dirk LOUIS
peter MÜLLER

2. Auflage

Java

EINE EINFÜHRUNG IN DIE PROGRAMMIERUNG

HANSER

Im Internet: Alle Beispiele und Setup-Dateien für Java 9

Louis/Müller

Java

Dirk Louis
Peter Müller

Java

Eine Einführung in die Programmierung

2. Auflage

HANSER

Die Autoren:

Dirk Louis, Saarbrücken, autoren@carpelibrum.de

Peter Müller, Saarbrücken, leserfragen@gmx.de

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Petra Kienle, Fürstenfeldbruck
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Stephan Rönigk
Layout: Kösel Media GmbH, Krugzell
Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen
Printed in Germany

Print-ISBN: 978-3-446-45194-0

E-Book-ISBN: 978-3-446-45362-3

Inhalt

Vorwort

Dieses Buch soll Sie auf leicht verständliche und gleichsam unterhaltsame Weise in die Programmierung mit Java einführen. Vorhandene Programmierkenntnisse können von Vorteil sein, werden aber nicht vorausgesetzt. Schritt für Schritt werden Sie sich in Java einarbeiten und erfahren, wie Sie die Mächtigkeit der Sprache für Ihre Zwecke nutzen können.

Sie werden Ihre ersten Java-Anwendungen schreiben, sich mit der objektorientierten Programmierung vertraut machen und erfahren, wie man in Java Anwendungen mit grafischen Benutzeroberflächen programmiert. In den letzten Kapiteln des Buchs wenden wir uns dann noch diversen Fortgeschrittenenthemen zu, wie der Implementierung von Threads und Animationen oder der Datenbankunterstützung.

Am Ende eines jeden Kapitels finden Sie eine Reihe von Testfragen und eine Zusammenfassung des behandelten Stoffs sowie einige Übungsaufgaben, die zur Vertiefung und Wiederholung des Stoffs dienen, vor allem aber auch die Lust am Programmieren anregen sollen.

Programmierkenntnisse sind wie erwähnt nicht erforderlich, aber Sie sollten auch nicht zu den Zeitgenossen gehören, die bisher noch jedem Computer erfolgreich aus dem Weg gegangen sind. Spaß am Programmieren können wir Ihnen nur dann vermitteln, wenn Sie selbst ein wenig guten Willen und Ausdauer mitbringen. Gute Laune, eine Portion Neugier, dieses Buch – was brauchen Sie noch?

Um mit Java programmieren zu können, benötigen Sie ein entsprechendes Entwicklungspaket, das sogenannte JDK (Java Development Kit). Optional können Sie zusätzlich zu dem JDK eine integrierte Entwicklungsumgebung wie z. B. NetBeans oder die Open-Source-Software Eclipse verwenden.

Für Java-Einsteiger ist der Einsatz einer integrierten Entwicklungsumgebung allerdings häufig recht verwirrend und lenkt vom eigentlichen Primärziel, die Programmiersprache Java zu lernen, unnötig ab. Schlimmer noch: Die meisten Entwicklungsumgebungen sind für fortgeschrittene Programmierer konzipiert und erleichtern deren tägliche Programmierarbeit, indem sie komplexe Arbeitsschritte automatisieren, vordefinierte Codegerüste anbieten, eigenständig Code erzeugen. Für Anfänger ist dies ein Desaster! Nehmen wir nur einmal das Codegerüst einer einfachen Konsolenanwendung, das in Java aus ungefähr vier bis sieben Zeilen Code besteht. Wenn Ihnen dieses Codegerüst stets von Ihrer Entwicklungsumgebung fertig vorgelegt wird, werden Sie sich kaum die Mühe machen, es je selbst einmal abzutippen. Sie werden es sich vielleicht anschauen und versuchen, es zu verstehen, aber Sie werden es nie wirklich verinnerlichen. Wenn Sie dann später einmal an einem Rechner

arbeiten müssen, auf dem keine Entwicklungsumgebung installiert ist, werden Sie mit Schrecken feststellen, dass es Ihnen unmöglich ist, das Grundgerüst aus dem Kopf nachzustellen. Nun, ganz so schlimm wird es vielleicht nicht kommen, aber der Punkt ist, dass die Annehmlichkeiten der Entwicklungsumgebungen den Anfänger schnell dazu verführen, sich mit zentralen Techniken und Prinzipien der Java-Programmierung nur oberflächlich auseinanderzusetzen.

Aus diesem Grund legen wir in diesem Buch Wert auf Handarbeit mit elementarsten Mitteln: Wir setzen unsere Quelltexte in einem einfachen Texteditor auf, wandeln die Quelltexte mithilfe des Java-Compilers *javac* aus dem JDK in ausführbare Programme um und führen diese dann mit dem Java-Interpreter *java* (ebenfalls im JDK enthalten) aus. Wie Sie dabei im Einzelnen vorgehen und was Sie beachten müssen, erfahren Sie in den einleitenden Kapiteln und im Anhang dieses Buchs.

Benötigte Software und Beispielsammlung

Die Java-Entwicklungsumgebung und die Beispielsammlung zu diesem Buch finden Sie zum Download im Internet. Die entsprechenden Links finden Sie im Anhang F, Hinweise zur Installation der Entwicklungsumgebung in Anhang B.

www.carpelibrum.de

Falls Sie während der Buchlektüre auf Probleme oder gar auf inhaltliche Fehler stoßen, sollten Sie nicht zögern, uns eine E-Mail unter Angabe von Buchtitel und Auflage zu senden. Allerdings schauen Sie bitte zuerst auf unserer Buchseite www.carpelibrum.de nach, ob sich nicht dort schon eine Antwort findet. Neben Aktualisierungen, Fehlerkorrekturen und Antworten auf typische Fragen finden Sie dort auch Hinweise auf weitere Bücher rund ums Thema Programmieren.

Viel Erfolg mit Java wünschen Ihnen

Dirk Louis (autoren@carpelibrum.de)

Peter Müller (leserfragen@gmx.de)

Saarbrücken, im Frühjahr 2018

1 Bevor es losgeht

Ich weiß, ich weiß – Sie sitzen vor Ihrem Bildschirm, haben bereits Ihre Java-Entwicklungsumgebung installiert, brennen darauf, Ihr erstes Java-Programm zu schreiben, und sind einigermaßen ungehalten, sich erst noch durch etliche Seiten theoretischer Ausführungen quälen zu müssen. Müssen Sie nicht! Wenn Sie Ihre Entwicklungsumgebung schon eingerichtet haben und mit der Programmerstellung prinzipiell vertraut sind, überspringen Sie dieses Kapitel einfach. Nur wenn Sie ein absoluter Neuling in der Programmierung sind oder von Java nicht viel mehr wissen, als dass es eine Programmiersprache ist, sollten Sie dieses Kapitel unbedingt vorab durchlesen. Alle anderen können nach Bedarf auch noch später hierher zurückkehren.

1.1 Was ist Java? –Teil I

Java ist heute eine der führenden Programmiersprachen, vielleicht die wichtigste Programmiersprache überhaupt. Programmierer weltweit schätzen Java für seine Robustheit, seine Vielseitigkeit, die problemlose Portierbarkeit seiner Anwendungen und, und, und.

Portierung bedeutet, dass ein Programm von einem Rechner auf einen anderen Rechner verschoben und ausgeführt wird.

Entstanden ist Java 1993 als Forschungsprojekt der Firma Sun, wobei schon auf diverse Vorarbeiten zurückgegriffen werden konnte. Der konkrete Anlass war der einsetzende Boom des World Wide Web, das nach einer geeigneten Programmiersprache verlangte. Java ist vor diesem Hintergrund zu betrachten und quasi ideal für den Einsatz im Internet – sei es, dass man seine Programme über das Internet vertreiben möchte, sei es, dass man Programme für Webseiten schreiben möchte, oder sei es, dass man über das Internet verteilte Anwendungen implementieren möchte.

Wie steht es in diesem Zusammenhang mit der Verwandtschaft von Java zu den anderen Programmiersprachen? Man entwickelt schließlich keine neue Programmiersprache, ohne die eigenen Erfahrungen mit den etablierten Programmiersprachen einfließen zu lassen.

Nun, C++-Programmierer wird es freuen zu hören, dass Java stark an C++ angelehnt ist. Die Gründe hierfür sind zweifellos in der Objektorientierheit, der Schnelligkeit und der Leistungsfähigkeit von C++ zu suchen, aber natürlich auch in der traditionellen Bedeutung dieser Sprache.

Allerdings hat Java viel unnötigen Ballast, den C++ mit sich schleppt, abgeworfen und ist dadurch wesentlich einfacher zu erlernen und zu programmieren. Diese Entschlackung dient nicht nur der Entlastung des Programmierers, sondern soll vor allem auch die Entwicklung „sicherer“ Programme gewährleisten. Natürlich liegt die Verantwortung für die Sicherheit der Anwendungen deswegen letztendlich immer noch beim Programmierer. Je komplizierter und undurchsichtiger die Konzepte einer Sprache aber sind, umso wahrscheinlicher ist es, dass der Programmierer unbeabsichtigt Fehler einbaut. In Java hat man dies erkannt und beispielsweise die gesamte Zeigerprogrammierung und die dynamische Speicherverwaltung aus den Händen des Programmierers genommen und Compiler und Interpreter übertragen.

Und wie steht es mit C#? Die Sprache C# ist im Grunde nichts anderes als eine Trotzreaktion auf Java. Microsoft wollte nämlich ursprünglich eine eigene Java-Variante etablieren, was dem Konzern aber gerichtlich verboten wurde. Danach konzipierte man eine ganze neue Sprache, eben C#, die aber eine verblüffende Ähnlichkeit mit Java aufweist. Umsteigern von C# wird also vieles bekannt vorkommen.

Falls Sie schon C++ beherrschen – die folgenden Konzepte gibt es in Java *nicht*:

1. Zeiger (die dynamische Speicherverwaltung wird intern vorgenommen)
2. Funktionen (statt alleinstehender Funktionen gibt es nur noch Methoden (Elementfunktionen) von Klassen)
3. Strukturen und Unions
4. Arrays und Zeichenfolgen gibt es nur als Objekte
5. Typendefinition (typedef)
6. Mehrfachvererbung (nur in gemäßigter Form)
7. Überladung von Operatoren

Java deshalb als Schmalspur-C++ zu bezeichnen, wäre aber völlig falsch. Von der Leistungsfähigkeit her steht Java C++ kaum in etwas nach. Betrachtet man obige Liste etwas genauer, lässt sich feststellen, dass viele Konzepte, die C++ von C übernommen hat, zugunsten einer konsequenteren objektorientierten Programmierung aufgegeben wurden (dies betrifft die Sprachelemente 2 bis 5, die alle im Klassenkonzept aufgegangen sind). Java ist daher mittlerweile die Standardprogrammiersprache an allen US-amerikanischen Universitäten und auch an deutschen Universitäten allgegenwärtig.

Andererseits wurde auf bestimmte objektorientierte Konzepte (Punkte 6 und 7), die im Wesentlichen der Wiederverwertung objektorientierten Quellcodes dienen, aber für Einsteiger (und auch oft noch für Fortgeschrittene) manchmal schwierig zu handhaben sind, verzichtet. Was geblieben ist, ist eine relativ leicht zu erlernende, konsequent objektorientierte Sprache, die Ihnen einiges zu bieten hat:

- Objektorientiertheit,
- statische Typbindung, aber späte Methodenbindung,
- dynamische Speicherverwaltung und Garbage Collection,
- Multithreading,
- Exception-Behandlung.

Achtung!

Verwechseln Sie Java nicht mit JavaScript. JavaScript wurde von Netscape entwickelt und als Erweiterung des HTML-Standards implementiert. Die JavaScript-Syntax ist an Java angelehnt, doch damit hört die Verwandtschaft zu Java auch schon auf. Mit Java können Sie echte eigenständige Programme schreiben (das Thema dieses Buchs), Sie können Java-Servlets schreiben, die dynamische Webseiten erzeugen, und Sie können spezielle Java-Module erzeugen, sogenannte Applets, die in Webseiten eingebettet und von Browsern mit installiertem Java-Plug-in ausgeführt werden können (früher eine Sensation, heute kaum noch von Bedeutung). Dagegen dient JavaScript-Code allein der Dynamisierung von Webseiten, weswegen der Code auch direkt in den HTML-Code eingefügt und vom Browser interpretiert wird (sofern die JavaScript-Unterstützung nicht ausgeschaltet wurde).

1.2 Was ist ein Programm?

Prinzipiell sind Programme nichts anderes als eine Folge von Befehlen, die an einen Computer gerichtet sind und von diesem befolgt werden. Im Grunde genommen funktionieren Programme also genauso wie Kochrezepte: Sie als Programmierer sind der Koch, der das Buch schreibt. Jedes Kochrezept entspricht einem Programm und der Computer, der Ihre Programme ausführt, ist der Leser.

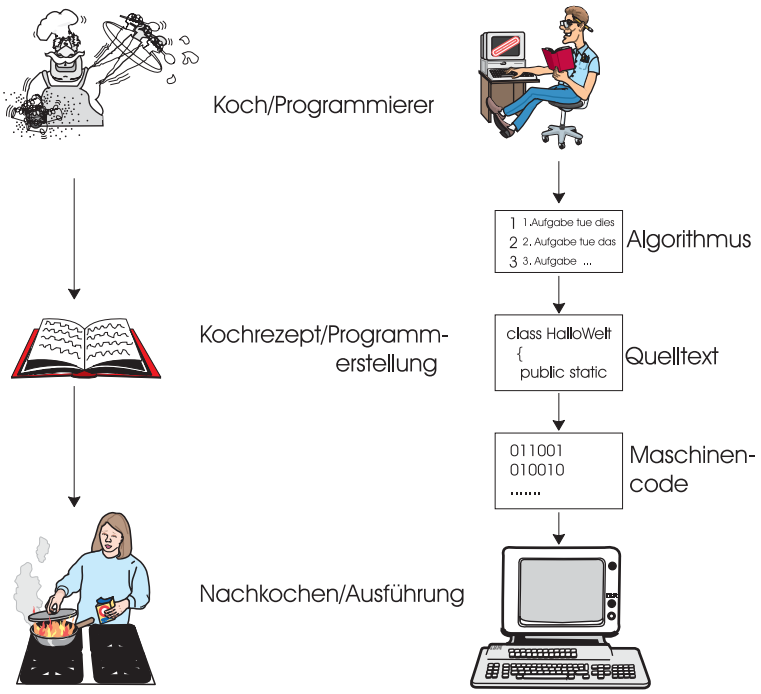

Bild 1.1 Analogie zwischen Programmen und Kochrezepten

Leider ist die Realität wie üblich etwas komplizierter als das Modell. Im Falle des Kochrezepts können wir einfach davon ausgehen, dass Schreiber und Leser die gleiche Sprache sprechen. Im Falle des Programmierers und des Computers ist dies natürlich nicht der Fall, denn Sie als Programmierer sprechen an sich Deutsch und der Computer spricht ... ja, welche Sprache versteht eigentlich ein Computer?

Ich wünschte, Sie hätten diese Frage nicht gestellt, denn die Antwort ist äußerst unerfreulich. Der Computer, in diesem Fall sollte man genauer von dem Prozessor des Computers sprechen, versteht nur einen ganz begrenzten Satz elementarer Befehle – den sogenannten Maschinencode, der zu allem Unglück noch binär codiert ist und daher als eine Folge von Nullen und Einsen vorliegt. Können Sie sich vorstellen, Ihre Programme als Folge von Nullen und Einsen zu schreiben? Wahrscheinlich genauso wenig, wie Ihr Computer in der Lage ist, Deutsch zu lernen. Wir haben also ein echtes Verständigungsproblem. Um dieses zu lösen, müssen Sie – als der Intelligentere – dem Computer schon etwas entgegenkommen.

Kehren wir noch einmal zu unserem Kochbuch zurück und stellen Sie sich vor, ein Chinese würde ein Kochbuch schreiben, das auf dem deutschen Buchmarkt erscheinen soll. Zwar findet der Chinese keinen geeigneten Übersetzer, der das Buch ordentlich vom Chinesischen ins Deutsche übersetzen könnte, aber er erinnert sich seiner Englischkenntnisse, die für ein Kochbuch absolut ausreichend sein sollten. Er schreibt also sein Buch in Englisch und lässt es dann von einem Übersetzer ins Deutsche übertragen. Gleiches geschieht auch bei der Programmierung. Anstatt Ihre Programme in Deutsch aufzusetzen, bedienen Sie sich einer Programmiersprache (wie Java, C, Pascal, Basic etc.), für die es einen passenden

Übersetzer gibt (in diesem Fall auch Compiler genannt), der Ihre Anweisungen in Maschinencode umwandeln kann.

Ich denke, das ist jetzt schon etwas klarer. Was aber genau ist jetzt das Programm? Die noch in Deutsch formulierten Befehle? Die in Java formulierten Befehle? Oder die binär codierten Maschinenanweisungen? Im weitesten Sinne können Sie in allen drei Fällen von Ihrem Programm reden. Wenn Sie es dagegen genau nehmen wollen, bezeichnen Sie die noch in Ihrer Sprache aufgesetzte Befehlsfolge als Algorithmus, die in Java formulierte Version des Algorithmus als Quelltext Ihres Programms und erst den vom Compiler erzeugten Maschinencode als Ihr Programm.

Mehr oder weniger unabsichtlich sind wir damit bereits in die Programmerstellung abgeglitten, die im nächsten Abschnitt noch einmal zusammengefasst wird.

1.3 Wie werden Programme erstellt?

Die Entwicklung von Computerprogrammen läuft unabhängig von der verwendeten Sprache üblicherweise nach dem folgenden Muster ab:

1. Man hat ein Problem, eine Idee, eine Aufgabe, zu deren Lösung man einen Computer einsetzen möchte.
2. Als Nächstes wird die Aufgabe als Algorithmus, also als eine Folge von Befehlen formuliert. Größere Probleme werden dabei in Teilaufgaben und Teilaspekte aufgeteilt. (Ob der Algorithmus tatsächlich auf dem Papier oder nur im Kopf des Programmierers entwickelt wird, hängt von der Komplexität der Aufgabe und der Genialität des Programmierers ab.)
3. Der Algorithmus wird in für den Computer verständliche Anweisungen einer Programmiersprache umgesetzt. Dies ergibt den sogenannten Quelltext oder Quellcode.
4. Dieser Quelltext muss dann durch ein spezielles Programm, den Compiler, in Maschinenanweisungen übersetzt werden, die das eigentliche Herz des Computers – der Prozessor – versteht und ausführen kann.
5. Das ausführbare Programm wird gestartet, das heißt in den Hauptspeicher geladen und vom Prozessor ausgeführt.

1.4 Von Compilern und Interpretern

Bei einigen Programmiersprachen fallen die Schritte 4 und 5 zusammen. Es wird also nicht das ganze Programm erst übersetzt (kompiliert) und dann bei Bedarf ausgeführt. Stattdessen wird bei der Ausführung des Programms der Quelltext Zeile für Zeile eingelesen, übersetzt und ausgeführt. In diesem Fall spricht man von Interpreter-Sprachen, weil das Programm nicht als ausführbare Datei, sondern bloß als Quelltext vorliegt, der nur mithilfe eines speziellen Programms (des Interpreters), welches die zeilenweise Übersetzung wäh-

rend des Programmlaufs übernimmt, ausgeführt werden kann. In dem Beispiel aus Abschnitt 1.2 würde dies bedeuten, dass der chinesische Koch seine Kochkünste nicht in Buchform, sondern als Hörkassette herausgegeben hat und Sie mit einem Dolmetscher (Interpreter) an der Seite diese Kassette abspielen und die Rezepte nachkochen.

Abgesehen davon, dass Sie wahrscheinlich niemals auf die Idee kommen werden, sich eine chinesische Kassette mit Kochrezepten zu kaufen, haften interpretierten Programmen zwei wesentliche Nachteile an:

- Da die Erzeugung des Maschinencodes erst während der Ausführung vorgenommen wird, dürfte klar sein, dass solche Programme wesentlich langsamer ablaufen als kompilierte Programme.
- Da diese Programme als Quelltext vertrieben werden, sind der nicht autorisierten Nutzung des Programmtextes und der für das Programm entwickelten Algorithmen Tür und Tor geöffnet.

Auf der anderen Seite haben diese Programme den Vorteil, dass sie sehr gut zu portieren sind, das heißt, die Übertragung von einem Computer auf einen anderen ist unproblematisch.

Bekannte Vertreter von Compiler-Sprachen sind C, C++ und Pascal. Das klassische Beispiel für eine interpretierte Programmiersprache ist Basic.

Jetzt fragen Sie sich sicherlich, wo Java einzuordnen ist.

1.5 Was ist Java? – Teil II

Ist Java nun eine Compiler- oder eine Interpreter-Sprache? Nun, die Einteilung ist hier nicht ganz so klar. Man kann zwar vielerorts lesen, es sei eine Interpreter-Sprache, aber das wird der Lage nicht ganz gerecht, denn tatsächlich ist Java beides.

Java-Quellcode wird zunächst mit einem Compiler (er heißt *javac*) übersetzt, allerdings nicht in den Maschinencode des jeweiligen Prozessors, sondern in sogenannten Bytecode. Man kann sich diesen Bytecode als einen Maschinencode eines virtuellen Prozessors vorstellen, das heißt eines Prozessors, den es gar nicht gibt!

Damit der Bytecode nun von einem echten Prozessor ausgeführt werden kann, muss er während des Programmlaufs in dessen Maschinencode übersetzt werden, das heißt, ein Interpreter ist zum Ausführen von Java-Programmen notwendig, den man einfach wie die Sprache getauft hat, also *java*.

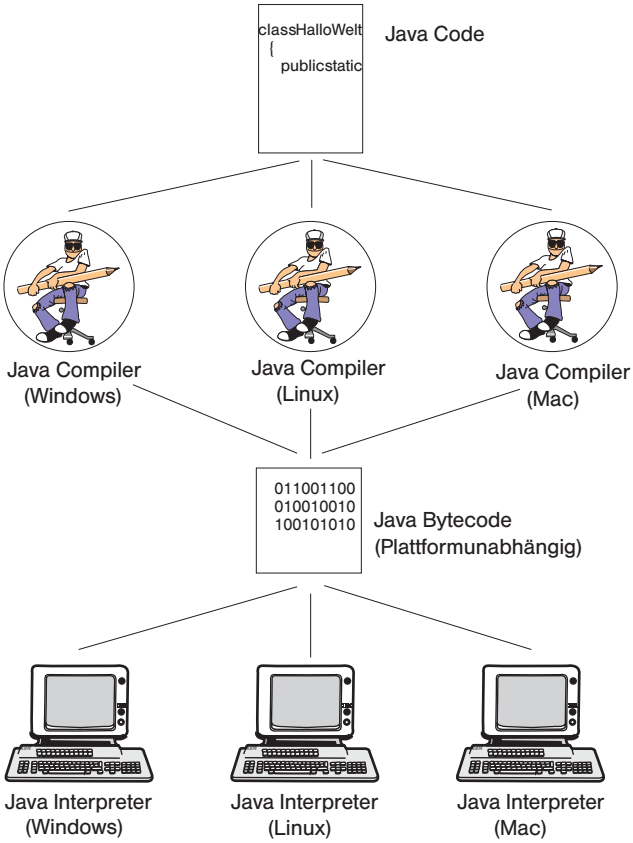

Bild 1.2 Erstellung von Java-Programmen

Warum diese seltsame Mischung aus Compiler- und Interpretersprache? Die Antwort liegt eigentlich auf der Hand. Es geht darum, die Vorteile beider Systeme miteinander zu verbinden:

- Der vorkompilierte Bytecode kann wesentlich schneller interpretiert werden.
- Die Übersetzung in Bytecode schützt Ihre Algorithmen vor unerwünschter Nachahmung.
- Der Bytecode ist plattformunabhängig. Gleichgültig, auf welchem Computer der Bytecode erstellt worden ist, er wird immer identisch aussehen. Es ist dadurch möglich, Bytecode von einem Rechner zum nächsten zu transferieren (zum Beispiel über das Internet) und dort ausführen zu lassen, ohne die *geringste* Änderung vornehmen zu müssen. (Wenn Sie schon in konventionellen Sprachen wie C programmiert haben und das Problem hatten, ein Programm zu entwerfen, das auf verschiedenen Betriebssystemen, Prozessoren und Compilern läuft, werden Sie diesen Vorteil von Java sicherlich zu schätzen wissen.)
- Der Interpreter hat die Möglichkeit, die Programmausführung zu überwachen und beispielsweise (absichtliche oder unabsichtliche) unkorrekte Speicherzugriffe oder Datenmanipulationen zu erkennen und direkt mit einer Fehlermeldung abzufangen. Dies ist insbesondere unter Sicherheitsaspekten interessant. Da Java für das Internet und somit

für den beliebigen Datenaustausch konzipiert ist, muss weitgehend sichergestellt sein, dass ein Java-Programm, das man mit seinem Browser durch einen mehr oder weniger achtlosen Klick startet, nicht irgendwelche üblen Sachen anstellt. Ein Interpreter, der ständig darauf achtet, was das Java-Programm gerade tun will, ist dafür ein geeigneter Ansatz.

Zusammenfassend lässt sich sagen, dass Java-Programme aufgrund des Compiler/Interpreter-Ansatzes hardwareunabhängig und somit portabel sind und gleichzeitig ein relativ hohes Maß an Sicherheit vor Manipulationen bieten. Diese Vorteile gehen jedoch zu Lasten der Geschwindigkeit. Interpretierte Java-Programme sind trotz Bytecode langsamer als vergleichbare C/C++-Programme. Dies sollte Sie aber nicht abschrecken. Moderne Prozessoren sind schnell genug und viele Programme verbringen die meiste Zeit sowieso mit Warten auf Benutzereingaben, sodass der Geschwindigkeitsnachteil gar nicht zum Tragen kommt. Zudem verwenden modernen Java-Interpreter spezielle Techniken, wie z. B. die sogenannte Just-In-Time-Compilation (JIT), die den Geschwindigkeitsnachteil minimal halten.[1]

Die Java-Laufzeitumgebung (JRE)

Sie haben gelernt, dass auf einem Rechner, auf dem Java-Programme ausführt werden sollen, ein Java-Interpreter installiert sein muss (siehe auch Bild 1.2). Dies war insofern etwas vereinfacht dargestellt, als nicht nur ein einzelnes Interpreter-Programm, sondern eine ganze Kombination aus Tools und Bibliotheken benötigt werden – die sogenannte Java-Laufzeitumgebung, abgekürzt **JRE** („Java Runtime Environment“).

Wenn Sie also Ihre Java-Progamme an Freunde weitergeben, sagen Sie ihnen, dass sie sich gegebenenfalls die JRE von der Site *http://www.oracle.com/technetwork/java/javase/downloads/index.html* herunterladen müssen. Ausgeführt werden die Programme dann durch Übergabe an den *java*-Interpreter, so wie Sie es gleich weiter unten lernen werden. Oder Sie erzeugen für Ihre Freunde eine direkt ausführbare *.jar*-, *.bat*- oder *.exe*-Datei, siehe Kapitel 16.

[1] Bei der Just-In-Time-Compilation übersetzt der Java-Interpreter *während der Programmausführung* Programmteile, die oft benötigt werden, in optimierten Maschinencode für den verwendeten Prozessor und hält diese Codeblöcke im Hauptspeicher, sodass bei Wiederverwendung während der aktuellen Programmausführung keine erneute Übersetzung notwendig ist. Zusätzlich werden häufig fortlaufend Hot-Spot-Analysen gemacht, d. h., sehr oft durchlaufene Code-Abschnitte (die „Hot-Spots“) werden parallel zur Programmausführung intensiv optimiert und dadurch noch schneller gemacht.

1.6 Vorbereitung zum Programmieren

Bevor wir im nächsten Kapitel voll in die Programmierung einsteigen, sollten Sie Ihren Computer vorbereitet haben. Folgende Zutaten sind für die Erstellung von Java-Programmen notwendig:

- **die Java-Entwicklungswerkzeuge (JDK)**

  Das JDK steht unter *http://www.oracle.com/technetwork/java/javase/downloads/index.html* zum Download zur Verfügung (siehe Anhang F, Installationshinweise siehe Anhang B).

  Das JDK[2] besteht aus einer Sammlung aller für die Programmierung erforderlichen Tools (Compiler, Interpreter, Debugger etc.) und den zu Java gehörenden Standardbibliotheken. Die JDK-Tools werden von der Konsole aus aufgerufen. Linux-Anwender öffnen hierzu ein Terminalfenster, Windows-Anwender die Eingabeaufforderung.

- **ein Texteditor**

  Für die Programmierung genügt ein einfacher Texteditor, wie er zur Grundausstattung jedes anständigen Betriebssystems gehört. Wichtig ist, dass der Editor unformatierten Text (also einfach nur die Buchstaben ohne Formatangaben wie fett, kursiv etc.) abspeichern kann.

  Mögliche Optionen wären *Editor* (*notepad.exe*) oder *Wordpad* unter Microsoft Windows (Aufruf über die Suche der Windows-Startseite (Windows 8) bzw. des Windows-Startmenüs (Windows 7)) bzw. *vi*, *KWrite* oder *emacs* unter Linux/Unix. Natürlich können Sie auch Textverarbeitungssoftware wie *Microsoft Word* einsetzen, allerdings müssen Sie darauf achten, nur reinen unformatierten Text (ASCII bzw. ANSI, Dateiendung *.txt*) abzuspeichern.

Auch wenn konventionelle Texteditoren wie *Editor* oder *Wordpad* grundsätzlich vollkommen ausreichen, lassen sie doch verschiedene Extras vermissen, die gerade dem Programmierer die Arbeit sehr erleichtern können – wie z. B. die automatische Syntaxhervorhebung oder die Anzeige der Zeilennummern[3]. Wer auf diese Funktionen nicht verzichten möchte, muss zu einem Editor greifen, der sich auf die Quelltextverarbeitung spezialisiert hat. Empfehlenswerte Vertreter dieser Gattung sind z. B. der *emacs* für Linux (gehört in der Regel zum Lieferumfang) und der *Notepad*++-Editor für Windows (kann von *http://notepad-plus.sourceforge.net/de/site.htm* kostenlos heruntergeladen werden).

[2] Abkürzung für „Java SE Development Kit“

[3] Der Compiler gibt Meldungen zu Syntaxfehlern immer unter Angabe der betroffenen Zeilennummer aus. Wenn Sie einen Editor mit Zeilennummerierung verwenden, müssen Sie nicht selbst zählen, um die verdächtige Zeile zu finden.

Einrichtung einer eigenen Entwicklungsumgebung

Als Erstes sollten Sie sich überlegen, wie Sie die Dateien Ihrer Programme auf der Festplatte verwalten wollen. Eine Möglichkeit wäre, unter einem eigenen übergeordneten Verzeichnis für die einzelnen Programme Unterverzeichnisse anzulegen. Zum Nachvollziehen der Beispiele in diesem Buch bietet es sich allerdings an, unter dem übergeordneten Verzeichnis Unterverzeichnisse für die Buchkapitel anzulegen.

Wenn Sie das übergeordnete Verzeichnis *Java* nennen, könnte Ihre Verzeichnisstruktur wie folgt aussehen:

C:\Java

C:\Java\Kap02

C:\Java\Kap03

C:\Java\Kap04

...

In den Verzeichnissen für die einzelnen Kapitel speichern Sie dann die Quelldateien der zugehörigen Beispiele. Für kleinere Beispiele können Sie die Quelldateien ruhig zusammen im Kapitelverzeichnis speichern. Für größere Beispiele, die aus mehreren Quelldateien bestehen, empfiehlt es sich, nochmals eigene Unterverzeichnisse für jedes Programm anzulegen. (Gleiches gilt, wenn zwei Programme gleichnamige Klassen definieren.)

1. Installieren Sie zuerst das *JDK*, siehe Anhang B.
2. Richten Sie danach auf Ihrem Rechner ein Verzeichnis *Java* ein.

   Unter Windows können Sie das Verzeichnis direkt unter *C:* anlegen, also als *C:\Java*. (Unterverzeichnisse können im Windows Explorer, Aufruf mit **WinBef+E**, erzeugt werden.)

   Unter Linux legen Sie das Verzeichnis unter Ihrem Home-Verzeichnis an.
3. Legen Sie ein Unterverzeichnis *Kap02* für Kapitel 2 an.

Jetzt sollten Sie Ihren Desktop noch so gestalten, dass Sie alle für die Programmierung typischen Aufgaben effizient erledigen können.

4. Lassen Sie den Windows Explorer (Ordnerfenster) aus Schritt 1 geöffnet. Sie brauchen ihn, um bei Bedarf weitere Kapitel- oder Programmverzeichnisse anzulegen. Außerdem können Sie aus dem Explorer heraus bestehende Quelldateien zur Ansicht oder Bearbeitung öffnen.
5. Rufen Sie den Texteditor Ihrer Wahl auf, mit dem Sie Ihren Quellcode als reinen Text (ASCII oder ANSI, kein Unicode) abspeichern können. Wenn Sie keinen spezialisierten Quelltexteditor besitzen (vgl. Anmerkung zu *Notepad++* weiter oben), verwenden Sie einfach einen der Editoren aus dem Lieferumfang Ihres Betriebssystems.

   Unter Windows eignet sich beispielsweise wie gesagt der *Editor* (*notepad.exe*), den Sie über die Windows-Suche, Suchbegriff *notepad*, aufrufen können. (Alternativ finden Sie den Editor je nach Windows-Version in der App-Liste unter der Kategorie **Windows-Zubehör** bzw. im Startmenü unter **Alle Programme/Zubehör**.)

   Unter Linux können Sie z. B. den *vi* oder *KWrite* verwenden.
6. Speichern Sie zur Probe im Verzeichnis *C:\Java\Kap02* eine Datei *HalloWelt.java*. Die Datei kann ruhig leer sein (oder einen beliebigen Text enthalten).

Achtung!

Achten Sie darauf, dass der Editor nicht die Dateiendung *.txt* an die gespeicherten Dateien anhängt (also aus *Dateiname.java* die *Dateiname.java.txt* macht). Nach Installation des JDK sollte dies an sich nicht passieren. Falls doch, haben Sie zwei Möglichkeiten. Die erste Lösung besteht darin, den kompletten Dateinamen, samt Dateiendung, in Anführungszeichen zu setzen: `"Dateiname.java"`. Die zweite Möglichkeit ist, die Dateiendung *.java* im Windows Explorer zu registrieren. Speichern Sie dazu nach Methode 1 eine Datei mit der Dateiendung *.java*. Wechseln Sie danach in den Windows Explorer und doppelklicken Sie auf die Datei. Ist die Dateiendung noch nicht registriert, erscheint jetzt ein Dialogfenster (oder eine Folge von Dialogfenstern), in denen Sie *Notepad* als gewünschtes Bearbeitungsprogramm auswählen und die Option setzen können, die dafür sorgt, dass dieser Dateityp immer mit dem ausgewählten Programm geöffnet wird. Wenn Sie anschließend den Dialog abschicken, wird die Dateiendung *.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 angehängten *.txt*-Dateiendungen haben.

7. Öffnen Sie ein Konsolenfenster.

   Im Gegensatz zu Linux-Anwendern sind viele Windows-Anwender heutzutage gar nicht mehr mit dem Umgang mit der Konsole vertraut. Unter Windows heißt die Konsole meist „Eingabeaufforderung“ und wird am einfachsten durch die Suche nach *Eingabeaufforderung* aufgerufen. (Alternativ finden Windows-10-Anwender die Eingabeaufforderung im **Alt+X**-Menü bzw. in der App-Liste unter der Kategorie **Windows-System** und Windows-7-Anwender im **Start/Alle Programme/Zubehör**-Menü.)

   Unter Linux heißt die Konsole oft auch „**Terminal**“ und kann über entsprechende Desktop-Symbole oder Links im Start-Menü geöffnet werden.

   Die Konsole ist ein zeilenorientiertes Befehlseingabefenster, in deren jeweils letzten Zeile Sie einen Systembefehl eingeben und mit der **Enter**-Taste abschicken können. Beachten Sie auch den „Prompt“ zu Beginn der Zeile.

   Der Prompt zeigt Ihnen an, dass die Konsole auf die **Eingabe** eines Befehls wartet. Das Aussehen des Prompts ist veränderbar. Meist ist er so konfiguriert, dass der Pfad zu dem aktuellen Verzeichnis und ein abschließendes > angezeigt werden. Hinter dem > tippen Sie Ihren Befehl ein.

   `C:\Java\Kap02>` *IhrBefehl*

Eine kurze Einführung in die Bedienung der Konsole finden Sie als Tutorial auf unserer Website *www.carpelibrum.de*.

8. In der Konsole wechseln Sie mithilfe des *cd*-Befehls („change directory“) in das Verzeichnis, in dem die zu kompilierende Quelldatei steht. Nehmen wir an, es war dies das Ver-

zeichnis *C:\Java\Kap02*. Dann tippen Sie hinter dem Prompt der Konsole den Befehl *cd C:\Java\Kap02*[4] ein und schicken ihn durch Drücken der **Enter**-Taste ab.

Tabelle 1.1 cd-Befehle der Konsole

cd-Befehl	Beschreibung
E:	Wechsel zum Laufwerk *E:*
cd ..	Wechsel in das übergeordnete Verzeichnis
cd Verzeichnis	Wechsel in das angegebene untergeordnete Verzeichnis
cd C:/Projekte/Verzeichnis	Wechsel in das angegebene Verzeichnis

Ihr Desktop sollte nun ungefähr wie in Bild 1.3 aussehen.

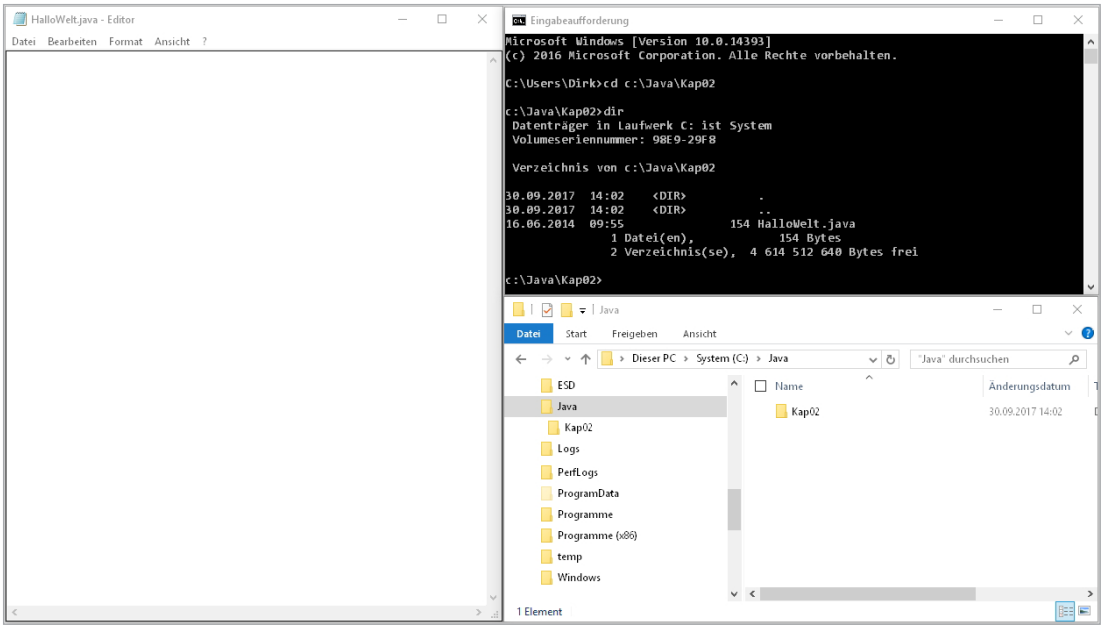

Bild 1.3 Desktop mit Editor, Konsole und Explorer (unter Windows 7)

Im Editor setzen Sie Ihren Quelltexte auf. Den fertig bearbeiteten Quelltext speichern Sie im zugehörigen Programmverzeichnis.

Nach dem Speichern wechseln Sie in die Konsole, von wo aus Sie das Programm kompilieren und ausführen (die entsprechenden Befehle werden Sie gleich im nächsten Kapitel kennenlernen).

[4] Achtung: Falls Sie die Datei auf einer anderen Partition der Festplatte abgespeichert haben als die Windows-Standardpartition *C:* (z. B. *D:*), dann müssen Sie vor dem *cd*-Befehl erst die richtige Partition anwählen, z. B. durch *d:* Enter und dann erst *cd D:\Java\Kap02*!

Nutzen Sie die Konsolen-History!

Treten bei der Kompilierung Fehler auf, korrigieren Sie die entsprechenden Zeilen im Quelltext, speichern, wechseln zur Konsole und kompilieren erneut. Die Konsolenbefehle müssen Sie dazu nicht noch einmal neu eingeben. Die Konsole merkt sich alle bereits abgeschickten Befehle in ihrer History, die Sie mithilfe der Pfeiltasten durchgehen können.

Damit sind Sie gerüstet und das Abenteuer kann beginnen!

Integrierte Entwicklungsumgebungen

Die Arbeit mit dem JDK ist wegen der vielen, nur von der Konsole aus zu bedienenden Programme etwas unhandlich. Viele Programmierer greifen daher zu integrierten Entwicklungsumgebungen, deren grafische Benutzeroberfläche den Komfort eines übergeordneten Bedienpults bietet, von dem aus sämtliche Programmierarbeiten bequem erledigt werden können. Die meisten integrierten Entwicklungsumgebungen warten daher nicht nur mit Menübefehlen zum Kompilieren und Ausführen der Programme auf, sondern auch mit

- integrierten Editoren, die speziell für die Erstellung von Quelltexten ausgelegt sind und den Programmierer mit Optionen wie Zeilennummerierung, Syntaxhervorhebung, automatischer Zeileneinzug, Codevervollständigung und Einblendung von Hilfetexten erfreuen,
- integriertem Debugger zum Aufspüren von Laufzeitfehlern,
- einer ausgeklügelten Projektverwaltung, die dem Programmierer dabei hilft, die Dateien seiner Programme (Quelltextdateien, kompilierte *.class*-Dateien, Ressourcen wie Bild- oder Sounddateien) übersichtlich zu verwalten,
- vorgefertigten Programmgerüsten und
- visueller Programmierung. (Betrifft vor allem die Erzeugung von Anwendungen mit grafischer Benutzeroberfläche (GUI). Die Fenster der Benutzeroberfläche können in einem grafischen Editor aufgebaut und bearbeitet werden, den zugehörigen Java-Code erzeugt die Entwicklungsumgebung automatisch.)

Grundsätzlich möchten wir Ihnen aber raten, anfangs auf den Einsatz einer integrierten Entwicklungsumgebung zu verzichten und rein mit dem JDK zu arbeiten. Gerade weil dem JDK der Komfort einer integrierten Entwicklungsumgebung und die Leichtigkeit und trügerische Sicherheit der visuellen Programmierung fehlt, halten wir ihn für den Einstieg ideal. Wir möchten, dass Sie erst einmal selbst lernen, wie man gute und sichere Programme schreibt, und sich nicht gleich von Anfang an auf eine Entwicklungsumgebung verlassen, die Ihnen zwar vieles abnimmt, aber auch vieles vor Ihnen verbirgt. Wir wollen nicht, dass Sie einfach aufs Geratewohl programmieren, Sie sollten auch verstanden haben, was Sie programmieren. Später können Sie dann jederzeit auf eine integrierte Entwicklungsumgebung umsteigen.

2 Der erste Kontakt

Bestimmt sind Sie schon ganz gespannt und wollen nun endlich wissen, wie man mit Java Programme schreiben kann. Wir werden uns daher auch nicht mehr mit langen Vorreden aufhalten, sondern gleich mit einem einfachen Beispiel beginnen – einer Java-Anwendung, die Sie mit einem freundlichen Hallo von der Konsole aus grüßt. Das Programm wird nur aus wenigen Zeilen Quelltext bestehen, aber diese werden es in sich haben. Sie sollten darüber jedoch nicht erschrecken und Sie brauchen sich auch nicht darum zu sorgen, wie es mit der Java-Programmierung weitergeht, wenn schon die einfachsten Programme so kompliziert und unverständlich sind. Das Problem, speziell für uns als Autoren, liegt darin, dass man in Java auch für die einfachsten Programme auf eine Reihe weit fortgeschrittener Konzepte vorgreifen muss. Versuchen wir einfach, aus der Not eine Tugend zu machen. Anstatt gleich alles bis ins Detail verstehen zu wollen, verschaffen wir uns erst einmal einen Überblick.

2.1 Die erste Java-Anwendung

In vielen Lehrbüchern über Programmiersprachen beginnt man mit dem Erstellen eines kleinen Programms, das die Meldung „Hello World“ auf den Bildschirm ausgibt. Wir wollen uns dieser Tradition anschließen und eine Anwendung erzeugen, die sich mit einem freudigen „Hallo Welt“ meldet.

Und so sieht das Programm aus:

Listing 2.1 HalloWelt

```
// Dies ist die erste Anwendung

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

Anwendungen erstellen und ausführen

Um diese – und jede andere – Anwendung auf Ihrem Computer zu erstellen und auszuführen, gehen Sie folgendermaßen vor:

1. *Öffnen Sie Ihren Texteditor.*

   Rufen Sie einen beliebigen Texteditor auf, mit dem Sie Ihren Quellcode als reinen Text (ASCII oder ANSI, kein Unicode) abspeichern können.

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 `public`-Klasse (hier also `HalloWelt`), wobei auch die Groß- und Kleinschreibung zu beachten ist.

   Weiterhin wichtig ist, dass die Datei die Dateiendung *.java* trägt und der Editor nicht eigenmächtig eine eigene Dateiendung anhängt (vgl. Anmerkung zu Notepad in Schritt 5 von Abschnitt 1.6 „Einrichtung einer eigenen Entwicklungsumgebung“).

Achtung!

Zu Anfang sollten Sie die Quelltexte bitte nicht aus der Beispielsammlung (siehe Anhang F) kopieren, sondern die Texte wirklich selbst eintippen. Es werden sich dabei zwar manche Tippfehler einschleichen, doch aus der Beseitigung dieser Fehler lernen Sie! Greifen Sie auf die Quelltexte der Beispielsammlung nur als letzte Referenz zurück, wenn Sie Ihre Programme gar nicht zum Laufen bringen.

3. *Kompilieren Sie den Quelltext.*

   Falls Sie es nicht bereits bei der Lektüre von Abschnitt 1.6 „Einrichtung einer eigenen Entwicklungsumgebung“ getan haben, öffnen Sie jetzt ein Konsolenfenster und wechseln Sie in das Verzeichnis Ihrer Java-Quelltextdatei. Von dort rufen Sie den Java-Compiler *javac* auf und übergeben ihm die zu kompilierende Quelltextdatei:

   Prompt:> *javac HalloWelt.java* **Enter**

   Dieser Aufruf erzeugt eine ausführbare Bytecode-Datei mit dem Namen der übergebenen Quelltextdatei, allerdings mit der Endung *.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. Sie müssen dann dem Programmnamen *javac* den vollständigen Pfad voranstellen, der zu dem Programm führt. Wenn Sie beispielsweise Java im Verzeichnis *C:\Program Files\Java\jdk-9* installiert haben, würde der Aufruf *C:\Program Files\Java\jdk-9\bin\javac* lauten. Bequemer ist es, wenn Sie den Pfad zu den Java-Entwicklungsprogrammen in Ihren Systempfad eintragen (siehe Anhang B „Installation des JDK“).

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 Konsole wird eingegeben:

   Prompt:> *java HalloWelt*Enter

   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 mithilfe des DOS-Befehls *dir*, ob die Datei *HalloWelt.class* im Verzeichnis angelegt wurde).

   Möglich ist auch, dass Sie aus Versehen die Dateiendung *.class* angehängt oder den Klassennamen nicht exakt so eingegeben haben, wie er im Quelltext definiert ist (auf gleiche Groß- und Kleinschreibung achten).

   Manchmal liegt es aber auch 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 genau geht, ist im Anhang B beschrieben.

```
Eingabeaufforderung
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Dirk>cd C:\Java\Kap02

C:\Java\Kap02>javac HalloWelt.java

C:\Java\Kap02>dir
 Datenträger in Laufwerk C: ist System
 Volumeseriennummer: 98E9-29F8

 Verzeichnis von C:\Java\Kap02

30.09.2017  18:44    <DIR>          .
30.09.2017  18:44    <DIR>          ..
30.09.2017  18:44               423 HalloWelt.class
16.06.2014  09:55               154 HalloWelt.java
               2 Datei(en),            577 Bytes
               2 Verzeichnis(se),  5 188 919 296 Bytes frei

C:\Java\Kap02>java HalloWelt
```

Bild 2.1 Kompilation und Ausführung der Anwendung HalloWelt im Verzeichnis C:\Java\Kap02

Wie Sie Java-Anwendungen so aufbereiten, dass sie ohne explizite Übergabe an *java* ausgeführt werden können, lesen Sie in Kapitel 16.

Der Quelltext der Anwendung

```
01 // Dies ist die erste Anwendung
02
03 public class HalloWelt {
04   public static void main (String[] args) {
05     System.out.println("Hallo Welt!");
06   }
07 }
```

Achtung!

Die Zeilennummern sind nicht Teil des Listings! Sie sollen Ihnen lediglich helfen, die besprochenen Codestellen leichter zu finden.

Analyse:

Gehen wir nun den Quelltext des Programms durch. Dabei sollte es uns nicht darum gehen, alle syntaktischen Elemente von Java auf einmal verstehen zu wollen. Wichtiger ist es zu verstehen, wie das Grundgerüst einer Java-Anwendung aussieht und woran der Compiler eine Java-Anwendung erkennt.

```
// Dies ist die erste Anwendung
```

Die erste Zeile des Quelltextes ist ein Kommentar. Eingeleitet wird der Kommentar durch die doppelten Schrägstriche //, die dem Compiler mitteilen, dass der Rest der Zeile nur als Gedächtnisstütze für den Programmierer gedacht ist und bei der Übersetzung ignoriert werden kann.

Mehrzeilige Kommentare können Sie mithilfe der „Klammern“ /* und */ einfügen:

```
/* Dies ist
alles ein
Kommentar */
```

Unter dem Kommentar wird es dann schon recht kryptisch. Springen wir gleich in die fünfte Zeile, in der wir den Text wiedererkennen, den die Anwendung ausgeben soll:

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

Damit der Compiler erkennt, dass es sich bei „Hallo Welt!“ um einen einfachen Text und nicht etwa um einen Programmierbefehl handelt, wird der Text in Anführungszeichen gesetzt. Woher aber weiß das Programm, was es mit diesem Text zu tun hat? Dazu bedarf es spezieller Anweisungen, die den Text auf Ihrem Bildschirm ausgeben, die Sie glücklicherweise aber nicht selbst zu implementieren haben. Es gibt schon eine vordefinierte Folge von Anweisungen, die als sogenannte Methode unter dem Namen `println()` zum Standardumfang von Java gehört. Um einen Text auszugeben, brauchen Sie diesen also lediglich der

Methode `println()` zu übergeben, wozu Sie den Text in Anführungszeichen zwischen die Klammern nach dem Methodennamen schreiben.

Wir sind damit auf ein ganz zentrales Konzept zur Modularisierung von Programmcode gestoßen.

Modularisierung durch Methoden

Grundsätzlich bestehen Programme aus Anweisungen – einzelnen Befehlen, die der Computer nacheinander ausführen soll (in den nächsten Kapiteln werden wir viele Beispiele für solche Anweisungen sehen). Man könnte nun ein Programm so aufsetzen, dass man die Befehle einfach in der Reihenfolge, in der sie abgearbeitet werden, untereinander aufschreibt (wie es früher in Basic der Fall war). Nun kehren aber bestimmte Aufgaben bei der Programmierung immer wieder, so zum Beispiel die Ausgabe von Text auf einen Bildschirm, die Berechnung von Sinuswerten, das Öffnen einer Datei und so weiter und so fort. Damit man nun nicht fortwährend das Rad neu erfinden muss, fasst man die Anweisungen zur Lösung eines solches Problems in einer Methode zusammen. Fortan muss man zur Ausgabe eines Textes nicht mehr die ganzen Anweisungen zur Textausgabe niederschreiben, sondern braucht nur noch die entsprechende Methode aufzurufen. Und da man den auszugebenden Text der Methode erst beim Aufruf übergibt, kann die Methode sogar beliebigen Text ausgeben (wie dies im Einzelnen funktioniert, werden wir in Abschnitt 4.2 klären).

Merksatz

Methoden kapseln Quellcode und dienen der Lösung genau definierter Teilprobleme.

Glücklicherweise sind für die wichtigsten Probleme bereits entsprechende Methoden vorhanden – sie gehören zur Standardausstattung von Java (wie zum Beispiel `println()` für die Textausgabe). Die Frage ist nur: Wo findet man diese Methoden? Oder um die Frage zu verallgemeinern: Wie wird der Quellcode in Java-Programmen organisiert?

Methoden, Klassen, Pakete

In Java-Programmen dürfen Anweisungen (also der eigentliche Code, der vom Computer ausgeführt werden soll) nicht einfach irgendwo im Quelltext herumstehen. Vielmehr ist es so, dass Anweisungen nur innerhalb von Methodendefinitionen erlaubt sind.

Aber auch Methodendefinitionen dürfen nicht an beliebiger Stelle stehen. Methoden sind in Java nur als Elemente von Klassen erlaubt (in einigen anderen Programmiersprachen können Methoden auch allein außerhalb von Klassen definiert werden und man nennt sie dann zur Unterscheidung *Funktionen*).

Klassen dürfen dagegen an beliebiger Stelle im Quelltext definiert werden.

Merksatz

Um es also noch einmal in aller Deutlichkeit zu sagen: Java-Programme bestehen praktisch nur aus Klassendefinitionen. Diese Klassen definieren Variablen und Methoden und nur in diesen Methoden stehen die eigentlichen Anweisungen, die das Programm auszuführen hat.

Betrachten wir jetzt noch einmal den Aufbau unseres ersten Programms:

```
01 // Dies ist die erste Anwendung
02
03 public class HalloWelt {
04   public static void main (String[] args) {
05     System.out.println("Hallo Welt!");
06   }
07 }
```

Analyse:

In Zeile 3 definieren wir eine eigene Klasse namens `HalloWelt`. Beginn und Ende der Klassendefinition werden durch das geschweifte Klammernpaar gekennzeichnet. Die Angabe `public` ist ein sogenannter Modifizierer und legt fest, welche anderen Klassen auf `HalloWelt` zugreifen dürfen[1]. Innerhalb der Klasse `HalloWelt` definieren wir die Methode `main()` (Zeile 4 bis 6). Die Anweisungen, die zu der Methode gehören, werden wiederum in geschweifte Klammern gefasst. In unserem Fall ist die einzige Anweisung der Aufruf der Methode `println()`.

Ähnlich wie man Methoden in Klassen organisiert (wobei alle Methoden einer Klasse üblicherweise einem gemeinsamen Aufgabengebiet angehören), kann man Klassen in sogenannten Paketen zusammenfassen. Mehr dazu erfahren Sie in Abschnitt 3.6 und am Ende von Abschnitt 4.2.

Die Java-Standardbibliothek

Es gibt schon viele fertige Klassen, die von Java über die Java-Standardbibliothek bereitgestellt werden (beispielsweise die Klasse System.) Eine tabellarische Übersicht der wichtigsten Pakete und Klassen aus der Standardbibliothek finden Sie in Anhang D. Die vollständige (englische) Referenz der Pakete und Klassen enthält die API-Dokumentation, die Sie im Internet finden (*http://docs.oracle.com/javase/9/docs/api/*).

Der eigentliche Code, die Anweisungen des Programms, ist also verteilt auf die Methoden des Programms. Und da eine Methode eine andere Methode aufrufen kann, ist es kein Problem, den Programmablauf zu steuern und von Methode zu Methode weiterzuführen. Mit welcher Methode wird nun aber begonnen? Welche Methode wird aufgerufen, wenn wir das Programm starten?

[1] Die Hauptklasse eines Programms, die die `main()`-Methode enthält, wird grundsätzlich als `public` deklariert.

Der Programmstart

Jede Java-Anwendung beginnt mit einer `main()`-Funktion, die folglich in jeder Java-Anwendung definiert werden muss. Genau dies geschieht in unserem kleinen Beispielprogramm.

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

Da Methoden nur innerhalb von Klassen definiert werden dürfen, müssen wir zuerst die Alibi-Klasse `HalloWelt` definieren. Diese enthält dann die Definition unserer `main()`-Methode. Und das war schon das ganze Programm.

Merksatz

Das Grundgerüst einer Java-Anwendung besteht aus einer Hauptklasse, in der die Methode `main()` definiert wird. Die Ausführung der Anwendung beginnt mit der ersten Anweisung im Funktionskörper von `main()`. Die Signatur der `main()`-Methode ist fest vorgegeben:

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

System.out.println() und das Problem der Umlaute

Bevor Sie es selbst herausfinden und dann enttäuscht sind, möchten wir es Ihnen lieber gleich beichten: Mit `System.out.println()` können Sie keine Umlaute wie ä, ö oder ß auf die Konsole ausgeben. Dafür müssten Sie eine Instanz der Klasse `Console` verwenden bzw. `System.console().printf()` aufrufen.

Dies jedoch nur als Hinweis und Vorgeschmack auf Kapitel 6, wo wir uns eingehender mit der Klasse `Console` und ihren Möglichkeiten beschäftigen werden. Im Moment jedoch, da eine Menge wichtiger Programmierkonzepte auf uns einströmen, wollen wir uns nicht weiter mit möglichen Ausgabetechniken belasten und begnügen uns daher mit der sehr einfach zu verwendenden `System.out.println()`-Methode.

Damit wäre der Einstieg in die Java-Programmierung geschafft. In den nächsten Kapiteln wird die Behandlung der syntaktischen Elemente von Java im Vordergrund stehen. Sie werden lernen, aus welchen Elementen Java-Programme aufgebaut sind und wie diese Elemente korrekt und sinnvoll eingesetzt werden. Danach sollten Sie in der Lage sein, beliebige Java-Programme lesen und verstehen zu können und eigene Java-Programme zur Lösung von Problemen aufzusetzen.

2.2 Zusammenfassung

Das Grundgerüst einer Java-Anwendung besteht aus einer Hauptklasse, die als `public class Klassenname` deklariert wird, mit einer Methode `public static void main (String[] args)`. Die Ausführung einer Java-Anwendung beginnt mit der ersten Anweisung in der `main()`-Methode.

2.3 Fragen und Antworten

1. Wie hängen Quelltext, Bytecode und Maschinencode zusammen?

   Quelltext wird in Java-Syntax aufgesetzt und ist für den Computer unverständlich. Der Quelltext muss daher vom Compiler übersetzt werden. Der Java-Compiler erzeugt aus dem Quelltext binären Bytecode, der noch prozessorunspezifisch und daher gut portabel ist. Erst der Java-Interpreter erzeugt aus dem Bytecode prozessorspezifischen Maschinencode, der auf dem Computer ausgeführt werden kann.

2. Wozu braucht man einen Java-Compiler, wozu einen Java-Interpreter?

   Der Java-Compiler übersetzt Ihren Quelltext in prozessorunspezifischen Bytecode. Dieser kann mithilfe eines passenden Java-Interpreters auf einem Computer ausgeführt werden. Den Compiler benötigt man also für die Erstellung, den Interpreter für die Ausführung von Java-Programmen.

3. Eine Frage für C++-Programmierer: Wo ist der Linker versteckt?

   Im Interpreter.

2.4 Übungen

1. Schreiben Sie eine eigene Anwendung, die folgenden Text ausgibt: „Hallo Welt. Hier ist <Ihr Name>“.
2. Speichern Sie den Quelltext Ihrer Anwendung unter einem Namen, der nicht wie der Name Ihrer Hauptklasse lautet. Versuchen Sie, die Anwendung zu kompilieren (speichern Sie danach den Quelltext wieder unter dem Namen der Hauptklasse).

3 Von Daten, Operatoren und Objekten

Anscheinend haben Sie dieses Buch nach dem vorangehenden Kapitel doch nicht in den Altpapiercontainer geworfen und das ist auch gut so. Denn der schwierigste, weil verwirrendste Teil liegt bereits hinter uns! Der erste Kontakt mit einer Programmiersprache ist nicht gerade leicht – und wenn dies schon für Sprachen wie Basic oder C gilt, so ist es erst recht wahr für Java mit seinem konsequent objektorientierten Aufbau. Viele unbekannte Konzepte, seltsame Schreibweisen und Begriffe prasseln da auf den Anfänger ein und die nagende Frage taucht auf: Soll ich mir das antun?

„Der Zweifel ist's, der Gutes böse macht!" (*Goethe, Iphigenie auf Tauris*)

Halten Sie also noch ein bisschen durch, ab diesem Kapitel wird alles leichter. Fortan werden wir systematisch an die Sache herangehen, wir werden uns die aufregende Welt der Java-Programmierung Stück für Stück erobern und unserer eigenen Kreativität als Programmierer Tür und Tor öffnen.

3.1 Variablen und Anweisungen

Die Aufgabe eines jeden Computerprogramms ist die Verarbeitung von irgendwelchen Informationen, die im Computerjargon meist Daten genannt werden. Das können Zahlen sein, aber auch Buchstaben, ganze Texte oder Bilder und Zeichnungen. Dem Rechner ist diese Unterscheidung gleich, da er letztlich alle Daten in Form von endlosen Zahlenkolonnen in Binärdarstellung (nur Nullen und Einsen) verarbeitet.

Zahlensysteme

Erinnern Sie sich noch, als in der Schule die verschiedenen Zahlensysteme durchgenommen wurden: das uns so vertraute aus Indien stammende Zehnersystem, das babylonische Sexagesimalsystem und das künstlich anmutende Dual- oder *Binärsystem*? Ich für meinen Teil fand es ebenso interessant zu erfahren, dass die Einteilung unserer Stunden in 60 statt 100 Minuten auf die Babylonier zurückgeht, wie es mich langweilte, Zahlen ins Dualsystem umzurechnen und als Folge von Nullen und Einsen darzustellen. Wer ist denn so dumm, freiwillig mit Binärzahlen zu rechnen? Nun, ich wünschte, meine Lehrer hätten mich

gewarnt, aber vermutlich wussten die Lehrer damals selbst noch nicht, was man alles mit Binärzahlen anfangen kann (vielleicht haben die Lehrer uns ja auch gewarnt und wir haben es nur verschlafen). Jedenfalls rechnen Computer nur im Binärsystem:

- zum einem, weil die beiden einzigen möglichen Werte 0 und 1 sich gut mit elektronischen Signalen darstellen lassen (Strom an, Strom aus),
- zum anderen, weil es sich im Binärsystem sehr leicht rechnen lässt, vorausgesetzt man stößt sich nicht an der kryptischen Darstellung der Zahlen.

Damit wären wir wieder beim eigentlichen Thema. Für den Computer sind also sämtliche Daten (nicht nur die Zahlen) Folgen von Nullen und Einsen, weil er diese am schnellsten und einfachsten verarbeiten kann. Wie diese Daten und die Ergebnisse seiner Berechnungen zu interpretieren sind, ist dabei nicht sein Problem – es ist unser Problem.

Datentypen machen das Leben leichter

Stellen Sie sich vor, wir schreiben das Jahr 1960 und Sie sind stolzer Besitzer einer Rechenmaschine, die Zahlen und Text verarbeiten kann. Beides allerdings in Binärformat. Um Ihre Freunde zu beeindrucken, lassen Sie den „Computer“ eine kleine Subtraktion berechnen, sagen wir:

```
8754 – 398 = ?
```

Zuerst rechnen Sie die beiden Zahlen durch fortgesetzte Division durch 2 ins Binärsystem um (wobei die nicht teilbaren Reste der aufeinanderfolgenden Divisionen, von rechts nach links geschrieben, die gewünschte Binärzahl ergeben).

```
10001000110010 – 110001110 = ?
```

Die Binärzahlen stanzen Sie sodann als Lochkarte und lassen diese von Ihrem Computer einlesen. Dann drücken Sie noch die Taste für Subtraktion und ohne Verzögerung erscheint das korrekte Ergebnis:

```
10000010100100
```

Zweifelsohne werden Ihre Freunde von dieser Maschine äußerst beeindruckt sein und ich selbst wünschte, ich hätte im Mathematikunterricht eine derartige praktische Hilfe gehabt. Trotzdem lässt sich nicht leugnen, dass die Interpretation der Binärzahlen etwas unhandlich ist, und zwar erst recht, wenn man neben einfachen ganzen Zahlen auch Fließkommazahlen, Texte und Bitmaps im Binärformat speichert.

Für die Anwender von Computern ist dies natürlich nicht zumutbar und die Computerrevolution – die vierte der großen Revolutionen (nach der Glorious Revolution, England 1688, der französischen Revolution von 1789 und der Oktoberrevolution, 1917 in Russland) – hätte nicht stattgefunden, hätte man nicht einen Ausweg gefunden. Dieser bestand nun einfach darin, es der Software – dem laufenden Programm – zu überlassen, die vom Anwender eingegebenen Daten (seien es Zahlen, Text, Bitmaps etc.) in Binärformat umzuwandeln und umgekehrt die auszugebenden Daten wieder vom Binärformat in eine leicht lesbare Form zu verwandeln.

„Gemeinheit“, höre ich Sie aufbegehren, „da wurde das Problem ja nur vom Anwender auf den Programmierer abgewälzt.“ Ganz so schlimm ist es nicht. Der Java-Compiler nimmt uns hier das Gröbste ab. Alles, was wir zu tun haben, ist, dem Compiler anzugeben, mit welchen Daten wir arbeiten möchten und welchem Datentyp diese Daten angehören (sprich, ob es sich um Zahlen, Text oder Sonstiges handelt).

Schauen wir uns gleich mal ein Beispiel an:

```
public class ErstesBeispiel {
  public static void main(String[] args) {
    int ersteZahl;
    int zweiteZahl;
    int ergebnis;

    ersteZahl = 8754;
    zweiteZahl = 398;
    System.out.println(" 1. Zahl  = " + ersteZahl);
    System.out.println(" 2. Zahl  = " + zweiteZahl);
  }
}
```

Das Grundgerüst, das bereits in Abschnitt 2.1 vorgestellt wurde, übernehmen wir einfach wie gehabt. Wenden wir unsere Aufmerksamkeit gleich den Vorgängen in der `main()`-Funktion zu.

Dort werden zuerst die für die Berechnung benötigten Variablen deklariert.

Variablen

Die Variablen eines Programms sind nicht mit den Variablen mathematischer Berechnungen gleichzusetzen. *Variablen* bezeichnen Speicherbereiche im RAM (Arbeitsspeicher), in denen ein Programm Werte ablegen kann. Um also mit Daten arbeiten zu können, müssen Sie zuerst eine Variable für diese Daten deklarieren. Der Compiler sorgt dann dafür, dass bei Ausführung des Programms Arbeitsspeicher für die Variable reserviert wird. Für den Compiler ist der Variablenname einfach ein Verweis auf den Anfang eines Speicherbereichs. Als Programmierer identifiziert man eine Variable mehr mit dem Wert, der gerade in dem zugehörigen Speicherbereich abgelegt ist.

Bei der *Deklaration* geben Sie nicht nur den Namen der Variablen an, sondern auch deren Datentyp. Dieser Datentyp gibt dem Compiler an, wie der Inhalt des Speicherbereichs der Variablen zu interpretieren ist. Im obigen Beispiel benutzen wir nur den Datentyp `int`, der für einfache Ganzzahlen steht.

Merksatz

Zu jeder Variablendeklaration gehört auch die Angabe eines *Datentyps*. Dieser gibt dem Compiler an, wie der Speicherinhalt der Variablen zu interpretieren ist.

```
int ersteZahl;
```

Dank des Datentyps können wir der Variablen `ersteZahl` direkt eine Ganzzahl zuweisen und brauchen nicht wie im obigen Beispiel des Lochkartenrechners die Dezimalzahl in Binärcode umzurechnen:

```
ersteZahl = 8754;
```

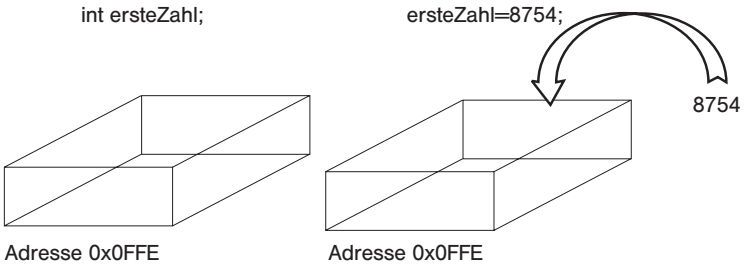

Bild 3.1 Deklaration und Zuweisung

Der „Wert“ der Variablen

Wenn eine Variable einen Speicherbereich bezeichnet, dann ist der Wert einer Variablen der interpretierte Inhalt des Speicherbereichs. Im obigen Beispiel wäre der Wert der Variablen `ersteZahl` nach der Anweisung

```
ersteZahl = 8754;
```

also 8754. Wenn Sie der Variablen danach einen anderen Wert zuweisen würden, beispielsweise

```
ersteZahl = 5;
```

wäre der Wert in der Folge gleich 5.

Achtung!

= bedeutet Zuweisung.

== bedeutet Vergleich.

Was die Variablen für den Programmierer aber so wertvoll macht, ist, dass er sich nicht mehr um die Speicherverwaltung zu kümmern braucht. Es ist zwar von Vorteil, wenn man weiß, dass hinter einer Variablen ein Speicherbereich steht, für die tägliche Programmierarbeit ist es aber meist nicht erforderlich. Wir sprechen nicht davon, dass wir mithilfe des Variablennamens einen eindeutig bezeichneten Platz im Arbeitsspeicher referenzieren und in diesen einen Wert schreiben, wir sagen einfach, dass wir der Variablen einen Wert zuweisen. Wir sprechen nicht davon, dass das interpretierte Bitmuster in dem Speicherbereich der `int`-Variablen `ersteZahl` gleich 5 ist, wir sagen einfach, `ersteZahl` ist gleich 5. Wir sprechen nicht davon, dass wir mithilfe des Variablennamens einen eindeutig bezeichneten