dem Aufruf das interne Zeichenarray, d.h. beide Objekte verwenden ein- und denselben Puffer. Normalerweise wäre diese Vorgehensweise indiskutabel, denn nach der nächsten Änderung des [StringBuffer](#)-Objekts hätte sich dann auch der Inhalt des [String](#)-Objekts verändert (was per Definition nicht erlaubt ist).

Um das zu verhindern, wird vom Konstruktor der [String](#)-Klasse während des Aufrufs von [toString](#) ein *shared*-Flag im [StringBuffer](#)-Objekt gesetzt. Dieses wird bei allen *verändernden* [StringBuffer](#)-Methoden abgefragt und führt dazu, daß - wenn es gesetzt ist - der Pufferinhalt vor der Veränderung kopiert und die Änderung auf der Kopie vorgenommen wird. Ein echter Kopiervorgang wird also solange nicht erforderlich, wie auf den [StringBuffer](#) nicht schreibend zugegriffen wird.

Die Unveränderlichkeit von String-Objekten

Da die Klasse [String](#) keine Möglichkeit bietet, die gespeicherte Zeichenkette nach der Instanzierung des Objekts zu verändern, können einige Operationen auf Zeichenketten sehr effizient implementiert werden. So erfordert beispielsweise die einfache Zuweisung zweier [String](#)-Objekte lediglich das Kopieren eines Zeigers, ohne daß durch *Aliasing* die Gefahr besteht, beim Ändern eines Strings versehentlich weitere Objekte zu ändern, die auf denselben Speicherbereich zeigen.

Soll ein [String](#) physikalisch kopiert werden, kann das mit Hilfe eines speziellen Konstruktors erreicht werden:

```
String s2 = new String(s1);
```

Da der interne Puffer hierbei kopiert wird, ist der Aufruf natürlich ineffizienter als die einfache Zuweisung.

Auch die Methode [substring](#) der Klasse [String](#) konnte sehr effizient implementiert werden. Sie erzeugt zwar ein neues [String](#)-Objekt, aber den internen Zeichenpuffer teilt es sich mit dem bisherigen Objekt. Lediglich die Membervariablen, in denen die Startposition und relevante Länge des Puffers festgehalten werden, müssen im neuen Objekt angepaßt werden. Dadurch ist auch das Extrahieren von langen Teilzeichenketten recht performant. Dasselbe gilt für die Methode [trim](#), die ebenfalls [substring](#) verwendet und daher keine Zeichen kopieren muß.

Durchlaufen von Zeichenketten

Soll ein [String](#) durchlaufen werden, so kann mit der Methode [length](#) seine Länge ermittelt werden, und durch wiederholten Aufruf von [charAt](#) können alle Zeichen nacheinander abgeholt werden. Alternativ könnte man auch zunächst ein Zeichenarray allozieren und durch Aufruf von [getChars](#) alle Zeichen hineinkopieren. Beim späteren Durchlaufen wäre dann kein Methodenaufruf mehr erforderlich, sondern die einzelnen Array-Elemente könnten direkt verwendet werden. Die Laufzeitunterschiede zwischen beiden Varianten sind allerdings minimal und werden in der Praxis kaum ins Gewicht fallen (da die Klasse [String](#) als *final* deklariert wurde und die Methode [charAt](#) nicht *synchronized* ist, kann sie sehr performant aufgerufen werden).

29.2.2 Methodenaufrufe

Eine der häufigsten Operationen in objektorientierten Programmiersprachen ist der Aufruf einer Methode an einer Klasse oder einem Objekt. Generell werden Methodenaufrufe in Java recht performant ausgeführt. Ihr Laufzeitverhalten ist jedoch stark von ihrer Signatur und ihren Attributen abhängig. [Tabelle 29.1](#) gibt einen Überblick über die Laufzeit (in msec.) von 10 Millionen Aufrufen einer trivialen Methode unter unterschiedlichen Bedingungen. Alle Messungen wurden mit dem JDK 1.2 Beta 4 auf einem PentiumII-266 unter Windows 95 vorgenommen.

Signatur/Attribute	Ohne JIT	Mit JIT
public	5650	280
public, mit 4 Parametern	7800	390

public static	5060	110
protected	5770	280
private	5820	50
public synchronized	9500	4660
public final	6260	50

Tabelle 29.1: Geschwindigkeit von Methodenaufrufen

Dabei fallen einige Dinge auf:

- In jedem Fall bringt der JIT einen erheblichen Geschwindigkeitsvorteil. Er liegt (mit Ausnahme der [synchronized](#)-Methode) in diesem Beispiel durchweg bei über einer Zehnerpotenz.
- Methoden des Typs [final](#) und [private](#) werden am schnellsten ausgeführt, insbesondere bei aktiviertem JIT.
- Klassenmethoden werden schneller ausgeführt als Instanzmethoden.
- Die Übergabe von Parametern erfordert zusätzliche Zeit. In unserem Beispiel wurden vier Argumente ([int](#), [String](#), [double](#) und [boolean](#)) übergeben.
- Mit Abstand am langsamsten ist der Aufruf von Methoden, die das [synchronized](#)-Attribut verwenden, denn der Zugriff auf die Sperre zur Synchronisation in Multi-Threading-Umgebungen kostet erhebliche Zeit. Auch der Just-In-Time-Compiler bringt hier keine nennenswerten Vorteile.

Weiterhin ist zu beachten, daß der polymorphe Aufruf von Methoden Zeit kostet (was nicht aus dieser Tabelle abgelesen werden kann). Ist beispielsweise aus einer Klasse *A* eine weitere Klasse *B* abgeleitet, so ist der Aufruf von Methoden auf einem Objekt des Typs *A* kostspieliger als der auf einem Objekt des Typs *B*.

Aus diesen Ergebnissen allgemeingültige Empfehlungen abzuleiten, ist schwierig. Zwar empfiehlt es sich offensichtlich, Methoden als [private](#) bzw. [final](#) zu deklarieren, wenn sicher ist, daß sie in abgeleiteten Klassen nicht aufgerufen bzw. überlagert werden sollen. Auch könnte man versuchen, verstärkt Klassenmethoden zu verwenden oder zur Vermeidung von polymorphen Aufrufen die Vererbungshierarchie zu beschränken oder mit Hilfe des Attributs [final](#) ganz abzuschneiden. All diese Entscheidungen hätten aber einen starken Einfluß auf das Klassendesign der Anwendung und könnten sich leicht an anderer Stelle als Sackgasse herausstellen.

Der einzig wirklich allgemeingültige Rat besteht darin, Methoden nur dann als [synchronized](#) zu deklarieren, wenn es wirklich erforderlich ist. Eine Methode, die keine Membervariablen verwendet, die gleichzeitig von anderen Threads manipuliert werden, braucht auch nicht synchronisiert zu werden. Eine Anwendung, die nur einen einzigen Thread besitzt und deren Methoden nicht von Hintergrundthreads aufgerufen werden, braucht überhaupt keine synchronisierten Methoden in eigenen Klassen.

29.2.3 Vektoren und Listen

Ein [Vector](#) ist ein bequemes Werkzeug, um Listen von Objekten zu speichern, auf die sowohl sequentiell als auch wahlfrei zugegriffen werden kann. Aufgrund seiner einfachen Anwendung und seiner Flexibilität bezüglich der Art und Menge der zu speichernden Elemente wird er in vielen Programmen ausgiebig verwendet. Bei falschem Einsatz können Vektoren durchaus zum Performance-Killer werden, und wir wollen daher einige Hinweise zu ihrer Verwendung geben.

Zunächst einmal ist der Datenpuffer eines Vektors als Array implementiert. Da die Größe von Arrays nach ihrer Initialisierung nicht mehr verändert werden kann, erfordert das Einfügen neuer Elemente möglicherweise das Allokieren eines neuen Puffers und das Umkopieren der vorhandenen Elemente. Ein [Vector](#) besitzt dazu die beiden Attribute *Kapazität* und *Ladefaktor*. Die Kapazität gibt an, wie viele Elemente insgesamt aufgenommen werden können, also wie groß der interne Puffer ist. Der Ladefaktor bestimmt, um wie viele Elemente der interne Puffer erweitert wird, wenn beim Einfügen eines neuen Elements nicht mehr ausreichend Platz vorhanden ist. Je kleiner die anfängliche Kapazität und der Ladefaktor sind, desto häufiger ist beim fortgesetzten Einfügen von Elementen ein zeitaufwendiges Umkopieren erforderlich.

Wird ein [Vector](#) ohne Argumente instanziiert, so hat sein Puffer eine anfängliche Kapazität von 10 Objekten und der Ladefaktor ist 0. Letzteres bedeutet, daß die Kapazität bei jeder Erweiterung *verdoppelt* wird (analog zur Klasse [StringBuffer](#), s. [Abschnitt 29.2.1](#)). Alternativ kann die Kapazität oder auch beide Werte beim Instanzieren an den Konstruktor übergeben werden. Durch die folgende Deklaration wird ein [Vector](#) mit einer anfänglichen Kapazität von 100 Elementen und einem Ladefaktor von 50 angelegt:

```
Vector v = new Vector(100, 50);
```

Ein weiteres Problem der Klasse [Vector](#) ist, daß die meisten ihrer Methoden als [synchronized](#) deklariert wurden. Dadurch kann ein [Vector](#) zwar sehr einfach als gemeinsame Datenstruktur mehrerer Threads verwendet werden. Die Zugriffsmethoden sind aber leider auch ohne Multi-Threading-Betrieb entsprechend langsam.

Seit der Version 1.2 des JDK stehen mit den Klassen [LinkedList](#) und [ArrayList](#) auch alternative Listenimplementierungen zur Verfügung, die anstelle von [Vector](#) verwendet werden können. Hier ist jedoch Vorsicht geboten, soll das Programm nicht langsamer laufen als vorher. Die Klasse [LinkedList](#) implementiert die Datenstruktur in klassischer Form als doppelt verkettete Liste ihrer Elemente. Zwar entfallen dadurch die Kopiervorgänge, die beim Erweitern des Arrays erforderlich waren. Durch die Vielzahl der allozierten Objekte, in denen die Listenelemente und die Zeiger gespeichert werden müssen, und die teilweise ineffiziente Implementierung einiger Grundoperationen (insbesondere [add](#)) hat sich [LinkedList](#) jedoch im Test als relativ ineffizient herausgestellt. Wesentlich bessere Ergebnisse gab es mit der Klasse [ArrayList](#). Sie ist ähnlich wie [Vector](#) implementiert, verzichtet aber (wie die meisten 1.2er Collections) auf die [synchronized](#)-Attribute und ist daher - insbesondere bei aktiviertem JIT und Zugriff mit [add](#) und [get](#) sehr - performant.

[Listing 29.6](#) zeigt drei Methoden, die jeweils ein String-Array übergeben bekommen und daraus eine bestimmte Anzahl von Elementen zurückgeben. Die erste Version verwendet einen [Vector](#), die zweite eine [LinkedList](#) und die dritte eine [ArrayList](#) zur Datenspeicherung. Im Test war die dritte Version eindeutig die schnellste. Bei aktiviertem JIT und Übergabe von 100000 Elementen, von denen jeweils die Hälfte zurückgegeben wurden, war das Verhältnis der Laufzeiten der drei Methoden etwa 3:18:1.

```

001 public static String[] vtest1(String el[], int retsize)
002 {
003     //Verwendet Vector
004     Vector v = new Vector(el.length + 10);
005     for (int i = 0; i < el.length; ++i) {
006         v.addElement(el[i]);
007     }
008     String ret[] = new String[retsize];
009     for (int i = 0; i < retsize; ++i) {
010         ret[i] = (String)v.elementAt(i);
011     }
012     return ret;
013 }
014
015 public static String[] vtest2(String el[], int retsize)
016 {
017     //Verwendet LinkedList
018     LinkedList l = new LinkedList();
019     for (int i = 0; i < el.length; ++i) {
020         l.add(el[i]);
021     }
022     String ret[] = new String[retsize];
023     Iterator it = l.iterator();
024     for (int i = 0; i < retsize; ++i) {
025         ret[i] = (String)it.next();
026     }
027     return ret;
028 }
029
030 public static String[] vtest3(String el[], int retsize)
031 {
032     //Verwendet ArrayList
033     ArrayList l = new ArrayList(el.length + 10);
034     for (int i = 0; i < el.length; ++i) {
035         l.add(el[i]);
036     }
037     String ret[] = new String[retsize];
038     for (int i = 0; i < retsize; ++i) {
039         ret[i] = (String)l.get(i);
040     }
041     return ret;
042 }

```

Listing 29.6: Vergleich von Listen und Vektoren

Ist es dagegen erforderlich, viele Einfügungen und Löschungen innerhalb der Liste vorzunehmen, sollte im allgemeinen eine zeigerbasierte

Implementierung der arraybasierten vorgezogen werden. Während es bei letzterer stets erforderlich ist, einen Teil des Arrays umzukopieren, wenn ein Element eingefügt oder gelöscht wird, brauchen bei den verzeigerten Datenstrukturen lediglich ein paar Verweise aktualisiert zu werden.

29.2.4 Dateizugriffe

Schreiben von Streams

Seit dem JDK 1.1 gibt es die [Writer](#)-Klassen, mit denen *Character-Streams* verarbeitet werden können. Passend zur internen Darstellung des [char](#)-Typs in Java verwenden sie 16-Bit breite UNICODE-Zeichen zur Ein- und Ausgabe. Um eine Datei zu erzeugen, kann ein [FileWriter](#)-Objekt angelegt werden, und die Zeichen werden mit den [write](#)-Methoden geschrieben. Um die Performance zu erhöhen, kann der [FileWriter](#) in einen [BufferedWriter](#) gekapselt werden, der mit Hilfe eines internen Zeichenpuffers die Anzahl der Schreibzugriffe reduziert. Im Test ergab sich dadurch ein Geschwindigkeitszuwachs um den Faktor drei bis vier gegenüber dem ungepufferten Zugriff. Die von [BufferedWriter](#) verwendete Standard-Puffergröße von 8 kByte ist in aller Regel ausreichend, weitere Vergrößerungen bringen keine nennenswerten Beschleunigungen.

Das Dilemma der [Writer](#)-Klassen besteht darin, daß die meisten externen Dateien mit 8-Bit-Zeichen arbeiten, statt mit 16-Bit-UNICODE-Zeichen. Ein [FileWriter](#) führt also vor der Ausgabe eine Konvertierung der UNICODE-Zeichen durch, um sie im korrekten Format abzuspeichern. Der Aufruf der dazu verwendeten Methoden der Klasse [CharToByteConverter](#) aus dem Paket [sun.io](#) kostet natürlich Zeit und vermindert die Performance der [Writer](#)-Klasse. Wesentlich schneller sind die (älteren) [OutputStream](#)-Klassen, die nicht mit Zeichen, sondern mit Bytes arbeiten. Sie führen keine aufwendige Konvertierung durch, sondern geben je Zeichen einfach dessen niederwertige 8 Bits aus. Das spart viel Zeit und führte im Test zu einer nochmals um den Faktor drei bis vier beschleunigten Ausgabe (wenn auch der [FileOutputStream](#) in einen [BufferedOutputStream](#) eingeschlossen wurde).

Die [OutputStream](#)-Klassen sind also immer dann den [Writer](#)-Klassen vorzuziehen, wenn entweder sowieso Binärdaten ausgegeben werden sollen oder wenn sichergestellt ist, daß keine UNICODE-Zeichen verwendet werden, die durch das simple Abschneiden der oberen 8 Bit falsch ausgegeben würden. Da der UNICODE-Zeichensatz in den ersten 256 Zeichen zum ISO-8859-1-Zeichensatz kompatibel ist, sollten sich für die meisten europäischen und angelsächsischen Sprachen keine Probleme ergeben, wenn zur Ausgabe von Zeichen die [OutputStream](#)-Klassen verwendet werden.

[Listing 29.7](#) zeigt das Erzeugen einer etwa 300 kByte großen Datei, bei der zunächst die [Writer](#)- und dann die [OutputStream](#)-Klassen verwendet wurden. Im Test lag die Ausführungsgeschwindigkeit der zweiten Variante um etwa eine Zehnerpotenz über der ersten.

Beispiel

```
001 public static void createfile1()
002 throws IOException
003 {
004     Writer writer = new FileWriter(FILENAME);
005     for (int i = 0; i < LINES; ++i) {
006         for (int j = 0; j < 60; ++j) {
007             writer.write('x');
008         }
009         writer.write(NL);
010     }
011     writer.close();
012 }
013
014 public static void createfile4()
015 throws IOException
016 {
017     OutputStream os = new BufferedOutputStream(
018         new FileOutputStream(FILENAME)
019     );
020     for (int i = 0; i < LINES; ++i) {
021         for (int j = 0; j < 60; ++j) {
022             os.write('x');
023         }
024         os.write('\r');
025         os.write('\n');
026     }
027     os.close();
028 }
```

Listing 29.7: Performance von Writer und OutputStream

Lesen von Streams

Die Performance des sequentiellen Lesens von Zeichen- oder Byte-Streams zeigt ein ähnliches Verhalten wie die des sequentiellen Schreibens. Am langsamsten war der ungepufferte Zugriff mit der Klasse [FileReader](#). Die größten Geschwindigkeitsgewinne ergaben sich durch das Kapseln des [FileReader](#) in einen [BufferedReader](#), die Performance lag um etwa eine Zehnerpotenz höher als im ungepufferten Fall. Der Umstieg auf das byte-orientierte Einlesen mit den Klassen [FileInputStream](#) und [BufferedInputStream](#) brachte dagegen nur noch geringe Vorteile. Möglicherweise muß der zur Eingabekonvertierung in den [Reader](#)-Klassen verwendete [ByteToCharConverter](#) weniger Aufwand treiben als ausgabeseitig nötig war.

RandomAccess-Dateien

Der wahlfreie Zugriff auf eine Datei zum Lesen oder Schreiben erfolgt in Java mit der Klasse [RandomAccessFile](#). Da sie nicht Bestandteil der [Reader](#)- [Writer](#)-, [InputStream](#)- oder [OutputStream](#)-Hierarchien ist, besteht auch nicht die Möglichkeit, sie zum Zweck der Pufferung zu schachteln. Tatsächlich ist der ungepufferte byteweise Zugriff auf ein [RandomAccessFile](#) sehr langsam, er liegt etwa in der Größenordnung des ungepufferten Zugriffs auf Character-Streams. Wesentlich schneller kann mit Hilfe der [read](#)- und [write](#)-Methoden gearbeitet werden, wenn nicht nur ein einzelnes, sondern ein ganzes Array von Bytes verarbeitet wird. Je nach Puffergröße und Verarbeitungsaufwand werden dann Geschwindigkeiten wie bei gepufferten Bytestreams oder höher erzielt. Das folgende Beispiel zeigt, wie man mit einem 100 Byte großen Puffer eine Random-Access-Datei bereits sehr schnell lesen kann.

```
001 public static void randomtest2()  
002 throws IOException  
003 {  
004     RandomAccessFile file = new RandomAccessFile(FILENAME, "rw");  
005     int cnt = 0;  
006     byte buf[] = new byte[100];  
007     while (true) {  
008         int num = file.read(buf);  
009         if (num <= 0) {  
010             break;  
011         }  
012         cnt += num;  
013     }  
014     System.out.println(cnt + " Bytes read");  
015     file.close();  
016 }
```

Listing 29.8: Gepufferter Zugriff auf Random-Access-Dateien

Das Programm liest die komplette Datei in Stücken von jeweils 100 Bytes ein. Der Rückgabewert von [read](#) gibt die tatsächliche Zahl gelesener Bytes an. Sie entspricht normalerweise der Puffergröße, liegt aber beim letzten Datenpaket darunter, wenn die Dateigröße nicht zufällig ein Vielfaches der Puffergröße ist. Die Performance von [randomtest2](#) ist sehr gut, sie lag auf dem Testrechner (Pentium II, 266 MHz, 128 MB, UW-SCSI) bei etwa 5 MByte pro Sekunde, was für ein Java-Programm sicherlich ein respektabler Wert ist. Ein wesentlicher Grund ist darin zu suchen, daß durch den programmeigenen Puffer ein Großteil der Methodenaufrufe zum Lesen einzelner Bytes vermieden werden (in diesem Fall sind es um den Faktor 100 weniger). Auf die gleiche Weise lassen sich auch die streamorientierten Dateizugriffe beschleunigen, wenn die Anwendung nicht unbedingt darauf angewiesen ist, *zeichenweise* zu lesen bzw. zu schreiben.

29.3 Einsatz eines Profilers

- [29.3 Einsatz eines Profilers](#)
 - [29.3.1 Erzeugen der Profiling-Informationen](#)
 - [29.3.2 Auswerten der Profiling-Informationen](#)
 - [29.3.3 Ausblick](#)

29.3.1 Erzeugen der Profiling-Informationen

Die bisher vorgestellten Tips und Tricks sind sicherlich eine Hilfe, um bereits während der Entwicklung eines Programms grundsätzliche Performance-Probleme zu vermeiden. Läuft das fertige Programm dann trotzdem nicht mit der gewünschten Geschwindigkeit (was in der Praxis häufig vorkommt), helfen pauschale Hinweise leider nicht weiter. Statt dessen gilt es herausfinden, welche Teile des Programms für dessen schlechte Performance verantwortlich sind. Bei größeren Programmen, die aus vielen tausend Zeilen Quellcode bestehen, ist das eine komplizierte Aufgabe, die nur mit Hilfe eines guten *Profilers* bewältigt werden kann. Der Profiler ist ein Werkzeug, mit dessen Hilfe im laufenden Programm *Performanceparameter*, wie beispielsweise die verbrauchte CPU-Zeit, die Anzahl der allozierten Objekte oder die Anzahl der Aufrufe bestimmter Funktionen, überwacht und gemessen werden können. Mit Hilfe eines Auswertungswerkzeugs kann dann festgestellt werden, welche Teile des Programms die größte Last erzeugt haben und daher verbesserungsbedürftig sind.

Das Standard-JDK enthält bereits seit der Version 1.0 einen eingebauten Profiler, der Informationen über Laufzeit und Aufrufhäufigkeit von Funktionen geben kann. Im JDK 1.2 wurde er erweitert und ist nun zusätzlich in der Lage, den Speicherverbrauch zu messen und Profiling-Informationen *threadweise* auszugeben. Im Vergleich zu spezialisierten Produkten sind seine Fähigkeiten etwas rudimentär. So ist es beispielsweise nicht möglich, *unterhalb* von Funktionen (etwa auf der Ebene einzelner Quelltextzeilen) Laufzeiten zu messen, und eine dynamische Analyse während des laufenden Programms wird nicht unterstützt. Zudem erfordert die vom Profiler erzeugte Ausgabedatei einigen Nachbearbeitungsaufwand, um zu aussagekräftigen Ergebnissen zu kommen. Dennoch ist der JDK-Profiler ein wichtiges und hilfreiches Instrument, und wir wollen uns in diesem Abschnitt mit seiner Bedienung vertraut machen.

Als Beispiel soll das folgende Programm verwendet werden. Seine Aufgabe besteht darin, eine vorgegebene Menge von Fließkommazufallszahlen zu erzeugen und sortiert auf dem Bildschirm auszugeben. Wir unterteilen die Aufgabe in die drei Funktionen *Erzeugen der Daten*, *Sortieren der Daten* und *Ausgeben der Daten*. Das Programm verwendet zum Anlegen der Daten einen [Vector](#) mit einem Ladefaktor von 1 und zum Sortieren den *Bubblesort*-Algorithmus. Dessen Laufzeit ist quadratisch und er ist so langsam, daß das Programm bereits ab einigen hundert Elementen praktisch unbrauchbar wird:

[ProfTest.java](#)

```

001  /* ProfTest.java */
002
003  import java.util.*;
004
005  public class ProfTest
006  {
007      static final int DATASIZE = 500;
008      static Vector data;
009
010      public static void createData()
011      {
012          System.out.println("Erzeugen der Daten");
013          Random rand = new Random();
014          data = new Vector(0, 1);
015          for (int i = 0; i < DATASIZE; ++i) {
016              data.addElement(new Double(rand.nextDouble()));
017          }
018      }
019
020      public static void sortData()
021      {
022          System.out.println("Sortieren der Daten");
023          boolean sorted;
024          do {
025              sorted = true;
026              for (int i = 0; i < DATASIZE - 1; ++i) {
027                  Double x1 = (Double)data.elementAt(i);

```



```

028         Double x2 = (Double)data.elementAt(i + 1);
029         if (x1.doubleValue() > x2.doubleValue()) {
030             data.setElementAt(x2, i);
031             data.setElementAt(x1, i + 1);
032             sorted = false;
033         }
034     }
035 } while (!sorted);
036 }
037
038 public static void printData()
039 {
040     Enumeration e = data.elements();
041     while (e.hasMoreElements()) {
042         Double x = (Double)e.nextElement();
043         System.out.println(x.doubleValue());
044     }
045 }
046
047 public static void main(String args[])
048 {
049     createData();
050     sortData();
051     printData();
052 }
053 }

```

Listing 29.9: Test-Programm für den Profiler

Um Profiling-Informationen zu diesem Programm zu erzeugen, ist zunächst der *Just-In-Time-Compiler* zu deaktivieren, denn sonst würden die Ergebnisse stark verfälscht. Ab dem JDK1.2 Beta 4 setzt man dazu die Umgebungsvariable [JAVA_COMPILER](#) auf den Wert **NONE**, denn standardmäßig ist der JIT aktiviert:

```
set JAVA_COMPILER=NONE
```

Nun wird das Programm im Java-Interpreter mit der Option **-Xprof** gestartet (das **X** muß groß geschrieben werden):

```
java -Xprof ProfTest
```

In den Prä-1.2-JDKs wurde der Profiler nicht mit der Option **-Xprof**, sondern mit **-prof** aufgerufen. Zudem durfte nicht der normale Interpreter verwendet werden, sondern mit **java_g** mußte dessen Debug-Version aufgerufen werden. Soll der Profiler bei einem Applet verwendet werden, kann die Aufrufoption mit dem Schalter **-J** an den Appletviewer übergeben werden.

Leider hat sich der Profiler *nach* dem Beta 4 des JDK 1.2 noch einmal verändert! Er wird nun nicht mehr mit den zuvor beschriebenen Optionen aufgerufen, sondern mit dem Schalter **-Xrunhprof**, der eine Vielzahl von Unteroptionen kennt (**-Xrunhprof:help** listet sie zusammen mit jeweils einem erläuternden Text auf). Auch das Ausgabeformat hat sich stark verändert und enthält zudem den Warnhinweis »WARNING! This file format is under development, and is subject to change without notice«. Zu allem Überfluß konnten einige der wichtigeren Unteroptionen bis zum Redaktionsschluß nicht zum Laufen gebracht werden, sondern führten regelmäßig zum Absturz der Testrechner. Der Leser sollte die nachfolgenden Ausführungen daher lediglich als *methodische Erläuterung* der Java-Profiling-Konzepte ansehen, je nach JDK-Version aber unter Umständen Detailunterschiede in Kauf nehmen. Wir werden versuchen, die Änderungen in der aktualisierten Online-Version nachzureichen, sobald sich der Profiler stabilisiert hat.

Das Programm erzeugt nun den Vektor mit den Fließkommazahlen, sortiert ihn und gibt das Ergebnis auf dem Bildschirm aus. Zusätzlich erzeugt es die Datei **java.prof**, in der nach Programmende die Profiling-Informationen stehen. Die Ausgabe erfolgt im ASCII-Textformat, und die ersten Zeilen der Datei könnten beispielsweise so aussehen:

```

count callee caller time
475048 java/util/Vector.elementAt(I)Ljava/lang/Object; \
      ProfTest.sortData()V 3945
132408 java/util/Vector.setElementAt(Ljava/lang/Object;I)V \
      ProfTest.sortData()V 1098

```

```

2008 sun/io/CharToByteISO8859_1.flush([BII)I \
      java/io/OutputStreamWriter.flushBuffer()V 34
1506 java/io/BufferedOutputStream.flushBuffer()V \
      java/io/BufferedOutputStream.flush()V 550
1506 java/io/OutputStream.flush()V \
      java/io/BufferedOutputStream.flush()V 19
1500 java/lang/Math.max(II)I \
      java/lang/Float.parseFloat(String)V 13
...

```

Da die Ausgabezeilen teilweise sehr lang sind, wurden sie hier manuell umbrochen, was durch einen Backslash am Ende der Zeile und eine Einrückung der folgenden Zeile angezeigt wird. In Wirklichkeit gibt der Profiler die Zeilen zusammenhängend aus.

Hinweis

Die erste Zeile gibt die Spaltenbelegung an. In diesem Beispiel hat jede Zeile vier Spalten, die nacheinander die *Aufrufhäufigkeit*, die *aufgerufene Methode*, den *Aufrufer* und den *Rechenzeitverbrauch* angeben. Die Sortierung der Ausgabe erfolgt nach der Aufrufhäufigkeit. So wurde beispielsweise die Methode `elementAt` der Klasse `java.util.Vector` insgesamt 475048mal von unserer eigenen Methode `sortData` aufgerufen und hat dabei 3945 Einheiten (Millisekunden) Rechenzeit verbraucht. Dagegen wurde die parameterlose Methode `flushBuffer` aus `java.io.BufferedOutputStream` 1506mal von der ebenfalls parameterlosen Methode `flush` der Klasse `BufferedOutputStream` aufgerufen und hat dabei 550 Einheiten Rechenzeit verbraucht.

Der Schalter `-Xprof` kennt die Parameter `file`, `type` und `format`. Mit `file` kann eine alternative Ausgabedatei bestimmt werden. `type` kann entweder `inst` oder `cpu` sein und entscheidet, ob in der vierten Spalte die Laufzeit oder der Speicherverbrauch angegeben wird. Der `format`-Parameter legt fest, ob die Ausgabe im alten JDK-1.1-Format erfolgt oder ob die Ergebnisse in einer fünften Spalte mit zusätzlichen Thread-Informationen versehen werden sollen. Um den Profiler also beispielsweise den Speicherverbrauch messen zu lassen und seine Ausgabe threadweise zu erzeugen, hätten wir folgendes Kommando verwendet:

```
java -Xprof:type=inst,format=perthread ProfTest
```

Die Ausgabedatei `java.prof` würde dann so aussehen (wieder mit eingefügten Umbrüchen zur besseren Lesbarkeit):

```

thread name
0 "Thread-0"
1 "Finalizer"
2 "Reference Handler"
3 "main"
count callee caller weight thread
1 java/lang/String.length()I \
  java/lang/StringBuffer.<init>(Ljava/lang/String;)V 3 0
1 java/security/AccessController.getInheritedAccessControlContext()\
  Ljava/security/AccessControlContext; \
  java/security/AccessControlContext.optimize()\
  Ljava/security/AccessControlContext; -1 0
1 java/lang/StringBuffer.<init>(Ljava/lang/String;)V \
  java/lang/Thread.<init>(Ljava/lang/ThreadGroup;\
  Ljava/lang/Runnable;)V 11 0
...

```

Hier erfolgt die Ausgabe nach Threads sortiert. Zunächst werden die Aufrufe in Thread 0 angezeigt, gefolgt von denen in Thread 1 usw.

29.3.2 Auswerten der Profiling-Informationen

Wenn man sich die Datei `java.prof` genauer ansieht, wird man feststellen, daß sie etwa 1000 Zeilen mit Methodenaufrufen enthält und darüber hinaus am Ende noch weitere Informationen anzeigt. Tatsächlich taucht scheinbar jede Methode darin auf, die mindestens einmal im Programm aufgerufen wurde. Das Hauptproblem bei der Auswertung der Datei besteht also darin, aus dieser Datenmenge die für die Optimierung des Programms relevanten Informationen herauszufinden.

Für noch nicht optimierte Programme gilt oft die 80/20-Regel, die besagt, daß 80 Prozent der Rechenzeit in lediglich 20 Prozent des Programmcodes verbraucht werden. Um zu aussagekräftigen Ergebnissen zu kommen, müßte der Profiler-Output also nach der vierten Spalte sortiert werden, denn sie gibt den Rechenzeitverbrauch bzw. die Anzahl der erzeugten Objektinstanzen an. Da dieser Wert nicht notwendigerweise mit der Anzahl der Methodenaufrufe korrelieren muß, ist die ursprüngliche Sortierung der Datei in aller Regel wenig hilfreich.

Das folgende Programm `ProfSort`, kann dazu verwendet werden, die Datei `java.prof` nach der vierten Spalte absteigend zu sortieren. Zusätzlich kann es die Ausgabe auf eine vorgegebene Anzahl von Methodenaufrufen beschränken. Zum Sortieren wird die Klasse `SortableVector` aus dem Paket `gk.util` verwendet (auf der CD-ROM im Verzeichnis `\misc` zu finden). Alternativ kann auch eine der sortierbaren Collections verwendet werden, wie sie in [Abschnitt 27.7](#) erläutert wurden.

```

001 /* ProfSort.java */
002
003 import java.io.*;
004 import java.util.*;
005 import gk.util.*;
006
007 /**
008  * Klasse zum Sortieren des JDK-Profiler-Outputs nach
009  * verbrauchter Rechenzeit. Kann sowohl mit dem alten als
010  * auch mit dem neuen Format umgehen (Optionen -Xprof und
011  * -Xprof:format=perthread).
012  */
013 public class ProfSort
014 {
015     InputStream in;
016     PrintStream out;
017     String header[];
018     SortableVector data;
019     int colcnt;
020
021     /**
022      * Instanziert ein neues ProfSort-Objekt.
023      */
024     public ProfSort(InputStream in, PrintStream out)
025     {
026         this.in = in;
027         this.out = out;
028     }
029
030     /**
031      * Einlesen des Headers und der Datenzeilen.
032      */
033     public void readData()
034     throws IOException
035     {
036         data = new SortableVector(1500);
037         BufferedReader reader = new BufferedReader(
038             new InputStreamReader(in));
039         String line;
040         //Suchen und Lesen der Kopfzeile
041         while ((line = reader.readLine()) != null) {
042             if (line.startsWith("count")) {
043                 StringTokenizer st = new StringTokenizer(line);
044                 colcnt = st.countTokens();
045                 if (colcnt == 4 || colcnt == 5) {
046                     header = new String[colcnt];
047                     for (int i = 0; i < colcnt; ++i) {
048                         header[i] = st.nextToken();
049                     }
050                 }
051                 break;
052             }
053         }
054         if (header.length <= 0) {
055             System.err.println("Unbekanntes Profiler-Format");
056             System.exit(1);
057         }
058         //Einlesen der Datenzeilen
059         while ((line = reader.readLine()) != null) {
060             // "<unknown caller>" verwirrt den Tokenizer
061             int pos = line.indexOf("<unknown caller>");
062             if (pos != -1) {
063                 line = line.substring(0, pos) + "<UnknownCaller>" +
064                     line.substring(pos + 16);

```

```

065     }
066     StringTokenizer st = new StringTokenizer(line);
067     if (st.countTokens() != colcnt) {
068         break;
069     }
070     ProfilerEntry entry = new ProfilerEntry();
071     entry.count = Integer.parseInt(st.nextToken());
072     entry.callee = st.nextToken();
073     entry.caller = st.nextToken();
074     entry.time = Integer.parseInt(st.nextToken());
075     if (colcnt >= 5) {
076         entry.thread = Integer.parseInt(st.nextToken());
077     }
078     data.addElement(entry);
079 }
080 }
081
082 /**
083  * Ausgeben der count rechenzeitintensivsten Methoden
084  * innerhalb des angegebenen Threads. Falls die
085  * Eingabedatei keine Threadnummern enthält oder -1
086  * in thread übergeben wurde, werden alle Threads
087  * berücksichtigt.
088  */
089 public void printData(int count, int thread)
090 {
091     //Sort data vector
092     data.qsort(new ElementComparator() {
093         public boolean preceeds(Object element1, Object element2)
094         {
095             ProfilerEntry entry1 = (ProfilerEntry)element1;
096             ProfilerEntry entry2 = (ProfilerEntry)element2;
097             return entry1.time > entry2.time;
098         }
099     });
100     //Kopfzeile ausgeben
101     out.print(Str.getFormatted("%6s ", header[3]));
102     out.print(Str.getFormatted("%6s ", header[0]));
103     out.print(header[1] + " " + header[2]);
104     out.println(colcnt >= 5 ? " " + header[4] : "");
105     //Datenzeilen ausgeben
106     int cnt = 0;
107     Enumeration e = data.elements();
108     while (e.hasMoreElements()) {
109         ProfilerEntry entry = (ProfilerEntry)e.nextElement();
110         if (colcnt < 5 || thread == -1 || entry.thread == thread) {
111             if (++cnt > count) {
112                 break;
113             }
114             out.println(entry.toString());
115         }
116     }
117 }
118
119 /**
120  * Aufrufbeispiel:
121  *
122  * type java.prof | java ProfSort 10 3
123  *
124  * Zeigt die 10 rechenzeitintensivsten Methoden des
125  * Threads 3 an. Die Datei java.prof muß vorher mit
126  * java -Xprof:format=perthread erstellt worden sein.
127  */
128 public static void main(String args[])
129 {

```

```

130     if (args.length < 1 || args.length > 2) {
131         System.err.println(
132             "Usage: java ProfSort <count> [<thread>]"
133         );
134         System.exit(0);
135     }
136     int count = Integer.parseInt(args[0]);
137     int thread = -1;
138     if (args.length == 2) {
139         thread = Integer.parseInt(args[1]);
140     }
141     try {
142         ProfSort ps = new ProfSort(System.in, System.out);
143         ps.readData();
144         ps.printData(count, thread);
145     } catch (IOException e) {
146         System.err.println(e.toString());
147         System.exit(1);
148     }
149 }
150 }
151
152 /**
153  * Hilfsklasse zur Speicherung eines Profiler-Eintrags.
154  */
155 class ProfilerEntry
156 {
157     public int    count;
158     public int    time;
159     public String caller;
160     public String callee;
161     public int    thread;
162
163     public ProfilerEntry()
164     {
165         thread = -1;
166     }
167
168     public String toString()
169     {
170         return Str.getFormatted("%6d ", time) +
171             Str.getFormatted("%6d ", count) +
172             callee + " " + caller +
173             (thread != -1 ? " (" + thread + ")" : "");
174     }
175 }

```

Listing 29.10: Programm zum Sortieren der Profiling-Informationen

`ProfSort` kann sowohl Dateien bearbeiten, die im alten Format erzeugt wurden, als auch solche, bei denen die Option `perthread` aktiviert war. Das Programm wird mit einem oder zwei Parametern aufgerufen:

```
java ProfSort <count> [<thread>]
```

count gibt die Anzahl der auszugebenden Methodenaufrufe an, und mit *thread* kann die Ausgabe auf einen bestimmten Thread beschränkt werden, wenn der Profiler mit der Option `format=perthread` aufgerufen wurde. Fehlt das *thread*-Argument oder wurde die Ausgabe ohne Thread-Informationen erzeugt, berücksichtigt das Programm alle Threads. Das Programm liest die Profilerdatei von der Standardeingabe, so daß ein beispielhafter Aufruf so aussehen könnte:

```
type java.prof | java ProfSort 10
```

Angewandt auf das erste Beispiel wäre die Ausgabe von `ProfSort` nun:

```

time  count callee caller
15681      1 ProfTest.main([Ljava/lang/String;)V \
          <UnknownCaller>

```

```

12090      1 ProfTest.sortData()V \
          ProfTest.main([Ljava/lang/String;)V
4332 485028 java/util/Vector.elementAt(I)Ljava/lang/Object; \
          ProfTest.sortData()V
3514      1 ProfTest.printData()V \
          ProfTest.main([Ljava/lang/String;)V
3463     500 java/io/PrintStream.println(D)V \
          ProfTest.printData()V
2969     500 java/io/PrintStream.print(D)V \
          java/io/PrintStream.println(D)V
2615     500 java/io/PrintStream.write(Ljava/lang/String;)V \
          java/io/PrintStream.print(D)V
2496    1004 java/io/PrintStream.write([BII)V \
          java/io/OutputStreamWriter.flushBuffer()V
2375    1004 java/io/BufferedOutputStream.flush()V \
          java/io/PrintStream.write([BII)V
2372     502 java/io/OutputStreamWriter.flushBuffer()V \
          java/io/PrintStream.write(Ljava/lang/String;)V
...

```

Die `main`-Methode des Programms hat also insgesamt 15681 Einheiten Rechenzeit verbraucht. Davon entfielen 12090 auf `sortData` und 3514 auf `printData`. Die dritte Methode `createData` ist nicht unter den Top-10, sie hat deutlich weniger Rechenzeit verbraucht.

Primäres Ziel unserer weiteren Optimierungsversuche wird also der Aufruf von `sortData`. Wir erkennen, daß innerhalb von `sortData` 4332 Einheiten auf den zum Vergleich der Elemente erforderlichen Aufruf von `Vector.elementAt` entfielen. Könnten wir diese Anzahl substantiell verringern, wäre eine deutliche Beschleunigung des Programms zu erwarten. Um weitere Informationen über das Laufzeitverhalten der Methode `sortData` zu erhalten, müssen wir die Datei [java.prof](#) nun nach den Zeilen durchsuchen, bei denen als Aufrufer (caller) unsere Methode `sortData` eingetragen ist. In unserem Beispiel wäre das:

```

4332 485028 java/util/Vector.elementAt(I)Ljava/lang/Object; \
          ProfTest.sortData()V
1091 126224 java/util/Vector.setElementAt(Ljava/lang/Object;I)V \
          ProfTest.sortData()V
      1      1 java/io/PrintStream.println(Ljava/lang/String;)V \
          ProfTest.sortData()V 1
      0      1 java/lang/Double.doubleValue()D \
          ProfTest.sortData()V 0

```

Dabei geht lediglich noch der Aufruf von `setElementAt` mit 1091 Einheiten nennenswert in das Ergebnis ein. Weder der einzelne Aufruf von `println` noch die Aufrufe von `doubleValue` (die möglicherweise *inline* ausgeführt werden und daher hier nicht explizit in Erscheinung treten) spielen für das Ergebnis eine wesentliche Rolle. Summiert man den Rechenzeitverbrauch, so kommt man auf 5424 Einheiten, immerhin noch 6666 Einheiten weniger als die gesamte Rechenzeit von `sortData`. Die verbleibende Zeit wird also nicht in untergeordneten Methodenaufrufen verbraucht, sondern in `sortData` selbst, beispielsweise im Code zur Ausführung der `for`-Schleifen oder zum Vergleichen der Fließkommazahlen.

Schaut man sich die doppelt geschachtelten Schleifen in `sortData` an, so erkennt man, daß der Schlüssel zur Optimierung der Methode im Einsatz eines besseren Sortierverfahrens liegt. Bubblesort hat bei gut gemischten Daten quadratische Laufzeit und benötigt daher fast $500 * 500 = 250000$ Elementvergleiche. Das sind knapp 500000 Zugriffe auf den Vektor, 500000 Zugriffe auf den Wert eines `Double`-Objekts, 250000 Aufrufe des Größer-Tests auf Fließkommazahlen und 250000 Additionen und Tests des Schleifenzählers.

Wenn man statt dessen eines der Sortierverfahren der Klasse $O(n \log_2 n)$ verwenden würde (beispielsweise *QuickSort*, *ShellSort* oder *HeapSort*), könnte man die Laufzeit von `sortData` auf einen Bruchteil seiner jetzigen Laufzeit verringern. Bei 1000 Elementen wäre die Geschwindigkeit dann bereits hundertmal höher, bei 10000 Elementen etwa 750mal. Das folgende Listing zeigt eine alternative Realisierung von `sortData` auf der Basis einer rekursiven Quicksort-Implementierung. Im Test hat dieses Verfahren tatsächlich die beschriebenen Verbesserungen gebracht und dazu geführt, daß `sortData` auch noch 50000 Elemente in etwa einer Sekunde sortiert hat.

[qsort.inc](#)

```

001 /* qsort.inc */
002
003 public static void qsortData(int left, int right)
004 {
005     if (left < right) {
006         if (right - left == 1) {
007             //Zwei Elemente werden direkt vertauscht
008             Double x1 = (Double)data.elementAt(left);
009             Double x2 = (Double)data.elementAt(right);
010             if (x1.doubleValue() > x2.doubleValue()) {
011                 data.setElementAt(x1, right);

```

```

012     data.setElementAt(x2, left);
013 }
014 } else {
015     //Pivot per Zufallszahl auswählen
016     int pos = left + (int)((right - left + 1) * Math.random());
017     double pivot = ((Double)data.elementAt(pos)).doubleValue();
018     int i = left - 1;
019     int j = right + 1;
020     Double x, y;
021     //Partitionieren
022     while (true) {
023         //Linken Rand suchen
024         do {
025             ++i;
026             x = (Double)data.elementAt(i);
027         } while (x.doubleValue() < pivot);
028         //Rechten Rand suchen
029         do {
030             --j;
031             y = (Double)data.elementAt(j);
032         } while (y.doubleValue() > pivot);
033         if (i >= j) {
034             break;
035         }
036         //Randelemente vertauschen
037         data.setElementAt(x, j);
038         data.setElementAt(y, i);
039     }
040     //Rekursiver qsort der Partition[en]
041     if (j == right) {
042         qsortData(left, j - 1);
043     } else {
044         qsortData(left, j);
045         qsortData(j + 1, right);
046     }
047 }
048 }
049 }
050
051 ...
052
053 //Der Aufruf erfolgt nun parametrisiert:
054 createData();
055 qsortData(0, DATASIZE - 1);
056 printData();
057
058 ...

```

Listing 29.11: Sortieren mit QuickSort

Die Laufzeitstruktur der Methode `printData` stellt sich vollkommen anders dar. Sie verbraucht 3463 von 3514 Einheiten (also 98,5 %) in den 500 Aufrufen von `java.io.PrintStream.println`. Verfolgt man [java.prof](#) über die Aufrufer-Kette weiter, erkennt man, daß die Aufrufe von [hasMoreElements](#) und [nextElement](#) mit 9 bzw. 11 Einheiten keine nennenswerte Rolle spielen. Auch die Konvertierung von [double](#) nach [String](#) spielt mit etwa 200 Einheiten keine große Rolle. Der Löwenanteil der Rechenzeit wird tatsächlich in Ausgabemethoden verbraucht.

Meist besteht die einzige Möglichkeit, den Rechenzeitverbrauch von Java-eigenen Methoden zu verringern, darin, sie weniger oft aufzurufen oder den Aufruf durch eine schnellere Alternative zu ersetzen. Beides ist oft problematisch, denn der Aufwand ist hoch und das Programm kann schnell unleserlich werden. Wir könnten in unserem Beispiel etwa auf die Idee kommen, die Ausgabe selbst zu puffern und zum Schluß alle Fließkommazahlen auf einen Schlag auszugeben:


```

001 public static void printData()
002 {
003     String NL = System.getProperty("line.separator");
004     StringBuffer sb = new StringBuffer(10000);
005     Enumeration e = data.elements();
006     while (e.hasMoreElements()) {
007         Double x = (Double)e.nextElement();
008         sb.append(String.valueOf(x.doubleValue()));
009         sb.append(NL);
010     }
011     System.out.print(sb.toString());
012 }

```

Listing 29.12: Eine optimierte Ausgabemethode

Tatsächlich würde das die Laufzeit von 3514 auf 812 Einheiten vermindern, was sicherlich ein guter Erfolg wäre. Zusammen mit dem verbessertem Sortiervorgang und Verbesserungen beim Erzeugen der Fließkommazahlen (z.B. durch Erhöhen des Ladefaktors) würde unser Programm in völlig neue Dimensionen vorstoßen. Problematisch ist hier natürlich der erhöhte Speicherverbrauch. Je formatierter Fließkommazahl werden inkl. Zeilenschaltung etwa 20 Bytes benötigt, in unserem Beispiel also bereits 10 kByte zusätzlicher Hauptspeicher. Bei größeren Datenmengen könnte hier schnell ein neuer Engpaß entstehen, der einen möglichen Nettogewinn durch die erhöhte Speicherbelastung wieder zunichte macht. Dieser Konflikt zwischen Laufzeitverhalten und Speicherverbrauch tritt in der Praxis recht häufig auf. Er sollte bei allen Optimierungen beachtet werden.

29.3.3 Ausblick

Egal, ob mit dem eingebauten Profiler das Laufzeitverhalten oder der Speicherverbrauch der Anwendung untersucht werden soll, die prinzipielle Vorgehensweise ist stets gleich:

- Zunächst wird mit einer der Optionen `-Xprof` eine Datei mit Profiling-Informationen erstellt.
- Diese wird absteigend nach Rechenzeitverbrauch sortiert, um die für die Optimierung wichtigsten Methoden zu ermitteln.
- Für jede zu optimierende Methode muß über die Aufrufer-Kette bestimmt werden, wie sich die Rechenzeit auf Untermethoden und lokalen Code verteilt.
- Stehen genügend Erkenntnisse zur Verfügung, kann das Programm verbessert werden.
- Ist die Performance zufriedenstellend, kann der Profiler deaktiviert werden, andernfalls beginnt das Spiel von vorne.

Ist der Anteil von lokalem Code am Rechenzeitverbrauch hoch, kann versucht werden, diesen zu verringern. Typische Ansatzpunkte dafür sind das Vermindern der Anzahl von Schleifendurchläufen (durch bessere Algorithmen), die Verwendung von Ganz- statt Fließkommazahlen, das Herausziehen von schleifeninvariantem Code, das Vermeiden der Doppelauswertung von gemeinsamen Teilausdrücken, das Wiederverwenden bekannter Teilergebnisse, die Verwendung alternativer Datenstrukturen, das Eliminieren von unbenutztem Code oder das Reduzieren der Stärke von Ausdrücken, indem sie durch algebraisch gleichwertige, aber schnellere Ausdrücke ersetzt werden. Wird ein großer Anteil der Rechenzeit dagegen in Aufrufen von Untermethoden verbraucht, kann versucht werden, deren Aufrufhäufigkeit zu vermindern, sie durch performantere Aufrufe zu ersetzen, oder - im Falle eigener Methoden - ihre Ablaufgeschwindigkeit zu erhöhen.

Der zielgerichtete Einsatz dieser Techniken erfordert gute Werkzeuge, namentlich einen guten Profiler. Bei kleineren Problemen mag es ausreichend sein, die Ablaufgeschwindigkeit mit Ausgabeanweisungen und `System.currentTimeMillis` zu ermitteln, und auch der JDK-Profiler kann mit Erfolg eingesetzt werden. Daneben gibt es mittlerweile einige kommerzielle und experimentelle Produkte mit wesentlich erweiterten Fähigkeiten. Beispiele für solche Profiler sind *JProbe* (<http://www.klg.com>), der auch für Teile der Software zu diesem Buch verwendet wurde, *OptimizeIt* (<http://www.optimizeit.com>) oder *JInsight* (<http://www.alphaWorks.ibm.com>). Zu ihren Fähigkeiten zählen beispielsweise:

- Die graphische Darstellung der Aufrufhierarchie mit Laufzeitinformationen zu Methoden und Aufrufen.
- Das Ermitteln von Profiling-Informationen bis auf die Ebene einzelner Quelltextzeilen hinab.
- Das Erstellen dynamischer Profile, um die Veränderung wichtiger Parameter bereits während des laufenden Programms beobachten zu können.
- Das Vergleichen von Profiling-Läufen, um die Auswirkungen von Optimierungsversuchen studieren zu können.

29.4 Zusammenfassung

- [29.4 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Allgemeine Bemerkungen zur Geschwindigkeit von Java-Programmen
- Interpretierter Code, kompilierter Code und Just-In-Time-Compiler
- Tuning von String-Zugriffen und Anwendung der Klasse [StringBuffer](#)
- Die Performance von Methodenaufrufen
- Hinweise zur Verwendung von Vektoren und Listen
- Tuning von Dateizugriffen
- Aspekte der Speicherzuordnung in Java
- Anwendung des Profilers und Auswerten der Profiling-Informationen

Kapitel 30

Datenbankzugriffe mit JDBC

- [30 Datenbankzugriffe mit JDBC](#)
 - [30.1 Einleitung](#)
 - [30.1.1 Grundsätzliche Arbeitsweise](#)
 - [30.1.2 Die Architektur von JDBC](#)
 - [Treibertypen](#)
 - [Mehrstufige Client-Server-Architekturen](#)
 - [SQL-2 Entry Level](#)
 - [30.2 Grundlagen von JDBC](#)
 - [30.2.1 Öffnen einer Verbindung](#)
 - [30.2.2 Erzeugen von Anweisungsobjekten](#)
 - [30.2.3 Datenbankabfragen](#)
 - [30.2.4 Datenbankänderungen](#)
 - [30.2.5 Die Klasse SQLException](#)
 - [30.3 Die DirDB-Beispieldatenbank](#)
 - [30.3.1 Anforderungen und Design](#)
 - [30.3.2 Das Rahmenprogramm](#)
 - [30.3.3 Die Verbindung zur Datenbank herstellen](#)
 - [30.3.4 Anlegen und Füllen der Tabellen](#)
 - [30.3.5 Zählen der Verzeichnisse und Dateien](#)
 - [30.3.6 Suchen von Dateien und Verzeichnissen](#)
 - [30.3.7 Die zehn größten Dateien](#)
 - [30.3.8 Speicherverschwendung durch Clustering](#)
 - [30.4 Weiterführende Themen](#)
 - [30.4.1 Metadaten](#)
 - [30.4.2 Escape-Kommandos](#)
 - [30.4.3 Transaktionen](#)
 - [30.4.4 JDBC-Datentypen](#)
 - [30.4.5 Umgang mit JDBC-Objekten](#)
 - [30.4.6 Prepared Statements](#)
 - [30.4.7 SQL-Kurzreferenz](#)
 - [Ändern von Datenstrukturen](#)
 - [Ändern von Daten](#)
 - [Lesen von Daten](#)
 - [30.5 Zusammenfassung](#)

30.1 Einleitung

- [30.1 Einleitung](#)
 - [30.1.1 Grundsätzliche Arbeitsweise](#)
 - [30.1.2 Die Architektur von JDBC](#)
 - [Treibertypen](#)
 - [Mehrstufige Client-Server-Architekturen](#)
 - [SQL-2 Entry Level](#)

Wir wollen uns in diesem Abschnitt zunächst mit der grundsätzlichen Architektur von JDBC und datenbankbasierten Java-Anwendungen vertraut machen. Anschließend werden die wichtigsten Bestandteile der JDBC-Schnittstelle vorgestellt und ihre jeweilige Funktionsweise kurz erläutert. Der nächste Abschnitt illustriert und verfeinert die Konzepte dann an einem praktischen Beispiel, bei dem eine Datenbank zur Speicherung von Verzeichnissen und Dateien entworfen, mit Daten gefüllt und abgefragt wird. Zum Abschluß werden wir einige spezielle Probleme besprechen und auf die Besonderheiten gebräuchlicher JDBC-Treiber eingehen.

Dieses Kapitel ist typisch für die *weiterführenden* Themen in diesem Buch. JDBC ist ein sehr umfassendes Gebiet, das auf den zur Verfügung stehenden Seiten nicht vollständig behandelt werden kann. Wir verfolgen statt dessen einen pragmatischen Ansatz, bei dem wichtige Grundlagen erläutert werden. Mit Hilfe von Beispielen wird ihre praktische Anwendbarkeit demonstriert. Insgesamt müssen aber viele Fragen offen bleiben, die durch die Lektüre weiterer Dokumentationen geschlossen werden können. Dazu zählen beispielsweise *Trigger*, *Blobs* und *Stored Procedures*, die Erweiterungen in JDBC 2.0 oder die Entwicklung eigener JDBC-Treiber.

30.1.1 Grundsätzliche Arbeitsweise

Kurz nachdem die Version 1.0 des Java Development Kit vorlag, begann die Entwicklung einer einheitlichen Datenbankschnittstelle für Java-Programme. Anstelle des von vielen Entwicklern erwarteten objektorientierten Ansatzes verfolgten die Designer dabei das primäre Ziel, die große Zahl vorhandener SQL-Datenbanken problemlos anzubinden. In konzeptioneller Anlehnung an die weitverbreitete ODBC-Schnittstelle wurde daraufhin mit *JDBC (Java Database Connectivity)* ein standardisiertes Java-Datenbank-Interface entwickelt, das mit der Version 1.1 fester Bestandteil des JDK wurde.

JDBC stellt ein *Call-Level-Interface* zur SQL-Datenbank dar. Bei einer solchen Schnittstelle werden die SQL-Anweisungen im Programm als Zeichenketten bearbeitet und zur Ausführung an parametrisierbare Methoden übergeben. Rückgabewerte und Ergebnismengen werden durch Methodenaufrufe ermittelt und nach einer geeigneten Typkonvertierung im Programm weiterverarbeitet.

Dem gegenüber steht ein zweites Verfahren, das als *Embedded SQL (ESQL)* bezeichnet wird. Hierbei werden die SQL-Anweisungen mit besonderen Schlüsselwörtern direkt in den Java-Quelltext eingebettet, und die Kommunikation mit dem Java-Programm erfolgt durch speziell deklarierte *Host-Variablen*. Damit der Java-Compiler durch die eingebetteten SQL-Anweisungen nicht durcheinander gebracht wird, müssen sie zunächst von einem Präprozessor in geeigneten Java-Code übersetzt werden. Während Embedded-SQL insbesondere bei Datenbank Anwendungen, die in C oder C++ geschrieben sind, sehr verbreitet ist, spielt es in Java praktisch keine Rolle und konnte sich gegenüber JDBC nicht durchsetzen.

30.1.2 Die Architektur von JDBC

Treibertypen

JDBC ist keine eigene Datenbank, sondern eine Schnittstelle zwischen einer SQL-Datenbank und der Applikation, die sie benutzen will. Bezüglich der Architektur der zugehörigen Verbindungs-, Anweisungs- und Ergebnisklassen unterscheidet man vier Typen von JDBC-Treibern:

- Steht bereits ein *ODBC-Treiber* zur Verfügung, so kann er mit Hilfe der im Lieferumfang enthaltenen *JDBC-ODBC-Bridge* in Java-Programmen verwendet werden. Diese Konstruktion bezeichnet man als *Typ-1-Treiber*. Mit seiner Hilfe können alle Datenquellen, für die ein ODBC-Treiber existiert, in Java-Programmen genutzt werden.
- Zu vielen Datenbanken gibt es neben ODBC-Treibern auch *spezielle* Treiber des jeweiligen Datenbankherstellers. Setzt ein JDBC-Treiber auf einem solchen proprietären Treiber auf, bezeichnet man ihn als *Typ-2-Treiber*.
- Wenn ein JDBC-Treiber komplett in Java geschrieben und auf dem Client keine spezielle Installation erforderlich ist, der Treiber zur Kommunikation mit einer Datenbank aber auf eine funktionierende *Middleware* angewiesen ist, handelt es sich um einen *Typ-3-Treiber*.
- Falls ein JDBC-Treiber komplett in Java geschrieben ist und die JDBC-Calls direkt in das erforderliche Protokoll der jeweiligen Datenbank umsetzt, handelt es sich um einen *Typ-4-Treiber*.

Mehrstufige Client-Server-Architekturen

Während die Typ-1- und Typ-2-Treiber lokal installierte und konfigurierte Software erfordern (die jeweiligen ODBC- bzw. herstellerspezifischen Treiber), ist dies bei Typ-3- und Typ-4-Treibern normalerweise nicht der Fall. Hier können die zur Anbindung an die Datenbank erforderlichen

Klassendateien zusammen mit der Applikation oder dem Applet aus dem Netz geladen und ggfs. automatisch aktualisiert werden. Nach der Veröffentlichung von JDBC gab es zunächst gar keine Typ-3- oder Typ-4-Treiber. Mittlerweile haben sich aber alle namhaften Datenbankhersteller zu Java bekannt und stellen auch Typ-3- oder Typ-4-Treiber zur Verfügung. Daneben gibt es eine ganze Reihe von Fremdherstellern, die JDBC-Treiber zur Anbindung bekannter Datenbanksysteme zur Verfügung stellen.

Mit JDBC können sowohl zwei- als auch drei- oder höherstufige Client-Server-Systeme aufgebaut werden (*Multi-Tier-Architekturen*). Während bei den zweistufigen Systemen eine Aufteilung der Applikation in Datenbank (Server) und Arbeitsplatz (Client) vorgenommen wird, gibt es bei den dreistufigen Systemen noch eine weitere Schicht, die zwischen beiden Komponenten liegt. Sie wird gemeinhin als *Applikations-Server* bezeichnet und dient dazu, komplexe Operationen vom Arbeitsplatz weg zu verlagern. Der Applikations-Server ist dazu mit dem Datenbank-Server verbunden und kommuniziert mit diesem über ein standardisiertes Protokoll (z.B. JDBC). Den Arbeitsplätzen stellt er dagegen höherwertige Dienste (z.B. komplette Business-Transaktionen) zur Verfügung und kommuniziert mit ihnen über ein spezielles Anwendungsprotokoll (z.B. HTTP, RMI, CORBA oder andere).

SQL-2 Entry Level

JDBC übernimmt die Aufgabe eines »Transportprotokolls« zwischen Datenbank und Anwendung und definiert damit zunächst noch nicht, welche SQL-Kommandos übertragen werden dürfen und welche nicht. Tatsächlich verwendet heute jede relationale Datenbank ihren eigenen SQL-Dialekt, und eine Portierung auf eine andere Datenbank ist nicht selten aufwendiger als ein Wechsel des Compilers.

Um einen minimalen Anspruch an Standardisierung zu gewährleisten, fordert SUN von den JDBC-Treiberherstellern, mindestens den *SQL-2 Entry-Level-Standard* von 1992 zu erfüllen. Mit Hilfe einer von SUN erhältlichen Testsuite können die Hersteller ihre JDBC-Treiber auf Konformität testen. Da praktisch alle großen Datenbanken in ihrer Funktionalität weit über besagten Standard hinausgehen, ist bei Verwendung dieser Features möglicherweise mit erheblichem Portierungsaufwand zu rechnen.

30.2 Grundlagen von JDBC

- [30.2 Grundlagen von JDBC](#)
 - [30.2.1 Öffnen einer Verbindung](#)
 - [30.2.2 Erzeugen von Anweisungsobjekten](#)
 - [30.2.3 Datenbankabfragen](#)
 - [30.2.4 Datenbankänderungen](#)
 - [30.2.5 Die Klasse SQLException](#)

30.2.1 Öffnen einer Verbindung

Bevor mit JDBC auf eine Datenbank zugegriffen werden kann, muß zunächst eine Verbindung zu ihr hergestellt werden. Dazu muß der Datenbanktreiber geladen und initialisiert und mit Hilfe des Treibermanagers ein Verbindungsobjekt beschafft werden. Es bleibt während der gesamten Verbindung bestehen und dient als Lieferant für spezielle Objekte zur Abfrage und Veränderung der Datenbank. Alle Klassen zum Zugriff auf die JDBC-Schnittstelle liegen im Paket [java.sql](#), das am Anfang des Programms importiert werden sollte:

```
import java.sql.*;
```

Jeder JDBC-Treiber hat einen statischen Initialisierer, der beim Laden der Klasse aufgerufen wird. Seine Aufgabe besteht darin, sich beim *Treibermanager* zu registrieren, um bei späteren Verbindungsanfragen von diesem angesprochen werden zu können. Das Laden der Treiberklasse wird üblicherweise durch Aufruf der Methode [forName](#) der Klasse [Class](#) erledigt (siehe [Abschnitt 31.2.2](#)). Um einen Treiber zu laden, muß man also seinen vollständigen Klassennamen kennen:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

[sun.jdbc.odbc.JdbcOdbcDriver](#) ist der Name der JDBC-ODBC-Bridge, mit der die oben erwähnten Typ-1-Treiber realisiert werden. Die Namen alternativer Treiber sind der Dokumentation des jeweiligen Herstellers zu entnehmen.

Nachdem der Treiber geladen wurde, kann er dazu verwendet werden, eine Verbindung zu einer Datenbank aufzubauen. Dazu wird an die statische Methode [getConnection](#) der Klasse [DriverManager](#) ein String und eventuell weitere Parameter übergeben, um den Treibertyp, die Datenbank und nötigenfalls weitere Informationen zu übergeben. [getConnection](#) gibt es in drei Ausprägungen:

[java.sql.DriverManager](#)

```
static Connection getConnection(
    String url
)
```

```
static Connection getConnection(
    String url,
    String user,
    String password
)
```

```
static Connection getConnection(
    String url,
    Properties info
)
```

Die erste Variante erwartet lediglich einen Connection-String als Argument, der in Form eines URL (Uniform Ressource Locator, siehe [Abschnitt 25.4.1](#)) übergeben wird. Der Connection-String besteht aus mehreren Teilen, die durch Doppelpunkte voneinander getrennt sind. Der erste Teil ist immer "jdbc" und zeigt an, daß es sich um einen JDBC-URL handelt. Der zweite Teil wird als *Sub-Protokoll* bezeichnet und gibt an, welcher Treiber verwendet werden soll. Die übrigen Teile sind treiberspezifisch. Connection-Strings für die JDBC-ODBC-Bridge beginnen immer mit "jdbc:odbc", gefolgt von einem weiteren Doppelpunkt, nach dem der Name der ODBC-Datenquelle angegeben wird:

```
con = DriverManager.getConnection("jdbc:odbc:DirDB");
```

Die zweite Variante von [getConnection](#) erlaubt es, zusätzlich den Benutzernamen und das Passwort an die Datenbank zu übergeben. Das ist bei vielen Datenbanken erforderlich, um eine Verbindung aufbauen zu können. Bei der dritten Variante können zusätzlich mit Hilfe eines

[Properties](#)-Objekts weitere, treiberspezifische Informationen übergeben werden. Welche Variante zu verwenden ist, muß der jeweiligen Treiberdokumentation entnommen werden.

Falls die Datenbank nicht geöffnet werden konnte, löst [getConnection](#) eine Ausnahme des Typs [SQLException](#) aus. Diese Ausnahme wird auch von fast allen anderen Methoden und Klassen verwendet, um einen Fehler beim Zugriff auf die Datenbank anzuzeigen.

Hinweis

Wenn die Verbindung erfolgreich aufgebaut werden konnte, liefert [getConnection](#) ein Objekt, das das Interface [Connection](#) implementiert. Dieses Verbindungsobjekt repräsentiert die aktuelle Datenbanksitzung und dient dazu, Anweisungsobjekte zu erzeugen und globale Einstellungen an der Datenbank zu verändern. Das [Connection](#)-Objekt kann durch Aufruf von [close](#) explizit geschlossen werden. Die Verbindung wird automatisch geschlossen, wenn die [Connection](#)-Variable vom Garbage Collector zerstört wird.

30.2.2 Erzeugen von Anweisungsobjekten

Alle Abfragen und Änderungen der Datenbank erfolgen mit Hilfe von *Anweisungsobjekten*. Das sind Objekte, die das Interface [Statement](#) oder eines seiner Subinterfaces implementieren und von speziellen Methoden des [Connection](#)-Objekts erzeugt werden können:

[java.sql.Connection](#)

```
Statement createStatement()
```

```
PreparedStatement prepareStatement(String sql)
```

```
CallableStatement prepareCall(String sql)
```

Die einfachste Form ist dabei das von [createStatement](#) erzeugte [Statement](#)-Objekt. Es kann dazu verwendet werden, unparametrisierte Abfragen und Änderungen der Datenbank zu erzeugen. Seine beiden wichtigsten Methoden sind [executeQuery](#) und [executeUpdate](#). Sie erwarten einen SQL-String als Argument und reichen diesen an die Datenbank weiter. Zurückgegeben wird entweder ein einfacher numerischer Ergebniswert, der den Erfolg der Anweisung anzeigt, oder eine Menge von Datenbanksätzen, die das Ergebnis der Abfrage repräsentieren. Auf die beiden übrigen Anweisungstypen werden wir weiter unten zurückkommen.

[Statement](#)-Objekte sind bei manchen Treibern *kostspielige* Ressourcen, deren Erzeugen viel Speicher oder Rechenzeit kostet. Das Erzeugen einer großen Anzahl von [Statement](#)-Objekten (beispielsweise beim Durchlaufen einer Schleife) sollte in diesem Fall vermieden werden. Viele JDBC-Programme legen daher nach dem Öffnen der Verbindung eine Reihe von vordefinierten [Statement](#)-Objekten an und verwenden diese immer wieder. Obwohl das im Prinzip problemlos möglich ist, kann es in der Praxis leicht dazu führen, daß ein [Statement](#)-Objekt, das noch in Gebrauch ist (beispielsweise, weil seine Ergebnismenge noch nicht vollständig abgefragt ist), erneut verwendet wird. Das Verhalten des Programms ist dann natürlich undefiniert. Wir werden weiter unten in [Abschnitt 30.4.5](#) eine Lösung für dieses Problem kennenlernen.

Warnung

30.2.3 Datenbankabfragen

Hat man ein [Statement](#)-Objekt beschafft, kann dessen Methode [executeQuery](#) verwendet werden, um Daten aus der Datenbank zu lesen:

[java.sql.Statement](#)

```
public ResultSet executeQuery(String sql)
    throws SQLException
```

Die Methode erwartet einen SQL-String in Form einer gültigen *SELECT-Anweisung* und gibt ein Objekt vom Typ [ResultSet](#) zurück, das die Ergebnismenge repräsentiert. Als Argument dürfen beliebige SELECT-Anweisungen übergeben werden, sofern sie für die zugrundeliegende Datenbank gültig sind. Die folgende SQL-Anweisung selektiert beispielsweise alle Sätze aus der Tabelle `dir`, deren Feld `did` den Wert 7 hat:

```
SELECT * FROM dir WHERE did = 7
```

Das zurückgegebene Objekt vom Typ [ResultSet](#) besitzt eine Methode [next](#), mit der die Ergebnismenge schrittweise durchlaufen werden kann:

[java.sql.ResultSet](#)

```
boolean next()
```

Nach dem Aufruf von [executeQuery](#) steht der Satzzeiger zunächst *vor* dem ersten Element, jeder Aufruf von [next](#) positioniert ihn auf das nächste Element. Der Rückgabewert gibt an, ob die Operation erfolgreich war. Ist er [false](#), gibt es keine weiteren Elemente in der Ergebnismenge. Ist er dagegen [true](#), konnte das nächste Element erfolgreich ausgewählt werden und mit Hilfe verschiedener [get...](#)-Methoden kann nun auf die einzelnen Spalten zugegriffen werden. Jede dieser Methoden steht in zwei unterschiedlichen Varianten zur Verfügung:

- Wird ein numerischer Wert *n* als Argument übergeben, so wird dieser als Spaltenindex interpretiert und der Wert der *n*-ten Spalte zurückgegeben. Wichtig: Anders als bei Arrays hat die erste Spalte den Wert 1.
- Wird ein [String](#) als Argument übergeben, so wird er als Name interpretiert und der Wert der Spalte mit diesem Namen zurückgegeben.

Diese Variante soll zwar marginal langsamer als die erste sein, ist aber weniger fehlerträchtig. Da der Aufruf nicht mehr von der Spaltenreihenfolge der Abfrage abhängt, ist ihr normalerweise der Vorzug zu geben.

Um dem Entwickler lästige Typkonvertierungen zu ersparen, gibt es alle `getXXX`-Methoden in unterschiedlichen Typisierungen. So liefert beispielsweise `getString` das gewünschte Feld als `String`, während `getInt` es als `int` zurückgibt. Wo es möglich und sinnvoll ist, werden automatische Typkonvertierungen durchgeführt; `getString` kann beispielsweise für nahezu alle Typen verwendet werden. [Tabelle 30.1](#) gibt eine Übersicht über die wichtigsten `get`-Methoden der Klasse `ResultSet`. In [Tabelle 30.4](#) findet sich eine Übersicht der wichtigsten SQL-Datentypen.

Rückgabewert	Methodenname
boolean	getBoolean
byte	getByte
byte[]	getBytes
Date	getDate
double	getDouble
float	getFloat
int	getInt
long	getLong
short	getShort
String	getString
Time	getTime
Timestamp	getTimestamp

Tabelle 30.1: `get`-Methoden von `ResultSet`

Soll festgestellt werden, ob eine Spalte den Wert `NULL` hatte, so kann das *nach* dem Aufruf der `get`-Methode durch Aufruf von `wasNull` abgefragt werden. `wasNull` gibt genau dann `true` zurück, wenn die letzte abgefragte Spalte einen `NULL`-Wert als Inhalt hatte. Bei allen Spalten, die `NULL`-Werte enthalten können, *muß* diese Abfrage also erfolgen. Bei den `get`-Methoden, die ein Objekt als Ergebniswert haben, geht es etwas einfacher. Hier wird `null` zurückgegeben, wenn der Spaltenwert `NULL` war.

Hinweis

30.2.4 Datenbankänderungen

Datenbankänderungen werden mit den SQL-Anweisungen `INSERT INTO`, `UPDATE` oder `DELETE FROM` oder den SQL-DDL-Anweisungen (*Data Definition Language*) zum Ändern der Datenbankstruktur durchgeführt. Im Gegensatz zu Datenbankabfragen geben diese Anweisungen keine Ergebnismenge zurück, sondern lediglich einen einzelnen Wert. Im Falle von `INSERT INTO`, `UPDATE` und `DELETE FROM` gibt dieser Wert an, wie viele Datensätze von der Änderung betroffen waren, bei DDL-Anweisungen ist er immer 0.

Um solche Anweisungen durchzuführen, stellt das Interface `Statement` die Methode `executeUpdate` zur Verfügung:

```
public int executeUpdate(String sql)
    throws SQLException
```

`java.sql.Statement`

Auch sie erwartet als Argument einen String mit einer gültigen SQL-Anweisung, beispielsweise:

```
INSERT INTO dir VALUES (1, 'x.txt', 0)
```

Könnte diese Anweisung erfolgreich ausgeführt werden, würde sie 1 zurückgeben. Andernfalls würde eine `SQLException` ausgelöst.

30.2.5 Die Klasse `SQLException`

Wenn SQL-Anweisungen fehlschlagen, lösen sie normalerweise eine Ausnahme des Typs `SQLException` aus. Das gilt sowohl, wenn keine Verbindung zur Datenbank zustande gekommen ist, als auch bei allen Arten von Syntaxfehlern in SQL-Anweisungen. Auch bei semantischen Fehlern durch falsche Typisierung oder inhaltlich fehlerhafte SQL-Anweisungen wird eine solche Ausnahme ausgelöst. `SQLException` ist eine Erweiterung der Klasse `Exception` und stellt folgende zusätzliche Methoden zur Verfügung:

```
int getErrorCode()
```

```
String getSQLState()
```

```
SQLException getNextException()
```

Mit [getErrorCode](#) kann der herstellerspezifische Fehlercode abgefragt werden, [getSQLState](#) liefert den internen SQL-Zustandscode. Etwas ungewöhnlich ist die Methode [getNextException](#), denn sie unterstützt die *Verkettung* von Ausnahmen. Jeder Aufruf holt die nächste Ausnahme aus der Liste. Ist der Rückgabewert [null](#), gibt es keine weiteren Ausnahmen. Code zum Behandeln einer [SQLException](#) könnte also etwa so aussehen:

```
001 ...
002 catch (SQLException e) {
003     while (e != null) {
004         System.err.println(e.toString());
005         System.err.println("SQL-State: " + e.getSQLState());
006         System.err.println("ErrorCode: " + e.getErrorCode());
007         e = e.getNextException();
008     }
009 }
```

Listing 30.1: Behandeln einer SQLException

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	≤	≤	≥	≥
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	-------------------	-------------------	-------------------	-------------------

30.3 Die DirDB-Beispieldatenbank

- 30.3 Die DirDB-Beispieldatenbank
 - 30.3.1 Anforderungen und Design
 - 30.3.2 Das Rahmenprogramm
 - 30.3.3 Die Verbindung zur Datenbank herstellen
 - 30.3.4 Anlegen und Füllen der Tabellen
 - 30.3.5 Zählen der Verzeichnisse und Dateien
 - 30.3.6 Suchen von Dateien und Verzeichnissen
 - 30.3.7 Die zehn größten Dateien
 - 30.3.8 Speicherverschwendung durch Clustering

30.3.1 Anforderungen und Design

In diesem Abschnitt wollen wir uns die zuvor eingeführten Konzepte in der Praxis ansehen. Dazu erzeugen wir eine einfache Datenbank *DirDB*, die Informationen zu Dateien und Verzeichnissen speichern kann. Über eine einfache Kommandozeilenschnittstelle können die Tabellen mit den Informationen aus dem lokalen Dateisystem gefüllt und auf unterschiedliche Weise abgefragt werden.

DirDB besitzt lediglich zwei Tabellen *dir* und *file* für Verzeichnisse und Dateien. Sie haben folgende Struktur:

Name	Typ	Bedeutung
did	INT	Primärschlüssel
dname	CHAR(100)	Verzeichnisname
fatherdid	INT	Schlüssel Vaterverzeichnis
entries	INT	Anzahl der Verzeichniseinträge

Tabelle 30.2: Die Struktur der dir-Tabelle

Name	Typ	Bedeutung
fid	INT	Primärschlüssel
did	INT	Zugehöriges Verzeichnis
fname	CHAR(100)	Dateiname
fsize	INT	Dateigröße
fdate	DATE	Änderungsdatum
ftime	TIME	Änderungszeit

Tabelle 30.3: Die Struktur der file-Tabelle

Beide Tabellen besitzen einen Primärschlüssel, der beim Anlegen eines neuen Satzes vom Programm vergeben wird. Die Struktur von *dir* ist baumartig; im Feld *fatherid* wird ein Verweis auf das Verzeichnis gehalten, in dem das aktuelle Verzeichnis enthalten ist. Dessen Wert ist im Startverzeichnis per Definition 0. Über den Fremdschlüssel *did* zeigt jeder Datensatz aus der *file*-Tabelle an, zu welchem Verzeichnis er gehört. Die Tabellen stehen demnach in einer 1:n-Beziehung zueinander. Auch die Tabelle *dir* steht in einer 1:n-Beziehung zu sich selbst. [Abbildung 30.1](#) zeigt ein vereinfachtes E/R-Diagramm des Tabellendesigns.

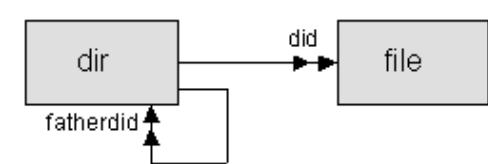


Abbildung 30.1: E/R-Diagramm für DirDB

Das Programm `DirDB.java` soll folgende Anforderungen erfüllen:

- Es soll wahlweise mit *MS-Access 7.0* oder mit *InstantDB 1.85* zusammenarbeiten.

- Es soll folgende Befehle interpretieren:
 - Leeren der Datenbank und Einlesen eines vorgegebenen Verzeichnisses inkl. aller Unterverzeichnisse.
 - Zählen der Dateien und Verzeichnisse in der Datenbank.
 - Suchen beliebiger Dateien oder Verzeichnisse.
 - Anzeige der größten Dateien.
 - Berechnen des durch Clustering verschwendeten Festplattenplatzes.

30.3.2 Das Rahmenprogramm

Wir implementieren eine Klasse `DirDB`, die (der Einfachheit halber) alle Funktionen mit Hilfe statischer Methoden realisiert. Die Klasse und ihre `main`-Methode sehen so aus:

```

001 import java.util.*;
002 import java.io.*;
003 import java.sql.*;
004 import java.text.*;
005 import gk.util.*;
006
007 public class DirDB
008 {
009     //---Constants-----
010     static int INSTANT185 = 1;
011     static int ACCESS95   = 2;
012
013     //---Pseudo constants-----
014     static String FILESEP = System.getProperty("file.separator");
015
016     //---Static Variables-----
017     static int          db = INSTANT185;
018     static Connection   con;
019     static Statement    stmt;
020     static Statement    stmt1;
021     static DatabaseMetaData dmd;
022     static int          nextdid = 1;
023     static int          nextfid = 1;
024
025     //---main-----
026     public static void main(String args[])
027     {
028         if (args.length < 1) {
029             System.out.println(
030                 "usage: java DirDB [A|I] <command> [<options>]
031                 ");
032             System.out.println("");
033             System.out.println("command          options");
034             System.out.println("-----");
035             System.out.println("POPULATE        <directory>");
036             System.out.println("COUNT");
037             System.out.println("FINDFILE        <name>");
038             System.out.println("FINDDIR         <name>");
039             System.out.println("BIGGESTFILES    <howmany>");
040             System.out.println("CLUSTERING      <clustersize>");
041             System.exit(1);
042         }
043         if (args[0].equalsIgnoreCase("A")) {
044             db = ACCESS95;
045         }
046         try {
047             if (args[1].equalsIgnoreCase("populate")) {
048                 open();
049                 createTables();
050                 populate(args[2]);
051                 close();
052             } else if (args[1].equalsIgnoreCase("count")) {

```

```

053         open();
054         countRecords();
055         close();
056     } else if (args[1].equalsIgnoreCase("findfile")) {
057         open();
058         findFile(args[2]);
059         close();
060     } else if (args[1].equalsIgnoreCase("finddir")) {
061         open();
062         findDir(args[2]);
063         close();
064     } else if (args[1].equalsIgnoreCase("biggestfiles")) {
065         open();
066         biggestFiles(Integer.parseInt(args[2]));
067         close();
068     } else if (args[1].equalsIgnoreCase("clustering")) {
069         open();
070         clustering(Integer.parseInt(args[2]));
071         close();
072     }
073 } catch (SQLException e) {
074     while (e != null) {
075         System.err.println(e.toString());
076         System.err.println("SQL-State: " + e.getSQLState());
077         System.err.println("ErrorCode: " + e.getErrorCode());
078         e = e.getNextException();
079     }
080     System.exit(1);
081 } catch (Exception e) {
082     System.err.println(e.toString());
083     System.exit(1);
084 }
085 }
086 }

```

Listing 30.2: Das Rahmenprogramm der DirDB-Datenbank

In [main](#) wird zunächst ein *usage*-Text definiert, der immer dann aufgerufen wird, wenn das Programm ohne Argumente aufgerufen wird. Die korrekte Aufrufsyntax ist:

```
java DirDB [A|I] <command> [<options>]
```

Nach dem Programmnamen folgt zunächst der Buchstabe "A" oder "I", um anzugeben, ob die Access- oder InstantDB-Datenbank verwendet werden soll. Das nächste Argument gibt den Namen des gewünschten Kommandos an. In der folgenden verschachtelten Verzweigung werden gegebenenfalls weitere Argumente gelesen und die Methode zum Ausführen des Programms aufgerufen. Den Abschluß der [main](#)-Methode bildet die Fehlerbehandlung, bei der die Ausnahmen des Typs [SQLException](#) und [Exception](#) getrennt behandelt werden.

Das vollständige Programm findet sich auf der CD-ROM zum Buch unter dem Namen [DirDB.java](#). In diesem Abschnitt sind zwar auch alle Teile abgedruckt, sie finden sich jedoch nicht zusammenhängend wieder, sondern sind über die verschiedenen Unterabschnitte verteilt. Das importierte Paket [gk.util](#) kann wie in [Abschnitt 8.2.3](#) beschrieben installiert werden.

Hinweis

30.3.3 Die Verbindung zur Datenbank herstellen

Wie in [Listing 30.2](#) zu sehen ist, rufen alle Kommandos zunächst die Methode [open](#) zum Öffnen der Datenbank auf. Anschließend führen sie ihre spezifischen Kommandos aus und rufen dann [close](#) auf, um die Datenbank wieder zu schließen.

Beim Öffnen der Datenbank wird zunächst mit [Class.forName](#) der passende Datenbanktreiber geladen und beim Treibermanager registriert. Anschließend besorgt das Programm ein [Connection](#)-Objekt, das an die statische Variable [con](#) gebunden wird. An dieser Stelle sind die potentiellen Code-Unterschiede zwischen den beiden Datenbanken gut zu erkennen:

- Während der Treibername für InstantDB die Bezeichnung [jdbc.idbDriver](#) hat, lautet er für die JDBC-ODBC-Bridge [sun.jdbc.odbc.JdbcOdbcDriver](#).
- Der Connection-URL bei InstantDB ist "jdbc:idb=", gefolgt vom Namen einer Property-Datei, die zusätzliche Konfigurationsangaben enthält. Bei der JDBC-ODBC-Bridge beginnt der URL immer mit "jdbc:odbc", gefolgt vom Namen der ODBC-Datenquelle.

```

001 /**
002  * Öffnet die Datenbank.
003  */
004 public static void open()
005 throws Exception
006 {
007     //Treiber laden und Connection erzeugen
008     if (db == INSTANT185) {
009         Class.forName("jdbc.idbDriver");
010         con = DriverManager.getConnection(
011             "jdbc:idb=dirdb.prp",
012             new Properties()
013             );
014     } else {
015         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
016         con = DriverManager.getConnection("jdbc:odbc:DirDB");
017     }
018     //Metadaten ausgeben
019     dmd = con.getMetaData();
020     System.out.println("");
021     System.out.println("Connection URL: " + dmd.getURL());
022     System.out.println("Driver Name:      " + dmd.getDriverName());
023     System.out.println(
024         "Driver Version: " + dmd.getDriverVersion()
025     );
026     System.out.println("");
027     //Statementobjekte erzeugen
028     stmt = con.createStatement();
029     stmt1 = con.createStatement();
030 }
031
032 /**
033  * Schließt die Datenbank.
034  */
035 public static void close()
036 throws SQLException
037 {
038     stmt.close();
039     stmt1.close();
040     con.close();
041 }

```

Listing 30.3: Öffnen und Schließen der DirDB-Datenbank

Nachdem die Verbindung hergestellt wurde, liefert der Aufruf von [getMetaData](#) ein Objekt des Typs [DatabaseMetaData](#). Es kann dazu verwendet werden, weitere Informationen über die Datenbank abzufragen. Wir geben lediglich den Connection-String und Versionsinformationen zu den geladenen Treibern aus. [DatabaseMetaData](#) besitzt darüber hinaus noch viele weitere Variablen und Methoden, auf die wir hier nicht näher eingehen wollen. Am Ende von [open](#) erzeugt das Programm zwei [Statement](#)-Objekte `stmt` und `stmt1`, die in den übrigen Methoden zum Ausführen der SQL-Befehle verwendet werden. Zum Schließen der Datenbank werden zunächst die beiden [Statement](#)-Objekte und dann die Verbindung selbst geschlossen.

30.3.4 Anlegen und Füllen der Tabellen

Unsere Anwendung geht davon aus, daß die Datenbank bereits angelegt ist, erstellt die nötigen Tabellen und Indexdateien aber selbst. Die dazu nötigen SQL-Befehle sind [CREATE TABLE](#) zum Anlegen einer Tabelle und [CREATE INDEX](#) zum Anlegen einer Indexdatei. Diese Befehle werden mit der Methode [executeUpdate](#) des [Statement](#)-Objekts ausgeführt, denn sie produzieren keine Ergebnismenge, sondern als DDL-Anweisungen lediglich den Rückgabewert 0. Das Anlegen der Tabellen erfolgt mit der Methode [createTables](#):

```

001 /**
002  * Legt die Tabellen an.
003  */
004 public static void createTables()
005 throws SQLException
006 {
007     //Anlegen der Tabelle dir
008     try {
009         stmt.executeUpdate("DROP TABLE dir");
010     } catch (SQLException e) {
011         //Nichts zu tun
012     }
013     stmt.executeUpdate("CREATE TABLE dir (" +
014         "did          INT," +
015         "dname        CHAR(100)," +
016         "fatherdid INT," +
017         "entries      INT)"
018 );
019     stmt.executeUpdate("CREATE INDEX idir1 ON dir ( did )");
020     stmt.executeUpdate("CREATE INDEX idir2 ON dir ( fatherdid )");
021     //Anlegen der Tabelle file
022     try {
023         stmt.executeUpdate("DROP TABLE file");
024     } catch (SQLException e) {
025         //Nichts zu tun
026     }
027     stmt.executeUpdate("CREATE TABLE file (" +
028         "fid          INT ," +
029         "did          INT," +
030         "fname        CHAR(100)," +
031         "fsize        INT," +
032         "fdate        DATE," +
033         "ftime        CHAR(5))"
034 );
035     stmt.executeUpdate("CREATE INDEX ifile1 ON file ( fid )");
036 }

```

Listing 30.4: Anlegen der DirDB-Tabellen

Um die Tabellen zu löschen, falls sie bereits vorhanden sind, wird zunächst die Anweisung `DROP TABLE` ausgeführt. Sie ist in einen eigenen [try-catch](#)-Block gekapselt, denn manche Datenbanken lösen eine Ausnahme aus, falls die zu löschenden Tabellen nicht existieren. Diesen »Fehler« wollen wir natürlich ignorieren und nicht an den Aufrufer weitergeben.

Hinweis

Nachdem die Tabellen angelegt wurden, können sie mit der Methode `populate` gefüllt werden. `populate` bekommt dazu vom Rahmenprogramm den Namen des Startverzeichnisses übergeben, das rekursiv durchlaufen werden soll, und ruft `addDirectory` auf, um das erste Verzeichnis mit Hilfe des Kommandos `INSERT INTO` (das ebenfalls an `executeUpdate` übergeben wird) in die Tabelle `dir` einzutragen. Der Code sieht etwas unleserlich aus, weil einige Stringliterals einschließlich der zugehörigen einfachen Anführungsstriche übergeben werden müssen. Sie dienen in SQL-Befehlen als Begrenzungszeichen von Zeichenketten.

Für das aktuelle Verzeichnis wird dann ein `File`-Objekt erzeugt und mit `listFiles` eine Liste der Dateien und Verzeichnisse in diesem Verzeichnis erstellt. Jede Datei wird mit einem weiteren `INSERT INTO` in die `file`-Tabelle eingetragen, für jedes Unterverzeichnis ruft `addDirectory` sich selbst rekursiv auf.

Am Ende wird mit einem `UPDATE`-Kommando die Anzahl der Einträge im aktuellen Verzeichnis in das Feld `entries` der Tabelle `dir` eingetragen. Der Grund für diese etwas umständliche Vorgehensweise (wir hätten das auch gleich beim Anlegen des `dir`-Satzes erledigen können) besteht darin, daß wir auch ein Beispiel für die Anwendung der `UPDATE`-Anweisung geben wollten.

Hinweis

```

001 /**
002  * Durchläuft den Verzeichnisbaum rekursiv und schreibt
003  * Verzeichnis- und Dateinamen in die Datenbank.
004  */
005 public static void populate(String dir)
006 throws Exception
007 {
008     addDirectory(0, "", dir);
009 }
010
011 /**
012  * Fügt das angegebene Verzeichnis und alle
013  * Unterverzeichnisse mit allen darin enthaltenen
014  * Dateien zur Datenbank hinzu.
015  */
016 public static void addDirectory(
017     int fatherdid, String parent, String name
018 )
019 throws Exception
020 {
021     String dirname = "";
022     if (parent.length() > 0) {
023         dirname = parent;
024         if (!parent.endsWith(FILESEP)) {
025             dirname += FILESEP;
026         }
027     }
028     dirname += name;
029     System.out.println("processing " + dirname);
030     File dir = new File(dirname);
031     if (!dir.isDirectory()) {
032         throw new Exception("not a directory: " + dirname);
033     }
034     //Verzeichnis anlegen
035     int did = nextdid++;
036     stmt.executeUpdate(
037         "INSERT INTO dir VALUES (" +
038         did + "," +
039         "\"" + name + "\", " +
040         fatherdid + "," +
041         "0)"
042     );
043     //Verzeichniseinträge lesen
044     File entries[] = dir.listFiles();
045     //Verzeichnis durchlaufen
046     for (int i = 0; i < entries.length; ++i) {
047         if (entries[i].isDirectory()) {
048             addDirectory(did, dirname, entries[i].getName());
049         } else {
050             java.util.Date d = new java.util.Date(
051                 entries[i].lastModified()
052             );
053             SimpleDateFormat sdf;
054             //Datum
055             sdf = new SimpleDateFormat("yyyy-MM-dd");
056             String date = sdf.format(d);
057             //Zeit
058             sdf = new SimpleDateFormat("HH:mm");
059             String time = sdf.format(d);
060             //Satz anhängen
061             stmt.executeUpdate(
062                 "INSERT INTO file VALUES (" +
063                 (nextfid++) + "," +
064                 did + "," +

```

```

065         "\" + entries[i].getName() + "\", " +
066         entries[i].length() + ", " +
067         "{d \" + date + \"\", " +
068         "\" + time + \"\"}";
069     };
070     System.out.println("  " + entries[i].getName());
071 }
072 }
073 //Anzahl der Einträge aktualisieren
074 stmt.executeUpdate(
075     "UPDATE dir SET entries = " + entries.length +
076     " WHERE did = " + did
077 );
078 }

```

Listing 30.5: Füllen der DirDB-Tabellen

Hier tauchen die beiden im Rahmenprogramm definierten statischen Variablen `nextdid` und `nextfid` wieder auf. Sie liefern die Primärschlüssel für Datei- und Verzeichnissätze und werden nach jedem eingefügten Satz automatisch um eins erhöht. Daß dieses Verfahren nicht mehrbenutzerfähig ist, leuchtet ein, denn die Zähler werden *lokal zur laufenden Applikation* erhöht. Eine bessere Lösung bieten Primärschlüsselfelder, die beim Einfügen *von der Datenbank* einen automatisch hochgezählten eindeutigen Wert erhalten. Dafür sieht JDBC allerdings kein standardisiertes Verfahren vor, meist kann ein solches Feld in der INSERT INTO-Anweisung einfach ausgelassen werden. Alternativ könnten Schlüsselwerte vor dem Einfügen aus einer zentralen Key-Tabelle geholt und transaktionssicher hochgezählt werden.

Hinweis

30.3.5 Zählen der Verzeichnisse und Dateien

Das `COUNT`-Kommando soll die Anzahl der Verzeichnisse und Dateien zählen, die mit dem `POPULATE`-Kommando in die Datei eingefügt wurden. Wir verwenden dazu ein einfaches `SELECT`-Kommando, das mit der `COUNT(*)`-Option die Anzahl der Sätze in einer Tabelle zählt:

```

001 /**
002  * Gibt die Anzahl der Dateien und Verzeichnisse aus.
003  */
004 public static void countRecords()
005 throws SQLException
006 {
007     ResultSet rs = stmt.executeQuery(
008         "SELECT count(*) FROM dir"
009     );
010     if (!rs.next()) {
011         throw new SQLException("SELECT COUNT(*): no result");
012     }
013     System.out.println("Directories: " + rs.getInt(1));
014     rs = stmt.executeQuery("SELECT count(*) FROM file");
015     if (!rs.next()) {
016         throw new SQLException("SELECT COUNT(*): no result");
017     }
018     System.out.println("Files: " + rs.getInt(1));
019     rs.close();
020 }

```

Listing 30.6: Anzahl der Sätze in der DirDB-Datenbank

Die `SELECT`-Befehle werden mit der Methode `executeQuery` an das `Statement`-Objekt übergeben, denn wir erwarten nicht nur eine einfache Ganzzahl als Rückgabewert, sondern eine komplette Ergebnismenge. Die Besonderheit liegt in diesem Fall darin, daß wegen der Spaltenangabe `COUNT(*)` lediglich ein einziger Satz zurückgegeben wird, der auch nur ein einziges Feld enthält. Auf dieses können wir am einfachsten über seinen numerischen Index 1 zugreifen und es durch Aufruf von `getInt` gleich in ein `int` umwandeln lassen. Das Ergebnis geben wir auf dem Bildschirm aus und wiederholen anschließend dieselbe Prozedur für die Tabelle `file`.

30.3.6 Suchen von Dateien und Verzeichnissen

Um eine bestimmte Datei oder Tabelle in unserer Datenbank zu suchen, verwenden wir ebenfalls ein `SELECT`-Statement. Im Gegensatz zu vorher lassen wir uns nun mit der Spaltenangabe `"*"` *alle* Felder der Tabelle geben. Zudem hängen wir an die Abfrageanweisung eine `WHERE`-Klausel an, um eine Suchbedingung formulieren zu können. Mit Hilfe des `LIKE`-Operators führen wir eine Mustersuche durch, bei der die beiden SQL-Wildcards `"%"` (eine beliebige Anzahl Zeichen) und `"_"` (ein einzelnes beliebiges Zeichen) verwendet werden können. Zusätzlich bekommt

InstantDB die (nicht SQL-92-konforme) Option `IGNORE CASE` angehängt, um bei der Suche nicht zwischen Groß- und Kleinschreibung zu unterscheiden (ist bei Access 7.0 nicht nötig).

Die von `executeQuery` zurückgegebene Ergebnismenge wird mit `next` Satz für Satz durchlaufen und auf dem Bildschirm ausgegeben. Mit Hilfe der Methode `getDirPath` wird zuvor der zugehörige Verzeichnisname rekonstruiert und vor dem Dateinamen ausgegeben. Dazu wird in einer Schleife zur angegebenen `did` (Verzeichnisschlüssel) so lange das zugehörige Verzeichnis gesucht, bis dessen `fatherdid` 0 ist, also das Startverzeichnis erreicht ist. Rückwärts zusammengebaut und mit Trennzeichen versehen ergibt diese Namenskette den kompletten Verzeichnisnamen.

Der in dieser Methode verwendete `ResultSet` wurde mit dem zweiten `Statement`-Objekt `stmt1` erzeugt. Hätten wir dafür die zu diesem Zeitpunkt noch geöffnete Variable `stmt` verwendet, wäre das Verhalten des Programmes undefiniert gewesen, weil die bestehende Ergebnismenge durch das Erzeugen einer neuen Ergebnismenge auf demselben `Statement`-Objekt ungültig geworden wäre.

Warnung

```
001 /**
002  * Gibt eine Liste aller Files auf dem Bildschirm aus,
003  * die zu dem angegebenen Dateinamen passen. Darin dürfen
004  * die üblichen SQL-Wildcards % und _ enthalten sein.
005  */
006 public static void findFile(String name)
007 throws SQLException
008 {
009     String query = "SELECT * FROM file " +
010                   "WHERE fname LIKE '\" + name + \"'";
011     if (db == INSTANT185) {
012         query += " IGNORE CASE";
013     }
014     ResultSet rs = stmt.executeQuery(query);
015     while (rs.next()) {
016         String path = getDirPath(rs.getInt("did"));
017         System.out.println(
018             path + FILESEP +
019             rs.getString("fname").trim()
020         );
021     }
022     rs.close();
023 }
024
025 /**
026  * Liefert den Pfadnamen zu dem Verzeichnis mit dem
027  * angegebenen Schlüssel.
028  */
029 public static String getDirPath(int did)
030 throws SQLException
031 {
032     String ret = "";
033     while (true) {
034         ResultSet rs = stmt1.executeQuery(
035             "SELECT * FROM dir WHERE did = " + did
036         );
037         if (!rs.next()) {
038             throw new SQLException(
039                 "no dir record found with did = " + did
040             );
041         }
042         ret = rs.getString("dname").trim() +
043             (ret.length() > 0 ? FILESEP + ret : "");
044         if ((did = rs.getInt("fatherdid")) == 0) {
045             break;
046         }
047     }
048     return ret;
049 }
```

Listing 30.7: Suchen nach Dateien in der DirDB-Datenbank

Das DirDB-Programm bietet mit dem Kommando `FINDDIR` auch die Möglichkeit, nach Verzeichnisnamen zu suchen. Die Implementierung dieser Funktion ähnelt der vorigen und wird durch die Methode `findDir` realisiert:

```
001 /**
002  * Gibt eine Liste aller Verzeichnisse auf dem Bildschirm
003  * aus, die zu dem angegebenen Verzeichnisnamen passen.
004  * Darin dürfen die üblichen SQL-Wildcards % und _
005  * enthalten sein.
006  */
007 public static void findDir(String name)
008 throws SQLException
009 {
010     String query = "SELECT * FROM dir " +
011                    "WHERE dname LIKE \'\' + name + "\'";
012     if (db == INSTANT185) {
013         query += " IGNORE CASE";
014     }
015     ResultSet rs = stmt.executeQuery(query);
016     while (rs.next()) {
017         System.out.println(
018             getDirPath(rs.getInt("did")) +
019             " (" + rs.getInt("entries") + " entries)"
020         );
021     }
022     rs.close();
023 }
```

Listing 30.8: Suchen nach Verzeichnissen in der DirDB-Datenbank

Wird das DirDB-Programm von der Kommandozeile aufgerufen, kann es unter Umständen schwierig sein, die Wildcards "%" oder "_" einzugeben, weil sie vom Betriebssystem oder der Shell als Sonderzeichen angesehen werden. Durch Voranstellen des passenden Escape-Zeichens (das könnte beispielsweise der Backslash sein) kann die Sonderbedeutung aufgehoben werden. In der DOS-Box von Windows 95 oder NT kann die Sonderbedeutung des "%" nur aufgehoben werden, indem das Zeichen "%" doppelt geschrieben wird. Soll beispielsweise nach allen Dateien mit der Erweiterung ".java" gesucht werden, so ist DirDb unter Windows wie folgt aufzurufen:

```
java DirDB I findfile %%.java
```

Weiterhin ist zu beachten, daß die Interpretation der Wildcards von den unterschiedlichen Datenbanken leider nicht einheitlich gehandhabt wird. Während das obige Kommando unter InstantDB korrekt funktioniert, ist bei der Access-Datenbank die Ergebnismenge leer. Der Grund kann - je nach verwendeter Version - darin liegen, daß entweder der Stern "*" anstelle des "%" als Wildcard erwartet wird oder daß die *Leerzeichen am Ende des Feldes* als signifikant angesehen werden, und daher auch *hinter* dem Suchbegriff ein Wildcard-Zeichen angegeben werden muß:

```
java DirDB A findfile %%.java%%
```

Hinweis

30.3.7 Die zehn größten Dateien

Eine ähnliche `SELECT`-Anweisung begegnet uns, wenn wir uns die Aufgabe stellen, die *howmany* größten Dateien unserer Datenbank anzuzeigen. Hierzu fügen wir eine `ORDER BY`-Klausel an und sortieren die Abfrage absteigend nach der Spalte *fsize*. Von der Ergebnismenge geben wir dann die ersten *howmany* Elemente aus:

```

001 /**
002  * Gibt die howmany größten Dateien aus.
003  */
004 public static void biggestFiles(int howmany)
005 throws SQLException
006 {
007     ResultSet rs = stmt.executeQuery(
008         "SELECT * FROM file ORDER BY fsize DESC"
009     );
010     for (int i = 0; i < howmany; ++i) {
011         if (rs.next()) {
012             System.out.print(
013                 getDirPath(rs.getInt("did")) +
014                 FILESEP + rs.getString("fname").trim()
015             );
016             System.out.println(
017                 Str.getFormatted("%10d", rs.getInt("fsize"))
018             );
019         }
020     }
021     rs.close();
022 }

```

Listing 30.9: Sortieren der Ergebnismenge

30.3.8 Speicherverschwendung durch Clustering

Bevor wir uns weiterführenden Themen zuwenden, wollen wir uns eine letzte Anwendung unserer Beispieldatenbank ansehen. Viele Dateisysteme (allen voran das alte FAT-Dateisystem unter MS-DOS und Windows) speichern die Dateien in verketteten Zuordnungseinheiten fester Größe, den *Clustern*. Ist die Clustergröße beispielsweise 4096 Bytes, so belegt eine Datei auch dann 4 kByte Speicher, wenn sie nur ein Byte groß ist. Immer, wenn die Größe einer Datei nicht ein genaues Vielfaches der Clustergröße ist, bleibt der letzte Cluster unvollständig belegt und wertvoller Plattenspeicher bleibt ungenutzt. Ist die Clustergröße hoch, wird vor allen dann viel Platz verschwendet, wenn das Dateisystem sehr viele kleine Dateien enthält. Die folgende Funktion `clustering` berechnet zu einer gegebenen Clustergröße die Summe der Dateilängen und stellt sie dem tatsächlich Platzbedarf aufgrund der geclusterten Speicherung gegenüber:

```

001 /**
002  * Summiert einerseits die tatsächliche Größe aller
003  * Dateien und andererseits die Größe, die sie durch
004  * das Clustering mit der angegebenen Clustergröße
005  * belegen. Zusätzlich wird der durch das Clustering
006  * "verschwendete" Speicherplatz ausgegeben.
007  */
008 public static void clustering(int clustersize)
009 throws SQLException
010 {
011     int truesize = 0;
012     int clusteredsize = 0;
013     double wasted;
014     ResultSet rs = stmt.executeQuery(
015         "SELECT * FROM file"
016     );
017     while (rs.next()) {
018         int fsize = rs.getInt("fsize");
019         truesize += fsize;
020         if (fsize % clustersize == 0) {
021             clusteredsize += fsize;
022         } else {
023             clusteredsize += ((fsize / clustersize) + 1)*clustersize;
024         }
025     }
026     System.out.println("true size      = " + truesize);
027     System.out.println("clustered size = " + clusteredsize);
028     wasted = 100 * (1 - ((double)truesize / clusteredsize));
029     System.out.println("wasted space   = " + wasted + " %");
030 }

```

Listing 30.10: Cluster-Berechnung mit der DirDB-Datenbank

Um beispielsweise den Einfluß der geclusterten Darstellung bei einer Clustergröße von 8192 zu ermitteln, kann das Programm wie folgt aufgerufen werden:

Beispiel

```
java DirDB I clustering 8192
```

Die Ausgabe des Programms könnte dann beispielsweise so aussehen:

```
InstantDB - Version 1.85
Copyright (c) 1997-1998 Instant Computer Solutions Ltd.

Connection URL: jdbc:idb:dirdb.prp
Driver Name:    InstantDB JDBC Driver
Driver Version: Version 1.85

true size      = 94475195
clustered size = 112861184
wasted space   = 16.290799323884464 %
```

30.4 Weiterführende Themen

- [30.4 Weiterführende Themen](#)
 - [30.4.1 Metadaten](#)
 - [30.4.2 Escape-Kommandos](#)
 - [30.4.3 Transaktionen](#)
 - [30.4.4 JDBC-Datentypen](#)
 - [30.4.5 Umgang mit JDBC-Objekten](#)
 - [30.4.6 Prepared Statements](#)
 - [30.4.7 SQL-Kurzreferenz](#)
 - [Ändern von Datenstrukturen](#)
 - [Ändern von Daten](#)
 - [Lesen von Daten](#)

In diesem Abschnitt wollen wir eine Reihe von Themen ansprechen, die bei der bisherigen Darstellung zu kurz gekommen sind. Aufgrund des beschränkten Platzes werden wir jedes Thema allerdings nur kurz ansprechen und verweisen für genauere Informationen auf die JDBC-Beschreibung (sie ist Bestandteil der mit dem JDK 1.2 ausgelieferten Online-Dokumentation) und auf weiterführende Literatur zum Thema JDBC.

30.4.1 Metadaten

In [Abschnitt 30.3.3](#) sind wir bereits kurz auf die Verwendung von Metadaten eingegangen. Neben den Datenbankmetadaten gibt es die Methode `getMetaData` der Klasse `ResultSet`:

```

java.sql.ResultSet

ResultSetMetaData getMetaData()
```

Sie liefert ein Objekt vom Typ `ResultSetMetaData`, das Meta-Informationen über die Ergebnismenge zur Verfügung stellt. Wichtige Methoden sind:

```

java.sql.ResultSetMetaData

int getColumnCount()

String getColumnName(int column)

String getTableName(int column)

int getColumnType(int column)
```

Mit `getColumnCount` kann die Anzahl der Spalten in der Ergebnismenge abgefragt werden. `getColumnName` und `getTableName` liefern den Namen der Spalte bzw. den Namen der Tabelle, zu der diese Spalte in der Ergebnismenge gehört, wenn man ihren numerischen Index angibt. Mit `getColumnType` kann der Datentyp einer Spalte abgefragt werden. Als Ergebnis wird eine der statischen Konstanten aus der Klasse `java.sql.Types` zurückgegeben.

30.4.2 Escape-Kommandos

Mit den *Escape-Kommandos* wurde ein Feature eingeführt, das die Portierbarkeit von Datenbank Anwendungen verbessern soll. In Anlehnung an ODBC fordert die JDBC-Spezifikation dazu, daß die JDBC-Treiber in der Lage sein müssen, *besondere* Zeichenfolgen in SQL-Anweisungen zu erkennen und in die spezifische Darstellung der jeweiligen Datenbank zu übersetzen. Auf diese Weise können beispielsweise Datums- und Zeitlitterale portabel übergeben oder eingebaute Funktionen aufgerufen werden. Die Escape-Kommandos haben folgende Syntax:

```
"{ " <Kommandoname> [<Argumente>] "}"
```

Am Anfang steht eine geschweifte Klammer, dann folgen der Name des Escape-Kommandos und mögliche Argumente, und am Ende wird das Kommando durch eine weitere geschweifte Klammer abgeschlossen.

Um beispielsweise unabhängig von seiner konkreten Darstellung einen Datumswert einzufügen, kann das Escape-Kommando "d" verwendet werden. Es erwartet als Argument eine SQL-Zeichenkette im Format "yyyy-mm-dd" und erzeugt daraus das zur jeweiligen Datenbank passende Datumsliteral. In [Listing 30.5](#) haben wir dieses Kommando verwendet, um das Änderungsdatum der Datei in die Tabelle *file* zu schreiben.

Beispiel

30.4.3 Transaktionen

Die drei Methoden [commit](#), [rollback](#) und [setAutoCommit](#) des [Connection](#)-Objekts steuern das Transaktionsverhalten der Datenbank:

[java.sql.Connection](#)

```
void commit()
```

```
void rollback()
```

```
void setAutoCommit(boolean autoCommit)
```

Nach dem Aufbauen einer JDBC-Verbindung ist die Datenbank (gemäß JDBC-Spezifikation) zunächst im *Auto-Commit-Modus*. Dabei gilt jede einzelne Anweisung als separate Transaktion, die nach Ende des Kommandos automatisch bestätigt wird. Durch Aufruf von [setAutoCommit](#) und Übergabe von [false](#) kann das geändert werden. Danach müssen alle Transaktionen explizit durch Aufruf von [commit](#) bestätigt bzw. durch [rollback](#) zurückgesetzt werden. Nach dem Abschluß einer Transaktion beginnt automatisch die nächste.

Wichtig ist auch der *Transaction Isolation Level*, mit dem der Grad der Parallelität von Datenbanktransaktionen gesteuert wird. Je höher der Level, desto weniger Konsistenzprobleme können durch gleichzeitigen Zugriff mehrerer Transaktionen auf dieselben Daten entstehen. Umso geringer ist aber auch der Durchsatz bei einer großen Anzahl von gleichzeitigen Zugriffen. Transaction Isolation Levels werden von der Datenbank üblicherweise mit Hilfe von gemeinsamen und exklusiven Sperren realisiert. JDBC unterstützt die folgenden Levels:

- [Connection.TRANSACTION_NONE](#)
- [Connection.TRANSACTION_READ_UNCOMMITTED](#)
- [Connection.TRANSACTION_READ_COMMITTED](#)
- [Connection.TRANSACTION_REPEATABLE_READ](#)
- [Connection.TRANSACTION_SERIALIZABLE](#)

Mit Hilfe der beiden Methoden [getTransactionIsolation](#) und [setTransactionIsolation](#) des [Connection](#)-Objekts kann der aktuelle Transaction Isolation Level abgefragt bzw. verändert werden:

[java.sql.Connection](#)

```
int getTransactionIsolation()
```

```
void setTransactionIsolation(int level)
```

Mit der Methode [supportsTransactionIsolationLevel](#) des [DatabaseMetaData](#)-Objekts kann abgefragt werden, ob eine Datenbank einen bestimmten Transaction Isolation Level unterstützt oder nicht.

30.4.4 JDBC-Datentypen

In den meisten Fällen braucht man keine *exakte* Kenntnis des Datentyps einer Tabellenspalte, wenn man diese abfragt. Die oben beschriebenen [get](#)-Methoden des [ResultSet](#)-Objekts führen geeignete Konvertierungen durch. Soll dagegen mit [CREATE TABLE](#) eine neue Datenbank definiert werden, muß zu jeder Spalte der genaue Datentyp angegeben werden. Leider unterscheiden sich die Datenbanken bezüglich der unterstützten Typen erheblich, und die [CREATE TABLE](#)-Anweisung ist wenig portabel. Die Klasse [java.sql.Types](#) listet alle JDBC-Typen auf und gibt für jeden eine symbolische Konstante an. Mit der Methode [getTypeInfo](#) der Klasse [DatabaseMetaData](#) kann ein [ResultSet](#) mit allen Typen der zugrundeliegenden Datenbank und ihren spezifischen Eigenschaften beschafft werden. In [Tabelle 30.4](#) findet sich eine Übersicht der wichtigsten SQL-Datentypen.

30.4.5 Umgang mit JDBC-Objekten

Wie oben schon erwähnt, sind die JDBC-Objekte des Typs [Connection](#) und [Statement](#) möglicherweise kostspielig bezüglich ihres Rechenzeit- oder Speicherverbrauchs. Es empfiehlt sich daher, nicht unnötig viele von ihnen anzulegen.

Während das bei [Connection](#)-Objekten einfach ist, kann es bei [Statement](#)-Objekten unter Umständen problematisch werden. Wird beispielsweise in einer Schleife mit vielen Durchläufen immer wieder eine Methode aufgerufen, die eine Datenbankabfrage durchführt, so stellt sich die Frage, woher sie das dafür erforderliche [Statement](#)-Objekt nehmen soll. Wird es jedesmal lokal angelegt, kann schnell der Speicher knapp werden. Wird es dagegen als statische oder als Klassenvariable angelegt, kann es zu Konflikten mit konkurrierenden Methoden kommen (die üblichen Probleme globaler Variablen).

Eine gut funktionierende Lösung für dieses Problem besteht darin, [Statement](#)-Objekte auf der [Connection](#) zu *cachen*, also

zwischenzuspeichern. Das kann etwa mit einer Queue erfolgen, in die nicht mehr benötigte [Statement](#)-Objekte eingestellt werden. Anstelle eines Aufrufs von [createStatement](#) wird dann zunächst in der Queue nachgesehen, ob ein recyclebares Objekt vorhanden ist und dieses gegebenenfalls wiederverwendet. Es sind dann zu keinem Zeitpunkt mehr [Statement](#)-Objekte angelegt, als *parallel* benötigt werden. Natürlich dürfen nur Objekte in die Queue gestellt werden, die nicht mehr benötigt werden; ihr [ResultSet](#) sollte also vorher möglichst geschlossen werden. Das [Statement](#)-Objekt selbst darf nicht geschlossen werden, wenn es noch verwendet werden soll.

Ein einfache Implementierung wird in dem folgendem Listing vorgestellt. Das Objekt vom Typ [CachedConnection](#) wird mit einem [Connection](#)-Objekt instanziiert. Die Methoden [getStatement](#) und [releaseStatement](#) dienen dazu, [Statement](#)-Objekte zu beschaffen bzw. wieder freizugeben. Die Klasse [LinkedQueue](#) wurde in [Abschnitt 27.4](#) vorgestellt.

[CachedConnection.java](#)

```
001 /* CachedConnection.java */
002
003 import java.sql.*;
004 import java.util.*;
005
006 public class CachedConnection
007 {
008     private Connection con;
009     private LinkedQueue cache;
010     private int stmtcnt;
011
012     public CachedConnection(Connection con)
013     {
014         this.con = con;
015         this.cache = new LinkedQueue();
016         this.stmtcnt = 0;
017     }
018
019     public Statement getStatement()
020     throws SQLException
021     {
022         if (cache.size() <= 0) {
023             return con.createStatement();
024         } else {
025             return (Statement)cache.retrieve();
026         }
027     }
028
029     public void releaseStatement(Statement statement)
030     {
031         cache.add(statement);
032     }
033 }
```

Listing 30.11: Die Klasse [CachedConnection](#)

Es ist wichtig, die JDBC-Objekte auch dann zu schließen, wenn eine Ausnahme während der Bearbeitung aufgetreten ist. Andernfalls würden möglicherweise Ressourcen nicht freigegeben und das Programm würde so nach und nach mehr Speicher oder Rechenzeit verbrauchen. Am einfachsten kann dazu die [finally](#)-Klausel der [try-catch](#)-Anweisung verwendet werden.

Eine weitere Eigenschaft der Klasse [ResultSet](#) verdient besondere Beachtung. Bei manchen JDBC-Treibern erlauben die zurückgegebenen [ResultSet](#)-Objekte das Lesen einer bestimmten Tabellenspalte nur einmal. Der zweite Versuch wird mit einer Fehlermeldung "No data" (oder ähnlich) quittiert. Manche Treiber erfordern sogar, daß die Spalten des [ResultSet](#) in der Reihenfolge ihrer Definition gelesen werden. In beiden Fällen ist es gefährlich, einen [ResultSet](#) als Parameter an eine Methode zu übergeben, denn die Methode weiß nicht, welche Spalten bereits vom Aufrufer gelesen wurden und umgekehrt. Eine Lösung könnte darin bestehen, einen [ResultSet](#) mit integriertem Cache zu entwickeln, der sich bereits gelesene Spaltenwerte merkt. Alternativ könnte man auch einen objekt-relationalen Ansatz versuchen, bei dem jeder gelesene Satz der Ergebnismenge direkt ein passendes Laufzeitobjekt erzeugt, das dann beliebig oft gelesen werden kann. Wir wollen auf beide Varianten an dieser Stelle nicht weiter eingehen.

Warnung

30.4.6 Prepared Statements

Prepared Statements sind parametrisierte SQL-Anweisungen. Sie werden zunächst deklariert und zum Vorkompilieren an die Datenbank übergeben. Später können sie dann beliebig oft ausgeführt werden, indem die formalen Parameter durch aktuelle Werte ersetzt werden und die so parametrisierte Anweisung an die Datenbank übergeben wird. Der Vorteil von Prepared Statements ist, daß die Vorbereitungsarbeiten nur einmal erledigt werden müssen (Syntaxanalyse, Vorbereitung der Abfragestrategie und -optimierung) und die tatsächliche Abfrage dann wesentlich schneller ausgeführt werden kann. Das bringt Laufzeitvorteile bei der wiederholten Ausführung der vorkompilierten Anweisung.

JDBC stellt Prepared Statements mit dem Interface [PreparedStatement](#), das aus [Statement](#) abgeleitet ist, zur Verfügung. Die Methode [prepareStatement](#) des [Connection](#)-Objekts liefert ein [PreparedStatement](#):

```
public PreparedStatement prepareStatement(String sql)
    throws SQLException
```

Als Argument wird ein String übergeben, der die gewünschte SQL-Anweisung enthält. Die formalen Parameter werden durch Fragezeichen dargestellt. Bei den meisten Datenbanken dürfen sowohl Änderungs- als auch Abfrageanweisungen vorkompiliert werden. Sie werden dann später mit [executeQuery](#) bzw. [executeUpdate](#) ausgeführt. Anders als im Basisinterface sind diese Methoden im Interface [PreparedStatement](#) parameterlos:

```
public ResultSet executeQuery()
    throws SQLException

public int executeUpdate()
    throws SQLException
```

Bevor eine dieser Methoden aufgerufen werden darf, ist es erforderlich, die vorkompilierte Anweisung zu parametrisieren. Dazu muß für jedes Fragezeichen eine passende [set](#)-Methode aufgerufen und das gewünschte Argument übergeben werden. Die [set](#)-Methoden gibt es für alle JDBC-Typen (siehe beispielsweise die analoge Liste der [get](#)-Methoden in [Tabelle 30.1](#)):

```
public void setBoolean(int parameterIndex, boolean x)
    throws SQLException

public void setByte(int parameterIndex, byte x)
    throws SQLException

...

```

Der erste Parameter gibt die Position des Arguments in der Argumentliste an. Das erste Fragezeichen hat den Index 1, das zweite den Index 2 usw. Der zweite Parameter liefert den jeweiligen Wert, der anstelle des Fragezeichens eingesetzt werden soll.

Als Beispiel wollen wir uns eine abgewandelte Form der in [Abschnitt 30.3.5](#) vorgestellten Methode [countRecords](#) ansehen, bei der anstelle eines [Statement](#)-Objekts ein [PreparedStatement](#) verwendet wird:

```
001 public static void countRecords()
002     throws SQLException
003 {
004     PreparedStatement pstmt = con.prepareStatement(
005         "SELECT count(*) FROM ?"
006     );
007     String aTables[] = {"dir", "file"};
008     for (int i = 0; i < aTables.length; ++i) {
009         pstmt.setString(1, aTables[i]);
010         ResultSet rs = pstmt.executeQuery();
011         if (!rs.next()) {
012             throw new SQLException("SELECT COUNT(*): no result");
013         }
014         System.out.println(aTables[i] + ": " + rs.getInt(1));
015     }
016     pstmt.close();
017 }

```

Listing 30.12: Verwenden eines PreparedStatement

Das [PreparedStatement](#) enthält hier den Namen der Tabelle als Parameter. In einer Schleife nehmen wir nun für die Tabellen "dir" und "file" jeweils eine Parametrisierung vor und führen dann die eigentliche Abfrage durch. Der Rückgabewert von [executeQuery](#) entspricht dem der

Beispiel

Basisklasse, so daß der folgende Code sich prinzipiell nicht von dem in [Abschnitt 30.3.5](#) unterscheidet.

Nicht alle Datenbanken erlauben es, *Tabellennamen* zu parametrisieren, sondern beschränken diese Fähigkeit auf Argumente von Such- oder Änderungsausdrücken. Tatsächlich läuft unser Beispiel zwar mit InstantDB, aber beispielsweise nicht mit MS Access 95.

Warnung

30.4.7 SQL-Kurzreferenz

Dieser Abschnitt gibt eine kurze Übersicht der gebräuchlichsten SQL-Anweisungen in ihren grundlegenden Ausprägungen. Er ersetzt weder ein Tutorial noch eine Referenz und ist zu keinem der bekannten SQL-Standards vollständig kompatibel. Trotzdem mag er für einfache Experimente nützlich sein und helfen, die ersten JDBC-Anbindungen zum Laufen zu bringen. Für "ernsthafte" Datenbank Anwendungen sollte zusätzliche Literatur konsultiert und dabei insbesondere auf die Spezialitäten der verwendeten Datenbank geachtet werden.

Die nachfolgenden Syntaxbeschreibungen sind an die bei SQL-Anweisungen übliche Backus-Naur-Form angelehnt:

- Terminalsymbole (Schlüsselworte) sind groß geschrieben.
- Weitere Terminalsymbole sind die runden Klammern, das Komma und die in den Suchausdrücken verwendeten Operatoren.
- Nichtterminalsymbole sind kursiv geschrieben.
- Optionale Teile stehen in eckigen Klammern.
- Der senkrechte Strich trennt Alternativen. Stehen sie in eckigen Klammern, können sie auch ganz weggelassen werden. Stehen sie in geschweiften Klammern, muß genau eine der Alternativen verwendet werden.
- Drei Punkte zeigen an, daß die davor stehende Anweisungsfolge wiederholt werden kann.

Ändern von Datenstrukturen

Mit [CREATE TABLE](#) kann eine neue Tabelle angelegt werden. Mit [DROP TABLE](#) kann sie gelöscht und mit [ALTER TABLE](#) ihre Struktur geändert werden. Mit [CREATE INDEX](#) kann ein neuer Index angelegt werden, mit [DROP INDEX](#) kann er wieder gelöscht werden.

```
CREATE TABLE TabName
  (ColName DataType [DEFAULT ConstExpr]
   [ColName DataType [DEFAULT ConstExpr]]...)
```

```
ALTER TABLE TabName
  ADD (ColName DataType
       [ColName DataType]....)
```

```
CREATE [UNIQUE] INDEX IndexName
  ON TabName
  (ColName [ASC|DESC]
   [, ColName [ASC|DESC]]...)
```

```
DROP TABLE TabName
```

```
DROP INDEX IndexName
```

TabName, *ColName* und *IndexName* sind SQL-Bezeichner. *ConstExpr* ist ein konstanter Ausdruck, der einen Standardwert für eine Spalte vorgibt. *DataType* gibt den Datentyp der Spalte an, die gebräuchlichsten von ihnen können [Tabelle 30.4](#) entnommen werden.

Bezeichnung	Bedeutung
CHAR(n)	Zeichenkette der (festen) Länge <i>n</i> .
VARCHAR(n)	Zeichenkette variabler Länge mit max. <i>n</i> Zeichen.
SMALLINT	16-Bit-Ganzzahl mit Vorzeichen.
INTEGER	32-Bit-Ganzzahl mit Vorzeichen.
REAL	Fließkommazahl mit etwa 7 signifikanten Stellen.
FLOAT	Fließkommazahl mit etwa 15 signifikanten Stellen. Auch als DOUBLE oder DOUBLE PRECISION bezeichnet.
DECIMAL(n,m)	Festkommazahl mit <i>n</i> Stellen, davon <i>m</i> Nachkommastellen. Ähnlich NUMERIC.
DATE	Datum (evtl. mit Uhrzeit). Verwandte Typen sind TIME und TIMESTAMP.

Tabelle 30.4: SQL-Datentypen

Ändern von Daten

Ein neuer Datensatz kann mit [INSERT INTO](#) angelegt werden. Soll ein bestehender Datensatz geändert werden, ist dazu [UPDATE](#) zu verwenden. Mit [DELETE FROM](#) kann er gelöscht werden.

```
INSERT INTO TabName
  [( ColName [,ColName] )]
VALUES (Expr [,Expr]...)

UPDATE TabName
  SET ColName = {Expr|NULL}
    [,ColName = {Expr|NULL}]...
  [WHERE SearchCond]

DELETE FROM TabName
  [WHERE SearchCond]
```

TabName und *ColName* sind die Bezeichner der gewünschten Tabelle bzw. Spalte. *Expr* kann eine literale Konstante oder ein passender Ausdruck sein. *SearchCond* ist eine Suchbedingung, mit der angegeben wird, auf welche Sätze die [UPDATE](#)- oder [DELETE FROM](#)-Anweisung angewendet werden soll. Wird sie ausgelassen, wirken die Änderungen auf alle Sätze. Wir kommen im nächsten Abschnitt auf die Syntax der Suchbedingung zurück. Wird bei der [INSERT INTO](#)-Anweisung die optionale Feldliste ausgelassen, müssen Ausdrücke für *alle* Felder angegeben werden.

Lesen von Daten

Das Lesen von Daten erfolgt mit der [SELECT](#)-Anweisung. Ihre festen Bestandteile sind die Liste der Spalten *ColList* und die Liste der Tabellen, die in der Abfrage verwendet werden sollen. Daneben gibt es eine Reihe von optionalen Bestandteilen:

```
SELECT [ALL|DISTINCT] ColList
FROM  TabName [,TabName]...
[WHERE SearchCond]
[GROUP BY ColName [,ColName]...]
[HAVING SearchCond]
[UNION SubQuery]
[ORDER BY ColName [ASC|DESC]
          [,ColName [ASC|DESC]]...]


```

Die Spaltenliste kann entweder einzelne Felder aufzählen, oder es können durch Angabe eines Sternchens "*" alle Spalten angegeben werden. Wurde mehr als eine Tabelle angegeben und sind die Spaltennamen nicht eindeutig, kann ein Spaltenname durch Voranstellen des Tabellennamens und eines Punkts qualifiziert werden. Zusätzlich können die Spaltennamen mit dem Schlüsselwort "AS" ein (möglicherweise handlicheres) Synonym erhalten. Die Syntax von *ColList* ist:

```
ColExpr [AS ResultName]
[,ColExpr AS ResultName]]...
```

Zusätzlich gibt es einige numerische Aggregatfunktionen, mit denen der Wert der als Argument angegeben Spalte über alle Sätze der Ergebnismenge kumuliert werden kann:

Bezeichnung	Bedeutung
COUNT	Anzahl der Sätze
AVG	Durchschnitt
SUM	Summe
MIN	Kleinster Wert
MAX	Größter Wert

Tabelle 30.5: SQL-Aggregatfunktionen

Die [WHERE](#)-Klausel definiert die Suchbedingung. Wurde sie nicht angegeben, liefert die Anweisung alle vorhandenen Sätze. Der Suchausdruck *SearchCond* kann sehr unterschiedliche Formen annehmen. Zunächst kann eine Spalte mit Hilfe der relationalen Operatoren <, <=, >, >=, = und <> mit einer anderen Spalte oder einem Ausdruck verglichen werden. Die Teilausdrücke können mit den logischen Operatoren [AND](#), [OR](#) und [NOT](#) verknüpft werden, die Auswertungsreihenfolge kann in der üblichen Weise durch Klammerung gesteuert werden.

Mit Hilfe des Schlüsselworts [LIKE](#) kann eine Ähnlichkeitssuche durchgeführt werden:

Expr LIKE Pattern

Mit Hilfe der Wildcards "%" und "_" können auch unscharf definierte Begriffe gesucht werden. Jedes Vorkommen von "%" paßt auf eine beliebige Anzahl beliebiger Zeichen, jedes "_" steht für genau ein beliebiges Zeichen. Manche Datenbanken unterscheiden zwischen Groß- und Kleinschreibung, andere nicht.

Mit Hilfe der Klauseln IS NULL und IS NOT NULL kann getestet werden, ob der Inhalt einer Spalte den Wert NULL enthält oder nicht:

ColName IS [NOT] NULL

Mit dem BETWEEN-Operator kann bequem festgestellt werden, ob ein Ausdruck innerhalb eines vorgegebenen Wertebereichs liegt oder nicht:

Expr BETWEEN Expr AND Expr

Neben den einfachen Abfragen gibt es eine Reihe von Abfragen, die mit *Subqueries* (Unterabfragen) arbeiten:

EXISTS (SubQuery)

Expr [NOT] IN (SubQuery)

Expr RelOp {ALL|ANY} (SubQuery)

Die Syntax von *SubQuery* entspricht der einer normalen SELECT-Anweisung. Sie definiert eine separat definierte Menge von Daten, die als Teilausdruck in einer Suchbedingung angegeben wird. Der EXISTS-Operator testet, ob die Unterabfrage mindestens ein Element enthält. Mit dem IN-Operator wird getestet, ob der angegebene Ausdruck in der Ergebnismenge enthalten ist. Die Ergebnismenge kann auch literal als komma-separierte Liste von Werten angegeben werden. Schließlich kann durch Angabe eines relationalen Operators getestet werden, ob zu mindestens einem (ANY) oder allen (ALL) Sätzen der Unterabfrage in der angegebenen Beziehung steht. Bei den beiden letzten Unterabfragen sollte jeweils nur eine einzige Spalte angegeben werden.

Die GROUP BY-Klausel dient dazu, die Sätze der Ergebnismenge zu Gruppen zusammenzufassen, bei denen die Werte der angegebenen Spalten gleich sind. Sie wird typischerweise zusammen mit den oben erwähnten Aggregatfunktionen verwendet. Mit HAVING kann zusätzlich eine Bedingung angegeben werden, mit der die gruppierten Ergebnissätze "nachgefiltert" werden.

Mit dem UNION-Operator können die Ergebnismengen zweier SELECT-Anweisungen zusammengefaßt werden. Das wird typischerweise gemacht, wenn die gesuchten Ergebnissätze aus mehr als einer Tabelle stammen (andernfalls könnte der OR-Operator verwendet werden).

Die ORDER BY-Klausel kann angegeben werden, um die Reihenfolge der Sätze in der Ergebnismenge festzulegen. Die Sätze werden zunächst nach der ersten angegebenen Spalte sortiert, bei Wertegleichheit nach der zweiten, der dritten usw. Mit Hilfe der Schlüsselworte ASC und DESC kann angegeben werden, ob die Werte auf- oder absteigend sortiert werden sollen.

30.5 Zusammenfassung

- [30.5 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Die grundsätzliche Arbeitsweise und wichtige Architekturmerkmale von JDBC
- Laden des JDBC-Treibers
- Zugriff auf MS Access und InstantDB
- Verwenden der JDBC-ODBC-Bridge
- Aufbau einer Datenbankverbindung mit den Klassen [DriverManager](#) und [Connection](#)
- Erzeugen von Anweisungobjekten
- Ausführen von Datenbankabfragen
- Abfrage von Meta-Informationen
- Der Umgang mit der Klasse [ResultSet](#) und ihre Besonderheiten
- Die JDBC-Datentypen
- Umgang mit NULL-Werten
- Ändern, Löschen und Einfügen von Daten
- Anlegen und Löschen von Tabellen und Indexdateien
- Escape-Kommandos
- Transaktionen
- Der Umgang mit Prepared Statements
- Die wichtigsten SQL-Befehle, -Datentypen und -ausdrücke

Kapitel 31

Reflection

- [31 Reflection](#)
 - [31.1 Einleitung](#)
 - [31.2 Die Klassen Object und Class](#)
 - [31.2.1 Die Klasse Object](#)
 - [31.2.2 Die Klasse Class](#)
 - [31.3 Methoden- und Konstruktorenaufrufe](#)
 - [31.3.1 Parameterlose Methoden](#)
 - [31.3.2 Parametrisierte Methoden](#)
 - [Funktionszeiger](#)
 - [31.3.3 Parametrisierte Konstruktoren](#)
 - [31.4 Zugriff auf Membervariablen](#)
 - [31.5 Zusammenfassung](#)

31.1 Einleitung

- [31.1 Einleitung](#)

Bei der Entwicklung des JDK 1.1 sahen sich die Entwickler mit einer Schwäche von Java konfrontiert, die die Entwicklung bestimmter Typen von Tools und Bibliotheken unmöglich machte: der statischen Struktur von Klassen und Objekten. Um ein Objekt anzulegen, eine seiner Methoden aufzurufen oder auf eine seiner Membervariablen zuzugreifen, mußte der Code der Klasse zur *Compilezeit* bekannt sein. Während dies für die meisten gewöhnlichen Anwendungen kein Problem darstellt, ist es für die Entwicklung generischer Werkzeuge und hochkonfigurierbarer Anwendungen, die durch PlugIns erweiterbar sind, unzureichend. Insbesondere die Entwicklung der Beans- und Serialisierungs-APIs war mit den in der Version 1.0 verfügbaren Spracheigenschaften nicht möglich.

Benötigt wurde vielmehr die Möglichkeit, Klassen zu laden und zu instanzieren (auch mit parametrisierten Konstruktoren), ohne daß bereits zur Compilezeit ihr Name bekannt sein mußte. Weiterhin sollten statische oder instanzbasierte Methoden aufgerufen und auf Membervariablen auch dann zugegriffen werden können, wenn ihr Name erst zur Laufzeit des Programmes bekannt ist.

Gesucht wurde also ein Mechanismus, der diese normalerweise vom Compiler angeforderten Fähigkeiten des Laufzeitsystems auch "normalen" Anwendungen zur Verfügung stellte. Mit dem Reflection-API des JDK 1.1 wurde eine Library-Schnittstelle geschaffen, die alle erwähnten Fähigkeiten (und noch einige mehr) implementiert und beliebigen Anwendungen als integralen Bestandteil der Java-Klassenbibliothek zur Verfügung stellt. Erweiterungen am Sprachkern waren dazu nicht nötig.

Wir wollen uns in diesem Kapitel die wichtigsten Eigenschaften des Reflection-APIs ansehen und ein paar nützliche Anwendungen vorstellen. Die als *Introspection* bezeichneten Erweiterungen für die Beans-Behandlung und den Zugriff auf Arrays via Reflection wollen wir bei unseren Betrachtungen ausklammern.

31.2 Die Klassen Object und Class

- [31.2 Die Klassen Object und Class](#)
 - [31.2.1 Die Klasse Object](#)
 - [31.2.2 Die Klasse Class](#)

31.2.1 Die Klasse Object

Die Klasse [Object](#) ist die Superklasse aller anderen Klassen. Jede Klasse, die keine [extends](#)-Klausel besitzt, wird direkt aus [Object](#) abgeleitet. Jede abgeleitete Klasse stammt schließlich von einer nicht mehr weiter abgeleiteten Klasse ab und ist damit letztlich selbst aus [Object](#) abgeleitet.

Die Klasse [Object](#) definiert einige elementare Methoden, die für alle Arten von Objekten nützlich sind:

```

boolean equals(Object obj)
protected Object clone()
String toString()
int hashCode()

```

Die Methode [equals](#) testet, ob zwei Objekte denselben Inhalt haben, [clone](#) kopiert ein Objekt, [toString](#) erzeugt eine [String](#)-Repräsentation des Objekts, und [hashCode](#) berechnet einen numerischen Wert, der als Schlüssel zur Speicherung eines Objekts in einer [Hashtable](#) verwendet werden kann. Damit diese Methoden in abgeleiteten Klassen vernünftig funktionieren, müssen sie bei Bedarf überlagert werden. Für [equals](#) und [clone](#) gilt das insbesondere, wenn das Objekt Referenzen enthält.

31.2.2 Die Klasse Class

Mit der Methode [getClass](#) der Klasse [Object](#) besitzt ein beliebiges Objekt die Fähigkeit, ein passendes *Klassenobjekt* zu liefern. Zu jeder Klasse, die das Laufzeitsystem verwendet, wird während des Ladevorgangs ein Klassenobjekt vom Typ [Class](#) erzeugt. Die Klasse [Class](#) stellt Methoden zur Abfrage von Eigenschaften der Klasse zur Verfügung und erlaubt es, Klassen dynamisch zu laden und Instanzen dynamisch zu erzeugen. Darüber hinaus ist sie der Schlüssel zur Funktionalität des Reflection-APIs.

Wir wollen uns zunächst an einem einfachen Beispiel das dynamische Laden und Instanzieren von Klassen ansehen. Das folgende Listing zeigt das Interface [HelloMeth](#) und die Klassen [CA](#), [CB](#), [CC](#) und [CD](#). [HelloMeth](#) deklariert die Methode [hello](#), die von den Klassen [CA](#) und [CB](#) implementiert wird. [CC](#) besitzt ebenfalls die Methode [hello](#), allerdings ohne das Interface [HelloMeth](#) zu implementieren. [CD](#) schließlich implementiert nicht [hello](#), sondern [hallo](#).

Beispiel

Die Hauptklasse liest zunächst einen Klassennamen von der Standardeingabe ein. Mit der Klassenmethode [forName](#) der Klasse [Class](#) wird dann ein Klassenobjekt zu einer Klasse dieses Namens beschafft. Das wird verwendet, um mit der Methode [newInstance](#) der Klasse [Class](#) ein neues Objekt zu erzeugen. Dieses Objekt wird schließlich in das Interface [HelloMeth](#) konvertiert und dessen Methode [hello](#) aufgerufen.

Das Programm ist in der Lage, die beiden Klassen [CA](#) und [CB](#) ordnungsgemäß zu instanzieren und ihre Methode [hello](#) aufzurufen. Bei [CC](#) und [CD](#) gibt es eine Ausnahme des Typs [ClassCastException](#), weil diese Klassen nicht das Interface [HelloMeth](#) implementieren. Alle anderen Klassennamen werden mit der Ausnahme [ClassNotFoundException](#) quittiert.

```

001 /* Listing3101.java */
002
003 import java.io.*;
004
005 interface HelloMeth
006 {
007     public void hello();
008 }
009
010 class CA
011 implements HelloMeth
012 {
013     public void hello()
014     {
015         System.out.println("hello CA");
016     }
017 }
018
019 class CB
020 implements HelloMeth
021 {
022     public void hello()
023     {
024         System.out.println("hello CB");
025     }
026 }
027
028 class CC
029 {
030     public void hello()
031     {
032         System.out.println("hello CC");
033     }
034 }
035
036 class CD
037 {
038     public void hallo()
039     {
040         System.out.println("hallo CD");
041     }
042 }
043
044 public class Listing3101
045 {
046     public static void main(String[] args)
047     {
048         String buf = "";
049         BufferedReader in = new BufferedReader(
050             new InputStreamReader(
051                 new DataInputStream(System.in)));
052         while (true) {
053             try {
054                 System.out.print("Klassenname oder ende eingeben: ");
055                 buf = in.readLine();
056                 if (buf.equals("ende")) {
057                     break;
058                 }
059                 Class c = Class.forName(buf);
060                 Object o = c.newInstance();
061                 ((HelloMeth)o).hello();
062             } catch (IOException e) {
063                 System.out.println(e.toString());
064             } catch (ClassNotFoundException e) {

```

```
065         System.out.println("Klasse nicht gefunden");
066     } catch (ClassCastException e) {
067         System.out.println(e.toString());
068     } catch (InstantiationException e) {
069         System.out.println(e.toString());
070     } catch (IllegalAccessException e) {
071         System.out.println(e.toString());
072     }
073 }
074 }
075 }
```

Listing 31.1: Dynamisches Laden von Klassen

Eine Beispielsitzung des Programms könnte so aussehen:

```
CA
hello CA
CB
hello CB
CC
java.lang.ClassCastException
CD
java.lang.ClassCastException
CE
Klasse nicht gefunden
ende
```

An diesem Beispiel ist zu sehen, wie Klassen geladen und instanziiert werden können, deren Name zur Compilezeit nicht bekannt ist. In allen anderen Beispielen in diesem Buch wurden Klassennamen als literale Konstanten im Sourcecode gehalten und der Compiler konnte den passenden Code erzeugen. Wir wollen die nötigen Schritte noch einmal zusammenfassen:

- Mit der statischen Methode `forName` wurde der `ClassLoader` beauftragt, eine Klasse des angegebenen Namens zu suchen und in die Java-Maschine zu laden. Die VM führt anschließend die statischen Initialisierungen aus (falls es nicht schon ein Klassenobjekt gibt) und liefert das zugehörige Klassenobjekt.
- Mit dem Klassenobjekt können nun verschiedene Aufgaben erledigt werden. Insbesondere ist es möglich, Informationen über Konstruktoren, Membervariablen oder Methoden der Klasse abzufragen.
- Zudem kann mit `newInstance` eine neue Instanz der Klasse angelegt werden. Ist der Typ der Klasse oder ihrer Basisklasse oder eines ihrer Interfaces bekannt, kann das Objekt auf eine Variable dieses Typs konvertiert und dessen Methoden aufgerufen werden.

31.3 Methoden- und Konstruktoreaufrufe

- [31.3 Methoden- und Konstruktoreaufrufe](#)
 - [31.3.1 Parameterlose Methoden](#)
 - [31.3.2 Parametrisierte Methoden](#)
 - [Funktionszeiger](#)
 - [31.3.3 Parametrisierte Konstruktoren](#)

31.3.1 Parameterlose Methoden

Wir wollen uns ein erstes Beispiel für die Anwendung des Reflection-APIs ansehen. In der Praxis stellt sich immer wieder das Problem, wohin bei neu entwickelten Klassen der Code zum *Testen* der Klasse geschrieben werden soll. Ein Weg besteht darin, an das Ende der Klasse eine Methode `public static void main` zu hängen und den Testcode dort zu plazieren:

```

001 public class Queue
002 {
003     //...
004     //Implementierung der Queue
005     //...
006
007     //---Testcode-----
008     public static void main(String args[])
009     {
010         Queue q = new Queue();
011         //...
012         //Code zum Testen der Queue
013         //...
014     }
015 }
```

Listing 31.2: Testcode in der main-Methode

Auf diese Weise läßt sich der Testcode einer Dienstleistungsklasse - wie beispielsweise einer Queue - ganz einfach mit dem Java-Interpreter aufrufen und reproduzierbar testen. Nachteilig ist natürlich, daß der eigentliche Code und der Testcode vermisch werden. Dadurch wird die Klassendatei unnötig groß, was im Hinblick auf gute Downloadzeiten nicht wünschenswert ist. Besser wäre es, wenn der Testcode in einer separaten Klasse verbleiben würde. Wir wollen dazu ein kleines Programm zum Testen von Java-Klassen schreiben. Es soll folgende Eigenschaften besitzen:

- Das Testing wird durch ein Rahmenprogramm `Test` ausgelöst, das als Argument den Namen der zu testenden Klasse erwartet.
- Die zu testende Klasse soll automatisch gefunden, geladen und mit dem Default-Konstruktor instanziiert werden.
- In der Klasse mit dem Testcode sollen automatisch alle öffentlichen, nicht-statischen Methoden ausgeführt werden, deren Name mit "test" beginnt.

Der Schlüssel zur Implementierung der Klasse `Test` liegt in der Anwendung des Reflection-APIs. Das Laden der Testklasse entspricht dem vorigen Beispiel, zum Aufzählen aller Methoden bedienen wir uns der Methode `getMethods` der Klasse `Class`:

```

public Method[] getMethods()
    throws SecurityException
    java.lang.Class
```

`getMethods` liefert ein Array von Objekten des Typs `Method`, das für jede öffentliche Methode der Klasse ein Element enthält. Um auch die nicht-öffentlichen Methoden aufzulisten, kann die Methode `getDeclaredMethods` verwendet werden. Die Klasse `Method` stellt einige Methoden zum Zugriff auf das Methodenobjekt zur Verfügung. Die wichtigsten sind:

```

String getName()
    java.lang.reflect.Method

int getModifiers()

Class[] getParameterTypes()

Object invoke(Object obj, Object[] args)
```

Mit `getName` kann der Name der Methode ermittelt werden. `getModifiers` liefert eine bitverknüpfte Darstellung der Methodenattribute (`static`, `private` usw.). Der Rückgabewert kann an die statischen Methoden der Klasse `Modifier` übergeben werden, um festzustellen, welche Attribute die Methode besitzt:

`java.lang.reflect.Modifier`

```
static boolean isAbstract(int mod)
static boolean isExplicit(int mod)
static boolean isFinal(int mod)
static boolean isInterface(int mod)
static boolean isNative(int mod)
static boolean isPrivate(int mod)
static boolean isProtected(int mod)
static boolean isPublic(int mod)
static boolean isStatic(int mod)
static boolean isStrict(int mod)
static boolean isSynchronized(int mod)
static boolean isTransient(int mod)
static boolean isVolatile(int mod)
```

`getParameterTypes` liefert ein Array mit Objekten des Typs `Class`, das dazu verwendet werden kann, die Anzahl und Typen der formalen Argumente der Methode festzustellen. Jedes Arrayelement repräsentiert dabei die Klasse des korrespondierenden formalen Arguments. Hat das Array die Länge 0, so ist die Methode parameterlos. Gibt es beispielsweise zwei Elemente mit den Typen `String` und `Double`, so besitzt die Methode zwei Parameter, die vom Typ `String` und `Double` sind.

Um auch primitive Typen auf diese Weise darstellen zu können, gibt es in den Wrapper-Klassen der primitiven Typen (siehe [Abschnitt 7.8.1](#)) jeweils ein statisches `Class`-Objekt mit der Bezeichnung `TYPE`, das den zugehörigen primitiven Datentyp bezeichnet. Ein `int`-Argument wird also beispielsweise dadurch angezeigt, daß der Rückgabewert von `getParameterTypes` an der entsprechenden Stelle ein Objekt des Typs `Integer.TYPE` enthält. Insgesamt gibt es neun derartige Klassenobjekte für die acht primitiven Typen und den Rückgabewert `void`:

Klassenobjekt	Typ
<code>java.lang.Boolean.TYPE</code>	<code>boolean</code>
<code>java.lang.Character.TYPE</code>	<code>char</code>
<code>java.lang.Byte.TYPE</code>	<code>byte</code>
<code>java.lang.Short.TYPE</code>	<code>short</code>
<code>java.lang.Integer.TYPE</code>	<code>int</code>
<code>java.lang.Long.TYPE</code>	<code>long</code>
<code>java.lang.Float.TYPE</code>	<code>float</code>
<code>java.lang.Double.TYPE</code>	<code>double</code>
<code>java.lang.Void.TYPE</code>	<code>void</code>

Tabelle 31.1: Klassenobjekte für die primitiven Typen

Die Methode `invoke` der Klasse `Method` dient dazu, die durch dieses Methodenobjekt repräsentierte Methode tatsächlich aufzurufen. Das erste Argument `obj` gibt dabei das Objekt an, auf dem die Methode ausgeführt werden soll. Es muß natürlich zu einem Objekt der Klasse gehören, auf der `getMethods` aufgerufen wurde. Das zweite Argument übergibt die aktuellen Parameter an die Methode. Ähnlich wie bei `getParameterTypes` wird auch hier ein Array angegeben, dessen Elemente den korrespondierenden aktuellen Argumenten entsprechen. Bei Objektparametern ist einfach ein Objekt des passenden Typs an der gewünschten Stelle zu platzieren.

Besitzt die Methode auch primitive Argumente, wird eine automatische Konvertierung vorgenommen (*unwrapping*), indem das entsprechende Arrayelement in den passenden primitiven Datentyp konvertiert wird. Erwartet die Methode beispielsweise ein `int`, so ist ein `Integer`-Objekt zu übergeben, das dann beim Aufruf automatisch "ausgepackt" wird.

Tip

Die Implementierung der Klasse `Test` sieht so aus:

```

001 /* Test.java */
002
003 import java.lang.reflect.*;
004
005 public class Test
006 {
007     public static Object createTestObject(String name)
008     {
009         //Klassennamen zusammenbauen
010         int pos = name.lastIndexOf('.');
011         if (pos == -1) {
012             name = "Test" + name;
013         } else {
014             name = name.substring(0, pos + 1) + "Test" +
015                 name.substring(pos + 1);
016         }
017         //Klasse laden
018         Object ret = null;
019         try {
020             Class testclass = Class.forName(name);
021             //Testobjekt instanzieren
022             System.out.println("=====");
023             System.out.println("Instanzieren von: " + name);
024             System.out.println("--");
025             ret = testclass.newInstance();
026         } catch (ClassNotFoundException e) {
027             System.err.println("Kann Klasse nicht laden: " + name);
028         } catch (InstantiationException e) {
029             System.err.println("Fehler beim Instanzieren: " + name);
030         } catch (IllegalAccessException e) {
031             System.err.println("Unerlaubter Zugriff auf: " + name);
032         }
033         return ret;
034     }
035
036     public static void runTests(Object tester)
037     {
038         Class clazz = tester.getClass();
039         Method methods[] = clazz.getMethods();
040         int cnt = 0;
041         for (int i = 0; i < methods.length; ++i) {
042             //Methodenname muß mit "test" anfangen
043             String name = methods[i].getName();
044             if (!name.startsWith("test")) {
045                 continue;
046             }
047             //Methode muß parameterlos sein
048             Class paras[] = methods[i].getParameterTypes();
049             if (paras.length > 0) {
050                 continue;
051             }
052             //Methode darf nicht static sein
053             int modifiers = methods[i].getModifiers();
054             if (Modifier.isStatic(modifiers)) {
055                 continue;
056             }
057             //Nun kann die Methode aufgerufen werden
058             ++cnt;
059             System.out.println("=====");
060             System.out.println("Aufgerufen wird: " + name);
061             System.out.println("--");
062             try {
063                 methods[i].invoke(tester, new Object[0]);
064             } catch (Exception e) {

```

```

065         System.err.println(e.toString());
066     }
067 }
068 if (cnt <= 0) {
069     System.out.println("Keine Testmethoden gefunden");
070 }
071 }
072
073 public static void main(String args[])
074 {
075     if (args.length <= 0) {
076         System.err.println("Aufruf: java Test <KlassenName>");
077         System.exit(1);
078     }
079     Object tester = createTestObject(args[0]);
080     runTests(tester);
081 }
082 }

```

Listing 31.3: Die Klasse *Test*

Das Hauptprogramm ruft zunächst die Methode `createTestObject` auf, um ein Objekt der Testklasse zu generieren. Falls als Argument also beispielsweise `Queue` übergeben wurde, wird ein Objekt des Typs `TestQueue` erzeugt. Ist es nicht vorhanden oder kann nicht instanziiert werden, liefert die Methode `null` als Rückgabewert.

Anschließend wird das Testobjekt an die Methode `runTests` übergeben. Diese besorgt sich das Klassenobjekt und ruft `getMethods` auf. Das zurückgegebene Array repräsentiert die Liste aller öffentlichen Methoden und wird elementweise durchlaufen. Zunächst wird überprüft, ob der Methodenname mit `test` anfängt. Ist das der Fall, wird geprüft, ob die Methode parameterlos ist und nicht das `static`-Attribut besitzt. Sind auch diese Bedingungen erfüllt, kann die Methode mit `invoke` aufgerufen werden. Als erstes Argument wird das Testobjekt übergeben. Als zweites folgt ein leeres Array des Typs `Object`, um anzuzeigen, daß keine Parameter zu übergeben sind.

Beachten Sie, daß am Anfang des Programms das Paket `java.lang.reflect` eingebunden wurde. Während die Klassen `Class` und `Object` aus historischen Gründen in `java.lang` liegen (und deshalb automatisch importiert werden), liegen sie für die übrigen Bestandteile des Reflection-APIs in `java.lang.reflect` und müssen deshalb explizit importiert werden.

Hinweis

Eine beispielhafte Implementierung der Klasse `TestQueue` könnte etwa so aussehen:

```

001 public class TestQueue
002 {
003     public TestQueue()
004     {
005         //Initialisierungen, z.B. Erzeugen eines zu
006         //testenden Queue-Objekts
007     }
008
009     public void test1()
010     {
011         //Erste Testmethode
012     }
013
014     public void test2()
015     {
016         //Zweite Testmethode
017     }
018
019     //...
020 }

```

Listing 31.4: Die Klasse *TestQueue*

Ein Aufruf von

```
java Test Queue
```

würde nun ein neues Objekt des Typs `TestQueue` instanzieren und nacheinander die Methoden `test1`, `test2` usw. aufrufen.

31.3.2 Parametrisierte Methoden

In diesem Abschnitt wollen wir uns den Aufruf *parametrisierter* Methoden ansehen, was nach den Ausführungen des vorigen Abschnitts nicht mehr schwierig ist. Als Beispiel soll ein Programm geschrieben werden, das die in Java nicht vorhandenen Funktionszeiger simuliert. Es soll eine Methode enthalten, die eine Wertetabelle für eine mathematische Funktion erzeugt, deren Name als String übergeben wurde. Nach den Überlegungen des vorigen Abschnitts können wir uns gleich das Listing ansehen:

[FloatTables.java](#)

```
001 /* FloatTables.java */
002
003 import java.lang.reflect.*;
004
005 public class FloatTables
006 {
007     public static double times2(double value)
008     {
009         return 2 * value;
010     }
011
012     public static double sqr(double value)
013     {
014         return value * value;
015     }
016
017     public static void printTable(String methname)
018     {
019         try {
020             System.out.println("Wertetabelle fuer " + methname);
021             int pos = methname.lastIndexOf('.');
022             Class clazz;
023             if (pos == -1) {
024                 clazz = FloatTables.class;
025             } else {
026                 clazz = Class.forName(methname.substring(0, pos));
027                 methname = methname.substring(pos + 1);
028             }
029             Class formparas[] = new Class[1];
030             formparas[0] = Double.TYPE;
031             Method meth = clazz.getMethod(methname, formparas);
032             if (!Modifier.isStatic(meth.getModifiers())) {
033                 throw new Exception(methname + " ist nicht static");
034             }
035             Object actargs[] = new Object[1];
036             for (double x = 0.0; x <= 5.0; x += 1) {
037                 actargs[0] = new Double(x);
038                 Double ret = (Double) meth.invoke(null, actargs);
039                 double result = ret.doubleValue();
040                 System.out.println("    " + x + " -> " + result);
041             }
042         } catch (Exception e) {
043             System.err.println(e.toString());
044         }
045     }
046
047     public static void main(String args[])
048     {
049         printTable("times2");
050         printTable("java.lang.Math.exp");
051         printTable("sqr");
052         printTable("java.lang.Math.sqrt");
053     }
054 }
```

Listing 31.5: Funktionszeiger mit Reflection nachbilden

Das Hauptprogramm ruft die Methode `printTable` viermal auf, um die Wertetabellen zu den statischen Funktionen `times2`, `java.lang.Math.exp`, `sqr` und `java.lang.Math.sqrt` zu erzeugen. Sie kann sowohl mit lokal definierten Methoden umgehen als auch

mit solchen, die in einer anderen Klasse liegen (in diesem Fall sogar aus einem anderen Paket).

In [Zeile 021](#) wird zunaechst der am weitesten rechts stehende Punkt im Methodennamen gesucht. Ist ein Punkt vorhanden, wird der String an dieser Stelle aufgeteilt. Der links davon stehende Teil wird als Klassenname angesehen, der rechts davon stehende als Methodename. Gibt es keinen Punkt, wird als Klassenname der Name der eigenen Klasse verwendet. Anschließend wird das zugehörige Klassenobjekt geladen. Ist der Klassenname zur Compilezeit bekannt, kann anstelle des Aufrufs von [forName](#) die abkürzende Schreibweise [.class](#) verwendet werden.

Anders als im vorigen Abschnitt generiert das Programm nun nicht eine Liste *aller* Methoden, sondern sucht mit [getMethod](#) ganz konkret nach einer bestimmten:

```
public Method getMethod(String name, Class[] parameterTypes)
```

Dazu müssen der Name der Methode und eine Beschreibung ihrer formalen Argumente an [getMethod](#) übergeben werden. Auch hier werden die Argumente durch ein Array mit korrespondierenden Klassenobjekten repräsentiert. Da die Methoden, die in diesem Fall aufgerufen werden sollen, nur ein einziges Argument vom Typ [double](#) haben sollen, hat unsere Parameterspezifikation [formParas](#) lediglich ein einziges Element [Double.TYPE](#). Wurde keine solche Methode gefunden, oder besitzt sie nicht das [static](#)-Attribut, löst [getMethods](#) eine Ausnahme des Typs [NoSuchMethodException](#) aus.

Würde die Methode anstelle des primitiven Typs [double](#) ein Argument des Referenztyps [Double](#) erwarten, hätten wir ein Klassenobjekt der Klasse [Double](#) übergeben müssen. Dafür gibt es verschiedene Möglichkeiten, beispielsweise die beiden folgenden:

```
formparas[0] = (new Double(0)).getClass();  
oder  
formparas[0] = Double.class;
```

Tip

Auch eine parametrisierte Methode kann mit [invoke](#) aufgerufen werden. Im Unterschied zur parameterlosen muß nun allerdings ein nicht-leeres [Object](#)-Array mit den aktuellen Argumenten übergeben werden. Hier zeigt sich ein vermeintliches Problem, denn in einem Array vom [Object\[\]](#) können keine primitiven Typen abgelegt werden. Um Methoden mit primitiven Parametern aufrufen zu können, werden diese einfach in die passende Wrapper-Klasse verpackt (siehe [Abschnitt 7.8.1](#)). Beim Aufruf von [invoke](#) werden sie dann automatisch »ausgepackt« und dem primitiven Argument zugewiesen.

Wir verpacken also den zu übergebenden Wert in ein [Double](#)-Objekt und stellen dieses in [Zeile 037](#) in das Array mit den aktuellen Argumenten. Beim Methodenaufruf in der nächsten Zeile wird es dann automatisch ausgepackt und steht innerhalb der Methode als [double](#)-Wert zur Verfügung. Das Programm ruft die Methode für jeden der Werte 0.0, 1.0, 2.0, 3.0, 4.0 und 5.0 auf und erzeugt so eine einfache Wertetabelle.

Anders als in [Listing 31.3](#) wird in diesem Beispiel als erstes Argument von [invoke](#) nicht das Objekt übergeben, an dem die Methode aufgerufen werden soll, sondern der Wert [null](#). Das liegt daran, daß wir eine *statische* Methode aufrufen, die keiner Objektinstanz, sondern dem Klassenobjekt zugeordnet ist. Bei nicht-statischen Methoden ist die Übergabe von [null](#) natürlich nicht erlaubt und würde zu einer [NullPointerException](#) führen.

Hinweis

Die Ausgabe des Programms ist:

```
Wertetabelle fuer times2  
0.0 -> 0.0  
1.0 -> 2.0  
2.0 -> 4.0  
3.0 -> 6.0  
4.0 -> 8.0  
5.0 -> 10.0  
Wertetabelle fuer java.lang.Math.exp  
0.0 -> 1.0  
1.0 -> 2.7182818284590455  
2.0 -> 7.38905609893065  
3.0 -> 20.085536923187668  
4.0 -> 54.598150033144236  
5.0 -> 148.4131591025766  
Wertetabelle fuer sqr  
0.0 -> 0.0  
1.0 -> 1.0  
2.0 -> 4.0  
3.0 -> 9.0  
4.0 -> 16.0
```

```

5.0 -> 25.0
Wertetabelle fuer java.lang.Math.sqrt
0.0 -> 0.0
1.0 -> 1.0
2.0 -> 1.4142135623730951
3.0 -> 1.7320508075688772
4.0 -> 2.0
5.0 -> 2.23606797749979

```

Funktionszeiger

Der in [Listing 31.5](#) vorgestellte Code ist eigentlich nicht zur Nachahmung empfohlen, sondern soll nur als Beispiel für den Aufruf parametrisierter Methoden mit Hilfe des Reflection-APIs dienen.

Funktionszeiger werden normalerweise mit Hilfe von Interfaces nachgebildet. Sie definieren eine einzelne Methode `x` des gewünschten Typs und werden dann von unterschiedlichen Klassen implementiert, die in `x` die gewünschte Berechnung ausführen. Anstelle des Methodennamens (oder -Zeigers, wie in C/C++) wird dann an `printTable` ein Objekt der entsprechenden Klasse übergeben. Der formale Parameter ist vom Typ des Interfaces und erlaubt es so, die Berechnungsmethode `x` aufzurufen.

Das folgende Listing zeigt ein zu [Listing 31.5](#) äquivalentes Programm, das diese Technik benutzt. Die Ausgabe des Programms ist praktisch mit der obigen identisch.

[Listing3106.java](#)

```

001 /* Listing3106.java */
002
003 public class Listing3106
004 {
005     public static void printTable(DoubleMethod meth)
006     {
007         System.out.println("Wertetabelle fuer " + meth.toString());
008         for (double x = 0.0; x <= 5.0; x += 1) {
009             System.out.println("  " + x + " -> " + meth.compute(x));
010         }
011     }
012
013     public static void main(String args[])
014     {
015         printTable(new Times2());
016         printTable(new MathExp());
017         printTable(new Sqr());
018         printTable(new MathSqrt());
019     }
020 }
021
022 interface DoubleMethod
023 {
024     public double compute(double value);
025 }
026
027 class MathExp
028 implements DoubleMethod
029 {
030     public double compute(double value)
031     {
032         return Math.exp(value);
033     }
034 }
035
036 class MathSqrt
037 implements DoubleMethod
038 {
039     public double compute(double value)
040     {
041         return Math.sqrt(value);
042     }
043 }
044

```

```

045 class Times2
046 implements DoubleMethod
047 {
048     public double compute(double value)
049     {
050         return 2 * value;
051     }
052 }
053
054 class Sqr
055 implements DoubleMethod
056 {
057     public double compute(double value)
058     {
059         return value * value;
060     }
061 }

```

Listing 31.6: Funktionszeiger mit Interfaces

31.3.3 Parametrisierte Konstruktoren

Die Methode [newInstance](#) der Klasse [Class](#) ruft immer den parameterlosen Konstruktor auf, um ein Objekt zu instanzieren. Mit Reflection ist es aber auch möglich, parametrisierte Konstruktoren zur dynamischen Instanzierung zu verwenden. Dazu besitzt [Class](#) zwei Methoden [getConstructors](#) und [getConstructor](#), die dazu verwendet werden, Konstruktorenobjekte zu beschaffen. Anders als [getMethods](#) und [getMethod](#) liefern sie allerdings kein Objekt des Typs [Method](#), sondern eines des Typs [Constructor](#) zurück.

Auch dieses besitzt die oben beschriebenen Methoden [getModifiers](#), [getName](#) und [getParameterTypes](#). Der Aufruf einer Methode erfolgt allerdings nicht mit [invoke](#), sondern mit [newInstance](#):

[java.lang.reflect.Constructor](#)

```
Object newInstance(Object[] initargs)
```

[newInstance](#) erwartet ebenfalls ein Array von Argumenten des Typs [Object](#). Diese werden gegebenenfalls in der zuvor beschriebenen Weise auf primitive Typen abgebildet und rufen schließlich den passenden Konstruktor auf. Als Rückgabewert von [newInstance](#) wird das neu instanzierte Objekt geliefert.

Das folgende Listing zeigt die Verwendung parametrisierter Konstruktoren mit dem Reflection-API:

[Listing3107.java](#)

```

001 /* Listing3107.java */
002
003 import java.lang.reflect.*;
004
005 public class Listing3107
006 {
007     public static void main(String args[])
008     {
009         Class clazz = TestConstructors.class;
010         //Formale Paramter definieren
011         Class formparas[] = new Class[2];
012         formparas[0] = String.class;
013         formparas[1] = String.class;
014         try {
015             Constructor cons = clazz.getConstructor(formparas);
016             //Aktuelle Argumente definieren
017             Object actargs[] = new Object[] {"eins", "zwei"};
018             Object obj = cons.newInstance(actargs);
019             ((TestConstructors)obj).print();
020         } catch (Exception e) {
021             System.err.println(e.toString());
022             System.exit(1);
023         }
024     }
025 }
026
027 class TestConstructors
028 {

```



```

029 private String arg1;
030 private String arg2;
031
032 public TestConstructors()
033 {
034     arg1 = "leer";
035     arg2 = "leer";
036 }
037
038 public TestConstructors(String arg1)
039 {
040     this();
041     this.arg1 = arg1;
042 }
043
044 public TestConstructors(String arg1, String arg2)
045 {
046     this();
047     this.arg1 = arg1;
048     this.arg2 = arg2;
049 }
050
051 public void print()
052 {
053     System.out.println("arg1 = " + arg1);
054     System.out.println("arg2 = " + arg2);
055 }
056 }

```

Listing 31.7: Parametrisierte Konstruktoren mit Reflection aufrufen

Das Programm erzeugt zunächst ein Klassenobjekt zu der Klasse `TestConstructors`. Anschließend wird ein Array mit zwei Klassenobjekten der Klasse `String` erzeugt und als Spezifikation der formalen Parameter an `getConstructor` übergeben. Das zurückgegebene `Constructor`-Objekt wird dann mit zwei aktuellen Argumenten "eins" und "zwei" vom Typ `String` ausgestattet, die an seine Methode `newInstance` übergeben werden. Diese instanziiert das Objekt, castet es auf die Klasse `TestConstructors` und ruft deren Methode `print` auf. Die Ausgabe des Programms ist:

```

arg1 = eins
arg2 = zwei

```

31.4 Zugriff auf Membervariablen

- 31.4 Zugriff auf Membervariablen

Nachdem wir uns in den vorangegangenen Abschnitten die beiden Reflection-Aspekte *Instanzierung* und *Methodenaufruf* angesehen haben, wollen wir uns nun mit dem Zugriff auf Membervariablen (also auf die »Felder« eines Objekts) mit Hilfe von Reflection beschäftigen. Prinzipiell gibt es keine großen Unterschiede gegenüber dem Zugriff auf eine Methode oder einen Konstruktor:

1. Zunächst wird ein Klassenobjekt für das zu bearbeitende Objekt beschafft.
2. Dann wird eine Methode aufgerufen, um ein Bearbeitungsobjekt für eine ganz bestimmte oder eine Liste aller Membervariablen zu beschaffen.
3. Das Bearbeitungsobjekt wird verwendet, um die Membervariable zu lesen, zu verändern oder andere Eigenschaften abzufragen.

Der erste Schritt erfolgt wie gewohnt durch Aufruf von `getClass` oder die `.class`-Notation. Die im zweiten Schritt benötigten Bearbeitungsobjekte sind vom Typ `Field`, sie können auf dem Klassenobjekt durch Aufruf einer der folgenden Methoden beschafft werden:

```

Field getField(String name)
Field[] getFields()
Field getDeclaredField(String name)
Field[] getDeclaredFields()

```

Mit `getField` wird die Membervariable mit dem angegebenen Namen beschafft, und `getFields` liefert ein Array mit `Field`-Objekten zu allen öffentlichen Membervariablen. Die beiden Methoden `getDeclaredField` und `getDeclaredFields` liefern darüber hinaus auch nicht-öffentliche Membervariablen.

Die Klasse `Field` besitzt Methoden zum Zugriff auf die Membervariable:

```

Class getType()
Object get(Object obj)
void set(Object obj, Object value)

```

Mit `getType` kann der Typ der Membervariable bestimmt werden. Das zurückgegebene Klassenobjekt beschreibt ihn in derselben Weise wie beispielsweise das Parameterobjekt von `getMethod`. Mit `get` kann auf den Wert zugegriffen werden, den die Membervariable in dem als Argument übergebenen Objekt hat. Handelt es sich um einen primitiven Typ, wird dieser automatisch in die passende Wrapper-Klasse verpackt und als Objekt zurückgegeben. Referenztypen werden unverändert zurückgegeben. Mit Hilfe der Methode `set` kann der Wert einer Membervariable verändert werden. Das erste Argument repräsentiert das zu verändernde Objekt und das zweite den neuen Wert der Membervariable. Soll ein primitiver Typ verändert werden, muß er vor der Übergabe in die passende Wrapper-Klasse verpackt werden.

Neben den generischen `get`- und `set`-Methoden gibt es diese auch in typisierter Form. So dient beispielsweise `getInt` dazu, ein `int`-Feld abzufragen, mit `getDouble` kann auf ein `double` zugegriffen werden usw. Das Gegenstück dazu sind die Methoden `setInt`, `setDouble` usw., mit denen die Membervariablen typisiert verändert werden können.

Tip

Wir wollen als Beispiel eine Klasse `PrintableObject` erstellen, die direkt aus `Object` abgeleitet ist und die Methode `toString` überlagert. Im Gegensatz zur Implementierung von `Object` soll unsere Variante von `toString` in der Lage sein, die Namen und Inhalte aller Membervariablen des zugehörigen Objekts auszugeben:

```

001 /* PrintableObject.java */
002
003 import java.lang.reflect.*;
004
005 public class PrintableObject
006 {
007     public String toString()
008     {
009         StringBuffer sb = new StringBuffer(200);
010         Class clazz = getClass();
011         while (clazz != null) {
012             Field fields[] = clazz.getDeclaredFields();
013             for (int i = 0; i < fields.length; ++i) {
014                 sb.append(fields[i].getName() + " = ");
015                 try {
016                     Object obj = fields[i].get(this);
017                     if (obj.getClass().isArray()) {
018                         Object ar[] = (Object[])obj;
019                         for (int j = 0; j < ar.length; ++j) {
020                             sb.append(ar[j].toString() + " ");
021                         }
022                         sb.append("\n");
023                     } else {
024                         sb.append(obj.toString() + "\n");
025                     }
026                 } catch (IllegalAccessException e) {
027                     sb.append(e.toString() + "\n");
028                 }
029             }
030             clazz = clazz.getSuperclass();
031         }
032         return sb.toString();
033     }
034
035     public static void main(String args[])
036     {
037         JavaProgrammer jim = new JavaProgrammer();
038         jim.name             = "Jim Miller";
039         jim.department       = "Operating Systems";
040         jim.age              = 32;
041         String langs[]      = {"C", "Pascal", "PERL", "Java"};
042         jim.languages        = langs;
043         jim.linesofcode     = 55000;
044         jim.jdk12           = true;
045         jim.swing           = false;
046         System.out.println(jim);
047     }
048 }
049
050 class Employee
051 extends PrintableObject
052 {
053     public String name;
054     public String department;
055     public int    age;
056 }
057
058 class Programmer
059 extends Employee
060 {
061     public String[] languages;
062     public int    linesofcode;
063 }
064

```

```
065 class JavaProgrammer
066 extends Programmer
067 {
068     public boolean jdk12;
069     public boolean swing;
070 }
```

Listing 31.8: Die Klasse PrintableObject

[toString](#) besorgt zunächst ein Klassenobjekt zum aktuellen Objekt. Mit [getDeclaredFields](#) wird dann eine Liste *aller* Felder (nicht nur der öffentlichen) dieser Klasse besorgt und jeweils mit [getName](#) sein Name und mit [get](#) sein Wert ausgegeben. Wir machen uns dabei die Fähigkeit zunutze, daß alle Objekte (auch die Wrapper-Klassen der primitiven Typen) eine Methode [toString](#) haben, die ihren Wert als String liefert. Ist die Membervariable ein Array, durchläuft das Programm zusätzlich seine Elemente und gibt sie einzeln aus.

Da eine Klasse Membervariablen aus ihren Vaterklassen erbt, ist es notwendig, die Vererbungshierarchie von unten nach oben zu durchlaufen, denn [getDeclaredFields](#) liefert nur die Membervariablen der aktuellen Klasse. Der Aufruf von [getSuperClass](#) am Ende der Schleife liefert zur aktuellen Klasse die Vaterklasse. Ist der Rückgabewert [null](#), so ist das Ende der Vererbungshierarchie erreicht ([clazz](#) repräsentiert dann die Klasse [Object](#)) und die Schleife wird beendet.

Die Klassen [Employee](#), [Programmer](#) und [JavaProgrammer](#) zeigen beispielhaft die Anwendung von [PrintableObject](#). [Employee](#) ist aus [PrintableObject](#) abgeleitet und erbt die modifizierte Methode [toString](#). [Programmer](#) ist aus [Employee](#) abgeleitet und [JavaProgrammer](#) aus [Programmer](#). Das Hauptprogramm erzeugt ein Objekt des Typs [JavaProgrammer](#) und weist seinen eigenen und den geerbten Membervariablen Werte zu, die durch den anschließenden Aufruf von [toString](#) auf dem Bildschirm ausgegeben werden. Die Ausgabe des Programms ist:

```
jdk12 = true
swing = false
languages = C Pascal PERL Java
linesofcode = 55000
name = Jim Miller
department = Operating Systems
age = 32
```

31.5 Zusammenfassung

- [31.5 Zusammenfassung](#)

Das Reflection-API ist ohne Zweifel ein wichtiger Bestandteil von Java. Es ist nicht nur Grundlage der Beans- und Serialisierungs-APIs, sondern eröffnet dem Tool-Entwickler auch eine Fülle von Möglichkeiten und Freiheiten bei der Entwicklung allgemein verwendbarer Werkzeuge. Zu beachten ist allerdings, daß Zugriffe auf Methoden und Membervariablen deutlich langsamer als bei direktem Zugriff ausgeführt werden. Im experimentellen Vergleich ergaben sich typischerweise Unterschiede um den Faktor 10 bis 100. Während auf der verwendeten Testmaschine der Aufruf einer Methode mit einem `double`-Parameter beispielsweise 120 ns. dauerte, wurden für denselben Aufruf per Reflection etwa 7.0 µs. benötigt. Bei deaktiviertem Just-In-Time-Compiler erhöhte sich die Zeit für den direkten Aufruf auf 620 ns., während der Reflection-Wert mit etwa 7.2 µs. nahezu unverändert blieb. Beim Zugriff auf Membervariablen wurden ähnliche Unterschiede gemessen. Bemerkenswert ist, daß der JIT bei Verwendung des Reflection-APIs praktisch keine Geschwindigkeitsvorteile bringt, denn alle nennenswerten Methoden sind native und können daher kaum optimiert werden.

In diesem Kapitel wurden folgende Themen behandelt:

- Die Klassen `Object` und `Class`.
- Dynamisches Laden und Instanzieren von Klassen.
- Aufruf parameterloser Methoden.
- Ein Hilfsprogramm zum Testen von Java-Klassen.
- Aufruf parametrisierter Methoden und das automatische Aus- und Einpacken von primitiven Typen in Wrapperklassen.
- Nachbilden von Funktionszeigern mit Reflection.
- Verwenden parametrisierter Konstruktoren.
- Zugriff auf Membervariablen.

Kapitel 32

Netzwerkprogrammierung

- [32 Netzwerkprogrammierung](#)
 - [32.1 Grundlagen der Netzwerkprogrammierung](#)
 - [32.1.1 Was ist ein Netzwerk?](#)
 - [32.1.2 Protokolle](#)
 - [32.1.3 Adressierung von Daten](#)
 - [IP-Adressen](#)
 - [Domain-Namen](#)
 - [32.1.4 Ports und Applikationen](#)
 - [32.1.5 Request for Comments](#)
 - [32.1.6 Firewalls und Proxys](#)
 - [32.2 Client-Sockets](#)
 - [32.2.1 Adressierung](#)
 - [32.2.2 Aufbau einer einfachen Socket-Verbindung](#)
 - [32.2.3 Lesen und Schreiben von Daten](#)
 - [32.2.4 Zugriff auf einen Web-Server](#)
 - [32.3 Server-Sockets](#)
 - [32.3.1 Die Klasse ServerSocket](#)
 - [32.3.2 Verbindungen zu mehreren Clients](#)
 - [32.3.3 Entwicklung eines einfachen Web-Servers](#)
 - [32.4 Daten mit Hilfe der Klasse URL lesen](#)
 - [32.5 Zusammenfassung](#)

32.1 Grundlagen der Netzwerkprogrammierung

- [32.1 Grundlagen der Netzwerkprogrammierung](#)
 - [32.1.1 Was ist ein Netzwerk?](#)
 - [32.1.2 Protokolle](#)
 - [32.1.3 Adressierung von Daten](#)
 - [IP-Adressen](#)
 - [Domain-Namen](#)
 - [32.1.4 Ports und Applikationen](#)
 - [32.1.5 Request for Comments](#)
 - [32.1.6 Firewalls und Proxys](#)

32.1.1 Was ist ein Netzwerk?

Man könnte ganz plakativ sagen, daß ein Netzwerk die Verbindung zwischen zwei oder mehr Computern ist, um Daten miteinander auszutauschen. Das ist natürlich eine sehr vereinfachte Darstellung, die am besten in die Anfangszeit der Entstehung von Computernetzen paßt. Heutzutage gibt es eine Vielzahl von Anwendungen, die auf dem Austausch von Daten über ein Netzwerk basieren. Zu ihnen zählen beispielsweise:

- Ein Anwender möchte auf eine Datei zugreifen, die ein anderer Anwender erstellt hat.
- Mehrere Arbeitsplätze sollen auf einen gemeinsamen Drucker oder ein zentrales Faxgerät zugreifen.
- Beim Booten sollen sich PC-Arbeitsplätze die aktuelle Uhrzeit von einem Server im Netz holen.
- Das Intranet eines Unternehmens gibt den Angestellten Zugriff auf oft benötigte Informationen.
- Über einen Internet-Zugang soll eine elektronische Mail an einen Bekannten oder Geschäftspartner in einem anderen Teil der Welt verschickt werden.
- Ein Wissenschaftler möchte ein Experiment auf einem weit entfernten Hochleistungsrechner laufen lassen.
- Ein Dutzend Rechner sollen zur Leistungssteigerung in einem Cluster verbunden werden.
- Das lokale Netz eines Unternehmens wird zur Übertragung von Sprach- und Bilddaten verwendet.
- In einem Chatroom treffen sich Interessierte aus der ganzen Welt, um zeitgleich über ein gemeinsames Thema zu diskutieren.

So vielfältig wie diese Anwendungen ist auch die dafür erforderliche Hard- und Softwaretechnik. Das Thema »Computernetze« hat in den letzten Jahren stark an Dynamik und Umfang zugenommen. Es gibt heute eigene Studiengänge, deren Schwerpunkt auf der Vernetzung von Computersystemen liegt. Fast jedes größere Unternehmen beschäftigt Mitarbeiter, die sich ausschließlich um den Betrieb und die Erweiterung der unternehmenseigenen Netze und ihrer Anbindung an öffentliche Netze kümmern.

32.1.2 Protokolle

Um überhaupt Daten zwischen zwei oder mehr Partnern austauschen zu können, müssen sich die Teilnehmer auf ein gemeinsames *Protokoll* geeinigt haben. Als *Protokoll* bezeichnet man die Menge aller Regeln, die notwendig sind, um einen kontrollierten und eindeutigen Verbindungsaufbau, Datenaustausch und Verbindungsabbau gewährleisten zu können. Es gibt sehr viele Protokolle mit sehr unterschiedlichen Ansprüchen. Ein weitverbreitetes Architekturmodell, das *ISO/OSI-7-Schichten-Modell*, unterteilt sie in Abhängigkeit von ihrem Abstraktionsgrad in sieben Schichten (siehe [Abbildung 32.1](#)).

Schicht 7: Anwendungsschicht
Schicht 6: Darstellungsschicht
Schicht 5: Sitzungsschicht
Schicht 4: Transportschicht
Schicht 3: Netzwerkschicht
Schicht 2: Leitungsschicht
Schicht 1: Physikalische Schicht

Abbildung 32.1: Das ISO/OSI-7-Schichten-Modell

Die derzeit in Java verfügbaren Netzwerkfähigkeiten basieren alle auf den Internet-Protokollen *TCP/IP* (bzw. *TCP/UDP*). Wir wollen uns in diesem Kapitel ausschließlich mit der TCP/IP-Familie von Protokollen beschäftigen. Hierfür wird häufig eine etwas vereinfachte Unterteilung in vier Ebenen vorgenommen (siehe [Abbildung 32.2](#)):

- Die unterste Ebene repräsentiert die physikalischen Geräte. Sie wird durch die Netzwerkhardware und -leitungen und die unmittelbar darauf laufenden Protokolle wie beispielsweise *Ethernet*, *FDDI* oder *ATM* repräsentiert.
- Die zweite Ebene repräsentiert die *Netzwerkschicht*. Sie wird in TCP/IP-Netzen durch das *IP-Protokoll* und dessen Kontrollprotolle (z.B. *ICMP*) implementiert.
- Die dritte Ebene stellt die *Transportschicht* dar. Sie wird durch die Protokolle TCP bzw. UDP repräsentiert.
- Die oberste Ebene steht für die große Klasse der *Anwendungsprotokolle*. Hierzu zählen beispielsweise FTP zum Filetransfer, SMTP zum Mail-Versand und HTTP zur Übertragung von Web-Seiten.

Schicht 4: Anwendungsschicht
Schicht 3: Transportschicht
Schicht 2: Netzwerkschicht
Schicht 1: Verbindungsschicht

Abbildung 32.2: Das vereinfachte 4-Ebenen-Modell

Die TCP/IP-Protokollfamilie wird sowohl in lokalen Netzen als auch im Internet verwendet. Alle Eigenarten des Transportweges werden auf der zweiten bzw. dritten Ebene ausgeglichen, und die Anwendungsprotokolle merken *prinzipiell* keinen Unterschied zwischen lokalen und globalen Verbindungen. Wurde beispielsweise ein SMTP-Mailer für den Versand von elektronischer Post in einem auf TCP/IP basierenden Unternehmensnetz entwickelt, so kann dieser im Prinzip auch dazu verwendet werden, eine Mail ins Internet zu versenden.

TCP ist ein *verbindungsorientiertes* Protokoll, das auf der Basis von IP eine sichere und fehlerfreie Punkt-zu-Punkt-Verbindung realisiert. Daneben gibt es ein weiteres Protokoll oberhalb von IP, das als *UDP (User Datagram Protocol)* bezeichnet wird. Im Gegensatz zu TCP ist UDP ein *verbindungsloses* Protokoll, bei dem die Anwendung selbst dafür sorgen muß, daß Pakete fehlerfrei und in der richtigen Reihenfolge beim Empfänger ankommen. Sein Vorteil ist die größere Geschwindigkeit gegenüber TCP. In Java wird UDP durch die Klassen [DatagramPacket](#) und [DatagramSocket](#) abgebildet. Wir wollen uns in diesem Abschnitt allerdings ausschließlich mit den populäreren Protokollen auf der Basis von TCP/IP beschäftigen, TCP/UDP wird im folgenden nicht weiter behandelt.

Hinweis

32.1.3 Adressierung von Daten

IP-Adressen

Um Daten über ein Netzwerk zu transportieren, ist eine *Adressierung* dieser Daten notwendig. Der Absender muß angeben können, an wen die Daten zu senden sind, und der Empfänger muß erkennen können, von wem sie stammen. Die Adressierung in TCP/IP-Netzen erfolgt auf der IP-Ebene mit Hilfe einer 32-Bit langen *IP-Adresse*.

Die IP-Adresse besteht aus einer *Netzwerk-ID* und einer *Host-ID*. Die Host-ID gibt die Bezeichnung des Rechners innerhalb seines eigenen Netzwerks an, und die Netzwerk-ID liefert die Bezeichnung des Netzwerks. Alle innerhalb eines Verbundes von Netzwerken sichtbaren Adressen müssen eindeutig sein; es darf also nicht zweimal dieselbe Host-ID innerhalb eines Netzwerks geben. Sind mehrere Netzwerke miteinander verbunden (wie es beispielsweise im Internet der Fall ist), müssen auch die Netzwerk-IDs innerhalb des Verbundes eindeutig sein.

Um diese Eindeutigkeit sicherzustellen, gibt es eine zentrale Institution zur Vergabe von Internet-Adressen und -namen, das *Network Information Center* (kurz *NIC*). Die zu vergebenden Adressen werden in drei Klassen A bis C eingeteilt (die zusätzlichen existierenden Klasse-D- und Klasse-E-Adressen spielen hier keine Rolle):

Klasse	Netzwerk-ID	Host-ID	Beschreibung

A	7 Bit	24 Bit	Ein Klasse-A-Netz ist für sehr große Netzbetreiber vorgesehen. Das erste Byte der Adresse liegt im Bereich von 0 bis 127, sein höchstwertiges Bit ist also immer 0. Ein Klasse-A-Netz bietet 2^{24} verschiedene Host-IDs innerhalb des Netzes. Insgesamt gibt es aber nur 128 verschiedene Klasse-A-Netze weltweit (tatsächlich werden seit einigen Jahren keine Klasse-A-Adressen mehr vom NIC vergeben).
B	14 Bit	16 Bit	Ein Klasse-B-Netz erlaubt immerhin noch die eindeutige Adressierung von 2^{16} unterschiedlichen Rechnern innerhalb des Netzwerks. Insgesamt gibt es maximal 16384 verschiedene Klasse-B-Netze weltweit. Das erste Byte der Adresse liegt im Bereich von 128 bis 191, seine höchstwertigen Bits haben also immer den Binärwert 10.
C	21 Bit	8 Bit	Klasse-C-Netze sind für kleinere Unternehmen vorgesehen, die nicht mehr als 256 unterschiedliche Rechner adressieren müssen. Insgesamt gibt es maximal 2097152 verschiedene Klasse-C-Netze weltweit. Das erste Byte der Adresse liegt im Bereich von 192 bis 223, seine höchstwertigen Bits haben also immer den Binärwert 110. Die meisten an das Internet angebundenen kleineren Unternehmen betreiben heute ein Klasse-C-Netz.

Tabelle 32.1: Klassen von IP-Adressen

Obwohl damit theoretisch etwa 4 Milliarden unterschiedliche Rechner angesprochen werden können, reichen die Adressen in der Praxis nicht aus. Durch die starren Grenzen sind bereits viele mittlere Unternehmen gezwungen, ein Klasse-B-Netz zu betreiben, weil sie die 256-Rechner-Grenze knapp überschreiten, oder damit rechnen, sie demnächst zu überschreiten. Dadurch wird ein fester Block von 65536 Adressen vergeben. Um diese Probleme in Zukunft zu umgehen, werden Internet-Adressen demnächst nach dem Standard *IPv6* definiert werden, der eine Adresslänge von 128 Bit vorsieht.

Hinweis

Domain-Namen

Während IP-Adressen für Computer sehr leicht zu verarbeiten sind, gilt das nicht unbedingt für die Menschen, die damit arbeiten müssen. Wer kennt schon die Telefonnummern und Ortsnetzkennzahlen von allen Leuten, mit denen er Telefonate zu führen pflegt? Um die Handhabung der IP-Adressen zu vereinfachen, wurde daher das *Domain Name System* eingeführt (kurz *DNS*), das numerischen IP-Adressen sprechende Namen wie *www.gkrueger.com* oder *java.sun.com* zuordnet.

Anstelle der IP-Adresse können bei den Anwendungsprotokollen nun wahlweise die symbolischen Namen verwendet werden. Sie werden mit Hilfe von *Name-Servern* in die zugehörige IP-Adresse übersetzt, bevor die Verbindung aufgebaut wird. Zwar kann sich hinter ein und denselben

IP-Adresse mehr als ein Name-Server-Eintrag befinden. In umgekehrter Richtung ist die Zuordnung aber eindeutig, d.h. zu jedem symbolischen Namen kann eindeutig die zugehörige IP-Adresse (und damit das Netz und der Host) bestimmt werden, zu der der Name gehört.

32.1.4 Ports und Applikationen

Die Kommunikation zwischen zwei Rechnern läuft oft auf der Basis einer *Client-Server-Beziehung* ab. Dabei kommen den beteiligten Rechnern unterschiedliche Rollen zu:

- Der Server stellt einen Dienst zur Verfügung, der von anderen Rechnern genutzt werden kann. Er läuft im Hintergrund und wartet darauf, daß ein anderer Rechner eine Verbindung zu ihm aufbaut. Der Server definiert das Protokoll, mit dessen Hilfe der Datenaustausch erfolgt.
- Ein Client ist der Nutzer von Diensten eines oder mehrerer Server. Er kennt die verfügbaren Server und ihre Adressen und baut bei Bedarf die Verbindung zu ihnen auf. Der Client hält sich an das vom Server vorgegebene Protokoll, um die Daten auszutauschen.

Ein typisches Beispiel für eine Client-Server-Verbindung ist der Seitenabruf im World Wide Web. Der Browser fungiert als Client, der nach Aufforderung durch den Anwender eine Verbindung zum Web-Server aufbaut und eine Seite anfordert. Diese wird vom Server von seiner Festplatte geladen oder dynamisch generiert und an den Browser übertragen. Dieser analysiert die Seite und stellt sie auf dem Bildschirm dar. Enthält die Seite Image-, Applet- oder Frame-Dateien, werden sie in gleicher Weise beim Server abgerufen und in die Seite integriert.

In der Praxis ist die Kommunikationsbeziehung nicht immer so einfach wie hier dargestellt. Es Hinweis gibt insbesondere auch symmetrische Kommunikationsbeziehungen, die nicht dem Client-Server-Modell entsprechen. Zudem kann ein Server auch Client eines anderen Servers sein. Auf diese Weise können mehrstufige Client-Server-Beziehungen entstehen.

Auf einem Host laufen meist unterschiedliche Serveranwendungen, die noch dazu von mehreren Clients gleichzeitig benutzt werden können. Um die Server voneinander unterscheiden zu können, gibt es ein weiteres Adressmerkmal, die *Port-Nummer*. Sie wird oberhalb von IP auf der Ebene des Transportprotokolls definiert (also in TCP bzw. UDP) und gibt die Server-Anwendung an, mit der ein Client kommunizieren will. Port-Nummern sind positive Ganzzahlen im Bereich von 0 bis 65535. Port-Nummern im Bereich von 0 bis 1023 sind für Anwendungen mit Superuser-Rechten reserviert. Jeder Servertyp hat seine eigene Port-Nummer, viele davon sind zu Quasi-Standards geworden. So läuft beispielsweise ein SMTP-Server meist auf Port 25, ein FTP-Server auf Port 21 und ein HTTP-Server auf Port 80. [Tabelle 32.2](#) gibt eine Übersicht der auf den meisten UNIX-Systemen verfügbaren Server und ihrer Port-Nummern.

Name	Port	Transport	Beschreibung
echo	7	tcp/udp	Gibt jede Zeile zurück, die der Client sendet
discard	9	tcp/udp	Ignoriert jede Zeile, die der Client sendet
daytime	13	tcp/udp	Liefert ASCII-String mit Datum und Uhrzeit
chargen	19	tcp/udp	Generiert ununterbrochen Zeichen
ftp	21	tcp	Versenden und Empfangen von Dateien
telnet	23	tcp	Interaktive Session mit entferntem Host
smtp	25	tcp	Versenden von E-Mails
time	37	tcp/udp	Liefert die aktuelle Uhrzeit als Anzahl der Sekunden seit 1.1.1900
whois	43	tcp	Einfacher Namensservice
tftp	69	udp	Vereinfachte Variante von FTP auf UDP-Basis
gopher	70	tcp/udp	Quasi-Vorgänger von WWW
finger	79	tcp	Liefert Benutzerinformationen
www	80	tcp/udp	Der Web-Server
pop3	110	tcp/udp	Übertragen von Mails
nntp	119	tcp	Übertragen von Usenet-News
snmp	161	udp	Netzwerkmanagement

Tabelle 32.2: Standard-Port-Nummern

32.1.5 Request for Comments

Die meisten der allgemein zugänglichen Protokolle sind in sogenannten *Request For Comments* (kurz *RFCs*) beschrieben. RFCs sind Dokumente des *Internet Activity Board (IAB)*, in denen Entwürfe, Empfehlungen und Standards zum Internet beschrieben sind. Auch Anmerkungen, Kommentare oder andere informelle Ergänzungen sind darin zu finden.

Insgesamt gibt es derzeit etwa 2500 RFCs, einige von ihnen wurden zu Internet-Standards erhoben. Alle bekannten Protokolle, wie beispielsweise FTP, SMTP, NNTP, MIME, DNS, HTML oder HTTP, sind in einschlägigen RFCs beschrieben. Sie sind nicht immer einfach zu lesen, aber oftmals die einzige verlässliche Quelle für die Implementierung eines bestimmten Protokolls. Es gibt viele Server im Internet, die RFCs zur Verfügung stellen. Bekannte Beispiele sind <http://rfc.fh-koeln.de/>, <http://www.cis.ohio-state.edu/hypertext/information/rfc.html>. Um beispielsweise an der FH Köln nicht immer den kompletten Index laden zu müssen, kann man über die Adresse <http://rfc.fh-koeln.de/rfc/html/rfcXXXX.html>

auch direkt zur gewünschten Seite springen. [XXXX](#) steht dabei für die gesuchte RFC-Nummer (also beispielsweise 0868 oder 1436).

Protokoll	Zuständige RFCs
IP	RFC791, RFC1060
ICMP	RFC792
TCP	RFC793
UDP	RFC768
DNS	RFC1034, RFC1035, RFC2136, RFC974, RFC1101, RFC1812
ARP / RARP	RFC826, RFC903
SMTP	RFC821, RFC822
MIME	RFC2045 - RFC2049
Content Types	RFC1049
POP3	RFC1939
NNTP	RFC977
HTML 3.2	Internal Draft
HTML 2.0	RFC1866
HTTP 1.0 / 1.1	RFC1945, RFC2068
FTP	RFC959, RFC765
TFTP	RFC1782, RFC1783, RFC1350
TELNET	RFC854
SNMP	RFC1157
X11	RFC1013
NTP	RFC1305
FINGER	RFC1288
WHOIS	RFC954
GOPHER	RFC1436
ECHO	RFC862
DISCARD	RFC863
CHARGEN	RFC864
DAYTIME	RFC867
TIME	RFC868
Assigned Numbers	RFC1700
Internet Protocol Standards	RFC2400
Hitchhikers guide to the Internet	RFC1118

Tabelle 32.3: Liste wichtiger RFCs

Die hier aufgelisteten RFCs finden sich auch auf der CD-ROM zum Buch. Sie liegen als Textdateien im Verzeichnis `\rfc`. Zusätzlich findet sich dort eine Datei `INDEX_rfc.html` mit einer Übersicht über alle RFCs (Stand: November 1998).

Hinweis

32.1.6 Firewalls und Proxys

Nicht alle Server in einen Netzwerk sind für alle Clients sichtbar. Aufgrund von Sicherheitserwägungen wird insbesondere die Verbindung zwischen einem lokalen Unternehmensnetz und der Außenwelt (z.B. dem Internet) besonders geschützt. Dazu wird meist eine *Firewall* verwendet, also ein spezielles Gateway mit Filterfunktion, das Netzwerkverkehr nur in bestimmten Richtungen und in Abhängigkeit von Port-Nummern, IP-Adressen und anderen Informationen zuläßt. Mit einer Firewall kann beispielsweise dafür gesorgt werden, daß nicht *von außen* auf den Mail-Server (Port 25) des Unternehmens zugegriffen werden kann. Oder es kann verhindert werden, daß die firmeninternen Anwender bestimmte Web-Server im Internet besuchen usw.

Normalerweise ist es insbesondere nicht erlaubt, IP-Daten zwischen einem beliebigen Arbeitsplatzrechner und einem außerhalb des Unternehmens liegenden Server hin- und herzusenden. Um dennoch beispielsweise das Anfordern von Web-Seiten von beliebigen Servern zu ermöglichen, kommuniziert der Web-Browser auf dem Arbeitsplatz mit einem *Proxy-Server* (kurz *Proxy*), der *innerhalb* der Firewall liegt. Anstatt die Seitenanfrage direkt an den Server zu schicken, wird sie an den Proxy übergeben, der sie an den Server weiterleitet. Die Antwort des Servers wird nach Erhalt vom Proxy an den anfordernden Arbeitsplatz gesendet. Die Firewall muß also lediglich dem Proxy eine HTTP-Verbindung ins Internet

gestatten, nicht allen Arbeitsplätzen.

Ein Proxy ist also eine Art »Handlungsbevollmächtigter« (so lautet die wörtliche Übersetzung), der Aufgaben erledigt, die dem einzelnen Arbeitsplatz nicht erlaubt sind. Proxys gibt es auch für andere Zwecke, etwa zum Zugriff auf Datenbanken. Da beispielsweise ein Applet aus Sicherheitsgründen nur zu dem Server eine TCP/IP-Verbindung aufbauen darf, von dem es geladen wurde, kann es auf Daten aus einer Datenbank nur zugreifen, wenn die Datenbank auf demselben Host liegt wie der Web-Server. Ist dies nicht der Fall, kann man sich mit einem Datenbankproxy auf dem Web-Host behelfen, der alle entsprechenden Anfragen an die Datenbank weiterleitet.

Tit	Inh	Ind	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	<<	≤	≥	>>
---------------------	---------------------	---------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------------	-------------------	-------------------	--------------------------

32.2 Client-Sockets

- [32.2 Client-Sockets](#)
 - [32.2.1 Adressierung](#)
 - [32.2.2 Aufbau einer einfachen Socket-Verbindung](#)
 - [32.2.3 Lesen und Schreiben von Daten](#)
 - [32.2.4 Zugriff auf einen Web-Server](#)

32.2.1 Adressierung

Zur Adressierung von Rechnern im Netz wird die Klasse [InetAddress](#) des Pakets [java.net](#) verwendet. Ein [InetAddress](#)-Objekt enthält sowohl eine IP-Adresse als auch den symbolischen Namen des jeweiligen Rechners. Die beiden Bestandteile können mit den Methoden [getHostName](#) und [getHostAddress](#) abgefragt werden. Mit Hilfe von [getAddress](#) kann die IP-Adresse auch direkt als [byte](#)-Array mit vier Elementen beschafft werden:

```

String getHostName()
String getHostAddress()
byte[] getAddress()

```

Um ein [InetAddress](#)-Objekt zu generieren, stehen die beiden statischen Methoden [getByName](#) und [getLocalHost](#) zur Verfügung:

```

public static InetAddress getByName(String host)
    throws UnknownHostException

public static InetAddress getLocalHost()
    throws UnknownHostException

```

[getByName](#) erwartet einen String mit der IP-Adresse oder dem Namen des Hosts als Argument, [getLocalHost](#) liefert ein [InetAddress](#)-Objekt für den eigenen Rechner. Beide Methoden lösen eine Ausnahme des Typs [UnknownHostException](#) aus, wenn die Adresse nicht ermittelt werden kann. Das ist insbesondere dann der Fall, wenn kein DNS-Server zur Verfügung steht, der die gewünschte Namensauflösung erledigen könnte (beispielsweise weil die Dial-In-Verbindung zum Provider gerade nicht besteht).

Das folgende Listing zeigt ein einfaches Programm, das zu einer IP-Adresse den symbolischen Namen des zugehörigen Rechners ermittelt und umgekehrt:

Beispiel

[Listing3201.java](#)

```

001 /* Listing3201.java */
002
003 import java.net.*;
004
005 public class Listing3201
006 {
007     public static void main(String args[])
008     {
009         if (args.length != 1) {
010             System.err.println("Usage: java Listing3201 <host>");
011             System.exit(1);
012         }
013         try {
014             //Get requested address
015             InetAddress addr = InetAddress.getByName(args[0]);
016             System.out.println(addr.getHostName());
017             System.out.println(addr.getHostAddress());
018         } catch (UnknownHostException e) {
019             System.err.println(e.toString());
020             System.exit(1);
021         }
022     }
023 }

```

```
022 }  
023 }
```

Listing 32.1: IP-Adressenauflösung

Wird das Programm mit `localhost` als Argument aufgerufen, ist seine Ausgabe:

```
localhost  
127.0.0.1
```

`localhost` ist eine Pseudo-Adresse für den eigenen Host. Sie ermöglicht das Testen von Netzwerkanwendungen, auch wenn keine wirkliche Netzwerkverbindung besteht (TCP/IP muß allerdings korrekt installiert sein). Sollen wirkliche Adressen verarbeitet werden, muß natürlich eine Verbindung zum Netz (insbesondere zum DNS-Server) aufgebaut werden können.

Hinweis

Die nachfolgende Ausgabe zeigt die Ausgabe des Beispielprogramms, wenn es nacheinander mit den Argumenten `java.sun.com`, `www.gkrueger.com` und `www.addison-wesley.de` aufgerufen wird:

```
java.sun.com  
204.160.241.19
```

```
www.gkrueger.com  
194.88.189.5
```

```
www.addison-wesley.de  
194.77.140.21
```

32.2.2 Aufbau einer einfachen Socket-Verbindung

Als *Socket* bezeichnet man eine streambasierte Programmierschnittstelle zur Kommunikation zweier Rechner in einem TCP/IP-Netz. Sockets wurden Anfang der achtziger Jahre für die Programmiersprache C entwickelt und mit Berkeley UNIX 4.1/4.2 allgemein eingeführt. Das Übertragen von Daten über eine Socket-Verbindung ähnelt dem Zugriff auf eine Datei:

- Zunächst wird eine Verbindung aufgebaut.
- Dann werden Daten gelesen und/oder geschrieben.
- Schließlich wird die Verbindung wieder abgebaut.

Während die Socket-Programmierung in C eine etwas mühsame Angelegenheit war, ist es in Java recht einfach geworden. Im wesentlichen sind dazu die beiden Klassen [Socket](#) und [ServerSocket](#) erforderlich. Sie repräsentieren Sockets aus der Sicht einer Client- bzw. Server-Anwendung. Nachfolgend wollen wir uns mit den Client-Sockets beschäftigen, die Klasse [ServerSocket](#) wird im nächsten Abschnitt behandelt.

Die Klasse [Socket](#) besitzt verschiedene Konstruktoren, mit denen ein neuer Socket erzeugt werden kann. Die wichtigsten von ihnen sind:

[java.net.Socket](#)

```
public Socket(String host, int port)  
    throws UnknownHostException, IOException
```

```
public Socket(InetAddress address, int port)  
    throws IOException
```

Beide Konstruktoren erwarten als erstes Argument die Übergabe des Hostnamens, zu dem eine Verbindung aufgebaut werden soll. Dieser kann entweder als Domainname in Form eines Strings oder als Objekt des Typs [InetAddress](#) übergeben werden. Soll eine Adresse mehrfach verwendet werden, ist es besser, die zweite Variante zu verwenden. In diesem Fall kann das übergebene [InetAddress](#)-Objekt wiederverwendet werden, und die Adressauflösung muß nur einmal erfolgen. Wenn der Socket nicht geöffnet werden konnte, gibt es eine Ausnahme des Typs [IOException](#) bzw. [UnknownHostException](#) (wenn das angegebene Zielsystem nicht angesprochen werden konnte).

Der zweite Parameter des Konstruktors ist die Portnummer. Wie in [Abschnitt 32.1.4](#) erwähnt, dient sie dazu, den Typ des Servers zu bestimmen, mit dem eine Verbindung aufgebaut werden soll. Die wichtigsten Standard-Portnummern sind in [Tabelle 32.2](#) aufgelistet.

Nachdem die Socket-Verbindung erfolgreich aufgebaut wurde, kann mit den beiden Methoden [getInputStream](#) und [getOutputStream](#) je ein Stream zum Empfangen und Versenden von Daten beschafft werden:

[java.net.Socket](#)

```
public InputStream getInputStream()  
    throws IOException
```

```
public OutputStream getOutputStream()  
    throws IOException
```

Diese Streams können entweder direkt verwendet oder mit Hilfe der Filterstreams in einen bequemer zu verwendenden Streamtyp geschachtelt werden. Nach Ende der Kommunikation sollten sowohl die Eingabe- und Ausgabestreams als auch der Socket selbst mit [close](#) geschlossen werden.

Als erstes Beispiel wollen wir uns ein Programm ansehen, das eine Verbindung zum *DayTime*-Service auf Port 13 herstellt. Dieser Service läuft auf fast allen UNIX-Maschinen und kann gut zu Testzwecken verwendet werden. Nachdem der Client die Verbindung aufgebaut hat, sendet der DayTime-Server einen String mit dem aktuellen Datum und der aktuellen Uhrzeit und beendet dann die Verbindung.

Beispiel

[Listing3202.java](#)

```
001 /* Listing3202.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class Listing3202
007 {
008     public static void main(String args[])
009     {
010         if (args.length != 1) {
011             System.err.println("Usage: java Listing3202 <host>");
012             System.exit(1);
013         }
014         try {
015             Socket sock = new Socket(args[0], 13);
016             InputStream in = sock.getInputStream();
017             int len;
018             byte b[] = new byte[100];
019             while ((len = in.read(b)) != -1) {
020                 System.out.write(b, 0, len);
021             }
022             in.close();
023             sock.close();
024         } catch (IOException e) {
025             System.err.println(e.toString());
026             System.exit(1);
027         }
028     }
029 }
```

Listing 32.2: Abfrage des DayTime-Services

Das Programm erwartet einen Hostnamen als Argument und gibt diesen an den Konstruktor von [Socket](#) weiter, der eine Verbindung zu diesem Host auf Port 13 erzeugt. Nachdem der Socket steht, wird der [InputStream](#) beschafft. Das Programm gibt dann so lange die vom Server gesendeten Daten aus, bis durch den Rückgabewert -1 angezeigt wird, daß keine weiteren Daten gesendet werden. Nun werden der Eingabestream und der Socket geschlossen und das Programm beendet. Die Ausgabe des Programms ist beispielsweise:

Sat Nov 7 22:58:37 1998

Um in [Listing 32.2](#) den Socket alternativ mit einem [InetAddress](#)-Objekt zu öffnen, wäre [Zeile 015](#) durch den folgenden Code zu ersetzen:

Tip

```
InetAddress addr = InetAddress.getByName(args[0]);
Socket sock = new Socket(addr, 13);
```

32.2.3 Lesen und Schreiben von Daten

Nachdem wir jetzt wissen, wie man lesend auf einen Socket zugreift, wollen wir in diesem Abschnitt auch den schreibenden Zugriff vorstellen. Dazu schreiben wir ein Programm, das eine Verbindung zum *ECHO-Service* auf Port 7 herstellt. Das Programm liest so lange die Eingaben des Anwenders und sendet sie an den Server, bis das Kommando **QUIT** eingegeben wird. Der Server liest die Daten zeilenweise und sendet sie unverändert an unser Programm zurück, von dem sie auf dem Bildschirm ausgegeben werden. Um Lese- und Schreibzugriffe zu entkoppeln, verwendet das Programm einen separaten Thread, der die eingehenden Daten liest und auf dem Bildschirm ausgibt. Dieser läuft unabhängig vom Vordergrund-Thread, in dem die Benutzereingaben abgefragt und an den Server gesendet werden.

```

001 /* EchoClient.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class EchoClient
007 {
008     public static void main(String args[])
009     {
010         if (args.length != 1) {
011             System.err.println("Usage: java EchoClient <host>");
012             System.exit(1);
013         }
014         try {
015             Socket sock = new Socket(args[0], 7);
016             InputStream in = sock.getInputStream();
017             OutputStream out = sock.getOutputStream();
018             //Timeout setzen
019             sock.setSoTimeout(300);
020             //Ausgabethread erzeugen
021             OutputThread th = new OutputThread(in);
022             th.start();
023             //Schleife für Benutzereingaben
024             BufferedReader conin = new BufferedReader(
025                 new InputStreamReader(System.in));
026             String line = "";
027             while (true) {
028                 //Eingabezeile lesen
029                 line = conin.readLine();
030                 if (line.equalsIgnoreCase("QUIT")) {
031                     break;
032                 }
033                 //Eingabezeile an ECHO-Server schicken
034                 out.write(line.getBytes());
035                 out.write('\r');
036                 out.write('\n');
037                 //Ausgabe abwarten
038                 th.yield();
039             }
040             //Programm beenden
041             System.out.println("terminating output thread...");
042             th.requestStop();
043             th.yield();
044             try {
045                 Thread.sleep(1000);
046             } catch (InterruptedException e) {
047             }
048             in.close();
049             out.close();
050             sock.close();
051         } catch (IOException e) {
052             System.err.println(e.toString());
053             System.exit(1);
054         }
055     }
056 }
057
058 class OutputThread
059 extends Thread
060 {
061     InputStream in;
062     boolean      stoprequested;
063
064     public OutputThread(InputStream in)

```



```

065 {
066     super();
067     this.in = in;
068     stoprequested = false;
069 }
070
071 public synchronized void requestStop()
072 {
073     stoprequested = true;
074 }
075
076 public void run()
077 {
078     int len;
079     byte b[] = new byte[100];
080     try {
081         while (!stoprequested) {
082             try {
083                 if ((len = in.read(b)) == -1) {
084                     break;
085                 }
086                 System.out.write(b, 0, len);
087             } catch (InterruptedException e) {
088                 //nochmal versuchen
089             }
090         }
091     } catch (IOException e) {
092         System.err.println("OutputThread: " + e.toString());
093     }
094 }
095 }

```

Listing 32.3: Lesender und schreibender Zugriff auf einen Socket

Eine Beispielsession mit dem Programm könnte etwa so aussehen (Benutzereingaben sind fett gedruckt):

```

guido_k@pcl:/home/guido_k/nettest > java EchoClient localhost
hello
hello
world
world
12345
12345
quit
closing output thread...

```

Wie im vorigen Beispiel wird zunächst ein Socket zu dem als Argument angegebenen Host geöffnet. Das Programm beschafft dann Ein- und Ausgabestreams zum Senden und Empfangen von Daten. Der Aufruf von [setSoTimeout](#) gibt die maximale Wartezeit bei einem lesenden Zugriff auf den Socket an (300 ms.). Wenn bei einem [read](#) auf den [InputStream](#) nach Ablauf dieser Zeit noch keine Daten empfangen wurden, terminiert die Methode mit einer [InterruptedException](#); wir kommen darauf gleich zurück. Nun erzeugt das Programm den Lesethread und übergibt ihm den Eingabestream. In der nun folgenden Schleife ([Zeile 027](#)) werden so lange Eingabezeilen gelesen und an den Server gesendet, bis der Anwender das Programm mit [QUIT](#) beendet.

Das Programm wurde auf einer LINUX-Version entwickelt, die noch kein präemptives Multithreading unterstützt. Die verschiedenen Aufrufe von [yield](#) dienen dazu, die Kontrolle an den Lesethread zu übergeben. Ohne diesen Aufruf würde der Lesethread gar nicht zum Zuge kommen und das Programm würde keine Daten vom Socket lesen. Auf Systemen, die präemptives Multithreading unterstützen, sind diese Aufrufe nicht notwendig.

Warnung

Die Klasse [OutputThread](#) implementiert den Thread zum Lesen und Ausgeben der Daten. Da die Methode [stop](#) der Klasse [Thread](#) im JDK 1.2 als *deprecated* markiert wurde, müssen wir mit Hilfe der Variable [stoprequested](#) etwas mehr Aufwand treiben, um den Thread beenden zu können. [stoprequested](#) steht normalerweise auf [false](#) und wird beim Beenden des Programms durch Aufruf von [requestStop](#) auf [true](#) gesetzt. In der Hauptschleife des Threads wird diese Variable periodisch abgefragt, um die Schleife bei Bedarf abbrechen zu können ([Zeile 081](#)).

Problematisch bei dieser Technik ist lediglich, daß der Aufruf von [read](#) normalerweise so lange blockiert, bis weitere Zeichen verfügbar sind. Steht das Programm also in [Zeile 083](#), so hat ein Aufruf [requestStop](#) zunächst keine Wirkung. Da das Hauptprogramm in [Zeile 048](#) die Streams und den Socket schließt, würde es zu einer [SocketException](#) kommen. Unser Programm verhindert das durch den Aufruf von [setSoTimeout](#) in

[Zeile 019](#). Dadurch wird ein Aufruf von [read](#) nach spätestens 300 ms. mit einer [InterruptedException](#) beendet. Diese Ausnahme wird in [Zeile 087](#) abgefangen, um anschließend vor dem nächsten Schleifendurchlauf die Variable `stoprequested` erneut abzufragen.

32.2.4 Zugriff auf einen Web-Server

Die Kommunikation mit einem Web-Server erfolgt über das HTTP-Protokoll, wie es in den RFCs 1945 und 2068 beschrieben wurde. Ein Web-Server läuft normalerweise auf TCP-Port 80 (manchmal läuft er zusätzlich auch auf dem UDP-Port 80) und kann wie jeder andere Server über einen Client-Socket angesprochen werden. Wir wollen an dieser Stelle nicht auf Details eingehen, sondern nur die einfachste und wichtigste Anwendung eines Web-Servers zeigen, nämlich das Übertragen einer Seite. Ein Web-Server ist in seinen Grundfunktionen ein recht einfaches Programm, dessen Hauptaufgabe darin besteht, angeforderte Seiten an seine Clients zu versenden. Kompliziert wird er vor allem durch die Vielzahl der mittlerweile eingebauten Zusatzfunktionen, wie beispielsweise Logging, Server-Scripting, Server-Side-Includes, Security- und Tuning-Features usw.

Fordert ein Anwender in seinem Web-Browser eine Seite an, so wird diese Anfrage vom Browser als [GET](#)-Transaktion an den Server geschickt. Um beispielsweise die Seite <http://www.gkrueger.com/index.html> zu laden, wird folgendes Kommando an den Server www.gkrueger.com gesendet:

```
GET /index.html
```

Der erste Teil gibt den Kommandonamen an, dann folgt die gewünschte Datei. Die Zeile muß mit einer CRLF-Sequenz abgeschlossen werden, ein einfaches `\n` reicht nicht aus. Der Server versucht nun die angegebene Datei zu laden und überträgt sie an den Client. Ist der Client ein Web-Browser, wird er den darin befindlichen HTML-Code interpretieren und auf dem Bildschirm anzeigen. Befinden sich in der Seite Verweise auf Images, Applets oder Frames, so fordert der Browser die fehlenden Seiten in weiteren GET-Transaktionen von deren Servern ab.

Die Struktur des GET-Kommandos wurde mit der Einführung von HTTP 1.0 etwas erweitert. Zusätzlich wird nun am Ende der Zeile eine Versionskennung und wahlweise in den darauffolgenden Zeilen weitere Headerzeilen mit Zusatzinformationen mitgeschickt. Nachdem die letzte Headerzeile gesendet wurde, folgt eine leere Zeile (also ein alleinstehendes CRLF), um das Kommandoende anzuzeigen. HTTP 1.0 ist weit verbreitet, und das obige Kommando würde von den meisten Browsern in folgender Form gesendet werden (jede der beiden Zeilen muß mit CRLF abgeschlossen werden):

```
GET /index.html HTTP/1.0
```

Wird HTTP/1.0 verwendet, ist auch die Antwort des Servers etwas komplexer. Anstatt lediglich den Inhalt der Datei zu senden, liefert der Server seinerseits einige Headerzeilen mit Zusatzinformationen, wie beispielsweise den Server-Typ, das Datum der letzten Änderung oder den MIME-Typ der Datei. Auch hier ist jede Headerzeile mit einem CRLF abgeschlossen, und nach der letzten Headerzeile folgt eine Leerzeile. Erst dann beginnt der eigentliche Dateiinhalt.

Das folgende Programm kann dazu verwendet werden, eine Datei von einem Web-Server zu laden. Es wird mit einem Host- und einem Dateinamen als Argument aufgerufen und lädt die Seite vom angegebenen Server. Das Ergebnis wird (mit allen Headerzeilen) auf dem Bildschirm angezeigt.

[Listing3204.java](#)

```
001 /* Listing3204.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class Listing3204
007 {
008     public static void main(String args[])
009     {
010         if (args.length != 2) {
011             System.err.println(
012                 "Usage: java Listing3204 <host> <file>"
013             );
014             System.exit(1);
015         }
016         try {
017             Socket sock = new Socket(args[0], 80);
018             OutputStream out = sock.getOutputStream();
019             InputStream in = sock.getInputStream();
020             //GET-Kommando senden
021             String s = "GET " + args[1] + " HTTP/1.0" + "\r\n\r\n";
022             out.write(s.getBytes());
023             //Ausgabe lesen und anzeigen
024             int len;
025             byte b[] = new byte[100];
```

```
026     while ((len = in.read(b)) != -1) {
027         System.out.write(b, 0, len);
028     }
029     //Programm beenden
030     in.close();
031     out.close();
032     sock.close();
033 } catch (IOException e) {
034     System.err.println(e.toString());
035     System.exit(1);
036 }
037 }
038 }
```

Listing 32.4: Laden einer Seite von einem Web-Server

Wird das Programm beispielsweise auf einem SUSE-Linux 5.2 mit frisch installiertem *Apache-Server* mit `localhost` und `/index.html` als Argument aufgerufen, so beginnt seine Ausgabe wie folgt:

```
HTTP/1.1 200 OK
Date: Sun, 08 Nov 1998 18:26:13 GMT
Server: Apache/1.2.5 S.u.S.E./5.1
Last-Modified: Sun, 24 May 1998 00:46:46 GMT
ETag: "e852-45c-35676df6"
Content-Length: 1116
Accept-Ranges: bytes
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
<TITLE>Apache HTTP Server - Beispielseite</TITLE>
</HEAD>
<BODY bgcolor=#ffffff>
<H1> Der Apache WWW Server </H1> <BR>
Diese Seite soll nur als Beispiel dienen.
Die <A HREF="./manual/">Dokumentation zum
Apache-Server</A> finden Sie hier.
<P>
...
```

32.3 Server-Sockets

- [32.3 Server-Sockets](#)
 - [32.3.1 Die Klasse ServerSocket](#)
 - [32.3.2 Verbindungen zu mehreren Clients](#)
 - [32.3.3 Entwicklung eines einfachen Web-Servers](#)

32.3.1 Die Klasse ServerSocket

In den bisherigen Abschnitten hatten wir uns mit dem Entwurf von *Netzwerk-Clients* beschäftigt. Nun wollen wir uns das passende Gegenstück ansehen, uns also mit der Entwicklung von *Servern* beschäftigen. Glücklicherweise ist auch das in Java recht einfach. Der wesentliche Unterschied liegt in der Art des Verbindungsaufbaus, für den es eine spezielle Klasse [ServerSocket](#) gibt. Diese Klasse stellt Methoden zur Verfügung, um auf einen eingehenden Verbindungswunsch zu warten und nach erfolgtem Verbindungsaufbau einen [Socket](#) zur Kommunikation mit dem Client zurückzugeben. Bei der Klasse [ServerSocket](#) sind im wesentlichen der Konstruktor und die Methode [accept](#) von Interesse:

```

public ServerSocket(int port)
    throws IOException

public Socket accept()
    throws IOException

```

Der Konstruktor erzeugt einen [ServerSocket](#) für einen bestimmten Port, also einen bestimmten Typ von Serveranwendung (siehe [Abschnitt 32.1.4](#)). Anschließend wird die Methode [accept](#) aufgerufen, um auf einen eingehenden Verbindungswunsch zu warten. [accept](#) blockiert so lange, bis sich ein Client bei der Serveranwendung anmeldet (also einen Verbindungsaufbau zu unserem Host unter der Portnummer, die im Konstruktor angegeben wurde, initiiert). Ist der Verbindungsaufbau erfolgreich, liefert [accept](#) ein [Socket](#)-Objekt, das wie bei einer Client-Anwendung zur Kommunikation mit der Gegenseite verwendet werden kann. Anschließend steht der [ServerSocket](#) für einen weiteren Verbindungsaufbau zur Verfügung oder kann mit [close](#) geschlossen werden.

Wir wollen uns die Konstruktion von Servern an einem Beispiel ansehen. Dazu soll ein einfacher ECHO-Server geschrieben werden, der auf Port 7 auf Verbindungswünsche wartet. Alle eingehenden Daten sollen unverändert an den Client zurückgeschickt werden. Zur Kontrolle sollen sie ebenfalls auf die Konsole ausgegeben werden:

```

001 /* SimpleEchoServer.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class SimpleEchoServer
007 {
008     public static void main(String args[])
009     {
010         try {
011             System.out.println("Warte auf Verbindung auf Port 7...");
012             ServerSocket echod = new ServerSocket(7);
013             Socket socket = echod.accept();
014             System.out.println("Verbindung hergestellt");
015             InputStream in = socket.getInputStream();
016             OutputStream out = socket.getOutputStream();
017             int c;
018             while ((c = in.read()) != -1) {
019                 out.write((char)c);
020                 System.out.print((char)c);
021             }
022             System.out.println("Verbindung beenden");
023             socket.close();
024             echod.close();
025         } catch (IOException e) {
026             System.err.println(e.toString());

```

```

027         System.exit(1);
028     }
029 }
030 }

```

Listing 32.5: Ein ECHO-Server für Port 7

Wird der Server gestartet, kann via Telnet oder mit dem [EchoClient](#) aus [Listing 32.3](#) auf den Server zugegriffen werden:

```
telnet localhost 7
```

Wenn der Server läuft, werden alle eingegebenen Zeichen direkt vom Server zurückgesendet und als Echo in Telnet angezeigt. Läuft er nicht, gibt es beim Verbindungsaufbau eine Fehlermeldung.

Wird das Programm unter UNIX gestartet, kann es möglicherweise Probleme gegen. Einerseits kann es sein, daß bereits ein ECHO-Server auf Port 7 läuft. Er könnte nötigenfalls per Eintrag in [inetd.conf](#) oder ähnlichen Konfigurationsdateien vorübergehend deaktiviert werden. Andererseits dürfen Server auf Ports kleiner 1024 nur mit Root-Berechtigung gestartet werden. Ein normaler Anwender darf dagegen nur Server-Ports größer 1023 verwenden.

Hinweis

32.3.2 Verbindungen zu mehreren Clients

Wir wollen das im vorigen Abschnitt vorgestellte Programm nun in mehrfacher Hinsicht erweitern:

- Der Server soll mehr als einen Client gleichzeitig bedienen können. Die Clients sollen zur besseren Unterscheidung durchnummeriert werden.
- Beim Verbindungsaufbau soll der Client eine Begrüßungsmeldung erhalten.
- Für jeden Client soll ein eigener Thread angelegt werden.

Um diese Anforderungen zu erfüllen, verändern wir das obige Programm ein wenig. Im Hauptprogramm wird nun nur noch der [ServerSocket](#) erzeugt und in einer Schleife jeweils mit [accept](#) auf einen Verbindungswunsch gewartet. Nach dem Verbindungsaufbau erfolgt die weitere Bearbeitung nicht mehr im Hauptprogramm, sondern es wird ein neuer Thread mit dem Verbindungs-Socket als Argument erzeugt. Dann wird der Thread gestartet und erledigt die gesamte Kommunikation mit dem Client. Beendet der Client die Verbindung, wird auch der zugehörige Thread beendet. Das Hauptprogramm braucht sich nur noch um den Verbindungsaufbau zu kümmern und ist von der eigentlichen Client-Kommunikation vollständig befreit.

[EchoServer.java](#)

```

001 /* EchoServer.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class EchoServer
007 {
008     public static void main(String args[])
009     {
010         int cnt = 0;
011         try {
012             System.out.println("Warte auf Verbindungen auf Port 7...");
013             ServerSocket echod = new ServerSocket(7);
014             while (true) {
015                 Socket socket = echod.accept();
016                 (new EchoClientThread(++cnt, socket)).start();
017             }
018         } catch (IOException e) {
019             System.err.println(e.toString());
020             System.exit(1);
021         }
022     }
023 }
024
025 class EchoClientThread
026 extends Thread
027 {
028     private int    name;
029     private Socket socket;
030
031     public EchoClientThread(int name, Socket socket)

```

```

032 {
033     this.name    = name;
034     this.socket = socket;
035 }
036
037 public void run()
038 {
039     String msg = "EchoServer: Verbindung " + name;
040     System.out.println(msg + " hergestellt");
041     try {
042         InputStream in = socket.getInputStream();
043         OutputStream out = socket.getOutputStream();
044         out.write((msg + "\r\n").getBytes());
045         int c;
046         while ((c = in.read()) != -1) {
047             out.write((char)c);
048             System.out.print((char)c);
049         }
050         System.out.println("Verbindung " + name + " wird beendet");
051         socket.close();
052     } catch (IOException e) {
053         System.err.println(e.toString());
054     }
055 }
056 }

```

Listing 32.6: Eine verbesserte Version des Echo-Servers

Zur besseren Übersicht werden alle Client-Verbindungen durchnummeriert und als erstes Argument an den Thread übergeben. Unmittelbar nach dem Verbindungsaufbau wird diese Meldung auf der Server-Konsole ausgegeben und an den Client geschickt. Anschließend wird in einer Schleife jedes vom Client empfangene Zeichen an diesen zurückgeschickt, bis er von sich aus die Verbindung unterbricht. Man kann den Server leicht testen, indem man mehrere Telnet-Sessions zu ihm aufbaut. Jeder einzelne Client sollte eine Begrüßungsmeldung mit einer eindeutigen Nummer erhalten und autonom mit dem Server kommunizieren können. Der Server sendet alle Daten zusätzlich an die Konsole und gibt sowohl beim Starten als auch beim Beenden eine entsprechende Meldung auf der Konsole aus.

32.3.3 Entwicklung eines einfachen Web-Servers

In [Abschnitt 32.2.4](#) war schon angeklungen, daß ein Web-Server in seinen Grundfunktionen so einfach aufgebaut ist, daß wir uns hier eine experimentelle Implementierung ansehen können. Die Kommunikation zwischen einem Browser und einem Web-Server entspricht etwa folgendem Schema:

- Der Web-Browser baut eine Verbindung zum Server auf.
- Er schickt eine Seitenanforderung (und zusätzliche Informationen) in Form eines *Requests gemäß HTTP-Spezifikation*.
- Der Server analysiert den Request und schickt die gewünschte Datei (bzw. eine Fehlermeldung) an den Browser.
- Der Server beendet die Verbindung.

Hat der Browser auf diese Weise eine HTML-Seite erhalten, interpretiert er den HTML-Code und zeigt die Seite formatiert auf dem Bildschirm an. Enthält die Datei IMG-, APPLET- oder ähnliche Elemente, werden diese in derselben Weise vom Server angefordert und in die Seite eingebaut. Die wichtigste Aufgabe des Servers besteht also darin, eine Datei an den Client zu übertragen. Wir wollen uns zunächst das Listing ansehen und dann auf Details der Implementierung eingehen:

[ExperimentalWebServer.java](#)

```

001 /* ExperimentalWebServer.java */
002
003 import java.io.*;
004 import java.util.*;
005 import java.net.*;
006
007 /**
008  * Ein ganz einfacher Web-Server auf TCP und einem
009  * beliebigen Port. Der Server ist in der Lage,
010  * Seitenanforderungen lokal zu dem Verzeichnis,
011  * aus dem er gestartet wurde, zu bearbeiten. Wurde
012  * der Server z.B. im Verzeichnis c:\tmp gestartet, so
013  * würde eine Seitenanforderung
014  * http://localhost:80/test/index.html die Datei
015  * c:\tmp\test\index.html laden. CGIs, SSIs, Servlets

```

```

016 * oder ähnliches wird nicht unterstützt.
017 * <p>
018 * Die Dateitypen .htm, .html, .gif, .jpg und .jpeg werden
019 * erkannt und mit korrekten MIME-Headern übertragen, alle
020 * anderen Dateien werden als "application/octet-stream"
021 * übertragen. Jeder Request wird durch einen eigenen
022 * Client-Thread bearbeitet, nach Übertragung der Antwort
023 * schließt der Server den Socket. Antworten werden mit
024 * HTTP/1.0-Header gesendet.
025 */
026 public class ExperimentalWebServer
027 {
028     public static void main(String args[])
029     {
030         if (args.length != 1) {
031             System.err.println(
032                 "Usage: java ExperimentalWebServer <port>"
033             );
034             System.exit(1);
035         }
036         try {
037             int port = Integer.parseInt(args[0]);
038             System.out.println("Listening to port " + port);
039             int calls = 0;
040             ServerSocket httpd = new ServerSocket(port);
041             while (true) {
042                 Socket socket = httpd.accept();
043                 (new BrowserClientThread(++calls, socket)).start();
044             }
045         } catch (IOException e) {
046             System.err.println(e.toString());
047             System.exit(1);
048         }
049     }
050 }
051
052 /**
053  * Die Thread-Klasse für die Client-Verbindung.
054  */
055 class BrowserClientThread
056 extends Thread
057 {
058     static final String mimetypes[][] = {
059         {"html", "text/html"},
060         {"htm", "text/html"},
061         {"txt", "text/plain"},
062         {"gif", "image/gif"},
063         {"jpg", "image/jpeg"},
064         {"jpeg", "image/jpeg"}
065     };
066
067     private Socket      socket;
068     private int         id;
069     private PrintStream out;
070     private InputStream in;
071     private String      cmd;
072     private String      url;
073     private String      httpversion;
074
075     /**
076      * Erzeugt einen neuen Client-Thread mit der angegebenen
077      * id und dem angegebenen Socket.
078      */
079     public BrowserClientThread(int id, Socket socket)
080     {

```

```

081     this.id      = id;
082     this.socket = socket;
083 }
084
085 /**
086  * Hauptschleife für den Thread.
087  */
088 public void run()
089 {
090     try {
091         System.out.println(id + ": Incoming call...");
092         out = new PrintStream(socket.getOutputStream());
093         in = socket.getInputStream();
094         readRequest();
095         createResponse();
096         socket.close();
097         System.out.println(id + ": Closed.");
098     } catch (IOException e) {
099         System.out.println(id + ": " + e.toString());
100         System.out.println(id + ": Aborted.");
101     }
102 }
103
104 /**
105  * Liest den nächsten HTTP-Request vom Browser ein.
106  */
107 private void readRequest()
108 throws IOException
109 {
110     //Request-Zeilen lesen
111     Vector request = new Vector(10);
112     StringBuffer sb = new StringBuffer(100);
113     int c;
114     while ((c = in.read()) != -1) {
115         if (c == '\r') {
116             //ignore
117         } else if (c == '\n') { //line terminator
118             if (sb.length() <= 0) {
119                 break;
120             } else {
121                 request.addElement(sb);
122                 sb = new StringBuffer(100);
123             }
124         } else {
125             sb.append((char)c);
126         }
127     }
128     //Request-Zeilen auf der Konsole ausgeben
129     Enumeration e = request.elements();
130     while (e.hasMoreElements()) {
131         sb = (StringBuffer)e.nextElement();
132         System.out.println("< " + sb.toString());
133     }
134     //Kommando, URL und HTTP-Version extrahieren
135     String s = ((StringBuffer)request.elementAt(0)).toString();
136     cmd = "";
137     url = "";
138     httpversion = "";
139     int pos = s.indexOf(' ');
140     if (pos != -1) {
141         cmd = s.substring(0, pos).toUpperCase();
142         s = s.substring(pos + 1);
143         //URL
144         pos = s.indexOf(' ');
145         if (pos != -1) {

```



```

146     url = s.substring(0, pos);
147     s = s.substring(pos + 1);
148     //HTTP-Version
149     pos = s.indexOf('\r');
150     if (pos != -1) {
151         httpversion = s.substring(0, pos);
152     } else {
153         httpversion = s;
154     }
155 } else {
156     url = s;
157 }
158 }
159 }
160
161 /**
162  * Request bearbeiten und Antwort erzeugen.
163  */
164 private void createResponse()
165 {
166     if (cmd.equals("GET")) {
167         if (!url.startsWith("/")) {
168             httpError(400, "Bad Request");
169         } else {
170             //MIME-Typ aus Dateierweiterung bestimmen
171             String mimestring = "application/octet-stream";
172             for (int i = 0; i < mimetypes.length; ++i) {
173                 if (url.endsWith(mimetypes[i][0])) {
174                     mimestring = mimetypes[i][1];
175                     break;
176                 }
177             }
178             //URL in lokalen Dateinamen konvertieren
179             String fsep = System.getProperty("file.separator", "/");
180             StringBuffer sb = new StringBuffer(url.length());
181             for (int i = 1; i < url.length(); ++i) {
182                 char c = url.charAt(i);
183                 if (c == '/') {
184                     sb.append(fsep);
185                 } else {
186                     sb.append(c);
187                 }
188             }
189             try {
190                 FileInputStream is = new FileInputStream(sb.toString());
191                 //HTTP-Header senden
192                 out.print("HTTP/1.0 200 OK\r\n");
193                 System.out.println("> HTTP/1.0 200 OK");
194                 out.print("Server: ExperimentalWebServer 0.5\r\n");
195                 System.out.println(
196                     "> Server: ExperimentalWebServer 0.5"
197                 );
198                 out.print("Content-type: " + mimestring + "\r\n\r\n");
199                 System.out.println("> Content-type: " + mimestring);
200                 //Dateiinhalt senden
201                 byte buf[] = new byte[256];
202                 int len;
203                 while ((len = is.read(buf)) != -1) {
204                     out.write(buf, 0, len);
205                 }
206                 is.close();
207             } catch (FileNotFoundException e) {
208                 httpError(404, "Error Reading File");
209             } catch (IOException e) {
210                 httpError(404, "Not Found");

```

```

211         } catch (Exception e) {
212             httpError(404, "Unknown exception");
213         }
214     }
215     } else {
216         httpError(501, "Not implemented");
217     }
218 }
219
220 /**
221  * Eine Fehlerseite an den Browser senden.
222  */
223 private void httpError(int code, String description)
224 {
225     out.print("HTTP/1.0 " + code + " " + description + "\r\n");
226     out.print("Content-type: text/html\r\n\r\n");
227     out.println("<html>");
228     out.println("<head>");
229     out.println("<title>ExperimentalWebServer-Error</title>");
230     out.println("</head>");
231     out.println("<body>");
232     out.println("<h1>HTTP/1.0 " + code + "</h1>");
233     out.println("<h3>" + description + "</h3>");
234     out.println("</body>");
235     out.println("</html>");
236 }
237 }

```

Listing 32.7: Ein experimenteller Web-Server

Der Web-Server besteht aus den beiden Klassen `ExperimentalWebServer` und `BrowserClientThread`, die nach dem in [Abschnitt 32.3.2](#) vorgestellten Muster aufgebaut sind. Nachdem in `ExperimentalWebServer` eine Verbindung aufgebaut wurde, wird ein neuer Thread erzeugt und die weitere Bearbeitung des Requests an ein Objekt der Klasse `BrowserClientThread` delegiert. Der in [run](#) liegende Code beschafft zunächst die Ein- und Ausgabestreams zur Kommunikation mit dem Socket und ruft dann die beiden Methoden `readRequest` und `createResponse` auf. Anschließend wird der Socket geschlossen und der Thread beendet.

In `readRequest` wird der HTTP-Request des Browsers gelesen, der aus mehreren Zeilen besteht. In der ersten wird die eigentliche Dateianforderung angegeben, die übrigen liefern Zusatzinformationen wie den Typ des Browsers, akzeptierte Dateiformate und ähnliches. Alle Zeilen werden mit CRLF abgeschlossen, nach der letzten Zeile des Requests wird eine Leerzeile gesendet. Entsprechend der Empfehlung in RFC1945 ignoriert unser Parser die '\r'-Zeichen und erkennt das Zeilenende anhand eines '\n'. So arbeitet er auch dann noch korrekt, wenn ein Client die Headerzeilen versehentlich mit einem einfachen LF abschließt.

Ein typischer Request könnte etwa so aussehen (in diesem Beispiel wurde er von Netscape 4.04 unter Windows 95 generiert):

```

GET /ansisys.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.04 [en] (Win95; I)
Host: localhost:80
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
HTTP/1.0 200 OK
Server: ExperimentalWebServer 0.5
Content-type: text/html

```

Unser Web-Server liest den Request zeilenweise in den `Vector request` ein und gibt alle Zeilen zur Kontrolle auf der Konsole aus. Anschließend wird das erste Element extrahiert und in die Bestandteile *Kommando*, *URL* (Dateiname) und *HTTP-Version* zerlegt. Diese Informationen werden zur weiteren Verarbeitung in den Membervariablen `cmd`, `url` und `httpversion` gespeichert.

Nachdem der Request gelesen wurde, wird in `createResponse` die Antwort erzeugt. Zunächst prüft die Methode, ob es sich um ein `GET`-Kommando handelt (HTTP kennt noch andere Kommandos). Ist das nicht der Fall, wird durch Aufruf von `httpError` eine Fehlerseite an den Browser gesendet. Andernfalls fährt die Methode mit der Bestimmung des Dateityps fort. Der Dateityp wird mit Hilfe der Arraykonstante `mimetypes` anhand der Dateierweiterung bestimmt und in einen passenden *MIME-Typ* konvertiert, der im Antwortheader an den Browser übertragen wird. Der Browser entscheidet anhand dieser Information, was mit der nachfolgend übertragenen Datei zu tun ist (Anzeige als Text, Anzeige als Grafik, Speichern in einer Datei usw.). Wird eine Datei angefordert, deren Erweiterung nicht bekannt ist, sendet der Server sie als [application/octet-stream](#) an den Browser, damit dieser dem Anwender die Möglichkeit geben kann, die Datei auf der Festplatte zu speichern.

Nun wandelt der Server den angegebenen Dateinamen gemäß den Konventionen seines eigenen Betriebssystems um. Dazu wird das erste "/" aus dem Dateinamen entfernt (alle Dateien werden lokal zu dem Verzeichnis geladen, aus dem der Server gestartet wurde) und alle "/" innerhalb des Pfadnamens werden in den lokalen Pfadseparator konvertiert (unter MS-DOS ist das beispielsweise der Backslash). Dann wird die Datei mit einem `FileInputStream` geöffnet und der HTTP-Header und der Dateiinhalt an den Client gesendet. Konnte die Datei nicht geöffnet werden, wird eine Ausnahme ausgelöst und der Server sendet eine Fehlerseite.

Der vom Server gesendete Header ist ähnlich aufgebaut wie der Request-Header des Clients. Er enthält mehrere Zeilen, die durch CRLF-Sequenzen voneinander getrennt sind. Nach der letzten Headerzeile folgt eine Leerzeile, also zwei aufeinanderfolgende CRLF-Sequenzen. HTTP 1.0 und 1.1 spezifizieren eine ganze Reihe von (optionalen) Headerelementen, von denen wir lediglich die Versionskennung, unseren Servernamen und den MIME-Bezeichner mit der Typkennung der gesendeten Datei an den Browser übertragen. Unmittelbar nach dem Ende des Headers wird der Dateiinhalt übertragen. Eine Umkodierung erfolgt dabei normalerweise nicht, alle Bytes werden unverändert übertragen.

Unser Server kann sehr leicht getestet werden. Am einfachsten legt man ein neues Unterverzeichnis an und kopiert die übersetzten Klassendateien und einige HTML-Dateien in dieses Verzeichnis. Nun kann der Server wie jedes andere Java-Programm gestartet werden. Beim Aufruf ist zusätzlich die Portnummer als Argument anzugeben:

```
java ExperimentalWebServer 80
```

Nun kann ein normaler Web-Browser verwendet werden, um Dateien vom Server zu laden. Befindet sich beispielsweise eine Datei `index.html` im Server-Verzeichnis und läuft der Server auf derselben Maschine wie der Browser, kann die Datei über die Adresse `http://localhost/index.html` im Browser geladen werden. Auch über das lokale Netz des Unternehmens oder das Internet können leicht Dateien geladen werden. Hat der Host, auf dem der Server läuft, keinen Nameserver-Eintrag, kann statt dessen auch direkt seine IP-Adresse im Browser angegeben werden.

Auf einem UNIX-System darf ein Server die Portnummer 80 nur verwenden, wenn er Root-Berechtigung hat. Ist das nicht der Fall, kann der Server alternativ auf einem Port größer 1023 gestartet werden:

Hinweis

```
java ExperimentalWebServer 7777
```

Im Browser muß die Adresse dann ebenfalls um die Portnummer ergänzt werden:
`http://localhost:7777/index.html`.

32.4 Daten mit Hilfe der Klasse URL lesen

- [32.4 Daten mit Hilfe der Klasse URL lesen](#)

Die Klasse [URL](#) wurde bereits in [Abschnitt 25.4.1](#) behandelt. Neben den dort beschriebenen Möglichkeiten besitzt sie Methoden, um Daten von der Quelle zu lesen, die durch den URL adressiert wird:

[java.net.URL](#)

```
public final InputStream openStream()
    throws IOException
```

```
public final Object getContent()
    throws IOException
```

```
public URLConnection openConnection()
    throws IOException
```

Mit [openStream](#) wird ein [InputStream](#) geliefert, der wie die Methode [getInputStream](#) der Klasse [Socket](#) zum Lesen der Quelldaten verwendet werden kann. [getContent](#) versucht darüber hinaus, die Daten zu interpretieren. Dazu können *Content Handler Factories* registriert werden, die beispielsweise Text-, Image- oder Archivdateien interpretieren und ein dazu passendes Java-Objekt liefern. Die Methode [openConnection](#) stellt eine Vorstufe von [getContent](#) dar. Sie liefert ein Objekt des Typs [URLConnection](#), das eine Abstraktion einer protokollspezifischen Verbindung zwischen einem Java-Programm und einem URL darstellt.

Als einfaches Beispiel wollen wir uns das folgende Programm [SaveURL](#) ansehen. Es wird mit einem URL und einer Datei als Argument aufgerufen. Mit Hilfe der Klasse [URL](#) stellt das Programm eine Verbindung zur angegebenen URL her und beschafft durch Aufruf von [openStream](#) einen [InputStream](#). Mit seiner Hilfe wird die Quelle gelesen und das Ergebnis in die als zweites Argument angegebene Datei geschrieben:

[SaveURL.java](#)

```
001 /* SaveURL.java */
002
003 import java.net.*;
004 import java.io.*;
005
006 public class SaveURL
007 {
008     public static void main(String args[])
009     {
010         if (args.length != 2) {
011             System.err.println(
012                 "Usage: java SaveURL <url> <file>"
013             );
014             System.exit(1);
015         }
016         try {
017             URL url = new URL(args[0]);
018             OutputStream out = new FileOutputStream(args[1]);
019             InputStream in = url.openStream();
020             int len;
021             byte b[] = new byte[100];
022             while ((len = in.read(b)) != -1) {
023                 out.write(b, 0, len);
024             }
025             out.close();
026             in.close();
027         } catch (MalformedURLException e) {
028             System.err.println(e.toString());
029             System.exit(1);
030         } catch (IOException e) {
031             System.err.println(e.toString());
032             System.exit(1);
033         }
034     }
035 }
```

```
034     }
035 }
```

Listing 32.8: Daten von einem URL lesen

Das Programm kann nun leicht verwendet werden, um den Inhalt beliebiger URLs auf der Festplatte abzuspeichern. Die folgenden beiden Aufrufe zeigen den Download der Hauptseite des Java-Servers von SUN und das Laden einer Testseite von unserem in [Abschnitt 32.3.3](#) vorgestellten Web-Server:

```
java SaveURL http://java.sun.com x.html

java SaveURL http://localhost/index.html y.html
```

32.5 Zusammenfassung

- [32.5 Zusammenfassung](#)

In diesem Kapitel wurden folgende Themen behandelt:

- Grundlagen und Anwendungen der Netzwerkprogrammierung
- Das ISO/OSI-7-Schichten-Modell und das vereinfachte Vier-Ebenen-Modell
- Grundlagen der Protokolle TCP und IP
- IP-Adressen, Netztypen und Domain-Namen
- Portnummern und Applikationen
- Request For Comments
- Das Konzept der Sockets
- Die Klasse [InetAddress](#) zur Adressierung von Socket-Verbindungen
- Die besonderen Adressen `localhost` und `127.0.0.1`
- Aufbau einer Client-Socket-Verbindung mit Hilfe der Klasse [Socket](#) und dem streambasierten Zugriff auf die Daten
- Kommunikation mit dem DayTime- und Echo-Server
- Lesender Zugriff auf einen Web-Server
- Konstruktion von Serverdiensten mit der Klasse [ServerSocket](#)
- Konstruktion mehrbenutzerfähiger Server mit Threads
- Entwurf eines experimentellen Web-Servers
- Daten mit Hilfe der Klasse [URL](#) lesen

Index

0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------

- 0 -

[4.1.2](#) [26.2.1.3](#)

100 % Pure Java Initiative	1.1
2D-API	15.1
7-Segment-Anzeige	23.2
<APPLET>	25.2.1
@author	4.1.2 26.2.1.3 26.3
@deprecated	26.1.1.3 26.2.1.3 26.3
@exception	4.1.2 26.2.1.3 26.3
@param	4.1.2 26.2.1.3 26.3
@return	4.1.2 26.2.1.3 26.3
@see	4.1.2 26.2.1.3 26.3
@since	26.2.1.3 26.3
@version	4.1.2 26.2.1.3 26.3
_blank	25.4.3
_parent	25.4.3
_self	25.4.3
_top	25.4.3

- A -	
Abhängigkeiten zwischen den Quelldateien	26.1.1.2
Ableiten einer Klasse	7.3.1
abs	7.8.2.4 12.6.2.1
absbottom	25.2.2
absmiddle	25.2.2
abstract	7.6 27.2.2
AbstractList	27.1 27.2.2
AbstractMap	27.6.2
AbstractSet	27.5.2
Abstract Windowing Toolkit	1.2.3 14.1
Abstrakte Klassen und Methoden	7.6
accept	13.5.4 32.3.1 32.3.2
acos	7.8.2.1
action	18.1.1
ACTION_EVENT_MASK	18.2.4
Action-Ereignisse	18.1.6.8
ActionEvent	18.1.2 18.1.6.8 20.5 22.3
Action-Events	20.5
ActionListener	18.1.6.8 20.2 20.5 20.6 22.3
actionPerformed	18.1.6.8 20.5 20.7.2 21.3 22.3 22.9
activeCount	10.5.2
Adapterklasse	18.1.1 18.1.5
add	12.7.3 20.2 20.3 20.4.4 20.6 21.1.3 21.2 21.2.2 21.2.3 21.2.4 21.4 22.1 22.8 22.11 23.3 24.1.2 27.2.1 27.3.2 27.5.1 27.5.2 29.2.3
addActionListener	18.1.6.8 20.5 22.3
addAdjustmentListener	18.1.6.9 22.10
addAll	27.2.1 27.5.1
addComponentListener	18.1.6.5 19.2 19.7
addContainerListener	18.1.6.6
addElement	12.1.1
addFocusListener	18.1.6.1 19.5 19.7
addImage	24.1.1.1
addItem	22.8
addItemListener	18.1.6.10 22.4 22.8
Addition	5.2
addKeyListener	17.3 18.1.6.2 18.2.1 18.2.2.1 18.2.3 19.6 19.7
addMouseListener	2.3 2.6 18.1.4 18.1.6.3 19.3 19.7
addMouseMotionListener	2.3 2.6 18.1.6.4 19.4 19.7
addPoint	14.3.3
addSeparator	20.3
addTextListener	18.1.6.11 22.6

addWindowListener	14.3 18.1.6.7 19.1 19.7
Adjustable	22.10 22.11
ADJUSTMENT_EVENT_MASK	18.2.4
Adjustment-Ereignisse	18.1.6.9 22.10
AdjustmentEvent	18.1.2 18.1.6.9 22.10
AdjustmentEvent.BLOCK_DECREMENT	22.10
AdjustmentEvent.BLOCK_INCREMENT	22.10
AdjustmentEvent.TRACK	22.10
AdjustmentEvent.UNIT_DECREMENT	22.10
AdjustmentEvent.UNIT_INCREMENT	22.10
AdjustmentListener	18.1.6.9 22.10
adjustmentValueChanged	18.1.6.9 22.10
after	12.7.3
aiff-Dateien	1.2.3
AIFF-Format	25.3.1
Aliasing	29.2.1.4
ALIGN	25.2.2
ALL	30.4.7.3
ALT	25.2.2
ALTER TABLE	30.4.7.1
anchor	21.2.4
and	12.4.2
AND	30.4.7.3
andNot	12.4.2
Anhalten eines Threads	10.2.3
Animation	24.2 in Applets
Anlegen eines Fensters	14.2.1
Anonyme Klassen	14.3 18.2.2.2
Anonyme lokale Klasse	27.4.2
Anweisungen	6 8.1.1
Anweisungsobjekt	30.2.2
Anwendungsprotokolle	32.1.2
ANY	30.4.7.3
Apache-Server	32.2.4
append	11.3.2.2 11.4.2 11.5 22.7 29.2.1.1
Applet	2.3 8.1.7 17.1 17.2 17.6 25.1.1 25.1.2 25.1.3 25.1.3.1 25.1.4.3 25.2.3
APPLET	25.2.3
Applet	25.3.1 25.3.2 25.4.3 25.6
APPLET	26.1.3.2
Applet	initialisieren starten stoppen zerstören
AppletContext	25.4.2
Applet-Kontext	25.4.2
Applets	1.2.2 2.2 8.1.7 25
APPLET-Tag	1.2.2 2.3 25.2.1 26.2.2.4
Appletviewer	2.3

appletviewer	2.7 3.5.2.1
Appletviewer	8.1.7 25.2.3 25.4.2
appletviewer	26.1
Appletviewer	26.1.3
appletviewer	26.1.3.2
application/octet-stream	32.3.3
Applikationen	2.2 8.1.6
Applikations-Server	30.1.2.2
ARCHIVE	25.2.2 26.2.2.4
Archivierungswerkzeug	26.2.2
Arial	15.2.1
ArithmeticException	9.3.3
Arithmetische Operatoren	5.2
Array	2.4 4.4 Mehrdimensionales
arraycopy	12.8.6 12.10
ArrayIndexOutOfBoundsException	12.8.6
ArrayList	27.1 27.2.2 27.9 29.2.3
Arrays	12.9 12.10
ArrayStoreException	12.8.6
ASC	30.4.7.3
asin	7.8.2.1
Assoziativitätsregeln	5.1
atan	7.8.2.1
ATM	32.1.2
Attribute von Klassen, Methoden und Variablen	7.4
au-Dateien	1.2.3
AudioClip	25.3.1
AU-Format	25.3.1
Aufruf einer Methode	7.2.2
Ausdrücke	1.2.1
Ausdrucksanweisungen	6.1.4
Ausgabe-Streams	13.2
Ausgabe von Sound	25.3
Auslösen einer Ausnahme	9.1 9.3.3
Ausnahmen	2.4 9
Auswertungsreihenfolge	5.1
Auto-Commit-Modus	30.4.3
Automatisches Speichermanagement	1.2.1
AWT	1.2.3 14.1
AWTEvent	18.1.2 18.2.4 19.2 20.5 22.6
AWTEvent.COMPONENT_EVENT_MASK	23.2.3
AWTEvent.FOCUS_EVENT_MASK	23.2.3
AWTEvent.KEY_EVENT_MASK	23.2.3
AWTEvent.MOUSE_EVENT_MASK	23.2.3

- B -	
Baratz, Alan	1.1
baseline	25.2.2
Bedingtes Kompilieren	6.2.1.4
before	12.7.3
Behandeln einer Ausnahme	9.1
Überladen von Methoden	7.2.5
Überlagern von Methoden	7.3.2
Übersetzen der Beispiele	3.5.2.1
Übersetzen des Quelltextes	3.5.1.3
Beschleunigertasten	20.4.3
BETWEEN	30.4.7.3
Bezeichner	4.1.3
Bildschirmflackern reduzieren	24.2.4
binarySearch	12.9 27.8.1
Bindungsregeln	5.1
binäre Suche	27.8.1
BitmapComponent	24.1.2
Bitmap laden und anzeigen	24.1.1
BitSet	12.4 12.4.1 12.4.2 12.10 27.1 27.5.2
Bitweise Operatoren	5.5
Blobs	30.1
Block	6.1.2 8.1.2
bmp	24.1.1
Booch, Grady	7.3
boolean	1.2.1 4.2 4.2.1 4.2.1.1 4.4.1 4.4.2 4.6 4.7 5.6 5.7.1.1 6.2.1.2 6.3.1.2 7.8.1
Boolean	7.8.1
boolean	7.8.1
Boolean	7.9
boolean	29.2.2 30.2.3 31.3.1
booleanValue	7.8.1.2
Bootstrap Classes	8.2.2.2
BorderLayout	21.1.2 21.1.3 21.2 21.2.3 21.2.6 21.4 22.1
Borland	1.1
bottom	25.2.2
break	1.2.1 6.2.1.4 6.2.2.2 6.3.3.3 6.4 9.2.6 in Schleifen in switch-Anweisung
Breite eines Zeichens	15.3.2
Bubblesort	29.3.1 29.3.2
BufferedInputStream	29.2.4.2
BufferedOutputStream	29.2.4.1 29.3.1
BufferedReader	3.5.3.2 13.3.2 13.3.3 13.3.3.1 13.3.3.2 13.6 29.2.4.2

BufferedWriter	13.2.2 13.2.3 13.2.3.1 13.2.3.2 13.2.3.3 13.6 29.2.4.1
Button	18.1.6.8 20.5 21.3 22.1 22.3 22.6 22.12 25.4.3
byte	4.2 4.2.3 4.4.2 4.7 6.2.2.2
Byte	7.8.1
byte	7.8.1
Byte	7.8.1.3 7.8.1.4 7.9
byte	13.4.4 30.2.3 31.3.1 32.2.1
ByteArrayInputStream	28.3.2
ByteArrayOutputStream	28.3.2
Bytecode	2.2 8.3.1
Byte-Streams	13.1
ByteToCharConverter	13.3.2.1 29.2.4.2

- C -	
Cafe	1.1
Calendar	12.7 12.7.1 12.7.3 12.7.4 12.10
Calendar.AM_PM	12.7.2
Calendar.DAY_OF_MONTH	12.7.2
Calendar.DAY_OF_WEEK	12.7.2
Calendar.DAY_OF_WEEK_IN_MONTH	12.7.2
Calendar.DAY_OF_YEAR	12.7.2
Calendar.DST_OFFSET	12.7.2
Calendar.ERA	12.7.2
Calendar.HOUR	12.7.2
Calendar.HOUR_OF_DAY	12.7.2
Calendar.MILLISECOND	12.7.2
Calendar.MINUTE	12.7.2
Calendar.MONTH	12.7.2
Calendar.SECOND	12.7.2
Calendar.WEEK_OF_MONTH	12.7.2
Calendar.WEEK_OF_YEAR	12.7.2
Calendar.YEAR	12.7.2
Calendar.ZONE_OFFSET	12.7.2
Callback-Methoden	8.1.7
call by reference	7.2.3 7.8.1
call by value	7.2.3
Call-Level-Interface	30.1.1
canRead	13.5.3
Canvas	22.11 23.1 23.2.2 23.4 24.1.2
canWrite	13.5.3
capacity	11.4.5
Card API	1.1
CardLayout	21.1.2 21.2
case	6.2.2.2
case-Label	6.2.2.2
catch	9.2.1 9.2.2 9.2.4 9.2.5 9.2.6 9.3.1 9.3.2 9.3.3 9.4 10.2.2 13.2.2.1 20.7.2 30.3.4 30.4.5
catch-or-throw-Regel	9.3.1
ceil	7.8.2.4
Center	21.1.3
CGI-Script	25.4.1
char	4.1.1 4.2 4.2.2 4.2.2.1 4.4.2 4.6 4.7 6.2.2.2 7.8.1 11.1 11.2.5 19.6 29.2.4.1 31.3.1
CHAR_UNDEFINED	19.6
Character	7.8.1 7.9 27.7.1
Character-Streams	13.1 29.2.4.1

CharArrayReader	13.3.2 13.6
CharArrayWriter	13.2.2 13.2.2.2 13.3.2.2 13.6
charAt	11.2.2 11.5 29.2.1.5
char-Literale	4.2.2.1
CharToByteConverter	13.2.2.1 29.2.4.1
charValue	7.8.1.2
charWidth	15.3.2
Checkbox	18.1.6.10 22.4 22.5 22.8 22.12
CheckboxGroup	22.5 22.12
CheckboxMenuItem	18.1.6.10 20.1 20.4.2 20.8
Choice	18.1.6.10 22.8 22.9 22.12
.class	2.2 7.1.1 7.7.1 8.1.6 8.2.2.1 8.2.4.1 8.3.1 8.3.2.1 13.4.4 26.1.1.2 26.1.2.2 26.2.2.2 26.3 28.2.1
Class	30.2.1 31.2.2 31.3.1
.class	31.3.2
Class	31.3.3
.class	31.4
Class	31.5
ClassCastException	20.7.2 27.2.1 31.2.2
classes.zip	8.2.2.3
Class.forName	30.3.3
ClassLoader	1.2.2 31.2.2
ClassNotFoundException	31.2.2
CLASSPATH	3.4.2.1 8.2.2.1 8.2.2.2 8.2.2.3 8.2.3 8.2.4.1 25.2.2 26.1.1.3 26.1.2.2 26.3
clear	12.4.1 27.6.1
clearRect	14.4.2 14.5
Client-Server-Beziehung	32.1.4
Clipboard	20.7.1 20.7.2 20.8
ClipboardOwner	20.7.2
Clipping-Region	14.4.3
clipRect	14.4.3 14.5
clone	4.5.1 7.2.3 7.7.6.3 28.3.2 31.2.1
Cloneable	7.7.6.3 7.9 28.3.2
close	13.2.1 13.2.2.1 13.3.1 13.4.1 30.2.1 32.2.2 32.3.1
Cluster	30.3.8
CODE	25.2.1 25.2.2
CODEBASE	25.2.2
Collection	27.1 27.2.1 27.3.1 27.3.2 27.5.1 27.6.1 27.7.2 27.9 28.3.1
Collection-API	1.2.4 12.1
Collection Framework	1.1
Collections	27 27.8 27.8.1 27.9
Color	2.3 16.2 16.3 16.4 16.5
Color.black	16.2

<code>Color.blue</code>	16.2
<code>Color.cyan</code>	16.2
<code>Color.darkGrey</code>	16.2
<code>Color.gray</code>	16.2
<code>Color.green</code>	16.2
<code>Color.lightGray</code>	16.2
<code>Color.magenta</code>	16.2
<code>Color.orange</code>	16.2
<code>Color.pink</code>	16.2
<code>Color.red</code>	16.2
<code>Color.white</code>	16.2
<code>Color.yellow</code>	16.2
Comboboxen	22.8
<code>commit</code>	30.4.3
<code>Comparable</code>	27.7.1 27.7.2 27.8.1 27.9
<code>Comparator</code>	12.9 27.7.1 27.7.2 27.7.3 27.8.1 27.9
<code>compare</code>	27.7.1 27.7.2
<code>compareTo</code>	11.2.4 11.5 27.7.1 27.7.2
Compiler	26.1.1
<code>comp.lang.java</code>	1.1 3.3.2.1
<code>Component</code>	15.4.3 17.1 17.2 17.5.3 17.5.4 17.5.5 17.6 18.1.1 18.1.4 18.1.6.1 18.1.6.2 18.1.6.3 18.1.6.4 18.1.6.5 18.2.4 19.2 19.3 19.4 19.5 19.6 21.1.2 22.1 22.12 23.1 23.2.2 23.2.3 24.1.1 24.1.2 24.2.4.1 25.1.2 25.1.4.1
<code>COMPONENT_EVENT_MASK</code>	18.2.4
<code>COMPONENT_SHOWN</code>	23.2.3
<code>ComponentAdapter</code>	18.1.5
<code>componentAdded</code>	18.1.6.6
Component-Ereignisse	18.1.6.5
<code>ComponentEvent</code>	18.1.2 18.1.6.5 19.1 19.2 19.5 19.6 19.7
Component-Events	19.2
<code>componentHidden</code>	18.1.6.5 19.2
<code>ComponentListener</code>	18.1.6.5 19.2 19.7
<code>componentMoved</code>	18.1.6.5 19.2
<code>componentRemoved</code>	18.1.6.6
<code>componentResized</code>	18.1.6.5 19.2
<code>componentShown</code>	18.1.6.5 19.2
<code>com.sun.image.codec.jpeg</code>	8.2.2
<code>Connection</code>	30.2.1 30.2.2 30.3.3 30.4.3 30.4.5 30.4.6 30.5
<code>Connection.TRANSACTION_NONE</code>	30.4.3
<code>Connection.TRANSACTION_READ_COMMITTED</code>	30.4.3
<code>Connection.TRANSACTION_READ_UNCOMMITTED</code>	30.4.3
<code>Connection.TRANSACTION_REPEATABLE_READ</code>	30.4.3
<code>Connection.TRANSACTION_SERIALIZABLE</code>	30.4.3

Constructor	31.3.3
Container	17.1 17.6 18.1.6.6 21.1.2 21.1.3 21.2.3 21.2.6 22.11 23.3 25.1.2
ContainerAdapter	18.1.5
Container-Ereignisse	18.1.6.6
ContainerEvent	18.1.6.6
ContainerListener	18.1.6.6
contains	12.3.2 27.1
containsAll	27.1
containsKey	12.3.2 27.6.1
containsValue	27.6.1
Content Handler Factories	32.4
continue	1.2.1 6.2.1.4 6.3.3.3 6.4 9.2.6
CoolEdit	25.3.1
copyArea	14.4.2 14.5
cos	7.8.2.1
Courier	15.2.1
Courier New	15.2.1
createCustomCursor	17.5.3
createImage	24.1.1 24.2.4.3
CREATE INDEX	30.4.7.1
createNewFile	13.5.5.2
createStatement	30.2.2 30.4.5
CREATE TABLE	30.4.7.1
createTempFile	13.5.5.1
currentTimeMillis	12.8.5 12.10
Cursor	17.5.3
Cursor.CROSSHAIR_CURSOR	17.5.3
Cursor.DEFAULT_CURSOR	17.5.3
Cursor.MOVE_CURSOR	17.5.3
Cursor.TEXT_CURSOR	17.5.3
Cursor.WAIT_CURSOR	17.5.3

- D -	
Dangling else	6.2.1.3
DatabaseMetaData	30.3.3 30.4.3 30.4.4
Data Definition Language	30.2.4
DataFlavor	20.7.2
DataFlavors	20.7.1
DatagramPacket	32.1.2
DatagramSocket	32.1.2
Date	12.7 12.7.4 12.10 13.5.3 30.2.3
Datei- und Verzeichnis-Handling	13.5
Datenbankzugriffe mit JDBC	30
Datenflußanalyse	4.2 5.1
Datentypen	Primitive
Datumswerte	12.7
Datums-/Zeitarithmetik	12.7.3
DAY_OF_WEEK	12.7.2
DayTime	32.2.2
Debugger	26.1.4
deep copy	28.3.2
default	6.2.2.2
default-Konstruktor	7.3.3.2
Default-Konstruktoren	7.2.6.1
Default-Label	6.2.2.2
Default-Paket	8.2.4.2
Definite Assignment	4.2 5.1
Deklaration von Variablen	4.3.2 6.1.3
Dekrement-Operator	4.3.1
Delegate	27.8.2
Delegation Based Event Handling	2.6 14.1 18.1.1
delete	Datei löschen Teilstring entfernen
deleteCharAt	11.4.3
DELETE FROM	30.2.4 30.4.7.2
deleteOnExit	13.5.5.2
deleteShortCut	20.4.3
delItem	22.9
deprecated	10.2.2 10.2.3 12.3.4 15.2.2 22.8 26.1.1.3
DESC	30.4.7.3
deselect	22.9
desktop	16.4
destroy	25.1.3.5 25.6
Destruktoren	7.2.7
Device-Kontext	14.2.2
Dialog	14.2.1 17.1 17.2 17.5.1 17.6 18.1.6.7 19.1 21.1.1 21.3 21.4 Modaler

Dictionary	12.3
Dimension	2.5 23.2.2
Disassembler	26.2.3
dispose	2.6 15.4.2 15.4.3 17.2 17.6 18.2.2.1
Division	5.2
Dämon	10.2.1
DNS	32.1.3.2
DNS-Konfiguration	26.1.4.3
do	1.2.1 6.3.2 6.3.2.2 6.3.3.3 6.4
Doclets	26.2.1.5
Dokumentation des JDK	3.4.2.2
Dokumentationsgenerator	26.2.1
Dokumentationskommentare	2.2 4.1.2 26.2.1.3
Domain Name System	32.1.3.2
Doppelpufferung	2.5 24.2.4.3
double	4.2 4.2.4 4.2.4.1
Double	4.2.4.1
double	4.6 4.7
Double	7.8.1
double	7.8.1
Double	7.8.1.3 7.8.1.4 7.9
double	12.6.2.1
Double	27.7.1
double	29.2.2
Double	29.3.2
double	29.3.2 30.2.3
Double	31.3.1
double	31.3.1 31.3.2
Double	31.3.2
double	31.3.2 31.4 31.5
Double.TYPE	31.3.2
doubleValue	7.8.1.2
Drag & Drop	2.6
Drag & Drop API	1.1 2.6
drawArc	14.3.5 14.5
drawBytes	15.1 15.5
drawChars	15.1 15.5
drawImage	2.5 24.1.1 24.2.4.3 24.3
drawLine	14.3.1 14.4.1 14.5
drawOval	14.3.4 14.3.5 14.5
drawPolygon	14.3.3 14.5
drawPolyline	14.3.3 14.5
drawRect	2.5 14.3.2 14.5
drawRoundRect	14.3.2
drawString	2.5 15.1 15.5

DriverManager	30.2.1 30.5
Drop-Down-Listboxen	22.8
DROP INDEX	30.4.7.1
DROP TABLE	30.4.7.1
Drucken	15.4
DST_OFFSET	12.7.2
Duff's Device	6.2.2.2
Dynamische Datenstrukturen	4.2 8.1.4
Dynamische Methodensuche	7.3.2.1
dynamisches Binden	7.3.2.1
Dynamisches Laden von Klassen	31.2.2

- E -	
East	21.1.3
ECHO-Service	32.2.3
Ein-/Ausgaben	3.5.3
Einerkomplement	5.5
Einfache Vererbung	7.3
Eingabe-Streams	13.3
Einschränkenden Konvertierungen	4.6
Einzeilige Kommentare	4.1.2
elementAt	12.1.2 29.3.1
elements	1.2.4 12.1.3 12.3.3
else	3.5.2.2 5.1 6.2.1.3
Embedded SQL	30.1.1
empty	12.2
EmptyStackException	12.2
enableEvents	18.2.4 20.6 23.2.3 23.4
end	15.4.2 15.4.3
endsWith	11.2.4 11.5
Entry	27.6.1
entrySet	27.6.1 27.6.2 27.7.3
Entwicklungszyklus	8.3
enumerate	10.5.2
Enumeration	1.2.4 7.7.6.2 7.9 12.1.3 12.3.3 12.3.4 12.5 12.5.2 12.10 27.1 27.3.1 27.6.2 28.3.1
equals	4.1.2 4.5.1 5.7.2.2 11.2.4 11.5 12.3 12.7.3 12.9 27.1 27.5.1 27.6.1 31.2.1
equalsIgnoreCase	11.2.4
ERA	12.7.2
Ereignisempfänger	18.1.1 18.1.3
Ereignisquellen	18.1.1 18.1.4
Ereignistypen	18.1.2
Ergebnistyp eines Ausdrucks	5.2
err	12.8.2
erreichbare Anweisungen	6.2.1.4
Error	9.2.3 9.4
Ersetzen von Zeichenketten	11.2.6
Erstellen eigener Pakete	8.2.4
Erweiternde Konvertierungen	4.6
Escape-Kommandos	30.4.2
Escape-Sequenzen	4.2.2.1
ESQL	30.1.1
Ethernet	32.1.2
Event	18.1.2 18.3

EventListener	18.1.1 18.1.3 18.1.4 18.2.1 18.2.2.1 18.2.4 18.3
EventObject	18.1.2
Event Sources	18.1.1
Exception	2.4 9.2.3 9.3.2.1 9.4 30.2.5 30.3.2
Exceptionhandling	1.2.1
Exceptions	9
executeQuery	30.2.2 30.2.3 30.3.5 30.3.6 30.4.6
executeUpdate	30.2.2 30.2.4 30.3.4 30.4.6
exists	13.5.3
EXISTS	30.4.7.3
EXKLUSIV-ODER-Operator	5.4
exp	7.8.2.3
Exponentialfunktion	7.8.2.3
extends	7.3.1 7.9 31.2.1
Extension Framework	1.1
Externalizable	28.2.1

- F -	
false	4.1.3 4.2 4.2.1.1 4.7 6.2.1.2 6.2.1.4 6.3.1.2 6.3.2.2 10.2.2 11.2.4 12.3.2 12.5.1 12.5.2 13.5.4 13.5.5.2 20.4.2 21.3 22.4 22.5 22.6 27.2.1 27.5.1 27.5.2 27.6.1 30.2.3 30.4.3 32.2.3
FAQs	3.3.2.3
FDDI	32.1.2
Fehlerobjekt	9.2.2
Fensterklassen	17.1
Fenstertitel	17.5.1
Field	31.4
File	13.2.2.1 13.3.2.1 13.4.1 13.5 13.5.1 13.5.2 13.5.3 13.5.4 13.5.5 13.5.5.1 13.5.5.2 13.6 30.3.4
FileDescriptor	13.2.2.1 13.3.2.1
FileDialog	17.1 17.2
FileInputStream	28.1.3 29.2.4.2 32.3.3
FilenameFilter	13.5.4
FileNotFoundException	13.3.2.1 13.4.1
FileOutputStream	28.1.2 29.2.4.1
FileReader	13.3.2 13.3.2.1 13.6 29.2.4.2
file.separator	12.8.1
FileWriter	13.2.2 13.2.2.1 13.2.2.2 13.2.3.2 13.2.3.3 13.6 29.2.4.1
fill	12.9 21.2.4
fillArc	14.4.1 14.5
fillOval	14.4.1 14.5
fillPolygon	14.4.1 14.5 23.2.2
fillRect	2.5 14.4.1 14.5
fillRoundRect	14.4.1 14.5
FilterReader	13.3.2 13.3.3 13.3.3.3 13.6
FilterWriter	13.2.2 13.2.3 13.2.3.3 13.6
final	7.3.1 7.3.2.1 7.4.2.5 7.5.2 7.7.5 7.9 11.2.2 11.3.1 29.2.1.5 29.2.2
finalize	7.2.7 7.9
finally	9.2.6 9.3.3 9.4 30.4.5
Firewall	32.1.6
first	27.7.2
firstElement	12.1.2
First Person, Inc.	1.1
Fließkommazahlen	4.2.4
Füllmodus	14.4.1
float	4.2 4.2.4 4.2.4.1
Float	4.2.4.1
float	4.6 4.7 5.2
Float	7.8.1
float	7.8.1
Float	7.8.1.3 7.8.1.4 7.9

float	12.6.2.1 30.2.3 31.3.1
floatValue	7.8.1.2
floor	7.8.2.4
FlowLayout	21.1.2 21.2 21.2.1 21.2.2 21.2.2.1 21.2.3 21.4 22.6 23.2.2
FlowLayout.CENTER	21.2.1
FlowLayout.LEFT	21.2.1
FlowLayout.RIGHT	21.2.1
flush	13.2.1 13.2.3.1 13.2.3.2 29.3.1
flushBuffer	29.3.1
FOCUS_GAINED	23.2.3
FOCUS_LOST	23.2.3
FocusAdapter	18.1.5
Focus-Ereignisse	18.1.6.1
FocusEvent	18.1.6.1 19.5 19.7
Focus-Events	19.5
focusGained	18.1.6.1 19.5
FocusListener	18.1.6.1 19.5 19.7
focusLost	18.1.6.1 19.5
Font	15.2.1 15.3.1 15.5
Font.BOLD	15.2.1
Font-Informationen	15.3.1
Font.ITALIC	15.2.1
FontMetrics	15.3.2 15.5
Font-Metriken	15.3.2
Font.PLAIN	15.2.1
font.properties	15.2.1
for	1.2.1 4.4.3 5.7.3 6.3.3.2 6.3.3.3 6.4 9.2.5 24.2.4.2 29.3.2
forName	30.2.1 31.2.2 31.3.2
for-Schleife	6.3.3
Fragezeichenoperator	5.7.1.1
Frame	14.2.1 14.3 15.3.1 17.1 17.2 17.3 17.4 17.5.1 17.5.2 17.6 18.1.6.7 18.2.1 18.2.4 19.1 19.3 19.5 20.2 21.1.1 21.3 21.4 22.11 24.2.4.3
Frequently Asked Questions	3.3.2.3
Funktionstasten	19.6
Funktionszeiger	mit Interfaces mit Reflection

- G -	
Gamelan	1.1 3.3.2.2
Garbage Collector	4.5.2 12.8.4
gc	12.8.4 12.10
Geschachtelte Schleifen	6.3.3.3
get	12.3.2 12.4.1 12.7.2 18.1.2 27.6.1 27.6.2 29.2.3 31.4
GET	32.2.4 32.3.3
getAbsolutePath	13.5.2
getActionCommand	20.5 22.3 22.6
getAddress	32.2.1
getAdjustable	22.10
getAdjustmentType	22.10
getApplet	25.4.2
getAppletContext	25.4.2
getAppletInfo	25.1.4.4 25.6
getApplets	25.4.2
getAscent	15.3.2
getAudioClip	25.3.1
getAvailableFontFamilyNames	15.2.2
getBestCursorSize	17.5.3
getBlockIncrement	22.10
getBlue	16.2
getBoolean	30.2.3
getBounds	14.4.3 14.5
getBuffer	13.2.2.2
getByName	32.2.1
getByte	30.2.3
getBytes	30.2.3
getCaretPosition	22.6
getChars	29.2.1.5
getCheckboxGroup	22.5
getClass	10.4.2.1 31.2.2 31.4
getClickCount	19.3
getClip	14.4.3 14.5
getClipBounds	14.4.3
getCodeBase	25.3.1 25.4.3 25.6
getColor	16.3 16.5
getColumnCount	30.4.1
getColumnName	30.4.1
getColumns	22.6
getColumnType	30.4.1
getComponent	19.2
getConnection	30.2.1

getConstructor	31.3.3
getConstructors	31.3.3
getContent	32.4
getContents	20.7.2
getCurrent	22.5
getDate	30.2.3
getDeclaredField	31.4
getDeclaredFields	31.4
getDeclaredMethods	31.3.1
getDefaultToolkit	15.2.2
getDescent	15.3.2
getDocumentBase	2.4 25.3.1 25.4.3 25.6
getDouble	30.2.3 31.4
getEchoCharacter	22.7
getErrorCode	30.2.5
getFamily	15.3.1
getField	31.4
getFields	31.4
getFilePointer	13.4.2
getFloat	30.2.3
getFont	15.2.1 15.3.1
getFontList	15.2.2
getFontMetrics	15.3.2
getGraphics	2.6 15.4.1 15.4.2 19.3 24.2.4.3
getGreen	16.2
getHAdjustable	22.11
getHeight	15.3.2 24.1.2
getHostAddress	32.2.1
getHostName	32.2.1
getID	18.1.2 19.1 19.2 19.3 19.5 19.6 20.5 22.6
getImage	2.4 24.1.1 24.3
getInputStream	32.2.2 32.4
getInsets	2.5 14.2.3 15.1 15.5
getInt	30.2.3 30.3.5 31.4
getItem	20.3 22.8 22.9
getItemCount	20.3 22.8
getItemSelectable	22.4 22.8 22.9
getKey	27.6.1 27.6.2
getKeyChar	18.1.2 19.6 23.2.3
getKeyCode	18.2.1 19.6
getLabel	20.4.1 22.3 22.4
getLeading	15.3.2
getLineNumber	13.3.3.2
getLocalGraphicsEnvironment	15.2.2
getLocalHost	32.2.1

getLocation	20.5 21.3
getLong	30.2.3
getMaximum	22.10
getMaximumCursorColors	17.5.3
getMaximumSize	23.2.2
getMenu	20.2
getMessage	9.2.2
getMetaData	30.3.3 30.4.1
getMethod	31.3.2 31.3.3 31.4
getMethods	31.3.1 31.3.2 31.3.3
getMinimum	22.10
getMinimumSize	23.2.2 23.2.3 23.4 24.1.2
getModifiers	18.1.2 31.3.1 31.3.3
getName	10.5.1 13.5.2 20.7.2 31.3.1 31.3.3 31.4
getNextException	30.2.5
getOutputStream	32.2.2
getPageDimension	15.4.2
getPageResolution	15.4.2
getParameter	2.4 25.2.3 25.6
getParameterInfo	25.1.4.3 25.1.4.4 25.6
getParameterTypes	31.3.1 31.3.3
getParent	10.5.2 13.5.2
getPath	13.5.2
getPoint	18.1.2 19.3
getPosition	19.3
getPreferredSize	22.11 23.2.2 23.2.3 23.4 24.1.2
getPrintJob	15.4.1 15.5
getPriority	10.5.1
getProperties	12.8.1
getProperty	12.3.4 12.8.1
getRed	16.2
getScreenResolution	15.4.2
getScreenSize	17.3
getScrollPosition	22.11
getSelectedIndex	22.8
getSelectedIndexes	22.9
getSelectedItem	22.8
getSelectedItems	22.9
getSelectedText	22.6
getSelectionEnd	22.6
getSelectionStart	22.6
getShort	30.2.3
getSize	2.5 15.1 15.3.1 15.5 20.5 22.11
getSource	18.1.2 19.1 19.2 19.3 19.5 19.6 20.5 22.6
getSQLState	30.2.5

<code>getState</code>	17.4 20.4.2 22.4
<code>getString</code>	30.2.3
<code>getStyle</code>	15.3.1
<code>getSuperClass</code>	31.4
<code>getSystemClipboard</code>	20.7.2
<code>getTableName</code>	30.4.1
<code>getText</code>	22.2 22.6
<code>getThreadGroup</code>	10.5.2
<code>getTime</code>	12.7.4 30.2.3
<code>getTimestamp</code>	30.2.3
<code>getTitle</code>	17.5.1 17.6
<code>getToolkit</code>	24.1.1
<code>getTransactionIsolation</code>	30.4.3
<code>getTransferData</code>	20.7.2
<code>getTransferDataFlavors</code>	20.7.2
<code>getType</code>	31.4
<code>getTypeInfo</code>	30.4.4
<code>getUnitIncrement</code>	22.10
<code>getVAdjustable</code>	22.11
<code>getValue</code>	22.10 27.6.1 27.6.2
<code>getViewportSize</code>	22.11
<code>getVisibleAmount</code>	22.10
<code>getWhen</code>	19.3
<code>getWidth</code>	24.1.2
<code>getWindow</code>	19.1
<code>getX</code>	19.3
<code>getY</code>	19.3
<code>gif</code>	2.1 2.7 24.1.1
<code>gk.util</code>	29.3.2 30.3.2 installieren
Gleichheits-Operator	5.3
Globale Funktionen	7.2.1
GoldWave	25.3.1
Gosling, James	1.1
Grafikkontext	2.5 14.2.2
Grafische Oberfläche	3.2
Graphics	2.5 14.2.2 14.3 14.3.4 14.4.2 14.4.3 14.5 15.1 15.2.1 15.3.1 15.4.1 15.4.3 16.3 19.3 23.1 24.1.1 24.2.3
GraphicsEnvironment	15.2.2
Green-Projekt	1.1
GregorianCalendar	12.7 12.7.1 12.7.2 12.7.3 12.10 13.5.3
Größergleich-Operator	5.3
Größer-Operator	5.3
GridBagConstraints	21.2.4
<code>GridBagConstraints.BOTH</code>	21.2.4
<code>GridBagConstraints.CENTER</code>	21.2.4
<code>GridBagConstraints.EAST</code>	21.2.4

GridBagConstraints.HORIZONTAL	21.2.4
GridBagConstraints.NONE	21.2.4
GridBagConstraints.NORTH	21.2.4
GridBagConstraints.NORTHEAST	21.2.4
GridBagConstraints.NORTHWEST	21.2.4
GridBagConstraints.SOUTH	21.2.4
GridBagConstraints.SOUTHEAST	21.2.4
GridBagConstraints.SOUTHWEST	21.2.4
GridBagConstraints.VERTICAL	21.2.4
GridBagConstraints.WEST	21.2.4
GridBagLayout	21.1.2 21.2 21.2.4
gridheight	21.2.4
GridLayout	21.1.2 21.2 21.2.2 21.2.2.1 21.2.3 21.2.4 21.2.6 21.4 22.6 23.2.2
gridwidth	21.2.4
gridx	21.2.4
gridy	21.2.4
GROUP BY	30.4.7.3
Grundlinie	15.3.2

- H -

handleEvent	18.1.1
Hardware-Voraussetzungen	3.4.1
hashCode	27.1 27.6.1 31.2.1
Hash-Funktion	12.3 27.6.2
HashMap	27.5.2 27.6.2
HashSet	27.5.2
Hashtabelle	27.6.2
Hashtable	1.2.4 7.7.6.2 12.1.3 12.3 12.3.1 12.3.2 12.3.3 12.3.4 12.5 12.8.1 12.10 27.1 27.6.2 28.3.1 31.2.1
hasMoreElements	1.2.4 7.7.6.2 12.1.3 12.3.3 12.5.2 27.3.1 29.3.2
hasMoreTokens	12.5.2
hasNext	27.3.1 27.4.2
hasPrevious	27.3.2
HAVING	30.4.7.3
headSet	27.7.2
HeapSort	29.3.2
HEIGHT	25.2.1
Helvetica	15.2.1
Höhe	einer Zeile
Himmelsrichtungen	21.2.3
Hintergrundfarbe	17.5
Hintergrund-Thread	10.2.1
Host-ID	32.1.3.1
Host-Namen	25.4.1
hosts	26.1.4.3
Host-Variablen	30.1.1
HotJava	1.1
HotJava Views	1.1
HSPACE	25.2.2
HTML-Adresse	25.4.1
HTML-Datei	2.3
HTML-Dokument	25.2.1
http	25.4.1

- -	
IAB	32.1.5
IBM	1.1
ICMP	32.1.2
Icon	17.5
ICONIFIED	17.4
if	1.2.1 3.5.2.2 6.2 6.2.1.2 6.2.1.3 6.4
if-Anweisung	6.2.1
if-else	6.2 6.2.1.2 6.2.1.3
IllegalArgumentException	27.2.1
Image	24.1.1 24.1.1.1 24.2.2 24.3
ImageObserver	24.1.1
immutable	7.8.1
immutable objects	27.5.1
Implementierung	eines Interfaces
implements	7.7.2
import	2.2 8.2.1 8.2.1.1 8.2.4.1 8.2.4.2 8.4
Import von java.lang	8.2.1.1
in	12.8.2
IN	30.4.7.3
.inc	14.3
indexOf	11.2.5 11.5
IndexOutOfBoundsException	9.2.5
InetAddress	32.2.1 32.2.2 32.5
inetd.conf	32.3.1
init	2.3 2.4 2.6 25.1.1 25.1.3.2 25.2.3 25.5 25.6
Initialisieren eines Arrays	4.4.1
Initialisieren von Variablen	4.3.2
Inkrement-Operator	4.3.1
Inner Classes	18.2.2.1
InputEvent	18.1.2 19.3 19.6
InputStream	12.3.4 13.1 20.7.2 28.1.3 28.3.2 29.2.4.3 32.2.2 32.2.3 32.4
InputStreamReader	3.5.3.2 13.3.2 13.3.2.1 13.6
insert	11.4.2 11.5 22.7
insertElementAt	12.1.1
INSERT INTO	30.2.4 30.3.4 30.4.7.2
insertSeparator	20.3
insets	21.2.4
Installation	3.4.2
instanceof	5.7.2.3 5.9 12.1.1 22.6
InstantDB 1.85	30.3.1
Instanzvariablen	4.3.1 4.3.3

int	4.2 4.2.3 4.2.3.1 4.4.1 4.4.2 4.6 4.7 5.2 5.6 5.7.2.7 6.2.2.2 7.8.1 9.2.1 10.5.1 12.6.2.1 13.2.1 13.3.1 13.4.4 15.2.1 16.2 22.8 22.9 28.1.2 28.1.3 29.2.2 30.2.3 30.3.5 31.3.1 31.4
Integer	7.8.1 7.8.1.3 7.8.1.4 7.9 8.3.2.1 22.9 27.5.1 31.3.1
Integer.TYPE	31.3.1
Integrale Typen	4.2.3
interface	7.7.1
Interfaces	7.3 7.7
Internet	3.3.2
Internet Activity Board	32.1.5
Interpreter	26.1.2
interrupt	10.2.2 10.6
interrupted	10.2.2 10.6
InterruptedException	2.4 8.3.2.1 10.2.2 10.2.4.1 24.1.1.1
InterruptedException	32.2.3
Introspection	31.1
intValue	7.8.1.2 22.9
InvalidClassException	28.2.1
invoke	31.3.1 31.3.2 31.3.3
IOException	12.3.4 13.2.2.1 13.4.1 32.2.2
IP-Adresse	32.1.3.1
ipadx	21.2.4
ipady	21.2.4
IP-Protokoll	32.1.2
IPv6	32.1.3.1
isAbsolute	13.5.3
isAbstract	31.3.1
isAlive	10.2.4.2 10.6
isAltDown	19.6
isControlDown	19.6
isDataFlavorSupported	20.7.2
isDirectory	13.5.3
isEditable	22.6
isEmpty	12.1.1 27.1 27.6.1
isEnabled	20.4.1
isExplicit	31.3.1
isFile	13.5.3
isFinal	31.3.1
isFocusTraversable	23.2.3
isHidden	13.5.3
isInterface	31.3.1
isInterrupted	10.2.2 10.6
isMetaDown	19.3 19.6
isModal	21.3
isNative	31.3.1
IS NOT NULL	30.4.7.3

IS NULL	30.4.7.3
ISO-8859-1	4.1.1
ISO/OSI-7-Schichten-Modell	32.1.2
isPopupTrigger	19.3 20.6
isPrivate	31.3.1
isProtected	31.3.1
isPublic	31.3.1
isResizable	21.3
isShiftDown	19.6
isStatic	31.3.1
isStrict	31.3.1
isSynchronized	31.3.1
isTemporary	19.5
isTransient	31.3.1
isVolatile	31.3.1
Item-Ereignisse	18.1.6.10 22.8
ItemEvent	18.1.2 18.1.6.10 22.4 22.8 22.9
ItemListener	18.1.6.10 22.4 22.8
itemStateChanged	18.1.6.10 22.4 22.8 22.9
Iterator	1.2.4 27.3.1
iterator	27.3.1
Iterator	27.3.2
iterator	27.4.1 27.4.2
Iterator	27.4.2
iterator	27.4.2
Iterator	27.5.1
iterator	27.6.1
Iterator	27.9

- J -	
J++	1.1 29.3.1
jar	3.4.2.3 25.2.2 26.1 26.2.2.2 26.2.2.3 26.2.2.4 26.3
JAR-Archiv	25.2.2
JARS	3.3.2.2
. java	2.2 3.5.2.1 8.2.4.1 8.3.2.1 26.1 26.1.1.2 26.1.2.2 26.1.2.3 26.2.2.3 26.3
Java 2D API	1.1 1.2.3
Java 2 platform	Vor 1.1
JAVA_COMPILER	26.1.2.2 29.3.1
java_g	26.1.2.3 29.3.1
java.applet	8.2.1.2 25.1.2
java.awt	8.2.1.2 14.2.1 14.5 18.1.2
java.awt.color	8.2.1.2
java.awt.datatransfer	8.2.1.2 20.7.1 20.8
java.awt.dnd	8.2.1.2
java.awt.event	18.1.2 18.1.5
java.awt.font	8.2.1.2
java.awt.geom	8.2.1.2
java.awt.im	8.2.1.2
java.awt.image	1.2.3 8.2.1.2 8.2.2
java.awt.image.renderable	8.2.1.2
java.awt.print	8.2.1.2 15.4.3
JavaBeans	1.1 1.2.4
java.beans	8.2.1.2
java.beans.beancontext	8.2.1.2
javac	2.2 2.7 3.5.1.3 3.5.2.1 26.1 26.1.1.2 26.3
java.class.path	12.8.1
java.class.version	12.8.1
java.compiler	26.1.2.2
Java Database Connectivity	1.2.4 30.1.1
Java Developers Connection	3.3.2.2
Java Developer's Journal	3.3.2.4
javadoc	2.2 2.7 4.1.2 26.1 26.2.1.2 26.2.1.3 26.2.1.4 26.2.1.5 26.3
Java Foundation Classes	1.1 1.2.3
javah	26.1
java.home	12.8.1
java.io	8.2.1.2 13.1 13.2.1 13.2.2.1 13.4 13.5 28.1.2
java.lang	4.2.4.1 8.2.1.1 8.2.1.2 10.2.1 12.8 12.8.1 27.7.1 31.3.1
java.lang.Boolean.TYPE	31.3.1
java.lang.Byte.TYPE	31.3.1
java.lang.Character.TYPE	31.3.1
java.lang.Double.TYPE	31.3.1

java.lang.Float.TYPE	31.3.1
java.lang.Integer.TYPE	31.3.1
java.lang.Long.TYPE	31.3.1
java.lang.Math	7.8.2 7.8.2.1
java.lang.ref	8.2.1.2
java.lang.reflect	8.2.1.2 31.3.1
java.lang.Short.TYPE	31.3.1
java.lang.Void.TYPE	31.3.1
JavaLobby	3.3.2.2
java.math	8.2.1.2
java.net	8.2.1.2 25.4.1 32.2.1
JavaOne	1.1
JavaOS 1.0	1.1
javap	26.1 26.2.3.2 26.3
java.prof	29.3.1 29.3.2
Java Report Online	3.3.2.4
java.rmi	8.2.1.2
java.rmi.dgc	8.2.1.2
java.rmi.registry	8.2.1.2
java.rmi.server	8.2.1.2
Java Runtime Environment	26.1.2.3
JavaScript	1.2.2
java.security	8.2.1.2
java.security.acl	8.2.1.2
java.security.cert	8.2.1.2
java.security.interfaces	8.2.1.2
java.security.spec	8.2.1.2
JavaSoft	1.1 3.3.2.2
java.specification.name	12.8.1
java.specification.vendor	12.8.1
java.specification.version	12.8.1
java.sql	8.2.1.2 30.2.1
java.sql.Date	12.7.4
java.sql.Types	30.4.1 30.4.4
JavaStation	1.1
java.text	8.2.1.2
Java-Usergruppen	1.1
java.util	1.2.4 8.2.1.2 12.1 12.8 12.8.1 12.9 18.1.2 18.1.3 27.1 27.8
java.util.Date	12.7.4
java.util.EventObject	18.1.2
java.util.jar	8.2.1.2
java.util.mime	8.2.1.2
java.util.zip	8.2.1.2
java.vendor	12.8.1
java.vendor.url	12.8.1

java.version	12.8.1
java.vm.name	12.8.1
java.vm.specification.name	12.8.1
java.vm.specification.vendor	12.8.1
java.vm.specification.version	12.8.1
java.vm.vendor	12.8.1
java.vm.version	12.8.1
javaw	26.1.2.2
Java World	3.3.2.4
javax.	8.2.1.2
jdb	26.1 26.1.4.2 26.1.4.3 26.3
JDBC	1.1 1.2.4 30 30.1.1
JDBC-ODBC-Bridge	30.1.2.1
JDC	3.3.2.2
JDC-Newsletter	Tech Tips
JDK 1.0	1.1
JFC	1.2.3
JInsight	29.3.3
JIT	26.1.2.2 29.1
join	10.2.2 10.2.4.3 10.6
Joy, Bill	1.1
jpeg	2.1 2.7 24.1.1
JProbe	29.3.3
jre	26.1.2.3
Just-In-Time-Compiler	26.1.2.2 29.1 29.3.1

- K -

KEY_PRESSED	23.2.3
KeyAdapter	18.1.5 18.2.2.1 18.2.2.2
Key-Ereignisse	18.1.6.2
KeyEvent	18.1.2 18.1.6.2 18.2.1 18.2.3 19.3 19.6 19.7
KeyEvent.CHAR_UNDEFINED	19.6
Key-Events	19.6
KeyEvent.VK_UNDEFINED	19.6
KeyListener	18.1.6.2 18.2.1 18.2.2.1 19.6 19.7
keyPressed	18.1.6.2 18.2.1 18.2.2.1 18.2.2.2 19.6
keyReleased	18.1.6.2 18.2.1 19.6
keys	12.3.3
keySet	27.6.1 27.6.2 27.7.3
keytool	26.1
keyTyped	18.1.6.2 18.2.1 19.6
Klammerung	5.1
Klassen	8.1.4
Klassenbibliothek	3.2
Klassenmethoden	2.4 7.5.3 8.1.3
Klassenobjekt	31.2.2
Klassenvariablen	4.3.1 4.3.3 7.5.1
Kleiner-Operator	5.3
Kleiner-Operator	5.3
Kommentare	4.1.2
Konstanten	7.5.2 in Interfaces
Konstruktoren	7.2.6
Konstruktorenverkettung	7.2.6.2 7.3.3.1
Kontextmenüs	20.6
Kontext-URL	25.4.3
Konvertierungen auf primitiven Datentypen	4.6
Konvertierungsfunktionen	11.2.7
Koordinatensystem	14.2.3
Kopieren von Flächen	14.4.2
Kopieren von Objekten	28.3.2
Kreis	14.3.4
Kreisbogen	14.3.5
Kritischer Bereich	10.4.2

- L -	
Label	22.2 22.12
Label.CENTER	22.2
Label.LEFT	22.2
Label.RIGHT	22.2
Ladefaktor	29.2.1.1
last	27.7.2
lastElement	12.1.2
lastIndexOf	11.2.5 11.5
lastModified	13.5.3
Laufzeitfehler	9.1
LayoutManager	21.2
Lebensdauer	4.3.3 7.4
Leere Anweisung	6.1.1
left	25.2.2
length	4.4.2 4.4.3 11.4.5 11.5 12.8.6 29.2.1.5 Array String StringBuffer
Lexikalische Elemente	4.1
LIFO-Prinzip	12.2
LIKE	30.4.7.3
Lineare Liste	12.1
LineNumberReader	13.3.2 13.3.3 13.3.3.2 13.6
line.separator	12.8.1 13.2.3.1
Linie	14.3.1
Linien- oder Füllmodus	14.4.1
LinkedList	27.1 27.2.2 27.4.1 27.8.1 27.9 29.2.3
list	13.5.4
List	18.1.6.8 18.1.6.10 20.5 22.8 22.9 22.12 27.1 27.2.1 27.2.2 27.3.2 27.5.1 27.7.1 27.9 Collection Dialogelement
Listboxen	22.9
Listenstrukturen mit Zeigern	27.4.1
listFiles	30.3.4
ListIterator	27.3.2 27.5.1 27.9
listRoots	13.5.4
Literale	für Fließkommazahlen für integrale Typen für logische Typen für Zeichentypen
Länge der Zeichenkette	11.2.3
load	12.3.4
localhost	32.2.1
Lockdateien	13.5.5.2
log	7.8.2.3
Logarithmus	7.8.2.3
Logische Operatoren	5.4
Logischer Typ	4.2.1

logische Verknüpfungen	5.5
Lokale Klassen	18.2.2.1
Lokale Variable	4.3.1 6.1.3.2
long	4.2 4.2.3 4.2.3.1 4.6 4.7
Long	7.8.1
long	7.8.1
Long	7.8.1.3 7.8.1.4 7.9
long	10.2.4.3 12.6.2.1 12.7 12.7.4 13.4.2 28.2.1 30.2.3 31.3.1
longValue	7.8.1.2
loop	25.3.1
lostOwnership	20.7.2
Lotus	1.1
lowerCase	13.2.3.3
Löschen von Flächen	14.4.2

- M -	
main	3.5.1.2 8.1.6 8.1.7 10.2.1 10.2.2 10.2.4.1 10.3.1 14.3 18.2.3 25.1.1 26.1.2.2 30.3.2
make	8.3.2.2
makefile	8.3.2.2
MalformedURLException	25.4.1
Map	27.1 27.6.1 27.6.2 27.7.1 27.7.3 27.9
Map.Entry	27.6.1 27.6.2
mark	13.3.1
markSupported	13.3.1
Math	2.4 7.5.3 7.9 12.6.2.1
Mauscursor	17.5 17.5.3
Mausereignisse	2.6
max	7.8.2.2
MAX_VALUE	4.2.4.1 4.7 7.8.1.4
McNealy, Scott	1.1
MediaTracker	2.4 24.1.1.1 24.2.2 24.3
Mehrdimensionale Arrays	4.4.3
Mehrfachselektion	22.9
Mehrfachvererbung	7.3 7.7
Mehrfachverzweigung	6.2.2.2
Mehrstufige Client-Server-Architekturen	30.1.2.2
Mehrzeilige Kommentare	4.1.2
Membervariablen	7.1.1
Member-Zugriff	5.7.2.5
Menü	17.5
Menüeinträge	20.4
Menüleiste	20.2
Menu	20.1 20.3 20.4.4 20.6 20.8
MenuBar	20.1 20.2 20.3 20.4.2 20.8
MenuItem	18.1.6.8 20.1 20.3 20.4.1 20.4.2 20.4.3 20.4.4 20.5 20.8
MenuShortcut	20.4.3
Meta-Ressourcen	3.3.2.2
Method	31.3.1 31.3.3
Methoden	7.2 8.1.3
Methodenaufruf	5.7.2.6
Methodenüberlagerung	7.3.2
Metriken für Fonts	15.3.2
Microsoft Internet Explorer	1.2.2
middle	25.2.2
Middleware	30.1.2.1
Midi-Dateien	1.2.3
Midi-Format	25.3.1

MIME-Spezifikation	20.7.1
MIME-Typ	32.3.3
min	7.8.2.2
MIN_VALUE	4.2.4.1 4.7 7.8.1.4
Minimum und Maximum	7.8.2.2
mkdir	13.5.4
makedirs	13.5.4
Modale Dialoge	21.3
Modifier	7.2.1 7.4.1 31.3.1
Modulo-Operator	5.2
Monitor	10.1
Monospaced	15.2.1 15.5
MOUSE_PRESSED	23.2.3
MouseAdapter	18.1.5
mouseClicked	18.1.3 18.1.4 18.1.6.3 19.3
mouseDragged	2.6 18.1.6.4 19.4
mouseEntered	18.1.3 18.1.4 18.1.6.3 19.3
Mouse-Ereignisse	18.1.6.3
MouseEvent	18.1.2 18.1.6.3 18.1.6.4 19.3 19.4 19.7
Mouse-Events	19.3
mouseExited	18.1.3 18.1.4 18.1.6.3 19.3
MouseListener	18.1.3 18.1.4 18.1.6.3 19.3 19.7
MouseMotionAdapter	18.1.5
MouseMotion-Ereignisse	18.1.6.4
MouseMotion-Events	19.4
MouseMotionListener	18.1.6.4 19.4 19.7
mouseMoved	18.1.6.4 19.4
mousePressed	2.6 18.1.3 18.1.4 18.1.6.3 19.3 20.6
mouseReleased	2.6 18.1.3 18.1.4 18.1.6.3 19.3 20.6
MS-Access 7.0	30.3.1
Multiplikation	5.2
Multi-Tier-Architekturen	30.1.2.2
mutable objects	27.5.1
MVC-Konzept	18.2.3

- N -	
NAME	25.2.2
Name-Server	32.1.3.2
NaN	4.2.4.1 4.7 7.8.1.4
Natürliche Ordnung	27.7.1
Naughton, Patrick	1.1
Navigator	1.1
NCSA Mosaic	1.1
Nebeneffekte	5.1 6.1.4.2
Nebenläufigkeit	10.1
NEGATIVE_INFINITY	4.2.4.1 4.7 7.8.1.4
Netscape	1.1 Navigator
Network Information Center	32.1.3.1
Networking-API	1.2.4
Netzwerk-ID	32.1.3.1
Netzwerkprogrammierung	32
Netzwerkschicht	32.1.2
new	1.2.1 2.3 4.4.1 4.5.1 4.5.2 5.9 7.1.2 7.2.6 7.3.3.1 28.1.3
newAudioClip	25.3.2
newInstance	31.2.2 31.3.3
newLine	13.2.3.1
new-Operator	5.7.2.4 7.1.2
Newsgroups	1.3.2
next	27.3.1 27.3.2 27.4.2 30.2.3 30.3.6
nextDouble	12.6.2.1
nextElement	1.2.4 7.7.6.2 12.1.3 12.3.3 12.5.2 27.3.1 29.3.2
nextFloat	12.6.2.1
nextGaussian	12.6.2.2
nextInt	27.3.2
nextInt	12.6.2.1
nextLong	12.6.2.1
nextToken	12.5.2
NIC	32.1.3.1
NICHT-Operator	5.4
NORMAL	17.4
North	21.1.3
NoSuchElementException	27.3.1 27.4.1
NoSuchMethodException	31.3.2
NOT	30.4.7.3
notify	10.4.3 10.6
NotSerializableException	28.1.2 28.2.4
null	4.1.3 4.5.1 7.1.2 10.5.2 12.3.1 12.3.2 12.3.4 12.8.1 13.3.3.1 13.5.2 15.4.1 20.4.3 21.2.5 22.4 25.2.3 27.4.2 27.6.1 27.6.2 28.2.3 29.1

NULL	30.2.3
null	30.2.3
NULL	30.2.3
null	30.2.5 31.3.1 31.3.2 31.4
Null-Layout	21.2 21.2.5
NullPointerException	31.3.2
NumberFormatException	9.2.1 9.2.5

- O -

Oak	1.1
Oberlänge	15.3.2
Object	4.1.2 7.2.3 7.3.3.3 8.2.1.1 10.2.2 10.4.3 11.2.4 12.1 12.1.1 12.5.2 12.9 20.7.2
OBJECT	25.2.2
Object	27.1 27.2.1 27.3.1 27.6.1 28.1.3 28.3.1 31.2.1 31.2.2 31.3.1 31.3.2 31.3.3 31.4 31.5
ObjectInputStream	28.1.3 28.2 28.4
ObjectOutputStream	28.1.2 28.1.3 28.2 28.2.3 28.4
ODBC-Treiber	30.1.2.1
ODER-Operator	5.4
Offscreen-Image	2.5 24.2.4.3
Online-Magazine und Dokumentationen	3.3.2.4
OOP-Sprache	1.2.1
openConnection	32.4
openStream	32.4
Operatoren	Arithmetische Bitweise für Objekte Logische Relationale Sonstige Zuweisungs-
Operator-Vorrangregeln	5.8
OptimizeIt	29.3.3
or	12.4.2
OR	30.4.7.3
Oracle	1.1
ORDER BY	30.4.7.3
os.arch	12.8.1
os.name	12.8.1
os.version	12.8.1
out	12.8.2
OutputStream	12.3.4 13.1 28.1.2 28.3.2 29.2.4.1 29.2.4.3
OutputStreamWriter	13.2.2 13.2.2.1 13.6

- P -	
pack	21.1.4 21.2.2.1 23.2.2
package	8.2.3 8.2.4.1 8.2.4.2
package-list	26.2.1.4
Pageable	15.4.3
paint	2.5 2.6 2.7 14.2.2 14.3 14.4.3 14.5 15.2.2 15.3.1 15.3.2 15.4.1 15.4.3 17.5.4 17.5.5 19.2 19.3 19.4 20.5 22.11 23.1 23.2.2 23.2.3 23.4 24.1.1 24.1.2 24.2.1 24.2.1.1 24.2.1.2 24.2.4 24.2.4.1 24.2.4.3 25.2.3 25.5
Pakete	8.1.5 8.2 Selbst erstellen
Panel	17.1 17.6 21.2.6 21.4 22.1 22.11 25.1.2
Parameter von Methoden	7.2.3
PARAM-Tag	2.3 25.2.1
parseByte	7.8.1.3
parseDouble	7.8.1.3
parseFloat	7.8.1.3
parseInt	7.8.1.3 9.2.1
parseLong	7.8.1.3
path.separator	12.8.1
peek	12.2
Performance-Tuning	29
Persistenz	28.1.1
PicoJava	1.1
PipedReader	13.2.2 13.3.2
PipedWriter	13.2.2 13.3.2
plainTextFlavor	20.7.2
play	25.3.1
Pluggable Look and Feel	1.2.3
Point	2.6 19.3
Polygon	14.3.3
pop	12.2
Popup-Menüs	20.6
PopupMenu	20.6 20.8
Port-Nummer	25.4.1 32.1.4
Positionierung des Dateizeigers	13.4.2
POSITIVE_INFINITY	4.2.4.1 4.7 7.8.1.4
Postdekrement	5.2
Postinkrement	5.2
pow	7.8.2.3
Prädekrement	5.2
PreparedStatement	30.4.6
Prepared Statements	30.4.6
prepareStatement	30.4.6
previous	27.3.2

previousIndex	27.3.2
Primitive Datentypen	4.2
Präinkrement	5.2
print	13.2.3.2 15.4.3
Printable	15.4.3
printAll	15.4.3
PrinterJob	15.4.3
PrintGraphics	15.4.1 15.4.3 15.5
PrintJob	15.4.1 15.4.3
println	12.8.2 13.2.3.2 13.2.3.3
printStackTrace	9.2.2
PrintStream	1.2.4 12.8.2
PrintWriter	13.2.2 13.2.3 13.2.3.2 13.2.3.3 13.6
private	1.2.1 7.3.2.1 7.4 7.4.1 7.4.2.1 7.9 28.1.2 29.2.2 31.3.1
processActionEvent	18.2.4
processComponentEvent	23.2.3
processEvent	18.2.4
processFocusEvent	23.2.3
processKeyEvent	18.2.4 23.2.3
processMouseEvent	18.2.4 20.6 23.2.3
Producer/Consumer-Beispiel	10.4.3
-prof	29.3.1
Profiler	26.1.2.3 29.3.1
Projektverwaltung	8.3.2
Properties	12.3.4 12.8.1 12.10 30.2.1
propertyName	12.3.4 12.8.1
protected	1.2.1 7.4.1 7.4.2.2 7.4.2.3 7.9 26.2.1.5 26.2.3.3
Protokoll	32.1.2
Proxy-Server	32.1.6
Präprozessor	4.1.4
public	1.2.1 2.2 7.4 7.4.1 7.4.2.3 7.9 8.2.4.3 8.3.2.1 26.2.1.5 26.2.3.3 28.2.3
push	12.2
PushbackReader	13.3.2 13.3.3 13.3.3.3 13.6
put	12.3.1 27.6.1
putAll	27.6.1

- Q -	
Queue	27.4.1
QuickSort	29.3.2

- R -	
Radiobuttons	22.5
Random	1.2.4 2.4 12.6.1 12.10
Random-Access-Dateien	13.4
RandomAccessFile	13.4 13.4.1 13.4.2 13.4.4 13.6 28.1.2 29.2.4.3
Rückgabewert einer Methode	7.2.4
read	13.3.1 13.4.3 13.4.4 29.2.4.3 32.2.3
readBoolean	13.4.3 28.1.3
readByte	13.4.3 28.1.3
readChar	13.4.3 28.1.3
readDouble	13.4.3 28.1.3
Reader	13.1 13.3.1 13.3.2 13.3.2.1 13.3.2.2 13.3.3 13.3.3.1 13.3.3.3 13.6 29.2.4.2 29.2.4.3
readFloat	13.4.3 28.1.3
readFully	13.4.3
readInt	13.4.3 28.1.3
readLine	13.3.3.1 13.4.3
readLong	13.4.3 28.1.3
read-Methoden	13.4.3
readObject	28.1.3 28.2.1
readShort	13.4.3 13.4.4 28.1.3
readUnsignedByte	13.4.3
readUnsignedShort	13.4.3
readUTF	13.4.3 28.1.3
ready	13.3.1
Rechteck	14.3.2
Rechtsschiebeoperator	5.5
Rectangle	24.2.3
Red-Black-Tree	27.7.2
Referenzgleichheit und -ungleichheit	5.7.2.2
Referenztypen	4.5
Reflection	28.1.2 31
Reflection-API	1.2.4
regionMatches	11.2.4 11.5
Registry	3.4.2.1
Relationale Operatoren	5.3
Remote Method Invocation	1.2.4
remove	20.2 20.3 21.1.3 22.9 27.2.1 27.3.1 27.3.2 27.4.1 27.4.2 27.6.1
removeAll	27.2.1 27.2.2
renameTo	13.5.4
repaint	2.6 19.2 19.6 23.2.3 24.2.1 24.2.1.1 24.2.1.2 24.2.2 24.2.4.1 25.5
repaint-Schleife	24.2.1.1
replace	11.2.6 11.3.1 11.3.3 11.5 String StringBuffer

replaceItem	22.9
replaceRange	22.7
requestFocus	19.5 23.2.3
Request For Comments	32.1.5
RESERVED_ID_MAX	18.1.2
reset	13.2.2.2 13.3.1 28.2.3
Ressourcendatei	20.1
Restwertoperator	5.2
ResultSet	30.2.3 30.3.6 30.4.1 30.4.4 30.4.5 30.5
ResultSetMetaData	30.4.1
resume	10.2.3 10.6
retainAll	27.2.1
return	5.7.2.6 6.2.1.4 7.2.4 7.9 9.2.6
RFC	32.1.5
RFC 1521	20.7.1
RFC 1522	20.7.1
RGB-Farbmodell	16.1
right	25.2.2
RMF-Format	25.3.1
rmix	26.1
rollback	30.4.3
round	7.8.2.4
rt.jar	8.2.2.3
run	7.7.6.1 10.1 10.2.1 10.2.2 10.2.4.2 10.3 10.3.1 10.3.2 10.6 24.2.1.2 24.2.3 25.5 32.3.3
Runden und Abschneiden	7.8.2.4
Runnable	7.7.6.1 7.9 10.1 10.3 10.3.1 10.3.2 10.6 24.2.1.2
RuntimeException	9.3.2.1 9.4

- S -	
SansSerif	15.2.1 15.5
save	12.3.4
Schachteln von Ausgabe-Streams	13.2.3
Schachteln von Eingabe-Streams	13.3.3
Schachteln von Layoutmanagern	21.2.6
Schiebeoperationen	5.5
Schieberegler	22.10
Schleifen	6.3
Schlüsseltransformation	12.3
Schlüsselwörter	4.1.3
Schnelleinstieg	3.5
Schnittstellen (Interfaces)	7.7
Schriftarten	15.2
Scrollbar	18.1.6.9 22.10 22.11
Scrollbar.HORIZONTAL	22.10
SCROLLBARS_BOTH	22.7
SCROLLBARS_HORIZONTAL_ONLY	22.7
SCROLLBARS_NONE	22.7
SCROLLBARS_VERTICAL_ONLY	22.7
Scrollbar.VERTICAL	22.10
ScrollPane	16.4 22.11 22.12
ScrollPane.SCROLLBARS_ALWAYS	22.11
ScrollPane.SCROLLBARS_AS_NEEDED	22.11
ScrollPane.SCROLLBARS_NEVER	22.11
search	12.2
seek	13.4.2
Segment7	23.2.2 23.2.3
select	22.6 22.8 22.9
SELECT	30.4.7.3
selectAll	22.6
SELECT-Anweisung	30.2.3
semidynamische Arrays	1.2.1
Separatoren	20.3
Serialisierung	1.2.4 28.1.1
Serializable	28.1.2 28.1.3 28.2 28.2.1 28.3.1 28.4
serialver	28.2.1
serialVersionUID	28.2.1 28.4
Serif	15.2.1 15.5
ServerSocket	32.2.2 32.3.1 32.3.2 32.5
Servlet-API	1.3.1.5
set	12.4.1 12.7.2 18.1.2
Set	27.1

set	27.3.2
Set	27.5.1 27.5.2 27.6.1 27.7.1 27.7.2 27.7.3 27.9
set	31.4
setActionCommand	20.4.3 20.5 22.3 22.6
setAutoCommit	30.4.3
setBackground	2.3 14.3 17.5.4 17.6
setBlockIncrement	22.10
setBounds	17.3 17.6 21.2.5
setCaretPosition	22.6
setCharAt	11.4.4 11.5
setCheckboxGroup	22.5
setClip	14.4.3 14.5
setColor	16.3 16.5
setColumns	22.6
setConstraints	21.2.4
setContents	20.7.2
setCurrent	22.5
setCursor	17.5.3 17.6
setDouble	31.4
setEchoCharacter	22.6 22.7
setEditable	22.6
setEnabled	20.4.1
setErr	12.8.2
setFont	15.2.1 15.3.1 17.5.5 17.6
setForeground	17.5.4 17.6
setIconImage	17.5.2 17.6
setIn	12.8.2
setInt	31.4
setLabel	20.4.1 22.3 22.4
setLayout	21.1.2 21.2 21.2.4 21.2.5 21.4
setLineNumber	13.3.3.2
setLocation	17.3 17.6 20.5
setMenuBar	20.2
setModal	21.3
setName	10.5.1
setOut	12.8.2
setPriority	10.5.1
setResizable	21.3
setScrollPosition	22.11
setShortCut	20.4.3
setSize	14.3 14.3.3 17.3 17.6 20.5 21.2.5 22.11
setSoTimeout	32.2.3
setState	17.4 17.6 20.4.2 22.4 22.5
setText	22.2 22.6
setTime	12.7.4 13.5.3

<code>setTimeZone</code>	12.7.2
<code>setTitle</code>	17.5.1 17.6
<code>setTransactionIsolation</code>	30.4.3
<code>setUnitIncrement</code>	22.10 22.11
<code>setValue</code>	22.10
<code>setVisible</code>	14.2.1 14.3 17.2 17.6 18.2.2.1 21.1.4
<code>setXORMode</code>	2.6
SGML	Vor
shallow copy	28.3.2
Shape	14.4.3
ShellSort	29.3.2
short	4.2 4.2.3 4.4.2 4.6 4.7 6.2.2.2
Short	7.8.1
short	7.8.1
Short	7.8.1.4 7.9
short	30.2.3 31.3.1
Short-Circuit-Evaluation	5.4
show	20.6
showDocument	25.4.2 25.4.3 25.6
showStatus	25.1.4.2 25.6
Sicherheitsmechanismen	1.2.2
Sichtbarkeit	4.3.3 7.4 7.4.1 lokaler Variablen von Blöcken
Signatur einer Methode	7.2.5.1
signierte Applets	25.1.1
sin	7.8.2.1
size	12.1.1 13.2.2.2 27.1 27.6.1
skip	13.3.1
skipBytes	13.4.2
sleep	10.2.2 10.2.4.1 10.6
Socket	32.2.2 32.3.1 32.4 32.5
SocketException	32.2.3
Sonstige Operatoren	5.7
sort	12.9 27.8.1
SortedMap	27.1 27.7.3
SortedSet	27.1 27.7.2 27.7.3
Sortierte Collections	27.7
Soundausgabe	1.2.3 25.3
South	21.1.3
Speichermanagement	4.5.2
Sprachmerkmale	1.2.1
SQL-2 Entry-Level	30.1.2.3
SQLException	30.2.1 30.2.4 30.2.5 30.3.2
sqrt	7.5.3 7.8.2.3
Stack	12.2 12.10 27.1
Standarderweiterungen	8.2.1.2

Standard-Font	<u>17.5</u>
Standardschriftarten	<u>15.2.2</u>
Standardwert	<u>4.2</u>
Star Seven	<u>1.1</u>
start	<u>10.2.1</u> <u>10.2.2</u> <u>10.3.1</u> <u>10.3.2</u> <u>10.6</u> <u>25.1.1</u> <u>25.1.3.3</u> <u>25.1.3.4</u> <u>25.5</u> <u>25.6</u>
startsWith	<u>11.2.4</u> <u>11.5</u>
Statement	<u>30.2.2</u> <u>30.2.3</u> <u>30.2.4</u> <u>30.3.3</u> <u>30.3.4</u> <u>30.3.5</u> <u>30.3.6</u> <u>30.4.5</u> <u>30.4.6</u>
static	<u>7.3.2.1</u> <u>7.4.2.4</u> <u>7.5</u> <u>7.5.1</u> <u>7.5.2</u> <u>7.5.3</u> <u>7.7.5</u> <u>7.8.2</u> <u>7.9</u> <u>28.1.2</u> <u>28.1.3</u> <u>28.2.1</u> <u>31.3.1</u> <u>31.3.2</u> <u>Statischer Konstruktor</u>
Statische Konstruktoren	<u>7.5.4</u>
Statuszeile des HTML-Browsers	<u>25.1.4.2</u>
Stelligkeit eines Operators	<u>5.1</u>
stop	<u>10.2.2</u> <u>10.2.4.2</u> <u>10.6</u> <u>25.1.3.4</u> <u>25.3.1</u> <u>25.6</u> <u>32.2.3</u>
store	<u>12.3.4</u>
Stored Procedures	<u>30.1</u>
Stream	<u>13.1</u>
String	<u>1.2.4</u> <u>3.5.3.2</u> <u>4.1.1</u> <u>4.2.2</u> <u>4.2.2.1</u> <u>4.5.1</u> <u>5.7.2.2</u> <u>7.4.2.5</u> <u>7.8.1.2</u> <u>8.2.1.1</u> <u>11.1</u> <u>11.2</u> <u>11.2.1</u> <u>11.2.2</u> <u>11.2.3</u> <u>11.2.5</u> <u>11.2.6</u> <u>11.2.7</u> <u>11.3.1</u> <u>11.3.2</u> <u>11.3.2.1</u> <u>11.3.2.2</u> <u>11.3.3</u> <u>11.4.1</u> <u>11.4.2</u> <u>11.4.4</u> <u>11.4.6</u> <u>11.5</u> <u>12.3.4</u> <u>12.5.2</u> <u>13.2.1</u> <u>13.2.2</u> <u>13.2.2.1</u> <u>13.2.2.2</u> <u>13.2.3.3</u> <u>13.3.2</u> <u>13.3.2.2</u> <u>13.3.3.1</u> <u>13.4.1</u> <u>13.4.3</u> <u>17.2</u> <u>17.5.1</u> <u>20.3</u> <u>20.4.1</u> <u>20.4.3</u> <u>20.7.2</u> <u>21.1.3</u> <u>21.2.3</u> <u>21.3</u> <u>22.3</u> <u>22.8</u> <u>22.9</u> <u>25.2.3</u> <u>25.4.1</u> <u>27.4.2</u> <u>27.5.1</u> <u>27.7.1</u> <u>27.7.2</u> <u>28.1.2</u> <u>28.3.1</u> <u>29.2.1.1</u> <u>29.2.1.3</u> <u>29.2.1.4</u> <u>29.2.1.5</u> <u>29.2.2</u> <u>29.3.2</u> <u>30.2.3</u> <u>31.2.1</u> <u>31.3.1</u> <u>31.3.3</u>
StringBuffer	<u>1.2.4</u> <u>4.5.1</u> <u>11.2.6</u> <u>11.3.2.2</u> <u>11.3.3</u> <u>11.4.1</u> <u>11.4.2</u> <u>11.4.3</u> <u>11.4.4</u> <u>11.4.5</u> <u>11.4.6</u> <u>11.5</u> <u>13.2.2.2</u> <u>29.2.1.1</u> <u>29.2.1.2</u> <u>29.2.1.3</u> <u>29.2.3</u> <u>29.4</u>
String einlesen	<u>3.5.3.2</u>
stringFlavor	<u>20.7.2</u>
StringIndexOutOfBoundsException	<u>11.2.2</u> <u>11.4.4</u>
String.intern	<u>5.7.2.2</u>
String-Literale	<u>4.2.2.1</u> <u>11.3.2.1</u>
StringReader	<u>13.3.2</u> <u>13.3.2.2</u> <u>13.6</u>
StringSelection	<u>20.7.2</u>
StringTokenizer	<u>12.1.3</u> <u>12.5</u> <u>12.5.1</u> <u>12.5.2</u> <u>12.10</u> <u>21.3</u>
String-Verkettung	<u>5.7.2.1</u> <u>11.2.3</u> <u>29.2.1.1</u>
stringWidth	<u>15.3.2</u>
StringWriter	<u>13.2.2</u> <u>13.2.2.2</u> <u>13.3.2.2</u> <u>13.6</u>
Structured Generalized Markup Language	<u>Vor</u>
Subqueries	<u>30.4.7.3</u>
subSet	<u>27.7.2</u>
substring	<u>11.2.2</u> <u>11.2.3</u> <u>11.3.3</u> <u>11.5</u> <u>29.2.1.4</u>
Subtraktion	<u>5.2</u>
Suchen in Zeichenketten	<u>11.2.5</u>

<code>sun.boot.class.path</code>	8.2.2.2
Sun HotJava	1.2.2
<code>sun.io</code>	13.2.2.1 13.3.2.1 29.2.4.1
<code>sun.jdbc.odbc.JdbcOdbcDriver</code>	30.2.1
Sun Microsystems	1.1
SunWorld '95	1.1
<code>super</code>	7.3.3.1 7.9
SuperCede	29.1
Superklassenkonstruktor	7.3.3.1
Superklassenmethoden	7.3.2.2
<code>supportsTransactionIsolationLevel</code>	30.4.3
<code>suspend</code>	10.2.3 10.6
Swing Toolset	1.1 1.2.3
<code>switch</code>	1.2.1 6.2 6.2.2.2 6.3.3.3 6.4
Symantec	1.1
Symboldarstellung	17.4
Synchronisieren	von Collections von Threads
<code>synchronized</code>	10.2.2 10.4.2 10.4.2.1 10.4.2.2 10.4.3 10.6 27.1 27.2.2 27.8.2 29.2.1.5 29.2.2 29.2.3
System	2.4 7.5.3 12.8 12.8.1 12.8.2 12.8.6 12.10
<code>SystemColor</code>	16.4
<code>SystemColor.activeCaption</code>	16.4
<code>SystemColor.activeCaptionBorder</code>	16.4
<code>SystemColor.activeCaptionText</code>	16.4
<code>SystemColor.control</code>	16.4
<code>SystemColor.controlDkShadow</code>	16.4
<code>SystemColor.controlHighlight</code>	16.4
<code>SystemColor.controlLtHighlight</code>	16.4
<code>SystemColor.controlShadow</code>	16.4
<code>SystemColor.controlText</code>	16.4
<code>SystemColor.desktop</code>	16.4
<code>SystemColor.inactiveCaption</code>	16.4
<code>SystemColor.inactiveCaptionBorder</code>	16.4
<code>SystemColor.inactiveCaptionText</code>	16.4
<code>SystemColor.info</code>	16.4
<code>SystemColor.infoText</code>	16.4
<code>SystemColor.menu</code>	16.4
<code>SystemColor.menuText</code>	16.4
<code>SystemColor.scrollbar</code>	16.4
<code>SystemColor.text</code>	16.4
<code>SystemColor.textHighlight</code>	16.4
<code>SystemColor.textHighlightText</code>	16.4
<code>SystemColor.textInactiveText</code>	16.4
<code>SystemColor.textText</code>	16.4
<code>SystemColor.window</code>	16.4
<code>SystemColor.windowBorder</code>	16.4

SystemColor.windowText	16.4
System.currentTimeMillis	29.3.3
System.err	26.1.2.2
System.exit	12.8.3 19.1
Systemfarben	16.4
System.in	1.2.4 3.5.3.2 26.1.2.2
System.out	1.2.4 26.1.2.2
System.out.println	2.5 3.5.3.1 12.8.2
System-Properties	12.8.1

- T -	
tailSet	27.7.2
tan	7.8.2.1
TCP/IP	32.1.2
TCP/UDP	32.1.2
Temporäre Dateien	13.5.5.1
text	16.4 22.2
TextArea	18.1.6.11 22.7 22.12
Textausgaben	1.2.4
TextComponent	22.6 22.7
TextEvent	18.1.2 18.1.6.11 22.6
TextField	18.1.6.8 18.1.6.11 20.5 22.6 22.7 22.9 22.12
textHighlight	16.4
textHighlightText	16.4
TextListener	18.1.6.11 22.6
textText	16.4
texttop	25.2.2
textValueChanged	18.1.6.11 22.6
this	4.3.3 6.1.3.2 7.2.2 7.2.6.2 7.3.3.1 7.5.3 8.1.2 10.4.2 10.4.2.1 10.4.2.2 18.2.2.1 20.5 24.1.1 24.1.1.1
Thread	7.7.6.1 8.2.1.1 10.1 10.2.1 10.2.2 10.2.3 10.2.4.1 10.3 10.3.1 10.3.2 10.5.2 10.6 24.2.1.1 24.2.1.2 32.2.3
ThreadGroup	10.5.2
Thread-Gruppen	10.5.2
Thread.MAX_PRIORITY	10.5.1
Thread.NORM_PRIORITY	10.5.1
Threads	10.1 für Animationen
Thread-Synchronisation	10.4
throw	6.2.1.4 9.3.3 9.4
Throwable	9.2.2 9.2.3 9.4
throws	7.8.1.1 9.3.1 9.3.2 9.3.3 9.4
Time	30.2.3
Times New Roman	15.2.1
TimesRoman	15.2.1
Timestamp	30.2.3
TimeZone.getDefault	12.7.2
Titelleiste	17.5
toArray	27.1
toCharArray	13.2.2.2
toLowerCase	11.2.6 11.5
Toolkit	15.2.2 15.4.1 15.4.2 17.3 17.5.3 20.7.2 24.1.1
top	25.2.2
toString	5.7.2.1 7.8.1.2 9.2.2 11.2.7 11.3.2.2 11.4.6 11.5 13.2.2.2 26.1.4.3 28.3.1 29.2.1.1 29.2.1.3 31.2.1 31.4

toUpperCase	11.2.6 11.5 13.2.3.3
Transaction Isolation Level	30.4.3
Transaktionen	30.4.3
Transferable	20.7.1 20.7.2 20.8
transient	7.4.2 28.1.2 28.1.3 28.2.1 28.2.2
Transportschicht	32.1.2
TreeMap	27.7.2 27.7.3
TreeSet	27.7.2 27.7.3
Treibermanager	30.2.1
Trigger	30.1
trim	11.2.2 11.5 29.2.1.4
true	4.1.3 4.2 4.2.1.1 4.7 5.7.2.3 6.2.1.2 6.3.1.2 6.3.3.2 10.2.2 10.2.4.2 11.2.4 12.3.2 12.4 12.5.1 12.5.2 13.2.2.1 13.3.1 13.5.4 13.5.5.2 17.2 19.3 20.4.2 21.3 22.4 22.5 22.9 23.2.3 27.2.1 27.3.1 27.5.1 27.6.1 30.2.3 32.2.3
try	9.2.1 9.2.4 9.2.5 9.2.6 9.3.1 9.3.2 9.3.3 9.4 13.2.2.1 20.7.2 30.3.4 30.4.5
try-catch-Anweisung	9.2.1
Typüberprüfungen	4.3.1
TYPE	31.3.1
Type-Cast-Operator	5.7.1.2
.TYPE-Objekte	31.3.1
Typkonvertierungen	4.6
Typsicherheit	8.3.2.1

- U -	
UDP	32.1.2
UND-Operator	5.4
Ungleichheits-Operator	5.3
Unicode	4.2.2
Unicode-Escape-Sequenzen	4.2.2.1
Unicode-Zeichensatz	4.1.1
Uniform Resource Locator	25.4.1
UNION	30.4.7.3
UnknownHostException	32.2.1 32.2.2
unread	13.3.3.3
UnsupportedOperationException	27.1 27.2.1 27.2.2 27.3.2 27.4.1 27.8.3
Unterlänge	15.3.2
Untermenüs	20.4.4
Unveränderliche Collections	27.8.3
unwrapping	31.3.1
update	2.5 24.2.4.1 24.2.4.2 24.2.4.3 25.5
UPDATE	30.2.4 30.4.7.2
URL	25.4.1 25.4.3 32.4 32.5 relativer
URLConnection	32.4
Usenet-Newsgroups	3.3.2.1
User Datagram Protocol	32.1.2
user.dir	12.8.1
user.home	12.8.1
user.name	12.8.1
user.timezone	12.7.2

- V -	
valueOf	11.2.7 11.5
values	27.6.1
van Hoff, Arthur	1.1
Variablen	4.3
Variablendeklarationen	6.1.3
Vector	1.2.4 3.2 7.7.6.2 10.4.3 12.1 12.1.1 12.1.3 12.5 12.10 24.2.3 24.2.4.2 27.1 27.2.2 28.3.1 29.2.1.1 29.2.3 29.3.1 32.3.3
Vektor	1.2.4
Verdecken von Variablen	8.1.2
Vererbung	7.3
Vergleichen von Zeichenketten	11.2.4
Verkettung	von Konstruktoren von Strings
Veränderbarkeit	7.4
veränderliche Objekte	27.5.1
Versionierung von Klassen	28.2.1
Verwalten von Threads	10.5
Verweise auf andere Seiten	25.4
Verzweigungen	6.2
View einer Collection	27.6.1
Viewport	22.11
virtuelle Ausgabefläche	22.11
virtuelle Java-Maschine	1.2.2
Virtuelle Maschine	29.1
Visual Cafe	1.1 29.1
VK_0	19.6
VK_9	19.6
VK_A	19.6
VK_BACK_SPACE	19.6
VK_DELETE	19.6 20.4.3
VK_DOWN	19.6
VK_END	19.6
VK_ENTER	19.6
VK_ESCAPE	18.2.1 19.6
VK_F1	19.6
VK_F12	19.6
VK_HOME	19.6
VK_INSERT	19.6
VK_LEFT	19.6
VK_PAGE_DOWN	19.6
VK_PAGE_UP	19.6
VK_RIGHT	19.6
VK_SPACE	19.6

VK_TAB	19.6
VK_UNDEFINED	19.6
VK_UP	19.6
VK_Z	19.6
VM	1.2.2
void	5.7.2.6 7.2.4
Void	7.8.1
void	7.8.1
Void	7.9
void	18.1.3 31.3.1
volatile	7.4.2
Vordefinierte Pakete	8.2.1.2
Vordergrundfarbe	17.5
Vorrangregeln	5.8
VSPACE	25.2.2

- W -

Wahlfreier Zugriff auf Dateien	13.4
wait	10.2.2 10.4.3 10.6
waitForAll	2.4 24.1.1.1
Warteliste	10.4.3
wasNull	30.2.3
wav-Dateien	1.2.3
WAV-Format	25.3.1
WebRunner	1.1
weightx	21.2.4
weighty	21.2.4
Weiterführende Informationen	3.3.2
Weitergabe einer Exception	9.3.2
West	21.1.3
WHERE	30.4.7.3
while	1.2.1 6.3.1 6.3.3.3 6.4
WIDTH	25.2.1 25.2.2
Window	14.2.1 17.1 17.2 17.3 17.6 21.1.4
windowActivated	18.1.6.7 19.1
WindowAdapter	14.3 18.1.5 19.1 19.5
windowClosed	18.1.6.7 19.1
windowClosing	14.3 18.1.6.7 19.1 19.5
windowDeactivated	18.1.6.7 19.1
windowDeiconified	18.1.6.7 19.1
Window-Ereignisse	18.1.6.7
WindowEvent	18.1.6.7 19.1 19.7
Window-Events	19.1
windowIconified	18.1.6.7 19.1
WindowListener	18.1.6.7 19.1 19.7
windowOpened	18.1.6.7 19.1
Winkelfunktionen	7.8.2.1
Wrapper-Klassen	7.8.1
write	13.2.1 13.2.2.1 13.2.2.2 13.2.3 13.2.3.1 13.2.3.3 13.3.1 13.4.3 13.4.4 29.2.4.1 29.2.4.3
writeBoolean	13.4.4 28.1.2
writeByte	13.4.4 28.1.2
writeBytes	13.4.4 28.1.2
writeChar	13.4.4 28.1.2
writeChars	13.4.4 28.1.2
writeDouble	13.4.4 28.1.2
writeFloat	13.4.4 28.1.2
writeInt	13.4.4 28.1.2
writeLong	13.4.4 28.1.2

write-Methoden	13.4.4
writeObject	28.1.2 28.2.1 28.2.2 28.2.3
Writer	13.1 13.2.1 13.2.2 13.2.2.1 13.2.2.2 13.2.3 13.2.3.1 13.2.3.2 13.2.3.3 13.3.1 13.3.2 13.6 29.2.4.1 29.2.4.3
writeShort	13.4.4 28.1.2
writeTo	13.2.2.2
writeUTF	13.4.4 28.1.2

- X -	
xor	12.4.2
XOR-Modus	2.6
-Xprof	29.3.1
-Xrunhprof	29.3.1

<div>- Y -</div>	
yield	32.2.3

- Z -	
ZapfDingbats	15.2.2
Zeichenextraktion	11.2.2
Zeichenketten	1.2.4 11.1
Zeichentasten	19.6
Zeichentyp	4.2.2
Zeilenabstand	15.3.2
Zeitzone	12.7
ZeitzoneAngabe	12.7.2
ZONE_OFFSET	12.7.2
Zufallszahlen	1.2.4 12.6.1
Zufallszahlengeneratoren	2.4
Zuweisung	4.3.1
Zuweisungsoperatoren	5.6
Zwischenablage	20.7