3 Das erste Programm

Die wichtigste Aufgabe des Programmierers, die Tätigkeit, die (hoffentlich) den größten Teil seiner Arbeitszeit ausmacht, die ihn ebenso begeistert wie ermüdet, die ihn höchste Ekstase und tiefste Depressionen durchleben lässt, die ihm Fluch und Leidenschaft ist – das ist das Aufsetzen der Programmquelltexte. Die nachfolgende Umwandlung dieser Quelltexte in ausführbare Programme ist dagegen nur eine Formalität. Immerhin sie muss gemacht werden. Wie dies geht und welche Hilfsmittel Ihnen dabei zur Verfügung stehen, ist Thema dieses Kapitels.

3.1 Programmerstellung in Java

Java-Programme müssen kompiliert und interpretiert werden. Umsteigern von rein kompilierten (C++, Pascal...) oder rein interpretierten Sprachen (Perl...) mag dies umständlich erscheinen, doch die Gründe liegen auf der Hand: Durch geschickte Kombination können die Vorteile beider Techniken genutzt werden.

Portierbarkeit und Robustheit

Java blickt auf eine recht bewegte Entwicklungsphase zurück, in der es schon zu manchen Verwerfungen, Kehrtwendungen und Neuorientierungen kam. Treu geblieben ist sich die Sprache aber stets in dem Ziel, eine Sprache für Programme in Netzwerken zu sein.

Anfangs – zu Zeiten von Oak – sollten dies Anwendungen sein, die vernetzte Verbrauchergeräte intelligenter machen, später – als aus Oak Java wurde – kam die erfolgreiche Neuausrichtung auf das World Wide Web und die Erstellung von Programmen für Webseiten (Applets). Heute können Sie mit Java »normale« Anwendungen, über Netzwerke verteilte Anwendungen, Applets und Programme für technische Kleingeräte (beispielsweise Handys und PDAs) schreiben.

Da Netzwerke nicht selten Endgeräte mit den unterschiedlichsten Plattformen (Kombination von Prozessor und Betriebssystem) verbinden, war von vornherein klar, dass Java interpretiert werden musste.

Plattformunabhängigkeit durch Interpretation Exkurs

Compiler übersetzen den Quelltext eines Programms traditionell in prozessor- und betriebssystemspezifischen Maschinencode. Meist ist dies der Maschinencode des aktuellen Systems, auf dem der Programmierer seinen Code erstellt.[1]

Stellen Sie sich nun vor, Sie würden mit einem solchen Compiler ein Applet für die Intel/Windows-Plattform erstellen. Das Applet betten Sie in eine Webseite ein und veröffentlichten diese im Web. In kürzester Zeit würden Sie eine große Zahl begeisterter E-Mail-Zuschriften von Webbesuchern erhalten, die die Webseite mit dem Applet-Code auf ihren Rechner heruntergeladen haben und, da sie keinen Intel/Windows-Rechner verwenden, statt der Ausgabe des Applets nur eine leere Fläche sehen – vielleicht noch mit einer Fehlermeldung, dass der Appletcode auf diesem Rechner nicht ausgeführt werden kann.

Die Lösung dieses Problems liegt darin, den Quelltext erst auf dem Rechner des Anwenders (hier der Websurfer) in plattformspezifischen Maschinencode zu übersetzen. Dies leistet der Interpreter. Voraussetzung ist natürlich, dass auf dem System des Anwenders ein passender Interpreter installiert ist.

Der Java-Interpreter ist ein kleines Programm namens *java*, das es für alle wichtigen Plattformen gibt. Allein mit dem Java-Interpreter können Java-Programme aber noch nicht ausgeführt werden.

Die Java-Standardbibliothek

Software, die über CD oder DVD ausgeliefert wird, unterliegt praktisch keinerlei Größenbeschränkung. Wenn Sie jedoch Programme schreiben, die der Anwender vor jeder Ausführung über das Internet herunterlädt, wie es beispielsweise bei den Applets der Fall ist, wird die Codegröße zu einem ganz entscheidenden Faktor. Wie aber lässt sich die Codegröße klein halten?

Heute gehört zu praktisch jeder Programmiersprache eine umfangreiche Standardbibliothek mit vordefinierten Funktionen oder Klassen, auf die der Programmierer in seinen Quelltexten zurückgreifen kann. Die Java-Standardbibliothek enthält beispielsweise Klassen für die Ein- und Ausgabe, für mathematische Berechnungen, für die GUI-Programmierung und vieles mehr. In traditionellen kompilierten Sprachen wird der Bibliothekscode, den der Programmierer in seinem Quelltext verwendet, bei der Programmerstellung in der Regel in die Programmdatei eingebunden. Je mehr Bibliothekselemente der Programmierer verwendet, umso größer wird die Programmdatei.

Um diese »unnötige« Aufplusterung der Programmdatei zu vermeiden, wird in Java der Bibliothekscode erst vom Interpreter eingebunden. Dies setzt natürlich voraus, dass die Standardbibliothek zusammen mit dem Java-Interpreter auf dem Rechner des Anwenders installiert ist.

Robustheit

Gibt es fehlerfreie Programme? Denkt man an die Produkte mancher Software-Häuser, so möchte man dies bezweifeln. Doch woran liegt es, dass hundertfach getestete Programme immer noch Fehlfunktionen aufweisen, ja sogar abstürzen? Die Gründe sind verschieden, müssen nicht einmal im Verantwortungsbereich der Programmierer liegen. Auf jeden Fall aber spielt die zunehmende Komplexität der Programme

1 Will der Programmierer seine Programme für andere Plattformen erstellen, muss er dazu Cross-Compiler (laufen auf einer Plattform, kompilieren für eine andere) verwenden oder sich für jede Plattform einen eigenen Entwicklungsrechner einrichten.

eine entscheidende Rolle – und die Frage, inwieweit die Sprache dem Programmierer hilft, mit wachsender Komplexität fertig zu werden.

Javas Meinung zu diesem Punkt könnte klarer nicht sein: Die Verantwortung für eine sichere, fehlerfreie Programmierung kann nicht allein dem Programmierer überlassen werden, die Sprache selbst muss den Programmierer soweit es geht entlasten, bzw. ihn zu sicherer, defensiver Programmierung anhalten.

Java setzt dies vorbildlich um. Nicht nur durch die konsequente Objektorientierung, die Absicherung der Bibliotheksmethoden durch Exceptions, die strenge Typisierung, sondern gerade und vor allem auch durch die automatische dynamische Speicherverwaltung – eine stete Fehlerquelle –, die nahezu vollständig aus den Händen des Programmierers in den Verantwortungsbereich des Interpreters gegeben wurde.

Javas dynamische Speicherverwaltung – Objekte werden dynamisch reserviert, der Zugriff auf die Objekte erfolgt über Referenzen, Objekte, auf die keine Referenzen verweisen, werden von der automatischen Speicherbereinigung, der »Garbage Collection« entsorgt – geht übrigens auf die objektorientierte Sprache Smalltalk zurück.

Die Java-Laufzeitumgebung

Das Gesamtpaket aus erweitertem Interpreter (die Java Virtual Machine, s.u.), der Java-Standardbibliothek (runtime classes), dem Java-Plugin für die Ausführung von Applets in Browsern sowie einige weitere Hilfsprogramme, die zum Starten und Ausführen von Java-Code benötigt werden, bezeichnet man als die Java-Laufzeitumgebung (Java Runtime Environment, kurz JRE).

Die JRE gibt es für eine Vielzahl von Plattformen und kann kostenlos von der Sun-Java-Website heruntergeladen werden. Eine aktuelle Version ist auch im Java-SDK enthalten.

Sun hat selbstredend großes Interesse daran, dass die JRE auf möglichst jedem Rechner vorhanden ist (so dass Anwender Java-Programme und Applets problemlos ausführen können). Software-Entwickler dürfen die JRE daher frei, aber bitte unverändert, mit ihren Java-Programmen an die Anwender weitergeben und auf deren System installieren. Bemühungen, die JRE automatisch mit dem neuen Betriebssystem Windows XP auszuliefern, sind vorerst gescheitert. Immerhin wurde Microsoft gerichtlich untersagt, sein Betriebssystem mit einer eigenen (womöglich inkompatiblen) JRE zu vertreiben.

Effizienz und Schutz geistigen Eigentums

Traditionell ist die Interpretation mit zwei schwer wiegenden Nachteilen verbunden:

- Die Programmausführung wird verlangsamt, da der Interpreter die gerade auszuführenden Codeabschnitte immer erst noch übersetzen muss.
- Das Programm wird als Quelltext an den Anwender übergeben. Das Know-how, das womöglich in dem Quelltext steckt, ist dadurch ungeschützt und kann leicht kopiert werden.

Kompilierte Sprachen kennen diese Probleme nicht, da die Übersetzung komplett vor der Auslieferung des Programms erfolgt und das Programm nicht als Quelltext, sondern als binärer Maschinencode vertrieben wird.

Um nun die Vorzüge der Kompilation mit der Notwendigkeit der Interpretation zu verbinden, geht Java einen Zwischenweg: der Java-Quelltext wird kompiliert und interpretiert.

Das Java-Modell und die Virtual Machine

Nachdem der Programmierer den Java-Quelltext fertig gestellt hat, übersetzt er ihn mit Hilfe des Java-Compilers *javac* – allerdings nicht in den Maschinencode eines bestimmten Prozessors, sondern in plattformunabhängigen **Bytecode**. Die resultierenden Class-Dateien (der Compiler schreibt den Bytecode jeder übersetzten Klassendefinition in eine eigene Datei mit der Extension *.class*) stellen das fertige Programm dar und werden vom Programmierer an die Anwender verteilt.

Der Anwender, auf dessen System die passende JRE installiert ist (oder von der Setup-Routine des erworbenen Programms installiert wird), führt das Programm mit Hilfe des Interpreters und den anderen Teilen der JRE aus. Der Interpreter übersetzt den Bytecode, beginnend mit der `main()`-Methode des Programms, schrittweise in plattformspezifischen Maschinencode und lässt diesen ausführen.

Die Interpretation des vorkompilierten Bytecodes ist zwar immer noch langsamer als die direkte Ausführung optimierten Maschinencodes, auf jeden Fall aber um einiges schneller als die Interpretation von reinem Quelltext. Moderne Java-Interpreter, die so genannten Just-In-Time-Compiler (JITter) beschleunigen die Programmausführung, indem sie oft benötigte Programmteile in optimierten Maschinencode übersetzen und im Arbeitsspeicher halten, so dass bei Wiederverwendung während der aktuellen Programmausführung keine erneute Übersetzung notwendig ist. Zusätzlich werden häufig fortlaufend Hot-Spot-Analysen gemacht, d.h. sehr oft[2] durchlaufene Codeabschnitte (die »Hot-Spots«) werden parallel zur Programmausführung intensiv optimiert und dadurch noch schneller. Manche Entwicklungsumgebungen bieten darüber hinaus auch Compiler an, die den Bytecode komplett in Maschinencode übersetzen (wie ein konventioneller C++-Compiler beispielsweise), so dass zur Ausführung gar kein Java-Interpreter mehr benötigt wird (auf Windows-Systemen hat man dann eine normale *.exe* Datei vorliegen).

Der erweiterte Java-Interpreter führt nicht nur den Bytecode aus, er reserviert auch den vom Programm benötigten dynamischen Speicher und gibt nicht weiter benötigten Speicher wieder frei (Speicherbereinigung). Aus Sicht des generierten Bytecodes gleicht er einem virtuellen Rechner, auf dem der Bytecode ausgeführt wird. Der erweiterte Java-Interpreter wird daher auch gemeinhin als die **Java Virtual Machine** (JVM) bezeichnet. Der Bytecode ist demnach der Maschinencode der Java Virtual Machine.

2 Im aktuellen JDK 1.4.2 liegt die Definition von »sehr oft« bei 1500 Durchläufen.

3.2 Installation des Java-SDK

Um Java-Programme erstellen und testen zu können, benötigen Sie grundsätzlich nichts weiter als das Java Development Kit (Java-SDK). Die zum Zeitpunkt der Drucklegung dieses Buches aktuelle Version, Java 2-SDK 1.4.2[3], finden Sie auf der Buch-CD. Ansonsten können Sie sich neuere wie ältere Versionen für die verschiedenen Plattformen von der Java-Site herunterladen (`http://java.sun.com/j2se/downloads.html`[4]). Das Java-SDK enthält sämtliche Tools, die Sie für die Java-Programmierung benötigen.

Die Java-SDK-Tools sind fast ausnahmslos Konsolenanwendungen, deren Bedienung dem einen oder anderen Leser antiquiert vorkommen wird. Wer lieber mit integrierten Entwicklungsumgebungen arbeitet, der findet im Internet eine Reihe von leistungsfähigen Java-Entwicklungsumgebungen namhafter Software-Häuser, von denen es zumeist auch kostenlose Download-Versionen gibt:

- Eclipse von IBM (`www.eclipse.org`). (Die Standardversion 2.1 ist auch auf der beiliegenden Buch-CD enthalten.)
- NetBeans von Sun, Nachfolger von Sun ONE (`www.netbeans.org`). (Die Standardversion 3.5 ist auf der beiliegenden Buch-CD enthalten.)
- JBuilder von Borland (`www.borland.de`). (Die Standardversion 9 ist auch auf der beiliegenden Buch-CD enthalten.)

Ausführlichere Hinweise zur Arbeit mit diesen Entwicklungsumgebungen finden Sie im Anhang. Beachten Sie, dass die meisten Entwicklungsumgebungen letztlich nur eine GUI-Bedienoberfläche zu den Java-SDK-Tools darstellen, die auf jeden Fall auf Ihrem System installiert sein müssen (meist wird das SDK zusammen mit der Entwicklungsumgebung installiert).

Ältere Java-SDK-Versionen benötigen Sie, wenn Sie Java-Programme erstellen möchten, die von älteren Virtual Machines ausgeführt werden können. Wenn Sie beispielsweise mit dem aktuellen Java-SDK Applets für Webseiten schreiben, kann es passieren, dass diese in den Browsern der Webbesucher nicht angezeigt werden, weil diese nicht das aktuelle Java-Plug-in verwenden. Um daraus resultierenden Beschwerde-Mails zu entgehen, können Sie entweder einen Link zum Download des aktuellen Plug-Ins anbieten oder Ihr Applet gleich für eine ältere Virtual Machine kompilieren.

Installieren Sie hierzu die ältere Java-SDK-Version parallel zu ihrer aktuellen Version und führen Sie eine Cross-Kompilierung durch (siehe Anhang zu Java-SDK-Tools).

3 Sun hat beim Sprung von der Version 1.1 zu 1.2 die Terminologie geändert und das JDK (JAva Development Kit) in J2SDK bzw. J2SE-SDK (»Java 2 Standard Edition«-Software Development Kit) umbenannt.

4 Beachten Sie, dass sich die Verzeichnisstruktur von Websites ändern kann. Falls Sie unter Unterverzeichnis */j2se/downloads.html* nichts mehr finden sollten, beginnen Sie einfach mit dem Home-Verzeichnis *java.sun.com* und suchen Sie sich Ihren Weg zum Java-SDK.

Installation

Die Installation des Java Development Kit (Java-SDK) ist relativ einfach. Nachdem Sie das Paket von der Buch-CD kopiert oder von Suns Webserver (`java.sun.com`) heruntergeladen haben, müssen Sie die selbst extrahierende Datei nur noch ausführen und die Fragen zur Lizenz und zur Konfiguration der Installation beantworten.

Windows

Führen Sie die EXE-Setupdatei aus (über den Dialog START/AUSFÜHREN oder durch Doppelklick auf die Datei im Windows Explorer).

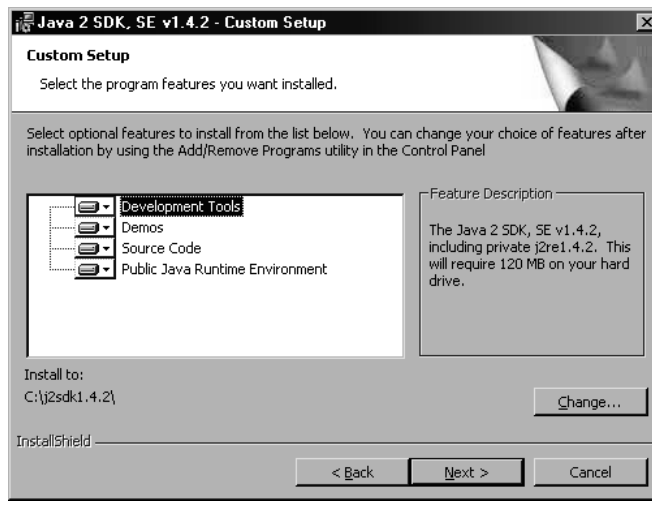


Abbildung 3.1: Auswahl der zu installierenden Komponenten

Lassen die den Java-SDK möglichst vollständig installieren (Voreinstellung, siehe Abbildung 3.1) und gestatten Sie dem Setup-Programm auch die Konfiguration Ihrer Browser zu aktualisieren.

INFO

Wenn Sie nicht genügend Festplattenspeicher zur Verfügung haben, deaktivieren Sie die Option PUBLIC JAVA RUNTIME ENVIRONMENT. Der SDK (Option Development Tools) enthält eine eigene private Version der JRE, die zum Testen der Programme verwendet werden kann. Die Optionen SOURCE CODE und DEMOS können Sie bei Speichermangel ebenfalls deaktivieren. Allerdings spart dies relativ wenig Mbytes, und insbesondere die Quelldateien der Java-API sind für fortgeschrittene Java-Programmierer eine wertvolle Referenzquelle!

Linux

Legen Sie ein Installationsverzeichnis für den SDK an und kopieren Sie in dieses die BIN-Datei. Danach öffnen Sie ein Konsolenfenster, wechseln zum Installationsverzeichnis und lassen die BIN-Datei ausführen.

Die Konfiguration der Installation ist unter Linux derzeit nicht möglich.

Abbildung 3.2: Start der Installation unter Linux/KDE

Deinstallation

Wenn Sie den Java-SDK irgendwann wieder deinstallieren möchten, rufen Sie die mitgelieferte Deinstallationsroutine über START/EINSTELLUNGEN/SYSTEMSTEUERUNG, Symbol SOFTWARE auf.

Wenn Sie für Ihre Browser statt des neu installierten Plug-Ins lieber Ihr altes Plug-In verwenden wollen, klicken Sie in der Systemsteuerung auf das Symbol JAVA PLUG-IN und wählen Sie auf der Dialogseite ERWEITERT das gewünschte Plug-In aus.

Linux-Anwender löschen einfach das Verzeichnis der Java-SDK-Installation.

Anpassen des Systems

Wenn die Installation erfolgreich abgeschlossen worden ist, ist ihre Festplatte um ca. 120 bis knapp 300 Mbytes (je nach den gewählten Installationsoptionen) ärmer und Sie um das Java-SDK der neuesten Version reicher. Jetzt müssen Sie Ihr System nur noch so konfigurieren, dass Sie mit den JDK-Tools bequem arbeiten können.

Erweiterung des Systempfads

Die Java-Tools (insbesondere *javac* und *java*) werden über die Konsole aufgerufen. Wenn Sie dabei nicht immer dem Programmnamen den vollständigen Pfad zur EXE-Datei des Programms voranstellen wollen:

```
Konsolenprompt:> c:\j2sdk1.4.2\bin\javac
```

müssen Sie den Pfad zu den JDK-Tools in den Systempfad des Betriebssystems einfügen. Nehmen wir an, dass Sie das Java-SDK in das Verzeichnis *c:\j2sdk1.4.2* (Windows) bzw. */home/myname/j2sdk1.4.2* (Linux) installiert haben.

- Windows 95/98: Legen Sie zur Sicherheit eine Kopie der Datei *c:\autoexec.bat* an und laden Sie die Datei danach in einen Editor (beispielsweise Notepad, Aufruf über START/PROGRAMME/ZUBEHÖR/EDITOR oder über START/AUSFÜHREN, NOTEPAD eingeben und abschicken). Suchen Sie nach einem PATH-Eintrag und fügen Sie das BIN-Verzeichnis der Java-SDK-Installation hinzu. Zum Beispiel:

  alter Eintrag: `SET PATH=.;c:\dos;`

  neuer Eintrag: `SET PATH=.;c:\dos;c:\j2sdk1.4.2\bin;`

Das Semikolon dient zur Trennung der einzelnen Verzeichnisangaben in PATH.

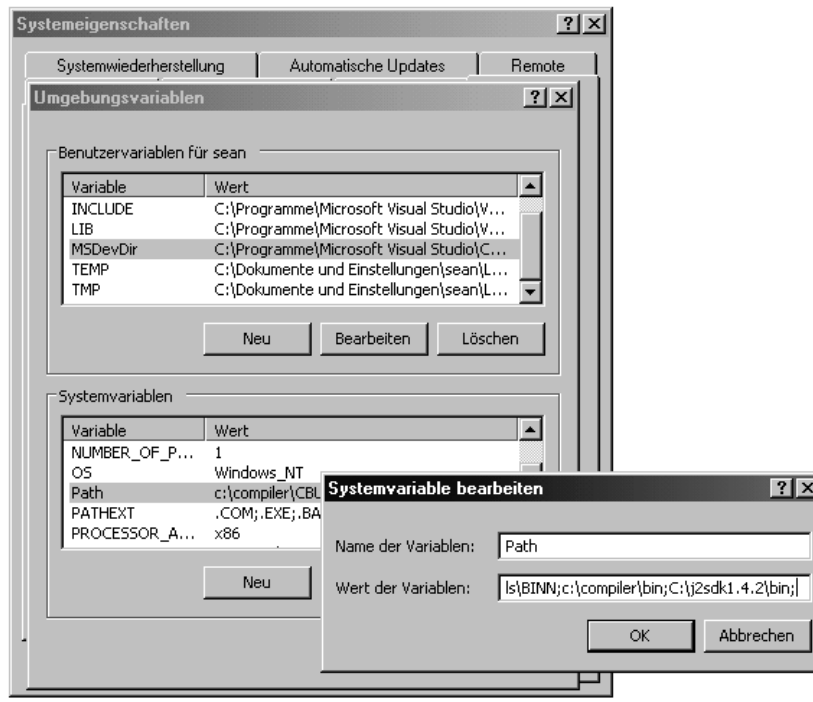
Wenn Sie gar keine PATH-Angabe finden, dann fügen Sie eine neue PATH-Anweisung hinzu, z.B.:

```
SET PATH=c:\j2sdk1.4.2\bin;
```

Windows ME/2000/XP: Die Umgebungsvariable `PATH` wird über den Dialog der Systemeigenschaften verwaltet. Der Weg dorthin ist lang und von Betriebssystem zu Betriebssystem verschieden. Unter Windows Me lautet er START/PROGRAMME/ZUBEHÖR/SYSTEMTOOLS/SYSTEMINFORMATIONEN, Menü TOOLS/SYSTEM KONFIGURATION, Register UMGEBUNG.

Unter Windows 2000 und Windows XP rufen Sie über START oder START/EINSTELLUNGEN die Systemsteuerung auf und gelangen via SYSTEM, Register ERWEITERT, Schalter UMGEBUNGSVARIABLEN zum Ziel. Danach können Sie die Systemvariable auswählen, zum Bearbeiten laden und den Pfad zu den Java-Programmen anhängen (siehe Abbildung 3.3).

Abbildung 3.3: Anpassung der PATH-Variable unter Windows XP



Unix/Linux: Analoges Vorgehen: Suchen Sie die Pfadangabe `path` in der zuständigen ini-Datei (je nach Konfiguration *.login*, *.profile*, *.tcshrc*, *.bashrc* o.ä.) und fügen Sie das Java-Bin-Verzeichnis */home/myname/j2sdk1.4.2/bin* in der nächsten Zeile nach der bisherigen Pfadangabe hinzu.

Für die C-Shell sieht dies beispielsweise wie folgt aus:

```
set path = (/home/myname/j2sdk1.4.2/bin . $path)[5]
```

Für die Bourne-Again-Shell (bash) könnte der Eintrag so lauten:

```
export PATH=/home/myname/j2sdk1.4.2/bin:.:${PATH}
```

oder

```
PATH="/home/myname/j2sdk1.4.2/bin:.:$PATH"[6]
```

Um den neuen Pfad zu testen, müssen Sie den Rechner neu starten und die Konsole öffnen (Windows-Anwender rufen dazu je nach Betriebssystem START/PROGRAMME/MSDOS-EINGABEAUFFORDERUNG, START/PROGRAMME/EINGABEAUFFORDERUNG oder START/PROGRAMME/ZUBEHÖR/EINGABEAUFFORDERUNG auf). Tippen Sie hinter dem Prompt *javac* ein und schicken Sie den Befehl durch Drücken der Eingabetaste ab. Statt einer Fehlermeldung, dass der Befehl nicht gefunden wurde, sollten die Aufrufoptionen des Java-Compilers erscheinen.

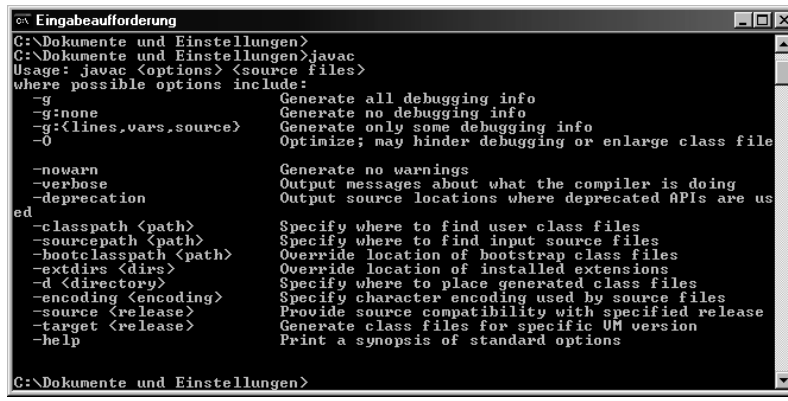


Abbildung 3.4: Der Pfad wurde korrekt aktualisiert, die Java-Tools können von jedem Verzeichnis aus aufgerufen werden.

Setzen des Klassenpfads

Die zweite Einstellung betrifft die CLASSPATH-Variable.

Java-Programme bestehen aus Klassen. Diese sind in Quelltextdateien (Extension *.java*) oder als Bytecode in Class-Dateien (Extension *.class*) definiert. Wenn Sie ein Java-Programm kompilieren oder ausführen, suchen sich die zuständigen Java-Tools (Compiler oder Interpreter) zu allen im Programm verwendeten Klassen die zugehörigen Quelltext- oder Class-Dateien zusammen.

Standardmäßig wird im aktuellen Arbeitsverzeichnis (also von wo aus z.B. der Interpreter aufgerufen worden ist) und in den Class-Archiven *.jar* im LIB-Verzeichnis der Java-SDK-Installation gesucht (gilt erst ab Java-SDK 1.2).

Solange Sie für jedes Programm ein eigenes Verzeichnis anlegen und die Quelltextdateien des Programms zusammen in diesem Verzeichnis speichern, gestaltet sich die Programmerstellung und -ausführung daher recht einfach.

5 Mehrere Verzeichnisangaben werden durch Leerzeichen getrennt.
6 Mehrere Verzeichnisangaben werden durch einen Doppelpunkt getrennt.

Probleme gibt es meist nur dann, wenn irgendein anderes Programm, bei der Installation die CLASSPATH-Variable gesetzt hat. Dann suchen die Java-SDK-Tools nämlich nicht mehr im aktuellen Verzeichnis – es sei denn, das aktuelle Verzeichnis – symbolisiert durch einen Punkt (.) – wurde explizit in der Umgebungsvariable mit aufgeführt. Meist ist dies aber nicht der Fall, so dass Sie den Eintrag für das aktuelle Verzeichnis selbst anhängen müssen.

alter Eintrag: `SET CLASSPATH=c:\programme\interbase`
neuer Eintrag: `SET CLASSPATH=c:\programme\interbase;.`

- Windows 95/98: Setzen oder passen Sie die CLASSPATH-Variable in der *autoexec.bat*-Datei:

  ```
  SET CLASSPATH = c:\programme\interbase;.
  ```

- Windows ME/2000/XP: Bearbeiten Sie die CLASSPATH-Umgebungsvariable im Dialogfeld der Systemeigenschaften (analog zu der oben beschriebenen Einrichtung der PATH-Variable).
- Linux: Setzen oder passen Sie die CLASSPATH-Variable in *.login*, *.profile*, *.tcshrc*, *.bashrc* o.ä.(analog zu der oben beschriebenen Einrichtung der PATH-Variable). Für csh beispielsweise:

  ```
  setenv CLASSPATH  /bin/interbase;.
  ```

Wo Sie weitere Hilfe finden

Weitere Informationen finden Sie auf der Support-Site zu diesem Buch, www.carpelibrum.de, und auf den Download-Webseiten des Sun-Servers.

Leider mussten wir in der Vergangenheit auch feststellen, dass die großen Browser- und Betriebssystemhersteller ihre Produktzyklen nicht nach den Überarbeitungszyklen unserer Bücher ausrichten. Sollten Sie Probleme mit neuen Betriebssystemen oder Browserversionen haben, schauen Sie bitte auf unserer Website nach.

Sollten die Hinweise im Buch und auf der Website nicht ausreichen, haben Sie keine Scheu sich per E-Mail an uns zu wenden (autoren@carpelibrum.de).

3.3 Welche Art Programm darf es sein?

In Java können Sie die verschiedensten Arten von Programmen schreiben. Die drei wichtigsten Grundtypen sind:

- Konsolenanwendungen
- GUI-Anwendungen und
- Applets.

Zum Einstieg in die Java-Programmierung konzentrieren wir uns ganz auf die Konsolenanwendungen. Der GUI-Programmierung ist Teil IV dieses Buches gewidmet, Applets werden Sie in Teil VI kennen lernen.

3.4 Konsolenanwendungen

Konsolenanwendungen sind – unserer Meinung nach zu Unrecht – die Stiefkinder der modernen Software-Entwicklung. Ohne Fenster und ohne grafische Benutzeroberfläche, fristen sie in der bunten Welt der Window-Manager ein kümmerliches Nischendasein. Viele PC-Benutzer, insbesondere aus der Microsoft-Windows-Gemeinde, wissen nicht einmal mehr, was eine Konsole ist und wie man mit ihr arbeitet. Die Schuld daran trägt nicht allein der Zeitgeist, sondern auch die Microsoft-Betriebssystemdesigner, die – nachdem sie zu einer Zeit, als Mac- und UNIX-Benutzer schon längstens von leistungsfähigen Window-Managern profitierten, ihre Anhänger mit mittelmäßigen Surrogaten wie Windows 3.1 oder Windows 95 quälten – nunmehr dazu übergegangen sind, die Erinnerung an MS-DOS zu löschen und die Konsole (unter Windows meist MS-DOS-Eingabeaufforderung oder nur Eingabeaufforderung tituliert) immer geschickter vor den Augen der Anwender verbergen.

Dabei haben Konsolenanwendungen durchaus ihre Vorzüge: die Programme sind klein und schlank, schnell in der Ausführung und – sofern man einigermaßen mit der Arbeit auf der Konsole vertraut ist – gut zu bedienen. Sie eignen sich vorzüglich als Lehr- und Übungsbeispiele zum Einstieg in die Programmierung (weswegen es sich bei den Beispielprogrammen aus den ersten Teilen dieses Buches fast ausnahmslos um Konsolenanwendungen handelt), werden aber auch weiterhin in der professionellen Software-Entwicklung eingesetzt – beispielsweise für Webserver-Software (gelegentlich ergänzt um fensterbasierte Konfigurationsprogramme), Tools für Systemadministratoren oder eben für Entwickler (siehe die Java-SDK-Tools).

Eine kurze Einführung in die Bedienung der Konsole finden Sie unter »Bildschirmausgaben und Konsolenanwendungen« auf Seite 47. Ausführlichere Informationen finden Sie auf unserer Website www.carpelibrum.de.

Ein einfaches Konsolen-Grundgerüst

Viele Programmierlehrbücher beginnen mit einem kleinen Programm, das die Meldung »Hello World« auf den Bildschirm ausgibt. Wir wollen uns dieser Tradition anschließen und eine Konsolenanwendung erzeugen, die sich mit einem freudigen »Hallo Welt« meldet.

Listing 3.1: HalloWelt.java – Ein einfaches Konsolen-Grundgerüst

```
// Das erste Konsolenprogramm
public class HalloWelt {

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

Das Programm besteht aus einer einzigen Klasse namens Programm, die wiederum über eine einzige Methode, `main()`, verfügt.

`main()` ist die Eintrittsmethode, mit der die Programmausführung beginnt. Jedes Programm muss genau eine `main()`-Methode enthalten und diese muss als `static` deklariert sein, damit sie von der Java Virtual Machine ohne Instanziierung ihrer Klasse ausgeführt werden kann.

`System` ist eine Klasse aus der Java-Standardbibliothek. Sie enthält ein statisches `PrintStream`-Objekt namens `out`, über dessen `println()`-Methode Strings auf den Konsolenbildschirm ausgegeben werden können.

Die `main()`-Methode muss immer mit dem Schlüsselwort `public` deklariert werden, damit sie von der Java Virtual Machine aufgerufen und ausgeführt werden kann. Die zugehörige Klasse muss nicht, wird aber meist ebenfalls als `public` deklariert. (Einfach weil es in Java sinnvoll ist, Klassen mit `public`-Elementen als `public` zu deklarieren.) Wenn Sie eine Klasse als `public` deklarieren, müssen Sie sie in einer Datei speichern, die den gleichen Namen hat wie die Klasse! (Woraus folgt, dass in einer Datei keine zwei `public`-Klassen definiert werden können.)

Kommentare

Die erste Zeile in obigem Quelltext ist ein Kommentar, der der Erläuterung des Quelltextes dient und vom Compiler ignoriert wird. Java kennt drei Arten von Kommentaren:

- `//`-Kommentare reichen bis zum Ende der aktuellen Zeile
- Mehrzeilige Kommentare werden mit `/*` eingeleitet und enden mit `*/`.
- Kommentare, die zwischen den Zeichenfolgen `/**` und `*/` stehen, können mit Hilfe des Java-SDK-Tools *javadoc* zu einer HTML-Dokumentation im Stil der Java-API-Dokumentation aufbereitet werden (siehe Anhang zu Java-SDK-Tools).

Konsolenanwendungen erstellen und ausführen

Um Konsolenanwendungen auf Ihrem Computer zu erstellen und auszuführen, gehen Sie folgendermaßen vor:

1. Öffnen Sie einen Texteditor.

   Rufen Sie einen beliebigen Texteditor auf, mit dem Sie Ihren Quellcode als ASCII-Text abspeichern können.

   Unter Windows eignet sich beispielsweise der Notepad-Editor, den Sie über das Start-Menü mit START/PROGRAMME/ZUBEHÖR/EDITOR aufrufen können (alternativ können Sie START/AUSFÜHREN aufrufen, in dem erscheinenden Dialogfenster *notepad* eingeben und abschicken). Unter Linux können Sie den vi oder KEdit verwenden. (Integrierte Entwicklungsumgebungen wie NetBeans, der JBuilder von Borland oder Eclipse von IBM verfügen hierfür über einen eingebauten Editor.)

2. Geben Sie den Java-Quelltext ein.

   Legen Sie in Ihrem Editor eine neue Datei an, tippen Sie obigen Quelltext ein und speichern Sie die Datei unter dem Namen *HalloWelt.java*. Wichtig ist dabei, dass die Quelltextdatei exakt den gleichen Namen trägt wie die in dem Quelltext definierte Hauptklasse (hier also `class HalloWelt`), wobei auch die Groß- und Kleinschreibung zu beachten ist. Weiterhin wichtig ist, dass die Datei die Dateiendung *.java* trägt!

Bei Verwendung von Notepad gibt es manchmal Probleme, weil der Notepad-Editor die Dateiendung *.txt* anhängt (aus *HalloWelt.java* wird dann *HalloWelt.java.txt*). Um dies zu vermeiden, gibt es zwei Möglichkeiten. Die erste Lösung besteht darin, den kompletten Dateinamen, samt Extension, in Anführungszeichen zu setzen: »*HalloWelt.java*«. Die zweite Möglichkeit, ist die Extension *.java* im Windows Explorer zu registrieren. Speichern Sie dazu nach Methode 1 eine Datei mit der Extension *.java*. Wechseln Sie danach in den Windows Explorer und doppelklicken Sie auf die Datei. Ist die Extension noch nicht registriert, erscheint jetzt der »Öffnen mit«-Dialog. Wählen Sie als gewünschtes Bearbeitungsprogramm *Notepad* aus und aktivieren Sie die Option DIESE DATEI IMMER MIT DIESEM PROGRAMM ÖFFNEN. Wenn Sie den Dialog jetzt abschicken, wird die Extension *.java* registriert und mit Notepad als Standardverarbeitungsprogramm verknüpft. Danach können Sie *.java*-Dateien per Doppelklick in Notepad laden und werden nie wieder Ärger mit an Java-Dateien angehängte *.txt*-Extensionen haben.

(NetBeans JBuilder, Eclipse und andere integrierte Entwicklungsumgebungen arbeiten meist mit Projekten. Lesen Sie im Anhang oder in der Dokumentation zu Ihrer Entwicklungsumgebung nach, wie Sie neue Projekte anlegen und Quelltexte eingeben.)

3. Kompilieren Sie den Quelltext.

   Falls Sie mit dem Java-SDK und einem einfachen Texteditor arbeiten (und nicht mit einer Entwicklungsumgebung), müssen Sie ein Konsolenfenster öffnen, in diesem zum Verzeichnis Ihrer Java-Quelldatei wechseln und dort den Java-Compiler *javac* aufrufen:

   ```
   Prompt:> javac HalloWelt.java <RETURN>
   ```

   Im Gegensatz zu Linux-Anwender sind viele Windows-Anwender heutzutage gar nicht mehr mit dem Umgang mit der Konsole vertraut. Unter Windows heißt die Konsole »MSDOS-Eingabeaufforderung« und wird über das Start-Menü aufgerufen (START/PROGRAMME/MSDOS-EINGABEAUFFORDERUNG). In der Konsole müssen Sie nun mit Hilfe des `cd`-Befehls in das Verzeichnis wechseln, in dem Sie die Datei *HalloWelt.java* abgespeichert haben. Nehmen wir an, es war das Verzeichnis *c:\Java\Kap03*. Dann tippen Sie hinter dem Prompt der Konsole den Befehl *cd c:\Java\Kap03* ein und schicken ihn durch Drücken der ↵-Taste ab.

   Nun kommen wir zum eigentlichen Kompilieren, bei dem der Java-Quelltext (die Datei mit der Extension *.java*) an den Java-Compiler (*javac*) übergeben wird. Tippen Sie zum Kompilieren folgenden Befehl ein:

   ```
   javac HalloWelt.java
   ```

   Schicken Sie den Befehl wiederum durch Drücken der ↵-Taste ab. Dieser Aufruf erzeugt eine ausführbare Bytecode-Datei mit der Extension *.class* – in unserem Beispiel also *HalloWelt.class*.

   Sollten Sie beim Abschicken des Befehls eine Meldung in der Form »Befehl oder Dateiname nicht gefunden« erhalten, ist ihr System nicht so eingerichtet, dass sie die Java-Entwicklungsprogramme aus jedem beliebigen Verzeichnis aufrufen können. Lesen Sie dann bitte noch einmal Abschnitt »Anpassen des Systems« auf Seite 81.

(In NetBeans, JBuilder, Eclipse und anderen integrierten Entwicklungsumgebungen rufen Sie den entsprechenden Menübefehl zur Kompilation Ihres Projektes auf.)

4. Lassen Sie die fertige Anwendung ausführen.

   Dazu rufen Sie den Java-Interpreter (*java*) auf und übergeben diesem als Parameter den Namen Ihrer kompilierten Bytecode-Datei, aber ohne die Endung *.class*, d.h. in der MS-DOS-Eingabeaufforderung wird eingegeben:

   ```
   Prompt:> java HalloWelt
   ```

   Sollten Sie daraufhin eine Fehlermeldung der Form »Exception in thread »main« java.lang.NoClassDefFoundError: HalloWelt« erhalten, bedeutet dies, dass der Interpreter die gewünschte Java-Klasse nicht findet. Dies kann daran liegen, dass die *.class*-Datei nicht erzeugt wurde (kontrollieren Sie nach dem Kompilieren mit Hilfe des DOS-Befehls *dir*, ob die Datei *HalloWelt.class* im Verzeichnis angelegt wurde). Möglich ist auch, dass Sie den Klassennamen nicht exakt so eingegeben haben, wie er im Quelltext definiert ist (auf gleiche Groß- und Kleinschreibung achten). Meist liegt es aber daran, dass irgendeines der auf Ihrem System installierten Programme die Java-Umgebungsvariable `CLASSPATH` so gesetzt hat, dass die class-Dateien im aktuellen Verzeichnis nicht mehr gefunden werden. Dann müssen Sie die `CLASSPATH`-Variable bearbeiten und um den Platzhalter für das aktuelle Verzeichnis (;.) erweitern. Wie dies geht, ist in Kapitel »Anpassen des Systems« beschrieben.

   (In JBuilder, Eclipse und anderen integrierten Entwicklungsumgebungen rufen Sie den entsprechenden Befehl zur Ausführung Ihres Projektes auf.)

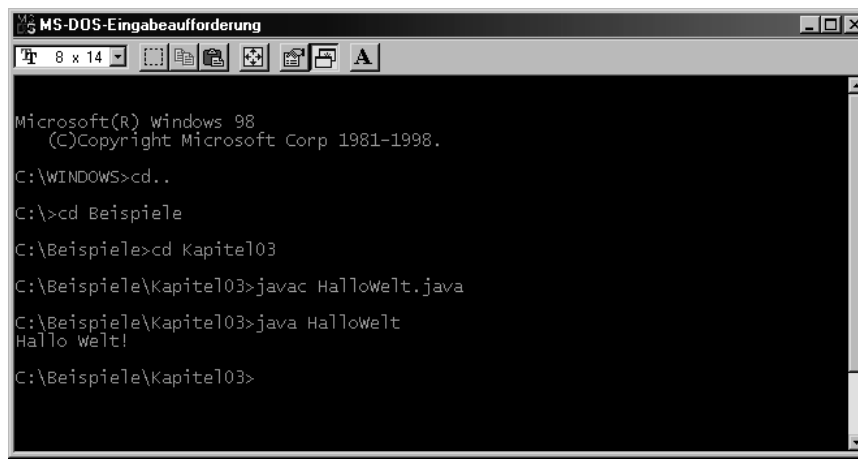


Abbildung 3.5: Kompilation und Ausführung der Anwendung HalloWelt im Verzeichnis c:\Beispiele\Kapitel03

Ein- und Ausgabe für Konsolenanwendungen

Ausgabe

Für die Ausgabe auf die Konsole steht das statische Objekt `out` der Bibliotheksklasse `System` zur Verfügung. Das Objekt präsentiert den Konsolenbildschirm.

Um Strings (Text) auszugeben, rufen Sie einfach die Methode `println()` des `System.out`-Objekts auf:

```
System.out.println("Hallo Welt!");
```

Mit der gleichen Methode können Sie auch Werte elementarer numerischer Datentypen ausgeben. Die Werte werden automatisch in ihre String-Darstellung umgewandelt:

```
double zahl = 0.234;
System.out.println(zahl);  // gibt den Text "0.234" auf die Konsole aus
```

Wenn Sie die Ausgabe nicht mit einem Zeilenumbruch beenden wollen, verwenden Sie statt `println()` die Methode `print()`:

```
System.out.print(zahl);
```

Eingabe

Während die Ausgabe auf die Konsole mit `System.out.println()` noch recht einfach ist, gestaltet sich das Einlesen von Werten über die Tastatur schon etwas schwieriger. Zwar gibt es als Pendant zu `System.out` auch ein Objekt `System.in`, das die Tastatur repräsentiert, doch verfügt dieses nur über Methoden, die Daten als unformatierte Byteströme einlesen. Um einzelne Werte (Strings, Integer- oder Gleitkommazahlen) formatiert einzulesen, muss man einen anderen Weg gehen.

Listing 3.2: Eingabe.java – Ein- und Ausgabe in Konsolenanwendungen

```
01 import java.io.*;
02
03 public class Eingabe {
04
05    public static void main(String[] args)
                                    throws IOException {
06       int zahl;
07
08       System.out.println("Geben Sie einen Integer-Wert ein: ");
09       BufferedReader tastatur =
                 new BufferedReader(new InputStreamReader(System.in));
10       String eingabe = tastatur.readLine();
11
12       zahl = Integer.parseInt(eingabe);
13       System.out.println(zahl);
14    }
15 }
```

Um die komplette Eingabe (bis zum <Return>) auf einmal einzulesen, erzeugt das obige Programm in Zeile 9 zuerst für das Standardeingabegerät (`System.in`) ein `InputStreamReader`-Objekt:

```
new InputStreamReader(System.in)
```

und auf der Basis dieses Objekts ein `BufferedReader`-Objekt:

```
BufferedReader tastatur =   new BufferedReader(...);
```

Dieses Objekt besitzt endlich eine Methode, mit der die Eingabezeile eingelesen werden kann: `readLine()`.

In Zeile 10 wird die Eingabe eingelesen und in dem String `eingabe` abgelegt.

Jetzt muss der String nur noch in einen `int`-Wert umgewandelt werden. Dies leistet die Methode `parseInt()`der Klasse `Integer` (Zeile 12):

```
zahl = Integer.parseInt(eingabe);
```

Tabelle 3.1: Ausgesuchte Umwandlungsmethoden

Umwandlung in	Methode
`int`	`Integer.parseInt(str);`
`long`	`Long.parseLong(str);`
`float`	`Float.parseFloat(str);`
`double`	`Double.parseDouble(str);`

Die Klassen `InputStreamReader` und `BufferedReader` sind in dem Paket `java.io` der Standardbibliothek definiert. Um den Paketnamen nicht dem Klassennamen voranstellen zu müssen, wurden die Klassennamen des Pakets mit der Anweisung `import java.io.*;` aus Zeile 1 importiert.

Die zum Einlesen und Umwandeln benutzten Methoden lösen Exceptions aus, wenn der Einlesevorgang unerwartet abgebrochen oder die gewünschte Umwandlung nicht durchführbar ist. Das Programm muss diese Exceptions abfangen und verarbeiten, sonst lässt es sich nicht kompilieren. Um uns vor der ordnungsgemäßen Fehlerbehandlung zu drücken – wir werden uns erst in Kapitel 16 intensiver mit der Fehlerbehandlung durch Exceptions beschäftigen –, deklarieren wir die Methode `main()` einfach so, dass sie IO-Exceptions weiterleitet (Zeile 5).

Beachten Sie, dass das Programm nur unzureichend gegen inkorrekte Benutzereingaben abgesichert ist. Gibt der Benutzer falsch formatierte Werte ein, die sich nicht in einen `int`-Wert umwandeln lassen (»3.24«, »drei«), oder schickt er mehrere Werte gleichzeitig an das Programm (»3 4 122«), löst das Programm eine Exception aus und wird beendet.

Klassen, Pakete und die Standardbibliothek

Ebenso wie C oder C++ gibt es auch in Java keine in die Sprache integrierten Befehle für typische Programmieraufgaben, wie z.B. die Berechnung eines Sinus, das Vergleichen von Strings oder auch nur das Einlesen von Daten über die Tastatur bzw. die Ausgabe von Strings auf die Konsole.

Zum Ausgleich wird Java mit einer umfangreichen Standardbibliothek ausgeliefert, die für nahezu jede grundlegende Programmieraufgabe (Abfragen der Systemzeit, Stringmanipulationen, Sinusberechnung, Ein- und Ausgabe und, und, und) eine Lösung bereit hält – allerdings stets in Form einer Klasse.

Manche dieser Klassen deklarieren statische Elemente, auf die Sie direkt über den Klassennamen zugreifen können – so z.B. die Klasse `Math`:

```
double hoehe = entfernung * Math.tan(Math.toRadians(winkel));
```

Andere Klassen instanziieren Sie, um Objekte der Klasse zu erzeugen – beispielsweise die Klasse `String`:

```
String gruss = new String("Hallo ");
```

Danach können Sie die Objekte mit Hilfe der `public`-Elemente der Klassen bearbeiten:

```
String name = new String("Fred");
gruss.concat(name);  // hängt name an gruss an
```

Nicht selten enthalten die Klassen überladene Methoden (und Konstruktoren), die eine Operation je nach übergebenen Argumenten unterschiedlich ausführen. Wenn Sie beispielsweise die statische Methode `println()` mit einem String-Argument aufrufen, gibt die Methode den String auf die Konsole aus und bricht danach die Zeile um. Wenn Sie die Methode ohne Argument aufrufen, gibt die überladene Methode nur einen Zeilenumbruch aus:

```
System.out.println("Text ausgeben");    // Text und Zeilenumbruch
System.out.println();                  // Zeilenumbruch
```

Pakete

Die Java-Standardbibliothek ist in so genannte **Pakete** organisiert. Diese ordnen die Klassen nicht nur nach ihrer Funktionalität, sondern helfen auch, Namenskonflikte (in einem Programm darf es keine zwei Klassen mit gleich lautenden Namen geben!) zu vermeiden.

Grundsätzlich gilt: Wenn Sie auf eine Klasse zugreifen wollen, müssen Sie den Namen des Pakets, in dem die Klasse definiert ist, voranstellen. Ist das Paket selbst Teil eines übergeordneten Pakets, müssen Sie den Pfad vom obersten Paket bis zum Paket der Klasse angeben.

Die Klasse `BufferedReader` ist beispielsweise im Paket `io` definiert, das selbst wieder Teil des Pakets `java` ist. Der korrekte Zugriff auf den Konstruktor der Klasse lautet daher:

```
java.io.BufferedReader( );
```

Wenn Ihnen die Voranstellung des Paketpfads zu lästig ist, können Sie die Klassennamen eines Pakets zu Anfang des Quelltextes einmalig importieren. Danach können Sie die Klassennamen direkt verwenden:

```
import java.io.BufferedReader;    // importiert den Klassennamen
                                  // BufferedReader
import java.io.*;                 // importiert alle KLassennamen aus
                                  // java.io, die im Quelltext ver-
                                         // wendet werden.
```

Lediglich die Namen der Klassen im Paket `java.lang` – hierzu gehören unter anderem `System`, `Integer` und `Math` – müssen nicht explizit angegeben oder importiert werden.

Beachten Sie, dass in Java die allgemeine akzeptierte Konvention gilt, dass Paketnamen (meist) mit Kleinbuchstaben geschrieben werden, während Klassennamen mit einem Großbuchstaben beginnen. Felder und Methoden beginnen mit Kleinbuchstaben. In Namen, die aus mehreren Wörtern zusammengesetzt sind, beginnen die nachfolgenden Wörter mit Großbuchstaben.

Mehr zum Konzept der Pakete in Kapitel 10.1.