
Oldenbourg

Einführung in die objektorientierte Programmierung mit Java

von
Prof. Dr. Ernst-Erich Doberkat
Dr. Stefan Dißmann
Universität Dortmund

2., überarbeitete Auflage

Oldenbourg Verlag München Wien

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Doberkat, Ernst-Erich:
Einführung in die objektorientierte Programmierung mit Java / von Ernst-Erich Doberkat ; Stefan Dißmann. – 2., überarb. Aufl.. - München ; Wien : Oldenbourg, 2002
ISBN 3-486-25342-5

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg-verlag.de

Lektorat: Irmela Wedler
Herstellung: Rainer Hartl
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: R. Oldenbourg Graphische Betriebe Druckerei GmbH

Inhaltsverzeichnis

Vorwort

Dieses Buch beschäftigt sich mit der Programmierung, und zwar mit der objektorientierten Variante dieser Disziplin der Informatik. Um unseren Lesern das objektorientierte Programmieren näher bringen zu können, benötigen wir eine geeignete Programmiersprache. Hierzu haben wir Java ausgewählt, die populäre Sprache, die häufig im Zusammenhang mit den Schlagworten »Internet«, »WWW-Browser« und »Applet« genannt wird. Diese Themen werden jedoch in dem hier vorliegenden Buch nur eine untergeordnete Rolle spielen, da wir Ihnen über das Vehikel Java die Gestaltung objektorientierter Algorithmen und Systeme nahe bringen wollen. In diesem Buch stehen daher Konzepte im Zentrum der Betrachtung, die konkrete Programmiersprache Java dient als Hilfsmittel zur Verdeutlichung und Formalisierung. Dieser Zugang hat Konsequenzen:

- Die Programmiersprache Java wird nicht in jedem Detail vorgestellt. Wir empfinden dies nicht als Manko, da es vorzügliche Bücher gibt, die die Sprache Java erschöpfend behandeln – und es gibt die sehr umfangreiche Sprachspezifikation von Gosling, Joy und Steele [GJS00], die alle syntaktischen Konstrukte ausführlich erläutert. Dem Leser des vorliegenden Buches raten wir, in Zweifelsfällen eine solche Sprachbeschreibung zu konsultieren.
- Im Vordergrund dieses Buches steht die sequentielle Algorithmik, soweit sie dem Anfänger zugänglich ist. Wichtige Algorithmen werden diskutiert und ihre Realisierungen in Java werden im Detail angegeben. Zu Beginn werden Details schärfer herausgearbeitet, in späteren Kapiteln wird das Niveau der Beschreibungen den wachsenden Kenntnissen der Leser angepaßt. Die klare Ausrichtung auf den Anfänger bedeutet auch, daß die Analyse der Algorithmen im Hinblick auf Laufzeit und Speicherplatzbedarf in den Hintergrund tritt, da meist noch nicht genügend mathematisches Wissen vorhanden ist, um solche Analysen mit Bedacht (und Genuß) durchzuführen.
- Im Verlauf des Buches wird ein Vorrat an wichtigen Datenstrukturen geschaffen – lineare Listen, binäre Bäume, Suchbäume, Heaps, gerichtete und ungerichtete Graphen und Hashtafeln. Wir zeigen auch, wie sich diese Datenstrukturen an konkrete Problemstellungen anpassen lassen. Die Entwicklung geht Hand in Hand mit einer informellen Diskussion abstrakter Datentypen, in deren Verlauf wir beispielsweise zeigen, wie sich aus Heaps Prioritätswarteschlangen gewinnen lassen, zuerst über den ganzen Zahlen, dann über beliebigen geordneten Grundtypen.

Inhalte

Seit dem Erscheinen der ersten Auflage vor vier Jahren hat unser Buch die Grundlage für verschiedene Lehrveranstaltungen gebildet. Mit dem Buch haben in dieser Zeit viele Studierende und einige Dozenten gearbeitet, die uns zahlreiche Anmerkungen und Verbesserungsvorschläge gegeben haben. Aufgrund dieser Anregungen haben wir uns entschlossen, den Aufbau und die Inhalte des Buches für die notwendige Neuauflage gründlich zu überarbeiten. Sichtbarstes Zeichen dieser Überarbeitung ist die um etwa ein Drittel gewachsene Seitenzahl, die durch das Hinzunehmen von zusätzlichen Kapiteln zu den Themenbereichen *Innere Klassen*, *Gestaltung graphischer Benutzungsoberflächen*, *Applets* und *Klassenbibliothek Swing* zustande kam. Obwohl wir uns auch in diesen Kapitel auf die Darstellung der konzeptionellen Aspekte beschränkt haben, hat das Buch damit einen Umfang angenommen, der nach unserer Einschätzung gerade noch in einer vierstündigen Lehrveranstaltung bewältigt werden kann.

Um dem Leser des Buches frühzeitig ein Ziel mitzugeben, haben wir die Problematik der Bestimmung des minimalen Gerüstes zu einem gegebenen Kostengraphen an den Anfang gestellt. Wir skizzieren in der Einleitung zunächst dieses Problem und seine Lösung in Form des Algorithmus von Kruskal, um in den folgenden Kapiteln eine komplexe Aufgabenstellung parat zu haben, anhand derer verschiedene Programmiertechniken motiviert werden können. Allerdings verzichten wir darauf, Java ausschließlich an der Implementierung des Algorithmus von Kruskal einzuführen. Sofern es uns hilfreich oder angemessen erscheint, haben wir auch andere Beispiele eingeflochten. Im Vergleich zur ersten Auflage haben wir auch die Zahl der im Buch vorgeschlagenen Übungsaufgaben deutlich erhöht, um dem Lernenden zusätzliche Ansatzpunkte für das Selbststudium zu bieten.

Das Buch beginnt mit der Schilderung eines Problems und einer informalen Beschreibung seiner Lösung. Diese Lösung, der Algorithmus von Kruskal, benötigt zu seiner Realisierung die Repräsentation eines Kostengraphen sowie von Mengen mit ihren Operationen. In den folgenden acht Kapiteln werden die programmier- und softwaretechnischen Grundlagen erarbeitet, die es uns ermöglichen, in Kapitel 10 eine Implementierung der Lösung vorzunehmen. Diese greifen wir dann in Kapitel 15 noch einmal auf und versehen sie in Teilen mit einer fensterorientierten Benutzungsschnittstelle.

Da unser Anliegen ist in erster Linie die Einführung in das Programmieren mit dem objektorientierten Paradigma und erst in zweiter Linie die Vermittlung der Programmiersprache Java ist, ergibt sich die Folge der Inhalte dieses Buches aus den sprachlichen Hilfsmitteln, die wir zur Bewältigung der in der Einleitung geschilderten Problemlösung benötigen: Wir führen zunächst das Programmieren im Kleinen ein (Kapitel 2) und kommen dann zur Strukturierung von Programmen durch Klassen (Kapitel 3). Rekursion (Kapitel 4) und Abstraktion (Kapitel 5) ermöglichen dann das Konzipieren und Implementieren der dynamischen Daten-

strukturen Liste und Graph (Kapitel 6), die anschließend die Grundlage unserer Implementierungsarbeiten bilden. Die Einführung von Vererbung (Kapitel 7), inneren Klassen (Kapitel 8) und Paketen (Kapitel 9) schafft die Grundlage für die Wiederverwendung der von Klassenbibliotheken vorformulierten Konzepte und für eine softwaretechnisch befriedigende Umsetzung des Algorithmus von Kruskal in Kapitel 10.

Die folgenden drei Kapitel setzen sich mit wesentlichen programmiertechnischen Konzepten auseinander, für die die Sprache Java eine einfach zu handhabende und leicht verständliche Unterstützung bietet: Ausnahmebehandlung (Kapitel 11), nebenläufige Prozesse (Kapitel 12) und Datenströme (Kapitel 13). Am Beispiel von Applets widmen wir uns dann der Implementierung graphischer Benutzungsoberflächen und der Behandlung von Ereignissen (Kapitel 14) und setzen die Beschäftigung mit Benutzungsschnittstellen durch eine Betrachtung der Klassenbibliothek Swing fort (Kapitel 15), die noch einmal die Anwendung fast aller eingeführten Sprachkonzepte erfordert und somit zugleich auch als Zusammenfassung angesehen werden kann. Den Abschluß des Buches bildet die Diskussion der objektorientierten Eigenschaften der Sprache Java (Kapitel 16), die dem Lernenden die Charakteristika dieser Sprachklasse verdeutlichen soll.

Die Programmbeispiele des Buches finden sie auch unter der folgenden URL:

```
http://ls10-www.cs.uni-dortmund.de/java-buch
```

Literaturhinweise

An erster Stelle muß das Buch *The Java Programming Language* [AG00] der Java-Gestalter Ken Arnold und James Gosling genannt werden. Es gibt eine kurz gefaßte Einführung in die Sprache, die ein gutes Gefühl für deren Aufbau vermittelt, ohne zu sehr in programmiersprachlichen Details herumzuwühlen. Allerdings werden beim Leser bereits Erfahrungen mit einer objektorientierten Sprache vorausgesetzt. Die algorithmische Seite kommt an manchen Stellen ein wenig zu kurz.

Das Buch von Campione und Walrath (*The Java Tutorial. Object-Oriented Programming for the Internet*, [CW01]) gibt eine recht vollständige Einführung in die Sprache Java und kann gut als Nachschlagewerk dienen, ist aber nicht als einführendes Buch in die Programmierung geeignet, da es sich ausschließlich auf die Vermittlung der Sprache beschränkt.

[Hen97] gibt zunächst eine knappe Einführung in die Sprache Java und betrachtet dann einige Konstrukte in Java, die im vorliegenden Text nur kurz behandelt werden wie die Gestaltung von Benutzungsoberflächen. Das Buch vernachlässigt etwas die algorithmische Sicht, der Verfasser ist vielmehr darum bemüht, die programmtechnische Seite zu betonen. Zusammen mit einiger Hintergrundliteratur kann das Buch sicher zur Vertiefung der Kenntnisse dienen.

Wenn Sie wirklich alles über Java als Programmiersprache wissen wollen, sollten Sie die Java-Spezifikation [GJS00] bemühen. Die Autoren erklären geduldig jedes einzelne Sprachkonstrukt, als Nachschlagewerk ist es unentbehrlich.

[Mey97] gibt eine Einführung in die objektorientierte Programmierung im Kontext von Eiffel, [Str92] führt in die Sprache C++ ein. Die Syntax von Java orientiert sich stark an den Konstrukten von C++, die dahinter stehenden Konzepte sind jedoch eher an Eiffel angelehnt. Kenntnisse in diesen beiden Programmiersprachen sind daher für das Verständnis von Java durchaus förderlich, können aber praktische Erfahrungen mit Java nicht ersetzen.

Der Klassiker [AHU74] ist eine reiche Sammlung von Algorithmen, an dem wir uns u.a. bei der Behandlung des Algorithmus von Kruskal orientiert haben; das Buch ist für Studierende im Grundstudium nicht einfach zu lesen, Interessenten sollten vielleicht zunächst einen Blick in die *Volksversion* [AHU83] werfen. Ebenfalls empfohlen werden kann das Kompendium [CLR89], in dem man meist mehr findet, als man sucht. Graphalgorithmen werden in [Knu93] ziemlich erschöpfend behandelt, das Buch sei auch wegen seiner didaktischen Qualitäten und seines hinreißenden Stils empfohlen (– das fundamentale Werk *The Art of Computer Programming* [Knu68] gehört ohnehin auf den Schreib- und den Nachttisch eines jeden Informatikers).

Sofern der Leser durch die Programmbeispiele dieses Buches angeregt wird, sich intensiver mit der systematischen Entwicklung (objektorientierter) Systeme auseinander zu setzen, bietet sich hierfür das umfassende Werk [GJM91] an. Hier wird auch ausführlich auf die Theorie der abstrakten Datentypen eingegangen.

Da viele der vorgestellten Algorithmen zur Folklore gehören, haben wir auf Angaben von Quellen im Text weitgehend verzichtet.

Dank

Dr. Doris Schmedding, Prof. Dr. Eberhard Bertsch und Hans-Gerald Sobottka haben die erste Ausgabe des Buches gründlich durchgesehen und wichtige Vorschläge zur Verbesserung gemacht. Christof Veltmann stand bei beiden Ausgaben für zahlreiche Diskussionen zur Verfügung. Astrid Baumgart hat Teile der ersten Auflage editiert, Alla Stankjewitschene hat einige der überarbeiteten Kapitel aus handschriftlichen Aufzeichnungen erstellt. Ihnen allen wollen wir herzlich danken. Unser Dank gilt weiterhin denjenigen Studierenden und Dozenten, die uns mit fundierten Kommentaren zur ersten Auflage versorgt haben.

Ernst-Erich Doberkat und Stefan Dißmann

Kapitel 1
Telephonieren in der Wiener Hofburg

Vielleicht ist die folgende Geschichte apokryph. Sie kann uns jedoch als Einführung in einige Probleme dienen, die wir in diesem Buch mit Hilfe der objektorientierten Softwarekonstruktion mit Java lösen wollen. Im alten Wien, das von Gold umglänzt ist und über das Fritz von Herzmanowski-Orlando [HO97] solche bittersüßen und abstrusen Geschichten erzählt, im alten Wien also, war die Hofburg der Mittelpunkt des politischen und gesellschaftlichen Geschehens. Heerscharen von Bediensteten arbeiteten in der Hofburg, sichtbare und sicher auch unsichtbare, und ein Gewährsmann hat uns von einer Klasse von Bediensteten berichtet, den Hofzwergen, die zwar im Untergrund ihre Arbeit verrichteten, gleichwohl aber durch die Fülle ihrer aus dem Leben gegriffenen Probleme einen dankbaren Untersuchungsgegenstand bilden. Wir werden uns hier zunächst auf ein Telekommunikationsproblem in der Hofburg konzentrieren, werden aber später auch sehen, daß die Hofzwerge eine recht interessante Besoldungsstruktur haben, über deren Modellierung und Implementierung wir uns Gedanken machen werden. Aber zunächst zur Telekommunikation der Hofzwerge in der Hofburg.

1.1 Das Problem

Wir wissen, daß Hofzwerge vielfältige Aufgaben übernehmen mußten: *Hofzwerge erster Klasse* waren u.a. auch dafür zuständig, die Adriatischen Winde aus der Gegend von Venedig einzufangen und in die Klimaanlage der Hofburg zu bringen, *Hofzwerge zweiter Klasse* waren für wichtige Arbeiten im Kanzleibereich im Amt verantwortlich, wozu insbesondere ein Bereitschaftsdienst gehörte, um die Schreibfedern des Sektionschefs stets schreibfähig zu halten. Diese überaus verantwortungsvollen Aufgaben machten zuverlässige Kommunikationsverbindungen zwischen den Hofzwergen notwendig, da sonst die Wiener Hofburg praktisch zum Erliegen gekommen wäre. Wir berichten nun kurz über die Vorgänge zu der Zeit, als die Entscheidung getroffen wurde, Telephonleitungen zwischen den Büros der einzelnen Hofzwerge zu verlegen. Die dazu hier wiedergegebenen Überlegungen sind aber eigentlich irgendwie zeitlos. Sie betreffen die Konstruktion eines interessanten Algorithmus, mit dessen Hilfe ein drängendes praktisches Problem dieser Vernetzung gelöst werden kann.

Nun sind zu der Zeit, über die wir hier berichten, die Telekommunikationsmöglichkeiten noch nicht systematisch in einem Maße vorhanden, wie dies zu Beginn des 21. Jahrhunderts der Fall ist – Begriffe wie *Wireless Network* gehören zu dieser Zeit noch einer fernen, Science-Fiction-umwölkten Zukunft an. Neue Leitungen werden schlicht und einfach verbuddelt oder durch die Gänge und Räume der Hofburg verlegt. Dabei muß Rücksicht auf die vorhandene Bausubstanz genommen werden: So ist ein Verlegen von Leitungen im Ballsaal ebensowenig denkbar wie Grabungsarbeiten auf dem Areal der Hofreitschule. Unsere Hofzwerge sind über die gesamte Hofburg und anliegende Dienstgebäude verteilt, so daß die erforderlichen Leitungen von unterschiedlicher Länge sind und insbesondere auch völlig unterschiedliche Kosten bei der Verlegung verursachen, Kosten, die betrachtenswert sein können und einen wichtigen Faktor für die wirtschaftlichen Planungen darstellen. Das große Problem des Hofkämmerers besteht nun darin, eine möglichst kostengünstige Vernetzung in Auftrag zu geben, die die Konnektivität zwischen den Hofzwergen so sicherstellt, daß jeder Hofzwerg mit jedem anderen – möglicherweise über Zwischenstationen – telephonieren kann.

In einer Vorstudie haben die technischen Betriebe der Hofburg bereits eine grobe Auswahl machbarer Verbindungen ermittelt, die zwischen den Büros von sieben ausgewählten Hofzwergen möglich wären. Diese sind in einer Tabelle so zusammengefaßt worden (Abb. 1.1), daß jeweils zwei miteinander verbundene Hofzwerge angegeben werden, denen die Kosten für das Verlegen einer Leitung zwischen ihnen als Betrag (in Gulden) zugeordnet wird. So könnte etwa der Hofzwerg *Boromeus* direkt mit dem Hofzwerg *Cyriakus* verbunden werden, die Herstellung dieser Verbindung würde dann 28 Gulden kosten. Wenn *Boromeus* mit *Cyriakus* verbunden wäre, so wäre natürlich auch *Cyriakus* mit *Boromeus* verbunden, denn wie wir aus dem täglichen Leben wissen, ist das Telephonieren keine Angelegenheit in Einbahnstraßen: Die Kommunikation verläuft in beide Richtungen.

Leitung	Kosten
Adalbert – Boromeus	23
Adalbert – Florian	20
Adalbert – Gabriel	1
Boromeus – Cyriakus	28
Boromeus – Gabriel	36
Cyriakus – Dorian	17

Leitung	Kosten
Cyriakus – Gabriel	25
Dorian – Emanuel	3
Dorian – Gabriel	16
Emanuel – Florian	15
Emanuel – Gabriel	9
Florian – Gabriel	4

Abb. 1.1: Kosten möglicher Verbindungen zwischen den Hofzwergen

Sie sehen, daß nicht für jeden Hofzwerg mit jedem anderen eine *direkte* Verbindung vorgesehen ist. So fehlt etwa zwischen *Dorian* und *Adalbert* eine unmittelbare Leitung, da sich bereits in der Vorstudie gezeigt hat, daß eine solche Verbindung mit

vertretbarem Aufwand nicht hergestellt werden kann. Das macht überhaupt nichts, denn diese beiden Hofzwerge könnten ja beispielsweise über eine bei *Gabriel* verfügbare Station miteinander telephonieren.

Die Darstellung, die wir in der Tabelle gewählt haben, ist ein wenig umständlich, da wir nicht auf den ersten Blick erfassen können, welche Eigenschaften das Telephonnetz denn nun hat. Hier erweist es sich als nützlich und hilfreich, eine graphische Darstellung einzuführen. Wir malen die einzelnen Hofzwerge als kleine Kreise, in die wir ihre Namen schreiben (– in der Tat ist es ausreichend, die ersten Buchstaben in die Kreise zu zeichnen). Wenn zwei Hofzwerge durch eine direkte Leitung miteinander verbunden sind, so deuten wir das durch eine Linie zwischen den entsprechenden Kreisen an. Die Kosten für die einzelnen Leitungen schreiben wir direkt an die Linie. Damit ergibt sich die in Abb. 1.2 vorgestellte zeichnerische Darstellung unseres kleinen Telephonnetzes. Wir werden in späteren Kapiteln diese graphische Darstellung kurz als *(ungerichteten) Graphen* bezeichnen. Aber damit wollen wir uns im Augenblick noch nicht befassen, uns kommt es vielmehr darauf an, näher auf die Telephonleitungen einzugehen.

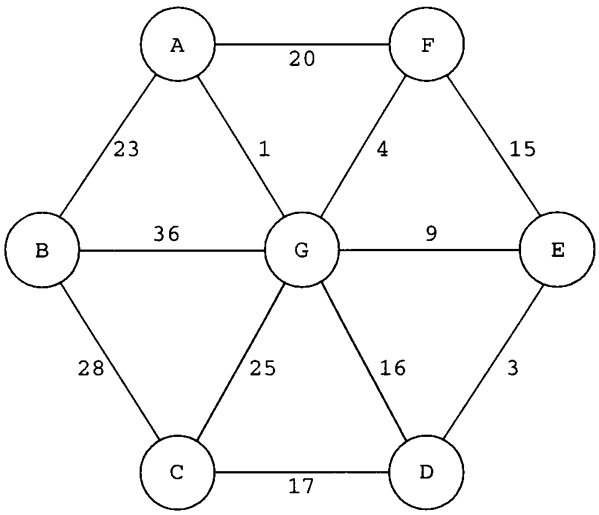

Abb. 1.2: Graphische Darstellung des möglichen Telephonnetzes

Wir sehen der graphischen Veranschaulichung der Ergebnisse der Vorstudie unmittelbar an, daß einige Leitungen entbehrlich sind: Verzichten wir auf das Verlegen einer Leitung zwischen *Boromeus* und *Gabriel*, so ist für das Telephonnetz das Fehlen dieser Leitung nicht so katastrophal. *Boromeus* könnte *Gabriel* immer noch erreichen, er müßte hierfür nur *Adalbert* als Zwischenstation nehmen oder, wenn ihm das nicht möglich sein sollte, *Cyriakus*. Die Konnektivität, also der Zusammenhang dieses Graphen, ist durch das Fehlen dieser Leitung nicht gestört. Wir können weitere

Leitungen entfernen, ohne die Konnektivität zu stören: Es ist z.B. möglich, die Leitung zwischen *Cyriakus* und *Gabriel* zu entfernen. Das übriggebliebene Telephonnetz bliebe zusammenhängend.

Wir können jedoch nicht beliebig viele Leitungen entfernen. Da wir sieben Hofzwerge haben, die die Knoten unseres Graphen bilden, sind mindestens sechs verbindende Leitungen erforderlich, um die Konnektivität des Netzes zu sichern. Und es ist auch klar, daß es nicht sechs beliebige Verbindungen sein können, sondern daß diese Verbindungen vielmehr mit Bedacht gewählt werden müssen. Wenn wir also unser Netz möglichst kostengünstig realisieren wollen, bedeutet das, daß wir solche Leitungen wählen sollten, die in ihrer Gesamtheit den Zusammenhang des dadurch gebildeten Teilnetzes gewährleisten und zugleich bei der Addition ihrer Kosten den die geringsten Gesamtbetrag ergeben. Damit haben wir zwei wichtige Bedingungen für die Auswahl unseres Teilnetzes identifiziert.

Fassen wir also noch einmal kurz zusammen, was wir in dem durch die Vorstudie gegebenen Telephonnetz der Hofzwerge suchen. Wir wollen ein Teilnetz konstruieren (also eine Menge von Verbindungen zwischen den einzelnen Hofzwergen), durch das gewährleistet wird, daß jeder Hofzwerg – möglicherweise über Zwischenstationen – mit jedem anderen telephonieren kann. Davon mag es mehrere geben, und wir suchen ein Teilnetz, das die geringsten Kosten verursacht.

1.2 Die Lösung von Joseph B. Kruskal

Das ist gut und schön: Wie aber finden wir dieses Teilnetz? Wir wollen im Folgenden eine Lösung skizzieren, die auf Joseph B. Kruskal, einen amerikanischen Telephoningenieur, zurückgeht; er hat eine Lösung für dieses Problem im Jahre 1956 in den *Proceedings of the American Mathematical Society* [Kru56] vorgeschlagen. Dieser Vorschlag dient dann als Grundlage für einen wichtigen graphentheoretischen Algorithmus, den wir in Kapitel 10 näher diskutieren werden. Jetzt kommt es uns freilich nur darauf an, Ihnen den Kruskalschen Algorithmus informell nahezubringen, die technischen Details delegieren wir in das spätere Kapitel.

Auf geht´s: Lassen wir unsere Hofzwerge überlegen, was man tun kann. Zunächst ist die Idee einleuchtend, daß die billigste Verbindung, die im Netz der Vorstudie auftritt, im ausgewählten Teilnetz realisiert sein muß. Warum?

Nehmen wir an, wir hätten ein Teilnetz ausgewählt, welches alle Knoten miteinander verbindet, dabei aber die billigste Verbindung nicht berücksichtigt. Ein Beispiel für ein solches Netz, in dem die billigste Verbindung von *Adalbert* zu *Gabriel* (mit Kosten von einem Gulden) nicht enthalten ist, zeigt die Abb. 1.3. Würden wir nun diese Verbindung in das Netz aufnehmen, so könnten wir im Gegenzug auf eine derjenigen Kanten verzichten, die bisher die Konnektivität zwischen *Adalbert* und *Gabriel* über die anderen Knoten des Netzes sicherstellen. In unserem Beispiel

gehören hierzu u.a. die Kanten von *Adalbert* zu *Florian* oder von *Gabriel* zu *Cyriakus*. Jede dieser Kanten muß zwangsläufig teurer sein als die neu eingeführte Kante von *Adalbert* nach *Gabriel*, da diese ja die billigste Verbindung in dem durch die Vorstudie vorgegebenen Netz darstellt. Wir sehen an diesem Widerspruch, daß wir aus ökonomischen Gründen immer ein Teilnetz wählen werden, welches die billigste Verbindung enthält.

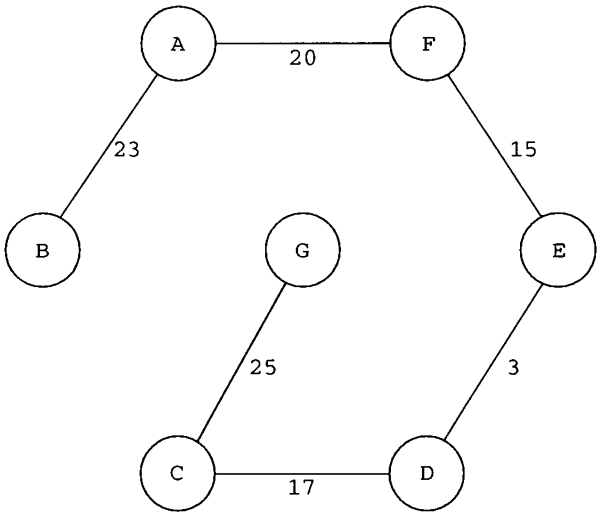

Abb. 1.3: Graphische Darstellung eines beliebigen Telephonnetzes

Die oben angeführte Argumentation gibt uns eine Anregung für die Konstruktion des billigsten Teilnetzes, das unseren Anforderungen entspricht. Wenn die billigste Leitung in diesem Netz enthalten sein muß, dann bietet es sich an, mit dieser Verbindung zu beginnen. Wir betrachten nun wieder das Ergebnis der Vorstudie (Abb. 1.2) und markieren in diesem zunächst die kürzeste Verbindung zwischen *Adalbert* und *Gabriel*, die auf jeden Fall zum Telephonnetz der Hofburg gehören wird.

Was ist mit der nächstgünstigen Verbindung? Das ist der Draht zwischen *Dorian* und *Emanuel*. Sowohl *Dorian* als auch *Emanuel* sollten analog wie *Adalbert* und *Gabriel* natürlich als Verbindungsstationen in unserem Netz vorhanden sein, so daß auch diese zweitbilligste Verbindung mit einem Argument ganz ähnlich zu dem, das wir oben für die billigste Kante angeführt haben, aufgenommen wird. Wir markieren also für spätere Verwendung die Leitung zwischen *Dorian* und *Emanuel* als die zweitbilligste Verbindung. Die nächstgünstige Verbindung besteht zwischen *Florian* und *Gabriel*, und mit der Argumentation, die wir nun schon kennen, nehmen wir auch diese Verbindung auf. Die nächstgünstige Verbindung liegt zwischen den

Hofzwergen *Gabriel* und *Emanuel*, sie wird ebenfalls aufgenommen. Wir können nun unser Zwischenergebnis, das aus einem Teilnetz mit vier Verbindungen zwischen fünf Hofzwergen besteht, in einer Tabelle festhalten:

Leitung	Kosten
Adalbert – Gabriel	1
Dorian – Emanuel	3
Florian – Gabriel	4
Emanuel – Gabriel	9

Die nächste Verbindung, die wir betrachten werden, wird sich als ein wenig tückisch erweisen: Es ist die Kante zwischen *Emanuel* und *Florian*. Würden wir diese Kante in unsere Auswahl aufnehmen, so hätten wir das folgende Problem: Beide Knoten besitzen bereits eine Verbindung zu *Gabriel*. Die direkte Verbindung von *Florian* zu *Emanuel* würde zusätzliche Kosten von 15 Gulden verursachen, während die indirekte Verbindung über *Gabriel* jedoch bereits ohne weitere Kosten verfügbar ist. Wir könnten bei Hinzunahme der direkten Verbindung auch keine teurere Verbindung aus dem bisher markierten Netz entfernen, da alle in diesem enthaltenen Verbindungen aufgrund unseres Vorgehens billiger als die zuletzt hinzugefügte Kante sind. Es bleibt daher nichts anderes übrig, als die Verbindung zwischen *Emanuel* und *Florian* nicht weiter zu berücksichtigen. In analoger Weise verwerfen wir auch die nächstgünstige Verbindung zwischen *Gabriel* und *Dorian*: Hier würde ebenfalls eine Situation entstehen, die unseren Zwecken widersprechen würde.

In den letzten beiden Schritten haben wir also zwei Verbindungen in Betracht gezogen, sie aber aus den genannten Gründen verworfen. Die nächstgünstige Verbindung zwischen *Cyriakus* und *Dorian* akzeptieren wir dagegen wieder, weil zwischen diesen Knoten in dem bisher geschaffenen Teilnetz noch keine Verbindung hergestellt werden kann.

Leitung	Kosten
Adalbert – Gabriel	1
Dorian – Emanuel	3
Florian – Gabriel	4
Emanuel – Gabriel	9
Cyriakus – Daniel	17

Jetzt fehlt uns eigentlich nur noch der Hofzwerg *Boromeus*, der bislang noch nicht die Möglichkeit hat, über ausgewählte Telephonleitungen mit seinen Kollegen zu telephonieren. Die Leitung, die wir für ihn auswählen, verbindet ihn mit *Adalbert*. Es ist die nächstbillige Verbindung. Wir kommen so zu dem in Abb. 1.4 dargestellten Telephonnetz an. Sie sehen unmittelbar, daß jeder Hofzwerg mit jedem anderen tele-

phonieren kann, so daß wir die noch nicht betrachteten Verbindungen zwischen *Cyriakus* und *Gabriel*, zwischen *Boromeus* und *Cyriakus* und schließlich zwischen *Boromeus* und *Gabriel* gar nicht weiter zu betrachten brauchen, denn sie können die gefundene Lösung aufgrund ihrer hohen Kosten nicht verbessern.

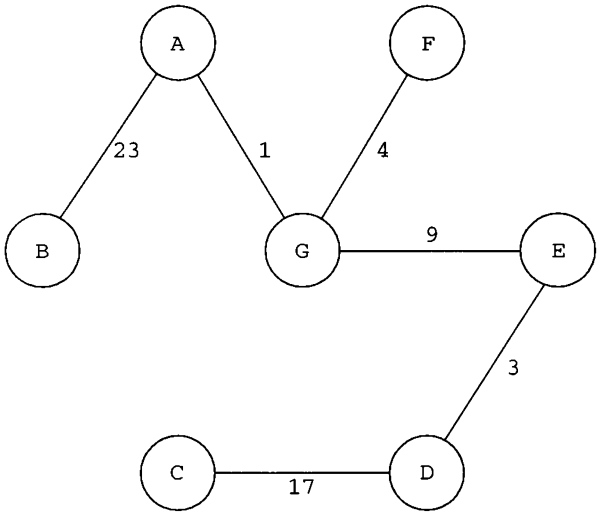

Abb. 1.4: Graphische Darstellung des kostengünstigsten Telephonnetzes

1.3 Überlegungen zum Vorgehen

An der Wiener Hofburg könnte jetzt ein Freudenfeuer angezündet werden, wir aber sollten uns überlegen, was wir eigentlich gemacht haben, um das Problem zu lösen.

Wir haben zunächst aus einer Problemstellung die Vorform eines mathematischen Modells abgeleitet. Die mathematischen Aspekte dieses Modells haben wir noch nicht voll ausformuliert, weil ihre Betrachtung an dieser Stelle nicht nötig ist (und vielleicht sogar ein wenig stören würde). Stattdessen konnten wir eine graphische Repräsentation unseres Modells gewinnen und daran das Problem, mit dem wir umgehen wollten, anschaulich studieren. Das Problem bestand darin, in dem in einer Vorstudie projektierten Telephonnetz eine Menge von Leitungen so zu identifizieren, daß der Zusammenhang aller Knoten bewahrt wird und gleichzeitig die Kosten des Teilnetzes minimal sind. Wir werden in Kapitel 10 eine solche Konstruktion als *minimales Gerüst* bezeichnen. Dieses minimale Gerüst haben wir dann mit Hilfe von Plausibilitätsbetrachtungen schrittweise konstruiert und nebenbei auch

gezeigt, daß es für unser Problem wirklich eine Lösung gibt. Das ist nicht bei jedem Problem trivial und nicht bei jedem Problem unbedingt schon bei der Problemstellung einsichtig.

Wir werden in Kapitel 10 die Lösung dieses Problems als Java-Programm darstellen. Es wird sich zeigen, daß wir die Idee der Vorgehensweise, wie sie hier geschildert wurde (– also die *Heuristik*), übernehmen können, es gleichwohl aber einiger teilweise recht trickreicher Überlegungen bedarf, hierfür ein Programm zu formulieren, das das gegebene Problem für eine beliebige Anzahl von Knoten und Verbindungen allgemein löst. Das Ziel dieses Buches ist es, Sie mit solchen Überlegungen vertraut zu machen und Sie in die Lage zu versetzen, für eine solche, schon recht komplexe Aufgabenstellung selbständig einen Algorithmus zu entwerfen und diesen dann in die Programmiersprache Java umzusetzen.

Um jedoch zu dem in Kapitel 10 in der Programmiersprache Java formulierten Algorithmus von Kruskal vordringen zu können, ist ein gutes Verständnis einer Vielzahl von Konzepten dieser Programmiersprache erforderlich. Die folgenden Kapitel dienen dazu, dieses Verständnis anhand von zahlreichen Beispielen zu vermitteln. Dabei werden uns die Wiener Hofburg und die in ihr arbeitenden Hofzwerge an einigen Stellen wiederbegegnen und uns die Motivation dafür liefern, über ausgewählte Sachverhalte und Problemstellungen intensiver nachzudenken. Wo es uns geeignet scheint, werden wir auch andere Szenarien hinzuziehen und daraus Lernziele für unsere Darstellungen ableiten. Die ersten dieser Ziele werden noch recht einfach sein, um die Bestandteile und Konzepte der Sprache Java zu motivieren, in den später folgenden Kapiteln des Buches wird neben der Sprache Java zunehmend die algorithmische Bewältigung der betrachteten Probleme an Bedeutung gewinnen.

Kapitel 2
Erste Schritte mit Java

Dieses Kapitel behandelt die ersten Schritte in Java. Dazu ist es erforderlich, daß wir uns zunächst Gedanken über die elementaren Grundbausteine der Sprache machen, die in Java *Klassen* genannt werden. Jede Java-Applikation, d.h. ein ausführbares, in Java geschriebenes Programm, besteht aus mindestens einer Klasse. Wir führen also zunächst Klassen ein und zeigen, wie sie im einfachsten Fall gegliedert sind. Klassen dienen dazu, Bestandteile eines Programms strukturell zusammenzufassen und gemeinsam zu beschreiben. Danach stellen wir mit *Variablen* einen wesentlichen Bestandteil von Programmen vor, anhand dessen wir primitive Typen diskutieren. Hierbei beschreiben wir kurz die in Java vordefinierten Typen und zeigen, wie man mit diesen Typen umgeht.

Anschließend befassen wir uns mit elementaren Anweisungen. Hier wird zunächst die Zuweisung behandelt, dann werden elementare Kontrollstrukturen eingeführt, mit denen es möglich ist, die Ausführungsreihenfolge von Anweisungen zu ändern. Schließlich führen wir das Feld als einfache Datenstruktur ein und zeigen, welche Operationen auf Feldern definiert sind. Abschließend diskutieren wir als erstes Beispiel für ein einfaches algorithmisches Problem das *Sortieren durch Einfügen*.

2.1 Klassen – ein erstes Beispiel

Klassen sind die elementaren Grundbausteine von Java-Programmen. Innerhalb von Klassen erfolgt die Beschreibung von (komplexen) Daten und den auf diesen Daten arbeitenden Operationen, die als *Methoden* bezeichnet werden. Zugleich unterstützen Klassen die Strukturierung von Programmen durch die Bereitstellung von Techniken zur Hierarchisierung, Verallgemeinerung und Spezialisierung.

Unser erstes Beispiel bietet die traditionelle Form der Einführung in eine Programmiersprache. Märchen fangen gewöhnlich mit den Worten »*Es war einmal ...*« an, der erste Kontakt zu einer Programmiersprache erfolgt häufig durch ein sogenanntes *Hello-World!-Programm*. Ein solches Programm zeigt, wie mit einfachen Mitteln der Programmiersprache ein sichtbares Ergebnis erzeugt werden kann, die Ausgabe des Textes »`Hello World!`« auf dem Bildschirm.

Da, wie bereits einleitend erwähnt, jede Java-Applikation aus mindestens einer Klasse bestehen muß, vereinbaren wir zur Lösung des *Hello-World!-Problems* eine Klasse mit dem Namen `EinfacheAusgabe`.

Beispiel:

```
class EinfacheAusgabe {
  public static void main(String args[]) {
    EA.println("Hello World!");
  }
}
```

Die Bestandteile dieses Beispiels haben folgende Bedeutungen:

- Das Schlüsselwort `class` zeigt den Beginn der Vereinbarung einer Klasse an, `EinfacheAusgabe` ist der Name der hier vereinbarten Klasse.
- Die Klammerung durch `{...}` wird als *Block* bezeichnet und faßt zusammengehörende Teile innerhalb des Programmiersprachentextes zusammen. Der Block, der unmittelbar hinter dem Namen `EinfacheAusgabe` beginnt, enthält die in dieser Klasse zusammengefaßten Bestandteile.
- `main` ist der Name einer *Methode*. Eine Methode faßt eine Folge von Aktionen zusammen, die unter einem Namen gemeinsam angesprochen und ausgeführt werden. Dem Namen folgt – wie aus der mathematischen Schreibweise von Funktionen bekannt – eine Liste von *formalen Parametern* in der Klammerung mit `(...)`, hier die Angabe eines Parameters `String args[]`.
- Der Name `main` ist besonders ausgezeichnet und kennzeichnet die Methode innerhalb eines Java-Programms, mit der dessen Ausführung begonnen wird. Ein ausführbares Programm wird auch als *(Java-)Applikation* bezeichnet. Um eine Klasse als Applikation ausführbar zu machen, muß daher in der Klasse eine Methode mit dem Namen `main` vereinbart werden. Die Angabe des Parameters `String args[]` ist für die Methode `main` obligatorisch, für das oben gezeigte Beispiel besitzt der Parameter jedoch keine Bedeutung.
- `public`, `static` und `void` sind sogenannte Schlüsselwörter, die Eigenschaften der Methode `main` charakterisieren. Auch diese Schlüsselwörter müssen unbedingt angegeben werden, eine Diskussion ihrer Bedeutung würde jedoch an dieser Stelle zu weit führen. Wir werden in Kapitel 3.3 darauf zurückkommen.
- Die Aktionen, die durch die Methode `main` ausgeführt werden, sind wiederum in einem Block durch `{...}` zusammengefaßt, der hinter der Vereinbarung des Methodenkopfes beginnt. Im Beispiel enthält dieser Block nur eine einzige Zeile Programmtext, die aus genau einer Anweisung besteht.
- `EA.println` ist der Aufruf einer vordefinierten Methode, die den ihr als als *aktuellen Parameter* übergebenen Wert – die Zeichenkette `"Hello World!"` – auf dem Bildschirm ausgibt und anschließend einen Zeilenwechsel durchführt.

- Der Name `EA.println` setzt sich aus zwei Bestandteilen zusammen, die durch ».« voneinander getrennt werden. Der vordere Teil `EA` bestimmt den Ort, an dem die Methode bei der Programmausführung gefunden werden kann. Den hinteren Teil bildet der eigentliche Name der Methode: `println`. Dieses Konstruktionsprinzip wird in Kapitel 3.1 erklärt und dort an weiteren Beispielen verdeutlicht[1].
- Wird die nur aus der Klasse `EinfacheAusgabe` bestehende Applikation ausgeführt, so wird deren einzige Methode `main` aufgerufen, die wiederum die Ausführung der vordefinierten Funktion `EA.println` bewirkt:
  Der Text `»Hello World!«` erscheint auf dem Bildschirm.

In diesem Buch zeigen wir Ihnen sowohl Ausschnitte aus Programmtexten als auch ganze Klassen, sagen Ihnen jedoch nicht im Detail, wie man diese Klassen in ausführbare Programme verwandelt. Hierzu sei auf die entsprechende Dokumentation des von Ihnen verwendeten Compilers[2] verwiesen. Es läßt sich allgemein feststellen, daß eine Java-Klasse in eine Umgebung eingebettet werden muß, die u.a. sagt, wo die verwendeten vordefinierten Klassen oder Methoden zu finden sind. Wir können diesen Aspekt hier nicht vertiefen, er wird in Kapitel 9 ausführlich diskutiert.

2.2 Variable

Eine *Variable* stellt einen Namen bereit, dem innerhalb eines Programms ein Wert zugeordnet werden kann, so daß über den Namen auf den Wert zugegriffen werden kann. Wie die Bezeichnung *Variable* ausdrückt, können die zugeordneten Werte im Verlauf der Programmausführung variieren, d.h. verändert werden. Einer Variablen können daher während der Ausführung eines Programms nacheinander unterschiedliche Werte zugeordnet werden, die über den sich nicht ändernden Namen der Variablen angesprochen werden können. Variablen in Java besitzen einen *Typ*. Das bedeutet:

- Der Typ einer Variablen schränkt den Wertebereich ein, aus dem die möglichen Werte der Variablen ausgewählt werden können. Der Typ ist der Variablen fest zugeordnet und muß vor der Verwendung der Variablen vereinbart werden. Einige Typen sind in Java vordefiniert, weitere können vom Programmierer selbst definiert werden.

---

[1] Das Einlesen von Zahlen über die Tastatur und die Ausgabe von Zahlen und Texten auf dem Bildschirm basiert in Java auf der Nutzung verschiedener Konzepte der Sprache, die an dieser Stelle nicht alle eingeführt werden können. Um die Handhabung für den Anfänger zu erleichtern, greifen wir daher auf die Methoden einer selbst erstellten Klasse `EA` zurück, deren Programmtext im Anhang A angegeben ist. Wird diese Klasse im gleichen Verzeichnis abgelegt wie die Klasse, in der ihre Methoden benutzt werden, kann auf die Methoden von `EA` durch das Voranstellen von `»EA.«` zugegriffen werden.

[2] Alle Implementierungen in diesem Buch wurden mit dem Java-Compiler des Java Development Kit JDK 1.3.1 der Firma SUN entwickelt und unter Windows 2000 getestet.

- Die Namen von Variablen, auch Bezeichner genannt, dürfen in Java aus einer Folge von Buchstaben, Ziffern oder dem Unterstrich »_« bestehen. Namen dürfen nicht mit einer Ziffer beginnen. Groß- und Kleinschreibung von Buchstaben wird unterschieden, auch deutsche Umlaute sind erlaubt.
- Einige Namen sind in der Sprache Java für Schlüsselwörter wie beispielsweise `class` vergeben und können nicht mit einer neuen Bedeutung belegt werden.

Das nachfolgende einfache Beispiel (Bsp. 2.1) zeigt den Einsatz von Variablen. Es werden drei ganze Zahlen eingelesen. Die eingelesenen Werte werden in drei Variablen abgelegt und nochmals ausgegeben. Aus den abgelegten Werten wird dann das Maximum bestimmt und ausgegeben.

```
class MaxAusDrei {                                              //  1
  public static void main (String args[]) {                    //  2
    /* Liest drei Zahlen ein und bestimmt ihr Maximum */        //  3
    int zahl1, zahl2, zahl3, maximum;                           //  4
    EA.println("1. Zahl?"); zahl1 = EA.readInt();              //  5
    EA.println("2. Zahl?"); zahl2 = EA.readInt();              //  6
    EA.println("3. Zahl?"); zahl3 = EA.readInt();              //  7
    EA.println                                                  //  8
      ("gelesene Zahlen: "+zahl1+", "+zahl2+", "+zahl3);        //  9
    maximum = Math.max(zahl1, zahl2);                           // 10
    maximum = Math.max(maximum, zahl3);                         // 11
    EA.println("Maximum: " + maximum);                         // 12
  }                                                             // 13
}                                                               // 14
```

Bsp. 2.1: Die Klasse `MaxAusDrei`

Im folgenden erläutern wir Bsp. 2.1 anhand der Zeilenangaben.

- Kommentare: Die Angabe der Zeilennummern im Beispiel erfolgt als Kommentar am Ende jeder Zeile. Der Text, der auf `//` bis zum Zeilenende folgt, wird bei der Programmausführung ignoriert. Eine weitere Möglichkeit, Kommentare im Programmtext zu kennzeichnen, wird in *Zeile 3* vorgestellt. Die Klammerung mit `/*...*/` könnte sich auch über mehrere Zeilen erstrecken.
- Vereinbarung einer Klasse *(Zeile 1)*: Es wird auf die bereits bekannte Weise eine Klasse mit dem Namen `MaxAusDrei` vereinbart.
- Vereinbarung der Methode `main` *(Zeile 2)*: Die Vereinbarung der Methode `main` erfolgt in der bereits bekannten Form. Der Block `{...}`, der auch als *Rumpf* der Methode bezeichnet wird, enthält den für die Erledigung der geplanten Aufgabe notwendigen Programmtext. Man erkennt am Text des Methodenrumpfes, daß jede einzelne Anweisung durch ein Semikolon abgeschlossen wird.

- Vereinbarung von Variablen *(Zeile 4)*: Es werden vier Variablen deklariert. Der zulässige Wertebereich dieser Variablen umfaßt die ganzen Zahlen und wird durch das Schlüsselwort `int` angezeigt, das den Variablennamen `zahl1`, `zahl2`, `zahl3` und `maximum` vorangestellt wird.
- Einlesen der Werte *(Zeilen 5-7)*: Mit der Ausgabeanweisung `EA.println` werden Abfragen nach drei Zahlen auf dem Bildschirm ausgegeben. Mit der Methode `EA.readInt` werden danach die zugehörigen Zahlenwerte von der Tastatur eingelesen und als Ergebnis des Aufrufs der Methode geliefert. Dieses Ergebnis wird mit dem Zuweisungsoperator »=« jeweils den einzelnen Variablen zugewiesen. `zahl1=EA.readInt()` muß daher gelesen werden als: »*Setze den Wert der Variablen* `zahl1` *auf den Wert, den die Ausführung von* `EA.readInt` *liefert*«.
- Konkatenation von Texten *(Zeile 9)*: Das Zeichen »+« verknüpft die in `"..."` gefaßten Texte und die Werte der Variablen zu einem Text. Dieser Text wird dann der Methode `EA.println` als aktueller Parameter übergeben und von dieser auf dem Bildschirm ausgegeben.
- Vordefinierte Methoden *(Zeilen 10/11)*: `max` ist die vordefinierte Methode, die das Maximum zweier ganzer Zahlen berechnet. `Math` kann hier zunächst als eine Art Ortsangabe aufgefaßt werden, die angibt, wo die Deklaration von `max` erfolgt ist. Kapitel 3.1 wird diese Form der Strukturierung von Programmen näher erläutern. Die Ausführung von `max` bestimmt den größeren Wert aus den beiden als aktuelle Parameter übergebenen Variablen und gibt diesen Wert als Ergebnis zurück. Der Wert wird dann der Variablen `maximum` zugewiesen. Nach der Ausführung von `max` hat daher `maximum` den Wert von `zahl1`, falls gilt `zahl1≥zahl2`, oder den Wert von `zahl2`, falls `zahl1<zahl2`.
- Verwendung von Variablen *(Zeile 11)*: Die Variable `maximum` dient sowohl als Parameter der Methode `max` als auch als Ziel der Zuweisung. Dies führt nicht zu einem Konflikt, da zunächst die Werte der als Parameter übergebenen Variablen in der Methode `max` bearbeitet werden. Dabei wird der größere Wert bestimmt, der dann `maximum` zugewiesen wird.

Das Beispiel zeigt, daß die Zuweisung an eine Variable von der rechten zur linken Seite des Zuweisungsoperators »=« vorgenommen wird. Der Wertfluß der Zuweisung folgt damit in Java – wie auch in vielen anderen Programmiersprachen – nicht dem Lesefluß. Wir werden Zuweisungen in Kapitel 2.6.3 näher betrachten.

Die Vereinbarung *(Deklaration)* von Variablen wird in Bsp. 2.1 am Beginn des Methodenrumpfes vorgenommen. Dies ist notwendig, um den Namen einer Variablen innerhalb des Blocks, der den Rumpf bildet, bekannt zu machen. Die Verwendung einer Variablen ist nur dann zulässig, wenn zuvor ihre Deklaration erfolgt ist. Werden, wie im Beispiel, mehrere Variablen gleichen Typs gemeinsam definiert, so können die Namen durch Kommata getrennt werden. Auch eine solche gemeinsame Deklaration wird durch ein Semikolon abgeschlossen. Es ist aber ebenfalls erlaubt, mehrere unabhängige Deklarationen für den gleichen Typ vorzunehmen.

Beispiele für Deklarationen:

```
int anzahl, nummer;
float durchmesser;
int sitzplatz;
```

Es werden drei Variablen des bereits bekannten Typs `int` mit den Namen `anzahl`, `nummer` und `sitzplatz` sowie eine Variable des Typs `float` vereinbart, in der Werte aus einem Teilbereich der reellen Zahlen abgelegt werden können.

Technisch geschieht bei einer Deklaration folgendes: Für die deklarierte Variable wird der für die Werte des entsprechenden Typs benötigte Speicherplatz reserviert. Dieser Speicherplatz, dessen Inhalt änderbar ist, wird an den Namen der Variablen gebunden. Über den Namen kann dann der auf den Speicherplatz zugegriffen werden und der dort abgelegte Wert gesetzt oder abgerufen werden.

2.3 Primitive Datentypen

Java stellt ganze Zahlen, reelle Zahlen, die Booleschen Werte und einzelne Zeichen als vordefinierte primitive Datentypen zur Verfügung. Wir wollen in diesem Abschnitt diese Typen kurz in einer Art tabellarischer Auflistung diskutieren, wobei wir zunächst die Typnamen und die zulässigen Wertebereiche angeben, dann die Operationen aufführen, die auf diesen Typen vordefiniert sind, und diese – wo es nötig erscheint – erläutern. Da es sich hier um mehr oder minder kanonische Angaben handelt, belassen wir es im wesentlichen bei einer Aufzählung der entsprechenden Eigenschaften.

Es sei noch angemerkt, daß wir die Datentypen `float` und `double`, die in Java Teilbereiche der reellen Zahlen repräsentieren, hier nur sehr kurz behandeln werden. Im vorliegenden Buch werden überwiegend Algorithmen und Konzepte betrachtet, die sich mit der Organisation und Strukturierung von Programmtexten und den damit bearbeiteten Daten beschäftigen. Es tauchen dabei keine Fragestellungen auf, deren Lösungen numerische Berechnungen mit reellen Zahlen erfordern würden.

Java definiert vier primitive Datentypen, die jeweils einen Ausschnitt aus den ganzen Zahlen repräsentieren. Diese Datentypen unterscheiden sich durch die Wertebereiche, aus denen die Werte gewählt werden können, die in den für diese Typen deklarierten Variablen abgelegt werden können. Werte eines größeren Wertebereichs haben einen erhöhten Platzbedarf, woraus sich unmittelbar ergibt, daß die Variablen dieser Typen sich durch die Größe des für sie zu reservierenden Speicherplatzes unterscheiden. Der Programmierer muß für seine Problemstellung entscheiden, welche Werte die Daten annehmen können und welcher Datentyp daher für eine Problemlösung angemessen erscheint.

- Datentypen für ganze Zahlen

  Datentyp `byte` mit dem Wertebereich: $-2^7 \ldots 0 \ldots +2^7-1$

  Datentyp `short` mit dem Wertebereich: $-2^{15} \ldots 0 \ldots +2^{15}-1$

  Datentyp `int` mit dem Wertebereich: $-2^{31} \ldots 0 \ldots +2^{31}-1$

  Datentyp `long` mit dem Wertebereich: $-2^{63} \ldots 0 \ldots +2^{63}-1$

  Operationen: `+, -, *, /, %`

  Für zwei Variablen `a` und `b`, die einen dieser vier ganzzahligen Datentypen besitzen, stellt für `b` $\neq$ `0` der Ausdruck `a/b` den ganzzahligen Anteil der Division von `a` durch `b` dar, der Rest der Division wird durch `a%b` (*modulo*-Operator) berechnet und ist definiert durch die Berechnung `a-b*(a/b)`.

  Vergleiche: `==, !=, >, >=, <, <=`

  Hierbei führt der Operator `==` einen Test auf Gleichheit durch, `!=` einen Test auf Ungleichheit. Das einzelne Zeichen »=« kennzeichnet – wie bereits bekannt – die Zuweisung und kann nicht für Vergleiche eingesetzt werden.

  vordefinierte Methoden: `Math.min, Math.max, Math.abs`[1]

  Konstante: Ganze Zahlen werden in dezimaler Darstellung notiert, ein Vorzeichen ist erlaubt. Ganzzahlige Konstanten haben immer den Typ `int`, sofern nicht `L` oder `l` angehängt ist, das sie dann explizit dem Typ `long` zuordnet. Andere Zahlensysteme sind möglich, sollen hier aber nicht behandelt werden. Zusätzlich stehen die symbolischen, d.h. durch Namen bezeichneten, Konstanten `MAX_VALUE` und `MIN_VALUE` zur Verfügung. Diese bezeichnen den größten bzw. kleinsten Wert der Datentypen `int` und `long`. Der Datentyp wird durch eine Angabe vor der Konstanten bestimmt, wobei die Namen `Integer` und `Long` verwendet werden müssen. `Integer.MAX_VALUE` liefert also den größten Zahlenwert, der in Variablen des Typs `int` abgelegt werden kann.

  Das Bsp. 2.2 zeigt die Berechnung eines einfachen arithmetischen Ausdrucks und den Umgang mit Variablen und Konstanten des Typs `int`. Der Wert für `i` wird auf die bereits bekannte Art eingelesen; dann wird das Minimum der aus dem Quadrat von `i` (`i*i`) und dem Zehnfachen von `i` (`10*i`) bestimmt, der Wert der Variablen `j` zugewiesen und ausgegeben. Anschließend wird mit dem *modulo*-Operator der Rest der Division von `j` durch `2` bestimmt und aufgrund des so berechneten Ergebnisses (`0` oder `1`) ausgegeben, ob `j` eine gerade Zahl ist.

---

[1] Da Java Groß- und Kleinschreibung unterscheidet, ist die hier angegebene Schreibweise verbindlich.

```
class IntAusdruck {
  public static void main(String args[]) {
    int i, j, k;
    EA.println("Zahl?"); i = EA.readInt();
    j = Math.min(i*i, 10*i);
    EA.println("Wert von j ist: " + j);
    k = 1-j%2;
    EA.println("Wert von j ist gerade, wenn Ausgabe > 0: " + k);
  }
}
```

Bsp. 2.2: Berechnung arithmetischer Ausdrücke

- Datentypen für reelle Zahlen

  Datentyp `float` mit dem Wertebereich: *etwa* $-3.4 < 10^{38} \ldots +3.4 < 10^{38}$

  Datentyp `double` mit dem Wertebereich: *etwa* $-1.8 < 10^{308} \ldots +1.8 < 10^{308}$

  Operationen: `+, -, *, /`

  Vergleiche: `==, !=, >, >=, <, <=`

  vordefinierte Methoden: `min, max, abs, sin, cos, tan, exp, log, sqrt, pow` und weitere. Alle aufgeführten Methoden müssen mit der vorangestellten Angabe `Math` aufgerufen werden.

  Konstante: Reelle Zahlen werden als Fließkommazahlen in dezimaler Darstellung notiert, ein Vorzeichen ist erlaubt. Fließkommakonstanten haben immer den Typ `double`, sofern nicht ein `f` oder `F` angehängt ist, das sie dem Typ `float` zuordnet. Der Typ `double` kann durch `d` oder `D` gekennzeichnet werden. Ein Exponent wird durch `e` oder `E` eingeleitet.

  Für reelle Zahlen sind die symbolischen Konstanten `MAX_VALUE` und `MIN_VALUE` sowie `NEGATIVE_INFINITY` und `POSITIVE_INFINITY` verfügbar. Der Wertebereich, auf den sich die Konstante bezieht, wird durch die vorangestellte Angabe von `Float` oder `Double` wie bei `Float.MIN_VALUE` festgelegt. Zusätzlich repräsentieren die Konstanten `Float.NaN` und `Double.NaN`[1] jeweils einen Wert, der nicht innerhalb des zulässigen Wertebereichs liegt und von den verschiedenen arithmetischen Operationen nur dann erzeugt wird, wenn das Ergebnis ihrer Auswertung undefiniert ist. Das Ergebnis `NaN` wird beispielsweise erzeugt durch den Ausdruck `Float.NEGATIVE_INFINITY/Float.POSITIVE_INFINITY`.

[1] `NaN` steht für *not a number.*

Es gibt zwei Konstanten, die ausgezeichnete Werte des Typs `double` repräsentieren: `Math.E` (= 2,71...) und `Math.PI` (= 3.14...).

Beispiele: `0.345, .32, 46d, -45.4e8, 7e-3f, Math.PI`

Eine Fließkommakonstante muß immer durch einen ihrer Bestandteile von einer entsprechenden ganzzahligen Konstante unterschieden werden. Beispielsweise wird die Zahl `47` immer dem Typ `int` zugeordnet, hingegen werden `47d`, `47e1` oder `47.0` dem Typ `double` zugeordnet.

- Datentyp für Wahrheitswerte: `boolean`

  Wertebereich und Konstanten: `true, false`

  Operationen: `!` (Negation), `&` (Konjunktion), `|` (Disjunktion), `^` (exklusive Disjunktion), `&&` bzw. `||` (Konjunktion bzw. Disjunktion, bei der der rechte Operand nur dann ausgewertet wird, wenn das Ergebnis nicht schon durch die Auswertung des linken Operanden festliegt).

  Durch die *shortcut*-Operatoren `&&` bzw. `||` können Ausdrücke formuliert werden, die bei vollständiger Auswertung Fehler erzeugen könnten. Während beispielsweise die Auswertung des Ausdrucks `(b!=0)&((a/b)==5)` eine Division durch `0` und damit einen Fehler bewirken könnte, wird die Division durch `0` bei dem Ausdruck `(b!=0)&&((a/b)==5)` vermieden, da `a/b` für den kritischen Fall `b==0` gar nicht ausgewertet wird.

  Vergleiche: `==, !=`

- Datentyp für einzelne Zeichen: `char`

  Wertebereich: Unicode-Zeichensatz[1]

  Vergleiche: `==, !=, >, >=, <, <=`

  vordefinierte Methoden: `Character.toLowerCase, Character.toUpperCase` und weitere.

  Konstante: Ein Zeichen wird in einfache Hochkommata `'...'` eingeschlossen, beispielsweise als `'a'` oder `'A'`.

---

[1] Der Unicode-Zeichensatz umfaßt Ziffern, Buchstaben, Satz- und Sonderzeichen und ist geordnet, so daß jedes Zeichen eine feste Position einnimmt. Einige Anmerkungen zum Unicode-Zeichensatz finden sich im Anhang auf Seite 425.

2.4 Typanpassung

In dem vorangehenden Abschnitt sind die Typen `byte`, `short`, `int` und `long` vorgestellt worden, die in Java Teilmengen der ganzen Zahlen repräsentieren. Damit kann innerhalb eines Programms die Situation eintreten, daß beispielsweise die ganzzahligen Werte zweier Variablen mit den unterschiedlichen Typen `short` und `long` miteinander verglichen werden sollen. Java erlaubt diesen auch mathematisch unproblematischen Vergleich und paßt dabei automatisch den Wert des bereichsmäßig kleineren Typs `short` an den Typ `long` an. Auch die Zuweisung eines Werts eines Typs mit einem kleineren zu einem Typ mit größerem Wertebereich ist zulässig.

Bei diesen Konvertierungen können keine Probleme auftreten, da der Wertebereich des Zieltyps immer den gesamten Wertebereich des Ausgangstyps umfaßt. Java bezieht auch die Fließkommatypen `float` und `double` mit in die Konvertierungsregel ein, so daß die Konvertierungen von `long` nach `float` und von `float` nach `double` erlaubt sind, wobei jedoch Rundungsfehler auftreten können. Darüber hinaus kann der Typ `char` in den Typ `int` konvertiert werden. Die dann erzeugte Zahl entspricht der Position des konvertierten Zeichens im Unicode-Zeichensatz.

Beispiel:

```
byte b; int i; float f;                 // 1
f = i = b = 5;                          // 2
f = f + i + 'a';                        // 3
```

In *Zeile 2* wird den ganzzahligen Variablen `b` und `i` die Konstante `5` zugewiesen. Die Variable `f` würde bei einer Ausgabe `5.0` ergeben. Die Addition in *Zeile 3* ist zulässig, da das Zeichen `'a'` des Typs `char` implizit in den Wert `97` des Typs `int` konvertiert wird – die Position von `'a'` im Unicode-Zeichensatz. Die Variable `f` erhält daher den Wert `107.0`.

Bei der Umwandlung von einem Wert eines Typs mit größerem Wertebereich in einen Typ mit kleinerem Wertebereich kann hingegen nicht sichergestellt werden, daß der konvertierte Wert in den Zielbereich paßt. Solche Umwandlungen werden daher nicht automatisch vorgenommen, sondern müssen durch den Aufruf eines *Typanpassungsoperators* explizit erzwungen werden. Der Anpassungsoperator besteht für jeden Zieltyp aus seinem in Klammern eingeschlossenen Namen, der dem Wert oder der Variablen des Ausgangstyps vorangestellt wird.

Beispiel:

```
byte b; int i; char c;                  // 1
i = 5;                                  // 2
b = (byte)i;                            // 3
c = (char)100;                          // 4
i = (int)5.0E9;                         // 5
```

In *Zeile 3* wird der Wert der Variablen `i` in den Typ `byte` umgewandelt und der Variablen `b` zugewiesen. In *Zeile 4* erhält die Variable `c` das Zeichen `'d'` zugewiesen, das die Position 100 im Unicode-Zeichensatz einnimmt. In *Zeile 5* wird die Fließkommazahl `5.0E9`, d.h. fünf Milliarden, in einen Wert des Typs `int` umgewandelt. Da dieser Wert außerhalb des Wertebereichs von `int` liegt, erhält `i` den größten darstellbaren Wert des Typs `int`: 2 147 483 647. Diese letzte Konvertierung zeigt die Problematik der explizit erzwungenen Typanpassung. Die Typanpassung wird in Kapitel 7.5 erneut aufgegriffen.

2.5 Prioritäten

Wie stark binden die verschiedenen Operatoren? Dem Leser ist aus der Mathematik sicherlich die Regel »*Punktrechnung geht vor Strichrechnung*« geläufig, nach der der Ausdruck 6+3*7 eben 27 und nicht 63 ergibt. In Java wird die Bindung von Operatoren durch die Vergabe von Prioritäten an die einzelnen Operatoren geregelt, so daß ein Operator mit hoher Priorität stärker bindet als einer mit niedriger. Bei gleichrangigen Operatoren erfolgt die Auswertung von links nach rechts, wobei immer der linke Operand vollständig ausgewertet wird, bevor mit der Auswertung des rechten Operanden begonnen wird.

Durch Klammerung kann man – wie in der Mathematik üblich – die durch die Prioritäten gegebene Auswertungsreihenfolge beeinflussen, so daß beispielsweise (6+3)*7 dann doch 63 ergibt. Auch läßt sich durch Klammerung sicherstellen, daß ein Ausdruck der Form `a==5==b==9` korrekt ausgewertet werden kann, wenn `a` und `b` vom Typ `int` vereinbart sind. Die Auswertung von links nach rechts für `a==5` würde hier einen Wert des Typs `boolean` liefern, dessen Vergleich mit dem ganzzahligen Wert von `b` zu einem Fehler führen würde, da diese Typen nicht ineinander überführt werden können. Die folgende Klammerung führt dagegen zu einer legalen Auswertungsreihenfolge: `(a==5)==(b==9)`.

Den bislang betrachteten Operatoren sind die in der folgenden Tabelle angegebenen Prioritäten zugeordnet, wobei ein höherer Prioritätswert stärker bindet. So besitzt beispielsweise der *modulo*-Operator `%` eine höhere Priorität als der Vergleichsoperator `>`.

Ein multiplikativer Operator bindet also stärker als ein relationaler, der wiederum stärker bindet als eine Zuweisung, so daß das folgende Programmfragment einer Booleschen Variablen `b` den Wert `true` zuweist:

```
boolean b;
b = 7 % 3 > 0
```

Art des Operators	Operator	Priorität
Zuweisung	=	0
Boolescher Operator	||	1
	&&	2
	|	3
	^	4
	&	5
relationaler Operator	==, !=	6
	>, >=, <, <=	7
additiver Operator	+, -	8
multiplikativer Operator	*, /, %	9
unärer Operator	!, -, +, (*Typ*)	10

2.6 Elementare Anweisungen

Bis jetzt haben wir den Aufbau von einfachen Klassen und Methoden betrachtet und dabei die primitiven Datentypen kennengelernt, die uns Java zur Verfügung stellt. In diesem Abschnitt diskutieren wir nun die Frage, wie diese Bestandteile eingesetzt werden, um Programme zu konstruieren, und stellen die elementaren Anweisungen der Programmiersprache Java vor. Nach einer kurzen Betrachtung von Blöcken und Vereinbarungen werden wir dann zunächst die Zuweisung behandeln, ein fundamentales Konzept, das den Austausch von Werten zwischen Variablen ermöglicht. Weiterhin werden wir betrachten, wie wir in Abhängigkeit von Werten den Kontrollfluß eines Programms steuern können, wie wir also beispielsweise Werte auf Gleichheit überprüfen und in Abhängigkeit vom Ergebnis dieser Überprüfung weitere Anweisungen ausführen können. Dies geschieht durch die bedingte oder die bewachte Anweisung. Danach zeigen wir, wie man eine Anweisung oder eine Anweisungsfolge mehrfach durchläuft. Dies kann im Prinzip auf zwei Arten geschehen, nämlich indem man den wiederholten Durchlauf von Bedingungen abhängig macht, oder indem man unmittelbar die Anzahl der Wiederholungen festlegt. Die erste Variante wird durch eine Bedingungsschleife realisiert, die zweite wird durch die Zählschleife ermöglicht. Wir werden dabei sehen, daß Zählschleifen in Java zugleich immer auch Bedingungsschleifen sein können. Zusätzlich werden wir Sprunganweisungen betrachten.

Anschließend werden wir das Feld als erste einfache Datenstruktur kennenlernen, zu deren Bearbeitung wir Schleifen und bedingte Anweisungen benötigen. Um Ihnen einen Eindruck von der Arbeitsweise mit den elementaren Anweisungen und Feldern zu geben, beschäftigen wir uns zum Abschluß des Kapitels mit einem sehr einfachen und recht ineffizienten Sortieralgorithmus, dem *Sortieren durch Einfügen*.

2.6.1 Blöcke

Ein *Block* faßt Anweisungen zusammen, die einzeln durch ein Semikolon abgeschlossen werden. Die Folge der Anweisungen innerhalb des Blocks bestimmt die Reihenfolge, in der diese Anweisungen bei der Programmausführung abgearbeitet werden. Ein Block stellt selbst wieder eine Anweisung dar; ein abschließendes Semikolon ist hinter einem Block allerdings nicht notwendig. Die Syntax eines Blocks hat folgende Form, wobei Aw_1 bis Aw_n wieder Anweisungen sind:

```
{
  Aw1 ;
  Aw2 ;
  ...
  Awn ;
}
```

Wir haben Blöcke bereits in den ersten Beispielen ohne besondere Erläuterung genutzt. Da Blöcke an vielen Stellen notwendig sind, um Programme geeignet zu strukturieren, werden sie in den folgenden Programmbeispielen immer wieder eingesetzt. Wir verzichten daher an dieser Stelle auf weitere Beispiele.

2.6.2 Deklarationen

Auch die durch Beispiele der vorangehenden Kapitel bereits eingeführte *Deklaration* von Variablen ist eine Anweisung, mit der ein Name innerhalb eines Programmabschnitts vereinbart wird. Deklarationen können somit an allen Stellen eines Programms auftreten, an denen Anweisungen erlaubt sind. Deklarationen müssen insbesondere nicht am Anfang eines Blocks erfolgen. Allerdings sind die vereinbarten Variablen auch erst ab der Position innerhalb des Programms bekannt, an der die Deklaration vorgenommen wurde. Wir werden die Sichtbarkeit von Variablen innerhalb des Programmtextes in Kapitel 2.7.1 erneut aufgreifen.

2.6.3 Zuweisungen

Die *Zuweisung* ist ein fundamentales Konzept, bei dem ein Wert an eine Variable übertragen wird. Wir unterscheiden in Java verschiedene Formen der Zuweisung: die einfache Zuweisung, die mehrfache Zuweisung, die Initialisierung, sowie Zuweisungsoperatoren in der Form von Infix-, Präfix- und Postfixoperatoren.

- einfache Zuweisung

  Notation: `a = b`

  Bedeutung: Die Variable `a` erhält den Wert der Variablen `b` zugewiesen.

  weitere Beispiele:

  `a = 5` `a` erhält den Wert der Konstanten `5` zugewiesen.

  `a = 5+5` `a` erhält den Wert `10`, das Ergebnis der Auswertung des Ausdrucks `5+5`. Die Auswertungsfolge richtet sich nach den in Abschnitt 2.5 eingeführten Prioritäten, die Addition besitzt eine höhere Priorität als die Zuweisung.

  Anmerkung: Die Zuweisung ist nicht nur eine Anweisung, sondern zugleich auch ein Ausdruck. Sie liefert selbst den zugewiesenen Wert als Ergebnis, das weiterverwendet werden kann. Das folgende Beispiel demonstriert den Effekt:

  ```
  a = (b=5)+(c=10)
  ```

  `b` erhält den Wert `5`, `c` den Wert `10` und `a` den Wert `15` zugewiesen, die Summe der von den Zuweisungen an `b` und `c` gelieferten Werte. Die Klammerung ist notwendig, da sonst die höhere Priorität der Addition zuerst die Berechnung von `5+c` erzwingen und daher `b` auf `15` gesetzt würde.

- mehrfache Zuweisung

  Notation: `a = b = 33`

  Bedeutung: Mehrfache Zuweisungen werden entgegen der üblichen Bearbeitungsreihenfolge immer von rechts nach links ausgeführt. Daher erhält zunächst `b` den Wert `33` und anschließend `a` den Wert von `b`, so daß letztlich beide Variablen den Wert `33` annehmen.

- Initialisierung bei der Deklaration

  Notation: `int a = 55`

  Bedeutung: Die Variable `a` vom Typ `int` wird deklariert und direkt mit dem Wert `55` belegt. In Java muß Variablen bei ihrer Deklaration kein gültiger Wert ihres Typs zugewiesen werden, der Compiler verlangt jedoch eine Zuweisung vor der ersten Auswertung einer Variablen. Bei der Übersetzung wird überprüft, ob eine auszuwertende Variable auf jeden Fall zuvor durch das Programm einen Wert zugewiesen bekommt.

- Infix-Zuweisungsoperator

  Notation: `a += 120`

  Bedeutung: Der Wert der Variablen `a` wird um `120` erhöht, d.h. die Variable, an die die Zuweisung erfolgt, wird zugleich als erster Operand der Addition

betrachtet. Der Infix-Zuweisungsoperator ermöglicht somit lediglich eine abkürzende Schreibweise für `a = a+120`.

Analog sind auch die Operatoren `-=`, `*=`, `/=`, `%=`, `&=`, `|=` und `^=` definiert. Infix-Zuweisungsoperatoren besitzen die gleiche Priorität wie die Zuweisung.

- Präfix-Zuweisungsoperator

  Notation: `++a`

  Bedeutung: Der Wert der Variablen `a` wird inkrementiert, d.h. um `1` erhöht. `++a` stellt daher eine abkürzende Schreibweise für `a = a+1` dar. Analog wird durch `--a` die Variable `a` dekrementiert, d.h. um `1` vermindert. Wie einfache Zuweisungen liefern auch Präfixoperatoren als Ergebnis den Wert der in- bzw. dekrementierten Variablen. Präfixoperatoren besitzen die hohe Priorität unärer Operatoren.

Die nachfolgend vorgestellten Postfixoperatoren unterscheiden sich von den anderen Formen der Zuweisung dadurch, daß als Ergebnis nicht der Wert *nach* der Ausführung der Zuweisung geliefert wird, sondern der Wert der Variablen, den diese *vor* der Zuweisung besessen hat. Dieses Verhalten führt dazu, daß eine durch den Postfixoperator bewirkte Veränderung erst bei einem nachfolgenden Zugriff auf die Variable sichtbar wird.

- Postfix-Zuweisungsoperator

  Notation: `a++`

  Bedeutung: Der Wert der Variablen `a` wird inkrementiert, d.h. um `1` erhöht. `a++` stellt daher – wie auch `++a` – eine abkürzende Schreibweise für `a = a+1` dar. Im Gegensatz dazu liefert `a++` jedoch den ursprünglichen Wert von `a` und nicht den inkrementierten Wert als Ergebnis der Zuweisung. Der Postfixoperator `a++` ermöglicht innerhalb eines komplexeren Ausdrucks ein Weiterarbeiten mit den alten Werten und zugleich ein verzögert wirksam werdendes Inkrementieren. Analog wird durch `a--` die Variable `a` dekrementiert, d.h. um `1` vermindert. Postfixoperatoren besitzen die hohe Priorität unärer Operatoren.

  Beispiel:

```
int z1 = 1, z2 = 2;                    // 1
EA.println(z1++ + z2++);               // 2
EA.println(z1 + z2);                   // 3
EA.println(++z1 + ++z2);               // 4
```

Die Postfixoperatoren *(Zeile 2)* inkrementieren die Variablen `z1` und `z2`, zurückgegeben und addiert werden jedoch deren ursprüngliche Werte `1` und `2`, so daß der Wert `3` ausgegeben wird. In *Zeile 3* werden `z1` und `z2`, deren Werte in *Zeile 2*

auf `2` bzw. `3` erhöht wurden, erneut addiert, so daß nun der Wert `5` angezeigt wird. Die Präfixoperatoren *(Zeile 4)* inkrementieren anschließend `z1` und `z2` und liefern unmittelbar die jeweils um `1` erhöhten Werte an die Addition: Es wird der Wert `7` ausgegeben.

2.6.4 Bedingte Anweisungen

Die *bedingte Anweisung* dient zur Auswahl genau einer Anweisung aus zwei alternativen Anweisungen. Sie hat in Java die folgende syntaktische Form:

```
if ( Ad )
  Aw1
else
  Aw2
```

Für die bedingte Anweisung gilt:

- *Ad* muß ein Ausdruck sein, dessen Auswertung einen Wert des Typs `boolean` liefert. *Ad* wird immer in Klammern eingeschlossen.
- Aw_1 und Aw_2 sind Anweisungen. Da Blöcke ebenfalls Anweisungen sind, sind an diesen Stellen insbesondere auch Blöcke erlaubt.
- Die bedingte Anweisung beginnt mit dem Schlüsselwort `if`. Sie endet dort, wo die Anweisung Aw_2 endet. In den meisten Fällen wird dort ein Semikolon als Trennzeichen dienen, um die bedingte Anweisung von anderen, in einem Block folgenden Anweisungen zu trennen.
- Bei der Ausführung der bedingten Anweisung wird zunächst *Ad* ausgewertet. Ergibt die Auswertung den Wert `true`, so wird Aw_1 ausgeführt und die bedingte Anweisung verlassen, ohne daß Aw_2 ausgeführt wurde. Ergibt die Auswertung den Wert `false`, so wird Aw_1 übersprungen und Aw_2 ausgeführt. Anschließend wird die bedingte Anweisung verlassen.
- Der Teil `else` Aw_2 kann entfallen. Liefert *Ad* in diesem Fall den Wert `false`, so wird die bedingte Anweisung ohne weitere Ausführung verlassen.

Neue Schlüsselwörter: `if, else`

Beispiel:

```
if (EA.readInt() == 10)
  EA.println("Eine Zehn wurde eingegeben!");
else
  EA.println("Es wurde keine Zehn eingegeben!");
```

Eine ganze Zahl wird eingelesen und mit dem Wert `10` verglichen. Das Ergebnis dieses Vergleichs wird durch die Ausgabe einer entsprechenden Zeichenkette angezeigt.

Beispiel:

```
int zahl = EA.readInt();
if (zahl >= 0)
  if (zahl > 0)
    EA.println("Es wurde eine positive Zahl eingegeben!");
  else
    EA.println("Es wurde eine Null eingegeben!");
```

Werden bedingte Anweisungen ineinander geschachtelt, so wird jeder Teil der Form `else` Aw_2 immer dem am nächsten davor stehenden `if` zugeordnet. Im Beispiel wird daher der Text »`Es wurde eine Null eingegeben!`« nur dann ausgegeben, wenn zunächst der erste Ausdruck `zahl >= 0` den Wert `true` liefert, so daß die innere bedingte Anweisung ausgeführt wird, und dabei die Auswertung der zugehörigen Bedingung `zahl > 0` den Wert `false` ergibt.

Dieses Beispiel gibt uns die Gelegenheit, als weiteren Operator den Auswahloperator »`?:`« einzuführen, dessen Wirkungsweise an die bedingte Anweisung erinnert. Der Auswahloperator verbindet drei Operanden und unterscheidet sich damit von den bisher eingeführten Operatoren, die nur höchstens zwei Operanden miteinander verknüpfen.

Ad_1 `?` Ad_2 `:` Ad_3 hat die folgende Wirkungsweise:

- Der Ausdruck Ad_1 muß einen Wert vom Typ `boolean` liefern.
- Ad_2 und Ad_3 sind zwei Ausdrücke, deren Auswertungen Ergebnisse des gleichen Typs ergeben.
- Ergibt Ad_1 den Wert `true`, so wird das Ergebnis der Auswertung von Ad_2 zum Ergebnis dieses Operators, ergibt Ad_1 den Wert `false`, so wird das Ergebnis von Ad_3 zurückgegeben.

Das zuletzt mit einer bedingten Anweisung formulierte Beispiel läßt sich mit dem Auswahloperator auch so formulieren:

```
if (zahl >= 0)
  EA.println(
    zahl > 0   ? "Es wurde eine positive Zahl eingegeben!"
               : "Es wurde eine Null eingegeben!"
  );
```

Funktional nicht völlig identisch, aber doch sehr ähnlich, ziemlich unübersichtlich und daher nicht empfehlenswert ist die folgende Fassung:

```
EA.println(
  zahl >= 0  ?
    ( zahl > 0   ? "Es wurde eine positive Zahl eingegeben!"
                 : "Es wurde eine Null eingegeben!"
    )          : ""
  );
```

Ausgegeben wird in beiden Fassungen jeweils das Ergebnis der Auswertung des Auswahloperators, d.h. ein vorgegebener Text. In der zweiten Fassung muß auch für den Fall `zahl < 0` ein Text geliefert werden. Da keine Ausgabe erfolgen soll, wird der durch `""` erzeugte »leere« Text als Wert an `EA.println` übergeben.

2.6.5 Bewachte Anweisungen

Die *bewachte Anweisung* dient zur Auswahl einer bestimmten Anweisungsfolge aus einer Menge von alternativen Anweisungsfolgen. Sie hat in Java die folgende syntaktische Form:

```
switch ( Ad ) {
  case K1:   AL1;
  case K2:   AL2;
  ...
  case Kn:   ALn;
  default:   ALdef
}
```

Hierbei gilt:

- Bei der Ausführung der bewachten Anweisung wird zunächst *Ad* ausgewertet. Stimmt der Wert von *Ad* mit der Konstanten K_i aus der Menge der K_1 ... K_n überein, so werden alle Anweisungen aus der Liste von Anweisungen AL_i und alle Anweisungen der nachfolgenden Anweisungslisten AL_{i+1}, ... ALn, AL_{def} ausgeführt. Stimmt der Wert von *Ad* mit keinem Wert K_i überein und ist die Angabe `default:` AL_{def} vorhanden, so werden die Anweisungen von AL_{def} ausgeführt, anschließend wird die bewachte Anweisung verlassen.
- *Ad* ist ein Ausdruck, dessen Auswertung einen Wert der Typen `byte`, `short`, `int` oder `char` liefert. *Ad* wird immer in Klammern eingeschlossen. Nach dem Ausdruck folgt immer ein Block.
- Die bewachte Anweisung beginnt mit dem Schlüsselwort `switch`. Sie endet mit dem zu ihr gehörenden Block.
- K_i sind unterschiedliche Konstanten desjenigen Typs, der von *Ad* geliefert wird.

- AL_i sind Listen von Anweisungen, bei denen die einzelnen Anweisungen jeweils durch ein Semikolon getrennt werden.
- Soll die Ausführung von nacheinander liegenden Anweisungenfolgen unterbrochen werden, so muß dies mit der Anweisung `break` explizit veranlaßt werden. Jedes `break` innerhalb der bewachten Anweisung verzweigt direkt zum Ende des Blocks der bewachten Anweisung.
- Der Teil `default:` AL_{def} darf entfallen. Liefert *Ad* in diesem Fall einen Wert, der mit keiner Konstanten K_i übereinstimmt, so wird die bewachte Anweisung verlassen, ohne daß eine Anweisung ausgeführt wird.

Das vorgestellte Konstrukt wird als *bewachte Anweisung* bezeichnet, da die Konstante K_i die Anweisungsliste AL_i *bewacht*.

Neue Schlüsselwörter: `switch, case, default, break`

Beispiel:

```
switch (EA.readInt()) {
  case 2:    EA.print("zwei, ");
  case 1:    EA.println("eins"); break;
  case 10:   EA.print("zehn und keine ");
  default:   EA.println("andere Zahl");
}
```

Der eingelesene Wert wird mit den Werten `1`, `2` und `10` verglichen. Stimmt der eingelesene Wert mit keiner der Konstanten überein, so wird die Bemerkung »`andere Zahl`« ausgegeben. Stimmt der Wert überein,

- so wird für den Wert `2` die Zeichenfolge »`zwei, eins`« ausgegeben, da alle Anweisungen bis zum Auftreten des ersten `break` ausgeführt werden,
- beim Wert `1` wird nur »`eins`« gedruckt,
- für den Wert `10` wird »`zehn und keine andere Zahl`« gedruckt, da auf die Ausgabe von »`zehn und keine `« nicht die Anweisung `break` folgt, und
- für alle anderen Werte wird nur der der Text »`andere Zahl`« ausgegeben.

Da die bewachte Anweisung nicht den Typ `boolean` für das Ergebnis des Ausdrucks *Ad* zuläßt, kann eine bedingte Anweisung nicht direkt in eine bewachte Anweisung umgewandelt werden. Auch kann die bewachte Anweisung nicht unmittelbar als Schachtelung oder Folge bedingter Anweisungen formuliert werden. Das Verhalten der bewachten Anweisung, alle folgenden Anweisungen bis zur nächsten `break`-Anweisung auszuführen, würde in bedingten Anweisungen eine Ergänzung von zusätzlichen Variablen zur Steuerung des Ablaufs erfordern. Die bedingte Anweisung und die bewachte Anweisung stellen daher in Java orthogonale Konzepte dar.

2.6.6 Bedingungsschleifen

Die *Bedingungsschleife* dient zur Wiederholung einer Anweisung. Eine der beiden in Java möglichen syntaktischen Formen ist:

```
while ( Ad )
  Aw
```

Für die Bedingungsschleife gilt:

- *Ad* ist ein Ausdruck, der einen Wert vom Typ `boolean` liefert.
- Bei der Ausführung der Bedingungsschleife wird zuerst *Ad* ausgewertet. Ergibt die Auswertung den Wert `true`, so wird zunächst die Anweisung *Aw* ausgeführt und anschließend mit der erneuten Auswertung des Ausdrucks *Ad* fortgefahren. Ergibt die Auswertung von *Ad* den Wert `false`, so wird die Ausführung der Bedingungsschleife beendet. Die Ausführung der Anweisung *Aw* wird also solange wiederholt, bis der Ausdruck *Ad* den Wert `false` ergibt.

Neues Schlüsselwort: `while`

Beispiel:

```
EA.println("Bitte zwei natürliche Zahlen eingeben!");
int zahl1, zahl2;
zahl1 = EA.readInt();
zahl2 = EA.readInt();
if (zahl1>0 & zahl2>0) {
  EA.println("Berechnung des ggt von " + zahl1 + " und " + zahl2);
  while (zahl1 != zahl2)
    if (zahl1 > zahl2)
      zahl1 -= zahl2;
    else
      zahl2 -= zahl1;
  EA.println("ggt ist: " + zahl1);
} else
  EA.println("Bitte positive natürliche Zahlen eingeben!");
```

Durch diese Schleife wird der größte gemeinsame Teiler der in den Variablen `zahl1` und `zahl2` abgelegten Werte nach dem klassischen Algorithmus von *Euklid* für natürliche Zahlen berechnet. Bei jedem Durchlauf durch den Rumpf der Schleife wird der Wert der größeren der beiden Zahlen um den Wert der kleineren Zahl vermindert. Solange sich die Zahlen unterscheiden, ergibt `zahl1!=zahl2` den Wert `true`, so daß die Schleife erneut ausgeführt wird. Sind beide Zahlen gleich, bricht die Schleife ab, da die Auswertung des Vergleichs den Wert `false` liefert. Der größte gemeinsame Teiler steht dann in beiden Variablen `zahl1` und `zahl2`.

Man erkennt, daß bei Bedingungsschleifen sichergestellt werden muß, daß nach einer endlichen Zahl von Durchläufen das Ergebnis der Auswertung des Ausdrucks *Ad* den Wert `false` ergibt, damit die Schleife abbricht. Man spricht davon, daß die Schleife *terminiert*. Das Terminieren einer Schleife kann nur dann garantiert werden, wenn die Ausführung der Anweisung *Aw* oder auch die Auswertung des Ausdrucks *Ad* eine Änderung von den Werten bewirken, die Einfluß auf die Berechnung des Ergebnisses des Ausdrucks *Ad* nehmen.

Die zweite syntaktische Form der Bedingungsschleife hat das folgende Aussehen:

```
do
  Aw
while ( Ad )
```

Für diese Form der Bedingungsschleife gilt:

- *Ad* ist wiederum ein Ausdruck, der einen Wert vom Typ `boolean` liefert.
- Bei der Ausführung der Bedingungsschleife wird zuerst *Aw* ausgeführt. Der Ausdruck *Ad* wird erst im Anschluß ausgewertet, so daß die Anweisung *Aw* im Gegensatz zu der oben vorgestellten Form der Schleife auf jeden Fall mindestens einmal ausgeführt wird. Ergibt die Auswertung von *Ad* den Wert `true`, so wird die Anweisung *Aw* erneut ausgeführt und auch eine erneute Auswertung des Ausdrucks *Ad* angeschlossen. Ergibt die Auswertung von *Ad* den Wert `false`, so wird die Ausführung der Bedingungsschleife beendet.

Neues Schlüsselwort: `do`

Beispiel:

```
int zahl;
do {
  EA.println("Bitte eine positive natürliche Zahl eingeben!");
  zahl = EA.readInt();
} while (zahl <= 0);
```

Das Beispiel zeigt einen typischen Einsatz der Schleife mit `do ... while`. Der Benutzer des Programms wird im Anweisungsteil zur Eingabe einer Zahl aufgefordert, die einer bestimmten Bedingung – positiv und natürlich – genügen soll. Da die Bedingung erst nach dem Einlesen überprüft werden kann, folgt die Bedingung dem Anweisungsblock. So wird sichergestellt, daß vor dem ersten Auswerten der Bedingung in jedem Fall ein Wert gelesen wurde. Solange die eingegebene Zahl kleiner oder gleich Null ist, also der vorgegebenen Bedingung nicht genügt, wird der Nutzer erneut zur Eingabe aufgefordert.

2.6.7 Zählschleife

Die *Zählschleife* ermöglicht wie die Bedingungsschleife ein mehrfaches Wiederholen einer Anweisung oder einer in einem Block zusammengefaßten Folge von Anweisungen. Zählschleifen dienen grundsätzlich dazu, eine vorgegebene Anzahl von Durchläufen (*Iterationen*) auszuführen. Eine Zählschleife hat den folgenden syntaktischen Aufbau:

```
for ( InitL ; Ad ; AktL )
  Aw
```

Für die Zählschleife gilt:

- Die Initialisierungsliste *InitL* wird nur einmal zu Beginn der Ausführung der Schleife ausgeführt. Die konzeptionelle Idee der Initialisierungsliste ist, Zählervariablen zu vereinbaren und mit Werten zu belegen, anhand derer anschließend die Anzahl der Schleifendurchläufe kontrolliert wird.

  InitL ist eine durch Kommata getrennte Liste von Ausdrücken, die entweder zur Initialisierung bereits vereinbarter oder zur Deklaration neuer Variablen dient. Zur Initialisierung sind grundsätzlich Zuweisungen geeignet, wobei die Ergebnisse von Auswertungen ignoriert werden. Alternativ kann in der Initialisierungsliste die Deklaration neuer Variablen erfolgen, die dann jedoch alle den gleichen Typ besitzen müssen. Diese Variablen sind dann nur innerhalb der Schleifenkonstruktion bekannt und können auch nur dort benutzt werden.

- *Ad* ist ein Ausdruck, dessen Auswertung einen Wert des Typs `boolean` ergeben muß. Ist dieser Wert `true`, so wird die Anweisung *Aw* ausgeführt, ist der Wert `false`, so wird die Schleife beendet.

- *AktL* ist eine Liste von Ausdrücken, die immer nach der Ausführung der Anweisung *Aw* ausgeführt werden. Diese Ausdrücke sollen dazu dienen, die Werte von Zählervariablen zu aktualisieren. Im Anschluß an die Auswertung von *AktL* wird der Ausdruck *Ad* erneut ausgewertet.

- Jeder Bestandteil der Zählschleife innerhalb der Klammerung, d.h. *InitL*, *Ad* oder *AktL*, kann auch entfallen, falls keine Notwendigkeit gesehen wird, eine entsprechende Angabe zu machen.

- *Aw* ist eine Anweisung.

Neues Schlüsselwort: `for`

Beispiele:

```
int summe = 0;
for (int zähler = 1; zähler <= 12; zähler++)
  summe += EA.readInt();
EA.println(summe);
```

Die Variable `summe` wird als ganzzahlig deklariert und mit dem Wert `0` initialisiert. Die Zählschleife liest zwölf Zahlen ein und addiert sie auf, indem der eingelesene Wert jeweils zu der Variablen `summe` hinzugefügt und das Resultat wieder `summe` zugewiesen wird. Nach Beendigung der Schleife wird der Wert der Summe ausgedruckt.

Im nächsten Beispiel werden mit der bekannten Methode `EA.readInt` zwölf Zahlen eingelesen und mit dem Wert der Variablen `grenze` verglichen. Die Variablen `kleiner` und `größer` geben anschließend an, wie viele Zahlen (strikt) kleiner bzw. größer als der Wert von `grenze` sind.

```
int zahl, grenze = 4, größer = 0, kleiner = 0;
for (int i = 1; i <= 12; i++) {
  zahl = EA.readInt();
  if (zahl < grenze) kleiner++;
  if (zahl > grenze) größer++;
}
EA.println("größer/kleiner: " + größer + "/" + kleiner);
```

Leider wird in Java nicht erzwungen, daß die Anzahl der Durchläufe einer Zählschleife ausschließlich über den Wert eines Zählers gesteuert wird. Die für die Zählschleife geltenden Regeln stellen beispielsweise nicht sicher, daß in dem Ausdruck *Ad* tatsächlich eine Zählervariable als Abbruchkriterium abgefragt wird oder daß in *AktL* ausschließlich Zähler aktualisiert werden. Da *InitL* und *AktL* auch unbelegt bleiben können, läßt sich vielmehr jede Bedingungsschleife der Form

`while (` *Ad* `)` *Aw*

auch als Zählschleife schreiben:

`for ( ;` *Ad* `; )` *Aw*

Sofern die Anweisung *Aw* selbst nur aus einer Folge von Zuweisungen besteht, können diese auch in *AktL* aufgenommen werden, so daß die Anweisung *Aw* der Schleife leer bleibt. Der folgende Programmtext zeigt das entsprechend veränderte Beispiel des Einlesens von zwölf ganzen Zahlen mit dem anschließenden Ausgeben ihrer Summe. Das Beispiel zeigt, daß der übersichtliche und verständliche Einsatz der Zählschleife in Java weitgehend dem Programmierer überlassen wird.

```
int summe = 0;
for (int zähler = 1; zähler<=12; summe += EA.readInt(), zähler++)
  /* Der leere Anweisungsteil der Schleife */  ;
EA.println(summe);
```

2.6.8 Markierte Anweisungen und Sprünge

Markierte Anweisungen sind beliebige Anweisungen, die einen Namen als *Marke* (engl. *label*) tragen. Die Ausführung einer so ausgezeichneten Anweisung kann dann zusätzlich durch die beiden Anweisungen `break` und `continue` kontrolliert werden. Syntaktisch haben markierte Anweisungen die Form `m:`*Aw*. Ist die Anweisung *Aw* aus verschiedenen syntaktischen Bestandteilen zusammengesetzt, beispielsweise also ein Block oder eine bedingte Anweisung, so gilt:

- Enthält sie die Anweisung `break m;` und wird diese ausgeführt, so wird mit der Anweisung fortgefahren, die unmittelbar auf die markierte Anweisung folgt.
- Enthält eine Schleife die Anweisung `continue m;` und wird diese ausgeführt, so wird an das Ende des Rumpfes der markierten Schleife gesprungen. Bei `for`-Schleifen wird zunächst die Aktualisierung von Zählern vorgenommen, anschließend wird bei allen Arten von Schleifen mit der Auswertung der Abbruchbedingung fortgefahren.

Neues Schlüsselwort: `continue`

Beispiele:

Es werden solange ganze Zahlen eingelesen, wie diese größer als der Wert `0` sind. Ist eine eingelesene Zahl größer oder gleich dem Wert `10`, so wird die markierte Schleife sofort erneut durchlaufen, ohne die Ausgabe zu erzeugen.

```
int eingabe = 0;
schleife:
  while ((eingabe = EA.readInt()) > 0) {
    if (eingabe >= 10)
      continue schleife;
    EA.println(eingabe + " ist kleiner als 10");
  }
```

Im nachfolgenden Beispiel werden maximal zwölf Zahlen gelesen und aufsummiert. Sobald die Summe `summe` jedoch größer als `19` ist, wird die Schleife auch dann schon verlassen, wenn noch nicht zwölf Zahlen eingelesen wurden. Die durch den Zähler kontrollierte Wiederholung wird vorzeitig beendet.

```
int summe = 0;
test19:
  for (int i = 1; i <= 12; i++) {
    summe += EA.readInt();
    if (summe > 19)
      break test19;
  }
```