→ 5., überarbeitete und erweiterte Auflage

Hanspeter Mössenböck

Sprechen Sie Java?

Eine Einführung in das systematische Programmieren

Bestseller in der 5. Auflage

dpunkt.verlag

Was sind E-Books von dpunkt?

Unsere E-Books sind Publikationen im PDF- oder EPUB-Format, die es Ihnen erlauben, Inhalte am Bildschirm zu lesen, gezielt nach Informationen darin zu suchen und Seiten daraus auszudrucken. Sie benötigen zum Ansehen den Acrobat Reader oder ein anderes adäquates Programm bzw. einen E-Book-Reader.

E-Books können Bücher (oder Teile daraus) sein, die es auch in gedruckter Form gibt (bzw. gab und die inzwischen vergriffen sind). (Einen entsprechenden Hinweis auf eine gedruckte Ausgabe finden Sie auf der entsprechenden E-Book-Seite.)

Es können aber auch Originalpublikationen sein, die es ausschließlich in E-Book-Form gibt. Diese werden mit der gleichen Sorgfalt und in der gleichen Qualität veröffentlicht, die Sie bereits von gedruckten dpunkt.büchern her kennen.

Was darf ich mit dem E-Book tun?

Die Datei ist nicht kopiergeschützt, kann also für den eigenen Bedarf beliebig kopiert werden. Es ist jedoch nicht gestattet, die Datei weiterzugeben oder für andere zugänglich in Netzwerke zu stellen. Sie erwerben also eine Ein-Personen-Nutzungslizenz.

Wenn Sie mehrere Exemplare des gleichen E-Books kaufen, erwerben Sie damit die Lizenz für die entsprechende Anzahl von Nutzern.

Um Missbrauch zu reduzieren, haben wir die PDF-Datei mit einem Wasserzeichen (Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer) versehen.

Bitte beachten Sie, dass die Inhalte der Datei in jedem Fall dem Copyright des Verlages unterliegen.

Wie kann ich E-Books von dpunkt kaufen und bezahlen?

Legen Sie die E-Books in den Warenkorb. (Aus technischen Gründen, können im Warenkorb nur gedruckte Bücher ODER E-Books enthalten sein.)

Downloads und E-Books können sie bei dpunkt per Paypal bezahlen. Wenn Sie noch kein Paypal-Konto haben, können Sie dieses in Minutenschnelle einrichten (den entsprechenden Link erhalten Sie während des Bezahlvorgangs) und so über Ihre Kreditkarte oder per Überweisung bezahlen.

Wie erhalte ich das E-Book von dpunkt?

Sobald der Bestell- und Bezahlvorgang abgeschlossen ist, erhalten Sie an die von Ihnen angegebene Adresse eine Bestätigung von Paypal, sowie von dpunkt eine E-Mail mit den Downloadlinks für die gekauften Dokumente sowie einem Link zu einer PDF-Rechnung für die Bestellung.

Die Links sind zwei Wochen lang gültig. Die Dokumente selbst sind mit Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer als Wasserzeichen versehen.

Wenn es Probleme gibt?

Bitte wenden Sie sich bei Problemen an den dpunkt.verlag:
Frau Karin Riedinger (riedinger (at) dpunkt.de bzw. fon 06221-148350).

Hanspeter Mössenböck ist Professor für Informatik an der Universität Linz. Seine Interessen liegen auf dem Gebiet der Programmiersprachen und der Systemsoftware, insbesondere des Übersetzerbaus. Von 1988 bis 1994 war er Assistenzprofessor an der ETH Zürich und Mitarbeiter von Prof. Niklaus Wirth im Oberon-Projekt. Seit 2000 kooperieren er und sein Institut mit dem Java-Team bei Sun Microsystems bzw. Oracle in Kalifornien. Dabei wurden neue Optimierungen im Java-Compiler und in der Java-VM entwickelt, die heute Teil der Java-Distribution sind. Ferner hatte er Gastprofessuren in Oxford und Budapest inne, wo ihm 2006 ein Ehrendoktorat verliehen wurde.

Mössenböck ist Verfasser der Bücher »Kompaktkurs C#« und »Objektorientierte Programmierung in Oberon-2« sowie Mitverfasser der Bücher »Die .NET-Technologie«, »Ein Compiler-Generator für Mikrocomputer« und »Informatik-Handbuch«.

dpunkt.lehrbuch
Bücher und Teachware für die moderne Informatikausbildung

Berater für die dpunkt.lehrbücher sind:
Prof. Dr. Gerti Kappel, E-Mail: gerti@big.tuwien.ac.at
Prof. Dr. Ralf Steinmetz, E-Mail: Ralf.Steinmetz@kom.tu-darmstadt.de
Prof. Dr. Martina Zitterbart, E-Mail: zit@telematik.informatik.uni-karlsruhe.de

Hanspeter Mössenböck

Sprechen Sie Java?

Eine Einführung in das systematische Programmieren

5., überarbeitete und erweiterte Auflage

dpunkt.verlag

Prof. Dr. Hanspeter Mössenböck
Johannes Kepler Universität Linz
Institut für Systemsoftware
Altenbergerstraße 69
A-4040 Linz
E-Mail: hanspeter.moessenboeck@jku.at
http://ssw.jku.at

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: FrameMaker-Dateien vom Autor
Herstellung: Birgit Bäuerlein
Umschlaggestaltung: Helmut Kraus, Düsseldorf
Druck: Koninklijke Wöhrmann B.V., Zutphen, Niederlande

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.

ISBN 978-3-86490-099-0
ISBN PDF 978-3-86491-493-5

5., überarbeitete und erweiterte Auflage 2014

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1

Vorwort zur 5. Auflage

Lange haben die Java-Entwickler darauf gewartet. Mit Java 8, das 2014 auf den Markt kam, wurden in Java endlich *Lambda-Ausdrücke* eingeführt.

Lambda-Ausdrücke sind namenlose Methoden, die man in Variablen speichern, als Parameter an andere Methoden übergeben und zu einer späteren Zeit ausführen kann. Sie erlauben es, viele Algorithmen der Java-Bibliothek mit Code zu parametrisieren. Ursprünglich stammen Lambda-Ausdrücke aus der Welt der funktionalen Sprachen. Dass sie nun auch in immer mehr imperative Sprachen Eingang finden, zeigt, dass Programmierparadigmen in mancher Hinsicht näher zusammenrücken und bewährte Konzepte zusammenwachsen.

Java 8 bringt noch einige weitere Neuerungen, die sich aber meist nur auf die Java-Klassenbibliothek beziehen oder fortgeschrittene Details betreffen, die den Rahmen dieses einführenden Lehrbuchs sprengen würden. Auf eine dieser Neuerungen wird allerdings dennoch kurz eingegangen, und zwar auf *Default-Methoden* in Interfaces. Um Lambda-Ausdrücke in der Klassenbibliothek einsetzen zu können, mussten einige Interfaces um neue Methoden erweitert werden. Da diese Interfaces jedoch weltweit in Verwendung sind, hätte das bedeutet, dass Millionen von Programmierern diese neuen Methoden implementieren hätten müssen. Java 8 erlaubt daher, Interface-Methoden nun auch mit Code zu versehen. Klassen, die solche Interfaces implementieren, erben diesen Code und müssen die betreffenden Methoden nicht selbst implementieren. Da solche Default-Methoden vor allem in Zusammenhang mit Lambda-Ausdrücken eingeführt wurden, werden sie kurz im Kapitel über Lambda-Ausdrücke beschrieben.

Die neue Auflage dieses Buches enthält auch eine kurze Beschreibung von *anonymen Klassen*, die seit Langem Teil von Java sind, aber als fortgeschrittenes Sprachmerkmal bisher in diesem Buch weggelassen wurden. Da Lambda-Ausdrücke aber in Java mittels anonymer Klassen implementiert werden, gibt es nun auch ein kurzes Kapitel über sie.

Linz, im Jänner 2014
Hanspeter Mössenböck

Vorwort zur 4. Auflage

Nach der Übernahme von Sun Microsystems durch Oracle im Januar 2010 wurde auch die Zukunft der Programmiersprache Java neu geplant. Im Herbst 2011 ist es nun endlich so weit: Die neue Version von Java kommt unter dem Namen Java 7 auf den Markt. Java 7 ist allerdings eher eine Zwischenversion, in der es auf Programmiersprachenebene nur wenige Neuigkeiten gibt. Erst die nächste Version (Java 8), die für Ende 2012 angekündigt ist, soll wieder mehr neue Konzepte bringen.

In Java 7 wurden einige Vereinfachungen eingebaut. So ist es nun zum Beispiel möglich, switch-Anweisungen auf String-Ausdrücke anzuwenden, mehrere Ausnahmetypen in einer einzigen catch-Klausel zusammenzufassen oder die Erzeugung von Objekten generischer Typen mit vereinfachter Syntax zu schreiben. Eine spezielle Form der try-Anweisung erlaubt es außerdem, Ressourcen wie Dateien oder Netzwerkverbindungen, die in der try-Anweisung geöffnet wurden, am Ende der try-Anweisung wieder automatisch zu schließen.

Trotz der geringen Änderungen schien eine Neuauflage dieses Buches angebracht, da damit der aktuelle Stand der Sprache dokumentiert wird und bei dieser Gelegenheit auch kleine Fehler eliminiert werden konnten. Als Neuerung gegenüber der letzten Auflage habe ich mich entschlossen, ein Kapitel über Annotationen einzufügen, also über Metainformationen, die an Programmelemente gehängt und zur Laufzeit ausgewertet werden können. Annotationen gehören zwar nicht zu den Grundlagen der Programmierung, finden aber in letzter Zeit vor allem in Bibliotheken immer stärkere Verbreitung, so dass Java-Programmierer mit ihnen vertraut sein sollten.

Wie immer bedanke ich mich bei den Leserinnen und Lesern für Feedback und Verbesserungsvorschläge, für die ich immer ein offenes Ohr habe.

Linz, im April 2011
Hanspeter Mössenböck

Vorwort zur 3. Auflage

In den letzten Jahren hat sich Java nicht nur in der Industrie immer mehr durchgesetzt, sondern ist auch zur primären Ausbildungssprache an vielen Universitäten und Fachhochschulen geworden.

Seit Sommer 2004 gibt es eine neue Version von Java (Java 5, in älteren Dokumenten auch Java 1.5 genannt), die als wesentliche Neuerungen *generische Typen*, *Enumerationstypen*, eine neue Form der *for-Schleife*, Methoden mit *variabler Parameteranzahl*, *Auto-Boxing* sowie *statisch importierte Klassen* zur Verfügung stellt. Diese Spracherweiterungen machten eine Neuauflage des vorliegenden Buches nötig. Seiner Intention gemäß, ein einführendes Lehrbuch zu sein, werden die neuen Sprachmerkmale jedoch nur so weit beschrieben, als sie für Programmieranfänger von Bedeutung sind. Fortgeschrittene Konzepte wie *Attribute*, die ebenfalls Teil von Java 5 sind, werden nicht behandelt.

Die wichtigsten Neuerungen sind zweifellos generische Typen und Enumerationstypen, denen je ein eigenes Kapitel gewidmet ist. Die anderen Neuerungen wurden in die bestehenden Kapitel eingebaut.

Auf Wunsch vieler Leser wurde ein weiteres Kapitel hinzugefügt, das einen Überblick über die wichtigsten Klassen der Java-Bibliothek gibt. Dort werden Collection-Klassen beschrieben (Listen, Mengen und Abbildungen) sowie Klassen für die Ein- und Ausgabe. Das Kapitel bietet auch einen Einstieg in die Programmierung grafischer Benutzeroberflächen mit *Swing*.

Die für dieses Buch entwickelten einfachen Ein-/Ausgabeklassen `In` und `Out` können wie bisher von *ssw.jku.at/JavaBuch/* heruntergeladen werden. Ferner enthält diese Webseite auch *Musterlösungen* zu den Übungsaufgaben am Ende jedes Kapitels.

Ich danke allen Lesern, die mir Feedback und Verbesserungsvorschläge geschickt haben, und bin auch in Zukunft für Fehlerhinweise und Verbesserungswünsche dankbar.

Linz, im Juli 2005
Hanspeter Mössenböck

Vorwort zur 2. Auflage

Die positive Aufnahme der ersten Auflage dieses Buches als Lehrbuch an zahlreichen Universitäten und Schulen hat zu Verbesserungsvorschlägen geführt, die nach einem ersten Nachdruck eine Neuauflage nahe legten.

In dieser zweiten Auflage wurden nicht nur Fehler korrigiert und Unklarheiten beseitigt, sondern vor allem auch die *objektorientierten Konzepte* von Java stärker betont. So gibt es jetzt ein neues Unterkapitel über *abstrakte Klassen* und eines über *Interfaces*. Auch die dynamische Bindung wurde mit weiteren Beispielen verdeutlicht.

Auf zahlreichen Wunsch gibt es jetzt auch Musterlösungen zu den Übungsaufgaben am Ende der einzelnen Kapitel. Damit aber Dozenten die Aufgaben in ihren Lehrveranstaltungen als Übungsbeispiele austeilen können, werden die Musterlösungen nicht allgemein zugänglich gemacht, sondern Dozenten, aber auch Leser, die das Buch im Selbststudium lesen, können die Musterlösungen beim Verlag (*neumann@dpunkt.de*) anfordern.

Ferner gibt es nun eine Webseite (*www.ssw.uni-linz.ac.at/Misc/JavaBuch/*[1]), auf der man begleitendes Material zu diesem Buch findet. Man kann von dieser Seite nicht nur die Klassen In und Out herunterladen, die für die Ein- und Ausgabe in diesem Buch verwendet werden, sondern man findet auch zahlreiche Verweise auf Java-Systeme, Java-Tutorials und weiterführende Dokumentationen. Insbesondere gibt es auf dieser Seite auch Folien im Powerpoint-Format, die ich für eine Lehrveranstaltung an der Universität Linz entwickelt habe und die sich an den Aufbau dieses Buches halten.

Für Verbesserungsvorschläge und Fehlermeldungen bin ich dankbar und erbitte eine Mitteilung an *moessenboeck@ssw.uni-linz.ac.at*.

Linz, im Januar 2003
Hanspeter Mössenböck

1. Die aktuelle URL lautet *http://ssw.jku.at/JavaBuch/*. Von dort können nun auch die Musterlösungen heruntergeladen werden.

Vorwort zur 1. Auflage

Als ich vor einiger Zeit vor der Aufgabe stand, eine einführende Programmier-Vorlesung mit Java zu halten, stellte ich fest, dass es zwar eine große Zahl von Büchern über Java-Programmierung gab, aber nur wenige, die sich als einführendes Lehrbuch eigneten. Die meisten Java-Bücher beginnen sofort mit Dingen wie Applets für das Internet, mit der Programmierung grafischer Benutzeroberflächen oder zumindest mit objektorientierten Konzepten. Wer noch nie programmiert hat, kann die Beispiele in diesen Büchern zwar nachcodieren und hat auf diese Weise auch Erfolgserlebnisse, er lernt aber nicht systematisch zu programmieren.

Ich ging also daran, ein eigenes Skriptum zu entwerfen, aus dem schließlich dieses Buch entstand. Mein Ziel war es, den Studenten[1] fundamentale Konzepte zu vermitteln, die sie auch in andere Sprachen übertragen konnten. Dazu gehören:

- *Algorithmisches Denken*. Wie formuliert man einen Algorithmus (d.h. ein Problemlösungsverfahren) für eine gegebene Aufgabe? Wie wählt man die richtigen Datenstrukturen und Anweisungsarten dafür? Wie führt man systematische Korrektheitsüberlegungen durch, die einem das Vertrauen geben, dass ein Programm auch wirklich das tut, was es soll?
- *Systematischer Programmentwurf*. Wie zerlegt man eine komplexe Aufgabe systematisch in kleinere Teilaufgaben, die dann als Bausteine (Pakete, Klassen und Methoden) einfach zu implementieren und modular zusammenzusetzen sind?
- *Moderne Softwarekonzepte*. Welche fundamentalen Konzepte gibt es in modernen Programmiersprachen? Dazu gehören zum Beispiel Rekursion, dynamische Datenstrukturen, Datenabstraktion, Vererbung, dynamische Bindung, Ausnahmebehandlung oder Parallelität. Die Beherrschung dieser (sprachunabhängigen) Konzepte zeichnet einen versierten Programmierer aus und gibt ihm einen Werkzeugkasten in die Hand, der ihn zum Meister macht und von Gelegenheitsprogrammierern unterscheidet.

1. Aus Gründen der Kürze und Lesbarkeit wird in diesem Buch nur die männliche Form von Personen verwendet. Selbstverständlich sind damit aber auch alle weiblichen Personen (Studentinnen, Programmiererinnen, Benutzerinnen) gemeint.

- *Programmierstil*. Gute Programme sind nicht nur korrekt, sondern auch elegant, effizient und lesbar. Diese Eigenschaften sind besonders schwierig zu lehren und zu lernen. Andererseits sind sie für die Softwareentwicklung im größeren Umfang essenziell.

Dieses Buch ist keine Sprachspezifikation von Java, sondern ein Programmierlehrbuch, das Java als Werkzeug verwendet. Java ist eine moderne Programmiersprache, die vor allem im Bereich der Web-Programmierung häufig verwendet wird. Ihre Vorzüge machen sie aber auch für alle anderen Bereiche der Programmierung bestens geeignet. Java unterstützt moderne Konzepte der Softwaretechnik wie Sicherheit, Objektorientierung, Parallelität, Ausnahmebehandlung oder Komponentenorientierung. Ihre reichhaltige Bibliothek erlaubt die Erstellung grafischer Benutzeroberflächen, verteilter Anwendungen, Applikationen aus den Bereichen Multimedia, Computergrafik, E-Commerce und vieles andere.

Das vorliegende Buch geht allerdings kaum auf die Java-Bibliothek ein. Die Benutzung dieser Bibliothek ist Katalogwissen, das man jederzeit, auch über das Internet, beziehen kann. Für Programmieranfänger ist die Java-Bibliothek mit ihren Hunderten von Klassen und Tausenden von Methoden sogar eher verwirrend. Wir verwenden sie daher nur dort, wo es unumgänglich ist, nämlich für die Zeichenkettenverarbeitung und für einige mathematische Hilfsfunktionen. Für die Ein-/Ausgabe wurde für dieses Buch eine einfachere Bibliothek in Form der beiden Klassen In und Out entwickelt, die im Anhang A beschrieben wird und die man von [JavaBuch] laden kann.

Dieses Buch ist als Lehrbuch gedacht. Seine Kapitel sollten daher in der angegebenen Reihenfolge gelesen werden. Meist umfasst ein Kapitel genau den Stoff, der in einer Vorlesungseinheit von 90 Minuten bewältigt werden kann. Am Ende jedes Kapitels finden sich Übungsaufgaben, die den gelernten Stoff vertiefen und Lehrveranstaltungsleitern Gelegenheit für praktische Übungen geben.

Ich möchte an dieser Stelle meinen Assistenten Wolfgang Beer, Dietrich Birngruber, Markus Hof, Markus Knasmüller, Christoph Steindl und Albrecht Wöß danken, die die Übungen zu meiner Vorlesung über Jahre hinweg betreut und zahlreiche Übungsaufgaben zu diesem Buch beigesteuert haben. Wolfgang Beer hat außerdem geholfen, etliche Fehler im Manuskript dieses Buches zu entdecken.

Ferner danke ich den vom Verlag eingesetzten Begutachtern Prof. László Böszörményi, Prof. Dominik Gruntz und Prof. Martin Hitz für die zahlreichen nützlichen Anregungen, die sie zu diesem Buch beigetragen haben.

Nun möchte ich Sie als Leser einladen, die spannende Welt des Programmierens zu entdecken. Ich hoffe, dass Ihnen das Programmieren genauso viel Spaß und intellektuelle Befriedigung verschafft, wie das bei mir immer der Fall war.

Linz, im Februar 2001
Hanspeter Mössenböck

Inhaltsverzeichnis

1 Grundlagen

Programmieren ist eine kreative Tätigkeit. Ein Programm ist ein Artefakt wie ein Haus, ein Bild oder eine Maschine, mit dem einzigen Unterschied, dass es immateriell ist. Seine Bausteine sind nicht Holz, Metall oder Farben, sondern Daten und Befehle. Trotzdem kann das Erstellen eines Programms genauso spannend und befriedigend sein, wie das Malen eines Bildes oder das Bauen eines Modellflugzeugs.

Programmieren bedeutet, einen Plan zur Lösung eines Problems zu entwerfen, und zwar so vollständig und detailliert, dass ihn nicht nur ein Mensch, sondern auch ein Computer ausführen kann. Computer sind ziemlich einfältige Geschöpfe, die zwar schnell rechnen, aber nicht denken können. Sie tun nur das, was wir ihnen ausdrücklich sagen und befolgen diese Befehle dafür auf Punkt und Komma genau. Man muss sich als Programmierer also angewöhnen, genau zu sein, an alle Eventualitäten und Fehlerfälle zu denken und nichts dem Zufall zu überlassen.

Die intellektuelle Herausforderung des Programmierens wird oft unterschätzt. Manche Leute bezeichnen das Programmieren abfällig als stumpfe Codierarbeit oder als Routinetätigkeit. Es ist aber alles andere als einfach oder gar langweilig, besonders wenn man elegante und effiziente Programme schreiben will. Programmieren ist eine höchst anspruchsvolle und kreative Tätigkeit, die nur wenige Leute meisterhaft beherrschen. Ein einfaches Programm zu schreiben ist zwar schnell erlernt, so wie jeder Grundschüler schnell Lesen und Schreiben lernt. Gute Software zu erstellen ist aber eher mit der Tätigkeit eines Schriftstellers zu vergleichen. Jeder Mensch kann schreiben, aber nur ganz wenige Menschen können gut schreiben.

Manche Informatik-Studenten fragen sich, wozu sie eigentlich Programmieren lernen sollen. Ihr Berufsziel ist vielleicht das eines IT-Managers oder eines Software-Beraters. Wozu muss man in diesen Berufen programmieren können? Die Antwort ist einfach: Auch wer später nicht selbst programmiert, muss verstehen, wie Software arbeitet. Anders wird er nicht in der Lage sein, Programmiererteams zu leiten und die Qualität von Software zu beurteilen. So wie ein Architekt über Baustoffe und Verfahren Bescheid wissen muss, so muss auch ein Informatiker das Programmieren als sein Grundhandwerk beherrschen. Ehrlich gesagt, macht Programmieren aber auch Spaß, und das ist nicht zuletzt ein wichtiger Grund, warum die meisten Informatiker gerne programmieren.

1.1 Daten und Befehle

Woraus bestehen eigentlich Programme? Jedes Stück Software besteht aus zwei Grundelementen, nämlich aus Daten und Befehlen.

Programm = Daten + Befehle

Die Daten sind jene Elemente, die das Programm verarbeitet. Das können Zahlen, Texte, aber auch Bilder oder Videos sein. Die Befehle sind die Operationen, die mit den Daten ausgeführt werden. Zum Beispiel gibt es Befehle, um Zahlen zu addieren, Texte zu lesen oder Bilder zu drucken.

Computer beherrschen nur einen sehr eingeschränkten Satz von Daten und Befehlen. Aber aus diesen einfachen Grundelementen lassen sich trotzdem fast unbegrenzt komplexe Anwendungen zusammenbauen. Sehen wir uns einmal die Daten und Befehle in einem Rechner genauer an.

Daten. Die Daten werden im Speicher eines Rechners abgelegt. Ein Speicher besteht aus Zellen, die wir uns wie kleine Schachteln vorstellen können. Jede Zelle enthält ein Datenelement, zum Beispiel eine Zahl. Damit wir die Zellen (von denen es Millionen gibt) einzeln ansprechen können, haben sie eine *Adresse*, die wir uns wie einen *Namen* vorstellen können. Ein Speicher besteht also aus benannten Zellen, die Werte enthalten (siehe Abb. 1.1).

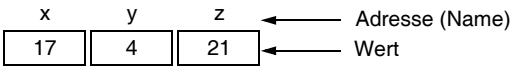

Abb. 1.1 *Speicherzellen*

Die Zelle mit der Adresse `x` enthält in Abb. 1.1 den Wert 17. Die Zelle mit der Adresse `y` enthält den Wert 4. Der Wert einer Zelle kann mit Hilfe von Befehlen geändert werden.

Die Werte in den Speicherzellen sind binär codiert, das heißt, sie bestehen aus Folgen von Nullen und Einsen. Die folgende Tabelle zeigt die Binärdarstellung der ersten 16 natürlichen Zahlen:

0 = 0000	4 = 0100	8 = 1000	12 = 1100
1 = 0001	5 = 0101	9 = 1001	13 = 1101
2 = 0010	6 = 0110	10 = 1010	14 = 1110
3 = 0011	7 = 0111	11 = 1011	15 = 1111

Die Binärdarstellung ist universell, das heißt, es lassen sich damit beliebige Informationen codieren. Neben positiven und negativen Zahlen kann man auch Texte, Bilder, Töne oder Videos binär codieren. Wir gehen hier nicht näher darauf ein; als Programmierer muss man es auch nicht wissen. Das Binärformat ist Sache des Computers. Der Programmierer denkt in höheren Begriffen.

Es sei noch erwähnt, dass eine Binärziffer (0 oder 1) als *Bit* bezeichnet wird. 8 Bits werden zu einem *Byte* zusammengefasst. Je nach Rechnertyp werden 2 oder 4 Bytes als *Wort* bezeichnet und 2 Worte als *Doppelwort*. Ein Rechner arbeitet also intern mit Bits, Bytes, Worten und Doppelworten. Wie wir sehen werden, denkt man als Programmierer aber in anderen Größen, nämlich in Variablen und Objekten.

Befehle. Ein Rechner besitzt eine Hand voll sehr einfacher Befehle, mit denen er die Datenzellen manipulieren kann. Ein einfaches Maschinenprogramm könnte zum Beispiel folgendermaßen aussehen:

ACC ← x	Lade den Wert der Zelle x in ein Rechenregister ACC (Accumulator)
ACC ← ACC + y	Addiere den Wert der Zelle y zu ACC
z ← ACC	Speichere den Wert aus ACC in Zelle z ab (d.h., ersetze den Wert der Zelle z durch den Wert von ACC)

Auch Befehle sind binär codiert, bestehen also aus Nullen und Einsen. Das zeigt, wie universell die Binärcodierung ist. Befehle werden wie Daten im Speicher eines Rechners abgelegt. Das ist bemerkenswert. Ein Programm kann die Befehle eines anderen Programms (ja sogar seine eigenen Befehle) als Daten betrachten. Es kann Programme erzeugen, inspizieren und sogar modifizieren.

Wie bei den Daten gibt es auch bei Befehlen verschiedene Abstraktionsebenen. Ein Programmierer arbeitet nur selten auf der Ebene von Maschinenbefehlen. Er benutzt mächtigere Befehle (so genannte *Anweisungen*) einer Programmiersprache wie *Java*, *C* oder *Pascal*. Die Anweisungen werden aber schlussendlich auf Maschinenbefehle zurückgeführt, denn ein Rechner kann nur Maschinenbefehle verstehen. Die Umsetzung von Anweisungen in Maschinenbefehle wird durch ein Übersetzungsprogramm vorgenommen, das man *Compiler* nennt. Abb. 1.2 zeigt die Schritte, die bei der Entstehung eines Programms von der Spezifikation bis zum Maschinenprogramm durchlaufen werden.

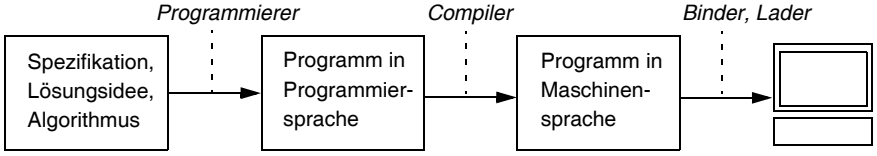

Abb. 1.2 *Entstehungsschritte eines Programms*

Alles beginnt mit einem Problem, das man lösen will, und mit seiner *Spezifikation*, d.h. mit einer genauen Beschreibung dessen, was eigentlich gesucht ist. Aus der Spezifikation ergibt sich eine erste *Lösungsidee* und aus dieser wiederum ein Lösungsverfahren für das Problem, ein so genannter *Algorithmus*. All das sind Vorarbeiten für das Programmieren.

Ein Programmierer erstellt schließlich aus einem oder mehreren Algorithmen ein Programm, das in einer Programmiersprache (z.B. in Java) geschrieben wird. Java-Programme sind zwar für den Menschen gut lesbar, nicht jedoch für einen Rechner, deshalb muss das Java-Programm von einem Compiler in ein Maschinenprogramm übersetzt werden, das aus Nullen und Einsen besteht. Dieses kann nun mit anderen Programmteilen *gebunden*, in den Speicher eines Rechners *geladen* und dort ausgeführt werden.

1.2 Algorithmen

Jedem Programm liegen *Algorithmen* zugrunde. In der Literatur finden sich verschiedene Definitionen dieses Begriffs. Wir verwenden eine sehr einfache, aber für unsere Zwecke völlig ausreichende Definition:

> *Ein Algorithmus ist ein schrittweises, präzises Verfahren zur Lösung eines Problems.*

Algorithmen sind noch keine Programme. Sie können sogar in Umgangssprache beschrieben werden. Ein Kochrezept ist zum Beispiel ein Algorithmus zur Zubereitung eines Gerichts. Eine Wegbeschreibung ist ein Algorithmus, der sagt, wie man von einem Ort zu einem anderen gelangt. In der Informatik stellen wir an Algorithmen jedoch die Forderung, dass sie schrittweise und präzise sein müssen.

Schrittweise bedeutet, dass ein Algorithmus aus einzelnen Schritten besteht, die in genau festgelegter Reihenfolge ausgeführt werden müssen. Dabei darf kein auch noch so nebensächlicher Schritt unerwähnt bleiben. Bedenken Sie: Ein Rechner ist nicht intelligent. Er kann nicht mitdenken.

Aus diesem Grund müssen Algorithmen in der Informatik auch präzise und eindeutig sein. In einem Kochrezept (das für Menschen geschrieben wurde) reicht die Anweisung »gut umrühren«. Für einen Rechner wäre das aber zu wenig präzise. Was heißt »gut«? In der Informatik muss jeder Schritt eines Algorithmus so klar beschrieben sein, dass ein Rechner ihn in eindeutiger Weise ausführen kann.

Nun wird es aber Zeit für ein Beispiel. Nehmen wir an, wir wollen die Summe der natürlichen Zahlen von 1 bis zu einer Obergrenze max berechnen. Ein Algorithmus dafür könnte folgendermaßen aussehen:

Summiere Zahlen (↓max, ↑sum)
1. Setze sum ← 0
2. Setze zahl ← 1
3. Wiederhole Schritt 3, solange zahl ≤ max
   3.1 Setze sum ← sum + zahl
   3.2 Setze zahl ← zahl + 1

Ein Algorithmus besteht aus drei Teilen:

1. Er hat einen *Namen* (Summiere Zahlen), über den man sich auf ihn beziehen kann.
2. Er kann *Eingangswerte* (max) und *Ausgangswerte* (sum) haben. Die Eingangswerte werden von außen (z.B. von einem anderen Algorithmus, dem so genannten *Rufer*) zur Verfügung gestellt und zur Berechnung von Ergebnissen verwendet. Nach Ablauf des Algorithmus kann sich der Rufer die Ergebnisse in den Ausgangswerten abholen. Die Flussrichtung der Parameter deuten wir durch Pfeile an.
3. Ein Algorithmus besteht aus einer Folge von Schritten, die Operationen mit den Eingangswerten und anderen Daten ausführen. In unserem Algorithmus sind die Schritte nummeriert und müssen in der Reihenfolge der Nummern ausgeführt werden. Schritt 3 besteht aus mehreren Teilschritten, die wiederholt ausgeführt werden, bis eine bestimmte Bedingung (hier zahl ≤ max) zutrifft. Dann wird die Wiederholung abgebrochen. Unser Algorithmus würde mit Schritt 4 fortfahren, wenn es ihn gäbe. Da es ihn nicht gibt, ist unser Algorithmus hier zu Ende.

Wir können diesen Algorithmus mit Papier und Bleistift durchspielen und uns für jeden durchlaufenen Schritt die Werte von sum und zahl notieren. Nichts anderes macht ein Computer. Am Ende des Algorithmus steht sein Ergebnis in sum bereit und wird als Ausgangswert an den Benutzer des Algorithmus geliefert.

Programmieren beginnt also damit, dass wir uns für ein gegebenes Problem einen Lösungsalgorithmus überlegen. Aus einem Algorithmus wird ein Programm, indem wir den Algorithmus in einer bestimmten Programmiersprache codieren.

> *Ein Programm ist die Beschreibung eines Algorithmus in einer bestimmten Programmiersprache.*

Ein Algorithmus kann in verschiedenen Programmiersprachen codiert werden, z.B. in Java oder in Pascal. Der Algorithmus ist also das universellere Konstrukt, ein Programm ist nur eine von vielen möglichen Implementierungen davon.

1.3 Variablen

Die Daten eines Programms werden in *Variablen* gespeichert. Eine Variable ist ein benannter »Behälter« für einen Wert. Abb. 1.3 zeigt zwei Variablen x und y mit den Werten 99 und 3.

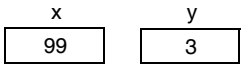

Abb. 1.3 *Variablen als benannte Behälter*

Der Begriff der Variablen ist Ihnen wahrscheinlich aus der Mathematik bekannt. Man sagt, dass die Gleichung

```
x + y = 5
```

zwei Variablen x und y enthält. Aber Vorsicht: In der Mathematik bezeichnen Variablen *Werte*. In der obigen Gleichung stehen x und y für alle Werte, die diese Gleichung erfüllen. In der Informatik hingegen ist eine Variable ein *Behälter* für einen Wert. Einer Variablen x kann man im Laufe eines Programms nacheinander zum Beispiel die Werte 3, 25 und 100 zuweisen. Der Inhalt einer Variablen ist also in der Informatik veränderlich.

Variablen haben nicht nur einen Namen, sondern auch einen *Datentyp*. Der Datentyp legt die Art der Werte fest, die man in einer Variablen speichern kann. In der einen Variablen möchte man zum Beispiel Zahlen speichern, in einer anderen Buchstaben eines Textes. Bildlich kann man sich den Datentyp wie die »Form« des Behälters vorstellen. Da Werte ebenfalls einen Datentyp (und somit eine Form) haben, passen nur jene Werte in eine Variable, deren Typ dem Typ der Variablen entspricht.

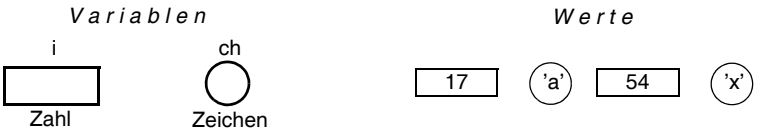

Abb. 1.4 *Variablen und Werte haben einen Datentyp (Form)*

In Abb. 1.4 ist die Variable i ein Behälter für Zahlen, ausgedrückt durch die eckige Form. Es passen z.B. die Werte 17 und 54 hinein, die in unserem Bild ebenfalls eckige Form haben. Die Variable ch ist hingegen ein Behälter für Zeichen, ausgedrückt durch ihre runde Form. Es passen z.B. die Zeichenwerte 'a' und 'x' hinein, die ebenfalls eine runde Form haben.

1.4 Anweisungen

Anweisungen greifen auf Werte von Variablen zu und führen damit die gewünschten Berechnungen durch. Programmiersprachen bieten zwar unterschiedliche Anweisungen an, aber eigentlich gibt es nur sehr wenige Grundmuster, die in den einzelnen Sprachen lediglich abgewandelt werden. Wir sehen uns nun diese Grundmuster an. Dabei verwenden wir keine Programmiersprache, sondern eine grafische Notation, ein so genanntes *Ablaufdiagramm*.

1.4.1 Wertzuweisung

Die häufigste Art einer Anweisung ist eine *Wertzuweisung*. Sie berechnet den Wert eines Ausdrucks und legt ihn in einer Variablen ab. Die Zuweisung

```
y ← x + 1
```

berechnet den Ausdruck x + 1, indem sie den Wert der Variablen x nimmt, 1 dazuzählt und das Ergebnis in der Variablen y abspeichert. Beachten Sie, dass sich der Wert von x dabei nicht ändert. Der alte Wert von y wird hingegen ersetzt durch den Wert des Ausdrucks x + 1. Man liest die Zuweisung als »y wird zu x + 1«.

Auf der linken Seite des Zuweisungssymbols ← muss immer eine Variable stehen, auf der rechten Seite ein Ausdruck aus Variablen oder Konstanten. Folgende Beispiele verdeutlichen das nochmals:

x ← 2	x enthält nun den Wert 2
y ← x + 1	y enthält nun den Wert 3, x hat noch immer den Wert 2
x ← x * 2 + y	x enthält nun den Wert 7 (2 * 2 + 3), y behält den Wert 3 (Der Operator * bedeutet eine Multiplikation)

Folgende Zuweisungen sind hingegen falsch:

3 ← x	auf der linken Seite muss eine Variable stehen (keine Zahl)
x + y ← x + 1	auf der linken Seite muss eine Variable stehen (kein Ausdruck)

1.4.2 Folge (Sequenz)

Man kann mehrere Anweisungen hintereinander schreiben. Sie werden dann in sequenzieller Reihenfolge ausgeführt. Das obige Beispiel zeigte bereits drei in Folge ausgeführte Zuweisungen:

```
x ← 2
y ← x + 1
x ← x * 2 + y
```

Oft deutet man den Programmablauf (den *Steuerfluss*) durch einen Pfeil an, der die Leserichtung vorgibt. Man erhält dadurch ein *Ablaufdiagramm* (siehe Abb. 1.5).

Das Ablaufdiagramm in Abb. 1.5 zeigt noch ein weiteres Beschreibungselement, nämlich eine *Assertion* (Zusicherung). Eine Assertion ist eine Aussage über den Zustand eines Algorithmus oder eines Programms an einer bestimmten Stelle. Sie wird vom Rechner nicht wie eine Zuweisung ausgeführt, sondern dient lediglich als Erläuterung für den Leser. Die Assertion in Abb. 1.5 sagt dem Leser zum Beispiel, dass nach Ausführung der drei Zuweisungen x den Wert 3, y den Wert 4 und z den Wert 12 hat. Die gestrichelte Linie zeigt an, an welcher Stelle die Assertion gilt.

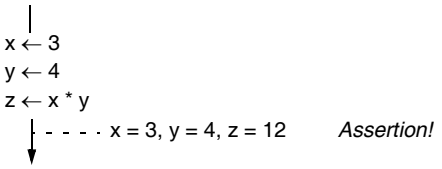

Abb. 1.5 *Ablaufdiagramm einer Sequenz*

1.4.3 Verzweigung (Selektion, Auswahl)

Anweisungen können nicht nur sequenziell hintereinander geschaltet werden, sondern man kann auch ausdrücken, dass eine Anweisung nur unter einer bestimmten Bedingung ausgeführt werden soll. Wir zeigen das wieder anhand eines Ablaufdiagramms (siehe Abb. 1.6):

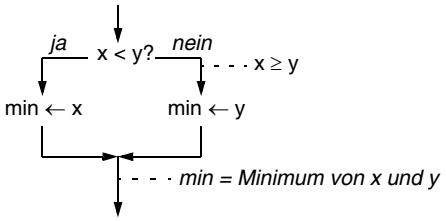

Abb. 1.6 *Ablaufdiagramm einer Verzweigung*

Der Steuerfluss erreicht in Abb. 1.6 zunächst die Abfrage x < y?, die prüft, ob x kleiner ist als y. Das Ergebnis kann »ja« oder »nein« lauten. Je nachdem geht man im Ablaufdiagramm nach links oder nach rechts. Der Steuerfluss verzweigt sich also an dieser Stelle. Ist x < y, werden die Anweisungen im linken Zweig ausgeführt (min ← x), andernfalls die Anweisungen im rechten Zweig (min ← y). Anschließend kommen die beiden Zweige wieder zusammen und der Steuerfluss geht wieder in einem einzigen Zweig weiter.

Was macht der Algorithmus aus Abb. 1.6 eigentlich? Er speichert in der Variablen min das Minimum von x und y, also den kleineren Wert der beiden Variablen. Ist x kleiner als y, geht man nach links und min bekommt den Wert von x, andernfalls geht man nach rechts und min erhält den Wert von y. Die Assertion am Ende des Diagramms gibt nochmals explizit an, was zum Schluss in min gespeichert ist.

Beachten Sie auch die Assertion x ≥ y am Beginn des nein-Zweiges. Da x hier nicht kleiner als y ist, muss gelten, dass es größer oder gleich y ist. Diese Assertion hilft uns beim Verstehen des Algorithmus, denn sie macht sofort deutlich, dass in diesem Zweig y das Minimum der beiden Zahlen ist.

1.4.4 Schleife (Iteration, Wiederholung)

Schleifen erlauben uns auszudrücken, dass eine Folge von Anweisungen mehrmals ausgeführt werden soll, bis eine bestimmte *Abbruchbedingung* eintritt. Abb. 1.7 zeigt, wie eine Schleife als Ablaufdiagramm dargestellt wird.

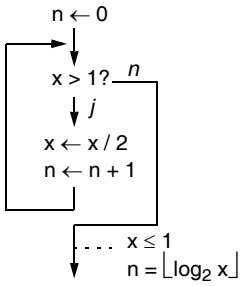

Abb. 1.7 *Ablaufdiagramm einer Schleife*

Nehmen wir an, dass x zu Beginn den Wert 4 enthält. Vor der Schleife wird n auf 0 gesetzt. Die Bedingung x > 1 trifft zu (denn x = 4), also geht man im Diagramm nach unten. x wird ersetzt durch x / 2 (x dividiert durch 2, also 4 / 2 = 2), und n wird zu n + 1, also zu 1.

Nun sehen wir, dass der Pfeil zurückführt und eine Schleife bildet. Wir gelangen wieder zur Abfrage x > 1, die wieder »ja« ergibt, weil x ja nun den Wert 2 hat, also gehen wir im Diagramm wieder nach unten. Die Abfrage und die Anweisungen der Schleife werden also mehrmals durchlaufen. Die Schleife bricht ab, wenn die Bedingung x > 1 nicht mehr zutrifft. In diesem Fall gehen wir im Diagramm nach rechts und verlassen die Schleife. Folgende Tabelle zeigt jeweils die Werte von x und n unmittelbar vor der Abfrage x > 1:

	x	n
1. Besuch	4	0
2. Besuch	2	1
3. Besuch	1	2

Beim dritten Besuch hat x den Wert 1, die Bedingung x > 1 trifft also nicht mehr zu, und die Schleife wird verlassen. n hat den Wert 2. Eine Assertion zeigt den Zustand des Algorithmus am Ende der Schleife. Es gilt hier, dass $x \leq 1$ ist, was sich durch Negation der Schleifenbedingung ergibt. Bei etwas Nachdenken können wir auch angeben, was für n gilt: n ist nämlich der ganzzahlige Logarithmus von x zur Basis 2. Somit erkennen wir auch den Zweck der Schleife: Sie berechnet den ganzzahligen Zweierlogarithmus von x.

Für Schleifen gibt es in Ablaufdiagrammen noch eine andere (kompaktere) Schreibweise (siehe Abb. 1.8). Die Abbruchbedingung wird dabei in ein langgestrecktes Sechseck eingeschlossen. Trifft die Bedingung zu, wird die Schleife betre-

ten. Trifft sie nicht zu, wird sie verlassen, das heißt hinter dem kleinen Kreis an ihrem Ende fortgesetzt. Vom Schleifenende führt ein Weg zurück zur Schleifenbedingung.

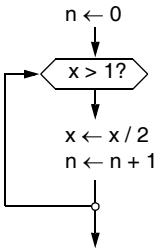

Abb. 1.8 *Schleife als Ablaufdiagramm (andere Darstellungsform)*

Die Schleifen in Abb. 1.7 und Abb. 1.8 sind völlig identisch. Die Schreibweise in Abb. 1.8 ist aber etwas kompakter und drückt besser aus, dass eine Schleife genau 1 Eingang und 1 Ausgang hat.

Schleifen sind für Programmieranfänger sicher die schwierigste Anweisungsart. Man sollte sich ihre Funktionsweise klar machen, indem man einige Schleifen mit konkreten Variablenwerten durchspielt, wie wir das oben getan haben.

Jede Software – von kleinen Beispielprogrammen bis zu komplexen Systemen wie z.B. einer Flugzeugsteuerung – besteht im Wesentlichen nur aus diesen vier Arten von Anweisungen: Zuweisungen, Anweisungsfolgen, Verzweigungen und Schleifen. Natürlich können sie miteinander kombiniert werden (eine Verzweigung kann eine Schleife enthalten, die wieder eine Verzweigung enthält usw.) und natürlich gibt es in den einzelnen Programmiersprachen noch verschiedene Varianten dieser Anweisungsarten. Aber im Wesentlichen haben Sie auf den letzten paar Seiten die Grundelemente des Programmierens kennen gelernt.

1.5 Beispiele für Algorithmen

Wir wollen nun einige Beispiele für Algorithmen betrachten, in denen Zuweisungen, Verzweigungen und Schleifen vorkommen.

1.5.1 Vertauschen zweier Variableninhalte

Gegeben seien zwei Variablen `x` und `y`. Gesucht ist ein Algorithmus, der die Werte der beiden Variablen vertauscht. Wenn `x` also zu Beginn den Wert `3` und `y` den Wert `2` enthält, soll `x` am Ende den Wert `2` und `y` den Wert `3` enthalten. Abb. 1.9 zeigt den Algorithmus als Ablaufdiagramm.

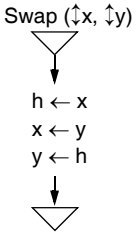

Abb. 1.9 *Algorithmus Swap*

Wir sehen hier ein neues Element eines Ablaufdiagramms: Ein kleines Dreieck deutet den Beginn und das Ende des Algorithmus an. Da es sich in Abb. 1.9 um einen vollständigen Algorithmus handelt, zeigen wir auch seinen Namen (Swap) sowie seine *Parameter*, das heißt die Liste der Eingangs- und Ausgangswerte, die vom Benutzer des Algorithmus übergeben werden und auch wieder an diesen zurückgelangen. Die Doppelpfeile deuten an, dass x und y sowohl Eingangs- als auch Ausgangswerte sind. Sie sind *Übergangswerte*.

Wie funktioniert nun das Vertauschen zweier Variablen. Wir benötigen dazu eine Hilfsvariable h, in die wir den Wert von x retten, so dass x durch den Wert von y ersetzt werden kann. Anschließend speichern wir den geretteten Wert nach y.

Ein *Schreibtischtest* hilft uns, den Algorithmus zu verstehen. Wir legen auf einem Blatt Papier eine kleine Tabelle an, die für jede Variable eine Spalte besitzt. Die erste Zeile der Tabelle füllen wir mit den Anfangswerten der Variablen. Nach jeder Zuweisung an eine Variable tragen wir an das Ende der entsprechenden Spalte den neuen Wert der Variablen ein. Auf diese Weise erhalten wir einen kleinen Papiercomputer, der zwar langsam rechnet, mit dem wir aber jeden Algorithmus durchsimulieren können.

Anfangstabelle

x	y	h
3	2	

h ← x

x	y	h
3	2	3

x ← y

x	y	h
~~3~~ 2	2	3

y ← h

x	y	h
~~3~~ 2	~~2~~ 3	3

1.5.2 Maximum dreier Zahlen berechnen

Gegeben seien drei Zahlen a, b und c. Gesucht ist das Maximum dieser Zahlen, das in der Variablen max gespeichert werden soll. Der Algorithmus ist in Abb. 1.10 dargestellt:

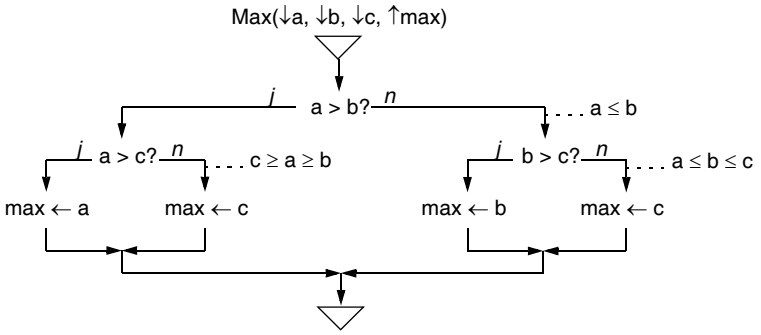

Abb. 1.10 *Algorithmus Max*

Wir sehen hier geschachtelte Verzweigungen. Zuerst wird geprüft, ob a größer als b ist. Wenn ja, wird geprüft, ob a auch größer als c ist. In diesem Fall ist a das Maximum, in anderen Fällen sind weitere Abfragen nötig. Der Algorithmus hat vier Zweige, die alle wieder zusammenkommen und in einem einzigen Zweig an das Ende des Algorithmus fließen.

Wir sehen in Abb. 1.10 auch nochmals den sinnvollen Einsatz von Assertionen. Wenn die Abfrage $a > b$ den Wert »falsch« ergibt, gilt offenbar $a \leq b$. Das ist eine wichtige Erkenntnis und wir schreiben sie als Assertion in das Diagramm. Sie hilft uns, die Logik des Algorithmus zu verstehen. Wenn anschließend auch $b > c$ den Wert »falsch« ergibt, wissen wir, dass $b \leq c$ sein muss. Wir wissen aber auch, dass $a \leq b$ ist, denn das wurde vor Ausführung der inneren Abfrage festgestellt. Daher können wir diese beiden Assertionen kombinieren und erhalten $a \leq b \leq c$. Aus dieser Assertion sehen wir sofort, dass das Maximum der drei Zahlen c ist. Die Assertionen haben uns geholfen, den Algorithmus zu formulieren.

Sie sollten sich angewöhnen, regelmäßig mit Assertionen zu arbeiten. Mit der Zeit wird es für Sie ganz selbstverständlich werden, dass im nein-Zweig einer Abfrage die Negation der Abfragebedingung gilt. Sie werden daher vielleicht eine so einfache Assertion nicht mehr anschreiben, aber Sie sollten sie im Kopf behalten, wenn Sie einen Algorithmus oder ein Programm lesen.

1.5.3 Anzahl der Ziffern einer Zahl bestimmen

Gegeben sei eine positive ganze Zahl n. Gesucht ist die Anzahl ihrer Ziffern. Die Zahl 17 hat z.B. zwei Ziffern, die Zahl 2006 hat vier Ziffern.

Um die Anzahl der Ziffern einer Zahl zu bestimmen, bedienen wir uns eines einfachen Tricks. Wenn wir die Zahl durch 10 dividieren, wird sie um eine Ziffer kürzer. Wir müssen also nur mitzählen, wie oft wir durch 10 dividieren können, bis die Zahl nur noch eine einzige Ziffer enthält, also kleiner als 10 ist. Abb. 1.11 zeigt diesen Algorithmus.

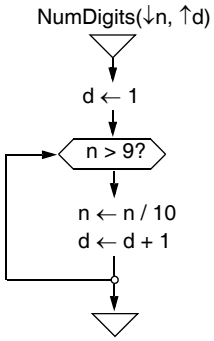

Abb. 1.11 *Algorithmus NumDigits*

Ein Schreibtischtest mit n = 123 zeigt uns, dass der Algorithmus wirklich d = 3 zurückgibt.

n	d
123	1

zu Beginn

n	d
~~123~~	~~1~~
12	2

nach dem ersten Schleifendurchlauf

n	d
~~123~~	~~1~~
~~12~~	~~2~~
1	3

nach dem zweiten Schleifendurchlauf

1.5.4 Größter gemeinsamer Teiler zweier Zahlen

Der folgende Algorithmus ist 2300 Jahre alt. Er wurde vom griechischen Mathematiker *Euklid* um 300 v. Chr. fomuliert. Natürlich verwendete Euklid dazu keine Ablaufdiagramme, sondern eine textuelle Beschreibung. Und natürlich dachte Euklid nicht an eine Umsetzung des Algorithmus in ein Computerprogramm. Aber er definierte das Lösungsverfahren, um für zwei positive ganze Zahlen x und y den größten gemeinsamen Teiler zu berechnen. Dies zeigt, dass Algorithmen »ewige Werte« darstellen können, während Programme oft nur wenige Jahre halten (manchmal nur bis zur nächsten Version der Programmiersprache, in der sie formuliert sind).

Abb. 1.12 zeigt den euklidischen Algorithmus als Ablaufdiagramm. Er berechnet zunächst den Rest der Division von x durch y. Ist dieser Rest 0, so ist y der größte gemeinsame Teiler. Ist der Rest nicht 0, so wird x durch y und y durch den Rest ersetzt. Anschließend wird erneut der Rest der Division x durch y berechnet.