Oldenbourg

Java 2

Von den Grundlagen bis zu Threads und Netzen

von
Prof. Dr. Ernst-Wolfgang Dieterich
Fachhochschule Ulm

2., überarbeitete Auflage

Oldenbourg Verlag München Wien

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Dieterich, Ernst-Wolfgang:
Java 2 : von den Grundlagen bis zu Threads und Netzen /
von Ernst-Wolfgang Dieterich. – 2., überarb. Aufl.. -
München ; Wien : Oldenbourg, 2001
ISBN 3-486-25423-5

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg-verlag.de

Lektorat: Irmela Wedler
Herstellung: Rainer Hartl
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: R. Oldenbourg Graphische Betriebe Druckerei GmbH

Inhaltsverzeichnis

1 Einleitung

Keine Programmiersprache hat jemals eine so große Popularität erzielt wie Java – und dies nicht nur in Programmiererkreisen. In Tageszeitungen und den Nachrichten im Fernsehen wird immer wieder über Java berichtet. Der Grund hierfür liegt sicherlich auch im Einsatzgebiet von Java, der Erstellung von Programmen für das Internet.

Die ersten Vorarbeiten zu Java gehen in das Jahr 1991 zurück, in dem eine Arbeitsgruppe um Patrick Naughton und James Gosling bei Sun Microsystems an einer Programmierumgebung namens *Oak* arbeiteten, mit der man leicht Anwendungen für elektronische Geräte oder das interaktive Fernsehen erstellen kann. Hier wird die Elektronik häufig durch modernere und billigere Komponenten ersetzt. Die Software soll sich dabei möglichst ohne Aufwand den neuen Gegebenheiten anpassen. Daraus ergaben sich die zentralen Randbedingungen für *Oak*: Es sollte klein und vollständig plattform-unabhängig sein. Die Entwickler griffen auf eine bekannte Technik zurück, die N. Wirth bereits 1971 bei seiner Programmiersprache Pascal angewendet hat [Wir71]: Das Quellprogramm wird in eine kompakte Zwischensprache übersetzt, die auf verschiedenen Rechner durch einen Interpreter ausgeführt werden. Der Interpreter realisiert eine hypothetische Maschine, die sog. *virtuelle Maschine*.

Das *Oak*-Projekt kam ins Stocken, da sich kein Anwender dafür interessierte und auch die Einführung des interaktiven Fernsehens nicht vorankam. Der Durchbruch des WWW, des World Wide Web, eröffnete dem *Oak*-Team plötzlich neue Perspektiven. Ein WWW-Browser stellt genau die Anforderungen, die das *Oak*-Team verfolgte: Der Browser muss auf allen denkbaren Plattformen laufen. Inzwischen hatte sich herausgestellt, dass der Name *Oak* bereits für eine andere Programmiersprache vergeben war. In einer Kaffeepause soll sich das Team für den neuen Namen *Java* entschieden haben.

Im Frühjahr 1995 wurde der erste WWW-Browser unter dem Namen *HotJava* vorgestellt, der in Java geschrieben war.

1996 veröffentlichte Sun die erste offizielle Entwicklungsumgebung für Java als Version 1.02. Im Oktober 1999 wurde die Version 1.2.2 veröffentlicht, die offiziell unter dem Namen Java 2 SDK verbreitet ist, auf der auch dieses Buch basiert. Zur Zeit ist die Version 1.3.1 aktuell. Sie kann kostenlos über die Adresse **`http://java.sun.com`** heruntergeladen werden. Neben vielen Doku-

mentationen, darunter auch die Original-Sprachspezifikation Java [GoJo96] und [GoJo00], findet man hier auch interessante Zusatztools und Diskussionsforen.

Eigenschaften von Java

Die Java-Entwickler beschreiben in ihrem *White Paper* die Ziele Javas mit folgenden Schlagworten:

einfach (und vertraut)
: Java enthält sehr wenige Sprachkonstrukte und basiert auf C++.

objektorientiert
: Java ist vollständig objektorientiert; nur Klassen werden programmiert.

verteilt
: Java stellt standardmäßig mächtige Werkzeuge zur Netzwerkprogrammierung zur Verfügung.

robust
: Java-Programme müssen zuverlässig arbeiten. Strenge Typ- und Ausnahmebehandlung sind hierfür Voraussetzungen. Die gefährlichen Sprachmittel von C++ – etwa Zeiger und Sprünge – wurden aus Java verbannt.

sicher
: Sicherheit im Netz ist ein sehr aktuelles Thema. Java verfügt über komfortable Klassen zur Sicherheitsbehandlung. Das Thema Sicherheit wird nie ganz abgeschlossen sein; eine Arbeitsgruppe arbeitet ständig am Sicherheitsmechanismus von Java.

neutral in Bezug auf Rechner-Architektur

interpretativ
: Java erzeugt einen Code, der von der JVM – der virtuellen Java-Maschine – interpretiert wird.

portabel
: Bereits bei der Definition der Basisdatentypen achtet Java – anders als C++ – auf Maschinenunabhängigkeit.

leistungsstark
: Dies ist in bezug auf die Zielrichtung von Java das fragwürdigste Schlagwort: Interpretative Bearbeitung ist naturgemäß langsamer als bei Programmen, die in Maschinencode *eines speziellen* Rechners übersetzt wird. Trotzdem wird an einem echten Java-Compiler gearbeitet!

Thread-unterstützend (multithreaded)
: Das Thread-Konzept zum Programmieren parallel laufender Programme ist fest in Java integriert.

dynamisch
: Klassenobjekte werden erst bei Bedarf zur Laufzeit geladen.

Zielsetzung des Buches

Das Buch wendet sich an Leser, die sich von Grund auf in das objektorientierte Programmieren in Java 2 einarbeiten wollen. Jede Sprachkonstruktion von Java wird genau beschrieben und an kleinen prägnanten Beispielen veranschaulicht. Java 2 verfügt über eine riesige Menge von sog. Packages, die Klassen für komplexe Anwendungen bereitstellen. Bei der Auswahl der behandelten Packages wurde darauf geachtet, dass jeder wichtige Themenbereich, der mit Java-Programmen abgedeckt werden soll, möglichst ausführlich behandelt wird. Bei der tabellarischen Beschreibung der Klassen und ihrer Fähigkeiten werden zumindest die in den Beispielen verwendeten Methoden genau beschrieben. Eine vollständige Beschreibung findet man in der Online-Dokumentation, die man sich mit der Entwicklungsumgebung kostenlos herunterladen kann.

Das Buch ist so aufgebaut, dass es sich zum Selbststudium ebenso eignet wie als Begleitlektüre zu einem Kurs über objektorientiertes Programmieren in Java. Das ausführliche Sachwortverzeichnis sowie die Syntaxdiagramme und das zugehörige Register machen das Buch auch zu einem nützlichen Nachschlagewerk.

Die Beispiele liegen mit einer ausführlichen Beschreibung im html-Format im Internet bereit, die über die Verlagsseite

`http://www.oldenbourg.de/verlag/index_inhalt.htm`

erreichbar sind.

Gliederung

Kapitel 2 beschreibt die im Java Development Kit (JDK) enthaltenen Werkzeuge. Kapitel 3 gibt einen ersten Einblick in die Leistungsfähigkeit von Java. Dort entwickeln wir drei sehr einfache Java-Programme.

Teil I beginnt mit der Einführung der primitiven Datentypen und Variablen (Kapitel 4) und der Ausdrücke (Kapitel 5). Kapitel 6 behandelt die Anweisungen.

> Leser mit Erfahrung in C++ sollten diese Kapitel überfliegen und dabei besonders auf die mit der hier verwendeten Markierung "C++" achten. Hier werden die kleinen, aber wesentlichen Unterschiede zwischen Java und C++ angegeben.

Teil II behandelt die objektorientierten Sprachmittel von Java: Klassen und Objekte (Kapitel 7), die Packages (Kapitel 8), die für die Modularisierung größerer Java-Programme zuständig sind, sowie die Vererbung (Kapitel 9). In Kapitel 10 werden Klassen für zusammengesetzte Datentypen behandelt. Einige wichtige Klassen der Java-Bibliothek werden in Kapitel 11 besprochen. Anders als in

C++ ist in Java die Ausnahmebehandlung konsequent und durchgängig realisiert; dies ist Thema von Kapitel 12.
Teil III beschäftigt sich mit der Anwendungsprogrammierung in Java 2. In Kapitel 13 werden die Grundlagen der Grafikprogrammierung gelegt. Applets sind Java-Programme, die in einem Browser ausgeführt werden. Diesem Thema widmet sich das Kapitel 14.
Java 2 enthält eine Fülle von Klassen, die der Programmierung grafischer Benutzeroberflächen dienen. Die Interaktion mit dem Benutzer ist ereignisgesteuert. Wie man dies in Java programmiert, wird in Kapitel 15 behandelt.
Java unterstützt die parallele Abarbeitung von Programmen. Hierfür werden sog. Threads verwendet, die wir in Kapitel 16 kennenlernen werden. Bei der Ein-/Ausgabe verwendet Java 2 das Stromkonzept, das auch schon in C++ realisiert wurde. Hier ist die Ein-/Ausgabe nicht nur auf Dateien beschränkt; auch die Datenübertragung im Netzwerk wird durch Ströme realisiert. Kapitel 17 beschäftigt sich mit diesem Thema, bevor im abschließenden Kapitel 18 auf die Programmierung von Netzwerkanwendungen eingegangen wird.
Teil IV enthält vier Anhänge. In Anhang A findet man einen alphabetisch sortierten Index der Syntaxdiagramme, die in den verschiedenen Kapiteln die Sprachkonstrukte von Java formal beschreiben. Anhang B enthält ein Verzeichnis der Beispiele. Das in fast allen Programmen verwendete Package **`JavaPack`** ist in Anhang C im Quellcode abgedruckt. Die Beispielsammlung, die man vom Internet herunterladen kann, enthält eine ausführliche Dokumentation des Packages im HTML-Format.
Nach dem Quellenverzeichnis in Anhang D bildet das ausführliche Sachwortverzeichnis den Abschluss dieses Buches.

Danksagung
Dem Lektorat DV/Informatik des Oldenbourg Wissenschaftsverlags danke ich für die gute Zusammenarbeit und das zügige und sorgfältige Lektorieren des Manuskripts. Mein Sohn Holger hat versucht, die letzten Unklarheiten aus dem Text aufzuspüren. Vielen Dank, Holger!

2 Java-Entwicklungsumgebung

Sämtliche Programme sowie die Dokumentationen werden von Sun Microsystems über das Internet kostenlos zur Verfügung gestellt. Die Entwicklungsumgebung JDK (Java Development Kit) enthält die Programme, die man zum Übersetzen und Ausführen von Java-Programmen benötigt. Sie sind alle Kommandozeilen-orientiert, das heißt, man muss sie in einem DOS-Fenster starten. Das JDK verfügt über einen komfortablen Dokumentationsgenerator, der speziell geschriebene Kommentare des Java-Programms erkennt und daraus eine Dokumentation im HTML-Format generiert. Dies wird im zweiten Abschnitt behandelt.

2.1 JDK von Sun Microsystems

Das JDK liegt in der WWW-Adresse `java.sun.com` zum Herunterladen bereit. Die Datei ist selbstentpackend. Am besten kopieren Sie diese gleich in Ihr Java-Verzeichnis und entpacken sie. Beachten Sie die Installationshinweise.
Zum Übersetzen und Ausführen von Java-Programmen benötigt man folgende Programme des JDK:

Der Java-Compiler `javac` übersetzt ein Java-Programm in ein sog. Bytecode-Programm. Das ist ein kompaktes Java-Programm, das interpretativ ausgeführt werden kann.

Der Java-Interpreter `java` führt eine Java-Anwendung aus. Das sind alle Java-Programme mit Ausnahme der Applets.

Ein Applet wird über eine HTML-Seite von einem Browser oder dem im JDK enthaltenen `AppletViewer` gestartet. In diesem Buch wird der `AppletViewer` und `Hotjava`, das ebenfalls bei Sun verfügbar ist, verwendet.

Wie die Übersetzung und Ausführung von Java-Programmen abläuft, wird im nächsten Kapitel erklärt.

2.2 Das Werkzeug `javadoc`

Java verfügt über spezielle Dokumentationskommentare, die folgendermaßen aufgebaut sind:

```
/**
Dokumentationskommentar mit Tags
*/
```

Das JDK enthält das Programm **`javadoc`**, das aus einem Java-Programm diese Dokumentationskommentare herausfiltert und daraus ein HTML-Dokument erstellt.
Mit Dokumentationskommentaren kann man Klassen und Interfaces sowie Methoden und Attribute (siehe Teil II) dokumentieren. Diese Kommentare müssen *unmittelbar vor* den entsprechenden Deklarationen im Quelltext stehen.
Zur Erzeugung spezieller Informationen kann man in den Dokumentationskommentar sog. Tags einfügen, die in der folgenden Tabelle zusammengestellt sind.

Wichtige Tags des Dokumentationskommentars	
`@version text`	Der **`text`** wird unter der Überschrift **Version:** in die Dokumentation übernommen. Ist nur in Klassen- und Interface-Dokus wirksam.
`@author text`	Der **`text`** wird unter der Überschrift **Author:** in die Dokumentation übernommen. Ist nur in Klassen- und Interface-Dokus wirksam.
`@param pn text`	**`pn`** ist ein Parametername der beschriebenen Methode; **`text`** enthält die zugehörige Beschreibung. Unter der Überschrift **Parameters:** wird der Parametername mit der Beschreibung **`text`** übernommen.
`@return text`	Hier wird der Ergebniswert einer Methode beschrieben. Der **`text`** wird unter der Überschrift **Returns:** in die Dokumentation übernommen.

Wichtige Tags des Dokumentationskommentars	
`@throws e txt` `@exception e txt`	**e** ist eine Ausnahme; **txt** enthält die zugehörige Beschreibung. Unter der Überschrift **Throws:** wird die Beschreibung **txt** übernommen. Beide angegebenen Tags bewirken dasselbe.
`@see name`	Hier werden Verweise auf andere Einträge unter der Überschrift **See Also:** erzeugt.
`{@link name txt}`	Fügt einen Verweis auf **name** ein, der durch den Text **txt** angezeigt wird.
`@depreciated txt`	Gibt unter der Überschrift **Depreciated:** den Text **txt** an, der besagt, dass dieses nicht mehr verwendet werden sollte, aber dennoch übersetzt wird.
`@since txt`	Unter der Überschrift **Since:** wird die Beschreibung **txt** übernommen.

Bei den beiden Tags **@see** und **@label** können die in HTML üblichen Verweise benutzt werden; ist **name** ein Klassen- oder Attributname, wird automatisch ein Verweis darauf erstellt.
Zur Demonstration dieser Tags betrachten wir das folgende Java-Programm. Natürlich sind hierbei die Sprachkonstruktionen aus Java noch nicht bekannt; schauen Sie sich dieses Beispiel nochmals an, wenn Sie die Klassen in Teil II kennengelernt haben.

❑ *Beispiel 2.2.1 Dokumentationskommentare*

```
import java.io.*;
/**
*  Klasse JavaDoc
*  @author E.-W. Dieterich
*  @version 2.0, Februar 2001
*/
public class JavaDoc
{
/**
*  Dummy-Methode.
*  Hierher wird verwiesen.
```

```
*  Es wird auf
*  {@link #Format(String,int,char) Format} verwiesen
*  @deprecated "ab 2. Auflage dieses Buches"
*  @throws IOException Standard-Ein-Ausgabe-Ausnahme
*/
   public void dummy() throws IOException
   {}
/**
*  Formatierte Ausgabe eines Strings.
*  @see JavaDoc#dummy
*  @param s auszugebender String
*  @param breite Ausgabebreite
*  @param fuell Füllzeichen
*  @return formatierter String
*  @since Java, Auflage 2
*/
   public static String Format(String s,int breite,
        char fuell)
   {  String erg=new String();
      // irgendwas
      return erg;
   }
}
```

■

Die folgende Abbildung zeigt einen Ausschnitt der Dokumentation, wie sie der Browser **HotJava** darstellt.

Class JavaDoc - Netscape

Datei Bearbeiten Ansicht Gehe Communicator Hilfe

Zurück Vor Neu laden Anfang Suchen Guide Drucken Sicherheit Shop Stop

Constructor Summary

`JavaDoc()`

Method Summary

`void`	**dummy**() **Deprecated.** *ab 2. Auflage dieses Buches*
`static java.lang.String`	**Format**(`java.lang.String s, int breite, char fuell`) Formatierte Ausgabe eines Strings.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

JavaDoc

```
public JavaDoc()
```

Method Detail

dummy

```
public void dummy()
          throws java.io.IOException
```

Deprecated. *ab 2. Auflage dieses Buches*

Dummy-Methode. Hierher wird verwiesen. Es wird auf Former verwiesen.

Dokument: Übermittelt

Abb. 2.1: Dokumentation mit Javadoc

3 Erste Eindrücke von Java

Nachdem die Java-Entwicklungsumgebung erfolgreich installiert wurde, können wir mit der Arbeit beginnen. An drei kleinen typischen Java-Programmen werden erste Eindrücke über die Leistungsfähigkeit von Java vermittelt.
Java kennt drei verschiedene Typen von Programmen:

Java-Anwendungen werden von der Kommandozeile aus gestartet und arbeiten ausschließlich im Textformat. Dies entspricht den ausführbaren Programmen, wie man sie von anderen Programmiersprachen kennt. In Teil I und II werden wir ausschließlich Java-Anwendungen verwenden.

Seit der Einführung von Betriebssystemen mit grafischer Benutzeroberfläche ist es Mode geworden, auch Anwenderprogramme in Grafikfenstern ablaufen zu lassen. In Abschnitt 3.2 wird dies an einem einfachen Beispiel demonstriert. In Kapitel 13 werden grafische Programme genauer behandelt.

Das Haupteinsatzgebiet von Java ist die Erstellung von Programmen, die über das Netz aufgerufen werden und in einem Browser ablaufen. Solche Programme heißen Applets. In Abschnitt 3.3 werden wir ein einfaches Applet entwikkeln. Der genaue Aufbau von Applets wird in Kapitel 14 besprochen.

3.1 Eine Java-Anwendung

In einem Texteditor wird folgendes Java-Programm geschrieben und unter dem Dateinamen **ersteAnwendung.java** abgespeichert. Achten Sie darauf, dass Sie genau diesen Namen verwenden.

❑ *Beispiel 3.1.1 Java-Anwendung*

```
public class ersteAnwendung
{ public static void main(String[] argv)
  { System.out.println("***************************");
    System.out.println("*        Java 2           *");
    System.out.println("* Von den Grundlagen bis *");
    System.out.println("* zu Threads und Netzen  *");
    System.out.println("*          von            *");
    System.out.println("*    E.-W. Dieterich      *");
```

```
    System.out.println("***************************");
  }
}
```

In Java wird wie in C zwischen Groß- und Kleinschreibung unterschieden. Auch bei der Wahl des Dateinamens ist die Groß-/Kleinschreibung zu beachten. Prinzipiell müssen Java-Programme unter dem Namen abgespeichert werden, der hinter **class** steht, und die Erweiterung **java** haben. ■

Um dieses Programm zu übersetzen, gibt man in der Befehlszeile folgendes ein:

```
javac ersteAnwendung.java
```

Dies startet den Java-Compiler, der bei fehlerfreiem Programm die Datei **ersteAnwendung.class** erzeugt. Wenn das Java-Programm Fehler enthält, meldet der Compiler diese in der üblichen Form mit Fehlertext und Zeilennummer der Fehlerstelle.
Das übersetzte Java-Programm wird vom Java-Interpreter ausgeführt, den man mit folgender Befehlszeile startet:

```
java ersteAnwendung
```

Achten Sie darauf, dass der Compiler **javac** die Endung **java** der Datei verlangt, dagegen wird beim Interpreter der Dateiname ohne Endung angegeben.
Das Programm gibt einen Text im Textfenster aus, wie die folgende Abbildung zeigt.

Abb. 3.1: Eine Java-Anwendung und ihre Übersetzung

3.2 Ein Grafikprogramm

Java stellt eine Fülle von Sprachmitteln für die grafische Programmierung zur Verfügung. Im folgenden Grafikprogramm wird der Text in einem Grafikfenster angezeigt.

❑ *Beispiel 3.2.1 Einfache Grafik*
Das folgende Programm wird in der Datei **ersteGrafik.java** gespeichert.

```
import java.awt.*;
import JavaPack.*;  // enthält Klasse FensterBeenden
import javax.swing.*;

public class ersteGrafik extends JFrame
{  ersteGrafik()
   {  addWindowListener(new FensterBeenden());  }
   public void paint(Graphics g)
   {  Font schrift=new Font("Courier",Font.BOLD,14);
      g.setFont(schrift);
      g.drawString("*************************",
        40,55);
      g.drawString("*       Java 2          *",
        40,65);
      g.drawString("* Von den Grundlagen bis *",
        40,75);
      g.drawString("* zu Threads und Netzen  *",
        40,85);
      g.drawString("*          von          *",
        40,95);
      g.drawString("*    E.-W. Dieterich    *",
        40,105);
      g.drawString("*************************",
        40,115);
   }
   public static void main(String[] argv)
   {  JFrame f=new ersteGrafik();
      f.setTitle("einfaches Grafikprogramm");
      f.setSize(300,130);
      f.setVisible(true);
   }
}
```

■

Dieses Programm wird genau wie das erste Programm übersetzt und gestartet. Es liefert folgende Ausgabe:

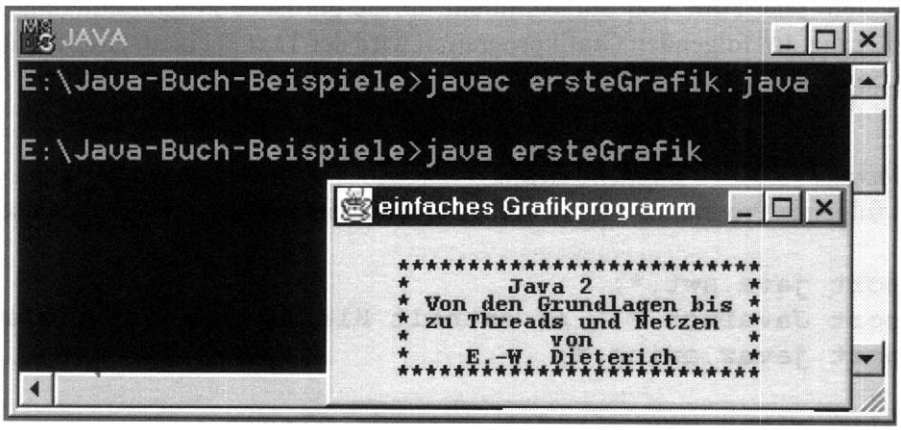


Abb. 3.2: Ein Grafikprogramm und seine Übersetzung

Hier eine kurze Erklärung, wie dieses Programm arbeitet:

Die Hilfsmittel zur Grafikprogrammierung stellt Java in den Packages **`java.awt`** und **`javax.swing`** zur Verfügung. Ein Package kann man sich vorläufig etwa so vorstellen wie eine Bibliothek in anderen Programmiersprachen. Unser Package **`JavaPack`** ist in diesem Beispiel für das Beenden des Grafikfensters zuständig; es definiert den Begriff **`FensterBeenden`**.

Betrachten wir zunächst das Hauptprogramm **`main`**: Es wird ein Grafikrahmen erzeugt (in Java ist dies ein **`JFrame`**) und die Fenstergröße festgelegt. Die letzte Zeile zeigt das Grafikfenster an.

Was in dem Grafikfenster gezeichnet werden soll, beschreibt der Block hinter **`paint`**: Zunächst wird eine Schriftart definiert und eingestellt. Hier wird Courier fett der Größe 14 Punkte gewählt. Die angegebenen Zeilen werden gezeichnet, wobei die beiden letzten Parameter von **`drawString`** die (x,y)-Position des Textanfangs im Fenster angeben.

3.3 Ein Java-Applet

Eines der wesentlichen Entwurfsziele von Java war die einfache Programmierung von Applets. Ein Applet ist ein Java-Programm, das von einem Browser geladen und im Browserfenster ausgeführt wird.

Die Sprachmittel zur Applet-Programmierung stellt Java im Package **javax.swing** zur Verfügung. Das folgende Applet-Programm enthält nur den **paint**-Teil des Grafikprogramms.

❑ *Beispiel 3.3.1 Einfaches Applet*
Dieses Programm benutzt nur das Package **javax.swing**, das in der ersten Zeile angegeben ist. Der Programmteil **extends JApplet** gibt an, dass das Programm ein Applet ist.

```
import java.awt.*;
import javax.swing.*;

public class erstesApplet extends JApplet
{  public void paint(Graphics g)
   {  Font schrift=new Font("Courier",Font.BOLD,14);
      g.setFont(schrift);
      g.drawString("*************************",
        40,55);
      g.drawString("*       Java 2          *",
        40,65);
      g.drawString("* Von den Grundlagen bis *",
        40,75);
      g.drawString("* zu Threads und Netzen  *",
        40,85);
      g.drawString("*          von          *",
        40,95);
      g.drawString("*    E.-W. Dieterich    *",
        40,105);
      g.drawString("*************************",
        40,115);
   }
}
```

■

Dieses Programm wird in einer Seite gestartet, die mit einem Browser betrachtet wird. Ein Browser interpretiert eine Datei, die in der Seitenbeschreibungssprache HTML geschrieben ist.
Die HTML-Anweisung, die das obige Applet startet, enthält in ihrer einfachsten Form die beiden folgenden Zeilen:

```
<APPLET CODE="erstesApplet.class" WIDTH=300 HEIGHT=130>
</APPLET>
```

Diese beiden Zeilen speichern wir in der Datei **`erstesApplet.html`**, die mit jedem Java-fähigen Browser geladen werden kann. Die Java-Entwicklungsumgebung enthält das Programm **`AppletViewer`**, welches hier verwendet wird. Die folgende Abbildung zeigt wieder das Kommandofenster mit der benötigten Befehlsfolge sowie die Ausgabe des Applets.

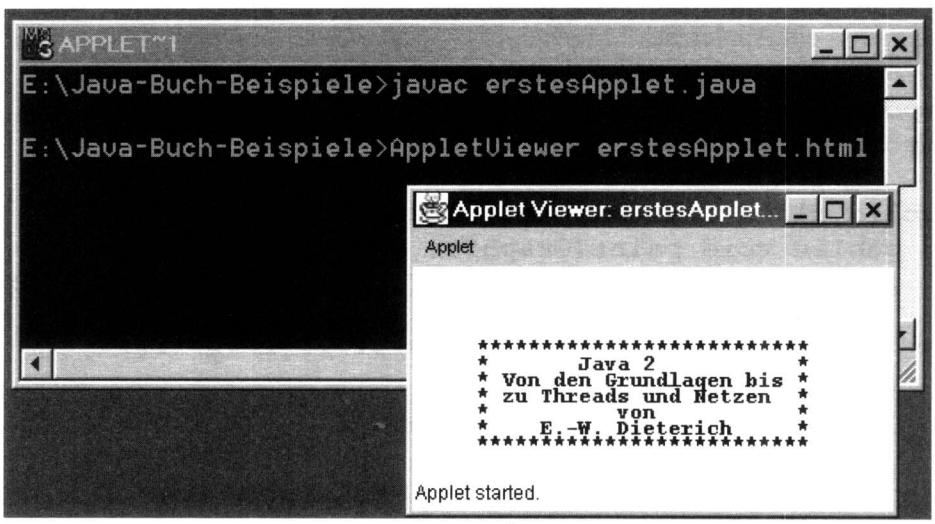


Abb. 3.3: Ein Applet und seine Übersetzung

Teil I.
Grundlagen

Übersicht

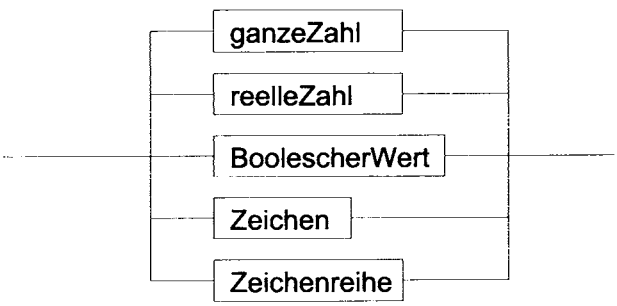
Dieser erste Teil behandelt die grundlegenden Sprachkonstrukte von Java. Nachdem in Kapitel 4 die in Java verfügbaren Datentypen vorgestellt werden, werden wir in Kapitel 5 die Operationen besprechen, mit denen Variablen und Werte verknüpft werden können. Das letzte Kapitel dieses Teils stellt die Kontrollstrukturen vor, mit denen man übersichtlich und komfortabel den Ablauf des Programms steuern kann.

Leser, die mit C++ vertraut sind, können diesen Teil überfliegen und dabei besonders auf solche Bemerkungen wie diese achten, die mit C++ markiert sind.

4 Primitive Datentypen und Variablen

Java ist eine Programmiersprache, die auf allen Rechnerplattformen gleich ablaufen soll. Dieses Ziel hatte bisher jede höhere Programmiersprache; die Festlegung der Basisdatentypen ist jedoch in allen höheren Programmiersprachen, wie Pascal oder C++, bereits maschinenabhängig. So belegt z. B. eine ganze Zahl ein Maschinenwort, das auf unterschiedlichen Plattformen verschiedene Längen (etwa 16 Bit oder 32 Bit) und damit verschiedene Zahlenbereiche hat.
Java legt dagegen für jeden primitiven Datentyp eine feste Speichergröße und damit einen festen Zahlenbereich fest. Java kennt ganzzahlige und reelle Datentypen mit Vorzeichen sowie je einen Datentyp für Wahrheitswerte und für Buchstaben.
In diesem Kapitel werden die konstanten Werte und die Deklaration von Variablen für die primitiven Datentypen von Java behandelt.

Konstante : (4-1)



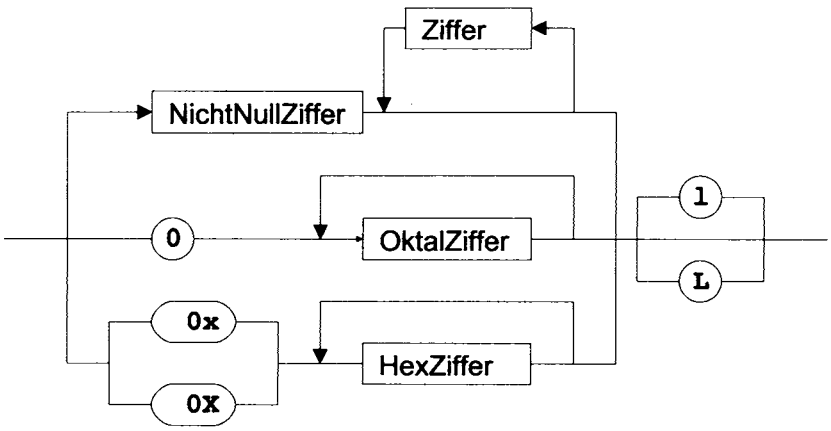
4.1 Ganze Zahlen

In Java gibt es vier ganzzahlige Datentypen, die in der folgenden Tabelle zusammengestellt sind. Bei der Deklaration kann einer ganzzahligen Variablen ein Anfangswert mitgegeben werden.

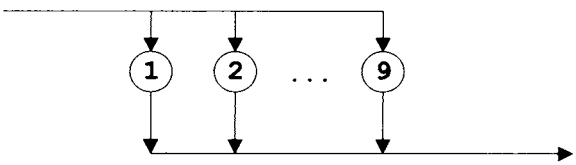
Typ	Bit	Zahlenbereich	
`byte`	8	`-128 .. 127`	-2^{7} .. $2^{7}-1$
`short`	16	`-32.768 .. 32.767`	-2^{15} .. $2^{15}-1$
`int`	32	`-2.147.483.648 .. 2.147.483.647`	-2^{31} .. $2^{31}-1$
`long`	64	ca $-92*10^{17}$ .. $92*10^{17}$	-2^{63} .. $2^{63}-1$

Ganzzahlige Konstanten können dezimal, hexadezimal oder oktal geschrieben werden. **`long`**-Konstanten werden mit einem angehängten **`L`** oder **`l`** gekennzeichnet. Da man den Buchstaben "**`l`**" leicht mit der Ziffer "**`1`**" verwechselt, sollte man das Suffix "**`L`**" für **`long`**-Konstanten benutzen. Die folgenden Syntaxdiagramme zeigen den Aufbau von ganzen Zahlen.

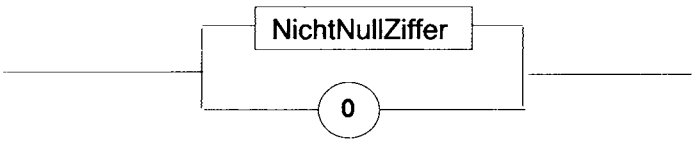
ganzeZahl : (4-2)



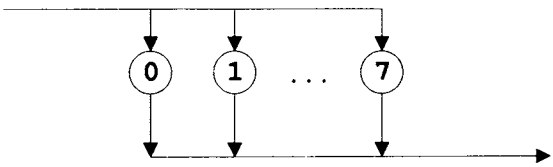
NichtNullZiffer : (4-3)



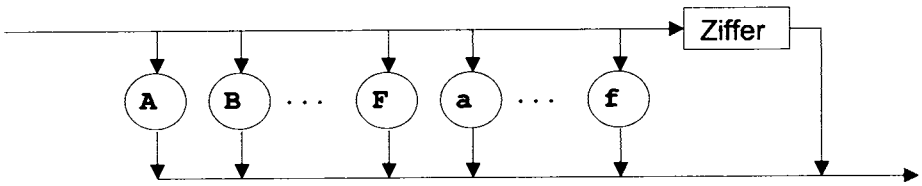
Ziffer : (4-4)



OktalZiffer : (4-5)



HexZiffer : (4-6)



❑ *Beispiel 4.1.1 Beispiele ganzer Zahlen*
Die ersten beiden Zeilen zeigen einige **int**-Konstanten, die beiden nächsten Zeilen enthalten Beispiele für **long**-Konstanten.

```
1997     03715            0x7CD          0X007CD
-4711    037777766631     0xffffed99     0XFFffEd99

1234567890L  011145401322L          0X499602D21
-987654321L  017777777777705103135 17L
                                    0xffffffffc521974f1
```

Die Zahlen in einer Zeile haben jeweils den gleichen Wert. Beachten Sie, dass das letzte Zeichen der Zahlen am Ende der beiden letzten Zeilen ein kleines **l** und keine Eins ist. ■

❑ *Beispiel 4.1.2 Maximale und minimale Werte*
Das folgende Beispiel gibt die maximalen und minimalen Werte für **int** und **long** aus.

```
public class MaximumMinimum
{  public static void main(String[] argv)
   {System.out.println("minimaler \"int\"-Wert    :"
          +Integer.MIN_VALUE);
    System.out.println("\tmaximaler \"int\"-Wert    :"
          +Integer.MAX_VALUE);
    System.out.println("minimaler \"long\"-Wert   :"
          +Long.MIN_VALUE);
    System.out.println("\tmaximaler \"long\"-Wert   :"
         +Long.MAX_VALUE);
   }
}
```

Das Programm liefert folgende Ausgabe:

```
minimaler "int"-Wert    :-2147483648
        maximaler "int"-Wert    :2147483647
minimaler "long"-Wert   :-9223372036854775808
        maximaler "long"-Wert   :9223372036854775807 ■
```

In Java haben alle primitiven Datentypen eine feste, maschinenunabhängige Länge.

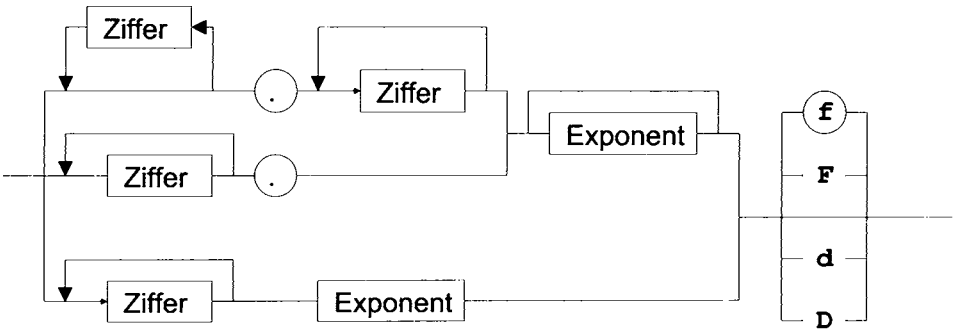
4.2 Reelle Zahlen

Für reelle Zahlen gibt es in Java zwei Datentypen: **float** und **double**. Die interne Darstellung folgt dem im *IEEE ANSI/IEEE Standard 754* (IEEE, New York) festgelegten Format.

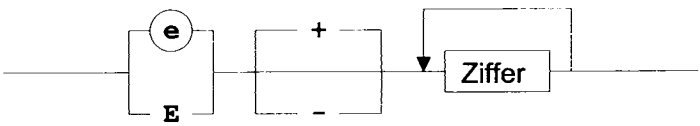
Typ	Bit	Zahlenbereich: **max .. min**
float	32	ca. $3.4028*10^{38}$ .. $1.402*10^{-45}$
double	64	ca. $1.798*10^{+308}$ .. $4.94*10^{-324}$

"Zahlenbereich" ist hier so zu verstehen, dass die erste Zahl **max** der betragsmäßig größte darstellbare Wert, die zweite Zahl **min** der betragsmäßig kleinste von 0 verschiedene Wert ist. Ferner sind bei diesen Typen noch spezielle Werte definiert: positiv und negativ unendlich, eine positive und negative Null sowie der Wert NaN (not a number).

reelleZahl : (4-7)



Exponent : (4-8)



Ganze Zahlen können auch als **float**- oder **double**-Konstanten verwendet werden. Kommt in einer Zahl ein Punkt und/oder ein Exponententeil vor, ist dies dann eine **double**-Konstante, falls kein Suffix angegeben wird. Das Suffix **F** bzw. **D** steht für **float**- bzw. **double**-Konstante.

❑ *Beispiel 4.2.1 Beispiele reeller Zahlen*
Reelle Zahlen sind

```
1.0     .432     12.345e6     -6.45e-123D     32.12f     ■
```

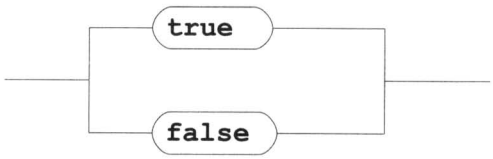
4.3 Wahrheitswerte

Der Datentyp **`boolean`** hat als Wertebereich die beiden Wahrheitswerte **`true`** und **`false`**.

Typ	Bit	Wertebereich
`boolean`	1	**`true, false`**

Der Datentyp **`boolean`** ist ein eigener Datentyp in dem Sinn, dass er nicht in einen anderen Datentyp umgewandelt werden kann.

BoolescherWert : (4-9)



Insbesondere kann der Datentyp **`boolean`** nicht als **`int`**-Wert behandelt werden.

4.4 Zeichen und Zeichenreihen

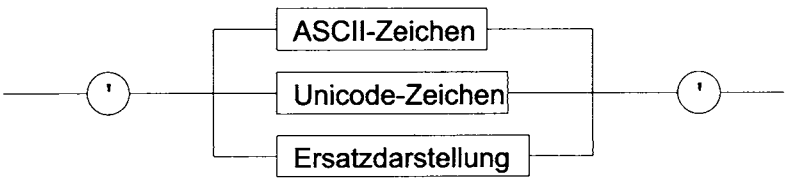
Üblicherweise verwendet man beim Programmieren immer den 7-Bit-ASCII-Code (ASCII = American Standard Code for Information Interchange). Lediglich in Kommentaren und Texten dürfen auch die erweiterten 8-Bit-Codes verwendet werden. Diese 8-Bit-Codes sind aber systemabhängig; so stellen die Buchstaben mit dem Code 128 bis 255 z. B. unter DOS andere Zeichen dar als unter Windows.
Für eine plattformunabhängige Programmiersprache ist dieser Zustand unhaltbar. Aus diesem Grund hat sich ein Komitee zusammengesetzt, um einen allgemeingültigen Zeichencode zu definieren, der neben Sonderzeichen auch nationale Zeichensätze wie Chinesisch, Japanisch und Kyrillisch normiert. Das Ergebnis ist der Unicode, ein 16-Bit-Code [Uni00]. Die ersten 128 Zeichen des Unicodes sind die üblichen 7-Bit-ASCII-Zeichen.
Java unterstützt den Unicode in vollem Umfang, d. h. Bezeichner und Texte können den Unicode-Zeichensatz verwenden.

Typ	Bit	Wertebereich
`char`	16	`'\u0000'...'\uffff'(positiv)`

Die Schreibweise `'\uxxx'` beschreibt ein Zeichen im Unicode mit dem Hexadezimalwert "xxxx", wobei jedes "x" eine Hexadezimalziffer ist. Einen ASCII-Wert kann man einfacher schreiben, indem man das Zeichen direkt in Apostrophs (') geklammert angibt.

C++ Anders als in C und C++ sind die Werte vom Typ `char` immer positiv, sie werden als 16 Bitwerte ohne Vorzeichen behandelt.

Zeichen : (4-10)



Um auch Steuerzeichen einfacher darstellen zu können, kennt Java spezielle Ersatzdarstellungen, die alle mit dem Backslash \ beginnen und in der folgenden Tabelle zusammengestellt sind.

Ersatzdarstellung	Zei.	Wert	Erklärung
`\b`	BS	`\u0008`	Zeichen zurück (backspace)
`\t`	HT	`\u0009`	horizontaler Tabulator
`\n`	LF	`\u000a`	Zeilentrenner (linefeed)
`\f`	FF	`\u000c`	neue Seite (form feed)
`\r`	CR	`\u000d`	Wagenrücklauf (carriage return)
`\"`	"	`\u0022`	Anführungszeichen
`\'`	'	`\u0027`	Apostroph
`\\`	\	`\u005c`	Backslash

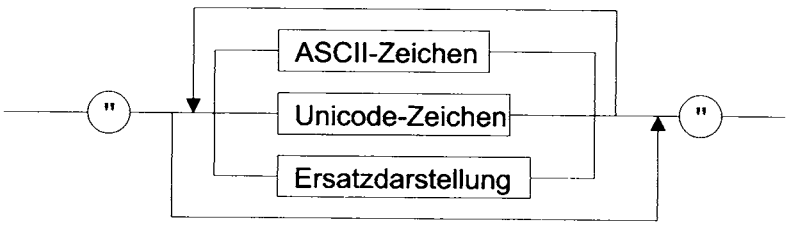
Ersatzdarstellung	Zei.	Wert	Erklärung
\xxx			Zeichen mit dem oktalen ASCII-Wert xxx (0000 xxx 0377)
\uxxx			Unicode-Zeichen mit der hexadezimalen Codierung "xxxx" (0x0000 xxxx 0xffff)

❑ *Beispiel 4.4.1 Beispiele für Zeichen*
Beispiele von **char**-Konstanten, wobei die drei letzten dasselbe Zeichen darstellen:

```
'A', '\n', 'ä',  '\u0084' , '\204'
```
■

In den früheren Programmbeispielen wurden schon mehrfach konstante Texte auf dem Bildschirm ausgegeben. Solche Zeichenreihen sind dabei nichts anderes als Folgen von Zeichenkonstanten, die man mit Anführungszeichen (") klammert. Zeichenreihen dürfen auch Ersatzdarstellungen aus der obigen Tabelle enthalten. Ferner können sie mit dem Operator + aneinandergehängt werden.

Zeichenreihe : (4-11)



❑ *Beispiel 4.4.2 Testausgabe auf Bildschirm*
Die Anweisung

```
System.out.print("Hallo Leute,\nwie "+
                 "geht es Euch?\n");
```

liefert die folgende Ausgabe auf dem Bildschirm:

```
Hallo Leute,
```

```
wie geht es Euch?
```

Man beachte, dass die Zeilenaufteilung in der Anweisung nichts damit zu tun hat, wie die Zeichenreihe ausgegeben wird. ■

C++ Zeichenreihen werden in Java nicht NUL-terminiert. Die Verwaltung der Länge wird in der Klasse **String** erledigt, die in Abschnitt 10.1 behandelt werden.

4.5 Eine einfache Klasse

Eine Klasse ist eine Programmeinheit, die eine ganz bestimmte Aufgabe erledigt. Sie besitzt Attribute (Variablen) und Methoden, die auf diesen Attributen operieren. In diesem Abschnitt wird eine einfache Klasse vorgestellt, die nur eine spezielle Methode **main** – das Hauptprogramm – enthält und mit der wir schon einfache Java-Programme schreiben können.
Diese einfache Klasse ist folgendermaßen aufgebaut:

```
public class einfacheKlasse
{   // Attributdeklarationen
    public static void main(String[] argv)
    {   // lokale Variablendeklarationen
        // Anweisungen
    }
}
```

Der in der ersten Zeile hinter **class** angegebene Name ist ein frei wählbarer Bezeichner (siehe Syntaxdiagramm (4-17)), der die Aufgabe der Klasse beschreibt. Das Java-Programm wird in einer Datei abgespeichert, die denselben Namen wie die Klasse und die Dateierweiterung "java" hat. Die obige Klasse steht also in der Datei **einfacheKlasse.java**. In Kapitel 7 werden Klassen vollständig behandelt.

4.6 Variablen und Variablendeklarationen

In jeder höheren Programmiersprache gibt es den Begriff der Variablen, mit der

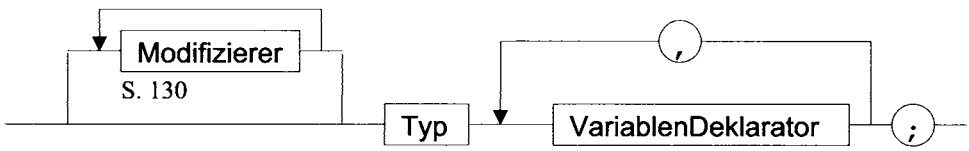
ein bestimmter Speicherbereich benannt wird. Eine Variable besteht aus den folgenden drei Angaben:

Name	`radius`
Datentyp	`float`
Wert	`17.5E3`

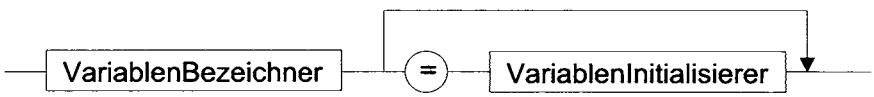
Der *Name* ist ein Bezeichner (siehe Syntaxdiagramm (4-17)) und identifiziert die Speicheradresse. Für verschiedene Variablen müssen verschiedene Namen gewählt werden.
Der *Datentyp* legt fest, in welchem Wertebereich der Wert der Variablen liegen muss. Intern wird durch den Datentyp auch noch festgelegt, wieviel Speicherplatz die Variable beansprucht und wie die interne Darstellung des Wertes aussieht. Im obigen Beispiel ist die Variable namens **`radius`** vom Typ **`float`**, d. h. sie belegt einen Speicherplatz von 32 Bit = 4 Byte und wird im IEEE-Format abgespeichert.
Der *Wert* der Variablen ist der momentane Inhalt des Speicherbereichs.

Variablen müssen vor ihrer Verwendung deklariert werden. Einige Alternativen der folgenden Syntaxdiagramme werden erst später besprochen. In den Syntaxdiagrammen ist dann ein Seitenverweis angegeben.

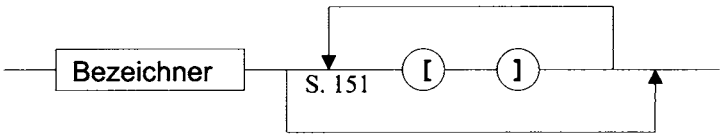
AttributDeklaration : (4-12)



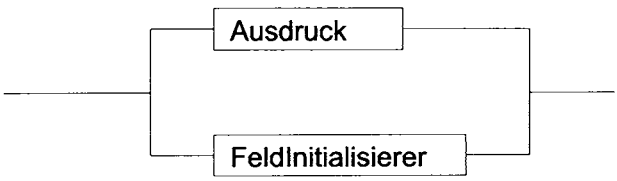
VariablenDeklarator : (4-13)



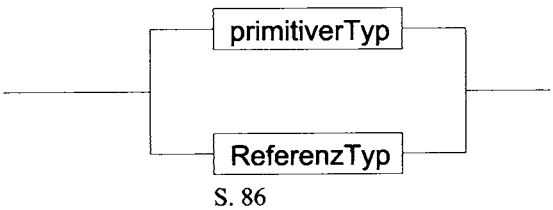
VariablenBezeichner : (4-14)



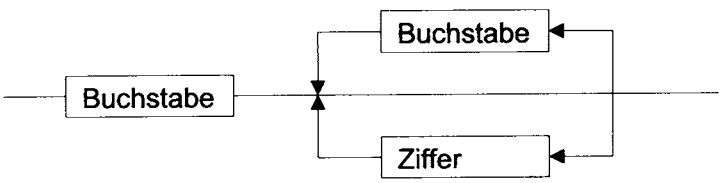
VariablenInitialisierer : (4-15)



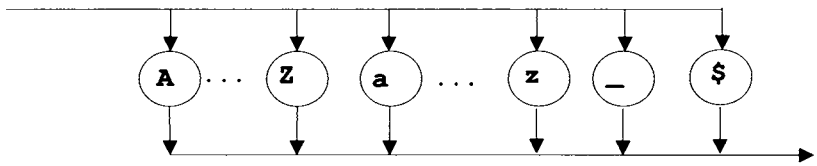
Typ : (4-16)



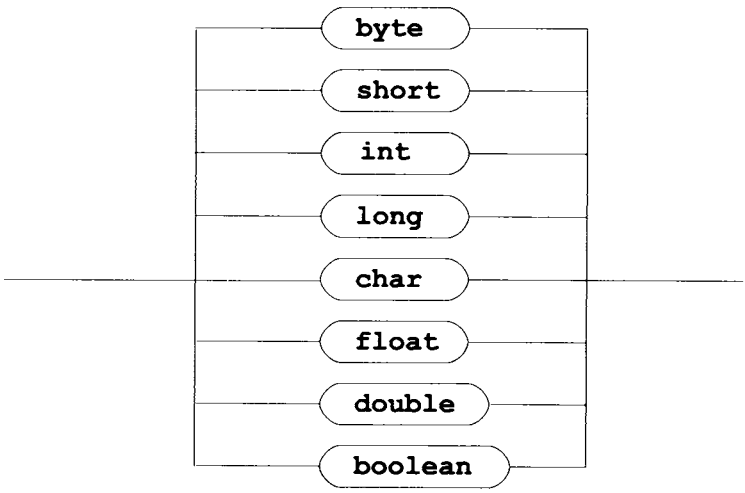
Bezeichner : (4-17)



Buchstabe : (4-18)



primitiverTyp : (4-19)



Variablen können initialisiert werden. Generell ist ein Initialisierer ein Ausdruck; wir verwenden vorerst nur Konstanten.

❑ *Beispiel 4.6.1 Einfache Java-Klasse*

```
public class einfacheKlasse
{   public static void main(String[] argv)
    {  float fl=12.34e5F;
       long lgl=4321567;
       char Ch='J';
       String s=
              "Moechten Sie die Zahlenwerte sehen? ";
       System.out.println(s);
```

```
        System.out.println(
              "Die Antwort ist (J/N) : "+Ch);
        System.out.println("f1 = "+f1);
        System.out.println("lg1= "+lg1);
    }
}
```

Die Methode **System.out.println** gibt Texte und Varaiablenwerte auf dem Bildschirm aus, gefolgt von einem Zeilenwechsel. Das Programm zeigt folgende Ausgabe:

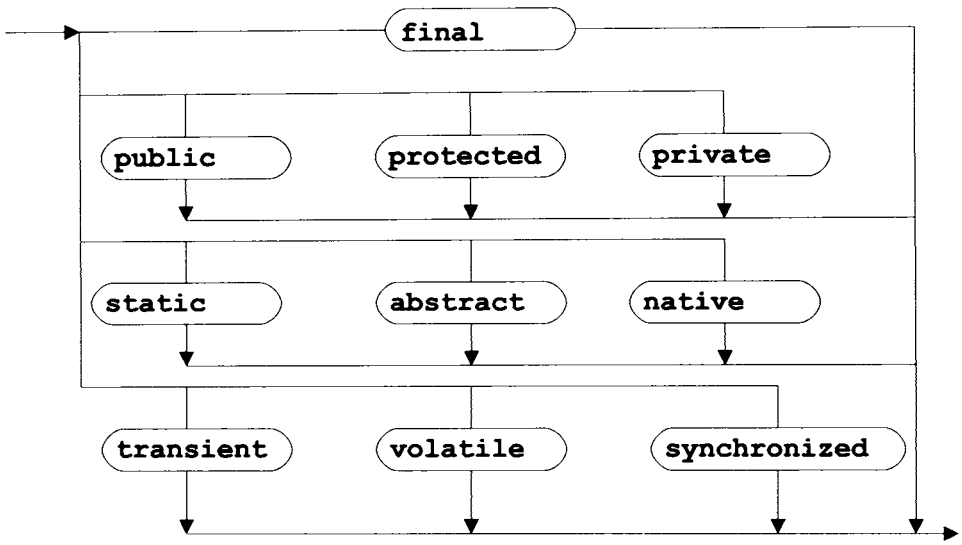
```
Moechten Sie die Zahlenwerte sehen?
Die Antwort ist (J/N) : J
f1 = 1234000.0
lg1= 4321567
```

■

4.7 Benannte Konstanten

Häufig verwendete Konstanten kann man benennen und dann im Programm diesen Bezeichner verwenden. Dies erhöht die Lesbarkeit und die Flexibilität des Programms. Zur Definition von benannten Konstanten benutzt man den Modifizierer **final**. Das folgende Syntaxdiagramm stellt alle in Java vorkommenden Modifizierer zusammen.

Modifizierer : (4-20)



Die Konstante **PI** ( ) kann man wie folgt definieren:

```
final float PI = 3.141592654f;
```

Der Bezeichner **PI** hat dann den angegebenen Wert, der fest mit dem Namen **PI** verbunden ist, d. h. der Wert von **PI** kann im Programm nicht verändert werden.

C++ In Java muss bei Konstanten immer der Datentyp angegeben werden; es gibt nicht die Abkürzung von C/C++, dass der Datentyp **int** ist, wenn man nichts angibt.

5 Ausdrücke

Ein Ausdruck besteht aus Operanden, Operatoren und Klammern, die zusammen eine Berechnungsvorschrift beschreiben. Operanden können die im vorigen Kapitel beschriebenen Variablen und Konstanten sowie Methodenaufrufe sein, die in Kapitel 7 behandelt werden. Bei den Operatoren gibt es verschiedene Stelligkeiten:

Unäre Operatoren wirken auf *einen* Operanden (meist den rechts stehenden).

Binäre Operatoren stehen zwischen ihren beiden Operanden.

5.1 Die Priorität von Operatoren

Kommen in einem Ausdruck mehrere Operatoren vor, muss vereinbart werden, in welcher Reihenfolge sie ausgeführt werden. Dies wird durch die Priorität festgelegt, die jedem Operator zugeordnet wird. Das Standardbeispiel für eine solche Festlegung ist die Regel "Punkt vor Strich", die wir schon in der Schule gelernt haben und die besagt, dass die Punktoperationen Multiplikation und Division vor den Strichoperationen Addition und Subtraktion ausgeführt werden. Operatoren gleicher Priorität können entweder von rechts nach links oder in der anderen Richtung ausgeführt werden.

In der folgenden Tabelle werden alle in Java verfügbaren Operatoren mit ihrer Priorität und Auswertungsrichtung zusammengestellt. Die Tabelle enthält neben der Priorität (Pri) die Stelligkeit (Stl.) und die Auswertungsrichtung (Rtg.) der Operatoren mit der Bedeutung *von rechts nach links* und *von links nach rechts*. In der letzten Spalte (S.) ist die Nummer der Seite angegeben, auf der der Operator besprochen wird.

Pri	Operator	Stl.	Zeichen	Rtg.	S.
13	logische Negation	unär	`!`		41
	Bitkomplement	unär	`~`		41
	positives Vorzeichen	unär	`+`		36
	negatives Vorzeichen	unär	`-`		36
	Inkrement	unär	`++`		52
	Dekrement	unär	`--`		52
	cast-Operator	unär	`(typ)`		46
12	Multiplikation	binär	`*`		36
	Division	binär	`/`		36
	Rest	binär	`%`		36
11	Addition	binär	`+`		36
	Subtraktion	binär	`-`		36
10	Linksschieben	binär	`<<`		39
	Rechtsschieben mit Vorzei.	binär	`>>`		39
	Rechtsschieben mit Nullen	binär	`>>>`		39
9	Vergleiche	binär	`<,>,<=,>=`		45
	Typvergleich	binär	`instanceof`		127
8	Gleichheit	binär	`==`		45
	Ungleichheit	binär	`!=`		45
7	bitweise UND	binär	`&`		41
	logisch UND	binär			41
6	bitweise exklusiv ODER	binär	`^`		41
	logisch exklusiv ODER	binär			41
5	bitweise ODER	binär	`|`		41
	logisch ODER	binär			41
4	bedingt UND	binär	`&&`		41
3	bedingt ODER	binär	`||`		41
2	bedingter Ausdruck	ternär	`? :`		46
1	Zuweisungen	binär	`=`		49
	`op { *,%,/,+,-, &,^,|,<<,>>,>>>}`	binär	`op=`		51

Zusätzlich gibt es die Klammern `( )`, mit denen man die Standardprioritäten der Operatoren durchbrechen kann. Es gilt hier: Zuerst wird der Ausdruck innerhalb eines Klammerpaares berechnet – beginnend mit dem innersten.

5.2 Interne Typkonvertierung

Die meisten binären Operatoren verlangen als linken und rechten Operanden Werte desselben Typs. In einigen Fällen leuchtet diese Einschränkung ein: So macht es wohl wenig Sinn, die Zeichenreihe "Hallo Leute" um 1 zu erhöhen. Anders sieht es bei der reellen Zahl 12.34 aus, die man sicherlich ohne große Schwierigkeiten um den ganzzahligen Wert 1 erhöhen könnte. In Java ist dies auch möglich, wobei der Compiler intern zuerst eine Typkonvertierung von der ganzen Zahl 1 in die reelle Zahl 1.0 ausführt und dann die beiden reellen Zahlen 12.34 und 1.0 addiert. Wenn man weiss, dass z. B. auf den PC's reelle Zahlen von einem anderen Prozessor bearbeitet werden als ganzzahlige Werte, ist diese etwas sture Forderung nach Operanden gleichen Typs durchaus verständlich. Der Java-Compiler führt bei Operanden unterschiedlicher Datentypen intern eine Typkonvertierung durch, bei der die Operanden auf den Datentyp des kompliziertesten Operanden und bei ganzzahligen Operanden mindestens auf den Typ `int` konvertiert werden. Eine solche Konvertierung "nach oben" ist immer ohne Informationsverlust möglich. Bei der Konvertierung eines großen ganzzahligen Wertes in einen reellen kann die Genauigkeit der Zahl Einbußen erleiden, da die Stellenanzahl bei reellen Zahlen nicht so groß ist wie z. B. bei `long`. Beim Erweitern "nach oben" wird ein `int`-Wert links mit dem Vorzeichenbit erweitert (wegen der Zweierkomplement-Darstellung negativer Zahlen), ein `char`-Wert wird dagegen links immer mit Nullen aufgefüllt, da `char`-Werte vorzeichenlos sind.
In den folgenden Abschnitten werden wir für die einzelnen Operatoren angeben, von welchem Datentyp die Operanden sein dürfen und welchen Datentyp dann das Ergebnis hat. Es wird dabei folgende Sprechweise benutzt:

Mit *ganz* bezeichnen wir die Datentypen `long`, `int`, `short`, `byte` und `char`. Wegen der oben angegebenen Typkonvertierung ist dann der Ergbnistyp `long`, wenn mindestens ein Operand vom Typ `long` ist; sonst ist der Ergebnistyp immer `int`.
Mit *reell* bezeichnen wir die Datentypen `float` und `double`.
Referenzen werden wir in Kapitel 7 besprechen.

Es gibt eine Vielzahl von Möglichkeiten, die einzelnen Ausdrücke zu verschachteln – man denke nur an geklammerte Ausdrücke. Da sich diese Verschachte-

lung auch durch sämtliche Syntaxdiagramme zieht, werden diese am Schluss dieses Kapitels in Abschnitt 5.10 zusammengefasst.

C++ Die interne Typkonvertierung in Java ist strikter als die von C/C++, wo auch implizit von einem größeren in einen kleineren Zahlentyp gewandelt wird.

5.3 Arithmetische Operatoren

Zu den vier Grundrechenarten **+**, **-**, ***** und **/** kommt in Java noch die Operation **%** (modulo oder Rest) hinzu, die – anders als in C und C++ – außer für ganzzahlige auch für reelle Operanden definiert ist und den Rest bei der Division liefert. Ferner kommen die Operatoren **+** und **-** auch als Vorzeichen sowie der Operator **+** als Konkatenation (Aneinanderhängen) von Strings vor.

Pri	Operator	Operanden	Ergebnis	Bemerkung
13	**Vorzeichen +**	ganz reell	**int,long** reell	positives Vorzeichen
11	**+**	ganz,ganz reell,reell String,String	**int,long** reell String	Summe Summe Konkatenation
13	**Vorzeichen -**	ganz reell	**int,long** reell	negatives Vorzeichen
11	**-**	ganz,ganz reell,reell	**int,long** reell	Differenz Differenz
12	*****	ganz,ganz reell,reell	**int,long** reell	Produkt Produkt
12	**/**	ganz,ganz reell,reell	**int,long** reell	siehe Text Quotient
12	**%**	ganz,ganz reell,reell	**int,long** reell	ganzzahliger Rest reeller Rest

Bei ganzzahligen Operanden liefert der Operator **/** den ganzzahligen Quotienten; das Ergebnis gibt also an, wie oft der Nenner in den Zähler passt. Bei der ganzzahligen Division bleibt i. A. ein Rest, den man mit dem Modulo-Operator **%** berechnen kann. Das Ergebnis des Modulo-Operators hat das Vorzeichen des

Zählers. Der Rest bei reeller Division ist folgendermaßen festgelegt: **`Rest = Zähler - Anzahl*Nenner`**, wobei **`Anzahl`** ganzzahlig und **`Rest<Nenner`** ist.
Bei der Division **`/`** und der Modulo-Operation **`%`** darf der zweite Operand nicht Null sein. Falls dies der Fall ist, wird bei ganzzahligen Operanden der Programmlauf mit einer entsprechenden Ausnahme abgebrochen; bei reellen Operanden ist das Ergebnis unendlich (**`Infinity`**). Mehr zur Ausnahmebehandlung erfahren Sie in Kapitel 12.
Kommen ganze und reelle Operanden gemischt vor, wird zunächst der ganze Operand nach reell gewandelt und dann der Operator für zwei reelle Operanden angewendet.

❑ *Beispiel 5.3.1 Arithmetische Operatoren*
Im folgenden Programm werden unterschiedliche Operationen ausgeführt, auch wenn das gleiche Operatorzeichen verwendet wird.

```
public class arithmetischeOperatoren
{  public static void main(String[] argv)
   {  float quotient,z=12.34E5f,q=-3e1f,rest,wert;
      int zaehler=13,ganz;
      System.out.println(zaehler + " / 4 = "
            + zaehler/4 + "  Rest = "+ zaehler%4);
      quotient=zaehler/4;
      System.out.println("Reeller Quotient = "
            + quotient + " ist ganzzahlig");
      quotient=zaehler/4.0F;
      System.out.println("Jetzt ist das Ergebnis "
            +"wirklich reell : " + quotient);
      rest=z%q;
      System.out.println("Rest fuer reelles modulo: "
            +rest);
   }
}
```

Das Programm liefert die folgende Ausgabe:

```
13 / 4 = 3  Rest = 1
Reeller Quotient = 3.0 ist ganzzahlig
Jetzt ist das Ergebnis wirklich reell : 3.25
Rest fuer reelles modulo: 10.0
```

■