1 Vorbemerkungen und Überblick

Im beginnenden neuen Jahrtausend ist es eigentlich nicht mehr notwendig, Begriffe wie Computer, Programm oder Software einzuführen. Wir werden in diesem Kapitel trotzdem einige Vorbemerkungen machen, um den Kontext dieses Buches und der behandelten Begriffe zu verdeutlichen.

1.1 Informatik, Algorithmen und Datenstrukturen

Informatik

Informatik ist ein Kunstwort aus den 60er Jahren, das die Assoziationen Informatik gleich Information oder Technik *oder* Informatik gleich Information und Mathematik erwecken sollte. Bei der Begriffsbildung sollte durchaus bewusst ein Gegensatz zum amerikanischen Begriff *Computer Science* aufgebaut werden, um zu verdeutlichen, dass die Wissenschaft Informatik nicht nur auf Computer beschränkt ist. Informatik als Begriff ist insbesondere im nicht englischsprachigen europäischen Raum gebräuchlich. Die Informatik hat zentral zu tun mit

- systematischer Verarbeitung von Informationen und
- Maschinen, die diese Verarbeitung automatisch leisten.

Wichtige Grundkonzepte der Informatik können in einer maschinenunabhängigen Darstellung vermittelt werden. Der Bezug zu den Themen diese Buches kann durch die folgende Aussage hergestellt werden:

Algorithmen und Daten

> Die »systematische Verarbeitung« wird durch den Begriff *Algorithmus* präzisiert, Information durch den Begriff *Daten*.

In einer ersten Näherung kann man das Konzept des Algorithmus wie folgt charakterisieren:

> Ein *Algorithmus* ist eine eindeutige Beschreibung eines in mehreren Schritten durchgeführten (Bearbeitungs-)Vorganges.

In der Informatik werden nun speziell *Berechnungsvorgänge* statt allgemeiner Bearbeitungsvorgänge betrachtet, wobei der Schwerpunkt auf der *Ausführbarkeit* durch (abstrakte) Maschinen liegt, die auch als Prozessoren bezeichnet werden:

Prozessor

> Ein *Prozessor* führt einen Prozess (Arbeitsvorgang) auf Basis einer eindeutig interpretierbaren Beschreibung (dem Algorithmus) aus.

In diesem Buch werden eine Reihe von Fragestellungen behandelt, die Aspekte des Umgangs mit Algorithmen betreffen. Die verschiedenen *Notationen* für die Beschreibung von Algorithmen führen direkt zu Sprachkonzepten moderner Programmiersprachen.

Ausdrucksfähigkeit verschiedener Notationen

Wenn verschiedene Notationen verwendet werden können, stellt sich die Frage der *Ausdrucksfähigkeit* dieser Algorithmensprachen. Man kann sich diese Fragestellungen daran verdeutlichen, dass man sich überlegt, wie ausdrucksfähig die Bienensprache, die ja konkrete Wegbeschreibungen ermöglicht, im Vergleich zu einer Programmiersprache zur Wegfindungsprogrammierung von Robotern ist, oder ob dressierte Hunde tatsächlich dieselbe Sprache verstehen wie Menschen.

Korrektheit und Zeitbedarf

Spätestens beim Übergang zu konkreten Programmen in einer Programmiersprache müssen natürlich die Fragestellungen der *Korrektheit* und des *Zeitbedarfs* bzw. der Geschwindigkeit von Programmen betrachtet werden.

In diesem Buch geht es primär um Algorithmen für Rechner, also schnelle (aber dumme) Prozessoren (ein Rechner ist im Vergleich zu einem Menschen ein »Hochgeschwindigkeitstrottel«, der falsche Anweisungen prinzipiell nicht erkennen kann, aber sehr schnell als Programm vorgegebene Anweisungen ausführen kann). Hierzu werden mathematisch formale Grundlagen von Algorithmen eingeführt – ein Rechner »versteht« nur Bits, so dass jede Anweisungssprache auf diese einfache Ebene heruntergebrochen werden muss. Die Umsetzung von menschenverständlichen Algorithmen in eine maschinennahe Darstellung geht natürlich nur, wenn die Bedeutung beider Darstellungen exakt formal festgelegt ist.

Datenstrukturen

Auch wenn sich beim Programmieren scheinbar alles um die Algorithmen dreht, ist das Gegenstück, die *Datenstrukturen*, ebenfalls ein zentraler Aspekt der Grundlagen der Informatik. Erst rechnerverarbeitbare Darstellungen von Informationen erlauben das Program-

mieren realistischer Probleme auf einer angemessenen Abstraktionsebene – die Zeit, in der ein Programmierer sich an den Rechner anpassen und direkt in Bits und Bytes denken musste, ist zumindest in der nicht stark hardwarenahen Programmierung vorbei.

1.2 Historischer Überblick: Algorithmen

Die in diesem Buch behandelten Konzepte der Informatik sind älter als die Geschichte der Computer, die Programme ausführen können. Die Informatik behandelt Konzepte, die auch ohne existierende Computer gültig sind – aber zugegebenermaßen erst durch den Siegeszug der Computer die heutige praktische Relevanz erlangten.

In diesem Abschnitt wollen wir kurz einige Ergebnisse aus der Zeit *vor* dem Bau des ersten Computers nennen, um dies zu verdeutlichen:

300 v. Chr.: Euklids Algorithmus zur Bestimmung des `ggT` (beschrieben im 7. Buch der Elemente), also des größten gemeinsamen Teilers, mit dem etwa

$$\texttt{ggT}(300, 200) = 100$$

sehr effizient berechnet werden kann, ist die erste Beschreibung eines Verfahrens, das modernen Kriterien für Algorithmen gerecht wird.

800 n. Chr.: Der persisch-arabische Mathematiker Muhammed ibn Musa abu Djafar alChoresmi (oft auch als »al Chworesmi« oder »al Charismi« geschrieben) veröffentlicht eine Aufgabensammlung für Kaufleute und Testamentsvollstrecker, die später ins Lateinische als *Liber Algorithmi* übersetzt wird.
Das Wort Algorithmus ist ein Kunstwort aus dem Namen dieses Mathematikers und dem griechischen »arithmos« für Zahl.

1574: Adam Rieses Rechenbuch verbreitet mathematische Algorithmen in Deutschland.

1614: Die ersten Logarithmentafeln werden algorithmisch berechnet – ohne Computer dauerte dies 30 Jahre!

1703: Binäre Zahlensysteme werden von Leibnitz eingeführt. Diese Zahlensysteme bilden die Grundlage der internen Verarbeitung in modernen Computern.

1815: Augusta Ada Lovelace, die erste »Computer-Pionierin«, wird geboren. Sie entwickelt schon früh Konstruktionspläne für verschiedenartige Maschinen, wird Assistentin von Babbage und entwirft Programme für dessen erste Rechenmaschinen.

1822: Charles Babbage entwickelt die sogenannte *Difference Engine*, die in einer verbesserten Version 1833 fertiggestellt wird. Später entwickelt er auch die *Analytical Engine*, die bereits die wichtigsten Komponenten eines Computers umfasst, aber niemals vollendet wird.

1931: Gödels Unvollständigkeitssatz beendet den Traum vieler damaliger Mathematiker, die gesamte Beweisführung aller Sätze in der Mathematik mit algorithmisch konstruierten Beweisen durchzuführen.

1936: Die Church'sche These vereinheitlicht die Welt der Sprachen zur Notation von Algorithmen, indem für viele der damaligen Notationen die gleiche Ausdrucksfähigkeit postuliert wird.

danach: Mit der Realisierung der ersten Computer erfolgte natürlich der Ausbau der Algorithmentheorie zu einem eigenen Wissenschaftsgebiet.

Auch für den Bereich der Datenstrukturen können lang zurückreichende Wurzeln gefunden werden, etwa das Indexierungssystem historischer Bibliotheken.

Ein historischer Überblick, der auch die technische Informatik, die Programmierung und Aspekte des Software Engineering behandelt, würde den Rahmen des vorliegenden Buches sprengen. So werden wir nur noch im folgenden Abschnitt, der die Wahl der Sprache Java als Ausbildungssprache motivieren soll, die Historie der Programmiersprachen kurz anreißen.

1.3 Historie von Programmiersprachen und Java

Bevor wir näher auf die in diesem Buch verwendete Programmiersprache Java eingehen, lohnt es sich, einen Blick in die Geschichte zu werfen. Die Ursprünge höherer Programmiersprachen reichen zurück an den Anfang der fünfziger Jahre, als erste Sprachen wie *Autocode* vorgeschlagen wurden, die bereits arithmetische Ausdrücke, Schleifen, bedingte Sprünge und Prozeduren umfassten. Daraus wurde dann *Fortran* (FORmula TRANslator) entwickelt, deren erste Fassung 1954 vorgestellt wurde und die auch heute noch speziell im wissenschaftlich-technischen Bereich zum Einsatz kommt. Anfang der sechziger Jahre wurden dann *Algol* (ALGOrithmic Language), eine Sprache, die u.a. Pascal nachhaltig beeinflusst hat, *Lisp*, als die Grundlage aller funktionalen Programmiersprachen und im Bereich der künstlichen Intelligenz immer noch häufig eingesetzt,

Autocode

Fortran

Algol

Lisp

und die vielleicht noch am weitesten verbreitete Programmiersprache *COBOL* (COmmon Business Oriented Language) entwickelt. Auch die Entwicklung von *BASIC* geht auf die Mitte der sechziger Jahre zurück.

COBOL

Die Methode der strukturierten Programmierung wurde wesentlich durch *Pascal* begründet, eine Entwicklung von Niklaus Wirth zu Beginn der siebziger Jahre. Mit Pascal wurden nicht nur Sprachmittel eingeführt, die heute in allen modernen Programmiersprachen zu finden sind, sondern es wurde auch erstmals die Programmausführung durch die Interpretation von Zwischencode (sogenannten *P-Code*) verwirklicht. Dieses Verfahren wurde dann beispielsweise für Java »wiederentdeckt« und hat wesentlich zur Verbreitung der Sprache beigetragen.

Pascal

Ebenfalls Anfang der siebziger Jahre wurde ausgehend von Sprachen wie *CPL* und *BCPL* die Programmiersprache *C* entwickelt und zur Implementierung des Betriebssystems UNIX eingesetzt. Damit wurde erstmals eine höhere Programmiersprache für die Betriebssystementwicklung verwendet. Der sich daraus ergebenden weitgehenden Maschinenunabhängigkeit ist es letztendlich zu verdanken, dass UNIX und die Vielzahl der Derivate (wie u.a. Linux) heute auf nahezu allen Hardwareplattformen zu finden sind.

C

Die Idee modularer Programmiersprachen wurde ab Mitte der siebziger Jahre wiederum von Wirth mit *Modula* entwickelt und speziell auch in *Ada* umgesetzt.

Modula

Objektorientierte Programmiersprachen wie Smalltalk, C++ oder Java haben ihren Ursprung in *Simula-67*, die um 1967 in Norwegen als eine Sprache für die ereignisorientierte Simulation entstanden ist. Das Potenzial objektorientierter Programmierung wurde zu dieser Zeit jedoch noch nicht wirklich erkannt. Erst mit der Entwicklung von *Smalltalk* durch Xerox PARC Ende der siebziger Jahre fanden Konzepte wie »Klasse« oder «Vererbung« Einzug in die Programmierung. Ausgehend vom objektorientierten Paradigma einerseits und der Systemprogrammiersprache C andererseits wurde 1983 bei AT&T die Programmiersprache C++ entworfen, die nicht zuletzt wegen der »Abwärtskompatibilität« zu C schnell eine weite Verbreitung fand.

Simula-67

Smalltalk

C++

Als Entwicklungen der letzten Jahre vor Java sind insbesondere *Eiffel* von Bertrand Meyer mit dem neuen Konzept der Zusicherungen bzw. des vertraglichen Entwurfs (*design by contract*), *Oberon* von Wirth mit dem Ziel einer Rückbesinnung auf einen einfachen, kompakten und klaren Sprachentwurf sowie die Vielzahl von Skriptsprachen wie *Perl*, *Tcl* oder *Python* zu nennen.

Eiffel

Die Arbeiten zu Java lassen sich bis in das Jahr 1990 zurückverfolgen, als bei Sun Microsystems eine Sprache für den Consumer-

Java

Electronics-Bereich unter dem Namen *Oak* entwickelt werden sollte. Entwurfsziele dieser Sprache waren bereits Plattformunabhängigkeit durch Verwendung von Zwischencode und dessen interpretative Ausführung, Objektorientierung und die Anlehnung an C/C++, um so den Lernaufwand für Kenner dieser Sprachen gering zu halten. Mit der Entwicklung des World Wide Web um 1993 fand im Entwicklerteam eine Umorientierung auf Webanwendungen und damit eine Umbenennung in *Java* – die bevorzugte Kaffeesorte der Entwickler – statt.

HotJava

1995 wurde die Sprache dann zusammen mit einer Referenzimplementierung und dem *HotJava*-Browser der Öffentlichkeit vorgestellt. Dieser Browser ermöglichte erstmals aktive Webinhalte – durch sogenannte Applets. Hinter diesem Begriff verbergen sich kleine Java-Programme, die als Teil eines Webdokumentes von einem Server geladen und in einem geschützten Bereich des Browsers ausgeführt werden können. Da der HotJava-Browser selbst in Java geschrieben wurde, war dies gleichzeitig die Demonstration der Eignung von Java für größere Projekte. Als Netscape – damals unbestrittener Marktführer bei Webbrowsern – kurz darauf die Unterstützung von Java-Applets in der nächsten Version des Navigators bekannt gab, fand Java in kürzester Zeit eine enorme Verbreitung. Dies wurde auch noch dadurch verstärkt, dass die Entwicklungsumgebung für Java von Sun kostenlos über das Web zur Verfügung gestellt wurde. Im Jahre 1997 wurde dann die Version 1.1 des *Java Development Kit (JDK)* freigegeben und alle großen Softwarefirmen wie IBM, Oracle oder Microsoft erklärten ihre Unterstützung der Java-Plattform. Mit der Veröffentlichung von Java 2 im Jahr 1998 und der Folgeversionen des JDK hat sich dann nicht nur die Anzahl der Systemplattformen, auf denen Java-Programme lauffähig sind, vergrößert, sondern insbesondere auch die Zahl der Klassenbibliotheken, die für Java verfügbar sind.

Netscape

Java Development Kit

J2SE 5.0

In der im Sommer 2005 freigegebenen Version *J2SE 5.0* (Java 2 Platform Standard Edition) wurden schließlich auch einige neue Sprachkonzepte wie Generics und variable Parameterlisten eingeführt, auf die wir an geeigneter Stelle eingehen werden. Die Nachfolgeversion *Java SE 6.0* erschien im Dezember 2006 und hat seitdem einige Updates erfahren. Wesentliche Änderungen an der Sprache selbst bietet diese Version jedoch nicht. Seit 2006 ist Java auch als Open-Source-Software – das OpenJDK – verfügbar. 2009/2010 wurde die Firma Sun Microsystems, und damit auch die Java-Technologien, vom Datenbankhersteller Oracle gekauft. Die Weiterentwicklung von Java, die im sogenannten *Java Community Process (JCP)* stattfindet, wird nun also von Oracle geleitet. Die nächste Version *Java SE 7.0* wurde im Juli 2011 veröffentlicht und enthält neben API-Erweiterungen nur ei-

Java SE 6.0

Java SE 7.0

nige kleinere Spracherweiterungen. 2014 wurde *Java SE 8* veröffentlicht. In dieser Version wurde mit den *Lambda*-Ausdrücken ein sehr interessantes Konzept aus dem Bereich funktionaler Programmiersprachen in die Sprache aufgenommen, das wir ebenfalls kurz vorstellen werden.

Java SE 8

Nach einer dreijährigen Pause wurde 2017 die Version *Java SE 9* freigegeben, mit der u.a. die Java-Shell eingeführt wurde. Die aktuelle Version ist die im März 2020 veröffentlichte Version *Java SE 14*.

Java SE 9

Java SE 14

Geblieben sind die ursprünglichen Ansprüche: eine leicht erlernbare, klare und kompakte Sprache, die die Komplexität beispielsweise von C++ vermeidet, aber dennoch das objektorientierte Paradigma umsetzt. Gleichzeitig ist Java architekturneutral, d.h., durch Verwendung eines plattformunabhängigen Zwischencodes sind Programme auch in kompilierter Form portabel – eine Eigenschaft, die für eine moderne Internet-Programmiersprache fundamental ist. Über die Sprache selbst hinaus spielt die Java Virtual Machine (JVM) eine wichtige Rolle als Plattform für andere Sprachen. Neben Umsetzungen anderer Sprachen auf die JVM wie JRuby oder JPython zählen hierzu insbesondere neue Sprachen wie Scala, Groovy, Clojure oder Kotlin.

1.4 Grundkonzepte der Programmierung in Java

Ein fundamentaler Begriff beim Programmieren – eigentlich beim Umgang mit Computern allgemein – ist das *Programm*. Egal ob man ein Textverarbeitungssystem oder ein kleines, selbst geschriebenes Java-Programm zur Berechnung der Fakultät einer Zahl betrachtet – in beiden Fällen handelt es sich um ein Programm (oder eine Sammlung davon).

Programm

Ein Programm ist die Formulierung eines Algorithmus und der zugehörigen Datenstrukturen in einer Programmiersprache. □

Definition 1.1 *Programm*

Die Grundbausteine von Programmen in höheren Programmiersprachen werden wir erst in den nächsten Kapiteln kennen lernen. Intuitiv wissen wir aber, dass zumindest Anweisungen wie elementare Operationen, bedingte Anweisungen und Schleifen, Datentypen zur Festlegung der Struktur der Daten und darauf definierte Operationen sowie die Möglichkeit der Formulierung von Ausdrücken benötigt werden.

Die syntaktisch korrekte Kombination dieser Bausteine allein führt jedoch noch nicht zu einem ausführbaren Programm. Da Programme meist in einer für Menschen schreib- und lesbaren Form erstellt werden, sind sie für Computer, die nur einen von der jeweiligen Prozessorarchitektur (etwa Intels x86-Familie, die SPARC-Prozessoren von Sun oder die PowerPC-Prozessorfamilie der Macs bzw. IBM-Rechner) abhängigen Maschinencode »verstehen«, nicht direkt ausführbar. Es ist daher eine Übersetzung oder auch Kompilierung (engl. compile) des Programms aus der Programmiersprache in den Maschinencode notwendig. Die Übersetzung erfolgt durch ein spezielles Programm, das als Compiler bezeichnet wird. Neben der Übersetzung in Maschinencode führt ein solcher Compiler auch noch verschiedene Überprüfungen des Programms durch, z.B. ob das Programm hinsichtlich der Syntax korrekt ist, ob benutzte Variablen und Prozeduren deklariert sind u.Ä. Programmiersprachen wie C, C++ oder (Turbo-)Pascal basieren auf diesem Prinzip.

Compiler

Da jede Änderung am Programmtext eine Neukompilierung erfordert, verzichtet man bei einigen Sprachen auf die Übersetzung, indem die Programme interpretiert werden. Das bedeutet, dass ein spezielles Programm – der sogenannte Interpreter – den Programmtext schrittweise analysiert und ausführt. Der Interpreter muss also Syntax und Semantik der Sprache kennen! Der Vorteil dieses Ansatzes ist, dass die Kompilierung entfällt und dadurch Änderungen vereinfacht werden. Auch ist das Programm (man spricht häufig auch von Skripten) unabhängig von einer konkreten Prozessorarchitektur. Diese Vorteile werden durch die Notwendigkeit eines Interpreters und eine durch die interpretierte Ausführung verschlechterte Performance erkauft. Beispiele für Sprachen, die mit diesem Prinzip realisiert werden, sind insbesondere die im Webumfeld beliebten Skriptsprachen wie JavaScript.

Interpreter

Skriptsprachen

Java basiert auf einem kombinierten Ansatz (Abbildung 1–1). Programme werden zwar kompiliert, jedoch nicht in einen prozessorspezifischen Maschinencode, sondern in einen sogenannten Bytecode. Hierbei handelt sich um Code für eine abstrakte Stack-Maschine. Dieser Bytecode wird schließlich von einem Java-Interpreter, der virtuellen Java-Maschine (*Java VM* – engl. Virtual Machine), ausgeführt. Da der Bytecode relativ maschinennah ist, bleibt die Interpretation und Ausführung einfach, wodurch der Performance-Verlust gering gehalten werden kann. Andererseits lassen sich auf diese Weise auch übersetzte Java-Programme auf unterschiedlichen Prozessorarchitekturen ausführen, sofern dort eine Java-VM verfügbar ist.

Bytecode

Aufbau von Java-Programmen

Wie ist nun ein Java-Programm aufgebaut? Ein solches Programm besteht aus einer oder mehreren Klassen, die damit das Hauptstrukturierungsmittel in Java darstellen. Wir werden auf den Begriff der

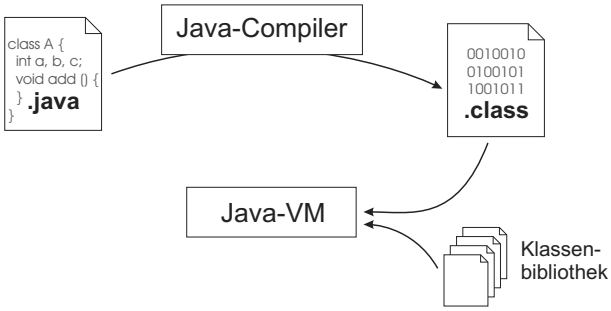

Abb. 1–1
Compiler und Interpreter bei Java

Klasse später im Zusammenhang mit Datenstrukturen noch genauer eingehen. Zunächst wollen wir darunter nur ein Modul verstehen, das eine Menge von Variablen und Prozeduren bzw. Funktionen kapselt. Im Sprachgebrauch von Java werden Prozeduren und Funktionen, d.h. die Zusammenfassung von Anweisungen zu einer logischen Einheit, auch als Methoden bezeichnet.

Klasse

Methoden

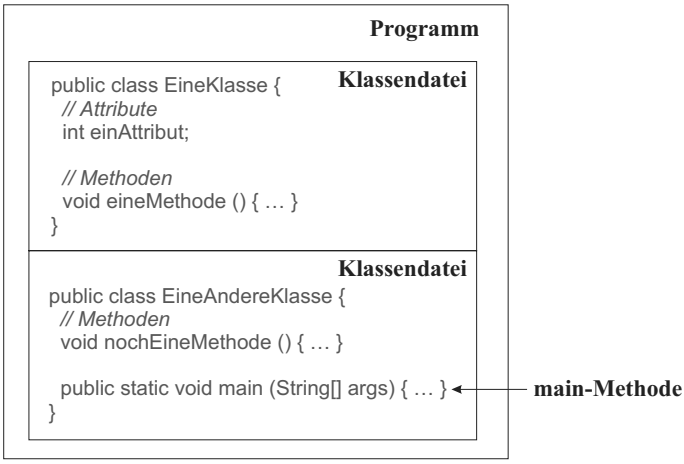

Abb. 1–2
Struktur eines Java-Programms

Abbildung 1–2 verdeutlicht den Aufbau eines Java-Programms aus mehreren Klassen. Jede Klasse ist durch einen Namen gekennzeichnet und kann Attribute und Methoden definieren. Mindestens eine Klasse muss eine spezielle Methode `main` besitzen, die den Einstiegspunkt für die Ausführung bildet. Wenn das Programm später ausgeführt werden soll, ist der Name der Klasse mit der `main`-Methode (engl. für »Haupt«-Methode) anzugeben. Beim Start wird dann zuerst diese Methode aufgerufen, die damit das »Hauptprogramm« bildet.

`main`-*Methode*

Dateien mit Klassendefinitionen werden in Java mit der Endung `.java` gespeichert, wobei für jede öffentliche, d.h. von »außen« nutzbare Klasse eine eigene Datei verwendet wird, die den Namen der

Klasse trägt. Die Klassen müssen zur Erstellung eines ausführbaren Programms zunächst mit dem Java-Compiler in Bytecode übersetzt (kompiliert) werden. Der Java-Compiler ist entweder in eine Entwicklungsumgebung wie Eclipse oder NetBeans integriert oder als ein Programm mit dem Namen `javac` verfügbar. Im ersten Fall erfolgt die Kompilierung einfach über die grafische Benutzerschnittstelle der Entwicklungsumgebung – meist sogar im Hintergrund, ohne dass explizit die Kompilierung angestoßen werden muss. Im zweiten Fall wird dies durch ein Kommando der folgenden Form vorgenommen:

Kompilierung

```
> javac Klasse.java
```

Als Ergebnis der Übersetzung entsteht für jede Klasse eine Datei mit der Endung `.class`, die den Bytecode enthält und durch die Java-Maschine ausgeführt werden kann. Diese Java-Maschine kann ein Interpreter für den Bytecode sein oder auch den Bytecode intern in direkt ausführbaren Maschinencode übersetzen. Letzteres wird als *Just-in-Time-Kompilierung* (*JIT-Kompilierung*) bezeichnet. Ein explizites Binden der übersetzten Klassen, wie es etwa bei C-Programmen durchgeführt werden muss, ist dagegen nicht notwendig. Zur Ausführung muss die Java-Maschine – ein Programm mit dem Namen `java` – gestartet werden, wobei der Name der Klasse als Parameter übergeben wird. Sind in der auszuführenden Klasse weitere Klassen referenziert, so wird der benötigte Bytecode vom Interpreter bei Bedarf nachgeladen. Dazu muss bekannt sein, wo dieser Code zu finden ist. Sowohl Compiler als auch Interpreter werten daher eine Umgebungsvariable `CLASSPATH` aus, die eine Liste von Verzeichnissen bzw. von Archiven mit `.class`-Dateien beinhaltet. Im einfachsten Fall ist dies das aktuelle Verzeichnis, das durch ».« bezeichnet wird. Sollen mehrere Verzeichnisse angegeben werden, so werden diese unter UNIX durch »:« bzw. unter Windows durch »;« getrennt.

Ausführung

`CLASSPATH`

Ein weiteres wichtiges Strukturierungsmittel in Java sind Packages oder Pakete. Hierbei handelt es sich um einen Mechanismus zur Organisation von Java-Klassen. Durch Pakete werden Namensräume gebildet, die Konflikte durch gleiche Bezeichnungen verhindern. So kann eine Klasse `Object` definiert werden, ohne dass es dadurch zu Konflikten mit der Standardklasse `java.lang.Object` kommt. Ein Paket `demo` wird beispielsweise durch die Anweisung

Package

```
package demo;
```

vereinbart, wobei dies am Anfang der Quelldatei anzugeben ist. Jede Klasse, in deren Quelldatei diese Anweisung steht, wird dem Paket `demo` zugeordnet. Über die Notation `demo.MeineKlasse` kann danach

eindeutig die im Paket `demo` definierte Klasse `MeineKlasse` identifiziert werden. Diese Schreibweise lässt sich jedoch abkürzen, indem zu Beginn der Quelldatei eine **`import`**-Anweisung eingefügt wird:

Importieren

```
import demo.MeineKlasse;
```

Danach kann die Klasse `MeineKlasse` ohne vorangestellten Paketnamen verwendet werden. Alle Klassen eines Paketes lassen sich durch Angabe eines $*$ importieren:

```
import demo.*;
```

Paketnamen können hierarchisch aufgebaut werden, wobei die einzelnen Namensbestandteile durch ».« getrennt werden. Allerdings wird durch Angabe des »*« beim Import kein rekursives Importieren durchgeführt, sondern es sind nur die Klassen des bezeichneten Paketes betroffen. Zu beachten ist weiterhin, dass die Pakete, die mit `java.` beginnen, zum Standard-API gehören und nicht verändert werden dürfen, d.h., eigene Paketnamen dürfen nicht mit `java.` beginnen.

Paketnamen

Standard-API

Pakete liefern dem Java-Compiler auch Hinweise, wo die kompilierten Klassendateien (Dateien mit der Endung `.class`) abgelegt werden sollen. Danach werden Punkte im Paketnamen durch den Separator für Verzeichnisnamen ersetzt. Eine Klassendatei der Klasse `algoj.sort.QuickSort` wird demzufolge im Verzeichnis `algoj/sort` unter dem Namen `QuickSort.class` abgelegt.

Betrachten wir nun am Beispiel einer einfachen Klasse die Schritte der Erstellung eines Java-Programms. Die Klasse `Hello` aus Programm 1.1 soll die typische Aufgabe eines jeden ersten Programms aus Einführungsbüchern zu Programmiersprachen erfüllen: die Ausgabe der Meldung »Hello World!«.

Programm 1.1 *»Hello World!« in Java*

```
// Hello World! in Java
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Für das Programm muss zunächst eine Klasse definiert werden, die als »Kapsel« dient: Attribute und Methoden können nur im Rahmen von Klassen definiert werden. Eine Klassendefinition wird durch das Schlüsselwort **`class`**, gefolgt vom Bezeichner der Klasse, eingeleitet. Das im obigen Programm vorangestellte Schlüsselwort **`public`** definiert zusätzlich noch die Sichtbarkeit der Klasse und gibt an, dass diese Klasse öffentlich ist, d.h. auch von anderen Modulen aus zugreifbar ist. Es sei an dieser Stelle noch betont, dass bei Java zwischen

Klassenbezeichner

Groß- und Kleinschreibung unterschieden wird: Die Schlüsselwörter der Sprache (z.B. **class**) werden grundsätzlich kleingeschrieben, bei Bezeichnern (wie etwa der Name einer Klasse) bleibt dies dem Entwickler überlassen, muss aber konsistent erfolgen. So bezeichnen `eineKlasse`, `EineKlasse` und `EINEKLASSE` verschiedene Klassen!

Definition der Attribute und Methoden

Im Anschluss an den Klassenbezeichner folgt in geschweiften Klammern die Definition der Attribute und Methoden. In unserem Beispiel umfasst die Klasse nur eine Methode `main`. Diese Methode ist öffentlich (**public**), liefert kein Ergebnis zurück (Typ **void**) und erwartet als Parameter ein Feld von Zeichenketten (`String[] args`). Obwohl die Parameter in unserem Beispiel nicht benötigt werden, müssen sie dennoch hier angegeben werden, da der Java-Interpreter eine Methode mit exakt der Signatur **static void** `main (String[])` sucht und aufruft. Das Schlüsselwort **static** kennzeichnet die Methode als *Klassenmethode*, d.h., sie kann ohne Objekt aufgerufen werden. Da wir die Konzepte der Objektorientierung erst in Kapitel 11 behandeln, werden wir zunächst alle Methoden als Klassenmethoden realisieren.

Klassenmethode

Ausgabe

Die `main`-Methode besteht nur aus einer Anweisung zur Ausgabe der Zeichenkette »Hello World!« auf den Bildschirm. Diese Anweisung ist der Aufruf der Methode `println`. Das vorangestellte `System.out` bezeichnet das Objekt, für das diese Methode aufgerufen wird – es handelt sich hier konkret um ein Objekt, das durch das Attribut `out` der Klasse `System` bezeichnet wird und die Ausgabe repräsentiert.

Kommentar

Ein Kommentar im Quelltext wird durch das Voranstellen der Zeichenfolge `//` gekennzeichnet. In diesem Fall erstreckt sich der Kommentar bis zum Zeilenende. Eine zweite Möglichkeit ist das Einschließen in */* Kommentar **/*. Hier kann sich der Kommentar auch über mehrere Zeilen erstrecken.

Übersetzung

Der Quelltext der obigen Klasse `Hello` wird in einer Datei `Hello.java` gesichert und mithilfe des Compilers übersetzt:

```
> javac Hello.java
```

Schließlich wird der erzeugte Bytecode mit dem Interpreter ausgeführt und dabei die Meldung »Hello World!« ausgegeben:

```
> java Hello
Hello World!
```

Zu beachten ist, dass nur der Name der auszuführenden Klasse als Parameter angegeben wird und nicht der vollständige Dateiname.

Java-Shell REPL

In der Version 9 wurde mit der *JShell* ein neues interaktives Werkzeug eingeführt. Hierbei handelt es sich um eine REPL (Read Eval

Print Loop), d.h., Java-Deklarationen und -Anweisungen können eingegeben werden und werden sofort ausgeführt. Dies bietet sich gerade für erste Schritte oder einfache Versuche an, da nicht jeweils der Zyklus Editor → Compiler → Interpreter durchlaufen werden muss. Außerdem können auch einzelne Anwendungen direkt ausgeführt werden, ohne dass ein vollständiges Programm zu implementieren ist:

```
> jshell
| Welcome to JShell - Version 14.0.2
| For an introduction type: /help intro

jshell> System.out.println("Hello Shell!");
Hello Shell!

jshell>
```

Applet

Eine spezielle Form von Java-Programmen sind die *Applets*. Das sind kleine Module, die im Rahmen von Webseiten in einem Webbrowser ausgeführt werden. Daraus ergibt sich, dass diese Applets gemeinsam mit dem Dokument vom Webserver geladen werden – das Installieren von Programmen kann damit entfallen. Applets dienen vor allem zur Gestaltung aktiver Elemente in Webseiten. Sie ermöglichen es, Anwendungslogik im Form von Java-Code im Browser auszuführen. So lassen sich beispielsweise Spiele realisieren oder Daten aus anderen Quellen interaktiv präsentieren und manipulieren. Ähnliche Konzepte existieren inzwischen auch für die Ausführung von Java-Programmen auf Mobiltelefonen. Da Applets jedoch aus Sicht der Programmierung keine wesentlichen Unterschiede aufweisen und mit der Version 11 entfernt wurden, werden wir uns im Weiteren auf »normale« Java-Programme beschränken.

```
    return head.getNext() == tail;
  }
}
```

Da für die Klasse `DList` die gleichen Methoden wie für die Klasse `List` aus Abschnitt 13.2 definiert sind, lassen sich die Datenstrukturen Stack und Queue in gleicher Weise unter Benutzung der doppelt verketteten Liste implementieren. Für die Queue ergibt sich daraus der Vorteil, dass auch das Herausnehmen eines Elementes vom Ende der Liste mit konstantem Aufwand möglich ist.

Die Datenstruktur der doppelt verketteten Liste wird manchmal auch als *Deque* für »double-ended queue« bezeichnet. Hierbei handelt es sich um eine Warteschlange, die das Einfügen und Herausnehmen von Elementen an beiden Enden erlaubt.

Deque

13.4 Skip-Listen

Ein Nachteil von verketteten Listen gegenüber Feldern ist, dass die binäre Suche nicht einsetzbar ist, da man nicht beliebig »springen« kann. Eine Alternative ist es, mehrere Listen übereinanderzulegen, die dieses Springen ermöglichen. Eine derartige Datenstruktur wird als *Skip-Liste* bezeichnet.

Abbildung 13–6 zeigt eine Skip-Liste, die direkt die binäre Suche realisiert. Die Ebenen der Skip-Liste sind mit L_i bezeichnet, wobei die Ebene L_0 die eigentliche, sortiert abgespeicherte Liste darstellt. Die Listen der Ebene L_i springen jeweils 2^i Schritte weit und realisieren damit die binäre Aufteilung.

Skip-Liste

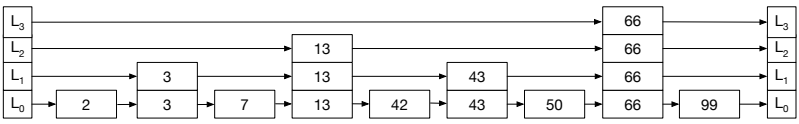

Abb. 13–6
Aufbau einer Skip-Liste

Es gibt jeweils einen initialen und einen terminalen Listenknoten, die die Werte $-\infty$ und $+\infty$ tragen. Grundsätzlich gibt es zwei Möglichkeiten zur Speicherung der Listenelemente auf mehreren Ebenen:

- Der Wert wird in separaten Listenelementen auf allen betroffenen Ebenen abgespeichert.
- Die Listenelemente werden verschmolzen, so dass Listenelemente mehrere Nachfolger haben können (ja nach abgedeckten Ebe-

nen). Auf diese Weise haben wir in der Abbildung diese Elemente schon zusammengeschoben.

In Abbildung 13–7 wird die Suche in einer solchen Liste skizziert. Man startet in der höchsten Ebene mit dem initialen Knoten. Wenn man einen Knoten ansieht, gibt es drei Möglichkeiten:

- Der aktuelle Wert ist der Suchwert. Dann ist man fertig.
- Der aktuelle Wert ist kleiner als der Suchwert. Dann wechselt man zum nächsten Knoten derselben Ebene.
- Der aktuelle Wert ist größer als der Suchwert. Befinden wir uns auf der Ebene L_0, ist der Suchwert nicht gespeichert. Befinden wir uns auf einer höheren Ebene, gehen wir zurück zum letzten Knoten und gehen dann eine Ebene tiefer.

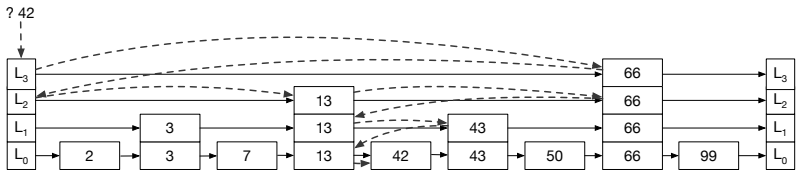

Abb. 13–7
Suche in einer Skip-Liste

Die bisher gezeigte Liste hat eine statische Struktur mit festen Schrittlängen, die durch das Einfügen und Löschen von Werten zerstört werden würde (etwa wenn die Werte 8 und 9 eingefügt werden). Allerdings ist der Algorithmus der Suche davon nicht betroffen: Selbst wenn 8 und 9 auf Ebene L_0 eingefügt würden, würde die Suche weiterhin funktionieren – jedoch bei der Suche nach der 9 mehrere Schritte auf der untersten Ebenen durchlaufen.

Daher reicht es aus, wenn wir nur eine gute Näherung an feste Schrittlängen haben. Eine Möglichkeit ist es dabei, neu eingefügte Werte zufällig einer Ebene zuzuweisen, wobei im Mittel die Hälfte der Werte der Ebene L_0 zugeordnet wird, ein weiteres Viertel der Ebene L_1, ein Achtel dann L_2 etc. Derartige Skip-Listen nennt man randomisiert. Abbildung 13–8 zeigt eine *randomisierte Skip-Liste*. Würde jetzt ein weiterer Wert hinzugefügt, würde er mit Wahrscheinlichkeit $\frac{1}{2^{i+1}}$ der Ebene L_i zugeordnet, wobei die Anzahl der Ebenen sich aus der Gesamtzahl der Werte ergibt.

Randomisierte Skip-Liste

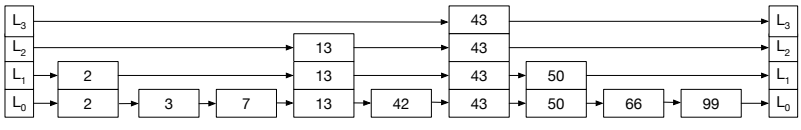

Abb. 13–8
Randomisierte Skip-Liste

Die zuerst eingeführten Skip-Listen mit Zweierpotenzen als festen Schrittlängen werden zur Abgrenzung von randomisierten Skip-Listen als *perfekte Skip-Listen* bezeichnet. Sie können insbesondere genutzt werden, wenn die Werte der Skip-Listen nicht mehr geändert oder ergänzt werden.

Betrachten wir eine einfache Implementierung der Skip-Liste. Zunächst benötigen wir eine Knotenklasse, die neben Vorgänger und Nachfolger auch die Verweise auf die Knoten der darunter liegenden Ebene L_{i-1} (down) und der darüber liegenden Ebene L_{i+1} (up) enthält. Diese Klasse ist in Programm 13.11 die Klasse SLItem, die als Element **int**-Werte enthalten kann. Wir implementieren dabei die Knotenvariante, bei der die Werte auf allen betroffenen Ebenen separat gespeichert sind:

Programm 13.11
Implementierung der Skip-Liste

```
public class SkipList {
  public static int negInf = Integer.MIN_VALUE; // -inf
  public static int posInf = Integer.MAX_VALUE; // +inf

  static class SLItem {
    public SLItem(int i) { element = i; }

    int element; // Element
    SLItem up, down, // Li+1 und Li-1
      prev, next; // Vorgänger, Nachfolger
  }

  SLItem head, tail;
  ...
  public boolean find(int o) {
    SLItem item = head;
    while (true) {
      // zunächst nach rechts suchen ...
      while (item.next.element != posInf &&
        item.next.element <= o)
        item = item.next;
      if (item.down != null)
        // eine Ebene nach unten
        item = item.down;
      else
        // Ebene L0 erreicht
        break;
    }
    return item.element == o;
  }
}
```

Weiterhin benötigen wir noch die `head`- und `tail`-Knoten sowie die speziellen Werte für $-\infty$ und $+\infty$, die wir als kleinsten bzw. größten `int`-Wert repräsentieren.

In Programm 13.11 ist nur die Suche über die Skip-Liste mit der Methode `find` dargestellt. Ausgehend vom `head`-Knoten wird die oberste Ebene zunächst nach rechts durchlaufen, solange der Knotenwert $\leq$ dem gesuchten Wert ist. Erreicht man einen Knoten, dessen Wert größer ist, wird versucht, nach unten zu gehen. Ist dies nicht mehr möglich, weil die L_0-Ebene erreicht ist, wird die Schleife mittels `break` abgebrochen. Da auf jeden Fall am Ende ein Knoten erreicht wird, muss abschließend noch geprüft werden, ob dessen Wert tatsächlich dem gesuchten Wert entspricht.

13.5 Das Iterator-Konzept

Durchwandern einer Kollektion

Die Implementierungen der Listen aus den vorigen Abschnitten weist noch ein Manko auf, welches die praktische Verwendbarkeit einschränkt. So ist es oft notwendig, eine Kollektion zu »durchwandern«, d.h. über alle Elemente zu navigieren. Dieses Navigieren ist zunächst abhängig von der Implementierung: Während beispielsweise ein Feld mittels einer Indexvariablen durchlaufen wird, ist für verkettete Listen das Verfolgen der `next`-Zeiger der einzelnen Knoten notwendig. Auch im Hinblick auf die Erhaltung des Prinzips der Kapselung ist daher ein Konzept wünschenswert, das eine einheitliche Behandlung des Navigierens unabhängig von der internen Realisierung unterstützt. In Java wird dieses Konzept durch *Iteratoren* verwirklicht. Hierbei handelt es sich um Objekte zum Iterieren über Kollektionen, deren Klasse die vordefinierte Schnittstelle `java.util.Iterator` implementiert. Ein Iterator verwaltet einen internen Zeiger auf die aktuelle Position in der zugrunde liegenden Datenstruktur. Auf diese Weise ist es möglich, dass mehrere Iteratoren gleichzeitig auf einer Kollektion arbeiten (Abbildung 13–9).

Iterator

Die Schnittstelle `java.util.Iterator` definiert die folgenden Methoden:

- **`boolean`** `hasNext()` prüft, ob noch weitere Elemente in der Kollektion verfügbar sind. In diesem Fall wird `true` geliefert. Ist dagegen das Ende erreicht, wird `false` zurückgegeben.
- `Object next()` liefert das aktuelle Element zurück und setzt den internen Zeiger des Iterators auf das nächste Element.