Nr. 19

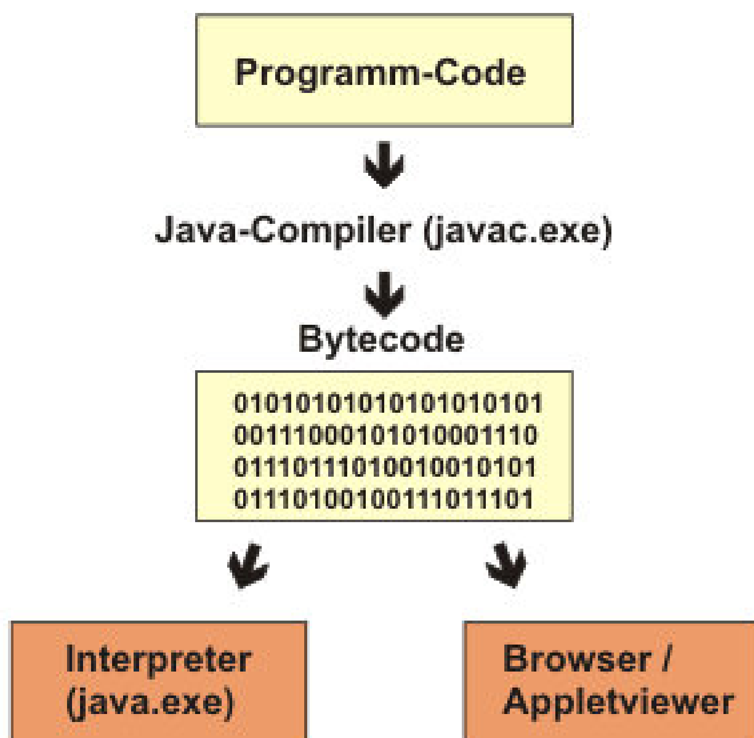
KnowWare PLUS

€ 4,-

Beispiele und Übungen

Java2

für Einsteiger



www.KnowWare.de

Dirk Ammelburger

Deutschland: 4,- EUR   Österreich: 4,60 EUR
Schweiz: 8 SFR   Luxemburg: 4,70 EUR   Italien: 5,50 EUR

Java für Einsteiger

Dirk Ammelburger, dirk@ammelburger.de

ISBN 87-90785-74-6, 1. Ausgabe, 1. Auflage: 2001-11
© Copyright 2001, Autor und KnowWare

Michael Maardt, verlag@knowware.de - Karl Antz, lektorat@knowware.de

Printer: OTM Denmark, Binder: Gramo Denmark, Published by KnowWare

Nachbestellung für Endverbraucher und Vertrieb für den Buchhandel

Bonner Presse Vertrieb
Möserstr. 2-3
D-49074 Osnabrück
Tel.: +49 (0)541 33145-20
Fax: +49 (0)541 33145-33
knowware@bpv-online.de
Ein Bestellformular findest du online hier:
www.knowware.de

Vertrieb für den Zeitschriftenhandel:

IPV Inland Presse Vertrieb GmbH
Postfach 10 32 46
D-20022 Hamburg
Tel.: (040) 23711-0
Fax: (040) 23711-215

Worum es geht

Hinter **KnowWare** steht der Gedanke, Wissen leichtverständlich zu vermitteln. Das Projekt startete im April 1993 mit der Herausgabe des ersten Computerheftes in Dänemark. Seitdem sind in vielen Ländern zahlreiche weitere Hefte mit Themen rund um den Computer erschienen.

www.knowware.de

Auf unserer Homepage findest du Beschreibungen und Bilder aller Hefte, geplante Hefte, Online-Bestellung, Anmeldung für einen kostenlosen Newsletter, Tipps & Tricks, Informationen über Sonderdruck für Firmen, neue Autoren, KnowWare in anderen Ländern, Autorenberatung, Händlerlisten usw.

Kostenlose Download

Auf unserer Homepage kannst du kostenlos einige Seiten aus jedem Heft im PDF Format downloaden. Ausverkaufte Hefte: das ganze Heft als PDF ist kostenlos.

Wo und wann sind die Hefte erhältlich?

Die Hefte sind im allgemeinen zwei Monate im Handel, und zwar bei Kiosken, im Bahnhofsbuchhandel und im Buchhandel – bei vielen Verkaufsstellen sowie im Buchhandel auch länger. Alle beim Verlag vorrätigen Titel sind jederzeit nachbestellbar.

Nachbestellung

Es gibt 2 Möglichkeiten:

- bei deinem KnowWarehändler - Bestellformular am Ende des Heftes ausfüllen!
- beim Bonner Presse Vertrieb, siehe links

www.knowware.de

Java für Einsteiger 4
Die Story 4
Was ist Java? 4
Wie funktioniert Java? 5
Technische Voraussetzungen 5
Das JDK 6
Die JDK-Entwicklungswerkzeuge 7
Das erste Programm 7
Kurz erklärt... 8
Java: Die Grundlagen 9
Java und C++ im Vergleich 9
Applets und Applikationen 9
Objektorientierte Programmierung 9
Sprache und Syntax von Java 12
Programmstruktur und Kommentare 12
Datentypen 13
Anweisungen 13
Schlüsselworte 14
OOP in Java 14
Klassen 14
Pakete und Verzeichnisstrukturen in Java – Teil 1 17
Objekte 18
Methoden 19
Objektvariablen direkt zuweisen und abrufen 20
Die Methode static main() 20
Klassenvariablen und -methoden 21
Konstruktoren und Destruktoren 23
Konstruktor mit Parametern 25
Mehrere Konstruktoren 26
Methoden überladen 27
Pakete und Verzeichnisstrukturen in Java - Teil 2 29
Sichtbarkeit und Zugriffsrechte 29
Beispiel zur Sichtbarkeit 30
Elementare Syntax von Java 32
Operatoren 32
Bedingungen und Entscheidungen 33
Schleifen in Java 37
Komplexe Datentypen 39
Referenzdatentypen 42
Vererbung 43
Arbeiten mit Java 48
Die Anweisung import 48
Das Paket java.lang 48
Das Paket java.util 51
Multitasking mit Threads 54
Applets und das Internet 58
Was ist ein Applet? 58
Applets in eine Webseite einbinden 58
Das erste Java Applet 59
Das Packet java.applet 60
Die Methoden eines Applets 60
Das Paket java.awt 62
Mehr Multimedia 68
Eventhandling 69
Applets und Animationen 74
Parameter an ein Applet 78
Das Sandkastenprinzip 78
Java im Browser 78
Applets in HTML 4.0 79
Was zum Teufel heißt deprecated?! 80
Weitere wichtige Pakete 80
Wichtige Seiten im Netz 81

Java für Einsteiger

In diesem Heft über Java wirst du alles lernen, was nötig ist, um erfolgreich mit der Sprache Java zu arbeiten. Auch wenn sich das Heft an Einsteiger richtet, empfehle ich zum besseren Verständnis dieses Heftes minimale Vorkenntnisse im Bereich der Programmierung. Das Erlernen von Java ist selbstverständlich auch ohne Programmiererfahrungen möglich – allerdings ist der Einstieg um einiges leichter, wenn man ein paar Vorkenntnisse mitbringt.

Alle vorgestellten Programme sind im Internet unter `www.kulturbrand.de` als Download bereitgestellt. Also viel Spaß und Erfolg...

Die Story

Java ist in aller Munde, und das liegt nicht nur daran, dass Java eine Kaffeesorte ist. Ganz im Gegenteil: Kaffee mag für Programmierer ein Grundnahrungsmittel sein, aber die wahre Euphorie für diese Sprache hat tiefere Gründe.

Im Jahre 1991 wurde in der Firma Sun Microsystems unter der Führung von James Gosling und Billy Joy das „Green Project“ ins Leben gerufen. Das Ziel war, eine Plattform zu schaffen, um kleine portable Programme zu erstellen, die sich über TV-Leitungen laden ließen, um Fernsehen interaktiver zu gestalten. Als Grundlage wurde ein objektorientiertes System geschaffen, das nicht auf ein Betriebssystem oder bestimmte Hardware festgelegt war.

Leider wuchs der Markt für diese Art von Software nicht so schnell, wie angenommen wurde; also wurde das gesamte Projekt umgekrempelt. Der Systemprototyp OAK (Object Application Kernel) wurde die Grundlage für die erste Java-Version, die im Jahre 1995 als Version 1.0 auf dem Markt erschien.

Java ist also eine sehr junge Sprache. Dank ihrer Fähigkeit, unabhängig von bestehenden Systemen zu arbeiten, wurde bald klar, dass Sun damit eine wunderbare Sprache für die Entwicklung von Webapplikation geschaffen hatte –das Internet ist nun einmal ein riesiges heterogenes Netzwerk aus den verschiedensten Systemen. Mit Java war man von diesen Systemen unabhängig, und der Code konnte beliebig portiert werden.

Was ist Java?

Java gehört zu den ersten Sprachen, die den objektorientierten Ansatz konsequent umsetzen. Im Gegensatz zu prozedural arbeitenden Sprachen werden in Java alle Daten und Funktionen in sogenannten Objekten gespeichert, die die komplette Syntax von Java bestimmen. Wie das genau funktioniert, sehen wir uns später an.

Aber Java kann noch mehr! Die Objektoriertiertheit an sich ist inzwischen nicht mehr so ußergewöhnlich. Heute unterstützen mehrere Sprachen diese Art der Programmierung.

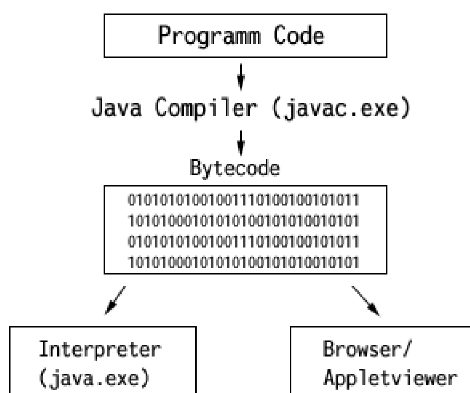
Das besondere an Java ist nach wie vor die schon erwähnte Plattformunabhängigkeit. Ein langes Wort – aber was bedeutet es eigentlich? Normalerweise wird jedes Programm für ein bestimmtes System geschrieben. Wenn du ein Programm z.B. in C schreibst, musst du schon vorher wissen, auf welchen System es laufen soll. Neben Besonderheiten, die du für bestimmte Algoritmen beachten musst, ist ein passender Compiler für das Zielsystem notwendig. Sind diese Vorrausetzungen nicht gegeben, kannst du für dieses System nicht entwickeln.

Mit Java ist das anders: Dank der Plattformunabhängigkeit kann man mit dieser Sprache völlig frei von irgendwelchen Einschränkungen durch das Betriebssystem programmieren. Das beste Beispiel sind Java-Applets. Diese kleinen Programme werden über das Internet direkt in den Browser des Anwenders übertragen und ausgeführt. Der Programmierer weiß zwar nicht, auf welchen Systemen (Apple, Power PC, Windows, UNIX/Linux, etc.) das Programm letztendlich laufen wird, aber das braucht ihn auch nicht zu kümmern. Java kapselt die Funktionalitäten der verschiedenen Systeme ein und nimmt dem Programmierer so die Arbeit einer Anpassung ab.

Wie funktioniert Java?

Leser, die schon etwas Erfahrung im Bereich der Programmierung gesammelt haben, wissen vielleicht, dass ein geschriebener Code vor der Ausführung kompiliert werden muss. Dabei wird der Quelltext von einem Übersetzungsprogramm oder *Compiler* in Maschinensprache übersetzt, die der Computer (speziell der Prozessor) lesen und ausführen kann. Dieses Vorgehen hat den Nachteil der Abhängigkeit von einem bestimmten System oder Prozessor, da der Code nach dem Übersetzen speziell auf eine Maschine zugeschnitten wird. Das Programm ist zwar schnell und unabhängig von weiterer Software auszuführen, allerdings funktioniert es nur in einem kleinen Bereich der Computerwelt.

Java unterscheidet sich in diesem Punkt von anderen Sprachen. Wenn du deinen Javacode kompilierst, wird der Quelltext nicht in die jeweilige Maschinensprache deines Systems übersetzt, sondern in einen sogenannten Bytecode. Dieser Bytecode ist unabhängig vom Prozessor des Systems und läuft auf jedem Rechner, der Java unterstützt. Voraussetzung ist allerdings ein Interpreter, der Java-Bytecode versteht. Die meisten Java-fähigen Browser verfügen über so einen Interpreter – einzelne Java-fähige Browser ohne Interpreter können auf den des jeweiligen Systems zurück greifen; so benutzt der Internet Explorer für das Mac-OS z.B. die Java-Umgebung des Mac-Systems MRJ (Macintosh Runtime for Java). In der Javaent-wicklungsumgebung wird dieser Interpreter mit dem Programm java.exe aufgerufen. Doch dazu später mehr.



Technische Voraussetzungen

Was brauche ich, um mit Java zu arbeiten? Als erstes natürlich die Entwicklungsumgebung für Java selbst, die kostenlos aus dem Internet gezogen werden kann. Die zur Zeit aktuelle Version des Java Development Kit ist das JDK 1.3. Das kann sich allerdings, je nach dem, wann du dieses Heft liest, schon geändert haben. Im Internet findest du die definitiv aktuelle Version unter `java.sun.com`.

> Um Missverständnissen vorzubeugen, möchte ich hier ein paar Worte zu den Versionen sagen. In diesem Heft geht es um die Programmiersprache Java2. Die Versionsnummer 1.3 bezieht sich auf die Version des JDK, das für die Arbeit mit Java benötigt wird. Es stellt die Werkzeuge zur Verfügung, die benötigt werden, um ein Javaprogramm zu kompilieren oder auszuführen. Die einzelnen Werkzeuge werden wir im nächsten Kapitel genau besprechen.

Als weiteres Tool benötigst du unbedingt einen ASCII-Editor, mit dem du deine Programme schreibst. Wenn du mit Windows arbeitest, kannst du den System-eigenen Notepad-Editor benutzen, der es ermöglicht, reine ASCII-Dateien zu erstellen.

> Bitte verwende keine Textverarbeitungen wie Word oder Starwrite, da solche Applikationen Steuerzeichen im Text hinterlassen. Dadurch wird der Code für den Compiler unbrauchbar.

Notepad findest du unter START|ZUBEHÖR.

Erfahrung hat mich allerdings gelehrt, dass es nicht immer einfach ist, den zunehmend komplexeren Sourcecode in einem einfachen Editor zu lesen. Dazu kommen sporadisch auftretende Probleme mit dem Zeilenumbruch oder plötzlich verschobener Formatierungen, wenn das Fenster verkleinert wird. Aus dem Grund möchte ich das Freeware Programm JOE (Java oriented editing) empfehlen, das speziell für Javaprogrammierung entwickelt wurde. Durch farbige Markierung der Syntax und einfacher Navigation bleibt der Überblick immer gewahrt.

Zusätzlich gibt es hier das angenehme Feature, den Javacompiler direkt in die IDE (Integrated

developing Enviroment) zu integrieren. So kann das Programm mit einem Knopfdruck kompiliert und ausgeführt werden.

Unter **www.javaeditor.de** findest du das Programm. Du installierst es per Doppelklick.

Damit du mit dem Programm bequem arbeiten kannst, musst du einige Einstellungen vornehmen. In den neueren Versionen werden diese Einstellungen automatisch gemacht – falls nicht, oder falls du sie ändern möchtest, gehst du so vor: Du wählst OPTIONEN| EINSTELLUNGEN. Hier kannst du den Compiler auf deine Bedürfnisse einrichten. Die wichtigsten Punkt sind gleich als erstes zu sehen: willst du die Programme aus der Entwicklungsumgebung (JOE) kompilieren, muss der Editor wissen, an welcher Stelle der Festplatte der Javacompiler liegt. Klicke auf DURCHSUCHEN und suche die Datei JAVAC.EXE, die im BIN-Verzeichnis des JDK liegt. Entsprechend findest du den Interpreter (JAVA.EXE) und den Appletviewer (APPLETVIEWER.EXE). Diese Einstellungen musst du mit OK bestätigen. Ist das geschafft, kannst du nun jedes Javaprogramm einfach per Knopfdruck kompilieren oder ausführen. Die Leiste mit den Buttons über dem Textfeld erspart dir in Zukunft (fast) jede Berührung mit der DOS-Eingabeaufforderung.

Falls bei der Programmausführung das DOS-Fenster sich sofort wieder schließt, muss der Menüpunkt PROGRAMM NACH AUSFÜHRUNG ANHALTEN aktiviert werden. Du findest ihn unter OPTIONEN im Bereich INTERPRETER.

Die nötigen Einstellungen innerhalb dieses Editors werden nur vorgenommen, wenn du das JDK schon installiert hast. Andernfalls wirst du diese Punkte per Hand eintragen müssen!

Für den Mac gilt prinzipiell das selbe wie für den PC. Wir brauchen einen Editor, der die Erstellung einfacher Textdateien erlaubt, ohne Anweisungen zur Formatierung im Quellcode zu hinterlassen. Da das neue Betriebssystem Mac OS X komplett auf Unix basiert, kann der Macianer aus einem riesigen Fundus von Programmen schöpfen. Es laufen sowohl Mac-Programme als auch Unix-Editoren.

Wer es ganz klassisch mag, kann sich mit dem UNIX-Editor *vi* versuchen, auch wenn das Programm nach heutigen Maßstäben nicht wirklich komfortabel ist.

Wer es einfacher mag, dem empfehle ich BBedit. Dieses Programm ist im Internet als Shareware verfügbar und bietet alle für unsere Programme benötigten Funktionen. Die Version BBedit lite ist sogar kostenlos zum Download freigegeben.

Die Adresse lautet **www.bbedit.com/**

Ansonsten lohnt es sicher immer, eine Suchmaschine zu bemühen und nach den Stichworten Editor und Mac zu suchen.

Schließlich benötigst du einen javafähigen Browser, um dir das Ergebnis deiner Javaapplets anzusehen. Empfehlenswert ist der Internet Explorer von Microsoft oder der Netscape Navigator. Wie das genau funktioniert, werden wir ab Seite 62 besprechen.

Das JDK

Bevor du Javaprogramme schreiben und kompilieren kannst, musst du das JDK auf deinen Rechner installieren. Das JDK enthält alle notwendigen Programme, um in Java programmieren zu können. Die jeweils aktuelle Fassung findest du wie gesagt im Internet unter **java.sun.com**.

Die Installation selbst ist schnell gemacht und passiert fast von alleine. Ein Doppelklick auf das Programm startet die Installationroutine. Alles weitere wird im Lauf der Installation erklärt. Ich empfehle sämtliche Komponenten des Istallationsfiles zu installieren. Besonders die Beispielprogramme können sich später als sehr nützlich erweisen.

Ist die Installation vollständig, sollte sich ein neues Verzeichnis auf deiner Festplatte finden, dass auf den Namen JDK gefolgt von der jeweiligen Version hört. In diesem Verzeichnis befinden sich ein paar Unterverzeichnisse, die je nach Version unterschiedlich sein können. Die wichtigsten zähle ich nachfolgend kurz auf:

BIN	Tools und Entwicklerwerkzeuge, die du Java benutzt.
DEMO	Beispiele für Javaprogrammierung, inclusive Sourcecode.
INCLUDE	Header-Dateien, die je nach Bedarf in Java-Programme eingebunden werden. Sie stellen Lösungen für Standardaufgaben der Programmierung bereit, etwa Ein- und Ausgabe am Bildschirm.
LIB	Bibliotheken zum Kompilieren.

Bevor du weitermachst, legst du am besten noch ein weiteres Verzeichnis an, das du „Programme" oder so ähnlich nennst. Hier werden dann unsere Arbeiten gespeichert. Als abschließenden Schritt musst du einen neuen öffentlichen Pfad im System eintragen, damit der Rechner die Werkzeuge des JDK findet. Du wählst START|AUSFÜHREN und schreibst **sysedit**, wodurch sich die Datei AUTOEXEC.BAT öffnet.

Hier fügst du folgende Zeile ein:

```
path=c:\jdk1.3\bin
```

Dann startest du deinen Rechner erneut. Jetzt sollte der Compiler überall verfügbar sein. Gibt es in der AUTOEXEC.BAT bereits eine **path**-Anweisung, trägst du den Pfad durch ein Semikolon (;) getrennt hinter dem letzten Eintrag ein. Neustarten nicht vergessen!

> Verwendest du den oben empfohlenen Editor JOE, ist diese Einstellung nicht notwendig!

Die JDK-Entwicklungswerkzeuge

Jetzt ist die Entwicklungsumgebung für Java komplett eingerichtet, und wir können uns etwas mit den einzelnen Werkzeugen auseinandersetzen. Zu diesem Zweck öffnest du das BIN-Verzeichnis unterhalb des JDK auf deiner Festplatte. Hier siehst du hoffentlich eine Reihe von Dateien, von denen wir die wichtigsten jetzt besprechen werden.

JAVA.EXE	Hinter diesem Programm verbirgt sich der Javainterpreter. Du benötigst es, um auf deinem Rechner Javaapplikationen auszuführen.
JAVAC.EXE	Der Javacompiler dient dazu, den Javasourcecode in den benötigten Bytecode zu übersetzen.
APPLET-VIEWER.EXE	Mit diesem kleinen Programm kannst du Javaapplets ausführen und anschauen. Alternativ zum Appletviewer kannst du auch einen javafähigen Browser verwenden.

Die drei Programme werden wir anfänglich benötigen, um erfolgreich in die Welt von Java einzusteigen. Falls es nötig sein sollte, werde ich im Laufe des Heftes weitere Werkzeuge des JDK vorstellen.

Das erste Programm

Damit nach der vielen Theorie etwas Praxis folgt, möchte ich das klassische Einsteigerprogramm aller Programmiersprachen vorstellen. Ziel ist, eine Begrüßung auf dem Bildschirm auszugeben.

Da weder die Welt noch sonst jemand je nett zu uns war, sparen wir uns den „Hallo Welt"-Quatsch und gehen gleich einen Schritt weiter. Wer die innovative Änderung an diesem Programm entdeckt, der darf sich stolz auf die Schulter klopfen.

Öffne deinen Editor und gib das nachfolgende Programm ein. Achte auf die Rechtschreibung des Programms, da Java zwischen großen und kleinen Buchstaben unterscheidet.

```
public class java1
{
public static void main(String argv[])
        {
        System.out.println ("Hallo Java!\n");
        }
}
```

Speichere das Programm unter dem Namen der Klassendefinition, wie du sie in der ersten Zeile siehst. Im Beispiel wäre das JAVA1. Die Endung der Datei muss .JAVA heißen, sonst findet der Compiler sie nicht. Bist du fertig, solltest du eine neue Datei JAVA1.JAVA im Programme-Ordner sehen.

Der Name ist willkürlich gewählt. Wichtig ist, dass die Klasse in der Datei, die die **main()**-Funktion enthält, den selben Namen hat. Später erfährst du mehr über diesen Punkt.

Soll das Programm ausgeführt werden, fehlen noch zwei Schritte. Zunächst muss es kompiliert werden, damit du den nötigen Bytecode erhältst. Da der Compiler ein DOS-Programm ist, das keine grafische Oberfläche besitzt, musst du über dein Startmenü eine DOS-Box öffnen. Darin wechselst du mit den Befehlen **cd..** und **cd** ***<Verzeichnisname>*** in dein Programmverzeichnis. Der Compiler wird mit dem Befehl **javac** gefolgt vom Dateinamen des Programms gestartet. Also gibst du ein:

```
javac java1.java <ENTER>
```

Hast du alles richtig gemacht, sollte einen Augenblick nichts passieren, dann beendet sich das Programm von selbst. Scheinbar passierte nichts – tatsächlich siehst du jetzt eine neue Datei namens JAVA1.CLASS in deinem Verzeichnis. Andernfalls gibt der der Compiler eine Fehlermeldung aus. Kontrolliere dein Programm noch einmal auf Fehler.

Benutzt du den empfohlenen Editor JOE und hast ihn korrekt eingebunden, reicht es, den Button fürs Kompilieren zu drücken und auf eine Erfolgsmeldung zu warten.

Die fertige CLASS-Datei enthält den benötigten Bytecode, um das Programm auszuführen. Dafür benötigst du den Interpreter JAVA.EXE, der ebenfalls im BIN-Verzeichnis liegt.

Gestartet wird er ähnlich wie der Compiler mit dem Aufruf **java** gefolgt vom Dateinamen der **class**-Datei:

```
java java1 <ENTER>
```

> Die class-Datei wird dem Interpreter ohne die .CLASS Endung übergeben. Im Gegensatz dazu muss dem Compiler immer ein Dateiname mit der .JAVA-Endung übergeben werden.

Bei JOE reicht es, das Programm mit einem Klick aus der Menüleiste heraus zu starten.

Das Ergebnis der Mühen? – die Begrüßung „Hallo Java!“. Erscheint eine Fehlermeldung oder Exception, liegt ein Fehler vor.

Kurz erklärt...

So einfach die eben gezeigte Aufgabe sein mag – der Sourcecode verwirrt zunächst. Ich kann mir vorstellen, dass erfahrene Programmierer etwas entmutigt auf den voluminösen Überbau des Programmes schauen, der wenig Sinn ergeben möchte. Zumindest ging es mir so, als ich mein ersten Javaprogramm schrieb. In prozeduralen Sprachen war das erste *Hallo Welt*-Programm in der Regel leicht zu durchschauen. Es lief auf einen einfachen Befehl hinaus, der die Ausgabe auf dem Bildschirm erlaubte, ummantelt von einer **main()**-Funktion oder etwas ähnlichem. In Java ist die Sache komplizierter, da die Sprache auf die komplette Einhaltung der Objekt-orientierten Syntax besteht. Was das ist, erfahren wir in den nächsten Kapiteln.

Grundsätzlich muss allerdings jetzt schon gelten, dass jedes Javaprogramm prinzipiell nur aus Klassen und deren Objekten bestehen darf. D.h. ein jedes Programm fängt mit einer Klassendefinition an, die den restlichen Sourcecode enthält. Gibt es mehrere Klassen in einem Programm, muss eine dieser Klassen als Startpunkt festgelegt werden. Das geschieht durch die Festlegung einer **main()**-Funktion (oder Methode, wie es korrekter heisst), die vom Interpreter als Einstiegspunkt gewählt wird. Konkret läuft das so ab, dass Java die übergebene Startklasse nach der **main()**–Funktion durchsucht und dann dort das Programm beginnt. Ähnlichkeiten zu C oder C++ sind in diesem Punkt nicht von der Hand zu weisen. Diese **main()**-Funktion wird dann Schritt für Schritt durchlaufen, und es werden alle Anweisungen ausgeführt, auf die der Compiler trifft. In unserem Fall ist das nur die Methode **println()**, die aus den übergeordneten Klassen **System.out** aufgerufen wurde. Sie ermöglicht die Ausgabe von Strings auf dem Bildschirm. Danach ist die **main()**-Funktion zu Ende, und das Programm beendet sich.

Java: Die Grundlagen

Ich bin zwar ein Freund praktischen Lernens anhand von Beispielen – dennoch werden wir an einem Stück Theorie nicht vorbeikommen. Ist dir das Prinzip der Objektorientierten Programmierung völlig unbekannt, verstehst su die Syntax und das gedankliche Prinzip hinter Java nicht ohne weiteres. Darum will ich etwas weiter ausholen und die Grundlagen schaffen, die wir für das weitere Voranschreiten in diesem Heft brauchen. Wer sich schon mit dieser Materie auseinandergesetzt hat, der kann die Kapitel schnell überfliegen und dann die Lektüre weiter hinten fortsetzen.

Für den Einstieg möchte ich auch gleich einige mögliche Missverständnissen ausräumen, die vielleicht später für Schwierigkeiten sorgen könnten. Also...

Java und C++ im Vergleich

Nach der Veröffentlichung des KnowWare-Heftes *C++ für Einsteiger* habe ich einige eMails mit der Frage bekommen, warum die kompilierten C++-Programme zwar unter Windows laufen, nicht aber auf UNIX/Linux oder dem Mac.

Die Frage habe ich im Prinzip schon im ersten Kapitel beantwortet. Der C++-Compiler schafft aus dem bestehenden Sourcecode eine Datei mit Maschinensprache, die direkt auf das System zugeschnitten ist. Dafür erhält man eine Applikation, die von weiterer Software wie z.B. Interpretern unabhängig ist. Da dieser Maschinencode sehr systemnah ist, lässt sich das entstandene Programm schnell interpretieren und ausführen. Der Nachteil ist Inkompatibilität zu anderen Systemen. Das Programm muss für jedes System kompiliert und angepasst werden.

Bei Javaprogrammen sind die Vorzeichen umgekehrt. Der Compiler schafft nicht etwa Maschinencode, sondern Bytecode, der auf jedem System gelesen werden kann. Dadurch sind Javaprogramme beliebig portierbar. Der „Preis“ für diese Freiheit ist schnell erklärt. Erstens ist dieser Bytecode von einem Interpreter abhängig, der den Code lesen und übersetzen kann. Der zweite Punkt ist die viel genannte Langsamkeit von Javaprogrammen in ihrer Laufzeit. Da der Interpreter den Bytecode bei jeder Ausführung erst in kompatible Maschinensprache übersetzen muss, geht viel Rechenzeit verloren, und ein Javaprogramm ist viel langsamer als eine eigenständige Applikation.

Beide Systeme haben also Vor- und Nachteile und je nach Situation Existensberechtigung. Ein Entwickler sollte nicht blind in irgendeiner Sprache programmieren, sondern vielmehr die möglichen Ansprüche an ein Programm prüfen und dann entscheiden auf welchem Weg er die Lösung entwickelt. Amen ☺

Applets und Applikationen

... zwei Schlagwörter, die oft genannt und gerne verwechselt werden. Ein paar Worte dazu, damit keine Missverständnisse auftreten!

Eine Applikation ist ein Programm, das auf einem Rechner ausgeführt werden kann. Die einfachste aller Applikationen haben wir eben selbst programmiert – sie gibt einen Satz auf dem Bildschirm aus. Eine Applikation ist ein „echtes“ Programm, das abgesehen vom Interpreter von einer Umgebung unabhängig ist.

Ein Applet dagegen ist in der Regel ein kleines grafisches Tool, das über das Internet geladen wird und in deinem Browser – und nur da! – erscheint. Ein Applet braucht eine bestimmte Umgebung, um zu funktionieren. Java wird gerne mit Applets gleichgesetzt, da die meisten Menschen Java nur in Form kleiner bunter Programme aus dem Internet kennen, aber Java kann viel mehr. Es ist verhältnismäßig einfach, ein Applet zu programmieren, da viele seiner Arbeitsschritte direkt vom Browser oder dem Betriebssystem übernommen werden.

Im Laufe des Heftes werden wir genauer auf dieses Thema eingehen. Um die Grundlagen kennenzulernen, programmieren wir zunächst nur Applikationen.

Objektorientierte Programmierung

Dieses Thema habe ich schon in anderen KnowWare-Heften besprochen. Da die OOP (Objektorientierte Programmierung) für Java elementar ist, werde ich das Thema hier etwas ausführlicher besprechen. Wer das Beispiel mit dem Dackel bereits kennt, wird hier wohl so manches Dejavue erleben ☺

Klassendefinition

Eine der großen Stärken des menschlichen Verstandes ist die Fähigkeit, Dinge zu typisieren und in Muster der Umwelt einzuordnen. Gehst du z.B. durch die Straßen und siehst einen Dackel, wirst du ihn als solchen erkennen und einordnen. Zugleich siehst du aber auch einen Hund, da der Dackel eindeutig in die Kategorie der Hunde eingeordnet wird. Gehen wir noch weiter: Hunde gehören der Gruppe Säugetiere an, die zur Gruppe der Warmblüter gehören. Diese sind Lebewesen, wie alle lebenden Geschöpfe. Du siehst nicht nur einen Dackel, sondern eine Reihe von höher geordneten Klassen von Lebewesen; und diese Kette von Begriffen definiert den Dackel in seiner Form.

Genauso machen wir es in Java: wir schaffen ein abstraktes Ebenbild der Wirklichkeit und erwecken es in einem Programm zum Leben.

Theoretisch ist ein Dackel eine Ansammlung von Eigenschaften und Fähigkeiten, die ihn mit seiner Außenwelt in Kontakt treten lassen.

Unser vereinfachtes Modell des Dackels wollen wir in einem Programm implementieren und mit Fähigkeiten und Eigenschaften ausstatten.

Die *Fähigkeiten* des Dackels werden durch *Funktionen* realisiert, seine *Eigenschaften* durch *Variablen*. So gibt es z.B. die Eigenschaften *Alter* und *Fellfarbe*, die Informationen über unser Modell sind. Fähigkeiten sind *bellen* und *schwanzwedeln*.

Du siehst –im Prinzip nichts neues. Was uns erlaubt, ein Modell der Realität zu schaffen, ist eine andere Sicht der Dinge. Im Grunde nehmen wir hier eine Zusammenfassung von Variablen und Funktionen zu einer Einheit vor, die man eine *Klasse* nennt. Die beschriebene Zusammenfassung heißt *Kapselung*.

So weit so gut – wir haben nun das Modell eines Dackels und können es bald einsetzen. Doch bisher ist dieser Dackel nur eine Vorlage in unseren Köpfen, die keine eigene Identität hat. Das ist auch gut so – denn unser Modell soll auch eines bleiben. Was wir haben, ist die Idee eines Dackels, wie er zu Tausenden über Wiesen rennt und Stöckchen holt. Dank dieses Vorbildes können wir eigene Dackel erstellen.

> Dieser Punkt ist wichtig: Erstellen wir eine Klasse, ist das eine Schablone, mit der wir später Abbilder erschaffen. Die Schablone ist leer und ohne Leben, jedes ihrer Abbilder aber kann eine eigenständige Existenz beginnen. Die Abbilder einer Klasse nennt man *Objekte*. Du kannst beliebig viele Objekte aus einer Klasse ziehen, ohne die Klasse dabei zu verändern.

Objekte

Da mehrere Objekte – oder Dackel – derselben Art nebeneinander existieren können, macht es Sinn, die typischen Merkmale und Charakterzüge in einer Typenbeschreibung zu speichern. Diesen Schritt besprachen wir im letzten Abschnitt: Durch die Typisierung gleichartiger Objekte erhalten wir eine Klasse, die als Vorlage für weitere Objekte dienen kann. Ein neues Objekt wird anhand einer bestehenden Klasse erschaffen. Dabei werden die in der Klassen definierten Methoden und Datenelemente an das Objekt übergeben, so dass ein in sich geschlossener Datensatz entsteht, der nur über definierte Schnittstellen oder Zugriffsmethoden mit der Außenwelt kommuniziert.

So kann sich ein Dackelobjekt, das wir hier Waldi nennen, nur über die Methoden `bellen()` oder `schwanzwedeln()` den Mitmenschen mitteilen. Gleichzeitig speichert das Objekt alle Daten in den von der Klasse definierten Datenelementen. So hat jeder Dackel sein eigenes Alter und Gewicht. Also sind Objekte unabhängige Abbilder einer Klasse.

Vererbung

Ein wesentlicher Punkt in der OOP ist die Vererbung. Das bedeutet innerhalb einer Programmiersprache, dass Klassen ihre Eigenschaften, also Methoden und Datenelemente, auf andere Klassen übertragen. So definiert man eine neue Klasse über einer bestehenden – sie wird aus ihrer Basisklasse abgeleitet. Die neue Klasse kann mit neuen Eigenschaften versehen werden, um eine Spezialisierung zu erreichen. Um das ein wenig klarer zu machen, bauen wir unser Dackelbeispiel weiter aus.

Betrachten wir dieses Tier einmal mit ganz neuen Augen: Was ist ein Dackel? Richtig, ein Hund! Und was ist ein Hund? Ein Säugetier. Ein Säugetier ist ein Lebewesen usw....

Wie du siehst, baut sich vor uns eine Kette von „....ist ein...“-Beziehungen auf, die vom speziellen ins allgemeine wandern.

Unter dieser Betrachtungsweise sind Zuordnungen grundsätzlich eindeutig – betrachtet man das Ganze aber aus einer anderen Richtung, sieht die Sache anders aus. Ein Hund muss nämlich nicht unbedingt ein Dackel sein, und ein Säugetier schon gar kein Hund. Erst die individuellen Änderungen an der Charakteristik des Hundes ermöglichen die Identifizierung eines Dackels. Das gilt für Säugetiere und Hunde sowie für alle Stufen dieser Treppe.

Wesentlich ist dabei aber, dass ein Hund alle Merkmale eines Säugetieres aufweisen muss, um in dieser Kette zu verbleiben.

Ein Hund kann beispielsweise bellen – und unterscheidet sich dank dieser Fähigkeit von Säugetieren im allgemeinen. Aber allein die Fähigkeit zu bellen macht noch keinen Hund aus. Neben einer Reihe spezieller Eigenschaften als Hund sind vor allem die Merkmale des Säugetiers wesentlich, damit ein Hund als solcher erkannt wird. Ein bellendes Tier, das Eier legt, würde kaum als Hund identifiziert.

Ein Hund erbt also alle Eigenschaften der Säugetiere und fügt neue Eigenschaften hinzu. Ein Dackel erbt alle Eigenschaften eines Hundes und fügt neue „Dackel-Elemente“ hinzu (kurze Beine, treudoofer Blick, etc.). Willst du nun eine spezielle Dackel Rasse „erschaffen“, muss diese Reihe weiter fortgesetzt werden. Ein Rauhaardackel unterscheidet sich z.B. durch spezielles Fell von seinen Artgenossen.

Zusammenfassend kann man also sagen: Eine neue Klasse erbt alle Eigenschaften einer bestehenden Klasse. Durch Veränderung einzelner Eigenschaften wird sie für spezielle Bedürfnisse angepasst.

Mehrfachvererbung

Unter dem Schlagwort Mehrfachvererbung versteht man die Möglichkeit, eine Klasse aus mehreren Basisklassen abzuleiten. So ist es möglich, die Funktionalität verschiedener „Mutterklassen“ in einer neuen Klasse zu vereinen und so auf alle Methoden und Datenelemente aus einer einzigen Klasse zuzugreifen. Dieses Vorgehen ist dann sinnvoll, wenn sich durch Kombination von Klassen neue, logisch sinnvolle Konstruktionen ergeben. In der Realität kommen solche Zusammensetzungen recht häufig vor, daher wird diese Technik von einigen Programmiersprachen auch in der Syntax umgesetzt.

Ein typisches Beispiel für die Mehrfachvererbung steht in den Garagen der meisten Leute. Ein Auto setzt sich im Laufe seines Produktionsprozesses aus den unterschiedlichsten Teilen zusammen, die erst im Zusammenspiel ein funktionierendes Auto ermöglichen. Unterteilt man das Auto in die Klassen Motor, Karosserie und Reifen, kann man jeder Klasse bestimmte Methoden (bremsen, rollen, beschleunigen, etc.) und Datenelemente (Farbe, Kubikzentimeter, Umfang, etc.) zuordnen. Die neue Klasse Auto würde in diesem Fall alle Eigenschaften der verschiedenen Klassen erben und in sich übernehmen. Die Klasse Auto hat also verschiedene Basisklassen. Wahrscheinlich würde man dieser Klasse weitere Eigenschaften wie Marke oder Preis hinzufügen, damit das Modell auch in der Wirklichkeit bestehen könnte.

Die Aufteilung einer Klasse in verschiedene Oberklassen ist sinnvoll, wenn man Teile der Ergebnisklasse auswechseln möchte. Es leuchtet ein, dass ein Auto mit verschiedenen Motortypen gebaut werden kann. Da nicht jedesmal die komplette Klasse neu zu schreiben ist, reicht es, die Oberklasse „Motor“ zu ändern.

Soviel zu Vorteilen der Mehrfachvererbung. Der Nachteil bei dieser Vorgehensweise ist rasch zunehmende Unübersichtlichkeit mit jeder weiteren Klasse, die in einem Programm auftaucht. Geht die Vererbungskette über viele verschiede Ebenen, wird der Sourcecode schnell zu einem nicht mehr durchschaubaren Chaos. Aus diesem Grund hat sich Sun entschieden, die Mehrfachvererbung in Java nicht zu unterstützen!

Wer jetzt enttäuscht aufstöhnt und fragt, warum er sich durch das Kapitel gequält hat, kann wieder aufatmen. Die Entwickler von Java haben ein Hintertürchen offengelassen, das Mehrfachvererbung indirekt zulässt. Ab Seite 49 werden wir diese Technik genau kennenlernen.

Sprache und Syntax von Java

Nachdem wir die Grundlagen von Java geklärt haben, können wir uns auf den praktischen Teil dieses Heftes stürzen. Die folgenden Kapitel beschreiben den Aufbau von Programmen und die technische Umsetzung der objektorientierten Theorie in der Praxis. Du wirst schnell feststellen, dass die OOP das Programmieren in vielerlei Hinsicht vereinfacht und besser strukturiert. In Java ist die Möglichkeit des schlampigen Programmierens weit geringer als in anderen Sprachen.

Programmstruktur und Kommentare

Die grundsätzliche Struktur eines Javaprogramms haben wir bereits in unserem ersten Beispiel kennen gelernt. Ein Javaprogramm muss aus mindestens einer Klassendefinition bestehen, damit der Compiler es akzeptiert. Wie diese Definition aufgebaut ist, erfahren wir ab Seite 18.

In der Praxis ist es allerdings so, dass ein Programm aus mehr als einer Klasse besteht. Während du das Programm schreibst, ist diese Feststellung nicht unbedingt wichtig, weil du die Klassen einfach untereinander schreiben kannst. Hast du das Programm allerdings erst kompiliert, wirst du feststellen, dass du für jede Klasse in deinem Projekt eine eigene CLASS-Datei in deinem Verzeichnis hast. Der Compiler legt also für jede Klasse eine eigene Datei an, die den selben Namen wie die Klasse trägt. Um das Programm starten zu können, musst du dem Interpreter die Datei übergeben, die die **main()**-Funktion enthält. Die übrigen Klassen werden automatisch vom Interpreter einbezogen.

Am Anfang des ersten Beispiels habe ich gesagt, dass die Arbeitsdatei den selben Namen bekommt wie die Klasse, in der die **main()**-Funktion liegt. Jetzt weißt du auch warum: Der Compiler benennt die Dateien automatisch nach ihren Klassen, damit er sie auseinanderhalten kann. Da die Sourcecode-Datei schon vorgegeben ist, übernimmt der Compiler den Namen dieser Datei. Ist dieser allerdings nicht gleich dem Klassennamen, würde der Rechner die Datei auf der Suche nach einer bestimmten Klasse nicht finden.

Kommentare

Zur besseren Strukturierung eines Programms können Kommentare in den Sourcecode eingebunden werden. Diese werden vom Compiler übersehen und können so dazu genutzt werden, bestimmte Bereiche des Codes zu erklären. Java unterstützt verschiedene Arten des Kommentierens.

Die am häufigsten verwendete Methode nutzt die Sonderzeichen Slash **/** und Sternchen *****, die den Kommentar umklammern.

```
Public class Dackel{
/* Dies ist ein Kommentar und wird
vom Compiler übersehen */
...
}
```

Mit dieser Technik lässt sich ein mehrzeiliger Kommentar erstellen. Der Compiler liest erst dann weiter, wenn er die „Abschlussklammer" gefunden hat. In einem Kommentar sind prinzipiell alle Zeichen erlaubt. Vorsichtig ist allerdings bei weiteren Kommentarzeichen geboten, wie z.B. dem Sternchen *. Der Grund liegt in der Definition eines Spezialkommentars innerhalb von Java, der für eine Art Dokumentation des Programms vorgesehen ist. Dafür ist das Programm JAVADOC.EXE zuständig, das ebenfalls im BIN-Verzeichnis zu finden ist. Das Programm erzeugt eine HTML-Dokumentation und liefert so eine Übersicht aller Programmteile. Der oben genannte Spezialkommentar ermöglicht die Dokumentation der HTML-Ausgabe.

Eine weitere Möglichkeit, Kommentare zu setzen, ist an die Syntax von C++ angelehnt. Durch doppelte Slash-Zeichen **//** kann ein einzeiliger Kommentar gesetzt werden. Dieser braucht kein Abschlusszeichen.

```
Public class Dackel{
//Das ist ein Kommentar!
...
}
```

Je nach Bedarf kann die eine oder andere Art verwendet werden. Technisch ergibt sich daraus kein Unterschied.

Datentypen

Schaut man im Handbuch nach, erfährt man, dass Java eine streng typisierte Sprache ist. Diese Feststellung bedeutet, dass Variablen, die in verschiedenen Formaten vorliegen (z.B. Ganzzahlen, Kommazahlen, Buchstaben, etc.) nicht miteinander verknüpft werden können. So ist es z.B. nicht möglich, Kommazahlen mit dem Datentyp für Buchstaben zu addieren – was ja auch durchaus logisch erscheint. Der Javacompiler kontrolliert bereits während des Kompilierens, ob im Programm eine Typen-verletzung vorliegt. Insgesamt wird zwischen drei verschiedenen Datentypen unterschieden: Standardtypen, Klassentypen und Feldtypen.

Wann immer du in einem Programm Daten speichern möchtest, musst du eine Variable erschaffen. Da der Compiler wissen muss, welche Art von Daten du speichern möchtest, bekommt jede neue Variable einen Typ zugewiesen, der diese genau definiert.

Standardtypen

Unter den Bereich der Standardtypen fallen alle die Typen, die der Compiler schon kennt. Beispiele für Standarddatentypen sind Ganzzahlen oder Kommazahlen. Java hat eine ganze Reihe von Standardtypen vorgegeben, die frei benutzt werden können. Im Gegensatz zu anderen Sprachen hat jeder Datentyp einen wohldefinierten Standardwert, der ihm bei seiner Definition automatisch zugewiesen wird. So kannst du immer sicher sein, dass dein Programm auch mit sinnvollen Werten arbeitet.

Typ	Inhalt	Standardwert	Größe	Wertebereich
`boolean`	True oder False	False	1 Bit	-
`char`	Unicode Zeichen	0000	16 Bit	0000 bis FFFF (Hex)
`byte`	Ganzzahl mit Vorzeichen	0	8 Bit	-256 bis 255
`short`	Ganzzahl mit Vorzeichen	0	16 Bit	-32768 bis 32767
`int`	Ganzzahl mit Vorzeichen	0	32 Bit	-2 ^31 bis 2^31-1
`long`	Ganzzahl mit Vorzeichen	0	64 Bit	-2^63 bis 2^63-1
`float`	Kommazahl	0.0	32 Bit	-
`double`	Kommazahl	0.0	64 Bit	-

Soll eine Variable eines bestimmten Typen deklariert werden, wird die Typenbezeichnung gefolgt vom Variablennamen aufgerufen.

```
boolean myBool;
int x = 32;
double y = 3.2;
char buchstabe;
```

Der Variablenbezeichner identifiziert die Vari-able innerhalb des Programms, damit man auf sie zugreifen kann. Wie die Beispiele zeigen, können Werte unmittelbar bei der Deklaration zugewiesen werden. In diesem Fall wird der definierte Standardwert einfach überschrieben.

Klassentypen

Durch die Implementierung neuer Klassen in dein Programm werden neue Typen definiert, die wir als Klassentypen bezeichnen. Dies ist in Java der einzige Weg, einen neuen Datentyp zu erschaffen. Die Werte (Variablen) dieser neuen Typen sind Objekte und können je nach Klassendefinition behandelt werden.

Feldtypen

Feldtypen sind in Java das, was in anderen Programmiersprachen Array oder auch Liste genannt wird. Diesem Datentyp werde ich auf Seite 42 ein eigenes Kapitel spendieren.

Strings

Strings nennt man Variable, die Zeichenketten, also Wörter oder Sätze, speichern können. Im Gegensatz zu vielen Programmiersprachen wird in Java zwischen Kostanten Strings und variablen Strings unterschieden. Mit dieser Materie werden wir uns ab Seite 53 befassen.

Anweisungen

Bevor wir die graue Theorie endgültig verlassen, ist oft erwähnte Begriff *Anweisung* zu klären. Ein Computerprogramm besteht in der Regel aus einer Reihe von Anweisungen, die dem Rechner sagen, was er tun soll. Diese Anweisungen werden je nach Sprache durch eine mehr oder weniger ausgeprägte Struktursyntax in eine feste Ordnung gebracht. Das Beispielprogramm aus dem vorhergehenden Kapitel bestand nur aus einer einzelnen Anweisung (`System.out.println`...), die die ganze Funktionalität des Programms ausmacht. Damit der Compiler diese Anweisung auch richtig interpretieren kann, ist sie von einer Funktionsdefinition umgeben, die wiederum Bestandteil einer Klasse ist. Dies ist ein sehr wesentlicher Punkt in Java. Anweisungen können im Gegensatz zu anderen Sprachen nicht für sich alleine stehen. Jeder Teil eines Programms muss grundsätzlich Teil einer übergeordneten Klasse sein. Selbst eine jede Klassendefinition ist wieder Teil einer höheren Klasse. Die oberste Klasse in Java, aus der sich alle weiteren Bestandteile ableiten, ist die Klasse Objekt.

Doch zurück zu den Anweisungen: Grob gesagt ist es so, dass immer dann, wenn der Computer zu einer bestimmten Aktion angewiesen wird, eine Anweisung nötig ist. Die Struktur in einem Programm bestimmt, wann welche Anweisung ausgeführt wird.
Die Syntax von Java verlangt, dass jegliche Anweisung mit einem Semikolon (`;`) abgeschlossen wird. Der Compiler weiß so, wann welche Anweisung zu Ende ist. In der Praxis könnte das so aussehen:

```
System.out.println („Eine
Anweisung!\n");
```

Der Rechner wird hier angewiesen, einen Buchstaben-String über die Standardausgabe auszugeben, sprich den Monitor.

Schlüsselworte

Es gibt in Java eine Reihe von Ausdrücken, die von Sun für die Sprache reserviert worden sind. Es ist nicht möglich, diese Worte innerhalb eines Programms als Namen für Variablen oder Klassen zu verwenden. Reservierte Worte sind zum Beispiel Funktionsnamen wie etwa `Println` oder vordefinierte Klassen wie `Object`. Im Internet findest du unter der Adresse `http://java.sun.com` die vollständige Liste der reservierten Wörter innerhalb der Java-API.

Als Programmierer solltest du die Benutzung reservierter Wörter unbedingt vermeiden, weil der Compiler sonst recht merkwürdige Fehlermeldungen generiert. So ist es z.B. nicht möglich, eine Methode `break()` zu nennen, weil dieser Ausdruck als Steueranweisung für Schleifen verwendet wird.

Einige Worte sind zwar von Sun reserviert, allerdings sind sie (noch) kein Teil der Syntax von Java. Ein typisches Beispiel ist `goto`: Der Ausdruck darf nicht verwendet werden – es gibt allerdings keine `goto`-Anweisung in Java. Vielleicht wird dieses Feature noch eingebaut, vielleicht wollte Sun auch nur verhindern, dass `goto` jemals in Java verwendet werden kann (Spagetticode). Die Zukunft wird es zeigen.

OOP in Java

Nachdem wir uns ausführlich mit der Theorie der OOP auseinandergesetzt haben und die Fachbegriffe genauerer spezifiziert wurden, können wir uns mit der praktischen Anwendung dieser Theorie in Java auseinandersetzen. Im folgenden werde ich stichpunkthaft die wichtigsten Merkmale der OOP in Java aufzeigen. In den nachfolgenden Kapiteln werden wir dann die besprochenen Konzepte anwenden.

- Java ist eine objektorientierte Sprache. Es gibt also keine Möglichkeit, globale Variablen oder Funktionen zu definieren.
- Alle Klassen haben die Oberklasse `Object`
- Java erlaubt das Überladen von Methoden
- Java unterstützt im Gegensatz zu C++ keine Mehrfachvererbung. Diese Einschränkung wird durch das Schnittstellenkonzept ausgeglichen
- Java unterstützt Polymorphismus

Mit dieser Grundlage sollte Java kein Problem mehr sein. Jetzt also ein wenig Praxis...

Klassen

Wie gesagt ist jede Klasse in Java eine Unterklasse des Klasse **`Object`**. Also hat jede Klasse in einem Javaprogramm eine Oberklasse.

Um eine neue Klasse zu erschaffen, bedarf es des Schlüsselwortes **`class`**, das den Kopf der Klassen bestimmt. Zusätzlich brauchen wir weitere Informationen, von denen manche optional sind. Die sicher wichtigste Information ist der Klassenname, über den die Klasse identifiziert wird. Dieser Bezeichner steht nach dem Schlüsselwort **`class`** und leitet damit die eigentliche Klassendeklaration ein.

```
class Dackel
{...}
```

Ein weiterer wichtiger Punkt ist die Sichtbarkeit einer Klasse innerhalb des Programms. Diese wird durch das Schlüsselwort **`public`** festgelegt, das seinen Platz am Anfang jeder Klassendefinition hat. Die Sichtbarkeit einer Klasse bestimmt, inwieweit andere Klassen auf die neue Klasse zugreifen können. Ist eine Klasse als **`public`** definiert, gilt sie als öffentlich, und jede Klasse aus dem Programm darf auf sie zugreifen (z.B. Objekte aus ihr erstellen oder eigene Klassen ableiten).

```
public class Dackel
{...}
```

Läßt du das Schlüsselwort **`public`** weg, ist die Klasse nur innerhalb ihres eigenen Paketes sichtbar, doch dazu später mehr. Im Gegensatz zu C++ dient nur das Schlüsselwort **`public`** dazu, die Sichtbarkeit zu bestimmen. Fehlt dieses Schlüsselwort, ist die Klasse automatisch „privat“.

Der Kopf einer Klasse definiert alle ihre formalen Eigenschaften. So weiß der Compiler, wie er mit diesem Teil des Programmcodes umgehen soll. Der eigentliche Hauptteil einer Klasse definiert ihre Funktionalität. Was soll sie eigentlich tun?

Formal gesehen brauchen wir dazu eine Reihe von Angaben:

- Klassenvariablen
- Klassenmethoden
- Instanzvariablen
- Instanzmethoden

Allerdings müssen nicht alle Deklarationen gemacht werden, sondern nur die, die für die Klasse sinnvoll sind. Auf die Funktion der Klassenvariablen und –methoden komme ich im nächsten Kapitel zu sprechen. Als erstes wollen wir eine einfache Klasse zu unserem Dackelbeispiel schaffen.

Im Kapitel über die OOP haben wir gelernt, dass jedes Objekt aus einer Klasse Methoden und Datenelemente übernimmt – und so die eigene Instanz definiert. Jeder Dackel hat sein eigenes Alter und sein eigenes Gewicht, also müssen diese Werte für jedes Objekt auch neu gespeichert werden. Damit sie auch in einem Objekt zur Verfügung stehen, müssen sie zuerst in der Klasse geschaffen werden. Also:

```
public class dackel
{
//Instanzvariablen
int Alter;


//Instanzmethoden
public void bellen()
{
      System.out.println
("WauWau!\n");
}
public int getAlter()
{
      return this.Alter;
}
public void setAlter(int x)
{
      this.Alter = x;
}
}
```

Dieses Beispiel lässt sich zwar kompilieren, aber nicht ausführen! Diese Klassendeklaration ist nur ein Beispiel, es fehlt noch die Methode **`main()`**

Der eben angeführte Sourcecode stellt unsere erste voll funktionstüchtige Klasse in Java dar. Sie ist als **`public`** definiert und so im ganzen Programm sichtbar. Als einzige Instanzvariable haben wir die integre Variable **`Alter`** gewählt, die das Alter des Dackels repräsentiert. Die nächste Deklaration ist die Instanzmethode **`bellen`**, die über den schon bekannten Befehl **`System.out.println()`** den String „WauWau!“ ausgibt. Das Zeichen **`\n`** am Ende des Strings, das dir sicher noch öfter begegnen wird, repräsentiert einen Zeilenumbruch. Aufmerksame Leser sind sicher über das **`public`** am Anfang der Methode gestolpert. Ähnlich wie bei der Klassendeklaration wird auch hier mit diesem Schlüsselwort die Sichtbarkeit bestimmt. Eine als **`public`** definierte Methode ist überall dort sichtbar, wo auch die Klasse sichtbar ist. Es gibt eine Reihe von weiteres Schlüsselwörtern, die die Sichtbarkeit steuern. Darauf komme ich auf Seite 48 zu sprechen.

Die letzten beiden Methoden **`getAlter()`** und **`setAlter()`** sind die Zugriffsmethoden für die Instanzvariable **`Alter`**. Über diese Funktionen kann das Alter des Dackels gelesen und wieder überschrieben werden. Es ist zwar nicht nötig, die Instanzvariablen auf diesem Wege zu verändern, aber es ist meiner Meinung nach eleganter. Außerdem zeigt es in diesem Beispiel schön die Funktion des Schlüsselworts **`this`**, das sich grundsätzlich auf das aktuelle Objekt bezieht. Innerhalb der Javasyntax werden die Objekte über die Punktnotation angesprochen. Da du allerdings nicht weißt, wie das Objekt heißen wird, das aus der Klasse **`dackel`** gezogen wird, musst du dich mit dem praktischen **`this`** aus der Affäre ziehen. Innerhalb der Klassendeklaration ist es selbstverständlich auch möglich, auf **`this`** zu verzichten, da die Zuordnungen immer eindeutig sind; allerdings wird der Quelltext dadurch klarer.

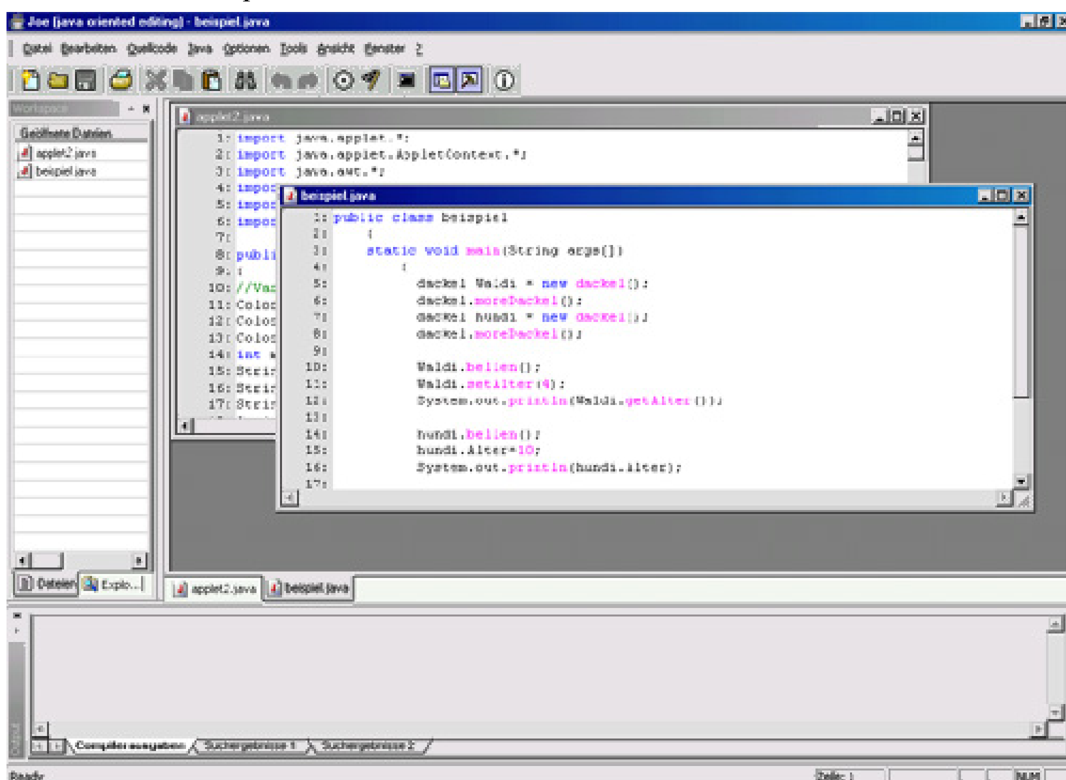
> Führe dir bitte noch einmal vor Augen, dass dieses Programm hier eigentlich kein richtiges Programm ist, weil es NICHTS tut! Wir haben lediglich die Struktur einer Klasse festgelegt, die nicht einmal Speicher belegt. Diese Struktur können wir in ein anderes Programm einbauen – und so die Funktionalität der Klasse **`dackel`** nutzen. Und damit es nicht langweilig wird, werden wir im nächsten Kapitel genau das tun.

Pakete und Verzeichnisstrukturen in Java – Teil 1

In Java wird jede kompilierte Klasse nach dem Kompilieren in einer eigenen **class**-Datei abgelegt, deren Name mit dem der Klasse identisch ist. Hast du in einer .JAVA-Datei also mehrere Klassen definiert, macht der Compiler daraus mehrere Dateien. Dieser Vorgang kann vor allem erfahrene Programmierer verwirren, die ihn eher in umgekehrter Richtung kennen. So zieht z.B. der C/C++-Compiler verschiedene Dateien (Headerdateien, Recourcecode,etc.) zu einem Programm zusammen, um daraus eine Applikation zu erstellen. Die Vorgangsweise des Javacompilers macht aber ebenfalls Sinn, da jeder Teil des Sourcecodes übersichtlich „verpackt" wird. Willst du beispielsweise nur eine Klasse ändern, brauchst du nicht den ganzen Code noch einmal zu kompilieren.

Damit kommen wir zum eigentlichen Punkt dieses Kapitels. Es ist durchaus möglich, den gesamten Sourcecode eines Javaprogramms, d.h. alle Klassen, in eine Datei zu packen und zu kompilieren. Wie gesagt spaltet Java dies dann in verschiedene .CLASS-Dateien auf. Meiner Meinung nach ist es aber sinnvoller, für jede Klasse – oder zumindest für zusammenhängende Klassenhirachien – eine eigene Datei zu erstellen. So lässt sich der Sourcecode getrennt und übersichtlich betrachten und bearbeiten.

Entwicklungsumgebungen wie JOE unterstützen dieses Vorgehen, indem sie es ermöglichen, verschiedene Dateien gleichzeitig zu öffnen und übersichtlich darzustellen.



Jede dieser Dateien muss in diesem Fall selbstverständlich für sich kompiliert werden. Greift ein Programm dann auf verschiedene Klassen zurück, reicht es, wenn die entsprechenden Dateien im selben Verzeichnis liegen.

Solltest du mehrere Klassen in einer .JAVA-Datei definieren – was bei kleinen Projekten durchaus Sinn macht –, musst du darauf achten, dass nur eine Klasse als **public** definiert ist – sonst gibt es einen Compilerfehler.

Verteilen sich die verschiedenen .CLASS-Dateien über mehrere Verzeichnisse – was bei großen Projekten sinnvoll ist –, gibt es die Möglichkeit, sogenannte *packages* zu definieren. Darauf kommen wir auf Seite 48 zurück.

Objekte

Eine Klasse ist wie eine Backform für einen ungebackenen Kuchen. Sie ist das Muster für ein Konzept, das nach ihrem Abbild in der Realität geformt werden soll. Soll das funktionieren, müssen wir an Hand einer Klasse Objekte erschaffen, die deren Funktionalität übernehmen. Das geht mit dem Schlüsselwort **new**, das auf der Basis einer Klasse ein neues Objekt erstellt. Ein Objekt der Klasse **dackel** namens Waldi erstellen wir z.B. so:

```
dackel Waldi = new dackel();
```

Der erste Teil der Deklaration gleicht der Erstellung einer Standardvariablen. Der Typ (Klasse) definiert den darauf folgenden Namen (des Objekts). Damit das Objekt einen Wert bekommt, wird es über das Schlüsselwort **new** gesetzt. Programmtechnisch gesehen ruft **new** den Konstruktor einer Klasse auf, der für den organisatorischen Teil eines Objektes sorgt. Er reserviert Speicher für das Objekt und sichert, dass alles seinen geordneten Gang geht. Der Konstruktor hat immer denselben Namen wie die Klasse selber, gefolgt von zwei Klammern. Das Ergebnis dieser Aktion ist die Rückgabe eines neuen Objektes, das ab sofort über den Namen **Waldi** zu erreichen ist. Waldi kann und weiss alles, was in der Klasse an Methoden und Datenelementen vorgesehen wurde.

Hier ein kleines Beispiel der Klasse Dackel:

```
public class beispiel
{
static void main(String args[])
      {
dackel Waldi = new dackel();
dackel hundi = new dackel();
      Waldi.bellen();
      Waldi.setAlter(4);

System.out.println(Waldi.getAlter())
;
hundi.bellen();
hundi.Alter=10;
System.out.println(hundi.Alter);
}
}
```

> Dieser Quelltext sollte in eine eigene Datei mit dem Namen BEISPIEL.JAVA geschrieben und im selben Verzeichnis gespeichert werden wie die Datei DACKEL.CLASS – ohne diese Datei geht es nicht! Wie ich bereits im vorigen Kapitel bemerkte, ergibt sich so eine sinnvolle Strukturierung, da wir auf diesen Beispiele aufbauen werden.

Im Beispielprogramm greifen wir auf die in der Datei DACKEL.CLASS definierte Klasse **dackel** zu. Sagen wir dem Compiler nicht explizit, wo diese Klasse liegt, sucht er als erstes im selben Verzeichnis – womit er in diesem Fall ja auch richtig liegt.

Die neue Klasse **beispiel** besteht nur aus der Methode **main()**, die als Einstiegspunkt in das Programm dient. Vergiss nicht, das Programm über die Datei BEISPIEL.CLASS aufzurufen. Der erste Schritt in **main()** ist die Deklaration eines neuen Objektes mit dem Namen **Waldi** aus der Klasse **Dackel**. Um zu zeigen, dass Objekte der selben Klasse mit verschiedenen Werten nebeneinander existieren können, wird danach ein weiteres Objekt mit dem Namen **hundi** vom selben Typ erschaffen. Der nächste Schritt ist nun der Aufruf der Objektmethode **bellen()** vom Objekt **Waldi**. Das geschieht über die überall in Java gültige Punktnotation, mit der in Objekten navigiert werden kann.

Willst du eine Methode aus einem Objekt aufrufen, muss du Java zuerst mitteilen, welches Objekt du meinst. Hier ist das **Waldi**. Dann folgt nach einem trennenden Punkt der Aufruf der eigentlichen Methode. Formal sieht das Ganze in seiner einfachsten Form so aus:

```
Objekt . Methode()
```

Der direkte Aufruf von Methoden in einem Programm ist nicht möglich, da diese global definiert sein müssten. Gerade das wird von der objektorientierten Theorie aber verboten. Es gibt keine Funktionen und keine Variablen, die unabhängig von einem Objekt sind. Da jedes Objekt jede Variable neu erschafft und mit einem eigenen Wert versieht, ist es durchaus möglich, mehrere Variablen mit demselben Namen in einem Programm zu verwenden. Unterschieden werden diese Werte durch das jeweilig übergeordnete Objekt.

Methoden

Die nächste Zeile des Programms ruft eine weitere Methode des Objektes **Dackel** auf, die ich bereits als Zugriffsmethode vorgestellt habe. **setAlter()** setzt die Objektvariable **Alter** auf den Wert des Übergabeparameters. Die Methode **setAlter()** wurde in der Klasse **Dackel** wie folgt definiert:

```
public void setAlter(int x)
{
this.Alter = x;
}
```

Dieser Prototyp der Methode definiert die eigentliche Funktion des Ganzen. Gehen wir ihn der Reihe nach durch. Als erstes wird festgelegt, dass die Methode eine öffentliche Methode ist (**public**). Das bedeutet, dass diese Methode für alle Klassen im Programm sichtbar ist. Es gibt eine ganze Reihe von verschiedenen Möglichkeiten, die Sichtbarkeit von Methoden wie auch Klassen festzulegen. Darauf gehen wir auf Seite 48 ein.

Nach dem Schlüsselwort **public** folgt das Schlüsselwort **void**, dass ebenfalls eine Eigenschaft der Methode festlegt. **void** sagt dem Javacompiler, dass diese Methode keinen Rückgabewert besitzt. Manchmal muss eine Methode einen Wert an das Programm zurückliefern können, um Informationen weiterzugeben. Diese Technik werden wir bei der Methode **getAlter()** kennen lernen.

Nach **void** folgt der eigentliche Name der Methoden, über den diese im Programm aufgerufen wird. Die darauf folgenden Klammern haben eine besondere Aufgabe: Hier wird festgelegt, mit welchen Parametern die Methode arbeiten kann bzw. welche sie erwartet. Diese Daten können frei festgelegt werden und ergeben sich in der Regel aus der zu erfüllenden Aufgabe. Die Methode **setAlter()** soll, wie ihr Name sagt, den Wert der Variablen **Alter** im jeweiligen Objekt der Klasse **Dackel** festsetzen. Dazu ist es nötig, dass beim Aufruf der Methode das Alter übergeben wird. Um das zu erreichen, wird im Kopf der Methode (so nennt sich die Anweisung vor den geschweiften Klammern) eine neue Variable initialisiert, die diesen Wert übernimmt. Diese Variable ist vom Typ **Integer** und heißt schlicht **x** (**int x**). Der Gültigkeitsbereich beschränkt sich auf die eigene Methode.

Nach dem Kopf der Methode folgen dann die eigentlichen Anweisungen, die von geschweiften Klammern umschlossen werden {...}

Hier wird die eigentliche Funktionalität der Methode realisiert. Beim Aufruf der Methode werden die hier stehenden Begriffe ausgeführt. Bei der Methode **setAlter()** beschränken sie sich allerdings auf eine einzige Zeile. Die Methodenvariable **Alter** wird auf den Wert von **x** gesetzt und somit das Alter übergeben. Damit hat die Methode hat ihre Aufgabe erfüllt und beendet sich.

Java kehrt in einem solchen Fall automatisch an die Stelle zurück, von der diese Methode aufgerufen wurde, und führt das Programm weiter aus. Im nächsten Schritt kommt der Compiler zur Zeile :

```
System.out.println(Waldi.getAlter())
;
```

Diese komplexe Anweisung besteht aus mehreren verschachtelten Befehlen, die der Reihe nach aufgelöst werden müssen. Hier geht Java von innen nach außen. Entsprechend muss zunächst einmal der Methodenaufruf

```
Waldi.getAlter()
```

ausgeführt werden, weil das Ergebnis der Parameter für **println()** ist. Die Methode **getAlter()** stellt sich auf den ersten Blick ähnlich wie **setAlter()** dar. Der Unterschied liegt im Detail: Im Gegensatz zur vorherigen Funktion soll hier die Variable **Alter** des entsprechenden Objektes der Klasse **Dackel** ausgelesen werden. Entsprechend sieht die Methode hier so aus:

```
public int getAlter()
{
      return this.Alter;
}
```

Anstelle von **void** finden wir hier das Schlüsselwort für Integervariablen vor dem Methodennamen. Hier wird dem Compiler einfach mitgeteilt, welche Art Wert diese Funktion zurückliefern wird – in unserem Fall **int**. Eine Methode kann ohne Einschränkungen

jeden beliebigen Typ eines Wertes zurück geben. Im Unterschied zu den Parametern einer Methode, die quasi die Informationen in die andere Richtung liefern, darf hier aber wirklich *nur ein einziger* Wert übergeben werden. Es ist nicht möglich, 2 oder mehr Variablen zurückzugeben.

Da die Methode für ihre Aufgabe keine Informationen benötigt, sind die Klammern im Kopf leer.

Die Anweisungen zwischen den geschweiften Klammern beschränken sich auch hier auf eine Zeile. Dabei stolpern wir über den eminent wichtigen Befehl **`return`**, der die inzwischen viel zitierte Aufgabe der Datenrückgabe hat. Wichtig ist, dass der von **`return`** übergebene Wert mit dem Typ im Methodenkopf überein stimmt. Es darf also z.B. kein **`char`**-Wert in einer **`int`**-Methode zurückgegeben werden. In unserem Fall ist das kein Problem, denn es wird schlicht das Alter des Objektes an das Hauptprogramm übergeben und so auch gleichzeitig die Methode beendet.

Da die **`getAlter()`**-Anweisung als Parameter in der **`println()`**-Methode steht, wird der zurückgegebene Wert automatisch an die Anweisung **`println()`** übergeben, die Java als nächstes aufruft. Da diese Methode direkt aus der Klasse **`out`** stammt, die wiederum im Paket **`System`** zu finden ist, muss sie etwas umständlich über zwei Schritte benannt werden. Das Ergebnis ist so simpel wie genial: Das Programm gibt das Alter des Objektes **`Waldi`** auf dem Bildschirm aus.

Im Laufe der folgenden Zeilen werden weitere Methoden aufgerufen, die teils vom Objekt **`Waldi`**,teils vom Objekt **`Hundi`** stammen. Da beide Objekte dieselbe Mutterklasse haben, dürfen sie die selben Methoden benutzen. Allerdings unterscheiden sich die Werte der Objektvariablen voneinander!

Objektvariablen direkt zuweisen und abrufen

Im letzten Kapitel haben wir den Einsatz von Objektmethoden kennengelernt, die es erlauben, Objektvariablen zu setzen oder auszulesen. Diese sogenannten Zugriffsmethoden dienen in der Regel zur besseren Strukturierung eines Programms und machen durch eine klare Variablenpolitk den Code transparenter. Allerdings ist der Einsatz von Zugriffsmethoden kein Muss, weil Objektvariablen auch direkt bearbeitet werden können. Im obigen Programm habe ich sie benutzt, um den Einsatz von Methoden zu demonstrieren. Wenn man das Programm weiter verfolgt, stößt man auf folgende Zeile:

```
hundi.Alter = 10;
```

Du ahnst es vielleicht schon – diese Zeile bewirkt dasselbe wie die Zugriffsfunktion **`setAlter()`**. Die Objektvariable **`Alter`** des Objektes **`hundi`** wird auf den Wert **`10`** gesetzt. Das ganze geschieht auch hier über die einfache Punktnotation, die ein Objekt von seiner jeweiligen Eigenschaft bzw. Methode trennt. Die Zuweisung erfolgt über ein Gleichheitszeichen.

Etwa ebenso erfolgt das Auslesen der Variablenwerte, wie es in der darauffolgenden Zeile im Beispielprogramm erscheint.

```
System.out.println(hundi.Alter);
```

Hier wird das die Variable **`Alter`** des Objekts **`hundi`** direkt ausgelesen und an die Funktion **`println()`** übergeben. Dies geschieht, genau wie beim Setzen der Variable, über das Objekt und den Namen des Wertes.

In unserem Programm wird der Wert **`10`** ausgegeben, den wir in der Zeile zuvor setzten.

Die Methode static main()

Unser Beispielprogramm bestand bei näherer Betrachtung eigentlich aus zwei verschiedenen Teilen, die wir im wesentlichen betrachtet haben: Zum einem aus der Klassendeklaration der Klasse **`dackel`** und zum anderen aus dcm eigentlichen Programm, das die Klassendatei benutzt. Verknüpft werden diese beiden Teile durch den Einsatz des Objektkonstruktors **`new`** und den Klassennamen. Java sucht automatisch nach einer passenden Datei und bindet sie in das Programm ein.

Wer versucht hat, die Datei DACKEL.CLASS für sich alleine zu starten, wird schnell gemerkt haben, dass dies nicht ohne weiteres geht. Der Interpreter gibt folgende Fehlermeldung aus:

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

und beschwert sich, dass in der Datei die Methode **main()** fehlt. Dieser Einwand ist berechtigt, denn der Javainterpreter nutzt diese Methode als Einstiegspunkt für den Start des Programms. Soll das problemlos funktionieren, müssen einige Voraussetzungen erfüllt sein.

Betrachten wir einmal die **main()**-Funktion aus unserem Beispielprogramm:

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

Als erstes fällt das Wörtchen **static** am Anfang der Zeile auf. Damit wird dem Java-Interpreter gesagt, dass es sich um eine Klassenmethode handelt. Was das genau ist, erfahren wir im nächsten Kapitel, doch soviel schon im Voraus: Klassenmethoden können ohne Objektbindung ausgeführt werden. Also darf kein Objekt der Klasse, die diese Methode enthält, erstellt werden, um sie zu benutzen. Wer einen Schritt weiterdenkt, wird schnell erkennen, warum das nötig ist. Beim Start des Programms gibt es noch nichts, womit es arbeiten könnte – weder Objekte noch die dazugehörigen Methoden können ausgeführt werden, weil der nötige Boden fehlt. Um überhaupt die ersten Schritte machen zu können, braucht Java eine Klassenmethode, die es ausführen kann. Diese Methode muss immer **main()** heißen und darf keine Rückgabewert besitzen – im Gegensatz zu C/C++.

Der Parameter dieser Funktion stellt eine Besonderheit dar, denn er besteht aus einer Liste aus Stringvariablen. Die Natur der Listen (Feldtypen) werden wir ebenfalls später kennenlernen. Doch wofür brauchen wir Übergabeparameter bei einer Funktion, die als allererste ausgeführt wird? Woher kommen die Werte, und wer übergibt sie? Die Antwort ist einfach und lautet: Vom Betriebssystem!

Wer schon mal an der DOS-Oberfläche gearbeitet hat, kennt sicher den praktischen Befehl *copy*, mit dem es möglich ist, Dateien zu kopieren. Die Syntax ist recht einfach:

```
copy [Quelldatei] [Zieldatei]
```

Dem Programm werden beim Start zwei Informationen mitgegeben, die verarbeitet werden. Für die Übergabe ist das Betriebssystem zuständig, das diese Werte nimmt und der ersten Funktion als Parameter übergibt – bei Java ist das **main()**. Diese werden in der Liste **args[]**
public class dackel
(Argument String) gespeichert. Auf Seite 42 werden wir ein komplettes Programm um diese Technik schreiben.

Der Rest der **main()**-Methode dürfte bekannt sein und behandelt keine neuen Themen mehr. Wesentlich ist die Erkenntnis, dass Java diese Methode als Einstiegspunkt wählt und hier das Programm startet. Es ist eine statische Methode, dass heißt es braucht kein Objekt erstellt werden, um sie zu benutzen. Fehlt diese Methode, funktioniert das Programm nicht.

Klassenvariablen und -methoden

Wie wir bereits in der OOP-Theorie gelernt haben, unterscheiden wir zwischen Objekten und den Klassen, aus denen sie erstellt wurden. Erstellst du ein Objekt aus einer Klasse, übernimmst du die gesamte Funktionalität und Datenstruktur in Form von Variablen und Methoden in dein Objekt. So hat zum Beispiel jedes Objekt der Klasse **Dackel** die Methode **bellen()** und das Datenelement **Alter**. Da jeder Dackel ein eigenes Alter hat und auch unabhängig von den anderen laufen möchte, ist es notwendig, für jedes Objekt der Klasse **Dackel** diese Daten neu zu speichern und zu definieren.

Doch was ist mit Daten wie: „Anzahl aller Dackel auf der Welt“ ? Diese Daten können nicht innerhalb eines Objektes gespeichert werden, da sie nicht von einem, sondern von allen Objekten abhängig sind. Für diesen Fall hat Sun in Java die Möglichkeit der Klassenvariablen bzw. –methoden vorgesehen.

Im Kapitel über die Methode **main()** haben wir uns dieser Thematik schon angenähert. Eine Klassenmethode bzw. –variable ist unabhängig von einer Objektbildung der jeweiligen Klasse. Sie kann direkt über die Klasse aufgerufen und gesetzt werden.

Soll eine Variable oder Methode als Klassenmethode gekennzeichnet werden, muss bei der Deklaration das Schlüsselwort **static** vor sie gesetzt werden.Für eine Demonstration dieser Möglichkeit werden wir unsere Dackelklasse ein wenig erweitern:

```
  {
//Instanzvariablen
int Alter;
//Klassenvariablen
  static int anzahl;
  //Instanzmethoden
public void bellen()
  {
  System.out.println ("WauWau!\n");
  }
public int getAlter()
  {
  return this.Alter;
  }
public void setAlter(int x)
  {
  this.Alter = x;
  }
  //Klassenmethoden
  static void moreDackel()
        {
        dackel.anzahl = dackel.anzahl + 1;
      }
  }
```

Es wurde eine neue Variable und eine neue Methode eingefügt, wobei beide mit dem Schlüsselwort **`static`** deklariert wurden. Die Variable **`Anzahl`** steht für die Anzahl aller bisher geschaffenen Dackelobjekte.

Jeweils um 1 erhöht wird die Variable durch die Klassenmethode **`moreDackel()`**. In dieser Methode erkennt man, wie der Zugriff auf die Klassenvariable erfolgt: Es wird einfach der Name der Klasse anstelle eines Objektnamen gesetzt.

In einem zweiten Schritt bringen wir die neuen Möglichkeiten in unserem Beispielprogramm unter:

```
public class beispiel
      {
      static void main(String args[])
            {
              dackel Waldi = new dackel();
              dackel.moreDackel();
              dackel hundi = new dackel();
              dackel.moreDackel();

              Waldi.bellen();
              Waldi.setAlter(4);
              System.out.println(Waldi.getAlter());

              hundi.bellen();
              hundi.Alter=10;
              System.out.println(hundi.Alter);
  System.out.println(dackel.anzahl);
            }
      }
```

Nach jedem neu erschaffenen Dackelobjekt erscheint der Methodenaufruf **`moreDackel()`**, der ebenfalls über den Klassennamen aufgerufen wird.Ganz am Ende des Programms gibt ein weiterer Befehl den Wert der Klassenvariable **`Anzahl`** aus, der – keine Überraschung – 2 beträgt. Klassenmethoden und –variablen können sich im vielen Situationen als sehr wichtig erweisen und dienen dazu, Übersicht zu schaffen.

Konstruktoren und Destruktoren

Nachdem wir uns ausgiebig mit den Grundlagen von Klassen und Objekten beschäftigt haben, können wir uns jetzt einigen interessanten Details widmen. Wer schon etwas Erfahrung mit anderen objektorientierten Sprachen hat, dem wird das Konzept des Konstruktors einer Klasse bekannt vorkommen. Das Prinzip ist so simpel wie genial: Ein Konstruktor ist nichts anderes als eine Methode die beim Erschaffen eines neuen Objektes automatisch aufgerufen wird. Dadurch ist es möglich, ein neues Objekt mit bestimmten Werten zu initalisieren oder sogar Steuermethoden aufzurufen. Soll Java eine Methode als Konstruktor erkennen und sie automatisch ausführen, muss sie den selben Namen tragen wie die Klasse selbst. Der Konstruktor der Klasse **`Dackel`** sieht also wie folgt aus:

```
public dackel()
{...}
```

Hat eine Klasse keinen expliziten Konstruktor, wird automatisch einer von Java gestellt. Programmtechnisch gesehen hat dieser Konstruktor allerdings keine Auwirkungen, wie unsere letzten Programme beweisen.

Alles innerhalb der geschweiften Klammern wird beim Erschaffen eines neuen Dackelobjektes automatisch ausgeführt.

```
public dackel()
{
System.out.println(„Geburt eines Dackel!\n");
}
```

Diese kleine Erweiterung der Klasse **`Dackel`** würde die Erschaffung der Objekte **`waldi`** und **`hundi`** mit den Satz „Geburt eines Dackels!“ auf dem Bildschirm quittieren.

Das genaue Gegenteil des Konstruktors ist der Destruktor, der die Aufgabe hat, hinter einem Objekt aufzuräumen. Das wird immer dann nötig, wenn das Objekt nicht mehr benötigt und sein Speicher an den Rechner zurück gegeben wird. Ein Destruktor hat die Aufgabe den sogenannten *Garbage Collector* zu starten, der gewisse Reinigungsarbeiten übernimmt. Unter anderem ist es seine Aufgabe, alle Daten zu löschen, die zu diesem Objekt gehören. Auch der Destruktor wird automatisch einem Objekt zugewiesen, wenn dieses keinen besitzt. Der Methodenname des Destruktors lautet unabhängig vom Methodennamen **`finalize()`**; er erscheint in folgender Form:

```
protected void finalize()
{..}
```

Alles innerhalb der geschweiften Klammern wird automatisch bei der Zerstörung eines Objektes aufgerufen. Das ist spätestens dann der Fall, wenn das Programm zu Ende ist.

Um den Sinn von Konstruktoren und Destruktoren zu verdeutlichen, habe ich ein kleines Beispiel entworfen, dass unser Programm rund um den Dackel sehr viel schlanker macht:

```
public class dackel
      {
       //Konstruktor
       public dackel()
            {
            System.out.println("Geburt eines Dackels!\n");
            dackel.anzahl=dackel.anzahl+1;
            this.Alter=1;
            }
      //Destruktor
      protected void finalize()
            {
            dackel.anzahl=dackel.anzahl-1;
            }
    //Instanzvariablen
    int Alter;
    //Klassenvariablen
      static int anzahl;
      }
```

Dazu die neue Programmdatei:

```
public class beispiel2
      {
      static void main(String args[])
            {
              dackel Waldi = new dackel();
              dackel hundi = new dackel();

              System.out.println(Waldi.Alter);

              hundi.Alter=10;
              System.out.println(hundi.Alter);
              System.out.println(dackel.anzahl);
              }  }
```

Im Gegensatz zu unserem alten Programm, in dem wir die Anzahl der Dackelobjekte in einer eigenen Klassenmethode hochzählen mussten, überließen wir diese Aufgabe hier dem Konstruktor. Dann wird die Objektvariable **Alter** mit dem Standardwert 1 initialisiert – so haben die Dackelobjekte einen sinnvollen Wert.

Der Destruktor verfeinert das Programm noch weiter und sorgt dafür, dass der Wert der Variable **Anzahl** immer aktuell ist, indem er bei der Zerstörung des Objektes herunterzählt.

Konstruktor mit Parametern

Im Beispiel des letzten Kapitels nutzen wir den Konstruktor, umden Objektvariablen bei ihrer Erschaffung Standardwerte zuzuteilen. So wurde verhindert, dass ein Dackelobjekt in der Variable **Alter** den Wert **0** hat.
Das Beispiel lässt sich allerdings noch weiter denken: Manchmal ist es nötig, dass ein Objekt nicht mit dem Standardwert initialisiert wird, sondern spezielle Werte zur Laufzeit des Programms zugewiesen bekommt. Um dies zu realisieren, unterstützt Java die Möglichkeit der Parameterübergabe an einen Konstruktor. Allerdings ist die Sache doch nicht halb so aufregend wie sie klingt, da ein Konstruktor letztendlich auch nichts anderes ist als eine einfache Methode. Das Prinzip funktioniert genau wie bei einer normalen Methode im Programm. Das setzt voraus, dass der Konstruktor den selben Namen hat wie die Klasse.

```
public Klassenname (Parameter)
```

Die Parameter werden innerhalb des Konstruktors wie Variablen behandelt und können entsprechend gelesen und gesetzt werden. Analog zm Beispiel oben könnten wir den Konstruktor für die Dackelklasse auch so schreiben:

```
public dackel(int var)
      {
System.out.println("Geburt eines
Dackels!\n");
dackel.anzahl=dackel.anzahl+1;
this.Alter=var;
      }
```

Der **int**-Parameter **var** übernimmt einen Wert und setzt im Konstruktor die Variable **Alter** des aktuellen Objektes darauf. Ansonsten ist alles beim alten geblieben.

Die Erstellung eines Dackel-Objektes im Beispielprogramm würde so aussehen:

```
dackel Waldi = new dackel(4);
```

Der Wert in den Klammern übergibt dem Konstruktor den Wert, den er übernehmen soll.

> Wenn du das Programm derart änderst, musst du jedes Objekt mit einem Parameter erstellen, sprich grundsätzlich einen eigenen Konstruktor verwenden. Hast du selbst einen Konstruktor erstellt, akzeptiert Java keinen Standard-Konstruktor mehr.

Mehrere Konstruktoren

Um die Bindung an einen einzigen Konstruktor zu umgehen, erlaubt Java die Erstellung mehrerer Konstruktoren. Das erfordert, dass alle Konstruktoren einer Klasse denselben Namen haben. Der Unterschied sind die Parameter, mit denen verschiedene Konstruktoren erkannt werden.

So kann man einen Standardkonstruktor ohne Parameter erstellen, der alle Objektvariablen auf einen simplen Startwert setzt. Ein weiterer Konstruktor übernimmt einen Parameter, den er nutzt, und noch einer kann mit zwei Parametern ein Objekt schaffen. Für unser Dackelobjekt könnte unsere Dackelklasse so aussehen:

```
public class dackel
       {
        //Konstruktor1
        public dackel()
              {
              System.out.println("Geburt eines Dackels!\n");
              dackel.anzahl=dackel.anzahl+1;
              this.Alter=1;
              this.Name="Dackel";
              }
        //Konstruktor2
        public dackel(int var)
              {
              System.out.println("Geburt eines Dackels!\n");
              dackel.anzahl=dackel.anzahl+1;
              this.Alter=var;
              this.Name="Dackel";
              }
        //Konstruktor3
              public dackel(int var, String svar)
              {
              System.out.println("Geburt eines Dackels!\n");
              dackel.anzahl=dackel.anzahl+1;
              this.Alter=var;
              this.Name=svar;
              }

       //Destruktor
       protected void finalize()
              {
              dackel.anzahl=dackel.anzahl-1;
              }
    //Instanzvariablen
    int Alter;
    String Name;
    //Klassenvariablen
       static int anzahl;
       }
```

Die Klasse **`Dackel`** enthält nun 3 verschiedene Konstruktoren. Der erste übernimmt keine Parameter und setzt alle Instanzvariablen auf Standardwerte. Damit es nicht langweilig wird, habe ich eine weitere Instanzvariable beigefügt – den Namen des Dackels. Der Typ ist **`String`** – wir lernen ihn später genauer kennen. Hier sei gesagt, dass wir mit einem String Buchstabenketten speichern können.

Werden beim Erstellen eines Objektes keine Parameter gesetzt, heißt der Dackel „Dackel", und sein Alter ist 1.

Anders sieht es beim zweiten Konstruktor aus, der uns schon bekannt ist. Hier wird das Alter übergeben, das so flexibel gesetzt werden kann. Der Name bleibt auch in diesem Fall schlicht „Dackel".

Der dritte Konstruktor übernimmt zwei Parameter und deckt so alle Objektvariablen ab. Sowohl das Alter als auch der Name können beliebig gewählt und als Parameter gesetzt werden. Diese Art der Programmierung erlaubt uns folgendes Beispielprogramm:

```
public class beispiel2
      {
      static void main(String args[])
            {
              dackel Fiffi = new dackel();
              dackel Waldi = new dackel(3);
              dackel Hundi = new dackel(4,"Hundi");

              System.out.println(Fiffi.Name);
              System.out.println(Waldi.Alter);
              System.out.println(Hundi.Alter);
              System.out.println(Hundi.Name);
              }

      }
```

Die ersten drei Zeilen in der Methode **`main`** nutzen alle drei Konstruktoren der Klasse **`dackel`**. Zuerst wird der Standardkonstruktor ohne Parameter genutzt, dann wird das Alter festgelegt und zum Schluss legen wir sowohl Alter als auch Name fest.

Die folgenden Zeilen geben die wesentlichen Werte unserer Objekte wieder.

Methoden überladen

Jeder etwas erfahrene Programmierer hat im letzten Kapitels sicher mehrere Male bemerkt, dass ich den Gaul von hinten aufzäumte.

Die Technik, mehrere Konstruktoren in einer Klasse zu benutzen, ist im Prinzip nichts anderes als das *Überladen einer Methode*. Überladen bedeutet einfach, dass ein und dieselbe Methode mehrmals geschaffen wird und sich einfach anhand ihrer Parameter unterscheidet. Der Einsatz bei Konstruktoren ist nur ein spezieller Sonderfall, der mir hier sehr gut in den Aufbau des Heftes passte.

Mit dieser Erkenntnis beschränkt sich diese Technik also nicht nur auf Konstruktoren, sondern auch auf normale Methoden, wie folgendes Beispiel beweist:

```
public class dackel2
       {
        //Konstruktoren
        public dackel2()
              {
              System.out.println("Geburt eines Dackels!\n");
              dackel.anzahl=dackel.anzahl+1;
              }
       //Objektmethoden
       public void bellen()
       {
       System.out.println("WauWau!");
       }
       public void bellen(String svar)
       {
       System.out.println(svar);
       }
       //Destruktor
       protected void finalize()
              {
              dackel.anzahl=dackel.anzahl-1;
              }
    //Instanzvariablen
    int Alter;
    String Name;
    //Klassenvariablen
       static int anzahl;
       }
```

Die Klasse habe ich **Dackel2** genannt, um sie von ihrer Vorgängerin zu unterscheiden. In dieser Version existiert zweimal die Methode **bellen()**, die sowohl mit als auch ohne Parameter genutzt werden kann.

Die Version ohne Parameter gibt „WauWau!“ auf dem Bildschirm aus, während du in der anderen Version den Inhalt der Ausgabe festlegen, also entscheiden kannst, wie dein Dackel bellt. Mein Vorschlag sieht so aus:

```
public class beispiel3
       {
       static void main(String args[])
              {
                dackel2 Fiffi = new dackel2();
                Fiffi.bellen();
                Fiffi.bellen("MIAU!");
                }
       }
```

In diesem kurzen Beispielprogramm, das auf der Klasse **dackel2** beruht – was nicht übersehen werden darf! –, werden beide Möglichkeiten des Bellens genutzt. Die Ausgabe dieses Programms ist „WauWau!“ gefolgt von einem „MIAU!“.

Pakete und Verzeichnisstrukturen in Java -Teil 2

Da ein Java-Programm in der Regel aus einer Reihe von verschiedenen **class**-Dateien besteht, ist es vor allem in größeren Projekten sinnvoll, die Dateien in einer eigenen Verzeichnisstruktur zu ordnen. So können z.B. Klassendateien, die reine Datenstrukturen enthalten (DACKEL.CLASS, PUDEL.CLASS) von Dateien getrennt werden, die das Programm steuern und ausführen. Diese Unterteilung erleichtert das Auffinden und Bearbeiten von bestimmten Dateien ungemein.

Damit Java bei der Programmausführung die entsprechenden Dateien auch findet, müssen sie in sogenannten Paketen zusammengefasst werden. Diese Pakete werden erstellt mit dem Schlüsselwort **package**; grundsätzlich geben sie schlicht die relative Verzeichnisstruktur um die Programmdatei wieder. Wird kein Paket angegeben, erstellt Java gewissermaßen ein Standardpaket, dass nur den Zugriff auf Dateien im selben Verzeichnis erlaubt. Alle bisher geschriebenen Programme haben auf diese Weise funktioniert.

Möchtest du z.B. die Klasse **dackel** in dem Paket **hunde** ablegen, um später weitere Hunderassen in (wie PUDEL.CLASS) hier zusammenzufassen, muss die erste Zeile der Datei so aussehen:

```
package hunde;
public class dackel
{...}
```

Damit wird dem Compiler gesagt, dass er diese Klasse im Unterverzeichnis HUNDE unter dem aktuellen Verzeichnis des auszuführenden Programms findet. Im Klartext heißt das: du musst ein Verzeichnis HUNDE anlegen und die Datei DACKEL.CLASS dort hinein kopieren. Willst du weitere Unterverzeichnisse anlegen, kannst du auf die bekannte Punktnotation zurückgreifen. Das Unterverzeichnis TIERE\HUNDE\ würde als Paket so aussehen:

```
package tiere.hunde;
```

Java umgeht mit dieser Technik die systemspezifischen Unterschiede von Verzeichnisstrukturen und wahrt so die Plattformunabhängigkeit.

Soll in einem Programm auf eine solche Klasse zugegriffen werden, muss auch hier die Punktnotation genutzt werden. Die Erstellung eines Objektes aus der Klasse **Dackel** würde so aussehen:

```
hunde.dackel Fiffi = new
hunde.dackel();
```

Der Aufruf eine Klassenmethode funktioniert entsprechend:

```
dackel.moreDackel();
```

Diese Technik nutzten wir schon öfter bei der Klassenmethode **println()**, die die Ausgabe von Daten auf dem Bildschirm errlaubt.

```
System.out.println()
```

Ein Programmierer kann dank der Pakete sein Programm logisch strukturieren und seine Arbeit so erleichtern. Auch wenn im Laufe des Heftes wenig mit Paketen gearbeitet wird (sind eben doch nur Beispielprogramme), empfehle ich diese Technik wärmstens. Spätestens beim einem etwas größeren Projekt ist man dankbar für die Routine, die man gesammelt hat.

Sichtbarkeit und Zugriffsrechte

Eine wichtige Rolle spielen Pakete im Bereich der Zugriffsrechte von Klassen und Objekten.

Java bietet eine Reihe von Schlüsselwörtern, die die Zugriffsrechte und Sichtbarkeit von Klassen und Methoden bestimmen. Nachfolgend sehen wir uns alle Möglichkeiten an und besprechen die jeweiligen Optionen in einem Programm. Der grundsätzliche Umgang mit Zugriffsrechten von Klassen und Methoden unterscheidet sich nur im Detail von dem in C++ angewandten Prinzip.

Java unterscheidet zwischen fünf Schlüsselwörtern, die wir der Reihe nach besprechen.

public

Eine Klasse, Methode oder Variable, die als **public** definiert wird, ist überall sichtbar. D.h. aus einer öffentlichen Klasse kann überall im Programm, auch in anderen Klassen ein Objekt erstellt werden. Eine öffentliche Methode kann überall im Programm genutzt werden. Das selbe gilt für eine Variable – **public** legt somit keinerlei Restriktionen auf.

protected

Eine als **protected** definierte Methode oder Variable ist nur innerhalb des eigenen Paketes

und in allen abgeleiteten Klassen sichtbar. Das bedeutet, dass die Methode **bellen()** in der Klasse **Dackel** nur von Klassen im selben Paket – z.B. **pudel.class** – genutzt werden kann. Das Prinzip der Ableitung lernen wir im Kapitel über Vererbung kennen. Eine Klasse kann nicht **protected** sein.

default

Eine Klasse, Methode oder Variable wird als **default** deklariert, wenn keine anderen Schlüsselwörter genutzt wurden. Die Sichtbarkeit beschränkt sich auf das eigene Paket.

private protected

Dieses Schlüsselwort bezieht sich nur auf Methoden und Variablen. Es sorgt dafür, dass die Sichtbarkeit auf alle abgeleiteten Klassen reduziert wird. Im Paket kann diese Methode bzw. Variable nicht mehr benutzt werden.

private

Private Methoden oder Variablen sind ausschließlich innerhalb der eigenen Klasse sichtbar. Klassen können nicht private sein.

Beispiel zur Sichtbarkeit

Damit das ganze Thema etwas klarer wird, habe ich dieses kleine Beispiel geschrieben:

```
public class seeme
{
public void sichtbar()
    {
      System.out.println("Dies ist
eine public Methode!");
    }
private void unsichtbar()
    {
      System.out.println("Dies ist
eine private Methode!");
      }
protected void usePrivate()
      {
      this.unsichtbar();
      }
}
```

Die Klasse **seeme** besteht aus drei Methoden, die jeweils unterschiedliche Zugriffsrechte besitzen. Hier das dazugehörige Programm:

```
public class sichtbarkeit
      {
static void main(String args[])
            {
seeme myEye = new seeme();


myEye.sichtbar();
myEye.usePrivate();
            }
      }
```

Der erste Schritt ist die Erschaffung eines Objektes aus der Klasse **seeme**. Dieser Schritt enthählt keine bahnbrechenden Neuerungen. Interessanter sind die folgenden Schritte: Die Methode **sichtbar** kann ohne weiteres vom Programm ausgeführt werden, so wie wir es bisher kennen gelernt haben. Die Methode ist als **public** definiert und somit überall sichtbar. Wolltest du allerdings die Methode **unsichtbar()** ausführen, würde dir der Compiler bereits beim Kompilieren folgenden Fehler an den Kopf werfen:

```
No Method matching unsichtbar()
found in class seeme!
```

Da die Methode als **private** gekennzeichnet ist, ist sie für die **main()**-Methode sowie für alle anderen Klassen unsichtbar. Um sie trotzdem nutzen zu können, haben wir die Methode **usePrivate()** geschaffen, die einzig und allein die Aufgabe hat, über **this** die Methode im eigenen Objekt zu starten. Da wir uns dabei in der selben Klasse befinden – **usePrivate()** und **unsichtbar()** sind beide in **seeme**! – gibt es keine Probleme.

Weiterhin sehen wir, dass die Zugriffsmethode **usePrivate()** als **protected** deklariert wurde. Wir dürfen sie trotzdem in unserer Programmklasse **sichtbarkeit** ausführen, die **main()** enthält, weil sich beide Klassen im selben Paket befinden. Da wir keine eigene Paketstruktur angelegt haben, befinden sich alle Klassen im Standardpaket, also im selben Verzeichnis.

Um den Überblick zu wahren, gibt es zum Abschluss noch eine kleine Tabelle mit allen Schlüsselwörtern im Bereich der Zugriffsrechte.

Zugriff	**public**	**protected**	**default**	**private protected**	**private**
Zugriff im selben Paket	Ja	Ja	Ja	Nein	Nein
Zugriff von anderen Paketen	Ja	Nein	Nein	Nein	Nein
Vererbt Klassen im selben Paket	Ja	Ja	Ja	Ja	Nein
Vererbt Klassen an andere Pakete	Ja	Ja	Nein	Ja	Nein

Elementare Syntax von Java

Soll eine Programmiersprache funktionieren, benötigt sie Regeln und Konventionen, die die Definition von Strukturen auf simple und effektive Weise erlauben. Außerde, muss eine Reihe von Operatoren zugänglich sein, die die Verknüpfung verschiedener Elemente erlauben. Diese elementare Syntax ist das Herz jeder Sprache; sie macht das effektive Arbeiten möglich. Die Syntax von Java ist stark an C++ angelehnt, die vielleicht schon einigen Lesern bekannt sein dürfte.

Operatoren

Operatoren in einer Programmiersprache habe die Aufgabe, Elemente einer Sprache zu verknüpfen und miteinander kommunizieren zu lassen. Einige Operatoren haben wir bereits benutzt, ohne uns dessen bewusst zu sein. Der elementarste Operator ist wohl das Zuweisungszeichen `=`. Dank seiner können wir Werte an andere Elemente der Sprache delegieren. Sei es eine einfache Variable oder die Zuweisung eines neuen Objektes, der Vorgang ist prinzipiell immer der gleiche:

```
x = 10;
dackel waldi = new dackel();
```

Das Gleichheitszeichen weist immer den rechten Wert dem auf der linken Seite zu. Im einen Fall ist es die `int`-Zahl 10, im anderen die Rückgabe des Konstruktors der `dackel`-Klasse, also ein ganzes Objekt.

Verknüpfungsoperatoren

Mathematik ist die wesentliche Voraussetzung für das Funktionieren eines Computers. Darum besteht die Grundlage jeder Programmiersprache aus den Grundrechenarten der Mathematik, auf denen alles weitere aufbaut. Für jede Rechenart existiert ein eigener Operator, der Berechnungen mit sämtlichen uns zur Verfügung stehenden Zahlentypen möglich macht.

Operator	Bedeutung	Beispiel
+	Addition	4 + 5 = 9
-	Subtraktion	5 – 4 = 1
*	Multiplikation	4 * 5 = 20
/	Division	20 / 4 = 5
%	Modulo	21 % 9 = 3

Die vier Grundrechenarten dürften einleuchten. Die Modulo-Rechnung hingegen ist ein wenig exotischer und bedarf einiger klärender Worte. Im Prinzip handelt es sich hierbei auch um eine Division der beiden Werte. Die Rückgabe bezieht sich aber nicht auf das Ergebnis der Division, sondern auf den ganzzahligen Rest, der danach übrig bleibt. Das Beispiel rechnet sich wie folgt:

Die Zahl 9 „passt“ genau 2 mal in 21 (18). Der Rest ist 3, da 21 – 18 den ganzzahligen Rest definiert. Diese Rechnung wird in der Schule auch „Division mit Rest“ genannt.

Diese Technik eignet sich, um Intervalle in Schleifen zu schaffen – mehr dazu später. Hier ein einfaches Beispiel mit den Operatoren:

```
class operatoren
{
static void main(String args[])
      {
      int x = 3 + 4;
      int y = 5 - 7;
      double z = 16.1 / 3.4;
      int m = 17 % 3;

      System.out.println(x);
      System.out.println(y);
      System.out.println(z);
      System.out.println(m);
      }
}
```

Ich denke, das Beispiel erklärt sich selber.

Kommazahlen in Java

Bevor wir mit den Operatoren fortfahren, möchte ich eine kurze Zwischenbemerkung zum Thema Kommazahlen machen. Wie wir schon im letzten Beispiel sahen, werden Kommazahlen nach dem US-Standard geschrieben, d.h. dass anstelle des Kommas (`,`) ein einfacher Punkt gesetzt wird (`.`) Diese Schreibweise ist in Europa zwar unüblich, muss aber bei fast allen Programmiersprachen beachtet werden, da sonst Compilerfehler entstehen. Ein Komma wird meist als Trennzeichen interpretiert (z.B. in Methodenköpfen) und darf deshalb nicht für Zahlen verwendet werden.

Inkrement-Operator

Schon in der Sprache C gab es die Möglichkeit, bestimmte Funktionen und Zuweisungen, die in der Programmierung häufig vorkommen, einfach abzukürzen. Das bekannteste Beispiel ist wohl der Inkrement-Operator, der inzwischen sogar im Namen von C++ verewigt wurde. Möchtest du eine Variable um den Wert 1 erhöhen, würde das auf herkömmlichen Wege so aussehen:

```
x = x + 1;
```

Die Variable x wird auf dem Wert von sich selbst +1 erhöht. Da diese Code-Zeile in Programmen häufig vorkommt, wurde der Operator ++ beigefügt, der das selbe bewirkt:

```
x ++;
```

Dekrement-Operator

Das genaue Gegenstück zur Inkrementierung ist die Dekrementierung, die einen Wert um 1 senkt. Dies wird mit dem Operator – ermöglicht. Statt **x = x – 1;** schreibt man schlicht **x --;**

Der Wert von x wird um eins gesenkt.

Arithmetische Zuweisungsoperatoren

Arithmetischen Zuweisungsoperatoren sind eine konsequente Weiterentwicklung der Inkrement- und Dekrement-Operatoren, die die Syntax der Sprache vereinfachen sollen. Es geht darum, oft vorkommende Codefragmente durch kürzere zu ersetzen. Hier wurden auch die oben vorgestellten fünf Rechenarten berücksichtigt. Die Verkürzung wird erreicht durch die Zusammenfassung des Verknüpfungs mit dem Zuweisungsoperator =.

Operator	Beispiel	Bedeutung
+=	x+=4	x = x + 4
-=	x -= 4	x = x – 4
*=	x *= 4	x= x * 4
/=	x /= 4	x = x / 4
%=	x %= 4	x = x % 4

Einfache oder sich wiederholende Rechnungen lassen sich so schnell codieren. Der Nachteil solcher Verkürzungen liegt in der Gefahr einer aneachsenden Unleserlichkeit des Sourcecodes. Zu viele Abkürzungen können vor allem für das ungeübte Auge für Verwirrung sorgen.

Bedingungen und Entscheidungen

Inn einem Programm müssen vielfach Entscheidungen getroffen werden. So muss z.B. differenziert auf Usereingaben reagiert oder eine Berechnung je nach Situation durchgeführt werden.

Eine Entscheidung basiert immer auf einer vorher festgelegten Bedingung, die das weitere Vorgehen festlegt. Die simpelsten Bedingungen werden in fast allen Programmiersprachen über das Schlüsselwort **if** realisiert. Diese simple „wenn dann"-Konstruktion erlaubt schnelle. einfache Strukturierung eines Programms. Die Syntax sieht so aus:

```
if (Bedingung) {Anweisungen}
```

Die bedingten Anweisungen in den Schweifklammern werden nur dann ausgeführt, wenn sich die Bedingung in den Rundklammern als wahr oder **true** erweist. Ist die Bedingung falsch oder **false**, ignoriert der Compiler die Anweisungen.

Doch was genau ist **true** und **false**? Und wie definiert es sich? Eine Aussage ist dann wahr, wenn sie vom logischen Standpunkt her als richtig interpretiert werden kann.

```
4 + 5 = 9 // Diese Aussage ist
richtig!
```

Aber Logik erstreckt auf alle möglichen Aussagen und Situationen, wo Werte verglichen werden.

```
himmel = „blau" // Auch richtig!
```

Falsch ist also jede Aussage, die vom logischen Standpunkt her nicht richtig ist.

```
4 + 5 = 8 // Das ist falsch!
himmel = rosa // Auch falsch!
```

Mit dieser elementaren Festlegung können wir jetzt Bedingungen erstellen.

Wesentlich für unsere Bedingungen in Java sind eine Reihe von Operatoren, die dem Compiler mitteilen, wie zwei Werte verglichen werden. Java kennt sechs Vergleichsoperatoren:

Operator	Beschreibung
= =	Gleichheit
!=	Ungleichheit
<	kleiner als
>	größer als
<=	kleiner gleich
>=	größer gleich

Mit diesen sechs Operatoren können wir präzise Bedingungen formulieren, die alle Möglichkeiten der Entscheidung frei stellen.

> Eine häufige Fehlerquelle in vielen Programmen ist der kleine aber wichtige Unterschied zwischen dem Zuweisungsoperator = und dem Vergleichsoperator = = . Es passiert sehr schnell, dass man in einer Bedingung statt zwei Gleichheitszeichen nur eins setzt. Der Fehler fällt kaum auf – zum Glück wird er aber vom Compiler bemerkt.

Eine einfache Bedingung in Java würde also so aussehen:

```
if (x = = 5) {x++;}
if (x < 5) {x--;}
if (x > 5) {x = x + 2;}
```

> Vergiss nie, die Anweisungen innerhalb der Klammer mit einem Semikolon abzuschließen.

Ein kleines Beispiel demonstriert, welche Möglichkeiten mit Bedingungen angeboten werden:

```
class myif
{
static void main(String args[])
{
      String user = "admin";
      String passw = "geheim";
      int verify = 0;

      if (user == "admin") {verify
++;}
      if (passw == "geheim") {verify
++;}
      if (verify > 1)
{System.out.println ("Zugang
erlaubt!");}
      if (verify != 2)
{System.out.println ("Zugang
verweigert!");}
      }
}
```

Die ersten beiden `if`-Bedingungen überprüfen die Werte `user` und `passw`. Wenn diese die richtigen Werte besitzen, wird die Variable `verify` um 1 erhöht. Die dritte und vierte Bedingung testet diese Variable, ob sie den nötigen Wert erreicht hat, der den Zugang ins System erlaubt. Je nach Ergebnis wird eine entsprechende Meldung ausgegeben. Das Programm ist etwas umständlich programmiert, demonstriert aber den Umgang mit Bedingungen.

if – else-Bedingungen

Eine etwas elegantere Methode für die Arbeit mit Bedingungen ist die Präsentation von Alternativen. Das Programm hat die Wahl zwischen zwei durch die `if`-Bedingung gegebenen Möglichkeiten. Ist die Bedingung `true`, führt es die Anweisungen in den Schweifklammern aus; ist sie `false`, verfolgt es den Weg, der über `else` gegeben wird. Die Syntax sieht so aus:

```
if (Bedingung) {Anweisungen}
else {andere Anweisungen}
```

Mit else-Anweisungen lassen sich Bedingungen zusammenfassen, die sonst in zwei separaten Schritten abgeprüft werden müssten. Die Überprüfung von `verify` im Passwortprogramm könnte auch so aussehen:

```
//Alternativ mit else:
if (verify !=2) {System.out.println
("Zugang verweigert!");}
else {System.out.println ("Zugang
erlaubt!");}
```

Ist `verify` ungleich 2, wird der Zugang verweigert, ansonsten wird er erlaubt.

Verschachtelte Bedingungen

Selbstverständlich ist es auch möglich – und bisweilen sogar nötig –, Bedingungen zu verschachteln. Das bedeutet, dass innerhalb eines `if`- oder `else`-Anweisungsblocks weitere Bedingungen stehen. Klingt etwas kompliziert – ist es aber nicht! Zum Beweis das Passwortprogramm noch einmal ganz anders:

```
class myif2
{
static void main(String args[])
       {
       String user = "keinadmin";
       String passw = "falsch";
       int verify = 0;

       if (user == "admin")
       {
       if (passw == "geheim") {verify+=2;}
       }

       if (verify !=2)
       {
       System.out.println ("Zugang verweigert!");
       if (user != "admin") {System.out.println ("User falsch!");}
       if (passw != "geheim") {System.out.println ("Passwort falsch!");}
       }
       else {System.out.println ("Zugang erlaubt!");}
       }
}
```

Die Überprüfung des Passworts rutschte in den Anweisungsblock der Überprüfung des Usernamen. Diese if-Bedingung wird also nur noch ausgeführt, wenn der Username als richtig erkannt wurde. Die zweite Neuerung ist die Ausgabe des Fehlers, die durch zwei weitere Bedingungen innerhalb der `verify`-Überprüfung realisiert wird. Der User erhält also eine Meldung über seinen Fehleingabe. Durch Manipulation der Variablen `user` und `passw` kannst du alle Möglichkeiten der Programmausgabe durchtesten.

Fallunterscheidung mit switch - case

Die Fallunterscheidung ist eine spezielle Möglichkeit, Bedingungen zu überprüfen und eine Reihe von Handlungsmöglichkeiten zu generieren. Während die `if–else` Konstruktion auf zwei Alternativen beschränkt ist, kannst du mit einer `swich–case`-Konstruktion beliebig viele Möglichkeiten schaffen.

Das Prinzip ist simpel: Innerhalb der `switch-`Anweisung wird ein Ausdruch berechnet und dann der Reihe nach in allen Möglichkeiten der `case`-Anweisungen verglichen. Findet sich eine Übereinstimmung, wird der Code hinter der case Anweisung ausgeführt.

```
switch (Ausdruck) {
      case Wert1: Anweisungen1;
      case Wert2: Anweisungen2;
      case Wert3: Anweisungen3;
      case Wert4: Anweisungen4;
            }
```

Die Bedingung kann auch ein einfacher Wert sein, der mit den einzelnen Möglichkeiten verglichen wird:

```
class mycase
{
static void main(String args[])
      {
        int x = 3;
   switch (x){
    case 1: System.out.println("x
ist 1!");
    case 2: System.out.println("x
ist 2!");
    case 3: System.out.println("x
ist 3!");
    case 4: System.out.println("x
ist 4!");
    case 5: System.out.println("x
ist 5!");
       }
     }
}
```

Wird das Programm gestartet, wird folgendes Ergebnis ausgegeben:

```
x ist 3!
x ist 4!
x ist 5!
```

Das klingt seltsam, besitzt **`x`** doch nur den Wert **`3`**. Die Wahrheit bezüglich der **`switch-case`**-Anweisung ist, das Java alles, was nach der ersten wahren Bedingung kommt, als **`true`** ansieht. Es wird also nichts mehr überprüft, sondern alles ausgeführt. Das verhindern wir mit der Anweisung **`break`**, die dazu führt, dass die Ausführung der **`switch-case`**-Konstruktion abgebrochen wird:

```
class mycase
{
static void main(String args[])
      {
        int x = 3;

   switch (x){
       case 1: System.out.println("x
ist 1!");
       break;
       case 2: System.out.println("x
ist 2!");
       break;
       case 3: System.out.println("x
ist 3!");
       break;
       case 4: System.out.println("x
ist 4!");
       break;
       case 5: System.out.println("x
ist 5!");
       break;
        }
     }
}
```

Sobald eine Übereinstimmung gefunden wurde, wird der hinter **`case`** stehende Code ausgeführt. Zwangsläufig stolpert das Programm bei 3 über die Anweisung **`break`**, die die **`switch-case`**-Konstruktion sofort abbricht. Vorher wird eine Meldung ausgegeben, die diesmal auch stimmt:

```
x ist 3!
```

Schleifen in Java

Eine weiterer wichtiger Punkt der Programmstruktur ist die Arbeit mit Schleifen. Sie geben die Möglichkeit, Teile eines Programms zu wiederholen und Entwicklungen in einem Programm dynamisch zu generieren. Auch hier orientiert sich die Syntax von Java starkt an C++, was erfahrenen Programmierern einen schnellen Einstieg erlaubt. Aber auch Anfängern werden keine unüberwindlichen Hindernisse in den Weg gestellt. Die logische Struktur einer Schleife ist im Prinzip immer dieselbe. Es wird eine Bedingung gestellt, die bei jedem Schleifendurchgang erneut geprüft wird. Solange diese Bedingung **true** ist, werden die Anweisungen der Schleife ausgeführt. Der Unterschied der einzelnen Schleifenmodelle liegt im Detail und wird über die Syntax bestimmt.

Die while-Schleife

Die while-Schleife ist die einfachste aller Schleifen. Sie besteht nur aus dem Schlüsselwort **while** verbunden mit einer Bedingung, die in Klammern formuliert wird. Der darauf folgende Klammerblock wird so lange ausgeführt, bis die Bedingung den Wert **false** hat. Ein simples Beispiel:

```
class schleifen
{
static void main(String args[])
    {
    int i = 10;
    while (i >= 0)
    {
    System.out.println("i hat den
Wert: " + i);
    i --;
    }
}}
```

Die **while**-Schleife überprüft zu Beginn der ersten Ausführung des Anweisungsblockes den Wert der Variable i. Da dieser 10 beträgt, ist die Bedingung **true (10 >=0)**. Das erlaubt die Ausführung aller Anweisungen in den Schweifklammern, was dazu führt, dass der aktuelle Wert von i auf dem Bildschirm ausgegeben und die Variable gleichzeitig um 1 reduziert wird.

Anschließend wird die Schleife wieder gestartet und die Bedingung überprüft – da 9 größer als 0 ist, steht einem neuen Schleifendurchgan nichts im Wege usw ... so lange, bis i gleich 0 ist und das Programm nach der Schleife fortfährt.

Im Prinzip gleicht eine Schleife einer einfachen **if**-Bedingung, mit dem Unterschied, dass diese immer wieder wiederholt wird.

Die do-while-Schleife

Die **do-while**-Schleife ist mit der eben besprochenen **while**-Schleife identisch, nur steht hier der zu prüfende Ausdruck am Ende des Anweisungsblockes. Das bedeutet, dass die Schleife auf jeden Fall einmal ausgeführt wird, bevor die Bedingung geprüft wird, auch wenn diese gleich beim ersten Durchgang **false** war. Unser Beispielprogramm ändert sich allerdings wenig:

```
class schleifen
{
static void main(String args[])
    {
    int i = 10;
    do
    {
   System.out.println("i hat den
Wert: " + i);
    i --;
    }
    while (i >= 0);
      }
}
```

Funktion und Ausgabe des Programms gleichen dem der **while**-Schleife. Einziger Unterschied ist die geänderte Syntax der **do-while**-Schleife.

In manchen Situationen ist es notwendig, in einem Programm einen Schleifendurchlauf garantieren zu können, um bestimmte Voraussetzungen zu erfüllen. Mit einer normalen while Schleife ist das nicht immer möglich, mit einer **do-whil**eSchleife hingegen schon.

Die for-Schleife

Mit der **for**-Schleife haben wir die Königsklasse der Schleifen erreicht – sie ist wohl das mächtigste Werkzeug für Schleifenkonstruktionen. Die **for**-Schleife bietet eine ganze Reihe von Möglichkeiten, das Verhalten der Schleife zu beeinflussen, ist dafür aber auch komplizierter zu benutzen. Der Schleifenkopf besteht aus drei Teilbereichen: Der Initialisierung, der Bedingung und der Angabe der Variablenänderung. Im Anschluss daran befindet sich der Ausführungsteil. Die Syntax sieht so aus:

```
for (Initialisierung; Bedingung;
Schleifenschritt)
{...}
```

Die Initialisierung gleicht einer simplen Variablenzuweisung. Hier wird die Variable bestimmt, die die zu prüfende Bedingung steuert. In der Regel ist das eine einfache **int**-Variable, die herauf- oder heruntergezählt wird. Die Bedingung bezieht sich auf die vorher bestimmte Variable und entscheidet über Fortlauf oder das Ende des Schleifenlebens. Der Schleifenschritt ist eine Änderung der Variable, so dass in jeder neuen Runde der Schleife eine dynamische Änderung erfolgt. Im Prinzip haben wir in der **for**-Schleife alle Punkte der **while**- und **do-while**-Schleife wieder im Spiel, nur hier im Schleifenkopf zusammengefasst. Ein Beispiel:

```
class schleifen
{
    static void main(String args[])
    {
    for(int i = 10; i >= 0; i--)
    {
    System.out.println("i hat den
Wert: " + i);
    }
    }
}
```

Auch dieses Beispiel gleicht in Funktion und Ausgabe unseren letzten beiden Programmen. Der Unterschied liegt wieder in der Syntax. Wurde vorher die Dekrementierung von **i** im Ausführungsblock vorgenommen, finden wir sie hier im Schleifenkopf wieder. Auch die Initialisierung von **i** ist jetzt dort zu finden. So fassen wir alle Bestandteile der Schleife an einem zentralen Punkt zusammen und schaffen damit ein wenig Ordnung. Die Schleife bricht automatisch ab, wenn der Wert der Bedingung **false** ist.

Schleife mit break verlassen

Die Anweisung **break** haben wir bereits bei der **switch-case**-Anweisung kennengelernt. Diese Anweisung unterbricht aber nicht nur Fallunterscheidungen, sondern auch Schleifen. Sobald Java innerhalb einer Schleife auf **break** stößt, wird sie bedingungslos verlassen, ohne dass irgendeine Bedingung noch einmal geprüft wird. Das Programm wird dann an der Stelle nach der letzten Schleifenanweisung fortgesetzt.

```
class schleifen
{
static void main(String args[])
    {
    for(int i = 10; i >= 0; i--)
    {
    System.out.println("i hat den
Wert: " + i);
    break;
    }
    }
}
```

Das obige Programm gleicht unserem Beispielprogramm für die **for**-Schleifen mit dem feinen Unterschied, dass wir eine **break**-Anweisung eingefügt haben. Die Wirkung ist allerdings überzeugend: Nach der ersten Zeilen wird die Schleife abgebrochen, und das Programm wird nicht fortgesetzt. Obwohl **i** noch den Wert 10 hat, wird die Schleife nicht fortgesetzt. Die **break**-Anweisung ist eine gute Möglichkeit, eventuelle Ausnahmen in einem Programm zu behandeln.

Schleifen mit continue wiederholen

Das Gegenteil der Anweisung **break** ist die Anweisung **continue**, die dafür sorgt, dass der aktuelle Schleifendurchgang wiederholt wird. Dabei wird nicht die ganze Schleife wiederholt, sondern nur der aktuelle Durchlauf. Alle Variablen behalten ihren Wert.

```
class schleifen
{
static void main(String args[])
    {
    int i;
    for(i = 10; i >= 0; i--)
    {
    if (i > 0) {continue;}
    System.out.println("i hat den
Wert: " + i);
    }
    }
}
```

Das obige Programm gibt nur die Zeile „i hat den Wert: 0“ aus, da alle anderen Durchgänge durch den Befehl **continue** vorzeitig abgebrochen werden. Erst wenn **i** gleich 0 ist, wird **continue** nicht mehr ausgeführt, und der Schleifendurchgang findet ein Ende.

> Übermäßiger Einsatz von **break** und **continue** kann zu einer starken Unleserlichkeit des Sourcecode führen. Die Sprunganweisungen sind nicht immer leicht nachzuvollziehen und sollten deshalb nur in Ausnahmefällen eingesetzt werden.

Komplexe Datentypen

Nachdem wir uns am Anfang des Heftes ausgiebig mit den primitiven Datentypen befasst haben, müssen wir uns etwas genauer mit den komplexen Datentypen beschäftigen. Im Gegensatz zu den primitiven Typen, die auf systemgegebenen Größen aufsetzen (Ziffern, Zahlen oder einzelne Zeichen), bestehen komplexe Datentypen aus einer Zusammenfassung mehrerer Primitiva. So lassen sich komplexe Zusammenhänge in einer Datenstruktur kapseln, wodurch das Programm schlanker wird.

Arrays

Arrays sind ein typisches Beispiel für komplexe Datentypen – grundsätzlich stellen sie keinen neuen Datentyp dar, sondern ermöglichen eine komplexerr Gestaltung der schon bekannten Datentypen. Aber was ist eigentlich ein Array?

In anderen Sprachen werden Arrays auch schlicht Listen genannt, und so kann man es sich auch vorstellen. Fenke an eine einfache Einkaufsliste:

```
Brot
Käse
Wurst
Eier
Milch
```

Diese Liste ist eine Datenstruktur, die aus fünf Elementen besteht, die sich durch Wert und Stellung voneinander unterscheiden. Eine Feststellung wie „Das dritte Element der Liste“ ist ebenso eindeutig wie der Ausdruck „Wurst“. Diese Charakteristik ist wesentlich für ein Array und eigentlich auch schon das ganze Geheimnis.

Ein Array fasst also Daten eines primitiven Datentyps in einer Liste zusammen, die dann in einem Programm bearbeitet werden kann.

Die Syntax ist im Prinzip genauso einfach zu handhaben wie das Array selbst. Dieser Datentyp wird genau wie ein Objekt über das Schlüsselwort **new** erzeugt, das den jeweiligen Konstruktor des gewünschten primitiven Datentyps aufruft. Dabei wird in eckigen Klammern die gewünschte Anzahl der Elemente festgelegt.

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

Diese Code-Zeile schafft ein Array des Datentyps **Integer**, dass zehn verschiedene **int**-Werte zu speichern vermag. Die einzelnen Plätze dieses Arrays sind allerdings noch unbesetzt. Zugriff auf die Daten eines Array erhält man über die Nummer des jeweiligen Elements. Diese wird in eckigen Klammern hinter den Array-Namen gestellt und wie jede Variable geschrieben:

```
liste[0]=99;
liste[1]=100;
liste[2]=101;
```

Diese Zeilen setzen die ersten drei Elemente auf die Werte 99, 100, 101. Java zählt dabei von der Stelle 0 an aufwärts, so dass unser Array seine zehn Elemente mit den Nummern 0 bis 9 durchnummeriert.

> Es entstehen viele Fehler in Java-Programmen, weil vergessen wird, dass das letzte Array nicht den Wert 10 hat, sondern den Wert 9. Java quittiert einen solchen Fehler mit einer klaren Meldung.

Der Abruf der Array-Werte erfolgt genau so wie ihre Setzung.

```
System.out.println(liste[0]);
System.out.println(liste[1]);
System.out.println(liste[2]);
```

Java gibt mit diesen Befehlen die ersten drei Elemente des Arrays **liste[]** aus. Du siehst also, dass du Arrayelemente wie Variablen behandeln kannst.

Eine weitere Möglichkeit, ein Array zu erzeugen, ist der direkte Weg über eine Initialisierung mit Werten. Der folgende Befehl:

```
int liste[] = {3, 5, 7, 9};
```

erschafft ein Array mit vier Elementen, das die Werte 3,5,7,9 speichert. Das Schlüsselwort **new** ist in diesem Fall nicht nötig. Im Gegensatz zu C dürfen die Werte in den Schweifklammern auch Variablen sein, sofern sie im Augenblick der Initialisierung nur eindeutig belegt sind.

Arrays werden in Java wie Objekte behandelt. Sie können also nicht manuell gelöscht werden, sondern werden automatisch über den Garbage Collector „entsorgt“, wenn sie nicht mehr benötigt werden. Weiterhin können über die Punktsyntax auch Werte abgerufen werden, die Informationen über das Array enthalten. Die Variable **length** enthält z.B. die Länge der Arrays.

Die Information ist wichtig, wenn man die Länge eines Arrays nicht kennt und nicht über das Ende hinauslesen will.

Parameter der Funktion static main()

Ein typischer Anwendungsfall für Arrays mit unbekannter Länge ist die Funktion **main()** der Startklasse jedes Programms. Wir haben uns bereits in einem der letzten Kapitel etwas näher mit dieser Funktion auseinander gesetzt, ohne genauer auf den Parameter einzugehen.

Jetzt wissen wir, dass es sich dabei um ein String-Array handelt, das die Parameter des Programmstarts von der Eingabeaufforderung des Betriebssystems aus speichert. Da das Programm vorher nicht weiß, wie viele Parameter der User beim Start übergibt, wird das Array ohne genauer bestimmte Länge initialisiert. Hier hilft uns die Information der Variable **length** des Arrays weiter. Es folgt ein kleines Programm, dass die Informationen der Parameterübergabe ausliest und auf dem Bildschirm ausgibt.

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

Startest du das Programm einfach – über JOE –, wirst du allerdings entäuscht. Auf dem ersten Blick geschicht nämlich gar nichts. Die wahre Funktion des Programms offenbart sich erst, wenn es von Hand an mit beliebigen Parametern der Befehlszeile gestartet wird. Für User, die bisher nur JOE benutzten, klingt das umständlich, die Sache ist aber halb so wild. Du gibst einfach folgenden Befehl ein:

```
java staticmain [Parameter1]
[Parameter2] ...
```

und bestätigst das ganze mit ENTER. Erscheint eine Fehlermeldung, wurde kein Pfad gesetzt, wie ich das am Anfang des Heftes beschrieben habe. Das Programm muss in diesem Falle direkt im BIN-Verzeichnis des JDK gestartet werden – in der Regel C:\JDK1.3\BIN oder höhere Version. Kopiere das Programm hierher und führe es aus.

Die Parameter in eckigen Klammer können dabei durch beliebige Worte ersetzt werden.

Mein Programmstart sah so aus:

```
java staticmain Hallo Java
```

Die Ausgabe sah zwangsläufig so aus:

```
Hallo
Java
```

Was ist passiert? Gehen wir das Programm durch. Im ersten Schritt wird eine integre Zählvariable i mit dem Wert 0 initialisert. Hiermit wird die folgende **while()**-Schleife gesteuert, die sich wiederholt, bis der Wert von **i** gleich der Anzahl der Elemente von **args[]** ist. Das erreichen wir mit der Bedingung **while(i < args.length)**

args ist der Name des Startparameter der **main()**-Funktion. Hier werden alle Parameter der Eingabeaufforderung gespeichert. Wird kein Parameter übergeben, ist **args.length** gleich 0, und die Schleife bricht sofort ab. Das ist der Effekt, wenn wir das Programm einfach aus JOE heraus starten. Übergeben wir aber Parameter wie „Hallo Java", ist der Wer von **args.length** gleich 2, und die Schleife wird gestartet. In der Schleife selbst passiert nichts aufregendes oder neues. Es wird das aktuelle Element von **args** über die Variable **i** ausgegeben, die danach um 1 erhöht wird. Dann wird die Schleife erneut gestartet. Das ganze Programm hat den Effekt, dass alle Daten aus dem Array **args** (und damit alle Übergabe-Parameter) auf dem Monitor ausgegeben werden.

> Warum heißt die Bedingung nicht wie folgt?
>
> ```
> while(i <= args.length)
> //Fehler!!!
> ```
>
> Immer daran denken: Weil Java ein Array von 0 an durchnummeriert! Die Schleife würde auch ausgeführt, wenn kein Parameter (also 0) existierte. Java quittiert dies mit einer Fehlermeldung.

Mehrdimensionale Arrays

Genau wie in C oder C++ ist es auch in Java möglich, Arrays über mehrere Dimensionen gehen zu lassen. Auch hier bedient sich Java einer ähnlichen Syntax, die ein mehrdimensionales Array als ein Array aus Arrays definiert. Wie kann man sich das also vorstellen? Eingentlich nicht anders als eine einfache Liste, die nicht aus Daten besteht, sonder wiederum aus Listen. Eine Liste die sozusagen weitere Listen verwaltet, denn: Ordnung muss sein ☺

Die Syntax ist sehr einfach. Für jede neue Ebene wird ein weiteres eckiges Klammerpaar hinter den Arraybezeichner geschrieben. Die einzelnen Klammern bezeichnen dabei je eine Dimension und enthalten die Anzahl der Elemente.

```
int mArray[][] = new int[3][4]
```

Diese Zeile erschafft ein zweidimensionales Array, dass aus zwei **int**-Arrays mit Längen von 3 und 4 Elementen besteht. Das Array kann also (3 * 4 =) 12 verschiedene **int**-Werte aufnehmen. Der Zugriff erfolgt über die bekannte Adressierung der Elemente.

```
mArray[1][2] = 17;
mArray[0][3] = 5,
```

Diese Zeilen setzen in der zweiten Zeile das dritte Element auf den Wert 17 und in der ersten Zeile den vierten Wert auf den 5. Wie gesagt zählt Java ab 0, also muss jeweils eins abgezogen werden. Der Adressraum erstreckt sich also von [0][0] bis [2][3], was bei genauem Durchzählen auch wieder 23 ergibt.

Bildlich kann man sich ein zweidimensionales Array wie ein Schachbrett mit entsprechend vielen Felder vorstellen. Die Adressierung der Elemente entspricht den Koordinaten eines Feldes.

Mehrdimensionale Felder funktionieren ganz genauso. Ein dreidimensionales Feld wird so initialisert:

```
int 3dArray[][][] = new int
[3][4][5];
```

Grafisch lässt sich das ganze mit einem Quader vergleichen, der aus (3 * 4 * 5 =) 60 einzelnen Würfeln besteht. In der vierten Dimension wird eine grafische Vorstellung schon schwieriger. Hier empfiehlt es sich, diese Arrays als einfache Datenkonstrukte zu sehen, die Daten zusammenfassen und kapseln.

Referenzdatentypen

Java unterscheidet zwischen primitiven und komplexen Datentypen. Was primitive Datentypen sind, haben wir geklärt, und auch mit den komplexen Datentypen haben wir uns etwas auseinander gesetzt, so dass es jetzt Zeit wird für einige Sätze Theorie. Wir haben in einem der letzten Kapitel gelernt, dass Arrays so wie Objekte behandelt werden – was allerdings nur die halbe Wahrheit ist. Arrays sind im Prinzip nichts anderes als Objekte der jeweiligen Klasse der primitiven Datentypen. Der logische Schluss ist, dass jedes Objekt (auch eines aus einer selbstgeschaffenen Klasse) als komplexer Datentyp verwaltet wird. Diese Feststellung wäre eigentlich nicht besonders aufregend – wenn Java nicht noch einen (recht intelligenten) Automatismus hätte. Im Gegensatz zu systemnahem Programmiersprachen wie C++ hat der Programmierer in Java keine direkte Möglichkeit, selber die Speicherverwaltung für das eigene Programm zu steuern. Java regelt diese Sache automatisch und nimmt ihm damit die Arbeit ab. Ob das gut oder schlecht ist, mag jeder für sich selbst entscheiden – wichtig ist zu wissen, nach welchem Regeln Java vorgeht.

Es wird grundsätzlich unterschieden zwischen den Prinzipen des **`Call-by-Value`** und **`Call-by-Reference`**, die jeweils eine eigene Art der Speicherpolitik beschreiben. **`Call-by-Value`**, also Aufruf über den Wert, besagt nichts anderes, als dass der Wert einer Variablen (etwa bei einer Parameterübergabe an eine Methode) unmittelbar weitergegeben wird. Dazu solltest du wissen, dass bei jedem Methodenaufruf eine Kopie aller betroffenen Variablen im Speicher angelegt wird. Das bedeutet natürlich auch, dass der benötigte Speicher zur Laufzeit der Methode stärker belastet wird also zur übrigen Programmausführung. Solange es sich dabei aber nur um einfache. d.h. primitive Datentypen handelt, hält sich dieser Speicherverbrauch in Grenzen, da jeweils nur wenige Byte belegt werden.

Anders verhält es sich bei komplexen Datentypen, die auf Grund ihrer Struktur relativ schnell große Speichermengen belegen können. Müssen hier ganze Objekte übergeben oder im Speicher kopiert werden, greift Java auf einen Trick zurück, der Resourcen spart: Statt die komplette Datenstruktur zu übergeben, wird die Adresse des komplexen Datentyps übergeben, so dass der Compiler weiss, wo im Speicher die Daten abgelegt sind. Das nennt man **`Call-by-Reference`**, also Aufruf über die Adresse.

Im Prinzip macht Java es so wie 90% unserer Mitmenschen: Statt die Telefonnummer der Freundin auswendig zu lernen, speichern wir sie im Handy und merken uns nur ihren Namen. Wir wissen, dass wir mit Namen und Handy die Nummer jederzeit abrufen können. Warum also die „komplexe" Information kopieren und in unserem Hirn speichern? ☺

Im normalen Programmieralltag ist diese Technik kaum zu bemerken, weil es fast egal ist, ob nur die Adresse oder alle Informationen in einer Variable gespeichert sind. Man erwischt Java bei dieser „Schummelei" mit diesem Programm:

```
public class referenzen
	{
	static void main(String
args[])
	{
dackel waldi = new dackel();
dackel fiffi = waldi;
waldi.Alter = 20;
System.out.println(fiffi.Alter);
	}
}
```

Wir erschaffen ein Objekt namens **`waldi`** aus unserer Klasse **`dackel`**. Dann erschaffen wir eine weitere Variable vom Typ **`dackel`**, ohne diesmal ein neues Objekt zu schaffen. Wir weisen **`fiffi`** einfach den Wert **`waldi`** zu. Dann wird der Objektvariable **`Alter`** von **`waldi`**(!) der Wert 20 zugewiesen. Bisher nichts neues – aber die letzte Zeile hat es in sich: Hier geben wir das Alter vom Objekt **`fiffi`**(!!!) aus, und siehe da, **`fiffi`** behauptet plötzlich, ebenfalls 20 Jahre alt zu sein. Hätte Java in Zeile 4 alle Daten kopiert, müsste **`fiffi`** hier schlicht das Standardalter von 1 (oder 0, je nach Dackelklasse) ausgeben. Da Java nur die Adresse übergeben hat, muss **`fiffi`** aber zwangsläufig auch 20 Jahre alt sein, da es sich um ein und denselben Hund handelt. Wir haben schließlich nur ein Dackelobjekt erschaffen, also kann es nur eins geben. Der Trick ist, dass unser Hund nun auf zwei verschiedene Namen hört.

Vererbung

Bevor wir uns auf das nächste Kapitel „Arbeiten mit Java“ stürzen, müssen wir uns mit einem wesentlichen Punkt der objektorientierten Theorie kümmern: Die Vererbung. In einem der ersten Kapitel über OOP haben wir dieses Prinzip schon angesprochen. Es geht um die Technik, aus einer bestehenden Klasse eine neue zu erschaffen, die Eigenschaften und Methoden der Mutterklasse übernimmt; letztere wird auch als Oberklasse oder Superklasse bezeichnet. Dieser Vorgang ist allerdings viel mehr als eine einfache Kopie der vorhandenen Daten in eine neue Klasse!

Es entsteht hier nämlich eine formale Verknüpfung zwischen den Klassen: Die neue oder abgeleitete Klasse nutzt die Eigenschaften und Methode so ähnlich, wie ein Benutzer auf eine Bibliothek zugreift. Die Daten werden nicht kopiert, sondern bei jeder Benutzung wird auf die Oberklasse zugegriffen. Dies ist ein ganz wesentlicher Punkt der OOP. Sobald nämlich die Mutterklasse geändert wird, ändert sich auch abgeleitete Klasse.

Klassen können beliebig oft voneinander abgeleitet werden. Auch bereits abgeleitete Klassen können neue Klassen „beerben“. Der so entstehende Vererbungsbaum kann also im Prinzip beliebig groß werden.

Möchtest du eine Klasse in Java aus einer bereits bestehenden ableiten, kannst du das mit dem Schlüsselwort **extends**:

```
public class rauhaardackel extends
dackel {}
```

Diese Zeile erschafft eine neue Klasse mit dem Namen **rauhaardackel**, die aus der Klasse **Dackel** abgeleitet wurde. Damit übernimmt **rauhaardackel** alle Eigenschaften und Methoden, die für die neue Klasse sichtbar sind mehr zur Sichtbarkeit auf Seite 31. Zur Erinnerung noch einmal die Klasse **dackel**, mit der wir arbeiten.

```
public class dackel
{
//Konstruktoren
public dackel()
{
      System.out.println("Geburt
eines Dackels!");
      this.Alter=1;
      this.Name="Dackel";
}
//Instanzmethoden
public void bellen()
{
System.out.println("WauWau!");
}
//Instanzvariablen
int Alter;
String Name;
}
```

Das folgende Programm, Vererbung genannt, beweist, dass auch der Rauhaardackel bellen kann.

```
public class vererbung
{
static void main(String args[])
{
    rauhaardackel wuffi = new
rauhaardackel();
    wuffi.bellen();
}
}
```

Diese Fähigkeit hat er von der Klasse **dackel** geeerbt. Java geht hier nach einem Prinzip vor. Zunächst wird die aufgerufene Methode in der eingenen Klasse gesucht. Gibt es sie hier nicht, wird in der nächsthöheren Ebene des Verzeichnisbaums, also in der Superklasse nach dieser Methode gesucht. Ist das ebenfalls ein Fehlschlag, geht es noch einen Schritt höher, bis die Methode gefunden ist. Das selbe gilt für Variablen.

Das Beispiel zeigt auch, dass der Konstruktor der Mutterklasse ebenfalls übernommen wurde. Jedes neue Objekt der Klasse **Rauhaardackel** wird mit dem Kommentar „Geburt eines Dackels!“ angekündigt.

Alle Klassen, die nicht von einer bestimmten Klasse abgeleitet wurden, also keine **extend**-Angabe haben, werden automatisch als Unterklasse von **Object** eingestuft. Diese „Mutter aller Klassen" steht an der Spitze der Java-Vererbungspyramide und ist sozusagen die Spitze des Eisbergs.

Die reine Weitergabe von Methoden und Variablen ist allerdings noch nicht alles. Ziel einer neuen Klasse ist natürlich die Erweiterung und Spezialisierung für bestimmte Anwendungsgebiete. Der Rauhaardackel unterscheidet sich z.B. in einer ganzen Reihe von Merkmalen, die in der Oberklasse **Dackel** nicht aufgeführt sind. So weisen wir ihm zum Beispiel die Methode **beissen()** und die Variable **Fellfarbe** zu.

```
public class rauhaardackel extends
dackel
{
public void beissen()
{
    System.out.println("Der Dackel
beisst!");
}
String Fellfarbe;
}
```

Diese Eigenschaften stehen neben den geerbten Eigenschaften jetzt ebenfalls in der Klasse **rauhaardackel** zur Verfügung – nicht aber in der Oberklasse **dackel**!

Im einzelnen sind folgende Dinge möglich:

- Methoden hinzufügen
- geerbte Methoden überschreiben
- Variablen hinzufügen

Das Überschreiben von Methoden werden wir im nächsten Kapitel besprechen.

Überschreiben von Methoden

Eine abgeleitete Klasse übernimmt alle sichtbaren Methoden und Variablen der Oberklasse. Unter Umständen kann es aber sein, dass die neue Klasse zwar bestimmte Methoden der Oberklasse übernehmen kann, aber Einzelheiten in der Funktionalität ändern muss. Also mus die betroffene Methode neu geschrieben oder besser: überschrieben werden. Hier wollen wir dem Bellen eines Rauhaardackels eine eigene Klangfärbung geben.

```
public void bellen()
{
System.out.println("BellBellBell!");
}
```

Diese Methode wird nun in die Klasse **rauhaardackel** geschrieben. Da sie den selben Namen hat wie die Methode in der Mutterklasse, wurde diese erfolgreich überschrieben.

```
public class vererbung
{
static void main(String args[])
{
    rauhaardackel wuffi = new
rauhaardackel();
    dackel bello = new dackel();
    wuffi.bellen();
    bello.bellen();
}
}
```

Dieses kleine Programm schafft sowohl ein Objekt **dackel** als auch ein Objekt **rauhaardackel**. Beide Objekte führen im nächsten Schritt die Methode **bellen** aus, welche hier aber zu unterschiedlichen Erbebnissen führt. **bello** bleibt seinem „WauWau" treu, **wuffi** aber greift nicht auf dieOberklasse zurück – obwohl er aus ihr abgeleitet wurde –, sondern benutzt die überschriebene Methode. Er bellt nun „BellBellBell". Nicht einfallsreich aber wohl charakteristisch.

Konstruktoren und Vererbung

Im ersten Kapitel über Vererbung sahen wir, dass eine abgeleitete Klasse den Konstruktor der Mutterklasse übernimmt. Also führt die Klasse **rauhaardackel** automatisch bei jeder Objekterstellung den Konstruktor der Klasse **dackel** aus, wie die Zeile „Geburt eines Dackels!" zeigt. Doch was passiert, wenn man der abgeleiteten Klasse einen eigenen Konstruktor spendiert? Wird der übergeordnete Konstruktor überschrieben? Mitnichten! Java ist konsequent: Nur Methoden mit demselbenNamen werden überschrieben, alles andere als eigene Methode betrachtet. Da der Konstruktor der Klasse **rauhaardackel** den selben Namen trägt wie die Klasse, wird das Problem anders gelöst. Probieren wir es aus:

```
public class rauhaardackel extends
dackel
```

```
{
//KONSTRUKTOR
public rauhaardackel()
{
    System.out.println("Noch ein
Rauhaardackel!");
}
public void beissen()
{
    System.out.println("Der Dackel
beisst!");
}
public void bellen()
{

System.out.println("BellBellBell!");
}
String Fellfarbe;
}
```

Hier noch einmal die komplette Klasse **rauhaardackel**, diesmal mit einem eigenen Konstruktor, der ebenfalls das entstandene Objekt kommentiert. Erschaffen wir jetzt ein **rauhaardackel**-Objekt:

```
rauhaardackel wuffi = new
rauhaardackel();
```

... überrascht uns Java mit folgender Ausgabe:

```
Geburt eines Dackels!
Noch ein Rauhaardackel!
```

Was ist passiert? Es gibt nur eine Erklärung: Die Erschaffung eines Objekts aus einer abgeleiteten Klasse führt dazu, dass zuerst der Konstruktor der Mutterklasse ausgeführt wird und dann der eigene Klassenkonstruktor. Auch bei größeren Vererbungsreihen besteht diese Reihenfolge. Zuerst führt Java den obersten Konstruktor aus und wandert dann langsam nach unten.

Dieses Prinzip gilt übrigens auch für den Destruktor, allerdings in umgekehrter Reihenfolge.

Mehrfachvererbung

Anfänglich wurde bereits gesagt, dass Java offiziell keine Mehrfachvererbung unterstützt, um eine chaotische Klassenpolitik zu vermeiden. Diese Entscheidung ist nicht ganz umstritten – in vielen Fällen bietet die Mehrfachvererbung eine phantastische Möglichkeit, Probleme der Programmierung einfach und realitätsnach zu lösen.

Auch wenn wir aus Gründen der Verständlichkeit im folgenden Kapitel unserem Dackel-Beispiel nicht treu bleiben, halten wir uns weiter an die Welt der Vierbeiner mit feuchten Nasen.

Stell dir vor, du schriebst die Klasse Wolf, die das Verhalten der Vorfahren unserer treuen Freunde perfekt simuliert und auf deren Basis einige Programme zur Erforschung der Tierwelt laufen. Nun bekommst du den Auftrag, die Veränderungen im Verhalten der Wölfe im Zuge der Domestizierung durch den Urmenschen in einem Programm darzustellen. Was machst du?

Die Methoden und Variablen der Klasse Wolf stellen nur Werkzeuge für das wilde und unabhängige Leben der Tiere dar. Der Einfluss von Menschen wurde hier nicht berücksichtig. Eine Möglichkeit wäre, die Klasse zu erweitern oder vollständig umzuschreiben und so die neue Funktionalität zu integrieren. Allerdings ist die Gefahr von Fehlern und unsauberer Programmierung sehr groß, denn es gibt kaum etwas schlimmeres alsden Umbau eines bereits fertigen Programms.

Die eleganteste Lösung bietet die Mehrfachvererbung, die in diesem Fall die Erstellung der neuen Klasse Haushund aus den Klassen Wolf und Haustier erlauben würde, welche eine Reihe von allgemeinen Methoden und Werten zum Verhalten domestizierter Tiere bereit stellt.

Leider läßt Java diese Technik aus den bekannten Gründen nicht zu. Doch gibt es hier ein Hintertürchen, das eine saubere und objektorientierte Konstruktion für unser Problem ermöglicht. Das Zauberwort lautet hier:

Schnittstellen

In Java lassen sich Schnittstellen definieren, die in anderen Programmen implementiert werden können. Diese Technik erlaubt die Umgehung des Verbots der Mehrfachvererbung ohne einen Verzicht auf übersichtliche Programmierung.

Eine Schnittstelle ist im Prinzip dasselbe wie eine Klasse, nur wird sie nicht mit dem Schlüsselwort **class**, sondern mit dem Wort **interface** definiert. Sie besteht aus blanken Definitionen von Methodenköpfen, die das Verhalten der Schnittstelle nach außen repräsentieren, ohne aber eine geeignete Funktionalität zu geben. Diese Aufgabe fällt der Klasse zu, die dies Interface implementiert.

```
public interface haustier
{
public void lernen(String befehl);
public void ausDerHandFressen();
public void amFußendeSchlafen();

boolean wasserscheu = true;
boolean frisstMenschen = false;
}
```

Hier haben wir ein kleines Interface mit dem Namen **haustier** definiert, das aus drei Methoden und zwei Variablen besteht, die wohl typisch für jedes Haustier sind. Möchtest du aus den Klassen **wolf** und **haustier** die Klasse **haushund** erschaffen, benötigst du neben den schon bekannten Befehlen das Schlüsselwort **implements**:

```
class haushund extends wolf
implements haustier
```

Über **extends** wird die Klasse **wolf** eingebunden, die alle Eigenschaften und Methoden der Wölfe mitbringt, und über **implements** wird die Schnittstelle **haustier** hinzugefügt, die die Charakteristik eines Haustieres enthält. Damit ist die neue Klasse fast komplett. Fehlt noch eine Kleinigkeit: Die Definition der Methodenköpfe aus der Schnittstelle der Haustiere. Java weiß jetzt, welche Methoden wir im Interface **haustier** definiert haben – noch weiß es aber nicht, wie diese genau funktionieren. Darum wird auch jeder Versuch, die neue Klasse **haushund** zu kompilieren, mit einer Fehlermeldung bestraft.

Also müssen wir nun die implementierten Methoden definieren.

```
class haushund extends wolf
implements haustier
{
public void lernen(String befehl)
{
System.out.println("Der Hund lernt
den Befehl: " + befehl);
}
public void ausDerHandFressen()
{
System.out.println("Der Hund frisst
dir aus der Hand!");
}
public void amFußendeSchlafen()
{
System.out.println("Der Hund rollt
sich zu deinen Füßen zusammen!");
}
}
```

Im Prinzip überschreiben wir schlicht die Methoden aus der Schnittstelle und geben ihnen so die nötige Funktionalität. Das Programm an sich sollte keine aufregenden Neuerungen bergen.

abstract und final

Will man Klassen über die üblichen Schlüsselwörter hinaus beschreiben, kann man ihnen bestimmte Eigenschaften zuordnen. Im Normalfall ist das nicht nötig, aber es wird sicher einmal der Punkt kommen, an dem man diese Technik braucht. Java bietet dafür zwei Schlüsselwörter an, die uns das Leben erleichtern.

Durch die Angabe von **abstract** wird eine Klasse deklariert, deren Definition noch nicht abgeschlossen ist. D.h. es gibt einige Methoden in dieser Klasse, die noch nicht geschrieben wurden und deshalb nur einen Kopf, aber noch keine Funktion besitzen. Eine abstrakte Klasse muss in Java abgeleitet werden, damit die fehlenden Methoden durch Überschreiben hinzugefügt werden können. Eine direkte Instanzierung – also die Erstellung eines Objekts – ist nicht möglich.

Das Konzept der abstrakten Klasse sollte aus dem Kapitel über die Schnittstellen bekannt sein, da alle Methoden einer Schnittstelle abstrakt sind.

Eine besondere Kennzeichnung ist in diesem speziellen Fall allerdings nicht nötig, da Schnittstellen automatisch abstract sind.

Das Gegenteil einer abstrakten Klasse ist die Kennzeichnung mit dem Schlüsselwort **`final`**. Es zeigt an, dass die Klasse in ihrer Definition abgeschlossen ist, und verhindert eine weitere Ableitung dieser Klasse. Es ist also nur möglich, Objekte aus dieser Klasse zu erstellen; neue Klassen dürfen nicht aus einer finalen Klasse abgeleitet werden.

Diese Technik ist sehr nützlich, wenn du verhindern willst, dass bestimmte Methoden aus deiner Klasse überschrieben werden können.

Innerhalb der javaeigenen Klassen – der Java-API – gibt es einige wenige finale Klassen, wie zum Beispiel die Klasse **`java.lang.Math`**. Hier werden viele mathematische Methoden und Konstanten definiert, die nicht überschrieben werden können. So ein Vorgehen wäre auch wenig sinnvoll, denn eine Sinusberechnung läßt in der Regel wenig Platz für kreativen Spielraum eigener Methoden.

Arbeiten mit Java

Nach soviel Einführung haben wir es geschafft. Alle wesentlichen Grundlagen sind besprochen und wir können uns nun auf die wirklichen Leckerbissen von Java stürzen. Die nächsten Kapitel beschreiben Schritt für Schritt einige wichtige Punkte der Javaprogrammierung, die für einen Programmierer unerläßlich sind.

Die Anweisung import

Die Spezifikation von Paketnamen für Klassen ist eine sehr nützliches Werkzeug für die Programmierung, kann aber auch schnell ziemlich viel Tipparbeit mit sich bringen, wenn plötzlich statische Methoden aus Konstruktionen wie **tiere.hund.dackel.getAnzahl()** gezogen werden müssen. Java bietet für eine solche Situation das Schlüsselwort **import** an, das es erlaubt, Klassen im Sourcecode über einen gekürzten Namen ansprechbar zu machen:

```
import tiere.hund.dackel;
```

Mit dieser Anweisung, die am Anfang des Sourcecodes steht, lässt sich die Methode **getAnzahl()** direkt ansprechen.

Die zweite Variante von Import ist noch eine Spur bequemer. Über das Wildcardzeichen * werden sämtliche Klassen eines Paketes auf einmal importiert.

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

Diese Zeile importiert alle Bestandteile des Paketes **java.lang** ins Programm und ermöglicht es, sie direkt anzusprechen.

Das Paket java.lang

Das erste Paket, das wir besprechen wollen, enthält die grundsätzlichen Werkzeuge zur Arbeit mit der Sprache Java. **lang** steht hier auch für das englische Wort language und unterstreicht so die Wichtigkeit der folgenden Klassen. Diese Liste ist natürlich keineswegs vollständig, aber ich denke, ich habe die wichtigsten Punkte herausgesucht. Dies ist das einzige Paket, dass vom Javacompiler *automatisch* in ein Programm importiert wird. D.h. es ist nötig, das Paket per Hand über **import** verfügbar zu machen.

Die Klasse Math

Die javaeigene Klasse **Math** ist eine finale Klasse, die nur aus Klassenmethoden und und mathematischen Konstanten wie **pi** oder der eulerschen Zahl **e** besteht.

Ein Aufruf aus der Klasse sieht so aus:

```
System.out.println(Math.E);
//Die Eulersche Zahl
Umfang = 2 * radius * Math.PI;
//Umfang eines Kreises
zufall = Math.random();
//Zufallszahl zwischen 0 und 1
int x = Math.round(7.54234);
//rundet zu Ganzahlen
```

Math hat natürlich noch eine ganze Menge mehr zu bieten, aber das würde hier sicher zu weit führen. Benötigst du mathematische Funktionen, empfehle ich dir die Java-API.

Die Klasse Object

Die Klasse **Object** ist die Mutter aller Klassen. Alle anderen Klassen werden aus dieser Superklasse abgeleitet und können deshalb eine Reihe von Methoden nutzen, die in **Object** deklariert wurden.

Die Methode **getClass()** liefert die Klasse eines Objekts zurück und gibt so Auskunft über die Herkunft einer Datenstruktur.

Die Methode **equals()** vergleicht zwei Objekte auf und gibt **true** oder **false** zurück. Diese Methode ist notwendig, da ein einfacher Vergleich über den Operator **= =** nur die Adressen im Speicher vergleicht. Siehe dazu auch das Kapitel über komplexe Datentypen.

Die Klasse Runtime

Die Klasse Runtime enthält eine Reihe von Methoden, die es erlauben, unmittelbar mit dem System des Rechner zu komunizieren. Um die javatypische Plattformunabhängigkeit zu wahren, ist es allerdings nötig, ein Objekt aus der Klasse **Runtime** zu erstellen. Dies wird automatisch über die Klassenmethode **getRuntime()** erledigt, die sich auf die systemspezifischen Umstände einstellt:

```
Runtime mySystem =
Runtime.getRuntime();
```

Nun haben wir das Objekt **mySystem**, über das wir alle Methoden ausführen können. Zwei der interessantesten Methoden sind **freeMemory()** und **totalMemory()**, die über den Speicherzustand unterrichten. Über eine Kombination der beiden lässt sich der genaue Speicherverbrauch des Programms oder einzelner Objekte erfahren.

Eine weitere wichtige Methode ist **exit()**, die den Interpreter veranlasst, das Programm sofort zu beenden. Diese Methode muss mit einem definierten integren Parameter aufgerufen werden, der über die Art der Programmbeendigung Auskunft gibt. Normalerweise ist es so, dass 0 einen korrekten Programmablauf signalisiert und jede andere Zahl einen Fehler anzeigt. Den genauen Fehlercode sollte man aber vorher festlegen.

Wer die Funktion **System()** aus C kennt, der findet in Java mit **exec()** das Gegenstück. Mit dieser Methode ist es möglich unabhängig vom Interpreter Systemprogramme zu starten. Allerdings verlangt Java für die Nutzung dieser Methode eine Ausnahmenbehandlung, die wir erst auf Seite 56 kennenlernen werden. Dann allerdings ausführlich...

Die Klasse String

Den Datentyp **String** haben wir in einigen unserer Programme schon verwendet, ohne uns groß Gedanken darüber zu machen. Ein String repräsentiert in Java eine Verkettung von Buchstaben oder Zeichen. Da ein String kein primitiver Datentyp ist, wird er immer über die Klasse String erzeugt, und zwar dann, wenn der Compiler auf ein doppeltes Anführungszeichen trifft („...“). Das alles geschieht jedoch automatisch, so dass wir den String wie gehabt recht einfach handhaben können.

Java bietet für Strings den Operator + an, mit dem mehrere Stringobjekte zusammengefasst werden können.

```
String s1 = "Hallo";
String s2 = "Java!";
String s3 = s1 + " " + s2;
System.out.println(s3);
```

Der String s3 wird aus den Strings **s1**, einem Leerzeichen und **s2** zusammengesetzt. Über **println()** werden dann die Worte „Hallo Java!“ ausgegeben.

Darüber hinaus bietet Java eine Reihe von Methoden, die die Arbeit mit Strings leichter machen. Die Methode **charAt()** gibt einen Buchstaben aus dem String zurück, der über einen integren Parameter bestimmt ist. Der Rückgabewert ist von der Form **char**. Das genaue Gegenteil dieser Methode ist **indexOf()**. Mit ihr kann die Position von Buchstaben oder Buchstabenketten in einem String festgestellt werden. Mit dieser Information können dann z.B. folgende Methoden „gefüttert“ werden:

substring() gibt einen neuen String zurück, der aus einem Teil des alten Strings besteht. Anhand der Parameter kannst du so ein Wort auseinanderschneiden. Die Länge eines Strings bestimmst du genau wie bei einem Array über **length()**.

```
String text = "Das ist ein
konstanter String!";
int laenge = text.length();
int pos = text.indexOf("String");
String text2 = text.substring(pos,
laenge);
System.out.println(text);
System.out.println(text2);
```

Das kleine Beispiel zeigt, wie mit den Stringmethoden gearbeitet wird. Das Ziel, das Wort „String!“ am Ende des Satzes herauszuschneiden, wird über einige Methoden erreicht, die uns zuerst Informationen über den Strings geben (Länge, Position des Wortes). Dann kann einfach per **substring()** der Wort herausgetrennt werden.

> Ein String ist nach der Erschaffung nicht mehr veränderbar! Es handelt sich also um konstante Zeichenfolgen, die sich zur Laufzeit des Programms nicht mehr ändern. Veränderbare Strings werden über die spezielle Klasse **StringBuffer** geschaffen.

Die Klasse Stringbuffer

Die Klasse **StringBuffer** ist die flexible Variante der Klasse **String**; sie dient der Behandlung von Zeichenketten, die veränderbar sein müssen. D.h. während der Laufzeit des Programms kann der Wert der Variablen neu gesetzt werden. Die Klasse **StringBuffer** hat allerdings auch einige nützliche Methoden, die die Arbeit mit den Strings etwas erleichtern.

Die Methode **setCharAt()**,sozusagen das Gegenstück von **charAt()**,erlaubt die Änderung einen bestimmten Buchstabens im String. Wer es brutaler mag, kann sich mit **delete()** anfreunden, denn diese Methode löscht gleich ganze Teile eines Strings heraus. Als Parameter werden Start- und Endpunkt angegeben. Noch eleganter macht es **replace()**. Diese Methode ersetzt einen Teilbereich eines Strings durch einen anderen. Die Methode **insert()** dagegen setzt einen String an eine Stelle des Objektes ein, ohne etwas zu löschen.

```
StringBuffer buffer = new
StringBuffer("Dieses KnowWareheft
behandelt die Sprache C++ fuer
Fortgeschrittene.");
buffer.replace(42, 45, "Java");
buffer.delete(52, buffer.length()-
1);
buffer.insert(buffer.length()-1,
"Einsteiger");
System.out.println (buffer);
```

In diesem Beispiel zerpflücken wir einen String, um ihn dann nach und nach wieder mit neuen Informationen zu füllen. Wichtig ist, dass – im Gegensatz zu **String** – ein **Stringbuffer ()** über den **new**-Operator erzeugt werden muss. Die Handhabung bleibt ansonsten allerdings gleich.

Die Klasse System

Die Klasse **System** definiert ähnlich wie **Runtime** Methoden, die als Schnittstelle zum System verwendet werden. Eine der wichtigsten Komponenten dieser Klasse ist die Behandlung von Datenströmen eines Systems. Unterschieden werden dabei die Standardeingabe (**in**), die Standardausgabe (**out**) und die Fehlerausgabe (err), die über Bildschirm oder Tastatur erfolgt. Die Standardausgabe haben wir bereits über die Methode **println()** kennen gelernt, die es erlaubt, Strings auf dem Bildschirm auszugeben.

```
System.out.println(„Hallo Java");
```

Die umgekehrte Richtung wird über **in** an Stelle von **out** realisiert. Allerdings wird hier von Java eine Fehlerbehandlung gefordert, die eventuelle Ausnahmen abfängt. Diese Technik wird im folgenden Kapitel genau besprochen, hier schon mal die Syntax für das Einlesen von Daten:

```
byte[] b = new byte[100];
System.in.read(b);
```

Alle Daten, die Java über einen Stream einliest, werden im Byteformat geliefert. Damit wir die Daten zwischenspeichern können, brauchen wir also ein Byte–Array in der entsprechenden Größe. Ist dieses erstellt, können wir über die Methode **read()** Daten vom User abfragen. Als Argument für **read()** übergeben wir **b**, in dem die Daten gespeichert werden. Das Programm stoppt hier und verlangt eine Eingabe über die Tastatur, die mit ENTER beendet wird. Erst dann wird das Programm fortgesetzt. Es werden nur so viele Daten im Array gespeichert wie der zugewiesene Speicherplatz zulässt – in unserem Fall also 100 kb.

> So wie das Programm oben geschrieben wurde, funktionieren es nicht. Java bemängelt die fehlende Behandlung der **IOException**. Wie diese gesetzt wird, erfährst du auf Seite 56.

Die Fehlerausgabe **err** funktioniert im Prinzip so wie **out** und ermöglicht die Aisgabe von Daten auf dem Monitor. Der Unterschied liegt im Detail und ist für unsere Zwecke auch mehr theoretischer Natur. Meldungen, die über **err** ausgegeben werden, sind für das System grundsätzlich Fehlermeldungen. In bestimmten Umgebungen (UNIX, Linux) werden diese in sogenannten Log-Files automatisch gespeichert und zur Auswertung aufgehoben. Das ist bei unserem Programm allerdings nicht der Fall. Ein guter Stil verlangt trotzdem, Fehlermeldungen über *err* auszugeben:

```
System.err.println("Schrecklicher
Fehler!!!");
```

Diese Meldung erscheint auch auf dem Schirm.

Das Paket java.util

Das Paket **java.util** enthält eine Reihe von Werkzeugen, die sich oft als recht nützlich erweisen. Implementiert wird das Paket über

```
import java.util.*;
```

Die Klasse Date

Die Klasse **Date** stellt eine Reihe von Werkzeugen für die Arbeit mit Datums- und Zeitfunktionen bereit. Um ein Datumsobjekt mit dem aktuellen Datum zu erzeugen, reicht es den Standardkonstruktor zu verwenden:

```
Date myDate = new Date();
```

Java stellt einige Methoden zur Verfügung, die die Arbeit mit diesem Objekt erlauben. Die simpelste Möglichkeit, den Wert des Objektes auszugeben, ist die Methode **toString()**. Sie wandelt das Datum in einen String um.

```
String datum = myDate.toString();
```

Dabei erhält man folgende Form:

```
Thu Sep 06 13:10:14 GMT+02:00 2001
```

Über **getTime()** erhält man den sogenannten UNIX-Timestamp, der die genaue Anzahl der Sekunden angibt, die seit dem 01.01.1970 vergangen sind. Diese Zahl ist ein Standard in der Zeitberechnung auf allen Rechnern.

Zwei weitere nützliche Methoden sind **before()** und **after()**, die als Parameter jeweils ein Datumsobjekt brauchen. Sie vergleichen das eigene Datumsobjekt mit dem Parameter und geben **true** oder **false** zurück, je nachdem ob es das Datum vor oder nach dem Vergleichstag liegt.

Erweiterte Funktionen rund um das Datum gibt es in der Klasse **Calendar** bzw. **DateFormat** in der Java-API.

Die Klasse Random

Die Klasse **Random** stellt einen Generator zur Erzeugung von Zufallszahlen zu Verfügung. Der Standardkonstruktor liefert eine Reihe von Zufallszahlen, die über verschiedene Instanzmethoden ausgelesen werden. Die Grundlage der Zahlen ist dabei die aktuelle Systemzeit.

```
Random zz = new Random();
```

Das Objekt **zz** enthält jetzt eine große Menge von Zufallszahlen, die über die folgenden Methoden abgerufen werden können:

Methode	Bereich	Typ
nextInt()	-2 ^31 bis 2^31-1	int
nextLong()	-2^63 bis 2^63-1	long
nextFloat()	0.0 bis 1.0	float
nextDouble()	0.0 bis 1.0	double

Im Programm würde das so aussehen:

```
int zahl1 = zz.nextInt();
long zahl2 = zz.nextLong();
System.out.println(zz.nextDouble());
System.out.println(zz.nextFloat());
```

Jeder Methodenaufruf produziert eine neue Zufallszahl des jeweiligen Typs aus dem Objekt. Die Methode **nextGaussian()** gibt eine Zufallszahl zurück, die nach dem Gauß-Verfahren ermittelt wurde.

Die Klasse Stack

Die Klasse **Stack** stellt einen Datenspeicher zur Verfügung, in dem Objekte abgelegt werden. Das Speicherprinzip folgt dabei der Regel *First in – Last out*. Stell dir das Ganze als eine Art Tellerstapel vor: Du legst den ersten Teller ab und beginnst damit deinen Stapel. Dann folgt der zweite Teller, dann der dritte usw. Möchtest du dann den ersten Teller wieder haben, musst du zuerst die übrigen Teller herunter nehmen, also zuerst den, den du als letzten aufgelegt hattest.

Ein Stack funktioniert nach demselben Prinzip. Zunächst benötigst du ein Objekt der Klasse:

```
Stack myStack = new Stack();
```

Mittels einiger Objektmethoden kannst du die Daten innerhalb des Stack verwalten. Die Methode **push()** legt ein Objekt auf dem Stack ab.

```
myStack.push("Hund");
```

In diesem Fall ist es ein einfaches Stringobjekt mit dem Wert „Hund“. Es ist egal, welche Objekte in einem Stack liegen und in welcher Reihenfolge das geschieht, also kann im nächsten Schritt ein Objekt der Klasse **dackel** auf dem Stack landen:

```
dackel waldi = new dackel();
myStack.push(waldi);
```

Als nächstes folgt wieder ein Stringobjekt mit dem Namen **Teller**:

```
myStack.push("Teller");
```

Möchtest du die gespeicherten Objekte wieder aus dem Stack herausholen, benutzt du die Methode **pop()**, die das oberste Objekt des Stack zurückgibt.

```
myStack.pop();
```

Diese Anweisung würde im aktuellen Stack den String „Teller“ zurückgeben. Gleichzeitig würde das zurückgegebene Objekt gelöscht, so dass ein anderes an seine Stelle rückt.

Möchtest du das Löschen verhindern, benutzt du die Methode **peek()**, die den obersten Wert nur ausliest.

```
myStack.peek();
```

Dadurch wird natürlich der oberste Platz nicht geändert, so das die restlichen Daten weiterhin gesperrt bleiben.

Ausnahmen (Exceptions)

Java ist eine der sichersten Sprachen, was die Behandlung von nicht vorhersehbaren Fehlern angeht. Der Programmierer wird durch die Syntax von Java in eine Struktur gezwungen, die jede erdenkliche Möglichkeit eines außerplanmäßigen Programmabbruchs verhindert. Ermöglicht wird dies durch eine rigide Sicherheitspolitik, die auf dem System von Ausnahmen (exceptions) beruht. Eine Exception wird von Java immer dann erzeugt – die korrekte Syntax lautet *geworfen* –, wenn während der Laufzeit des Programm ein Fehler auftritt. In diesem Fall tritt eine Ausnahmebehandlung in Kraft, die auf den Fehler reagieren kann und so einen Programmabsturz verhindert. C-Programmierer kennen die **try-catch**-Syntax vielleicht schon. Der große Unterschied zu C ist, dass Java diese Behandlung an vielen Stellen im Programm zwingend verlangt.

```
try{
Hier werden unsichere Anweisungen
ausgeführt!
}
catch(ExeptionKlasse e)
{
Falls ein Fehler Auftritt, kann
diese hier behandelt werden.
}
finally{
Die her stehenden Anweisungen werden
auf jeden Fall ausgeführt!
}
```

Eine Ausnahmenbehandlung besteht aus mindestens zwei, optional drei Programmteilen. Der erste Teil ist der *try-Block*: Hier stehen eine oder mehrere Anweisungen, die Exceptions werfen können. Java *versucht* hier sozusagen diese Anweisungen auszuführen. Tritt ein Fehler auf, wird er von Java abgefangen, und eine der folgenden *catch-Anweisungen* tritt in Kraft. Java definiert eine Reihe von möglichen Fehlern, die in der **catch**-Anweisung genannt werden müssen. Je nach Anzahl der Anweisungen können zwei oder noch mehr verschiedene Ausnahmen auftreten, von denen jede in einer eigenen **catch**-Anweisung abgefangen werden muss. Ein typisches Beispiel ist das Öffnen und Schreiben von Java-externen Programmen. Der Versuch, ein nicht existentes Programm zu starten, wirft eine **IOExecption** aus.

Eine Exception ist im Prinzip eine Klasse, die je nach Fehlertyp Informationen der Ausnahme speichert und so eine Weiterverarbeitung zulässt. Dazu ist es nötig, für jeden Fehlertyp ein Objekt zu erschaffen, mit dem in der **catch()**-Anweisung gearbeitet werden kann.

Der Anweisungsblock nach **final** wird in jedem Fall ausgeführt, unabhängig davon, ob ein Fehler auftrat ist oder nicht. Dieser Block ist optional und muss nicht eingesetzt werden.

Die Ausnahmenbehandlung von Java ist ein sehr mächtiges Werkzeug, das unerwünschte Effekte in einem Programm elegant umschifft.

Um das Prinzip der Ausnahmenbehandlung an einem Beispiel zu besprechen, sehen wir uns hier die Methode **exec()** der Klasse **Runtime** an. Wie wir bereits gesehen haben, lassen sich mit **exec()** Java-unabhängige Programme starten, die parallel zum Interpreter laufen. Dieses Vorgehen ist allerdings nicht ganz risikolos, da Java die Existenz und vor allem die Wirkung des fremden Programms nicht abschätzen kann. Zu Recht wird hier eine Ausnahmebehandlung gefordert.

```
import java.io.*;
public class trycatch
{
static void main(String args[])
{
Runtime mySystem =
Runtime.getRuntime();
try{
      mySystem.exec("notepad.exe");
}
catch(IOException e)
{
    System.err.println("Es ist ein
Fehler aufgetreten: " + e);
}
finally
{
System.out.println("Kritischer
Bereich wurde ausgeführt!");
}
}
}
```

Dieses kleine Programm hat die Aufgabe, das Java-externe Windowsprogramm Notepad zu starten. Da Java nicht weiß, ob es ein solches Programm überhaupt gibt oder was es macht, müssen wir eine Ausnahmebehandlung einbauen. Der mögliche Ausnahmefehler der Methode **exec()** hört auf den schönen Namen **IOExceptions**. Versuchst du dieses Programm ohne **try-catch** zu bauen, macht dich Java mit einer Fehlermeldung auf diesen Umstand aufmerksam.

Wir beginnen damit, über die Klassenmethode **getRuntime()** ein Objekt der Klasse **Runtime** zu erstellen. Im nächsten Schritt möchten wir **exec()** benutzten, müssen dies aber mit einer **try**-Anweisung schützen. Der eigentliche Aufruf des Windows-Editors Notepad wird in Anführungszeichen als Parameter an **exec()** übergeben. In der **catch**-Anweisung wird dann die eigentliche Ausnahmebehandlung angegeben, die sich in unserem Fall auf eine einfache Fehlermeldung beschränkt. Wichtig ist, dass in der runden Klammer direkt hinter **catch** die Art der Ausnahme angegeben wird. Dazu ist es nötig ein Objekt der Ausnahmeklasse zu erzeugen, dass ich einfach **e** genannt habe. **e** speichert die Art und Auswirkung des Fehlers und wird in diesem Programm gemeinsam mit der Fehlermeldung ausgegeben. Auch eine Log-Datei wäre denkbar, die alle Fehlermeldungen eines Programms speichert und überschaubar archiviert.

Der letzte Punkt ist der **finally**-Block, der in jedem Fall ausgeführt wird. Hier wird eine Statusmeldung ausgegeben, die das Programm abschließt.

Jetzt gehts ans Ausprobieren: Hast du das Programm Notepad auf deinem Rechner hast – und davon gehe ich aus! –, sollte es sich mit dem Programmstart öffnen und ein leeres Editorfeld zeigen. Existiert das Programm nicht, erscheint die Fehlermeldung aus der **catch**-Anweisung. Um diese Fehlermeldung zu provozieren, reicht es, anstelle von NOTEPAD.EXE ein anderes Programm wie GIBTESNICHT.EXE anzugeben. In jedem Fall endet das Programm mit der Meldung „Kritischer Bereich wurde ausgeführt!“. Computer lügen nun mal nicht....

> Java bietet eine Reihe von Möglichkeiten, die fast alle mit einer eigenen Exception behandelt werden. Es ist natürlich möglich, alle auswendig zu lernen – oder noch schlauer in der Java-API nachzuschauen, aber es geht auch leichter. Versuchst du nämlich das entsprechende Programm ohne Ausnahmebehandlung zu starten, liefert der Javacompiler die fehlende Exception frei Haus.

Eigene Ausnahmen erzeugen (throw)

In Java ist es möglich, durch die Anweisung **throw** eine eigene Ausnahme zu erzeugen. Java erlaubt so, fast jede Situation in einem Programm zu kontrollieren und je nach Sachlage darauf zu reagieren. Ein simples Beispiel ist die Wurzelberechnung von Zahlen:

```
class unsicher
{
public static void
unsichereMethode(int zahl)
{
    if (zahl < 0)
    {throw (new
ArithmeticException("Schlimme
Ausnahme ist aufgetreten!"));}
    else
    {System.out.println("Die Zahl
lautet " + zahl + ". Die Wurzel der
Zahl ist " + Math.sqrt(zahl) + "
!");}
}
}
```

Über den Operator **new** wird ein neues Objekt der Ausnahmeklasse **ArithmeticException** geschaffen, das die Fehlermeldung „Schlimme Ausnahme ist aufgetreten!“ liefert. Diese wird über die Anweisung **throw** geworfen, wenn die Variable **zahl** kleiner als 0, die Wurzelberechnung also nicht möglich ist. Andernfalls gibt die statische Methode eine Meldung mit dem Ergebnis aus.

Wird diese Methode verwendet:

```
try{
unsicher.unsichereMethode(-1);
}
catch(ArithmeticException e)
{
    System.err.println("Fehler: " +
e);
}
```

... kann sie einfach über einen **try-catch**-Block abgefangen werden. Sicher nicht die eleganteste Methode, negative Zahlen zu melden – dafür aber wohl eine der unbedingt kompliziertesten.

Multitasking mit Threads

Multitasking ist ein Wort, dass in den letzten Jahren recht häufig zu hören ist. Es beschreibt die Fähigkeit eines Systems, mehrere Aufgaben oder Tasks scheinbar parallel zu bearbeiten. Warum scheinbar? Die meisten Heimcomputer arbeiten auf der Basis eines einzelnen Prozessors, der alle Aufgaben vom System zugewiesen bekommt. Technisch ist ein Prozessor nicht in der Lage, mehr als eine Aufgabe gleichzeitig zu bearbeiten, da seine ganze „Aufmerksamkeit“ jeweils nur einer Berechnung gilt. Der Trick von Multitaskingsystemen ist der schnelle Wechsel zwischen verschiedenen Aufgaben, so dass der Eindruck ensteht, der Prozessor würde verschiedene Dinge gleichzeitig tun. In Wirklichkeit nutzt er z.B. die Zeit, die ein Programm braucht, um Daten auf die Festplatte zu schreiben, für die Bearbeitung einer anderen Aufgabe. Gesteuert wird dieser Wechsel vom Betriebssystem, so dass jederzeit eine optimale Auslastung des Prozessors gewährleistet ist. In Systemen mit zwei oder mehr Prozessoren ist hingegen echtes Multitasking möglich, da die verschiedenen Aufgaben von den Prozessoren tatsächlich gleichzeitig bearbeitet werden. Für den Enduser ist der Unterschied meist aber nur theoretischer Natur, da der Wechsel nicht nachvollziehbar und vor allem nicht steuerbar ist.

Was ist ein Thread?

Grob gesagt beschreibt ein Thread eine der vielen Aufgaben, mit den sich ein Prozessor jeweils befasst. Jedem Thread wird durch das Betriebssystem eine ID zugewiesen, anhand derer er identifiziert wird. Der Prozessor springt zwischen den verschiedenen Threads hin und her und bearbeitet sie abwechselnd. Bisher bestanden Programme nur aus einem einzelnen Thread, den der Prozessor einfach abarbeiten konnte. Es ist allerdings ebenso gut möglich, innerhalb eines Programmes zwei verschiedene Threads zu schaffen, die quasi parallel nebeneinander laufen. Das klingt zwar kompliziert, ist es aber nicht, da die Drecksarbeit vom Betriebssystem übernommen wird.

Aber wozu dient das Ganze? Und was haben Threads eigentlich in einem Einsteigerheft zu suchen?

Nun, es gibt zwei Gründe: Der erste ist, dass Java das System der Threadverwaltung hervorragend anwendet, so dass es selbst Einsteigern nicht schwer fällt, mit Threads zu arbeiten. Der andere Grund ist, dass wir Threads brauchen, sobald wir uns etwas mit der Programmierung von Applets auseinandersetzen. Applets sind Programme mit eine grafischen Benutzeroberfläche, wie jedes Programm unter Windows. Sobald wir über die einfache Reaktion auf Usereingaben hinausgehen, etwa für eine Animation, erfordert das die Arbeit mit mehreren Threads – einem für die Animation und den anderen für die Kommunikation für dem User. Doch dazu später mehr.

Arbeiten mit Threads

Die Klasse **`Thread`** ist ein Bestandteil des Pakets **`java.lang`**, dass automatisch in jedes unserer Programme eingebunden ist. Soll eine Klasse multitaskingfähig gemacht werden, gibt es verschiedene Wege. Der hier beschriebene Weg ist der gängigste und wird in der Regel auch bei allen Appletprogrammen eingesetzt.

Der erste Schritt ist immer das Erstellen einer neuen Klassen über die Schnittstelle **`Runnable`**. Sie erlaubt die Nutzung des Threadobjekts und erstellt eine neue Methode, die auf den Namen **`run()`** hört. Diese Methode, die den eigentlichen Code des Threads enthält, muss überschrieben werden.

```
public class multi implements
Runnable
```

Der nächste Schritt bezieht sich auf die Methode **`main()`**, der hier nur eine Nebenrolle zufällt. Ein Thread kann nur über ein Objekt einer Klasse gestartet werden, darum beschränkt sich die statische main() Methode auf lediglich zwei Zeilen Code:

```
static void main(String args[])
{
multi test = new multi();
test.start();
}
```

Als erstes wird ein neues Objekt der eigenen Klasse erstellt und dann die eigentliche Startmethode aufgerufen. Mehr sollte in **`main()`** nicht passieren, da diese Methode ohne Objekt gestartet wird. Der wirklich aufregende Part passiert in **`start()`**:

```
public void start()
{
    Thread myThread = new
Thread(this);
        myThread.start();
        myThread.stop();
}
```

Der Name der Methode ist frei wählbar, doch **`start()`** bietet sich in diesem Fall an, weil hier der eigentliche Thread gestartet wird. Der erste Schritt ist die Erstellung eines neuen Objekts aus der Klasse **`Thread`**, deren Konstruktor das aktuelle Objekt als Parameter übergeben bekommt. Damit wird Java gesagt, wo sich die Methode **`run()`** befindet. Es ist durchaus möglich, **`run()`** in eine andere Klasse auszulagern; in diesem Fall müsste dem Konstruktor der aktuelle Objektname übergeben werden.

Nachdem wir erfolgreich ein neues Objekt erstellt haben, sind wir bereit, einen Thread zu starten. Das geschieht mit der Methode **`start()`**, die parallel zum eigentlichen Programm den Code in **`run()`** ausführt. Soll der Thread wieder gestoppt werden, benutzt du die Methode **`stop()`**, die eben dafür exisistiert.

Um das ganze etwas deutlicher zu machen, hier der Code mit einem kleinen Beispiel:

```
public class multi implements Runnable
{
static void main(String args[])
{
multi test = new multi();
test.start();
}

public void start()
{
    Thread myThread = new Thread(this);
      myThread.start();

      int j = 0;
      while (j < 500)
      {
      System.out.println("Hallo Main: " + j);
      j++;
      }
      myThread.stop();
}

public void run()
      {
           while(true)
         {
         System.out.println("Hallo Thread!");
         }
      }
}
```

Nach dem Start des Threads wird im Hauptprogramm eine Schleife gestartet, die 500 mal den Satz „Hallo Main" ausgibt und gleichzeitig zur Kontrolle eine Variable hoch zählt. An sich nichts aufregendes, doch in Verbindung mit der Methode `run()`, die dank einem Thread parallel läuft, wird schön ersichtlich, wie ein Thread funktioniert. Die Methode `run()` ist ähnlich wie das Hauptprogramm aufgebaut. Eine Endlosschleife gibt die Worte „Hallo Thread!" aus und zeigt damit an, dass sich Java gerade im Thread befindet.

Startet man das Programm, wechseln sich die Ausgaben der beiden Sätze mehr oder weniger gleichmäßig ab und zeigen jeweils den Zustand des Programms an. Wenn `j` den Wert 500 erreicht hat, wird der Thread gestoppt und das Programm beendet.

Thread Zustände beeinflussen

Neben den Methoden `start()` und `stop()` gibt es ein paar weitere Möglichkeiten, einen Thread zu steuern. Möchte man z.B. dafür sorgen, dass ein Thread eine bestimmte Zeit pausiert, benutzt man die Methode `sleep()`.

Anhand des Parameters kann man festlegen, wie lange ein Thread „schlafen“ soll. Die Angabe erfolgt immer in Millisekunden.

```
myThread.sleep(2000);
```

Der Thread wird mit dieser Anweisung zwei Sekunden ausgesetzt. Die Anweisung **sleep()** wirft die Ausnahme **InterruptedException** die in jedem Fall abgefangen werden muss.

```
try{
mThread.sleep(100);
}
catch(InterruptedException e) {
System.out.println("Fehler: " + e);
}
```

Soll die Pause von einem bestimmten Ereignis abhängig gemacht werden, ist die Methode **sleep()** ungeeignet. Als Alternative bietet Java die Methoden **suspend()** und **resume()** an, die ähnlich arbeiten wie **sleep()**. Wird **suspend()** aufgerufen, pausiert der Thread, bis über **resume()** er wieder gestartet wird.

```
myThread.suspend();
...Tue etwas wichtiges...
if (Erreignis) {myThread.resume());
```

Die Holzhammermethode der Threadsteuerung ist **destroy()**. Hiermit kann ein Thread ohne Rücksicht auf Verluste mit sofortiger Wirkung beendet werden. Das Threadobjekt wird dabei zerstört und muss neu erstellt werden.

In der aktuellen Version des JDK sind die Methode **stop()**, **suspend()** und **resume()** als **deprecated** gekennzeichnet. Das besagt, dass Sun diese Methoden als veraltet einstuft. In komplexen Programmen, die mit einer Reihe von Threads arbeiten, besteht die Gefahr, dass diese Methoden Deadlocks oder unvorhergesehene Fehler auslösn. Sun empfiehlt, Threads über Steuervariablen zu manipulieren.

Für kleine Programme oder Applets, die nur mit einem Thread arbeiten ist, diese Technik allerdings unerheblich, da hier nichts schief gehen kann. Beim Kompilieren solcher Programme spuckt der Compiler eine Warnmeldung aus, die darauf hinweist, dass du mit einer „veralteten“ Methode arbeitest. Das Programm wird dennoch compilert und ist voll nutzbar.

Ich empfehle deshalb trotz allem die Nutzung dieser Methoden. Für weitere Informationen lies bitte das Kapitel *„Was zum Teufel heißt deprecated?!“* auf Seite 80 und die Java-API.

Applets und das Internet

Willkommen im interessantesten und gleichzeitig letzten Kapitel dieses Heftes. Hier werden wir uns mit der Programmierung von Java-Applets beschäftigen – ein Thema, das zu den spannendsten Feldern rund um Java gehört. Die meisten werden Applets bereits aus dem Internet kennen, wo sie für die unterschiedlichsten Aufgaben eingesetzt werden. Ob es einfache Newsticker sind, die eine Webpage verschönern, oder komplette Webbanking-Projekte, die der Onlinebroker anbietet, alle haben eines gemeinsam: Sie sind immer in Java programmiert und können weit mehr als man ihnen zutraut.

Was ist ein Applet?

Im Gegensatz zu den bisherigen Programmen, die wir geschrieben haben, handelt es sich bei Applets nicht um eigenständige Programme oder Applikationen, sondern um eine browserabhängige Programmierung, die nur in dieser bestimmten Umgebung funktioniert. Es lassen sich in Java zwar eigenständige grafische Applikationen schreiben, doch ist das nicht ein Thema dieses Heftes. Applets sind kleine Programme, die speziell für das Internet geschrieben werden und vom User beim Aufruf einer Webseite automatisch auf seinen Rechner geladen werden. Damit sind wir auch bei einem wichtigen Punkt: Applets werden grundsätzlich lokal auf dem Rechner des jeweiligen Besuchers gestartet. Der Server hat mit Applets eigentlich nichts zu tun – er speichert sie nur, um sie bei Abruf bereit zu stellen. Dieser Unterscheidung ist sehr wichtig für die spätere Arbeit.

Ein Applet funktioniert in der Regel so, dass es unabhängig von der Webseite in einem ihm zugewiesenen Punkt im Browser läuft. Das Applet wird zwar über HTML ausgegeben, läuft im übrigen aber unabhängig ab. Dennoch wird für seine Betrachtung ein Browser benötogt. Die einzige Alternative ist der Appletviewer von Java, der beim JDK mitgeliefert wird. Dieser ist allerdings nur fürs Testen und Ausprobieren während des Programmierens sinnvoll.

Applets in eine Webseite einbinden

Bevor wir uns ans Programmieren von Applets machen, müssen wir wissen, wie diese in eine Webseite eingebunden werden. JOE hat zwar die Möglichkeit, Applets während des Schreibens automatisch im Appletviewer anzuschauen, aber spätestens beim Release des fertigen Programms wird ein bißchen HTML nötig sein – ohne das der Appletviewer auch nicht auskommt.

Der Appletviewer wird von JOE aus mit einem Druck auf die Taste F8 gestartet. Du wirst gefragt, ob eine HTML-Datei für den Appletviewer erstellt werden soll. Diese Frage kannst du bejahen – genau das wollen wir nämlich jetzt tun. Die fertige Datei wird dann im Appletviewer gestartet, lässt sich aber auch in jedem Browser darstellen.

Willst du ein fertiges Applet später in eine Webseite einbinden, benutzt du den HTML-Tag **<applet>**, der das Applet mit einer Reihe von Parametern für den Browser beschreibt. Eine fertige Seite könnte so aussehen:

```
<html>
<head>
<title>Mein erstes Applet</title>
</head>
<body>
<b>Mein erstes Applet:<br></b>
<applet code="myApplet.class" width=150 height=150>
<param name="var1" value="Hallo">
<param name="var2" value="Java!">
</applet>
</body></html>
```

Wichtig ist der fettgedruckte Teil. Innerhalb des Applet-Tags wird dem Browser der Name der Programmdatei übergeben. Diese Datei wird vom Server geladen und auf dem Rechner des Besuchers deiner Seite ausgeführt. Liegt die Datei schon auf dem Rechner, was bei unseren Testprogrammen der Fall ist, sucht der Browser in dem Verzeichnis, wo auch die HTML-Datei liegt. Die Parameter **width** und **height** beschreiben ähnlich wie bei einem Bild Höhe und Breite der Appletausgabe. Stimmen diese Werte mit der tatsächlichen Größe des Programms nicht überein, wird die Ausgabe abgebrochen.

Innerhalb des Applet-Tags ist es möglich, optionale Parameter an das Programm zu übergeben. Java vermag diese Werte auszulesen und mit ihnen zu arbeiten – mehr dazu später.

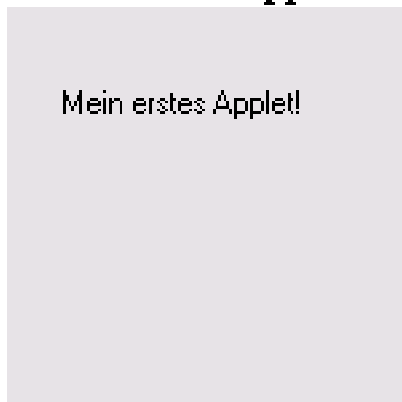
Abgeschlossen wird das Ganze mit einem abschließenden Tag, der dem Browser signalisiert, dass der Code für das Applet hier endet.

Das erste Java Applet

Um das gelernte gleich umzusetzen, folgt hier der Code für unser erstes Java-Applet. Wir werden das Programm in den folgenden Kapiteln nach und nach besprechen – also keine Panik wenn nicht alles sofort klar ist.

```
import java.applet.*;
import java.awt.*;
public class myApplet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("Mein erstes Applet!", 20, 40);
    }
}
```

Wird das kompilierte Programm über einen Browser dargestellt, sieht das Ganze etwa so aus:



Ich habe für diesen Screenshot den oben angegebenen HTML-Code genutzt. Das eigentliche Applet ist der graue Bereich, der Rest wird über HTML ausgegeben. Die übergebenen Parameter spielen in diesem Programm noch keine Rolle.

Je nach Browser kann der Standardhintergrund von Applets berschieden aussehen. Im Internet Explorer ist er grau. Im Appletviewer von Sun ist er fast immer weiß. Im Netscape Navigator ist er hellgrau. Leicht abweichende Ergebnisse sollten dich also nicht verunsichern.

Das Packet java.applet

Der erste Schritt in deinem ersten Applet ist der import vom Paket **java.applet**. Dieses Paket enthält nur eine einzige Klasse namens **Applet**, die allerdings für die Appletprogrammierung zwingend notwendig ist. Zusätzlich enthält das Programm einige Schnittstellen, die zur Komunikation mit der Außenwelt genutzt werden, d.h. dem Browser, Betriebssystem usw., sowie zur Ausgabe von Soundeffekten.

Willst du ein Applet schreiben, solltest du die Klasse **Applet** ableiten, damit die notwendigen Methoden zur Verfügung stehen. Dies geschieht am einfachsten über den bekannten Befehl **extends**, der unsere Klasse **myApplet** aus der Oberklasse **Applet** ableitet. Im Prinzip haben wir jetzt schon ein lauffähiges Applet vor uns, denn alle Methoden, die für die Kontrolle des Programms nötig sind, wurden bereits in der Klasse **Applet** geschrieben. Auch wenn sich das Programm ohne Probleme kompilieren läßt, ist das Ergebnis nicht sehr aufregend: Ein trauriger grauer Kasten.

Möchtest du etwas mehr Leben in dein Applet bringen, musst du die jeweiligen Methoden der Klasse **Applet** überschreiben.

Die Methoden eines Applets

Die Oberklasse **Applet** vererbt der jeweils abgeleiteten Klasse eine Reihe von Methoden zur Steuerung des Applets. Willst du auf die Funktionalität dieser Methode zugreifen, müssen diese überschrieben werden. Das Besondere an diesem Methoden ist, dass sie automatisch vom Browser oder Appletviewer aufgerufen werden. Es ist fast nie nötig, diese Methode per Hand zu starten, da der Browser diese Arbeit übernimmt.

Die Methode init()

Die Methode **init()** wird automatisch bei der Initialisierung des Applets aufgerufen. Hier werden Variablen und Parameter für das Programm gesetzt, die für den Ablauf nötig sind. Die Methode **init()** wird während der Laufzeit des Applets nur ein einziges Mal aufgerufen, und zwar gleich zu Beginn. Wird die Methode **init()** nicht überschrieben, lädt der Browser eine Standardversion aus der Mutterklasse **Applet**. Der Rückgabewert von **init()** ist void.

Die Methode start()

Die Methode **start()** dient zur Implementierung der Funktion des Applets. Sie wird automatisch aufgerufen, wenn das Applet im sichtbaren Browser-Bereich erscheint. Das ist zum Beispiel beim Start des Applets der Fall, aber auch, wenn das Programm im Browser in den sichtbaren Bereich gescrollt wird. Diese Methode kann also mehr als einmal aufgerufen werden. Häufig wird die Methode **start()** dazu benutzt. einen Thread zu starten und so z.B. eine Animation zu realisieren. Der Rückgabewert ist ebenfalls **void**.

Die Methode stop()

Die Methode **stop()**, das Gegenstück zu **start()**, übernimmt die Aufgabe, das Applet zu beenden. Sie wird immer dann aufgerufen, wenn das Applet den sichtbaren Bereich verläßt, also der Browser geschlossen oder sein Inhalt weiter gescrollt wird. Wie **start()** kann auch **stop()** im Programmablauf mehrmals ausgeführt werden. Diese Methode wird gerne benutzt, um gestartete Threads zu stoppen oder zu beenden. Auch hier ist der Rückgabewert **void**.

Die Methode destroy()

Mit **destroy()** haben wir die letzte wesentliche Steuermethode eines Applets erreicht. Sie ist das Gegenstück zu **init()**, denn **destroy()** tut das, was der Name vermuten läßt: Es zerstört das Applet. Der Unterschied zu **stop()** besteht darin, dass diese Methode das Programm entgültig beendet, also alle Resourcen freigibt und via Garbage Collector den Speicher wieder räumt . Wie **init()** wird **destroy()** auch nur einmal aufgerufen – und zwar wenn der User die Seite verläßt bzw. den Browser schließt. Auch diese Methode gibt **void** zurück.

Diese vier Methoden bestimmen das Leben eines einfachen Applets. Sie treten in der hier beschriebenen Reihenfolge auf. Wunderst du dich jetzt, warum keine der hier genannten Methoden in unserem Beispielprogramm auftaucht, hat das einen recht einfachen Grund: Das Beispielapplet übernahm alle Methoden aus der Mutterklasse, ohne sie zu verändern. Da wir nur einen simplen Schriftzug ausgegeben haben, war es nicht nötig, die Funktionalität zu erweitern. Die einzige notwendige Methode ist **paint()**, die nicht direkt aus der Klasse **Applet** stammt.

Die Methode paint()

Die Methode **paint()** nimmt im Applet eine besondere Stellung ein, da sie nicht direkt aus der Klasse **Applet** stammt. Der Ursprung von **paint()** liegt in der Oberklasse **Component**, die auch für die grafische Ausgabe von Applikationen verantwortlich ist. Das spielt für die Programmierung allerdings kaum eine Rolle, da **paint()** genau wie die die Steuerfunktionen aus der Klasse **Applet** geerbt wird.

Die Methode **paint()** wird aufgerufen, wenn das Applet neu gezeichnet werden muss. Das ist zum Beispiel dann der Fall, wenn das Programm gestartet wird oder das Browserfenster sich irgendwie verändert.

Paint dient der reinen Grafikausgabe und wird deshalb mit einem Objekt der Klasse **Graphics** gestartet. Über dieses Objekt werden dann alle Ausgaben auf dem Bildschirm realisiert.

```
public void paint(Graphics g)
{
g.drawString("Mein erstes Applet!",
20, 40);
}
```

Das Objekt kann einen beliebigen Namen haben. **g** bietet sich aufgrund der kürze einfach an.

Immer wenn **paint()** aufgerufen wird, wird zuvor die Methode **update()** gestartet, die ebenfalls aus der Klasse Component geerbt wurde. **update()** sorgt dafür, dass das Applet vor dem Neuzeichnen durch **paint()** komplett gelöscht wird. Normalerweise muss man sich nicht um diesen Aufruf kümmern, da alles von allein geschieht – in manchem Fällen ist es aber nötig, **update()** zu überschreiben.

Ein Applet im Überblick

Sammelt man alle Methoden eines Applets, kann man eine Art Gerüst konstruieren, das seine Struktur zeigt. Ein solcher Bauplan gestaltet die Programmierung eines Applets sehr bequem:

```
import java.applet.*;
import java.awt.*;

public class newApplet extends Applet
{
    public void init()
    {
        //initialisierung aller nötigen Werte für das Programm
    }

    public void start()
    {
        //Start des Applets
    }

    public void paint(Graphics g)
    {
        //Grafikausgabe über das Objekt g !
    }

    public void stop()
    {
        //Stop des Applets
    }

    public void destroy()
    {
        //Ende des Applets
        //Speicher wird freigegeben
    }
}
```

Das Paket java.awt

Das Paket **`java.awt`** dient zur Bearbeitung und Ausgabe von Grafiken und Fensterelementen in Applikationen und Applets. Neben dem Paket **`java.lang`** ist **`java.awt`** die umfangreichste Bibliothek in der Java-API. Die Abkürzung AWT steht für *advanced window toolkit*.

Da wir uns in diesem Heft nur mit der grafischen Appletprogrammierung befassen, betrachten wir nur einen kleinen Teil dieses Paketes – der nun allerdings elementar für uns ist, bewegen wir uns doch jetzt in einem komplett grafischen System abseits der Befehlseingabe der Eingabeaufforderung. Dies musst du dir verdeutlichen, wenn du dich mit der Appletprogrammierung beschäftigst.

Deine Programme werden jetzt nicht mehr linear von oben nach unten abgearbeitet, sondern sind vielen „Gefahren“ ausgesetzt, z.B. überlappende Fenster, Scrollmechanismen und Mauszeiger. Java hat für alle diese Probleme eine Lösung, nur benötigen wir eine saubere grafische Oberfläche. Dafür sorgt das Paket **java.awt**. Neben dem Paket **java.applet** ist **java.awt** eine Grundvoraussetzung für ein Applet.

Die Klasse Graphics

Die wohl wichtigste Klasse im **awt**-Paket ist die abstrakte Klasse **Graphics**. Hier werden alle Methoden und Ausgabemechanismen für Grafiken, Schriften und Bilder systemunabhängig deklariert. Wann immer du etwas in dein Applet malen, schreiben oder zeichnen willst, benötigst du ein Objekt dieser Klasse.

Da wir in einem Applet die Grafikausgabe über die Methode **paint()** realisieren, wird dieser als Parameter ein Objekt der Klasse **Graphics** übergeben:

```
public void paint(Graphics g)
```

Das Objekt kann dann in der Methode genutzt werden, um in das Applet zu zeichnen. Möchtest du unabhängig von der **paint()**-Methode ein **Graphics**-Objekt erzeugen, benutzt du dazu die Methode *getGraphics()*, die ein Ausgabeobjekt des aktuelles Programms erstellt und zurück gibt.

```
Graphics g = getGraphics();
```

Je nach Betriebssystem und den Anforderungen werden über eine Schnittstelle von Graphics alle nötigen Methoden implementiert; so bleibt die Plattformunabhängigkeit von Java gewahrt. Mit dem fertigen Objekt der Klasse können dann alle Grafikmethoden aus Graphics genutzt werden.

Wunderst du dich jetzt, wozu wir bei einer Textausgabe eine Grafikroutine benötigen, halte dir bitte vor Augen, dass in einem grafischen Benutzer-Interface wie Windows, auch GUI genannt, alles als Grafik betrachtet wird. Die Methode malt den String sozusagen in das Applet und trägt deshalb auch den treffenden Namen **drawString()**. Als Parameter übergeben wir eine x- und eine y-Koordinate, deren Startpunkte die untere linke Ecke des von der Methode auszugebenden Textes definieren – die Werte definieren also die *untere linke* Ecke des ersten Wortes.

Im Gegensatz zum aus der Schule bekannten Koordinatensystem beginnt Java allerdings in der oberen linken Ecke bei 0,0 zu zählen. Der Ursprung wäre also nicht *unten* links, sondern *oben* links zu finden.

Die Klasse Color

Über die Klasse **Color** werden Farbeigenschaften grafischer Objekten definiert. Über den Konstruktor dieser Klasse wird einem neuen Farbobjekt die gewünschte Farbe im RGB-Modus zugewiesen. Sie setzt sich also, wie aus Grafikprogrammen wie Photoshop bekannt, aus den Grundfarben Rot, Grün und Blau zusammen. Pro Farbwert stehen 255 verschiedene Möglichkeiten der Farbdefinition zur Verfügung, die über einen dezimalen Zahlenwert zwischen 0 und 255 beschrieben werden. Eine neues Farbobjekt wird also mit drei Zahlen definiert, die dem Konstruktor übergeben werden:

```
Color farbe = new Color(255,0,0);
```

Das neue Objekt **farbe** wird über die Parameter 255, 0 und 0 erschaffen. Für den Konstruktor klingt das übersetzt etwa so:

Nimm 100% der Farbe Rot und mische diese mit 0% der Farbe Grün und füge dann noch 0% von Blau hinzu.

Heraus kommt, ganz klar, die Farbe rot. Die Prozentangaben beziehen sich auf die Anteile der Farben zwischen 0 und 255, die zur Verfügung stehen. Auf diesem Wege sind auch verschiedene Mischfarben möglich:

```
Color farbe = new Color(127,63,127);
```

Diese Anweisung setzt das Objekt **farbe** auf einen interessanten Lilawert mit Braunanteil, über den jeder sein eigenes Urteil bilden möge. Die Farbmischung ist 50% Rot, 25% Grün und wieder 50% Blau.

Ein kühler Rechner wird schnell bemerken, dass auf diesem Wege genau (256 * 256 + 256 =) 16.777.216 verschiedene Farben zu Verfügung stehen, womit wir uns auf der TrueColor-Ebene von Windows bewegen. Allerdings muss das jeweilige System diese Farbanzahl auch unterstützen, was nicht sicher ist. Wird dein Applet auf einen Rechner mit nur 256 Farben gestartet, kann das manchmal wirklich hässlich sein.

Wer auf Nummer sicher gehen will, der sollte die vordefinierten Farben von Java nutzen. Die Klasse **Color** stellt eine Reihe von Klassenvariablen zu Verfügung, die klassische Farbtöne definieren. Aufgerufen werden diese über die bekannte Punktnotation:

```
Color.green //grün
Color.blue //blau
Color.white //weiß
```

Hier eine Liste der oft verwendeten Farben.

Konstante	Farbe
Color.white	weiss
Color.yellow	gelb
Color.orange	orange
Color.grey	grau
Color.pink	pink
Color.red	rot
Color.magenta	magenta
Color.green	grün
Color.blue	blau
Color.darkGray	dunkelgrau
Color.lightGray	hellgrau
Color.black	schwarz

Diese Farben sollten auf jedem System zu finden sein.

Wie aber kommt die Farbe nun ins Applet? Recht einfach – über die Methode **setColor()** aus der Klasse **Graphics**. Die Farbe wird für jedes Graphicsobjekt einzeln gesetzt. Für uns heißt das, dass wir die Methode über das Objekt **g** der Klasse **Graphics** starten und als Parameter die jeweilige Farbe übergeben.

```
g.setColor(farbe);
g.setColor(Color.blue);
```

Für alle folgenden Zeichenfunktionen wird jetzt die hier angegebene Farbe verwendet.

```
Color farbe = new Color(0,0,255);
g.setColor(farbe);
g.drawString(„Malt auch in bunt!",
20, 40);
```

Fügst du diese Code-Zeilen in die **paint()**-Methode ein, wird der Satz in blau ausgegeben.

Die Hintergrundfarbe eines Applets kann über die Methode **setBackground()** verändert werden, die aus der Klasse **Applet** übernommen wurde. Sie ist unabhängig von einem Objekt der Klasse Graphics, wird aber wie **setColor()** mit einem Colorobjekt als Parameter gestartet.

```
setBackground(Color.green);
```

So wird der Hintergrund grün. Wer es wirklich geschmacklos mag, der sollte dazu eine gelbe oder violette Schrift versuchen ☺

Zwei weitere nützliche Methoden seien hier erwähnt. Mit **brighter()** und **darker()** kann über ein Farbobjekt eine neue Farbe definiert werden. Java wählt dabei die nächst-hellere oder dunklere Farbe in der Farbpalette und erlaubt so Farbübergänge:

```
Color farbe2 = farbe.darker();
```

Diese Zeile erschafft eine neue Farbe, mit dem Namen **farbe2**, die einen Tick dunkler ist als die Ausgangsfarbe. Umgekehrt funktionert es entsprechend mit **brighter**, was die Farbe einen Tick heller macht.

Die Klasse Font

Die Klasse **`Font`** stellt das wichtigste Werkzeug im plattformunabhängigen Umgang mit Schriften dar. Java definiert zu diesem Zweck fünf Schriftarten,die jedes System unterstützen sollte. Des weiteren verfügt Java über eine Standardschrift (Default), die immer dann verwendet wird, wenn die Schriftart nicht definiert ist oder das System die Schrift nicht unterstützt. In der folgenden Tabelle findest du alle Schriftarten, wie sie unter Windows heißen, und wie Java sie definiert.

Windows	**Java**
Arial	Helvetica
Courier New	Corier
MS Sans Serif	Dialog
Times New Roman	TimesRoman
WingDings	ZapfDingbats
Arial	default

Die Default-Schrift in Java ist Arial, die wir bisher automatisch verwendet haben.

Willst du in Java ein neues Fontobjekt, also eine neue Schrift erstellen, musst du dem Konstruktor der Klasse einige Informationen übergeben. Eine Schrift ist in diesem Fall aus drei Parametern zusammengesetzt:

1. Schriftart
2. Schriftstil (fett, kursiv, etc.)
3. Schriftgröße

Die Schriftart wird über den Namen der Schrift definiert und als String übergeben. Der Schriftstil in Java wird über drei Klassenkonstanten definiert, als da wären **`plain`** (normal), **`italic`** (kursiv) und **`bold`** (fett). Der letzte Parameter ist die Schriftgröße, die in Pixeln angegeben wird. Alles zusammen gibt folgendes Bild:

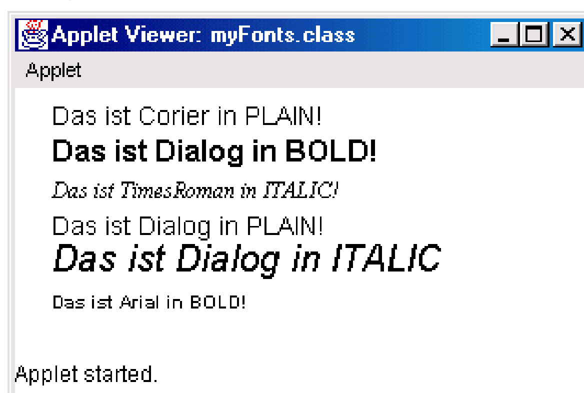
```
Font f = new Font("Corier",
Font.ITALIC, 14);
```

Die Schrift **`f`** ist eine 14 Pixel große Schrift des Typs Courier in kursivierter Form.

Willst du die Schrift im Applet anwenden, benutzt du die Methode **`setFont()`** aus der Klasse **`Graphics`**. Ähnlich wie eine Farbe wird die Schriftart für ein Applet gesetzt. Es folgt ein Beispiel, dass einen Überblick über die Schriftarten in Java gibt:

```
import java.applet.*;
import java.awt.*;

public class myFonts extends Applet
{
    Font f1 = new Font("Corier", Font.PLAIN, 14);
    Font f2 = new Font("Dialog", Font.BOLD, 16);
    Font f3 = new Font("TimesRoman", Font.ITALIC, 12);
    Font f4 = new Font("Dialog", Font.PLAIN, 14);
    Font f5 = new Font("Dialog", Font.ITALIC, 20);
    Font f6 = new Font("Arial", Font.BOLD, 10);

    public void paint(Graphics g)
    {
    g.setFont(f1);
    g.drawString("Das ist Corier in PLAIN!", 20, 20);
    g.setFont(f2);
    g.drawString("Das ist Dialog in BOLD!", 20 , 40);
    g.setFont(f3);
    g.drawString("Das ist TimesRoman in ITALIC!", 20, 60);
    g.setFont(f4);
    g.drawString("Das ist Dialog in PLAIN!", 20, 80);
    g.setFont(f5);
    g.drawString("Das ist Dialog in ITALIC", 20, 100);
    g.setFont(f6);
    g.drawString("Das ist Arial in BOLD!", 20, 120);
    }
}
```

Fertig compilert sieht es im Browser dann so aus:



Die Klasse Image

Die abstrakte Klasse **`Image`** ist für die plattformunabhängige Darstellung von Bildern verantwortlich. Ähnlich wie **`Graphics`** darf **`Image`** nicht über einen Konstruktor instanziert werden, sondern muss vielmehr über die Klassenmethode **`getImage()`** ein Objekt erschaffen. Diese Methode hat die Aufgabe, ein Bild zu laden und es Java zur Verfügung zu stellen. Da Applets in der Regel im Internet laufen, benötigt diese Methode die Angabe eines URL gefolgt vom Namen des Bildes. Dies lässt sich allerdings recht einfach über die Methode getCodeBase() erledigen, die ein entsprechendes URL Objekt zurückgibt.

Dabei handelt es sich um das Verzeichnis, in dem das Programm liegt. Im Programm sieht der Vorgang dann so aus:

```
Image myImage =
getImage(getCodeBase(),
"Earth.jpg");
```

Das Bild EARTH.JPG wird aus demselben Verzeichnis wie das Programm geladen und im Imageobjekt **myImage** gespeichert.

Willsz du im Applet ein Bild anzeigen, benutzt du die Methode **drawImage()** der Klasse **Graphics**. Die Methode übernimmt 4 bzw. 6 Parameter, mit denen die Ausgabe des Bildes gesteuert werden kann. Die einfache Variante sieht so aus:

```
g.drawImage(Image img, int x, int y,
Observer Objekt);
```

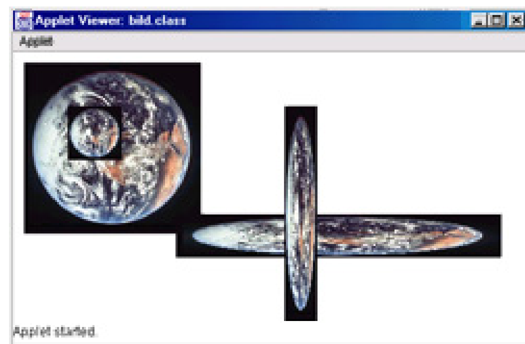
Der erste Parameter ist das Imageobjekt selber, wie es gezeichnet werden soll. Über die Koordinaten x und y wird die Position im Applet bestimmt. Der letzte Punkt der Methode verlangt nach einem Steuerobjekt für die Zeichnung des Bildes. In unserem Fall ist es das Applet selber, also **this**.

Die zweite Variante erlaubt über zwei weitere Parameter die Manipulation der Höhe und Breite des Bildes.

```
g.drawImage(Image img, int x, int
width, int height, int y, Observer
Objekt);
```

width und **height** legen die Ausmaße des Bildes fest und erlauben so, sie zu ändern oder das Bild zu verzerren.

```
import java.applet.*;
import java.awt.*;

public class bild extends Applet
{
Image myImage;

public void init()
{
      myImage = getImage(getCodeBase(), "Earth.jpg");
}
public void paint(Graphics g)
{
    g.drawImage(myImage, 10, 10, this);
    g.drawImage(myImage, 50, 50, 50, 50, this);
    g.drawImage(myImage, 150, 150, 300, 40, this);
    g.drawImage(myImage, 250, 50, 30, 200, this);
}
}
```



Die Klasse Button

Die Klasse **Button** stellt eines der meist genutzten Steuerlemente des grafischen Systems bereit: die *Buttons*, die wir überall in Windows ganz selbstverständlich nutzen. Die Verwendung dieser Klasse ist sehr einfach gestrickt, da der Konstruktor lediglich einen Parameter braucht, und zwar die Aufschrift des Button:

```
Button myB = new Button("Klick");
```

Diese Code-Zeile schafft ein neues **Button**-Objekt namens **myB** mit der Aufschrift „Klick“.

Damit auch etwas zu sehen ist, muss der Button ausgegeben werden. Das geschieht über die Methode **add()** aus der Klasse **Applet**.

```
myApplet.add(myB);
```

ebenso möglich:

```
this.add(myB);
```

Ein Button wird,wie jedes andere Steuerelement, andersn als Grafiken nicht gezeichnet, sondern gesetzt. Dies geschieht im Leben eines Applet nur einmal – und muss darum in der Appletmethode **init()** untergebracht werden. Ein weiterer Unterschied zu Grafiken ist die Platzierung eines Steuerelements: Es wird nicht nach Koordinaten gesetzt, sondern anhand eines *Layout Managers* im Applet platziert. Es gibt mehrere solcher Layouthilfen, die wir aus Platzgründen leider nicht besprechen können. Die einfachste, automatisch benutzte Form beschränkt sich darauf, die neuen Elemente in der Reihenfolge ihrer Erschaffung auf dem Applet unterzubringen. Darum erscheint unser Button auch oben im Programm.

Jeder weitere Button würde daneben oder darunter platziert werden, je nach Platzangebot.

Möchte man die Beschriftung des Buttons nachträglich ändern, ist das problemlos über die Methode **setLabel()** möglich.

```
myB.setLabel("Klick mich!");
```

Die Größe des Buttons wird dann automatisch angepasst. Die Methode **setLabel()** liest die Beschriftung des Buttons aus und gibt sie in einem String zurück.

Ein Klick auf dem Button löst ein Event aus, auf das man im Programm reagieren kann. Das Eventhandling betrachten wir auf Seite 73.

Mehr Multimedia

Die Klasse **Graphics** bietet natürlich eine ganze Menge mehr als die simple Ausgabe von Text in verschiedenen Farben. Es stehen dir hier eine ganze Reihe von verschiedenen Methoden zur Verfügung, mit denen du nach Herzenslust im Applet zeichnen und malen kannst.

Die drawLine() Methode

Die einfachste Technik in einem Applet ist das zeichnen von Linien. Die Methode **drawLine()** ermöglicht das Zeichnen von Linien zwischen zwei definierten Punkten in der aktuellen Farbe.

```
g.drawLine(x1, y1, x2, y2);
```

Dieser Aufruf in der Paint-Methode zieht eine Linie zwischen den Punkten x1, y1 und x2, y2.

Die Methoden drawRect() und fillRect()

Die Methode **drawRect()** wird genau wie die Methode **drawLine()** verwendet. Es werden vier Parameter übergeben, zwischen denen ein Recheck gezeichnet wird.

```
g.drawRect(100, 100, 200, 300);
```

Dieser Aufruf würde beispielsweise ein Rechteck mit den Eckkoordinaten 100, 100 und den Längen 200 und 300 zeichnen.

Mit dieser Methode zeichnest du allerdings nur die Umrandung eines Rechtecks. Soll es auch gefüllt werden, benutzt du dafür die Methode **fillRect()**, die ein gefülltes Rechteck zeichnet, ansonsten aber so funktioniert wie die Methode **drawRect()**.

Abgerundete Rechtecke mit drawRoundRect()

Neben normalen Rechtecken bietet Java auch die Möglichkeit, Rechtecke mit abgerundeten Ecken zu zeichnen. Die Methode **drawRoundRect()** wird mit sechs Parametern aufgerufen, von denen allerdings nur zwei neu sind. Nach den Koordinaten und den jeweiligen Längenangaben verlangt Java die Angabe von Winkeldaten für die horizontale und vertikale Ebene. Je größer die Zahl ist, desto stärker ist die Rundung.

```
g.drawRoundRect(100, 100, 200, 300,
45, 55);
```

Dieser Aufruf zeichnet sehr stark gerundete Ecken. Um ein gefülltes „Rundeck“ zu schaffen benutzt du die Methode **fillRoundRect()**.

Kreise und Ellipsen

Kreise und Ellipsen werden in Java mit ein und derselben Methode geschaffen: **drawOval()**. Dabei geht Java so vor wie beim Zeichnen von Rechtecken, d.h. es werden beim Methodenaufruf die selben Parameter verlangt. Du gibst die obere linke Ecke des Rechtecks an, in den der Kreis gezeichnet werden soll. Die letzten beiden Para-

meter sind Breite und Höhe des Kastens. Sind sie gleich, ergibt das einen Kreis, andernfalls eine Ellipse.

```
g.drawOval(100, 200, 400, 100);
```

Die Methode **fillOval()** erlaubt das Zeichnen von gefüllten Kreisen und Ellipsen.

Kopieren eines Bereichs

Die Methode **copyArea()** erlaubt das Kopieren einen bestimmten (sichtbaren) Bereich eines Applets. Über die Parameter werden Quell- und Zielkoordinaten angegeben. Die Methode kopiert alle Grafiken des definierten Bereichs in den Zielbereich.

```
g.copyArea(x1, y1, x2, y2, zielx,
ziely);
```

Der Zielbereich wird nur über die obere Linke Ecke definiert, da Höhe und Breite bereits über die Quellkoordinaten festgelegt sind.

Eventhandling

Das Schlagwort Eventhandling (zu deutsch: Ereignisbehandlung) taucht zum ersten mal im Bereich der grafischen Programmierung auf. In einfachen Kommandozeilenprogrammen, die linear von oben nach unten abgearbeitet werden, ist eine besondere Eventpolitik nicht notwendig, da das Programm immer die Kontrolle über die Handlungen des Users hat. Es legt genau fest, wann der User etwas eingeben darf und wann er ENTER drückt, um ein „Ereignis“ auszulösen. Das Programm führt den User sozusagen an der Hand.

Ein grafisches Interface funktioniert anders. Hier wurden die Rollen zwischen dem aktiven und passiven Anteil der Kommunikation getauscht, und das Programm muss auf den User reagieren. Es hat keine Ahnung, ob als nächstes der Button „weiter“ geklickt wird oder ob der User die Maus einfach nur über das Applet bewegt. Wenn es ganz dumm kommt, begeht er Fehler, die das Programm dann abfangen muss.

Das neue Programmprinzip funktioniert also völlig anderns als bisher gekannt. Das Programm *wartet* auf die Handlung des Users und reagiert erst dann.

Dazu ist es nötig, eine Reihe von Ereignisse zu definieren, die in der Laufzeit eines Programm eintreten können. Das ist bei jedem Programm unterschiedlich, das Prinzip ist aber immer das selbe: Jedes Ereignis löst eine Methode aus, die auf das Ereignis reagieren kann. So gibt es zum Beispiel die Methode **mousePressed()**, die automatisch gestartet wird, wenn irgendwo im aktiven Applet die Maustaste gedrückt wird.

Dieses Vorgehen lässt sich mit den Steuermethoden eines Applets vergleichen, die im Prinzip auch nur auf Ereignisse reagieren. Die Methode **init()** reagiert auf das Ereignis „Applet gestartet“, wohingegen die Methode **destroy()** auf das Ende des Applets reagiert. Die Methode **paint()** reagiert immer dann, wenn die Notwendigkeit „Neuzeichnen“ eintritt, die ebenfalls ein Event ist.

Hast du dich schon mit der Windowsprogrammierung mit C/C++ befasst (KnowWare-Heft EXTRA Nr. 6), ist dir diese Technik sicher geläufig. Java greift im Prinzip auf die Ereignispolitik des jeweiligen Betriebsystems zurück.

Eventhandling in Java

Java definiert eine Reihe von Schnittstellen, die für die verschiedenen Bereiche des Eventhandling zuständig sind. Sollen diese Schnittstellen nutzt werden, muss das Paket **java.awt.events** eingebunden werden:

```
import java.awt.event.*;
```

Mit dieser Anweisung stehen alle notwendigen Schnittstellen zur Ereignisbehandlung zur Verfügung. Ab der Version 1.3 des JDK wird die Ereignisbehandlung in Java über ein neues System abgewickelt. Es kommen sogenannte „Listener“ zum Einsatz, die auf bestimmte Bereiche eines Programms angesetzt werden, um im Falle eines Ereignisses die richtige Methode auszulösen. Für jede Art von Listener gibt es eine eigene Schnittstelle, von denen wir hier die wichtigsten besprechen werden.

Mouse Events

Die häufigsten Events in einem grafischen System sind natürlich Mausereignisse, die mit einer ganzen Reihe eigener Eventfunktionen ausgestattet wurden. Soll auf Mausereignisse reagiert werden, muss die Schnittstelle **MouseListener** in deiner Klasse implementiert werden.

```
public class myApplet extends Applet
implements MouseListener
{...}
```

Dieser Schritt fügt unserem Applet eine Reihe von Methode hinzu, die wir überschreiben müssen. Jede dieser Methoden behandelt ein bestimmtes Mouse Event, und alle sind sie gleich gestrickt:

```
public void mousePressed(mouseEvent
e)
{...}
```

Der Methodename gibt Aufschluss über die eigentliche Aufgabe bzw. das Ereignis der Methode, auf das sie reagiert. Insgesamt müssen fünf Methode überschrieben werden, damit sich der Compiler nicht beschwert.

Methode	Aufgabe
`mousePressed()`	Maustaste wird gedrückt
`mouseReleased()`	Maustaste wird losgelassen
`mouseClicked()`	Mausklick
`mouseEntered()`	Maus bewegt sich in das Applet
`mouseExited()`	Maus verlässt das Applet

Jeder Methode wird dabei ein Objekt der Klasse **mouseEvent** übergeben, das alle signifikanten Werte dieses Ereignisses speichert.

> Mit welcher Maustaste wurde geklickt? Da Java eine plattformunabhängige Sprache ist, wird nur eine Maustaste unterstützt, d.h. Java kann nicht zwischen der linken und rechten Taste unterscheiden. Wäre es nicht so, hättest du spätestens auf einem Mac Probleme mit deinem Applet.

Bevor unser *MouseListener* loslegen kann, muss er noch aktiviert werden. Das geschieht über die Methode **addMouseListener()** aus der Klasse **Applet**. Als Argument wird das Objekt übergeben, auf das der Listener achten soll, in unserem Fall also das gesamte Applet.

```
addMouseListener(this);
```

Diese Methode solltest du in der **init()**-Methode unterbringen, da sie nur einmal aufgerufen werden muss.

Jetzt steht einem erfolgreichem Eventhandling nichts mehr im Wege, und das Programm muss nur noch mit den entsprechenden Funktionen versehen werden. Um das ganze anschaulicher zu machen, wollen wir am Ende des Kapitels die ganze Materie in einem Programm anwenden.

Die wichtigste Punkt bei einem Mausereignis ist die einfache Frage: „Wo wurde geklickt?“. Das lääst sich mit den Methoden **getX()** und **getY()** feststellen, die die jeweiligen Koordinaten vom Event-Objekt zurückliefern.

```
public void mousePressed(mouseEvent
e)
{
int x = e.getX();
int y = e.getY();
}
```

Die Variablen x und y speichern die Position, an der die Maustaste gedrückt wurde. Mit diesen Werten kann dann im Programm entsprechend weitergearbeitet werden.

Mouse Motion Events

Die Schnittstelle **MouseMotionListener** enthält eine Reihe zusätzlicher Methoden, die weitere Mouseevents behandeln können. Genauer gesagt sind es zwei neue Methoden, die bei näherem Hinsehen allerdings eminent wichtig sind und zwar die Methoden **mouseMoved()** und **mouseDragged()**, die für bewegte Mausereignisse zuständig sind. Der neue Listener muss ebenfalls über eine eigene Methode aktiviert werden, die wie die letzte funktioniert.

```
addMouseMotionListener(this);
```

Auch hier müssen die Methoden überschrieben werden, auch wenn man sie nicht nutzt, weil das andernfalls zu einem Compilererror führt.

Als Argumente werden hier ebenfalls Objekte der Klasse **MouseEvent** übergeben.

```
public void mouseMoved(mouseEvent e)
{...}
```

Das Objekt **e** speichert alle für die Bearbeitung des Ereignissen notwendigen Daten.

Action Events

Eine weitere große Gruppe von Ereignissen sind verschiedene Aktionen, die der User auslösen kann. Ein typischen Beispiel ist der Klick auf einen Button. Für diesen Fall benötigen wir die Schnittstelle **ActionListener**, die ebenfalls über **implements** eingefügt wird.

```
public class myApplet extends Applet
implements ActionListener
{...}
```

Im Gegensatz zu Applet-weiten Ereignissen wie Mausbewegungen muss der Listener in einem solchen Fall nur auf ein einzelnes Element im Programm achten. Dazu wird allen Steuerelementen die Methode **addActionListener()** beigefügt, die einen Listener auf dieses Objekt ansetzt.

```
Button rotB = new Button("Rot");
this.add(rotB);
rotB.addActionListener(this);
```

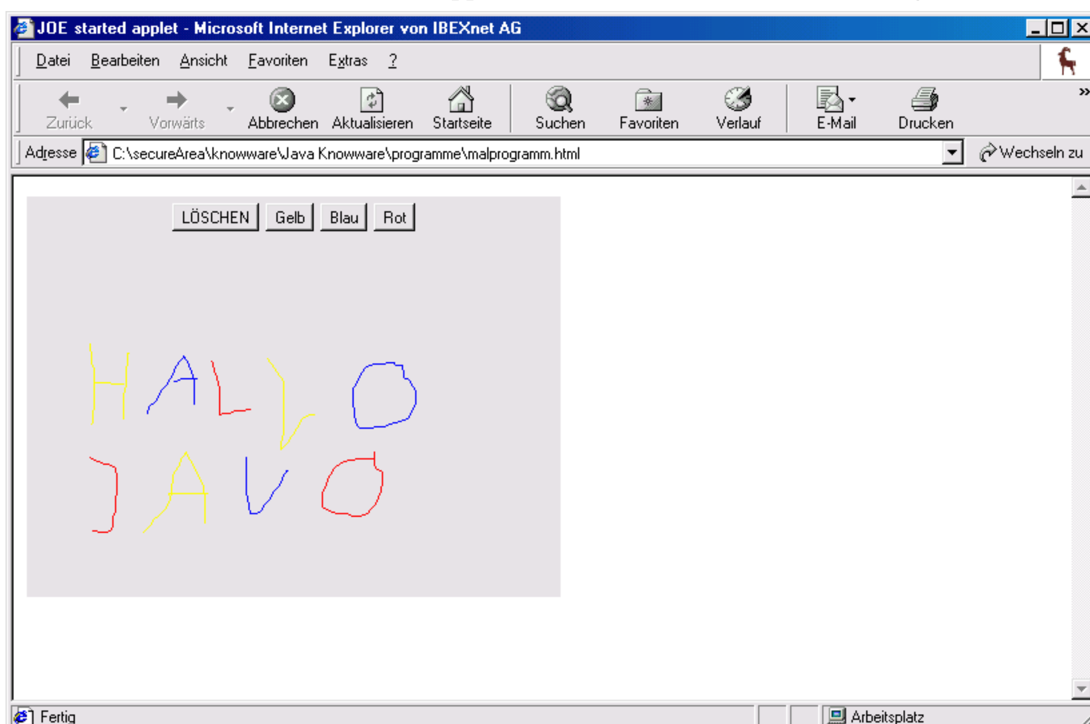
Diese Zeilen zeigen die typische Vorgangsweise für einen Button: Zunächst wird er erstellt, dann gesetzt und schließlich mit einem Actionlistener versehen. Klickt der User ihn an, ruft der Listener die Methode **actionPerformed()** auf, die als Parameter ein Objekt der Klasse ActionEvent übernimmt. Anhand dieses Objektes ist eine Unterscheidung mehrerer Buttons möglich. Die Methode **getActionCommand()** gibt den String zurück, der bei der Initialisierung des Buttons als Aufschrift an den Konstruktor übergeben wurde.

```
public void
actionPerformed(ActionEvent e)
{
String Klick = e.getActionCommand();
}
```

Die Variable Klick würde die Aufschrift des Buttons speichern, so dass man über eine simple if-Abfrage den Button identifizieren kann.

Beispiel für Events: Ein einfaches Malprogramm

Wie versuchen uns nun an einem einfachem Malprogramm in einem Applet. Das Ergebnis unserer Programmierkünste ist ein nettes kleines Applet, dass eine Webseite etwas bunter gestalten kann.



```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class malprogramm extends Applet implements MouseListener,
ActionListener, MouseMotionListener
{
    int X = 0;
    int Y = 0;
    int prevX = 0;
    int prevY = 0;
    int farbe = 0;

      public void init()
      {
        addMouseListener(this);
        addMouseMotionListener(this);

        setBackground(Color.lightGray);

            Button delB = new Button("LÖSCHEN");
            Button gelbB = new Button("Gelb");
            Button blauB = new Button("Blau");
            Button rotB = new Button("Rot");
        this.add(delB);
        delB.addActionListener(this);
        this.add(gelbB);
        gelbB.addActionListener(this);
        this.add(blauB);
        blauB.addActionListener(this);
        this.add(rotB);
        rotB.addActionListener(this);
      }

      public void mousePressed(MouseEvent e)
      {
            prevX = e.getX();
            prevY = e.getY();
      }

      public void mouseDragged(MouseEvent e)
    {
```

```
        X = e.getX();
        Y = e.getY();
        Graphics g = getGraphics();
        if (farbe == 1) {g.setColor(Color.yellow);}
        if (farbe == 2) {g.setColor(Color.red);}
        if (farbe == 3) {g.setColor(Color.blue);}
        g.drawLine(X, Y, prevX, prevY);
        prevX = X;
        prevY = Y;
    }

    public void actionPerformed(ActionEvent e)
    {
        String Klick = e.getActionCommand();
        Graphics g = getGraphics();
        if(Klick == "LÖSCHEN") {repaint(); farbe = 0;}
        if(Klick == "Gelb") {farbe = 1;}
        if(Klick == "Rot") {farbe = 2;}
        if(Klick == "Blau") {farbe = 3;}
      }
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e)  {}
    public void mouseMoved(MouseEvent e)   {}
    public void mouseReleased(MouseEvent e){}

}
```

Zwei Seiten Code wirken zunächst sicher etwas erschlagend – aber das ganze ist halb so wild. Wir beginnen mit einer Klassendeklaration, die die drei Schnittstellen **MouseListener**, **MouseMotionListener** und **Actionlistener** fasst. Das erfordert die Aufnahme einer ganzen Reihe von Methoden in die Klasse, die eigentlich das ganze Programm ausmachen. Einzige Ausnahme ist die Methode **init()**, in der wir die nötigen Listener installieren, gefolgt von vier Buttons, die die Programmkontrolle erleichtern. Jeder Button wird mit einem eigenen Actionlistener versehen. Das eigentliche Programm besteht aus drei Methoden, die die Usereingaben verarbeiten. Die erste Methode wird bei Druck auf die Maustaste aufgerufen. Sinn von **mousePressed()** ist es, die aktuelle Position in zwei Variablen zu speichern. Wird die Maustaste festgehalten und dabei bewegt, ist es möglich, im Applet zu zeichnen. Das wird durch die Methode **mouseDragged()** realisiert, die jeweils die vorherigen Mauskoordinaten mit den jetzigen verbindet. Damit wir nicht den Umweg über die **paint()**-Methode nehmen müssen, wird hier kurzerhand ein **Graphics**-Objekt erzeugt. Die letzte Methode steuert die vier Buttons im Applet, die eine Farbauswahl ermöglichen. Wird einer der Buttons geklickt, wird die Steuervariable auf einen **int**-Wert gesetzt, der die Farbe in der **mouseDragged()**-Methode neu setzt. Der Button „Löschen“ ruft die einzige neue Methode in diesem Programm auf: Mit **repaint()** wird das Applet komplett gelöscht.

Applets und Animationen

Willst du mehr Leben in ein Applet bringen, kannst du mit Animationen arbeiten. Da das Thema Animationen mit Java leicht ausufern kann, werden wir in diesem Kapitel nur auf ein paar grundlegende Überlegungen eingehen: Was ist eine Animation, und wie wird sie in Java realisiert?

Die erste Frage ist relativ leicht zu beantworten: Etwas soll sich auf dem Bildschirm bewegen. Die zweite Frage ist schon komplizierter, da wir uns in einem Multitaskingsystem bewegen.

Java muss neben der Animation gleichzeitig auch noch auf eventuelle Ereignisse des Betriebssystems achten, um z.B. das Fenster rechtzeitig neu zu malen oder das Programm zu beenden. Soll dies nebeneinander funktionieren, benötigen wir Threads.

Den praktischen Einsatz von Threads haben wir bereits auf Seite 58 kennen gelernt, jetzt werden wir ein Applet Multithreading-fähig machen. Dabei gehen wir nicht anders von, als bei einer einfachen Applikation.

```
import java.applet.*;
import java.awt.*;
public class multiapplet extends
Applet implements Runnable
{...}
```

Die Klasse **multiApplet** wird aus der Klasse **Applet** abgeleitet und um die **Runnable**-Schnittstelle erweitert.

> Nicht vergessen – vorher die richtigen Pakete in das Programm importieren!

Die **start()**- und **stop()**-Methoden im Applet werden auf die reine Steuerung des Threads reduziert und sehen danach etwa so aus:

```
public void start()
{
if(mThread == null)
   {
          mThread = new Thread(this);
          mThread.start();
   }
}

public void stop()
{
   if(mThread != null)
   {
      mThread.stop();
      mThread = null;
   }
}
```

Es wird jeweils überprüft, welchen Status der Thread hat, um ihn gegebenenfalls zu starten oder zu stoppen. Die Methode *Thread.***stop()** provoziert unter Umständen eine Warnung beim Kompilieren, die du allerdings ignorieren kannst. Den Grund dafür erfährst du auf Seite 60.

Nachdem wir dieses Programmgerüst aus den verschiedenen Appletmethoden geschaffen haben, können wir uns nun dem eigentlichen Programm widmen, dass sich zum größten Teil in der Methode **run()** abspielt. Hier arbeitet Java die eigentliche Animation ab, die in der Regel über eine Schleife realisiert wird.

```
public void run()
{
   while(true)
      {
      //Berechnung der Animation
      repaint();
      }
}
```

Sobald der Animationsthread gestartet wird, läuft er in eine Endlosschleife, die erst aufgehalten werden kann durch die Methode **`stop()`** des Applets (da hier der Thread gestoppt wird). Innerhalb der Schleife wird dann die eigentliche Animation berechnet, die normalerweise aus Koordinaten oder Farbänderungen zusammengesetzt wird. Sind die entsprechenden Steuervariablen neu gesetzt, wird die **`paint()`**-Methode über **`repaint()`** aufgerufen, die zuvor den Bildschirm löscht.

Oft muss eine sogenannte Framebremse in das Programm eingebaut werden, damit die Animation nicht zu schnell abläuft. Der simpelste Weg ist hier, den Thread ein paar Millisekunden schlafen zu lassen.

```
try{mThread.sleep(5);}
catch(InterruptedException e)
{System.out.println("Fehler: " +
e);}
```

Da **`sleep()`** eine Ausnahmebehandlung braucht, muss diese Anweisung in einem **`try()-catch()`**-Block aufgerufen werden. Sie sollte nach dem **`repaint()`**-Aufruf stehen.

Der letzte und wohl wichtigste Schritt ist die Methode **`paint()`**. Hier wird die Animation auf den Bildschirm gebracht, denn **`paint()`** ist für die bildweise Ausgabe der Animation zuständig.

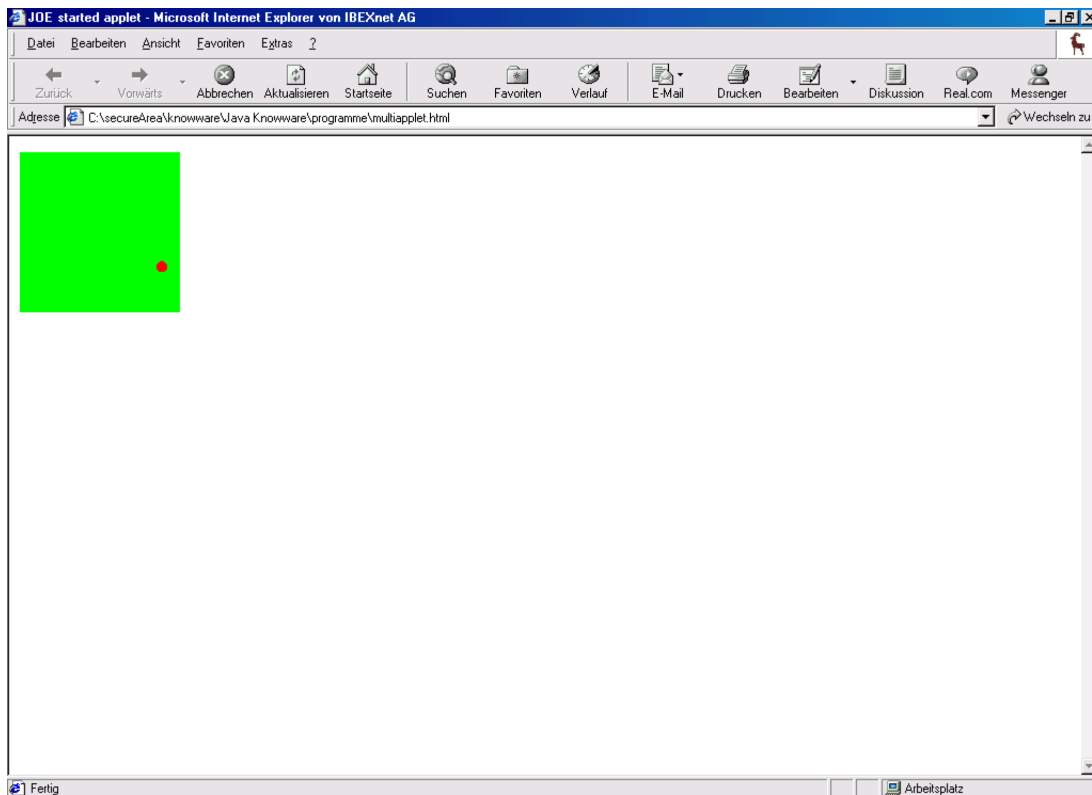
Eine kleine Animation, die das Arbeiten von Threads in Applets zeigt, demonstriert das Zusammenspiel der verschiedenen Methoden.

```
import java.applet.*;
import java.awt.*;

public class multiapplet extends Applet implements Runnable
{
      //Steuervariablen werden erzeugt
      Thread mThread;
      //Startpunkte
      int x = 50;
      int y = 100;
      //Richtunsvariablen
      int dirX = 0;
      int dirY = 0;

      public void init()
      {
            setBackground(Color.green);
      }

      public void start()
      {
      //Thread wird erstellt
      if(mThread == null)
```

```
        {
            mThread = new Thread(this);
            mThread.start();
        }
    }

    public void stop()
    {
          //Thread wird zerstört!
        if(mThread != null)
        {
                mThread.stop();
                mThread = null;
        }
    }

    public void run()
    {
    //Endlosschleife
        while(true)
          {
          //Weg wird bestimmt!
                if(dirX == 0) {x += 1;}
                if(dirX == 1) {x -= 1;}
                if(dirY == 0) {y += 1;}
                if(dirY == 1) {y -= 1;}

                //neue Richtung wird bestimmt!
                if(x == 140) {dirX = 1;}
                if(x == 0) {dirX = 0;}
                if(y == 140) {dirY = 1;}
                if(y == 0) {dirY = 0;}

                repaint();
                //Pause von 5 Millisekunden!
                try{mThread.sleep(5);}
                catch(InterruptedException e)
                {System.out.println("Fehler: " + e);}
        }
    }
```

```
        public void paint(Graphics g)
        {
              //grafische Ausgabe
           g.setColor(Color.red);
           g.fillOval(x, y, 10, 10);
        }
}
```

Das Applet zeichnet einen grünes Quadrat auf den Bildschirm, in dem ein roter Kreis hin- und herspringt. Dabei prallt er von den Wänden ab und wird jeweils in die entgegengesetzte Richtung weitergeleitet. Keine besonders aufregende Animation, aber ein Anfang. Wer sich über das mögliche Flackern (vor allem auf langsameren Rechnern) ärgert, der muss wissen, dass dies nicht der Weisheit letzter Schluss ist. Java bietet eine Reihe von Techniken, Animationen sauber auf den Bildschirm zu bringen, allerdings wird die Grundlage immer so eine Threadkonstruktion sein. Wer auf den Geschmack gekommen ist, kann im Internet unter den Stichwort „Double Buffering“ weitere Informationen finden.



Parameter an ein Applet

Wie wir schon im ersten Kapitel über Applets auf Seite 58 sahen, kann man über den HTML-Code Parameter an ein Applet übergeben. Die Syntax auf der HTML-Seite kennen wir bereits. Hier beschäftigen wir uns mit der anderen Seite, nämlich dem Applet selbst.

Um die Parameter auslesen zu können, bietet die Klasse **Applet** die Methode **getParameter()**, die als eigenen Parameter den Namen des Parameters auf der HTML-Seite als String übernimmt. Fieser Satz, ist aber ganz einfach!

Nehmen wir an, wir haben unser Applet wie folgt eingebunden:

```
<applet code="myApplet.class"
width=150 height=150>
<param name="var1" value="Hallo">
<param name="var2" value="Java!">
</applet>
```

Es existieren also zwei Parameter mit den Namen **var1** und **var2**, die jeweils einen String an das Applet übergeben. Um diese Werte auszulesen, geht man wie folgt vor:

```
String var1 = getParameter("var1");
String var2 = getParameter(„var2");
```

Die String-Variablen **var1** und **var2** enthalten nun die Werte der Parameter, und du kannst im Programm mit ihnen Arbeiten.

Diese Technik ist sehr nützlich, wenn man sein Applet flexibel an eine Webseite anpassen will, ohne den Code ständig neu zu kompilieren. So bietet sich z.B. an, die Hintergrundfarbe eines Applets derart festzulegen, damit es auf verschiedenen Webseiten eingesetzt werden kann.

Das Sandkastenprinzip

Applets sind von sich aus nicht lauffähig; sie benötigen die Umgebung eines Browsers oder Appletviewers, um zu funktionieren. Im Gegensatz zur statischen HTML-Programmierung steht für ein Applet aber eine komplette Bibliothek von Systemmethoden und Dateifunktionen bereit, die den direkten Eingriff in das Innenleben des Rechners erlauben. So ist es in Java problemlos möglich, Dateien zu löschen oder sensible Daten von der Festplatte zu lesen und über das Internet zu publizieren.

Selbst ein hartgesottener Surfer wird bemerken, dass dies gewisse Gefahren in sich birgt, da ein Applet *automatisch* auf den Rechner geladen und ausgeführt wird. Rufst du eine Webseite auf, die ein Applet im Layout hat, wird es ohne weitere Anfrage lokal in deinem System gestartet. Diese Funktion kann man zwar abstellen –,die meisten Browser haben diese Einstellung aber als Standard aktiviert. Böse Menschen könnten durchaus auf die Idee kommen, Applets zu bauen, die deine Festplatte formatieren, sobald du eine bestimmte Webseite aufrufst. Keine gute Idee, wenn man sich die Sache genauer überlegt.

Um das zu verhindern, wurde das Sandkastenprinzip für Browser entwickelt, das die Funktionalität eines Applets stark einschränkt. Alle Browser, die Javaapplets unterstützen, schränken die Möglichkeiten des Programms auf den Bereich des Browserfensters ein. Der Zugriff auf lokale Daten (Festplatte oder andere Laufwerke) ist unterbunden, genau wie Lesen und Schreiben von Systemdateien. Abgesehen von eventuellen Sicherheitslücken der verschiedenen Browser betrifft diese Restriktion folgende Punkte:

- Zugriff auf die Betriebsmittel des lokalen PCs
- Auslesen von Informationen
- Netzwerkkontakte
- Ausführen von lokalen Programmen

Jeder wird einsehen, dass diese Maßnahmen um der Sicherheit im Internet willen sinnvoll sind.

Eigenständige Javaapplikationen können ohne Einschränkung die komplette Java-API nutzen.

Java im Browser

Dieses Kapitel soll einen kurzen Einblick in die Javaumgebung innerhalb eines Browsers geben. Im Gegensatz zu den Applikationen, die wir im ersten Teil des Heftes geschrieben haben, werden Applets unabhängig vom JDK ausgeführt. Das ist auch notwendig, denn sonst würden Applets nur auf Rechnern laufen, die das JDK auch installiert haben. Selbst nach diesem Heft dürfte das nur ein verschwindend geringer Prozentsatz sein.

Will man Java dennoch im Internet nutzen, benötigt man einen javatauglichen Browser mit dem entsprechenden Plug-In. Die meisten modernen Browser unterstützen Java, auch wenn einige Versionen auf das Plug-In verzichten, um Speicherplatz zu sparen.

Mit dem *Microsoft Explorer* oder dem *Netscape Communicator* bist du auf jeden Fall eingedeckt. Der Browser von Opera bietet zwei verschiedene Versionen zum Download an. Die Version mit Java ist 9 MB größer, die sich allerdings lohnen.

Fehlt das Java Plug-In im Browser, kann es nachträglich geladen werden. Spätestens auf einer Seite mit Javaapplets wirst du darauf hingewiesen.

Einstellungen im Browser

Der *Microsoft Explorer* bietet mehrere Möglichkeiten, zur Steuerung von Applets. Die einzelnen Punkte stehen unter EXTRAS|INTERNETOPTIONEN. Hier wählst du ERWEITERT und scrollst bis zum Punkt Microsoft VM, was für Virtual Machine steht, dem Java Interpreter von Microsoft. Dort findest du drei Punkte zur Steuerung von Java. Der erste Punkt „Java-Compiler aktiviert“, sollte angehakt sein – hier wird die Unterstützung von Applets ein- und ausgeschaltet. Der nächste Punkt steuert die Java-Console innerhalb des Browsers, über die ein Applet kontrolliert werden kann. Hast du schon einmal versucht, über den Befehl `System.out.println()` einen String in einem Applet auszugeben, hast du dich sicher gefragt, wohin die Ausgabe verschwand. Nun – sie erscheint in der Console, die du unter ANSICHT|JAVA-BEFEHLSZEILE findest.

Der letzte Punkt aktiviert eine Protokollfunktion im Browser, die alle Aktivitäten eines Applet aufzeichnet. So können auch im nachhinein die Ausgaben eines Applets nachvollzogen werden.

Der *Netscape Navigator* bietet ähnliche Funktionen, wenn auch nicht in dieser Ausführlichkeit. Unter BEARBEITEN|EINSTELLUNGEN findest du unter ERWEITERT „Java aktivieren“, was die Javaunterstützung ein- und ausschaltet.

Im Gegensatz zu Microsoft hat Netscape keinen eigenen Java-Interpreter gebastelt, um ihn im Browser zu integrieren. Im Navigator wird die originale Virtual Machine von SUN verwendet, die die Unterstützung der kompletten Java-API garantiert. Der Unterschied zur Microsoft VM ist allerdings gering und fällt nur selten ins Gewicht. Allerdings werden neue Möglichkeiten dadurch nicht immer von Microsoft unterstützt. Oft dauert es eine Weile, bis sie hinzugefügt werden.

Applets in HTML 4.0

Auf Seite 62 lernten wir, wie man Applets über das Tag **<APPLET>** in eine Webseite einbindet. Dieser Befehl stammt noch aus der Ära von HTML 3.2, was allerdings nicht heißt, dass er veraltet wäre. Wer es allerdings etwas flexibler mag, für den hat HTML 4.0 das Tag **<OBJECT>** erweitert. Zuvor wurde es für die Einbindung von ActiveX-Komponenten oder Videostreams verwendet. Dank zahlreicher Attribute lässt sich das Applet hiermit sehr viel genauer steuern als mit **<APPLET>**.

classid

Hiermit wird das Applet eingebunden.

```
<Object classid =
„java:myApplet.java“>
```

Wichtig ist das Schlüsselwort **java** vor dem eigentlichen Appletnamen. So weiß der Browser, dass es sich um ein Applet handelt.

standby

Dieses Attribut ermöglicht die Generierung eines Statuszeilentextes im Browser, der beim Laden des Applets angezeigt wird.

align

Legt fest, wie sich das Applet auf der Seite ausrichtet. Möglich sind folgende Werte:

- middle
- left
- right
- center
- textbottom
- texttop
- textmiddle

height

Legt die Höhe des Applets fest.

width

Legt die Breite des Applets fest.

hspace

Legt den Abstand links und rechts zwischen dem Applet und dem nächsten Element der Seite fest.

hspace

Legt den Abstand oben und unten zwischen dem Applet und dem nächsten Element der Seite fest.

Was zum Teufel heißt deprecated?!

Java ist eine recht junge Sprache und hat als solche die schätzenswerte Eigenschaft, sich mehr oder weniger regelmäßig weiter zu entwickeln. Auf der anderen Seite bedeutet das natürlich, dass Klassen und Methoden aus früheren Versionen irgendwann veraltet sind und durch neue ersetzt werden. Mit jeder neuen Augabe des JDK werden bestimmte Teile von Java überarbeitet und nach und nach ersetzt. Damit alte Programme nicht ihre Gültigkeit verlieren und weiterhin funktionieren, unterstützen alle neuen Versionen des JDK die veralteten Methoden. Damit der Programmierer aber merkt, dass er nicht mehr „up to Date“, gibt der Compiler eine Warnung aus, die zeigt, dass er veraltet Techniken benutzt.

```
test.java uses or overrides a
deprecated API.
Note:
Recompile with -deprecation for
details.
```

So oder so ähnlich wird der Hinweis aussehen. Java bemerkt, dass der Programmierer eine Methode benutzt, die von SUN als **deprecated** angesehen wird, also als überholt. Derartige Bestandteile werden nicht mehr weiter entwickelt und sollen nicht mehr genutzt werden. Startet man den Compiler mit –**deprecation**, werden alle veralteten Teile angezeigt.

Da es für jede überholte Methode eine neue – und wahrscheinlich auch bessere – Methode gibt, lohnt sich eigentlich immer ein Blick in die aktuelle Java-API, um die neuen Techniken kennenzulernen. Ich gebe allerdings zu, dass es manchmal nerven kann, unerwartet über diese Meldung zu stolpern, wenn man sich gerade an eine Methode gewöhnt hat.

Weitere wichtige Pakete

Die Java-API besteht aus einer ganzen Reihe von Paketen, die wir in diesem Heft nicht alle erwähnen können. Damit du einen kleinen Überblick bekommst, werde ich hier kurz die wichtigsten Pakete vorstellen.

java.net

Das Paket **java.net** enthält Klassen und Schnittstellen für Netzwerkentwicklung im Internet und Intranet.

java.beans

Dieses Paket ermöglicht die Entwicklung wieder verwendbarer Komponenten für unterschiedliche Programme.

java.security

Wie der Name sagt, enthält dieses Paket Routinen für eine erhöhte Sicherheit innerhalb von Applikationen und Applets.

java.rmi

Ein Paket, über das man streiten kann, denn rmi erlaubt die „Verteilung“ eines Programms über ein Netzwerk. Die verschiedenen Klassen werden über ein eigenes Netzwerkprotokoll angesprochen und so auf andere Rechner ausgelagert.

java.sql

Dieses Paket ist ermöglicht die Anbindung an SQL-Datenbanken. In Verbindung mit JDBC (Java Database Connectivity) erlaubt Java den Kontakt zu fast allen Datenbanktypen.

javax.servlet

Dieses Paket ermöglich die Entwicklung von Applikationen, die serverseitig laufen, und so die Möglichkeiten des Common Gateway Interface zu nutzen. Diese Technik setzt allerdings eine Virtual Machine auf dem Server voraus.

Wichtige Seiten im Netz

`www.sun.com` : Die Seite der Macher von Java. Hier gibt es News rund um die Sun Cooperation.

`java.sun.com` : Die offizielle Javaseite. Hier gibt es die jeweils neuesten Versionen von Java und allerlei Tipps und Tricks rund um Java.

`java.sun.com/j2se/1.3/docs/api/` : Die Seite der aktuellen Java-API. Hier findest du alle Informationen zur JDK Version 1.3.

`java.sun.com/j2se/` : Der direkte Link zu neuesten Java Version. Hier werden alle neuen Releases rund um Java gelistet.

`kulturbrand.de` : Meine Seite. Hier findest du alle besprochenen Programme zum Download, sowie einiges mehr ☺

`java.de` : Deutschlands größte Java Community.

Brauchst du mehr? Füttere einfach eine beliebige Suchmaschine mit dem Stichwort „Java".

abgeleitet, 43
Ableitung, 47
abstrakte Klasse, 46
Animation, 55; 74
Anweisungsblockes, 37
Applet, 9; 60
Appletprogrammierung, 62
Appletviewer, 58
Applikation, 9
Applikationen, 58
Array, 39
ASCII, 5
Autoexec.bat, 7
AWT, 62
Bauplan, 62
BBedit, 6
Betriebssystem, 54
Bilder, 66
Bildern, 66
break, 38
Browser, 6; 78
Buchstaben, 49
Button, 68
Byteformat, 50
C++, 9
Call-by-Value, 42
class, 14
Component, 61
Dackel, 10
Datenströmen, 50
Datum, 51
deprecated, 80
Destruktoren, 24
Dimensionen, 41
DOS-Fenster, 6
Download, 4
Elemente, 39
Entscheidung, 33
Entwicklungsumgebung, 7
Ereignisbehandlung, 69
Ereignissen, 71
Eventfunktionen, 70
Eventpolitik, 69
Fallunterscheidung, 35
false, 33
Fehlermeldungen, 50
Garbage Collector, 24
goto, 14
Grafik, 63
Hallo Welt, 7
HTML, 59
HTML 4.0, 79
IDE, 5
importieren, 48
Instanzmethoden, 15
Instanzvariablen, 15
Javaapplet, 79
JDK, 5
JOE, 5; 58
Kapselung, 10
Klassendeklaration, 15
Klassenmethoden, 15
Klassenvariablen, 15
komplexe Datentypen, 42
komplexe Datentypen, 39
Konstruktor, 23
Koordinaten, 75
Länge eines Arrays, 40
Layout Managers, 68
Liste, 39
Listener, 69
Mac, 6
Maschinencode, 9
Maustaste, 70
Mehrfachvererbung, 11; 45
Methoden, 19
multitaskingfähig, 55
Multitaskingsystem, 54; 74
Mutterklassen, 11
new, 18
Notepad, 53
OAK, 4
Object, 14
Objekte, 10
Objektvariablen, 20
OOP, 9
Operator, 32
Paketnamen, 48
Parameterübergabe, 25
path, 7
Plattformunabhängigkeit, 4; 63
Programmabbruch, 52
Prototyp, 19
Prozessor, 54
Punktnotation, 29
Quelltext, 5
return, 20
Runtime, 48
Schleife, 37
Schleifendurchgang, 37
Schleifenkopf, 38
Schlüsselwörtern, 16
Schnittstellen, 69
Schriften, 65
Semikolon, 14
Server, 59
Sourcecode, 8
Speicherzustand, 49
Stack, 51
Standardausgabe, 50
Standardeingabe, 50
Standardkonstruktor, 26
Standardschrift, 65
Standardtypen, 13
Startparameter, 41
Steuerlemente, 67
Steuermethoden, 23
Struktursyntax, 14
Sun Microsystems, 4
Superklasse, 43
switch, 35
Systemmethoden, 78
Tasks, 54
Tellerstapel, 51
Thread, 54
Timestamp, 51
true, 33
TrueColor, 64
Typen, 13
überholte Methode, 80
Überladen, 27
und Call-by-Reference, 42
UNIX, 6
Variable, 13
veraltete Methoden, 80
Vererbung, 10
Vererbungspyramide, 44
Verkettung, 49
Webpage, 58
weitere Pakete, 80
Windowsprogramm, 53
Zahlentypen, 32
Zeichen, 49
Zeichenketten, 13
zeichnen, 68
Zugriffsrechte, 29