o'reillys basics

Praxiswissen Ruby

→ Von den Grundlagen bis zur Objektorientierung

→ Web- und Netzwerkanwendungen mit Ruby

→ Mit wertvollem Zusatzwissen

Sascha Kersken

Praxiswissen Ruby

Sascha Kersken

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.
Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Alle Rechte vorbehalten einschließlich der Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
Tel.: 0221/9731600
Fax: 0221/9731608
E-Mail: kommentar@oreilly.de

Copyright der deutschen Ausgabe:
© 2007 by O'Reilly Verlag GmbH & Co. KG
1. Auflage 2007

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

Lektorat: Volker Bombien
Fachgutachten: Sven Riedel, München
Korrektorat: Oliver Mosler, Köln
Satz: G&U Language & Publishing Services GmbH, Flensburg; www.GundU.com
Umschlaggestaltung: Michael Oreal, Köln
Coverabbildung: Perkin Elmer Electronics, Wiesbaden & Dipl.-Designer Guido Bender
Produktion: Andrea Miß, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-89721-478-1

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Vorwort IX
Aufbau des Buchs XI
Danksagungen XIII

1 Einführung 1
Etwas Ruby-Geschichte 1
Merkmale und Vorzüge von Ruby 3
Ruby installieren 4
Ruby-Code eingeben und ausführen 10
'Hello World, hello Ruby' – das erste Ruby-Programm 17
Zusammenfassung 21

2 Sprachgrundlagen 23
Praktische Einführung 24
Grundlagen der Syntax 31
Variablen, Ausdrücke und Operationen 35
Kontrollstrukturen 72
Mustervergleiche mit regulären Ausdrücken 85
Iteratoren und Blöcke 101
Zusammenfassung 107

3 **Ruby-Klassen** **111**
Was ist Objektorientierung? 111
Ein- und Ausgabe 114
Datum und Uhrzeit 138
Einige weitere Klassen 145
Die Ruby-Hilfe ri 148
Zusammenfassung 150

4 **Eigene Klassen und Objekte erstellen** **153**
Objektorientierte Programmierung – Eine praktische Einführung 153
Klassen entwerfen und implementieren 163
Weitere objektorientierte Konstrukte 198
Zusammenfassung 215

5 **Netzwerkanwendungen** **219**
Kurze Einführung in TCP/IP 219
Sockets 229
Web-Clients mit Net::HTTP 241
Prozesse und Threads 256
Zusammenfassung 271

6 **Klassische Webanwendungen** **273**
Den Webserver Apache 2 installieren 274
CGI-Skripten mit Ruby 286
Zugriff auf Datenbanken 314
Zusammenfassung 330

7 **Ruby on Rails** **333**
Rails installieren und in Betrieb nehmen 336
Die erste Rails-Anwendung 338
Realistische Anwendung: Eine Online-Rock-n-Roll-Datenbank 341
Zusammenfassung 355

A **Ruby-Kurzreferenz** **357**
Syntax 357
Ausdrücke 357
Kontrollstrukturen 360
Reguläre Ausdrücke 361

Klassendefinition 362
Klassenreferenz 363

B Ressourcen und Tools 371
Bücher 371
Web-Ressourcen 373

Index 375

Vorwort

A Programmer's Best Friend
– Slogan auf der Ruby-Website[1]

Im Jahr 2002 entdeckte ich auf einem Wühltisch einer Kölner Buchhandlung ein verbilligtes Exemplar des Buchs »Programmieren mit Ruby« (die deutsche Ausgabe der ersten englischsprachigen Veröffentlichung zum Thema). Die Tatsache, dass das Buch bald nach Erscheinen verramscht wurde, zeigt, dass Ruby damals in Europa leider noch keine große Bedeutung hatte. Aber als leidenschaftlicher Sammler von Computerbüchern musste ich einfach zugreifen, denn von dieser Sprache hatte ich zwar bereits gehört, aber noch keinerlei praktische Erfahrungen damit gesammelt.

Unmittelbar nach Erwerb des Buchs probierte ich Ruby aus – es ist schließlich Open Source-Software, die man kostenlos herunterladen kann. Ich war praktisch sofort begeistert von der Klarheit und Eleganz der Formulierungen und vom Leistungsumfang der mitgelieferten Bibliotheken. Praktische Nutzanwendungen waren aber auch für mich erst einmal nicht zu entdecken, denn mein Sprachen-Werkzeugkasten war damals einigermaßen wohlgeordnet: Perl für Admin-Skripten, PHP fürs Web und Java für größere Anwendungen; gelegentlich noch etwas C und ein paar Shell-Skripten.

Der eigentliche Durchbruch für Ruby in der Entwicklerwelt außerhalb Japans – Rubys Heimat – kam erst 2004. Fachzeitschriften und Blogs begannen, über Ruby on Rails zu berichten, eine neue, gut durchdachte Umgebung für Webanwendun-

1 *http://www.ruby-lang.org*

gen. Durch eine saubere Trennung von Datenmodell, Programmierlogik und Inhalt macht Rails Schluss mit dem üblichen Durcheinander klassischer Webanwendungen. Wobei man fairerweise erwähnen sollte, dass nicht diese Anwendungen oder ihre Programmiersprachen an dem Chaos schuld sind, sondern die Abfolge einzelner Webseiten (in Kapitel 6 dieses Buchs erfahren Sie Näheres).

Es ist gut möglich, dass Sie dieses Buch lesen, weil Sie sich begeistert auf Rails gestürzt und dann plötzlich gemerkt haben, dass Sie noch nicht genug über die zugrunde liegende Sprache Ruby wissen. Sie bietet nämlich zahlreiche – wohlgemerkt überaus positive – Überraschungen. Die wichtigsten dieser Überraschungen werden in diesem Buch enthüllt, so dass Sie danach effizienter und sicherer mit Rails arbeiten können. Eine erste Einführung in Rails selbst finden Sie übrigens im letzten Kapitel.

Vielleicht geht es Ihnen aber gar nicht um Ruby on Rails, sondern Sie möchten einfach Ihren Horizont erweitern und eine neue Programmiersprache erlernen. Auch dann sind Sie hier vollkommen richtig. An ausgewählten Stellen werden Sie Bemerkungen finden, die Ruby mit anderen Programmiersprachen vergleichen. Ansonsten wird »The Ruby Way« gründlich erklärt, ohne übertrieben viel Zeit mit der Erläuterung theoretischer Konzepte zu verbringen.

Oder möglicherweise haben Sie überhaupt noch nie programmiert, und Sie möchten es nun anhand von Ruby endlich lernen. Eine exzellente Wahl, denn Ruby ist durch klare Schreibweisen und eindeutige Formulierungen besonders einsteigerfreundlich. Im Übrigen können Sie nirgends leichter und konsequenter das Prinzip der Objektorientierung erlernen, ohne das Programmierung heute kaum noch vorstellbar wäre. Kurz: Wenn Sie als Programmierneuling experimentierfreudig sind und ein recht schnelles Lerntempo mögen, ist dieses Buch auch für Sie genau das Richtige.

Egal, zu welcher der drei genannten Gruppen Sie gehören: Wenn Sie Ruby lernen, steigen Sie in eine schnell wachsende, lebendige Gemeinschaft von Programmierern ein. Denn Ruby gewinnt täglich neue Freunde aus aller Welt. Viele von ihnen stellen ihre Entwicklungen wieder der Allgemeinheit zur Verfügung, und so gibt es für beinahe jede Spezialaufgabe eine oder auch mehrere gute Lösungen. Wenn Sie erst über die Grundlagen hinaus sind, können auch Sie vielleicht die Ruby-Gemeinschaft bereichern, indem Sie Ihre eigenen Skripten öffentlich verfügbar machen. Im Anhang finden Sie die Adressen einiger Websites und Mailinglisten, wo jede Hilfe willkommen ist.

Dieses Buch beantwortet bei jedem Thema die Frage nach dem konkreten Nutzen und liefert praxisnahe Beispiele. Es versucht dabei, »self-contained« zu sein, das heißt die behandelten Themen weitgehend ohne Referenzen auf externe Quellen abzudecken. So erhalten Sie nebenher allerhand nützliches Zusatzwissen, zum Beispiel über TCP/IP-Netzwerke, Web- und Datenbankserver oder die CGI-Schnitt-

stelle. Solches Wissen brauchen Programmierer heutzutage dringend, denn immer mehr Software wird nicht für einen einzelnen Rechner, sondern für komplexe, verteilte Systeme geschrieben. Falls Sie über ein Thema bereits Bescheid wissen, steht es Ihnen natürlich frei, die betreffenden Abschnitte zu überblättern.

Auf der Website zum Buch bei O'Reilly (*http://www.oreilly.de/catalog/rubybasger*) sowie auf meiner eigenen Site (*http://buecher.lingoworld.de/ruby*) können Sie alle Listings herunterladen und erhalten nach und nach wichtige Zusatzinformationen. Bedenken Sie aber, dass Sie beim manuellen Abtippen unendlich viel mehr lernen, als wenn Sie einfach die fertigen Listings öffnen und ausführen ;-).

Und nun Leinen los und Anker lichten! Sie segeln in ein unentdecktes Land voller Abenteuer. Mit Entdeckergeist und Mut werden Sie bald viele seiner Geheimnisse lüften und für Ihre tägliche Arbeit nutzen können. Ich zumindest habe meinen erwähnten Werkzeugkasten inzwischen etwas umgepackt: Ruby für Admin-Skripten, Ruby on Rails oder Ruby-CGIs (und seltener PHP) fürs Web und nur im Notfall Java. Ruby hat mein (Programmierer-)Leben verändert, und auch Sie werden es gewiss nicht bereuen, diese Sprache zu lernen.

Aufbau des Buchs

Dieses Buch ist in sieben Kapitel und zwei Anhängen mit folgenden Inhalten unterteilt:

- Kapitel 1, *Einführung*, verliert zunächst einige Worte über Geschichte und Merkmale von Ruby. Danach wird gezeigt, wie Sie die Sprache und ihre Werkzeuge auf Ihrem Rechner installieren, und Sie schreiben Ihr erstes Ruby-Skript.
- In Kapitel 2, *Sprachgrundlagen*, werden alle Grundbausteine der Programmiersprache Ruby vorgestellt: Die Syntax, das heißt die korrekte Schreibweise von Befehlen, danach Ausdrücke und Variablen, Fallentscheidungen, Schleifen und so weiter, bis hin zu einigen Spezialwerkzeugen, die so nur Ruby zu bieten hat.
- Kapitel 3, *Ruby-Klassen*, ist der erste Teil einer gründlichen Einführung in die Objektorientierung: Hier lernen Sie, mit vielen nützlichen Klassen zu arbeiten, die zum Lieferumfang gehören. Die wichtigsten Themen sind Ein- und Ausgabe (auf der Konsole, in Dateien und so weiter), Datum und Uhrzeit sowie einige andere eingebaute Klassen.
- Kapitel 4, *Eigene Klassen und Objekte*, beschäftigt sich danach mit dem aktiven Teil der Objektorientierung: Sie lernen Schritt für Schritt, wie Sie Ihre eigenen Datenmodelle und Arbeitsabläufe in Klassen abbilden können. Das gesamte Handwerkszeug der objektorientierten Programmierung, das Ruby in besonders vollendeter Form zur Verfügung stellt, wird dabei gründlich erläutert.
- In Kapitel 5, *Netzwerkanwendungen*, erfahren Sie, wie Sie mit Hilfe eigener Programme auf lokale Netzwerke oder auf das Internet zugreifen können. Sie erhal-

ten zunächst einige Hintergrundinformationen über den Aufbau der TCP/IP-Protokolle und entwickeln dann auf dieser Grundlage Clients und Server. Ein Teil des Kapitels beschäftigt sich zudem mit dem Problem der *Nebenläufigkeit*, um Server zu schreiben, die mehrere Clients gleichzeitig bedienen.

- Kapitel 6, *Klassische Webanwendungen*, beschreibt zunächst, wie Sie den Webserver Apache einrichten und konfigurieren. Er dient als Plattform für die nachfolgend entwickelten Anwendungen. Hier geht es um die traditionelle Art der Webprogrammierung, die Sie erst als Grundlage verstanden haben sollten, bevor Sie nur noch Rails verwenden.
- In Kapitel 7, *Ruby on Rails*, erhalten Sie schließlich eine praxisorientierte Einführung in die Arbeit mit diesem äußerst komfortablen Web-Framework.
- Anhang A, *Ruby-Kurzreferenz*, enthält eine Kurzübersicht der wichtigsten Ruby-Anweisungen und -Klassen.
- Anhang B, *Ressourcen und Tools*, schließlich empfiehlt einige Bücher zum Weiterlernen sowie zu angrenzenden Themen. Daneben erhalten Sie die Adressen diverser wichtiger Websites zum Thema.

Typografische Konventionen

In diesem Buch werden folgende typografische Konventionen verwendet:

Kursivschrift

Wird für Datei- und Verzeichnisnamen, E-Mail-Adressen und URLs, für Schaltflächenbeschriftungen und Menüs sowie bei der Definition neuer Fachbegriffe und für Hervorhebungen verwendet.

`Nichtproportionalschrift`

Wird für Codebeispiele und Variablen, Funktionen, Befehlsoptionen, Parameter, Klassennamen und HTML-Tags verwendet.

`Nichtproportionalschrift fett`

Bezeichnet Benutzereingaben auf der Kommandozeile.

`Nichtproportionalschrift kursiv`

Kennzeichnet innerhalb von Codebeispielen Platzhalter, die Sie durch Ihre eigenen spezifischen Angaben ersetzen müssen.

Fett

Wird für Tastenkürzel und Shortcuts eingesetzt.

Die Glühbirne kennzeichnet einen Tipp oder einen generellen Hinweis mit nützlichen Zusatzinformationen zum Thema.

Der Regenschirm kennzeichnet eine Warnung oder ein Thema, bei dem Sie Vorsicht walten lassen sollten.

In Kästen mit einem Mikroskop wird ein Thema genauer unter die Lupe genommen.

Danksagungen

Zunächst einmal und vor allen Dingen danke ich **Ihnen**. Sie haben dieses Buch entweder schon gekauft, in einer Bücherei ausgeliehen oder lesen dieses Vorwort gerade in einer Buchhandlung, um festzustellen, ob es das richtige Buch für Sie ist. In all diesen Fällen zeigen Sie Interesse für meine Arbeit, und das ehrt mich bereits, denn ohne interessierte Leserschaft könnte sich jeder Autor seine Mühe gleich sparen.

Weiterer Dank geht an den O'Reilly Verlag, vor allem an meinen Lektor Volker Bombien. Er hat alle Teile des Manuskripts kritisch gelesen und mich stellenweise auf den besonderen Erklärungsbedarf von Programmieranfängern hingewiesen.

Besonderer Dank gebührt natürlich auch Yukihiro »Matz« Matsumoto, dem Erfinder der Programmiersprache Ruby, sowie den zahllosen bekannten oder unbekannten Mitgliedern der weltweiten Ruby-Gemeinschaft, die diese Sprache durch ihren unermüdlichen Einsatz zu einem so großartigen Werkzeug gemacht haben.

Zuletzt wäre keines meiner Bücher ohne die endlose Geduld und Ausdauer meiner Frau und meines Sohnes realisierbar gewesen. Auch euch also wieder einmal meinen herzlichsten Dank.

In diesem Kapitel:

- Etwas Ruby-Geschichte
- Merkmale und Vorzüge von Ruby
- Ruby installieren
- Ruby-Code eingeben und ausführen
- 'Hello World, hello Ruby' – das erste Ruby-Programm

KAPITEL 1
Einführung

Ruby my love
You'll be my love
You'll be my sky above
Ruby my light
You'll be my light
You'll be my day and night
You'll be mine tonight

– Cat Stevens, »Ruby Love«

In diesem Kapitel erfahren Sie zunächst das Wichtigste über die Entwicklungsgeschichte und die Eigenheiten der Programmiersprache Ruby. Danach wird beschrieben, wie Sie das Ruby-Paket auf Ihrem Rechner installieren können. Anschließend unternehmen Sie Ihre ersten Schritte mit der Sprache und ihren Werkzeugen.

Etwas Ruby-Geschichte

Bereits als Informatikstudent machte sich der japanische Programmierer Yukihiro Matsumoto (genannt »Matz«) in den 1980er Jahren allerlei Gedanken über Programmiersprachen, speziell über Skriptsprachen. Im Jahr 1993 begann er dann mit der Entwicklung seiner eigenen Sprache. Er wählte den Namen *Ruby* (Rubin) – ein Edelstein, der entsprechend einen deutlichen Bezug zu *Perl* bildet. Erst im Nachhinein bemerkte er, dass der Rubin als Geburtsstein[1] des Monats Juli direkt auf die Perle (Juni) folgt, was interessant ist, weil Ruby einige Perl-Konzepte weiterentwickelt.

1 Siehe zum Beispiel *http://en.wikipedia.org/wiki/Birthstone#Birthstones*.

Bei der Entwicklung von Ruby ließ sich Matz durch mancherlei Vorbilder inspirieren – insbesondere durch andere Skriptsprachen: Perl (wie bereits erwähnt), besonders bezüglich der Möglichkeit einer knappen Ausdrucksweise, aber auch *Python*, was die im Vergleich zu Perl klarere Syntax mit weniger kryptischen Sonderzeichen erklärt. Im Übrigen entwickelte er Ruby so konsequent objektorientiert, wie es nach dem Klassiker *Smalltalk* aus den 1970er Jahren kaum eine Sprache mehr war. *C++* beispielsweise, die wohl verbreitetste objektorientierte Sprache, macht zugunsten der Abwärtskompatibilität mit dem nicht objektorientierten *C* eine Menge Kompromisse, und selbst das modernere *Java* weist diverse Inkonsequenzen auf *Objektorientierung* ist übrigens kurz gesagt eine Technik, die die programmiertechnische Nachbildung von Gegenständen und Prozessen aus der realen Welt extrem erleichtert. In Kapitel 4 wird genau erläutert, wie es funktioniert.

1995 veröffentlichte Matz die Sprache als Open Source Software. Sie steht unter der GNU General Public License (GPL), die auch so wichtige Software wie das Betriebssystem Linux oder den Datenbankserver MySQL schützt. In Japan verbreitete Ruby sich rasch und ist seither recht beliebt und mindestens so verbreitet wie Python in den USA und Europa.

Die internationale Verbreitung von Ruby begann ganz allmählich im Jahr 2000, als das erste englischsprachige Buch über die Sprache erschien: das wegen seines Covers »Pickaxe Book« (Spitzhacken-Buch) genannte *Programming Ruby* von Dave Thomas (Pragmatic Programmers). Dieses Buch liegt inzwischen in zweiter Auflage vor; in Anhang B finden Sie bibliografische Angaben sowie die Adresse der Buch-Website.

Ruby wird bis heute stetig weiterentwickelt. Außer Matz arbeitet eine über die ganze Welt verteilte, sehr aktive Community an der Sprache, an zahllosen Erweiterungen und an der Dokumentation. Die Koordination der verschiedenen Arbeitsgruppen erfolgt über das Internet; die Adressen der wichtigsten Websites und Mailinglisten stehen in Anhang B. Die meisten Mitglieder dieser Gruppen sind freundlich und beantworten bereitwillig Fragen, auch von Einsteigern. Achten Sie aber darauf, dass Sie zunächst in den FAQs oder Archiven der betreffenden Foren und Mailinglisten nachsehen, ob Ihre Frage nicht bereits beantwortet wurde.

Erst in den letzten zwei Jahren erhielt Ruby einen neuen, ungeahnten Popularitätsschub: Es sprach sich unter Webentwicklern schnell herum, dass das Framework *Ruby on Rails* die schnelle Entwicklung robuster und sicherer Webanwendungen ermöglicht (mehr darüber lesen Sie in Kapitel 7). Auch die benutzerfreundlichen, reaktionsschnellen Ajax- und Web-2.0-Anwendungen profitieren von Ruby on Rails. Um mit Rails vernünftig arbeiten zu können, empfiehlt es sich, zunächst die zugrunde liegende Sprache Ruby zu erlernen. Insofern ist das vorliegende Buch auch eine ideale Vorbereitung für angehende Rails-Entwickler.

Merkmale und Vorzüge von Ruby

Ohne technische Details vorwegzunehmen, wird in diesem kurzen Abschnitt das Wesen der Programmiersprache Ruby vorgestellt. Falls Sie noch nie programmiert haben, können Sie mit manchen Begriffen vielleicht nichts anfangen – aber keine Sorge, im Laufe dieses Buches wird alles Schritt für Schritt eingeführt.

Ruby ist eine *interpretierte Skriptsprache*. Das bedeutet, dass ein Ruby-Programm nicht direkt vom Prozessor eines Computers ausgeführt wird. Stattdessen wird zunächst der *Ruby-Interpreter* geladen. Dieses spezielle Programm öffnet die Ruby-Dateien, liest sie Zeile für Zeile und wandelt die Ruby-Anweisungen in Code um, den der Prozessor verstehen kann. Das Gegenteil von den Interpreter-Sprachen sind kompilierte Sprachen. Bei diesen kommt ein Übersetzungsprogramm namens *Compiler* zum Einsatz, das den gesamten Programmcode in die Maschinensprache des jeweiligen Prozessors überträgt und dann dauerhaft als selbstständig lauffähiges Programm abspeichert.

Kompilierte Programme werden zweifellos schneller ausgeführt als interpretierte (wobei die konkrete Geschwindigkeit heutzutage stark von der Qualität der zugrunde liegenden Programmbibliotheken abhängt). Hier stellt sich allerdings die Frage des Anwendungszwecks – ein 3-D-Action-Spiel muss fast immer kompiliert werden, um ruckelfrei zu laufen. Bei Anwendungen, die auf langsamen Aspekten wie Benutzereingaben oder Netzwerkverbindungen basieren, macht es dagegen kaum einen Unterschied, ob Sie eine Compiler- oder eine Skriptsprache wählen.

Ein weiteres Problem besteht darin, dass der Interpreter auf jedem Rechner installiert werden muss, auf dem die in der betreffenden Sprache geschriebenen Skripten ausgeführt werden sollen. Kompilierte Software ist dagegen ohne weitere Voraussetzungen auf jedem Rechner lauffähig, der denselben Prozessor und dasselbe Betriebssystem nutzt wie der Computer, auf dem sie kompiliert wurde.

Neben diesen offensichtlichen Nachteilen besitzen Skriptsprachen aber auch einige wichtige Vorteile. Beispielsweise lässt sich ein Interpreter leichter auf verschiedene Plattformen portieren als ein Compiler, so dass Ihre Programme tendenziell auf mehr unterschiedlichen Betriebssystemen laufen – oft sogar ohne jegliche Änderungen, weil die Interpreter-Versionen für die verschiedenen Plattformen die Unterschiede bereits berücksichtigen.

Konkret ist Ruby (mindestens) für folgende Nicht-Unix-Plattformen verfügbar:

- Alle Windows-Versionen ab Windows 95 und NT 4.0
- MS-DOS, und damit auch die klassischen Windows-Versionen bis 3.11
- Klassisches Mac OS bis Version 9
- IBM OS/2

Im Unix-Bereich werden unter anderem folgende Betriebssysteme unterstützt:

- Linux (alle Distributionen; viele liefern sogar eine Version mit)
- Mac OS X (enthält Ruby ebenfalls ab Werk)
- FreeBSD, OpenBSD und NetBSD
- Sun Solaris
- ... und viele andere Unix-Varianten

Ein weiterer Vorteil von Interpreter-Sprachen betrifft die Programmierung selbst: Da die Skripten jeweils direkt aus dem Quellcode ausgeführt werden, lassen sie sich schneller testen – Sie müssen nicht bei jeder Änderung auf einen Compiler warten. Fehlermeldungen beziehen sich ebenfalls direkt auf ihren Code, so dass Sie Fehler leichter beheben können.

Ruby besitzt eine klare und gut verständliche Sprachsyntax mit wenigen Ausnahmen und Sonderregeln. Der Sprachentwurf folgt dem sogenannten *Principle of Least Surprise* (auf Deutsch: Prinzip der geringsten Überraschung). Die Sprache sollte sich also für einigermaßen erfahrene Programmierer in der Regel so verhalten wie erwartet. Das liegt unter anderem daran, dass die Objektorientierung in Ruby absolut konsequent durchgehalten wird. Es würde zu weit führen, an dieser Stelle näher darauf einzugehen – lassen Sie sich auf den nachfolgenden Seiten einfach Schritt für Schritt von der Eleganz und Leistungsfähigkeit der Ruby-Syntax bezaubern. Bereits im nächsten Kapitel werden Sie Ihre ersten nützlichen Programme schreiben; der Einstieg gestaltet sich bei Ruby wesentlich leichter als bei vielen anderen Sprachen.

Ruby installieren

Bevor Sie mit Ruby arbeiten können, müssen Sie den Interpreter und die zugehörigen Werkzeuge und Bibliotheken auf Ihrem Rechner installieren. Dieser Abschnitt beschreibt, wie Sie das unter Windows sowie auf Linux- und anderen Unix-Systemen erledigen können.

Installation unter Windows

Als Windows-Benutzer sind Sie bei der Ruby-Installation klar im Vorteil: Für Sie gibt es ein komplettes Installer-Paket, das heißt eine einzige Datei, die Sie per Doppelklick installieren können. Sie enthält unter anderem folgende Bestandteile:

- Den Ruby-Interpreter
- Die Kernbibliotheken
- Zahlreiche populäre Erweiterungen
- Den Texteditor SciTE, der unter anderem Syntax-Highlighting für Ruby zu bieten hat

- Die grafische Ruby-Entwicklungsumgebung FreeRIDE
- Umfangreiches Dokumentationsmaterial in den Formaten HTML und Windows Help, darunter den Volltext des oben erwähnten »Pickaxe-Buchs« *Programming Ruby – The Pragmatic Programmers' Guide* (1. Auflage)

Sie finden den Installer unter *http://rubyinstaller.rubyforge.org*. Klicken Sie den Link *Downloads* an und laden Sie sich die jeweils neueste Version herunter (Stand Januar 2007: *1.8.5-21 stable*). Die Datei mit der Endung *.exe* (zurzeit *ruby185-21.exe*) ist der Installer selbst.

Zu jeder *.exe*-Datei gehört eine gleichnamige *.md5*-Datei, etwa *ruby185-21.md5*. Diese Datei enthält einen sogenannten MD5-Hash, eine nach einem komplexen Verfahren berechnete Prüfsumme. Damit können Sie selbstständig die Integrität Ihres Downloads überprüfen, um Fehler auszuschließen – Sie brauchen nur den MD5-Hash der *.exe*-Datei zu erstellen und mit der herunterladbaren *.md5*-Datei zu vergleichen.

Windows enthält ab Werk kein md5-Tool. Sie können aber einfach eins aus dem Web herunterladen – ein gutes Beispiel sind die Win32-GNU-Utilities von der Website *http://unxutils.sourceforge.net*[2], die noch weitere Windows-Versionen verbreiteter Unix-Utilities enthalten. Laden Sie die ZIP-Datei herunter und entpacken Sie sie am besten in ein Verzeichnis, das zum *PATH* ausführbarer Konsolenprogramme gehört – zum Beispiel das Systemverzeichnis, normalerweise *C:\Windows*. Danach können Sie die Eingabeaufforderung öffnen (siehe den Abschnitt »'Hello World, hello Ruby' – das erste Ruby-Programm«), in Ihr Ruby-Download-Verzeichnis (zum Beispiel *Desktop* oder *Eigene Dateien*) wechseln und *md5sum* mit dem Namen der *.exe*-Datei als Argument eingeben. Hier ein Beispiel:

```
> md5sum ruby185-21.exe
8830dfeb25e39fd7a5cccfbd02030337 *ruby185-21.exe
```

Vergleichen Sie die Ausgabe mit der *.md5*-Datei, die sich entweder direkt im Browser öffnet oder aber nach dem Download mit jedem beliebigen Texteditor betrachten lässt. Wenn die beiden Werte nicht übereinstimmen, müssen Sie den Installer erneut herunterladen.

Die eigentliche Installation können Sie vornehmen, indem Sie auf die *.exe*-Datei doppelklicken. Unter Windows-Versionen ab XP, Service Pack 2, erscheint zunächst eine Sicherheitswarnung, da der Installer keinen »offiziellen Segen« von Microsoft besitzt – klicken Sie auf *Ausführen*. Erst danach geht es mit dem eigentlichen *Ruby Setup Wizard* los, wobei Sie nacheinander folgende Screens zu sehen bekommen:

2 Bitte aufpassen, es heißt tatsächlich *unxutils* und nicht etwa *unixutils*.

1. *Welcome* – Information, dass Ruby installiert werden soll. Klicken Sie hier, wie auch zum Bestätigen der nachfolgenden Seiten, auf *Next*.
2. *License Agreement* – in einer Textbox mit Scrollbalken erscheinen einige Hinweise zur installierten Version sowie zur Lizenz. Ruby steht unter der GNU General Public License (GPL), der verbreitetsten Open Source-Lizenz. Sie erlaubt Ihnen, die Software in jeder Hinsicht frei zu nutzen, zu verändern und (unter derselben Lizenz) weiterzugeben, und soll vor allem verhindern, dass kommerzielle Softwarefirmen sich den Code exklusiv zu eigen machen. Insofern können Sie wohl guten Gewissens auf *I Agree* klicken.
3. *Choose Components* – wählen Sie aus, welche Bestandteile Sie installieren möchten. Dieser Schritt wird in Abbildung 1-1 gezeigt. Folgende Optionen stehen zur Verfügung:
   - *Ruby* (die Sprache selbst; nicht abwählbar)
   - *SciTE* (der Texteditor)
   - *FreeRIDE* (die Entwicklungsumgebung)
   - *Enable RubyGems* – aktiviert den Ruby-Erweiterungs-Manager RubyGems
   - *European Keyboard* – Tastatur-Anpassung für Interactive Ruby (IRB)

   Auf einem halbwegs aktuellen Rechner dürfte es kein Problem sein, alle Komponenten auszuwählen. Links unten teilt Ihnen die Angabe *Space required* mit, wie viel Festplattenspeicher Ihre aktuelle Auswahl benötigt (die Vollinstallation belegt in der aktuellen Version knapp 100 Megabyte).

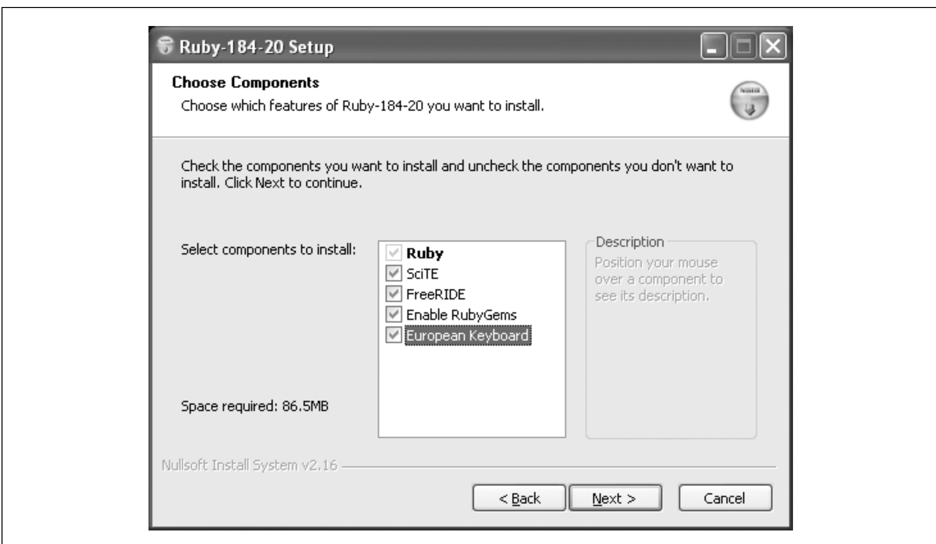

Abbildung 1-1: Ruby-Installation unter Windows: Auswahl der Komponenten

4. *Choose Install Location* – wählen Sie das Verzeichnis, in dem Ruby installiert werden soll. In der Regel dürfte der automatische Vorschlag *C:\ruby* in Ordnung sein. Andernfalls können Sie auf *Browse* klicken und einen anderen Ordner wählen.
5. *Choose Start Menu Folder* – hier können Sie einstellen, unter welchem Eintrag Ruby im Windows-Startmenü erscheinen soll. Normalerweise ist *Ruby-<Versionsnr.>*, wie vorgeschlagen, in Ordnung – es sei denn, Sie haben Ihre eigene Ordnung im Startmenü.

Damit sind alle Informationen vollständig. Klicken Sie auf *Install*, um die eigentliche Installation durchzuführen, oder auf *Back*, um letzte Korrekturen durchzuführen.

Nach der Installation sollten Sie prüfen, ob Ruby funktioniert. Begeben Sie sich dazu in die Eingabeaufforderung, indem Sie der Anleitung im Abschnitt »Zu Unrecht gefürchtet: das Arbeiten mit der Konsole« auf Seite 10 folgen. Der Ruby-Interpreter heißt unter Windows *ruby.exe*, wobei Sie die Endung *.exe*, wie bei Befehlen in der Eingabeaufforderung üblich, weglassen können. Mit der Option `-e` können Sie unmittelbar einzelne Anweisungen ausführen. Geben Sie Folgendes ein, um sich zum ersten Mal von Ruby begrüßen zu lassen (das klappt aus jedem beliebigen Arbeitsverzeichnis, da der Installer Ihren `PATH` ausführbarer Programme anpasst):

```
> ruby -e "puts 'Hallo, hier ist Ruby.'"
Hallo, hier ist Ruby.
```

Wenn Sie die obige Ausgabe erhalten, hat die Installation funktioniert, und Sie können im nächsten Abschnitt Ihr erstes richtiges Programm schreiben.

Installation unter Linux und anderen Unix-Systemen

Die Ruby-Entwickler selbst geben keine fertigen Ruby-Binärpakete für Linux und andere Unix-Systeme heraus. Allerdings ist Ruby ab Werk in den meisten modernen Linux-Distributionen, aber auch in Mac OS X oder FreeBSD, enthalten, so dass Sie es nur in Ausnahmefällen selbst kompilieren müssen. Überprüfen Sie also als Erstes, ob Ruby bereits installiert ist. Öffnen Sie dazu ein Terminal[3] und geben Sie Folgendes ein:

```
$ ruby -v
```

Wenn keine Fehlermeldung erscheint, sondern eine Versionsnummer ab 1.8.x, dann haben Sie bereits die passende Ruby-Version für die Arbeit mit diesem Buch und können im nächsten Abschnitt mit dem ersten Beispiel weitermachen. Andernfalls sollten Sie die Paketverwaltung Ihrer Distribution oder die Website Ihres Systemanbieters durchforsten und nachschauen, ob eine aktuelle Ruby-Version verfügbar ist.

---

3 Falls Sie mit der Verwendung von Terminals nicht vertraut sind, finden Sie weiter unten im Abschnitt »Zu Unrecht gefürchtet: das Arbeiten mit der Konsole« auf Seite 10 einige Hinweise dazu.

Sollte es für Ihre Distribution gar kein Ruby oder keine ausreichend neue Version geben, können Sie die Sprache immer noch leicht selbst kompilieren. Beachten Sie, dass Sie dazu die GNU-Entwicklungswerkzeuge GCC und make benötigen, die bei einigen gängigen Linux-Distributionen – zum Beispiel SUSE Linux – nicht automatisch installiert werden. Installieren Sie sie also zunächst gegebenenfalls mit dem Paketmanager Ihrer Distribution nach.

Als Nächstes können Sie sich das aktuellste Quellcode-Paket herunterladen. Besuchen Sie dazu die Site *http://www.ruby-lang.org* und folgen Sie den Download-Links. Dort erhalten Sie ein Paket namens *ruby-1.8.x.tar.gz* (zurzeit *ruby-1.8.5.tar.gz*). Laden Sie es herunter und überprüfen Sie auch hier – wie weiter oben für Windows erläutert – die MD5-Prüfsumme. Danach können Sie das Paket entpacken:

```
$ tar xzvf ruby.tar.gz
```

Bei einigen exotischeren Unix-Varianten, die kein GNU-`tar` enthalten, müssen Sie dafür allerdings zwei Befehle eingeben:

```
$ gunzip ruby.tar.gz
$ tar -xvf ruby.tar
```

Danach können Sie jedenfalls in das neu erstellte Unterverzeichnis *ruby-1.8.x* wechseln. Zum Beispiel:

```
$ cd ruby-1.8.5
```

Dort muss zunächst das Shell-Skript `configure` ausgeführt werden, das die Makefiles für Ihre konkrete Systemplattform anpasst. Anschließend wird `make` für die eigentliche Kompilierung und `make install` für die Installation ins gewünschte Verzeichnis verwendet.

`configure` kennt zahlreiche Optionen für spezielle Installationswünsche. Geben Sie Folgendes ein, um sie zu lesen:

```
$ ./configure --help |less
```

(Bei manchen älteren UNIX-Arten müssen Sie `more` statt `less` eingeben.) Blättern Sie mit den Pfeiltasten in der Beschreibung und drücken Sie **Q**, sobald Sie alles Nötige gelesen haben. Anschließend können Sie `configure` mit den gewünschten Optionen aufrufen. Das folgende Beispiel bereitet die Makefiles zur Installation des kompletten Ruby-Verzeichnisbaums unter */usr/local/ruby* vor:

```
$ ./configure --prefix=/usr/local/ruby
```

Nun können `make` und `make install` aufgerufen werden; zumindest Letzteres müssen Sie als `root` erledigen, weil gewöhnliche Benutzer keinen Schreibzugriff auf Verzeichnisse unter */usr* besitzen:

```
$ make
$ su
Password: [root-Passwort; Eingabe wird nicht angezeigt]
# make install
```

Auf einem Computer, auf dem Sie keine `root`-Rechte besitzen, können Sie Ruby alternativ in einem Verzeichnis unter Ihrem Home-Verzeichnis installieren. Geben Sie dazu beim `configure`-Aufruf das entsprechende Präfix an. Zum Beispiel:

```
$ ./configure --prefix=/home/meinname/ruby
```

Für den ersten Test nach der Installation müssen Sie bei dieser selbstkompilierten Variante in das *bin*-Verzeichnis von Ruby wechseln:

```
$ cd /usr/local/ruby/bin
```

Nun können Sie folgende Zeile eingeben, um eine Ruby-Anweisung auszuführen, die Sie begrüßt:

```
$ ./ruby -e "puts 'Hallo, hier ist Ruby!'"
```

Auf Dauer sollten Sie das *bin*-Verzeichnis zum Pfad ausführbarer Programme (der Umgebungsvariablen `PATH`) hinzufügen, damit Sie Ruby und seine Hilfsprogramme aus jedem Arbeitsverzeichnis aufrufen können. Die Konfigurationsdateien, in die solche Änderungen permanent eingetragen werden, unterscheiden sich je nach konkretem System und je nach Distribution. Bei gängigen Linux-Distributionen wird */etc/profile* für alle Shells und alle Benutzer verwendet, während */home/Username/.bashrc* beispielsweise für die bash-Shell und nur für den jeweiligen Benutzer gilt.

Nachdem Sie sich für eine Konfigurationsdatei entschieden haben, können Sie sie in einem Editor öffnen. Suchen Sie eine Zeile, in der `PATH` einen Wert erhält, das heißt so etwas wie

```
export PATH=/pfad/nr/1:/pfad/nr/2:...
```

oder:

```
export PATH=$PATH:/weiterer/pfad:/noch/ein/pfad
```

Fügen Sie einen weiteren Doppelpunkt und Ihren Ruby-`bin`-Pfad hinzu, also etwa `:/usr/local/ruby/bin`.

Falls die gewählte Datei gar keine `PATH`-Definition enthält, können Sie eine Zeile wie die folgende verwenden, die Ihren bisherigen Suchpfad um das Ruby-Verzeichnis ergänzt:

```
export PATH=$PATH:/usr/local/ruby/bin
```

Beachten Sie, dass die hier gezeigte Syntax für die bash gilt; wenn Sie eine andere Shell verwenden, müssen Sie stattdessen die für diese geltenden Kommandos zur Variablendefinition einsetzen.

Nachdem Sie die Konfigurationsdatei gespeichert haben, gelten die Änderungen, sobald Sie ein neues Terminalfenster öffnen. Probieren Sie es aus, indem Sie sich die obige Begrüßung noch einmal aus einem anderen Arbeitsverzeichnis heraus und ohne vorangestelltes `./` ausgeben lassen. Zum Beispiel:

```
user@rechner: ~ > ruby -e "puts 'Hallo, hier ist noch mal Ruby!'"
```

Ruby-Code eingeben und ausführen

Ruby ist eine Programmiersprache und kein gewöhnliches Anwendungsprogramm. Das bedeutet, dass Sie nicht einfach ein Anwendungsfenster öffnen, um mit Ruby zu arbeiten. Stattdessen besteht die Sprache aus mehreren kleinen Einzelprogrammen. Diese besitzen keine grafische Oberfläche, sondern werden jeweils durch eine Befehlseingabe in einer Textkonsole ausgeführt. In diesem kurzen Abschnitt lernen Sie die wichtigsten Möglichkeiten zum Ausführen von Ruby-Anweisungen oder -Skripten kennen.

Zu Unrecht gefürchtet: das Arbeiten mit der Konsole

Bevor Sie die Ruby-Dienstprogramme verwenden können, müssen Sie ein Programm zur Befehlseingabe öffnen. Dies ist unter Windows die bereits erwähnte Eingabeaufforderung und auf UNIX-Systemen ein beliebiges Terminalfenster. Zum Öffnen der Windows-Eingabeaufforderung gibt es zwei Möglichkeiten:

- Wählen Sie *Start* → *Alle Programme* → *Zubehör* → *Eingabeaufforderung*. Beachten Sie, dass es in älteren Windows-Versionen vor XP *Programme* statt *Alle Programme* heißt.
- Alternativ können Sie *Start* → *Ausführen* wählen, `cmd` eingeben und *OK* anklicken beziehungsweise **Enter** drücken.

Wie Sie in Ihrem UNIX-artigen System ein Terminalfenster öffnen, ist je nach Distribution, Version und grafischer Oberfläche verschieden. Hier nur einige Beispiele:

- Der beliebte Desktop KDE für Linux und einige andere UNIX-Varianten enthält ein komfortables Terminalprogramm namens *Konsole*, das Sie in der Regel über ein Bildschirm-Symbol im Panel (die Leiste am unteren Bildschirmrand) öffnen können.
- Der andere verbreitete Desktop, GNOME, besitzt ebenfalls eine eigene Terminalemulation, die einfach *GNOME Terminal* heißt. Sie öffnen es am einfachsten, indem Sie mit der rechten Maustaste eine leere Stelle auf dem Desktop anklicken und *Terminal öffnen* aus dem Kontextmenü wählen.
- Unter Mac OS X befindet sich das *Terminal* im Systemordner *Applications*. Wenn Sie ernsthaft mit dem Programmieren beginnen, werden Sie es öfter benötigen und sollten es daher ins Dock ziehen.

Beachten Sie, dass innerhalb von UNIX-Terminalfenstern unterschiedliche *Shells* (Befehls-Interpreter) ausgeführt werden können, wodurch sich die Syntax mancher Eingaben etwas ändert. In den drei genannten Beispielfällen ist es so gut wie immer die `bash`. Soweit die Arbeit mit der Shell in diesem Abschnitt geschildert wird, macht dies aber keinen Unterschied.

Nachdem Sie die jeweilige Konsole geöffnet haben, wird ein *Prompt* (die eigentliche Eingabeaufforderung) angezeigt. Windows-Rechner verwenden standardmäßig die Schreibweise `Arbeitsverzeichnis>`. Zum Beispiel:

```
C:\Dokumente und Einstellungen\Sascha\Eigene Dateien>
```

Bei UNIX-Systemen kann der Prompt sehr unterschiedlich aussehen. Recht häufig ist die Form `Username@Rechner:Arbeitsverzeichnis >`. Das eigene Home-Verzeichnis, meist `/home/Username`, wird dabei in der Regel durch ~ abgekürzt. Somit sieht der gesamte Prompt beispielsweise wie folgt aus:

```
sascha@linuxbox:~ >
```

Wenn Sie als `root` arbeiten, wird meistens kein Username angezeigt, und das Schlusszeichen wechselt von `>` oder `$` zu einer Raute (`#`). Zum Beispiel:

```
linuxbox:/home/sascha #
```

In diesem Abschnitt und im Rest dieses Buchs werden normalerweise (solange der *konkrete* Prompt keine Rolle spielt) folgende Zeichen verwendet, um den Prompt zu kennzeichnen:

- >: Windows-Prompt sowie allgemeiner Prompt, wenn eine Eingabe für alle Betriebssysteme gilt
- $: UNIX-Prompt; beliebiger Benutzer einschließlich `root` (wobei Sie normale Aufgaben aus Sicherheitsgründen nicht als `root` erledigen sollten)
- #: UNIX-Prompt für `root`

Wenn Sie zum ersten Mal in einer Konsole arbeiten, werden Sie einige grundlegende Befehle benötigen. Diese betreffen vor allem den Umgang mit Verzeichnissen, wie etwa den Wechsel des Arbeitsverzeichnisses oder das Anlegen neuer Unterverzeichnisse. Hierbei spielt die unterschiedliche Organisation des Dateisystems, also die Verzeichnishierarchien, eine wichtige Rolle:

- Auf Windows-Rechnern beginnen Dateipfade mit einem Laufwerksbuchstaben, darauf folgen die ineinander verschachtelten Verzeichnisnamen und zum Schluss der Dateiname. Das Trennzeichen ist ein Backslash (\), der auf einer deutschen Windows-Tastatur mit **Alt Gr + ß** erzeugt wird. Das folgende Beispiel ist der Pfad der Ruby-Datei *hello.rb* im Ordner *myruby* unter dem »Privatverzeichnis« des Users Sascha:

  ```
  C:\Dokumente und Einstellungen\Sascha\Eigene Dateien\myruby\hello.rb
  ```

- UNIX-Systeme kennen keine Laufwerksbuchstaben. Das Dateisystem besitzt eine gemeinsame Wurzel namens `/`, wobei sich die diversen Standardverzeichnisse auf verschiedenen Laufwerken oder Partitionen befinden können – die genaue Anordnung wird durch Konfigurationsdateien geregelt. Als Trennzeichen zwischen Unterverzeichnissen sowie zwischen Verzeichnis und Datei dient dabei der Slash (`/`). Die UNIX-Entsprechung des oben gezeigten Windows-Pfades wäre daher:

```
/home/sascha/myruby/hello.rb
```

Beachten Sie, dass UNIX bei Datei- und Verzeichnisnamen zwischen Groß- und Kleinschreibung unterscheidet, Windows aber nicht. Tun Sie sich zur Sicherheit selbst einen Gefallen und schreiben Sie konsequent alles klein. Ebenso sollten Sie in allen selbst gewählten Namen Leerzeichen und Sonderzeichen (außer dem Unterstrich _) vermeiden. Spätestens wenn Sie Ihre Dateien im Web publizieren, kann es sonst zu Problemen kommen.

Manche Konsolenbefehle können mit Platzhaltern umgehen, die auf mehrere Dateien zutreffen. Dabei steht ein * für beliebig viele beliebige Zeichen und ein ? für genau ein beliebiges Zeichen. Auch hier gibt es einen kleinen Plattformunterschied: In UNIX-Systemen steht * für alle Dateien in einem Verzeichnis, bei Windows dagegen `*.*`, weil die Dateiendung gesondert betrachtet wird.

Um das aktuelle Arbeitsverzeichnis zu wechseln, verwenden sowohl UNIX als auch Windows das Kommando `cd` (kurz für »change directory«). Unter Windows ist dieser Befehl nicht dafür zuständig, um das Laufwerk zu wechseln. Dies geschieht durch die einfache Eingabe des Laufwerksbuchstaben mit nachfolgendem Doppelpunkt. Das folgende Beispiel vollzieht einen Wechsel auf die Festplatte `C:`:

```
> C:
```

Mit `cd` können Sie unter Windows innerhalb eines Laufwerks einen absoluten, das heißt vollständigen Pfad angeben, um in das entsprechende Verzeichnis zu wechseln. Wenn Verzeichnis- oder Dateinamen Leerzeichen enthalten, müssen Sie diese (oder wahlweise den gesamten Pfad) in Anführungszeichen setzen. Das folgende Beispiel wechselt – aus einem beliebigen Verzeichnis auf der Festplatte `C:` – in das Verzeichnis *Eigene Dateien* des Benutzers Sascha:

```
> cd "\Dokumente und Einstellungen\Sascha\Eigene Dateien"
```

Bei UNIX-Systemen funktioniert der Verzeichniswechsel per absolutem Pfad im Prinzip genauso. Zum Beispiel:

```
$ cd /home/sascha
```

Wenn Sie vom aktuellen Verzeichnis aus in ein untergeordnetes Verzeichnis wechseln möchten, müssen Sie den Namen dieses Unterverzeichnisses ohne führenden Backslash beziehungsweise Slash angeben. Hier ein Windows-Beispiel:

```
C:\Dokumente und Einstellungen\Sascha\Eigene Dateien> cd myruby
C:\Dokumente und Einstellungen\Sascha\Eigene Dateien\myruby>
```

Auf diese Weise lassen sich auch mehrere Hierarchiestufen überwinden. Dazu sehen Sie hier ein UNIX-Beispiel:

```
sascha@linuxbox:~ > cd myruby/kapitel1
sascha@linuxbox:~/myruby/kapitel1 >
```

Um in das übergeordnete Verzeichnis zu wechseln, wird auf beiden Plattformen der spezielle Verzeichnisname `..` verwendet. Zum Beispiel (unter Windows):

```
C:\Dokumente und Einstellungen\Sascha\Eigene Dateien> cd ..
C:\Dokumente und Einstellungen\Sascha>
```

Diese Techniken lassen sich kombinieren, um über sogenannte relative Pfade von jedem beliebigen Verzeichnis in jedes andere zu wechseln. Das folgende Beispiel vollzieht auf einem UNIX-Rechner einen Wechsel aus dem oben gezeigten Verzeichnis *kapitel1* in das »Geschwister-Verzeichnis« *kapitel2*:

```
sascha@linuxbox:~/myruby/kapitel1 > cd ../kapitel2
sascha@linuxbox:~/myruby/kapitel2 >
```

Um unterhalb des aktuellen Arbeitsverzeichnisses ein neues Verzeichnis zu erstellen, wird das Kommando `mkdir` verwendet (Windows erlaubt auch die Kurzfassung `md`). Hier wird beispielsweise schon einmal das Projektverzeichnis für Kapitel 3 vorbereitet:

```
sascha@linuxbox:~/myruby > mkdir kapitel3
```

Beachten Sie, dass Sie auf einer UNIX-Maschine nur innerhalb Ihres eigenen Home-Verzeichnisses neue Verzeichnisse erstellen dürfen. In anderen Bereichen des Dateisystems darf dies nur der Superuser `root`. Geben Sie `su` und das `root`-Passwort ein, wenn Sie vorübergehend als `root` arbeiten müssen, und `exit`, sobald Sie damit fertig sind.

Als Nächstes sollten Sie noch das Kommando kennen, mit dem Sie sich den Inhalt des aktuellen Verzeichnisses ausgeben lassen können. Unter Windows heißt es `dir`:

```
> dir
```

Auf UNIX-Rechnern lautet der Befehl dagegen `ls`. Wenn Sie die Option `-l` hinzufügen, erhalten Sie ausführliche Informationen über jede Datei – beispielsweise den Eigentümer, die Zugriffsrechte und die Größe:

```
$ ls -l
```

Um den Überblick zu behalten, ist es manchmal nützlich, den Fensterinhalt zu löschen und den Prompt wieder nach links oben zu setzen. Geben Sie dazu in der Windows-Eingabeaufforderung Folgendes ein:

```
> cls
```

In den meisten UNIX-Terminals lautet der Befehl dagegen:

```
$ clear
```

Noch praktischer ist, dass Sie bei fast allen UNIX-Varianten einfach **Strg + L** drücken können, um denselben Effekt zu erzielen.

Tabelle 1-1 stellt die wichtigsten Konsolenbefehle für beide Plattformen noch einmal gegenüber, wobei noch einige zusätzliche Anweisungen hinzukommen.

Tabelle 1-1: Die wichtigsten Konsolenbefehle für Windows und UNIX

Gewünschte Wirkung	Windows-Befehl	UNIX-Befehl
Laufwerk wechseln	*Laufwerksbuchstabe*`:`, z.B. `C:` oder `F:`	–
Arbeitsverzeichnis wechseln – absoluter Pfad	`cd \Verz.[\Unterv.\...]`	`cd /Verz.[/Unterv./...]`
In Unterverzeichnis des aktuellen Arbeitsverzeichnisses wechseln	`cd Verz.[\Unterv.\...]`	`cd Verz.[/Unterv./...]`
In übergeordnetes Verzeichnis wechseln	`cd ..`	`cd ..`
In das eigene Home-Verzeichnis wechseln	–	`cd ~`
Neues Verzeichnis erstellen	`mkdir` *Name* `md` *Name*	`mkdir` *Name*
Inhalt des aktuellen Verzeichnisses anzeigen	`dir`	`ls` (ausführlich: `ls -l`)
Datei löschen	`del` *Name*	`rm` *Name*
Datei kopieren	`copy` *AltName NeuName*	`cp` *AltName NeuName*
Platzhalter: alle Dateien im aktuellen Verzeichnis	`*.*`	`*`
Bildschirm löschen	`cls`	`clear` (oft auch **Strg + L**)

Ruby-Anweisungen und -Skripten eingeben

Wenn Sie sich erst einmal auf der Konsole zu Hause fühlen, gibt es verschiedene Möglichkeiten, Ruby-Code auszuführen. Die interessanteste Variante für einen praktischen Einstieg ist die interaktive Ruby-Shell `irb` (*Interactive Ruby*). Wenn Sie Ruby gemäß der Anleitung in diesem Kapitel installiert haben, genügt folgende Eingabe auf der Konsole, um dieses Programm zu starten:

```
> irb
```

Nun können Sie beliebige Ruby-Anweisungen eingeben. Ein praktischer Vorteil von `irb` besteht darin, dass nicht einmal Ausgabebefehle nötig sind, weil der Wert von Ausdrücken jeweils sofort angezeigt wird. Auf diese Weise können Sie das Programm etwa als einfachen Taschenrechner nutzen. Geben Sie als erstes Beispiel etwa Folgendes ein:

```
irb(main):001:0> 3+7
=> 10
```

Auf diese Weise können Sie beliebige Rechenausdrücke eingeben, wobei die Grundrechenarten durch folgende Zeichen dargestellt werden: `+` (Addition), `-` (Subtraktion), `*` (Multiplikation) und `/` (Division). Probieren Sie einfach ein paar Möglichkeiten aus. Hier einige unverbindliche Beispiele:

```
irb(main):002:0> 7*6
=> 42
irb(main):003:0> 76-53
=> 23
irb(main):004:0> 5/2
=> 2
```

Das Ergebnis der Division scheint allerdings nicht korrekt zu sein. Das Problem besteht darin, dass ganze Zahlen und Fließkommazahlen verschiedene Datentypen sind – 2 ist das korrekte Ergebnis der ganzzahligen Division. Versuchen Sie es für die mathematisch korrekte Lösung mit:

```
irb(main):005:0> 5.0/2
=> 2.5
```

Fließkommazahlen müssen in Ruby – wie in jeder Programmiersprache – gemäß der englischen Schreibweise mit einem Punkt (.) statt mit Komma geschrieben werden. Es heißt also beispielsweise `2.5` statt `2,5`.

Eine weitere interessante Entdeckung ist der Unterschied zwischen mathematischen Ausdrücken und Textausdrücken. Geben Sie dazu Folgendes ein:

```
irb(main):006:0> "3+7"
=> "3+7"
```

Das Ergebnis ist und bleibt der Text `"3+7"`; es findet keine arithmetische Berechnung statt. Hier noch eine interessante Abwandlung:

```
irb(main):007:0> "3"+"7"
=> "37"
```

Wenn Sie Texte (in Anführungszeichen) durch `+` verbinden, werden diese also verkettet und nicht etwa numerisch addiert – selbst dann nicht, wenn ausschließlich Ziffern enthalten sind.

Wenn Sie `irb` nach einigen weiteren Experimenten beenden möchten, können Sie `exit` eingeben:

```
irb(main):008:0> exit
```

Eine andere einfache Möglichkeit, eine einzelne Ruby-Anweisung auszuführen, besteht darin, den Ruby-Interpreter (`ruby`) mit der Option `-e` aufzurufen. Dabei müssen Sie allerdings Ausgabebefehle – am einfachsten `puts` – verwenden, um Text auszugeben. Das folgende Beispiel führt ebenfalls eine Berechnung aus:

```
> ruby -e "puts 9*6"
54
```

Die gesamte Ruby-Anweisung hinter dem `-e` muss in Anführungszeichen stehen. Da Sie für die Anweisung selbst manchmal auch Anführungszeichen benötigen, können Sie eine der beiden Sorten – einfache beziehungsweise doppelte – in die jeweils andere verschachteln. Zum Beispiel:

```
> ruby -e "puts 'Hallo, Ruby!'"
Hallo, Ruby!
```

Oder eben umgekehrt:

```
> ruby -e 'puts "Hallo, Ruby!"'
Hallo, Ruby!
```

Bitte beachten Sie, dass die zweite Variante in Windows-Versionen vor XP nicht funktioniert.

Bei Bedarf können Sie in einem solchen Aufruf auch mehrere Anweisungen unterbringen, indem Sie diese durch Semikola voneinander trennen. Im Gegensatz zu vielen anderen Sprachen ist dies bei Ruby standardmäßig nicht nötig, sondern nur dann, wenn Sie mehrere Befehle in eine Zeile schreiben. Das folgende Beispiel speichert das Ergebnis einer Berechnung zwischen und gibt es dann aus:

```
> ruby -e "ergebnis=21*2; puts ergebnis"
42
```

Spielen Sie auch mit dieser Variante ein wenig herum, um ein Gefühl dafür zu bekommen.

Wenn Sie die Option `-e` nicht verwenden, erfüllt der Ruby-Interpreter seine Standardaufgabe: Er liest ein Ruby-Skript aus einer Datei und führt es aus. Dazu müssen Sie den entsprechenden Dateinamen angeben. Falls Sie sich gerade im Verzeichnis der betreffenden Datei befinden, genügt der reine Dateiname, andernfalls können Sie auch einen relativen oder absoluten Pfad angeben. Das folgende Beispiel führt das Skript `test.rb` aus, falls es sich im aktuellen Verzeichnis befindet:

```
> ruby test.rb
```

Ein konkretes Beispiel für diese Herangehensweise finden Sie im nächsten Abschnitt.

Im Übrigen können Sie das Programm `ruby` auch ohne einen Dateinamen aufrufen. Dann haben Sie die Möglichkeit, beliebig viele Ruby-Codezeilen einzugeben. Abschließen müssen Sie die Eingabe mit EOF (»End of File«) – dieses Sonderzeichen erzeugen Sie auf einem Windows-Rechner mit **Strg + Z** und unter UNIX mit **Strg + D**. Hier ein kurzes Beispiel:

```
> ruby
ergebnis=13+10
puts ergebnis
Strg + D / Strg + Z
23
```

'Hello World, hello Ruby' – das erste Ruby-Programm

Wie Sie wahrscheinlich schon einmal gehört haben, ist es Teil der ehrwürdigen Programmierer-Tradition, als Erstes ein Programm zu schreiben, das die Worte »Hello World« (oder zu Deutsch »Hallo Welt«) ausgibt.[4] Um dieser Tradition treu zu bleiben und dennoch ein einigermaßen lehrreiches erstes Skript zu verfassen, soll das Beispiel Folgendes leisten:

1. »Hallo Welt!« ausgeben
2. Datum und Uhrzeit (gemäß Systemuhr) ausgeben
3. Den Benutzer nach dessen Namen fragen
4. »Hallo <Name>!« ausgeben

Sie können das Programm in einen Texteditor (siehe Kasten) eingeben und unter dem Namen `hallo.rb` speichern. Am besten legen Sie sich in Ihrem Benutzerverzeichnis – *Eigene Dateien* (Windows) beziehungsweise */home/Username* (UNIX-Systeme) – einen Ordner für Ihre gesammelten Ruby-Experimente an. Nennen Sie ihn beispielsweise *myruby*. Wenn Sie möchten, können Sie darunter auch noch Unterverzeichnisse für die einzelnen Kapitel oder Projekte erzeugen – größere Beispiele in späteren Kapiteln werden jeweils aus mehreren Dateien bestehen.

Texteditoren

Ein Texteditor ist ein Programm, mit dem Sie reinen Text eingeben und bearbeiten können – im Unterschied zu einem Textverarbeitungsprogramm, das vielerlei Text- und Absatzformatierungen zulässt und daher für die Eingabe von Programmquellcodes ungeeignet ist, weil diese Formatierungen mit in den Dateien gespeichert werden, so dass ein Interpreter oder Compiler sie nicht versteht. Wenngleich sich die Aufgabe der reinen Texteingabe trivial anhört, gibt es hunderte von Texteditoren mit unterschiedlichen Fähigkeiten und Besonderheiten.

Der einfachste Editor für Windows, der auch gleich mit dem System geliefert wird, heißt *Notepad*. Sie können ihn über *Start → [Alle] Programme → Zubehör → Editor* starten. Die Alternative ist *Start → Ausführen*; geben Sie dann `notepad` ein und drücken Sie **Enter**. Die Bedienung dieses Editors ist sehr einfach, zumal er kaum Optionen besitzt. Er wird komplett über das Menü gesteuert.

→

4 Unter *http://www2.latech.edu/~acm/HelloWorld.shtml* können Sie »Hello World«-Programme in mehreren hundert Programmiersprachen bestaunen, darunter natürlich auch in Ruby.

Da Notepad aufgrund seines geringen Leistungsumfangs vor allem für Programmierer nicht ausreicht, gibt es für Windows zahlreiche Drittanbieter-Editoren, sowohl kommerzielle als auch Open Source-Programme. Mein persönlicher Favorit ist *TextPad*. Es handelt sich um Shareware, die Sie unter *http://www.textpad.com* herunterladen und ausgiebig testen können, bevor Sie sich zum Kauf für überaus günstige US-$ 29,-- entschließen. TextPad besitzt etliche Features, die einem Programmierer das Leben erleichtern, zum Beispiel Zeilennummern, Mustersuche, Makros, Syntax-Highlighting[5] (Hervorhebung verschiedener Elemente von Programmiersprachen durch Einfärbung) und so weiter.

Auch der Ruby-Installer für Windows liefert einen Editor mit: *SciTE*. Nach der Installation können Sie ihn mittels *Start → [Alle] Programme → Ruby (Version) → SciTE* starten. Dieser Editor ist zwar nicht ganz so mächtig wie TextPad, liefert aber beispielsweise ebenfalls Syntax-Highlighting. Auch SciTE wird weitgehend über ein Menü gesteuert.

Noch komfortabler ist *FreeRIDE* – kein reiner Texteditor mehr, sondern eine grafische Ruby-Entwicklungsumgebung. Sie wird ebenfalls über das Ruby-Untermenü im Windows-Startmenü aufgerufen.

Auf Linux und anderen UNIX-Systemen sind vor allem zwei Editoren weit verbreitet: *vi* – beziehungsweise dessen komfortablere Weiterentwicklung *vim* (Vi IMproved) – sowie *Emacs*. Irgendeine Variante von vi ist auf so gut wie jedem UNIX-Rechner installiert. Geben Sie auf der Kommandozeile einfach

```
$ vi Dateiname
```

ein, um die angegebene Datei zu bearbeiten oder neu zu erstellen. Die Handhabung von vi(m) ist recht gewöhnungsbedürftig, weil es mehrere Arbeitsmodi gibt: Im *Befehlsmodus* rufen einfache Tastendrücke spezielle Funktionen auf – drücken Sie beispielsweise die Taste **X**, um das Zeichen unter dem Cursor zu löschen.

Sobald Sie im Befehlsmodus **I** drücken, wechseln Sie in den Eingabemodus. Dort können Sie ganz normal Text eingeben. Mit **Esc** gelangen Sie zurück in den Befehlsmodus. Es gibt zahlreiche Befehle; wenn Sie sie erlernen möchten, können Sie im Befehlsmodus **:help Enter** eingeben und sich durch die Themen führen lassen. Geben Sie zum Schluss – ebenfalls im Befehlsmodus – **:wq Enter** zum Speichern und Beenden ein oder **:q! Enter**, wenn Sie vi(m) beenden möchten, ohne zu speichern.

Den Emacs starten Sie mit

```
$ emacs Dateiname
```

→

5 Ab Werk enthält TextPad keine Syntaxdefinition für Ruby. Sie können sie aber von *http://www.textpad.com/add-ons/synn2t.html* herunterladen.

auf der Kommandozeile. Unterschiedliche Modi wie bei vi gibt es hier nicht; die unzähligen Befehle, Optionen und Hilfsprogramme werden mit **Strg**- oder **Alt**-Tastenkombinationen aufgerufen. Beachten Sie die speziellen Konventionen der Emacs-Dokumentation: **Strg** + *Taste* wird darin als C-Taste bezeichnet, **Alt** + *Taste* als M-Taste. Beispiele: `C-d`, also **Strg** + **D**, löscht das Zeichen unter dem Cursor, während `M-f`, also **Alt** + **F**, ein Wort weiterspringt. Die Hilfe, in der Sie die Beschreibung aller Befehle finden, erreichen Sie entsprechend mit `C-h`. Um die aktuelle Datei zu speichern, können Sie `C-x C-s` eingeben; mit `C-x C-c` beenden Sie Emacs.

Beachten Sie, dass Versionen von Vim und Emacs auch für Windows verfügbar sind – Sie finden sie auf den jeweiligen Projekt-Websites, *http://www.vim.org* beziehungsweise *http://www.gnu.org/software/emacs/emacs.html*. Eine etwas ausführlichere Einführung in die Arbeit mit beiden Editoren finden Sie beispielsweise unter *http://www.galileocomputing.de/openbook/kit/itkomp04002.htm* (in der kostenlos zugänglichen Online-Ausgabe meines Buchs *Kompendium der Informationstechnik*).

In Beispiel 1-1 sehen Sie zunächst den Quellcode. Geben Sie ihn einfach in Ihren Lieblingseditor ein. Keine Sorge – wenn Sie sich vertippen, werden Sie eine entsprechende Fehlermeldung erhalten. Ausführliche Erläuterungen folgen nach dem Listing.

Beispiel 1-1: Das erste Ruby-Programm, hello.rb

```
puts "Hallo Welt!"
jetzt = Time.new
puts "Es ist jetzt #{jetzt}."
puts "Wie heissen Sie?"
name = gets
name.chomp!
puts "Hallo #{name}!"
```

Begeben Sie sich nach der Eingabe in das Verzeichnis, in dem Sie das Skript gespeichert haben. Rufen Sie es folgendermaßen auf:

```
ruby hello.rb
```

In beiden Fällen wird das Skript nun ausgeführt. In Abbildung 1-2 sehen Sie den Verzeichniswechsel und die Ausführung in der Windows-Eingabeaufforderung. Hier ein Textbeispiel:

```
ruby hello.rb
Hallo Welt!
Es ist jetzt Tue Aug 08 20:37:23 Westeuropaeische Normalzeit 2006.[6]
Wie heissen Sie?
Sascha
Hallo Sascha!
```

6 Dieses Beispiel wurde noch mit Ruby 1.8.4 erstellt. In Version 1.8.5 wurde das Standardformat für Datum und Uhrzeit leicht geändert; hier lautet die Ausgabe `Tue Aug 08 20:37:23 +0100 2006`. Statt der Zeitzone wird also nun die Differenz zur UTC angezeigt. Diese Schreibweise heißt RFC-1123-Datumsformat und wird zum Beispiel in vielen Logdateien verwendet.

```
Eingabeaufforderung
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Dokumente und Einstellungen\Sascha>cd "Eigene Dateien\myruby"

C:\Dokumente und Einstellungen\Sascha\Eigene Dateien\myruby>ruby hello.rb
Hallo, Welt!
Es ist jetzt Mon Aug 21 12:30:53 Westeuropõische Sommerzeit 2006.
Wie heißen Sie?
Sascha
Hallo, Sascha!

C:\Dokumente und Einstellungen\Sascha\Eigene Dateien\myruby>
```

Abbildung 1-2: Die Ausführung des ersten Ruby-Skripts in der Windows-Eingabeaufforderung

Die einzelnen Codezeilen bedeuten Folgendes:

`puts "Hallo Welt!"`

Das Kommando `puts` gibt den übergebenen Text (oder einen beliebigen Ausdruck, der dann zunächst berechnet wird), gefolgt von einem Zeilenumbruch, aus. Die Ausgabe landet in der ersten freien Zeile des Konsolen- oder Eingabeaufforderungs-Fensters, in dem Sie das Skript ausführen. Konstanter Text – hier »Hallo Welt!« – ist an den Anführungszeichen erkennbar; er wird einfach Zeichen für Zeichen ausgegeben. Wenn Sie `puts` mehrere durch Kommata getrennte Ausdrücke übergeben, wird jeder in einer eigenen Zeile dargestellt.

`jetzt = Time.new`

Diese Anweisung speichert Datum und Uhrzeit gemäß der aktuellen Systemzeit des Rechners unter dem Namen `jetzt`.

`puts "Es ist jetzt #{jetzt}."`

Auch diese `puts`-Anweisung gibt Text aus – mit einer Besonderheit: Zeichenfolgen in geschweiften Klammern nach einer Raute werden zunächst ausgewertet und dann ausgegeben. Es wird also nicht etwa der ziemlich sinnlose Text »`#{jetzt}`« ausgegeben, sondern der Wert von `jetzt`, das heißt Datum und Uhrzeit.

`puts "Wie heissen Sie?"`

Eine weitere einfache Textausgabe.

`name = gets`

`gets` liest eine Zeile, normalerweise von der Tastatur. Dabei wird das Ergebnis, also die Benutzereingabe, in `name` gespeichert. Der Zeilenumbruch, der durch das Drücken von **Enter** entsteht, wird dabei mit gespeichert. Das ist in der Regel – wie auch hier – nicht erwünscht und wird im nächsten Schritt geändert.

`name.chomp!`

`chomp!` entfernt einen Zeilenumbruch am Ende eines Texts.

`puts "Hallo #{name}!"`

Auch hier wird der (zuvor vom Benutzer eingegebene) Wert von `name` in den statischen Text eingefügt und dann ausgegeben.

Zusammenfassung

Der Anfang ist gemacht: Nachdem Sie etwas Hintergrundwissen zu Ruby erhalten haben, ist der Interprter auf Ihrem Rechner installiert. Danach haben Sie `irb` kennen gelernt, eine interaktive Umgebung, in der Sie einzelne Ruby-Kommandos eingeben können und sofort Feedback erhalten. Zum Schluss haben Sie Ihr erstes vollständiges Skript eingegeben und mit Hilfe des Ruby-Interpreters ausgeführt.

Sie sind nun also bereit, die wichtigsten Merkmale der Programmiersprache Ruby Schritt für Schritt kennen zu lernen. Lesen Sie weiter, und Sie erfahren zunächst alles über ihre Grundbausteine. Bei einer natürlichen Sprache würde man sagen, Sie erlernen nun die Grundlagen von Rechtschreibung, Zeichensetzung, Wortschatz und Grammatik – und im Grunde ist es bei einer Programmiersprache auch nicht anders.

In diesem Kapitel:

- Praktische Einführung
- Grundlagen der Syntax
- Variablen, Ausdrücke und Operationen
- Kontrollstrukturen
- Mustervergleiche mit regulären Ausdrücken
- Iteratoren und Blöcke

KAPITEL 2

Sprachgrundlagen

If you want a language for easy object-oriented programming, or you don't like the Perl ugliness, or you do like the concept of LISP, but don't like too much parentheses, Ruby may be the language of your choice.[1]

– Aus der Einleitung der ruby-Manpage[2]

Nachdem Sie im vorigen Kapitel Ruby installiert und Ihre ersten Schritte damit unternommen haben, ist es nun an der Zeit, systematisch an die Sprache heranzugehen. Nach einer kurzen Einführung mit zwei Vorab-Praxisbeispielen wird das Basisvokabular der Sprache Ruby sehr gründlich, aber möglichst praxisnah vorgestellt.

Natürlich enthält auch der lange Theorieteil jede Menge Ruby-Code, und zwar zum Teil in Form einzelner Codezeilen, aber auch einige kurze Skripten. Um die einzelnen Zeilen auszuprobieren, ist die bereits erwähnte interaktive Ruby-Shell `irb` besonders praktisch, weil Sie auf jede Eingabe sofort Feedback erhalten. Die im Text gezeigten Beispiele wurden mit folgender Startvariante von `irb` erstellt:

```
> irb --simple-prompt
```

Dadurch wird die übliche Nummerierung der Eingabezeilen deaktiviert und lediglich >> als Eingabeaufforderung ausgegeben.

---

1 Übersetzung: Wenn Sie eine Sprache für die einfache objektorientierte Programmierung brauchen, wenn Ihnen die Hässlichkeit von Perl nicht zusagt oder wenn Sie das Konzept von LISP, aber nicht die damit einhergehenden zahlreichen Klammern mögen, könnte Ruby die Sprache Ihrer Wahl sein.

2 Geben Sie auf einem UNIX-System `man ruby` ein, um sie zu lesen.

Praktische Einführung

Dieses Kapitel kommt nicht umhin, einige recht trockene theoretische Konzepte zu beschreiben; die Kenntnis dieser Konzepte vervielfacht jedoch Ihren Programmiererfolg. Um das Ganze möglichst praxisnah zu gestalten, geht es in diesem Abschnitt mit zwei Beispielskripten los, in denen die Sprachelemente noch nicht ausführlich erläutert, aber in ihrem typischen Zusammenhang präsentiert werden.

Ein Ruby-Taschenrechner

Das erste Beispiel – ein kleiner Rechner – stellt die wichtigsten Ruby-Grundelemente vor: Variablen als Datenspeicher, Ein- und Ausgabe, Fallentscheidungen und Schleifen. Nach der Eingabe zweier Zahlen und einer der vier Grundrechenarten wird die entsprechende Rechenoperation ausgeführt. Bei einer fehlerhaften Eingabe geht es von vorn los; nach einer erfolgreichen Berechnung werden Sie gefragt, ob Sie einen weiteren Durchgang wünschen.

Ab dem nachfolgenden Skript wurden alle längeren Listings in diesem Buch mit Zeilennummern versehen. Diese dienen der Beschreibung des Codes, aber Sie dürfen sie nicht mit abtippen.

Beispiel 2-1 zeigt den kompletten Code. Geben Sie ihn, wie im vorigen Kapitel erwähnt, in Ihren Texteditor ein. Das bedeutet zwar ein wenig Tipparbeit, vermittelt Ihnen aber ein Gefühl für Ruby-Formulierungen.[3] Speichern Sie die Datei unter dem Namen *rechner.rb* in Ihrem Verzeichnis mit Ruby-Beispielen. Wenn Sie möchten, können Sie aus Gründen der Übersichtlichkeit ein Unterverzeichnis für dieses Projekt oder für das ganze Kapitel anlegen.

Beispiel 2-1: Der Ruby-Taschenrechner, rechner.rb

```
 1  # Ueberschrift ausgeben
 2  puts "Ruby-Rechner"
 3  puts "============"
 4  puts

 5  # Endlosschleife starten
 6  loop do
 7    print "Bitte die erste Zahl:      "
 8    # Eingabe direkt in Fliesskommazahl umwandeln
 9    # und in z1 speichern
10    z1 = gets.to_f
11    print "Bitte die zweite Zahl:     "
12    # Eingabe direkt in Fliesskommazahl umwandeln
13    # und in z2 speichern
14    z2 = gets.to_f
```

3 Sie können das Beispiel auch downloaden: *http://www.oreilly.de/catalog/rubybasger* oder *http://buecher.lingoworld.de/ruby*. Wir empfehlen Ihnen jedoch, den Code zunächst selbst einzugeben.

Beispiel 2-1: Der Ruby-Taschenrechner, rechner.rb (Fortsetzung)

```
15      print "Rechenoperation (+|-|*|/)? "
16      # Operator einlesen und anschliessenden Zeilenumbruch entfernen
17      op = gets.chomp

18      # Gueltigkeit des Operators pruefen
19      if op !~ /^[\+\-\*\/]$/
20        puts "Ungueltige Operation: #{op}"
21        puts
22        next
23      end

24      # Bei Division 0 als zweiten Operanden ausschliessen
25      if op == "/" && z2 == 0
26        puts "Division durch 0 ist verboten"
27        puts
28        next
29      end

30      # Ergebnis je nach Operator berechnen
31      case op
32        when "+"
33          ergebnis = z1 + z2
34        when "-"
35          ergebnis = z1 - z2
36        when "*"
37          ergebnis = z1 * z2
38        when "/"
39          ergebnis = z1 / z2
40      end

41      # Ausgabe des Ergebnisses
42      puts "Ergebnis: #{z1} #{op} #{z2} = #{ergebnis}"
43      print "Noch eine Berechnung (j/n)? "
44      nochmal = gets.chomp
45      puts
46      break if nochmal =~ /^n/i
47    end
```

Nachdem Sie das Rechner-Skript eingegeben haben, sollten Sie es zunächst ausprobieren. Wechseln Sie dazu auf der Konsole in das entsprechende Verzeichnis und geben Sie Folgendes ein:

```
> ruby rechner.rb
```

Nun können Sie gemäß der Eingabeaufforderung durch das Programm ein paar Rechenoperationen eingeben. Abbildung 2-1 zeigt einige Beispiele.

```
C:\rubytest>ruby rechner.rb
Ruby-Rechner
============

Bitte die erste Zahl:      5
Bitte die zweite Zahl:     2
Rechenoperation (+|-|*|/)? +
Ergebnis: 5.0 + 2.0 = 7.0
Noch eine Berechnung (j/n)? j

Bitte die erste Zahl:      45
Bitte die zweite Zahl:     0
Rechenoperation (+|-|*|/)? /
Division durch 0 ist verboten

Bitte die erste Zahl:      53
Bitte die zweite Zahl:     16
Rechenoperation (+|-|*|/)? *
Ergebnis: 53.0 * 16.0 = 848.0
Noch eine Berechnung (j/n)? n

C:\rubytest>_
```

Abbildung 2-1: Der Ruby-Rechner im Einsatz

Was die einzelnen Anweisungen des Skripts tun, soll an dieser Stelle noch nicht im Detail erklärt werden. Hier erhalten Sie lediglich einige kurze Hinweise. Beachten Sie auch die Zeilen, die mit `#` beginnen – es handelt sich dabei um Kommentare, die von Ruby selbst ignoriert werden und lediglich Ihrer eigenen Orientierung dienen.

Nach Ausgabe der Überschrift (Zeile 2 bis 4) läuft der eigentliche Rechner in einer Endlosschleife, wird also immer wieder ausgeführt. Dazu dient der folgende Block (Zeile 6 bis 47):

```
loop do
  ...
end
```

Innerhalb der Schleife werden zunächst die beiden Zahlen vom Benutzer erfragt (Zeile 7 bis 14). Das Eingabeergebnis `gets` wird mit `to_f` sofort in eine Fließkommazahl umgewandelt und dann in `z1` beziehungsweise `z2` abgelegt. Beim eingegebenen Operator wird dagegen mittels `chomp` der Zeilenumbruch abgeschnitten.

In Zeile 19 bis 29 erfolgen zwei Tests: Wenn der Operator keines der vier zulässigen Zeichen `+`, `-`, `*` oder `/` ist (Zeile 19), wird eine Fehlermeldung ausgegeben; anschließend wird mittels `next` (Zeile 22) der nächste Schleifendurchgang gestartet. Als Nächstes wird für den Fall, dass die Operation eine Division ist, die unzulässige 0 als zweiter Operand ausgeschlossen.

Das Ergebnis wird in Zeile 31 bis 40 mit Hilfe einer `case`/`when`-Struktur berechnet. Die `when`-Fälle prüfen nacheinander verschiedene Einzelwerte und führen die Operation für den passenden Fall durch. In Zeile 42 wird das auf diese Weise berechnete Ergebnis ausgegeben.

Zeile 42 bis 46 kümmern sich um die Frage, ob der Benutzer eine weitere Berechnung wünscht. Wenn seine Antwort mit `n` beginnt, wird die Schleife mit `break` verlassen (Zeile 46). Da nach dem `end` in Zeile 47 keine weitere Anweisung folgt, ist damit auch das gesamte Programm beendet.

In Tabelle 2-1 sehen Sie noch einmal die wichtigsten Codezeilen des Taschenrechner-Beispiels. Daneben können Sie nachlesen, in welchem Abschnitt dieses Kapitels (oder gegebenenfalls in welchem anderen Kapitel) das jeweilige Thema vertieft wird.

Tabelle 2-1: Ruby-Sprachelemente im Taschenrechner-Beispiel und in diesem Buch

Skriptzeilen	Code	Thema	Seite
1	`# Ueberschrift`	Kommentare	33

2–4	`puts "Ruby-Rechner"`
	`puts "============"`
	`puts`
		Einfache Ausgabe	114

6, 40	`loop do`
	`  ...`
	`end`
		Endlosschleifen	81

10	`z1 = gets.to_f`	Einfache Eingabe	114
		Methoden zur Typumwandlung	70
17	`op = gets.chomp`	String-Methoden	66

19, 23	`if op !~ /^[\+\-*\/]$/`
	`  ...`
	`end`
		if, elsif, else und unless	73

		Mustervergleiche mit regulären Ausdrücken	85
25	`if op == "/" && z2 == 0`	Vergleichsoperationen	52
		Logische Operationen	55

31, 40	`case op`
	`  ...`
	`end`
		case-when-Fallentscheidungen	76

42	`puts "Ergebnis: #{z1}`
	`  #{op} #{z2}`
	`  = #{ergebnis}"`
		Arithmetische Operationen	51

Ein Textmanipulierer

Wie bereits einige Male erwähnt wurde, ist Ruby eine objektorientierte Programmiersprache. Die theoretischen Grundlagen dazu werden im übernächsten Kapitel behandelt, aber als kleiner Vorgeschmack wurde das zweite Einführungsbeispiel in objektorientierter Schreibweise geschrieben: Zunächst wird eine Klasse definiert – eine Vorlage für beliebig viele Objekte, die eine Datenstruktur und Funktionen zu deren Manipulation (Methoden) miteinander verknüpfen. Im vorliegenden Fall ist die Datenstruktur ein kurzer Text, und die Methoden sind verschiedene Manipula-

tionsvariationen. Anschließend wird ein Objekt der Klasse erzeugt, und die Methoden werden aufgerufen.

Geben Sie den Code aus Beispiel 2-2 zunächst wieder in Ihren Texteditor ein (natürlich ohne Zeilennummern) und speichern Sie ihn in Ihrem Arbeitsverzeichnis als *modtext.rb*.

Beispiel 2-2: modtext.rb, der objektorientierte Textmanipulierer

```
 1  class ModText

 2    # Konstruktor: wird bei Objekterzeugung
 3    # mit new aufgerufen
 4    def initialize(txt = "")
 5      @txt = txt
 6    end

 7    # Enthaltenen Text nachtraeglich aendern
 8    def set_text(txt = "")
 9      @txt = txt
10    end

11    # Enthaltenen Text zurueckliefern
12    def get_text
13      @txt
14    end

15    # Als Text-Entsprechung des Objekts
16    # ebenfalls den Text zurueckliefern
17    def to_s
18      get_text
19    end

20    # Den Text rueckwaerts zurueckliefern
21    def turn
22      @txt.reverse
23    end

24    # Den Text mit * statt Vokalen zurueckliefern
25    def hide_vowels
26      @txt.gsub(/[aeiou]/i, "*")
27    end

28    # Den Text in "Caesar-Code" zurueckliefern
29    def rot13
30      @txt.tr("[A-Z][a-z]", "[N-ZA-M][m-za-m]")
31    end

32  end

33  # Neues ModText-Objekt mit Inhalt erzeugen
34  mtext = ModText.new("Hallo, meine liebe Welt!")
```

Beispiel 2-2: modtext.rb, der objektorientierte Textmanipulierer (Fortsetzung)

```
35  # Ausgabe der verschiedenen Methoden
36  printf "Originaltext:      %s\n", mtext.get_text
37  printf "Umgekehrt:         %s\n", mtext.turn
38  printf "Versteckte Vokale: %s\n", mtext.hide_vowels
39  printf "ROT13:             %s\n", mtext.rot13

40  # Text aendern
41  mtext.set_text("Diese Worte sind neu!")

42  # Ausgabe des Objekts als Text
43  # (ruft automatisch to_s auf)
44  printf "Neuer Text:        %s\n", mtext
```

Führen Sie das Skript nun wie gewohnt aus:

```
> ruby modtext.rb
```

Sehen Sie sich die mehrzeilige Ausgabe an (siehe Abbildung 2-2). Vergleichen Sie sie mit den dafür verantwortlichen `printf`-Anweisungen im letzten Teil des Skripts.

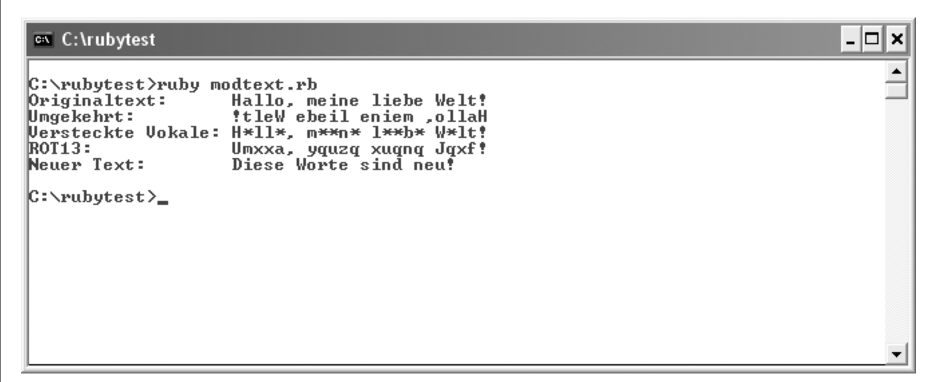

Abbildung 2-2: Ausgabe des objektorientierten Textmanipulierers

Das vorliegende Programm besteht aus drei logischen Teilen. Der erste Teil (Zeile 1 bis 32) ist die Definition der Klasse `ModText`:

```
class ModText
  ...
end
```

Die einzelnen `def`-Blöcke definieren die Methoden der Klasse, die im letzten Teil des Skripts aufgerufen werden. Außer `initialize` und `setText` liefern sie alle einen Wert zurück – in Ruby genügt es, den entsprechenden Wert als einzelne Anweisung hinzuschreiben.

Die Einzelheiten brauchen Sie jetzt noch nicht zu verstehen; jede verwendete Anweisung wird in diesem oder im nächsten Kapitel ausführlich erläutert. Wissen

sollten Sie an dieser Stelle nur noch, dass das Konstrukt `@txt` (eine Variable mit vorangestelltem `@`-Zeichen) für jedes Objekt der Klasse dauerhaft einen Wert speichert. Genau das macht den praktischen Nutzen der Objektorientierung aus: Die Daten und der Code zu ihrer Verarbeitung werden zusammengehalten (Fachbegriff: gekapselt); Sie brauchen sie nicht getrennt zu verwalten.

Hier eine kurze Übersicht über die Aufgaben der einzelnen Methoden, ohne eine genauere Beschreibung ihrer Funktionsweise:

- `initialize` (Zeile 4 bis 6) wird automatisch aufgerufen, wenn ein neues Objekt erzeugt wird, und weist dem gekapselten Text seinen Anfangswert zu – entweder den übergebenen Inhalt oder einen leeren Text.
- `set_text` (Zeile 8 bis 10) ändert den enthaltenen Text des Objekts nachträglich.
- `get_text` (Zeile 12 bis 14) liefert den Text zurück.
- `to_s` (Zeile 17 bis 19) ruft `get_text` auf, da beide Methoden dieselbe Aufgabe erfüllen. Besonderheit: Wenn etwas als Text eingesetzt werden soll, wird seine Methode `to_s` – falls vorhanden – automatisch aufgerufen.
- `turn` (Zeile 21 bis 23) dreht den Text mit Hilfe der Ruby-Methode `reverse` herum und liefert das Ergebnis zurück.
- `hide_vowels` (Zeile 25 bis 27) verwendet den sogenannten regulären Ausdruck (Suchmuster) `[aeiou]`, der auf alle Vokale im Text passt, und ersetzt diese durch Sternchen. Auch hier wird das Endergebnis zurückgeliefert.
- `rot13` (Zeile 29 bis 31) wendet den sogenannten »Cäsar-Code« auf den Text an: Jeder Buchstabe wird um 13 Zeichen verschoben; da das Alphabet 26 Buchstaben besitzt, sind die Vorgänge der Codierung und Decodierung identisch.

Der zweite Teil (ausschließlich Zeile 34) ist die Erzeugung eines konkreten `ModText`-Objekts mit Textinhalt:

```
mtext = ModText.new("Hallo, meine liebe Welt!")
```

Das ruft automatisch die Methode `initialize` auf und speichert den übergebenen Text dauerhaft in der Variablen `@txt` des neuen Objekts `mtext`.

Im dritten Teil (ab Zeile 36) werden schließlich die verschiedenen Methoden des Objekts aufgerufen; außer `setText` zum Wechseln des Inhalts dienen sie alle dem Auslesen des (meist manipulierten) Textinhalts von `mtext`. Die Anweisung `printf` ersetzt die im Text enthaltenen `%`-Platzhalter übrigens der Reihe nach durch die nachfolgenden Werte; `%s` steht dabei für einen Textwert (*String*).

Auch für den Textmanipulierer finden Sie in Tabelle 2-2 wieder eine Auflistung der Themen; diese beziehen sich, wie vermerkt, fast alle auf Kapitel 3 und 4.

Tabelle 2-2: Ruby-Sprachelemente im Textmanipulierer-Beispiel und in diesem Buch

Skriptzeilen	Code	Thema	Seite

1-32	`class ModText`
	`  ...`
	`end`
		Klassen entwerfen und implementieren	163

4-6	`def initialize`
	`  ...`
	`end`
		Der Konstruktor	167

8-31	`def ...`
	`  ...`
	`end`
		Methoden	169

5, 9	`@txt = txt`	Der Konstruktor – Instanzvariablen	167
13	`@txt`	Rückgabewerte	175
22	`@txt.reverse`	String-Methoden	66
26	`@txt.` `  gsub(/[aeiou]/i, "*")`	Reguläre Ausdrücke im Einsatz	97
30	`@txt.tr("[A-Z][a-z]",` `  "[N-ZA-M][m-za-m]")`	Reguläre Ausdrücke im Einsatz	97
34	`mtext =` `  ModText.new("...")`	Der Konstruktor – Instanziierung	167
36-39	`printf "...", ...`	Einfache Ausgabe	114

41	`mtext.`
	`  set_text("Diese Worte`
	`  sind neu!")`
		Methoden	169

Grundlagen der Syntax

Wenn Sie eine neue Sprache erlernen, besteht eine erste Herausforderung darin, zu erkennen, wie (sinnvolle) Sätze – bei einer Programmiersprache *Anweisungen* genannt – konstruiert werden. Dieser Grundaufbau einer Sprache wird als *Syntax* bezeichnet. Beachten Sie dabei den entscheidenden Unterschied zwischen natürlichen Sprachen und Programmiersprachen: Erstere bieten meist eine enorme Bandbreite an Ausdrucksmöglichkeiten sowie eine hohe Fehlertoleranz, während Letztere Ihnen sehr enge Grenzen stecken. Das ist keine böse Absicht der Programmiersprachenentwickler, sondern liegt daran, dass der Computer als digitale Maschine eben nicht denken, sondern nur ziemlich schnell rechnen kann.

Für Skriptsprachen wie Ruby ist es typisch, dass es kein formales Hauptprogramm oder Ähnliches gibt. Zunächst können Sie also ohne großartige Formalitäten eine Abfolge von Anweisungen hintereinander schreiben, was das Erlernen der Ruby-Grundlagen enorm erleichtert. In diesem Abschnitt wird zunächst einmal geklärt, aus welchen Komponenten Anweisungen bestehen können und wie sie voneinander getrennt werden.

Anweisungen und ihre Elemente

Die einzelnen Arbeitsschritte eines Computerprogramms werden als *Anweisungen* (auf Englisch *instructions*) bezeichnet. Im Verlauf dieses Kapitels und des restlichen Buchs werden Sie verschiedene Arten von Anweisungen kennenlernen.

In Ruby werden Anweisungen normalerweise durch Zeilenumbrüche voneinander getrennt; das in vielen anderen Programmiersprachen nötige Semikolon braucht in diesem Fall nicht gesetzt zu werden. Es besteht allerdings die Möglichkeit, mehrere Anweisungen in dieselbe Zeile zu schreiben, indem Sie sie durch Semikola trennen. Formal könnten Anweisungsfolgen also unter anderem folgendermaßen aussehen:

- Durch Zeilenumbruch getrennt:

  ```
  Anweisung1
  Anweisung2
  ...
  Anweisungn
  ```

  Beispiel:

  ```
  puts "Bitte geben Sie Ihren Namen ein"
  print "> "
  name = gets.chomp
  ```

- Durch Semikolon getrennt:

  ```
  Anweisung1; Anweisung2; ...; Anweisungn
  ```

  Beispiel:

  ```
  print "Noch ein Versuch (j/n)? "; auswahl = gets.chomp
  ```

- Mischformen wie diese:

  ```
  Anweisung1; Anweisung2
  Anweisung3
  ...; Anweisungn
  ```

  Beispiel:

  ```
  puts "Bitte geben Sie eine Zahl ein"; print "> "
  zahl = gets.to_f
  ```

Die meisten Anweisungen bestehen aus mehreren einzelnen Elementen. Das kleinste untrennbare Element einer Anweisung wird als *Token* bezeichnet; der erste Schritt bei der Übersetzung oder Interpretation von Quellcode besteht darin, ihn in diese Token zu zerlegen. Wichtige Token-Arten sind unter anderem:

- *Literale*, das heißt »wörtlich gemeinte« Einzelwerte wie die Zahl `42` oder der Text `"Das Restaurant am Ende des Universums"`.
- *Variablen*, also benannte Speicherplätze, die bei der Verarbeitung Ihres Skripts durch ihren aktuellen Wert ersetzt werden.
- *Operatoren* zur Verknüpfung von Werten, zum Beispiel arithmetische Operatoren wie `+` und `-` oder Vergleichsoperatoren wie etwa `==` (ist gleich) und `<` (kleiner als).

- *Methoden*, das heißt benannte Anweisungsfolgen, die bestimmte Aufgaben erledigen oder Werte berechnen. `puts "Text"` gibt beispielsweise den angegebenen Text aus; `Zahl.round` rundet die entsprechende Zahl.

Bei vielen Token ist es übrigens egal, ob Sie sie durch Leerzeichen voneinander trennen oder nicht:

```
3*5+2
```

bedeutet beispielsweise genau dasselbe wie

```
3 * 5 + 2
```

Hier einige wenige Anwendungsbeispiele, wobei deren Bedeutung und Funktionsweise später in diesem Kapitel genauer erläutert wird:

- Wertzuweisung – einen Wert speichern:

  ```
  gruss = "Hallo"
  ```

- Methodenaufruf – hier ein Ausgabebefehl:

  ```
  puts "Hier ist Ruby"
  ```

- Bedingte Ausführung (Glückwunsch, wenn `punkte` größer als 100 ist):

  ```
  if punkte > 100
    puts "Sie haben gewonnen"
  end
  ```

Kommentare

Wenn Sie eine Zeile mit einer Raute (#) beginnen, gilt diese als *Kommentar* und wird vom Ruby-Interpreter ignoriert. Gleiches gilt für den Rest einer Zeile nach einer Raute. Hier zwei Beispiele:

```
# Ein alleinstehender Kommentar
puts "Text"  # Kommentar hinter einer Anweisung
```

Sie sollten sich angewöhnen, Ihre Skripten stets ausführlich zu kommentieren. So wissen Sie während der Entwicklung und auch beim späteren Lesen eines Skripts noch genau, welche Aufgaben die jeweiligen Sektionen haben. Kommentare in separaten Zeilen sind dabei in der Regel zu bevorzugen.

Ab diesem Kapitel werden alle längeren Skriptbeispiele dieses Buchs mit Kommentaren versehen. Die Kommentarfrequenz entspricht dabei etwa meinen eigenen Angewohnheiten – nicht zu sparsam und nicht zu geschwätzig. Wenn wenige Codezeilen in zu vielen Kommentaren ertrinken, kann dies ein Skript nämlich genauso unleserlich machen, als wäre es unzureichend oder gar nicht kommentiert. Hilfreich ist natürlich in jedem Fall ein guter Editor, der Kommentare per Syntax-Highlighting andersfarbig darstellt.

Wenn Sie längere Erklärungspassagen einfügen möchten, die ebenfalls vom Interpreter ignoriert werden sollen, können Sie diese zwischen `=begin` und `=end` (jeweils in eigenen Zeilen) platzieren. Hier ein Beispiel:

```
puts "Hier steht Programmcode."
=begin
Hier steht Dokumentation.
Hier auch.
=end
puts "Jetzt geht das Programm weiter."
```

Das Besondere an diesen Passagen ist, dass man sie zusätzlich mit speziellen Schlüsselwörtern versehen und dann automatisch als sogenannte *RDoc*-Dokumentation exportieren kann.

Die Shebang-Zeile

Eine besondere Form des Kommentars, die Sie als erste Zeile eines Programms einsetzen können, ist die sogenannte *Shebang-Zeile* – kurz für `#` (*sharp*) und `!` (*bang*). Sie ermöglicht es Ihnen auf UNIX-Systemen, das Ruby-Skript wie ein binäres Programm ausführen, das heißt, ohne den Interpreter noch einmal auf der Kommandozeile aufzurufen. Die Shebang-Zeile teilt dem System nämlich mit, welches Programm das vorliegende Skript ausführen soll. Schematisch sieht diese Zeile so aus:

```
#!/Pfad/zu/ruby
```

In vielen Fällen lautet der konkrete Pfad:

```
#!/usr/bin/ruby
```

Wenn Sie Ruby nach der Anleitung im vorigen Kapitel selbst kompiliert haben, muss der Pfad angepasst werden; schreiben Sie beispielsweise Folgendes:

```
#!/usr/local/ruby/bin/ruby
```

Damit die automatische Ausführung tatsächlich funktioniert, müssen Sie dem Skript allerdings noch das *Executable*-Bit zuweisen. Geben Sie dazu in dessen Verzeichnis das Kommando `chmod a+x Skriptname` ein:

```
$ chmod a+x hello.rb
```

Anschließend können Sie das Skript einfach wie folgt starten, solange Sie im selben Arbeitsverzeichnis bleiben:

```
$ ./hello.rb
```

Auf einem Windows-System müssen Sie diese Zeile durch

```
#!C:/ruby/bin/ruby.exe
```

(beziehungsweise Ihren entsprechenden Installationspfad) ersetzen. Sie können sie aber auch einstweilen weglassen; unter Windows wird sie erst zwingend erforderlich, wenn Sie in Ruby CGI-Programme fürs Web schreiben (siehe Kapitel 6). Die automatische Ausführung nach UNIX-Art ist dort jedenfalls ohnehin nicht möglich.

Hinter der Shebang-Zeile wird häufig der Schalter `-w` angegeben. Zum Beispiel:

```
#!/usr/bin/ruby -w
```

Es handelt sich dabei um eine Kommandozeilenoption für den Ruby-Interpreter. Sie besagt, dass *Warnungen* aktiviert werden sollen. Eine Warnung ist nicht direkt eine Fehlermeldung, sondern ein Hinweis auf eine Ungenauigkeit oder unsauberen Programmierstil. Gerade wenn Sie Ruby neu lernen, sind die Warnungen also überaus nützlich.

Wenn Sie auf einer UNIX-Plattform arbeiten, sollten Sie es gleich ausprobieren: Stellen Sie den beiden Beispielskripten aus dem ersten Abschnitt eine Shebang-Zeile voran, machen Sie sie mit `chmod` ausführbar und starten Sie sie ohne expliziten `ruby`-Aufruf.

Für Windows gibt es noch eine einfachere Möglichkeit, auf den Befehl `ruby` zu verzichten: Nehmen Sie die Dateiendung *.rb* in die Umgebungsvariable `PATHEXT` auf, indem Sie *Start* → *Systemsteuerung* → *System* → *Erweitert* → *Umgebungsvariablen* wählen und den Wert dort anpassen. Beachten Sie, dass die Änderung erst gilt, sobald Sie eine neue Eingabeaufforderung öffnen. Seit der neuen Ruby-Version 1.8.5 wird diese Anpassung sogar automatisch vorgenommen.

Variablen, Ausdrücke und Operationen

Eine der wichtigsten Fähigkeiten von Programmiersprachen ist die Auswertung der sogenannten *Ausdrücke*. Die Bedeutung des Begriffs Ausdruck entspricht in etwa derjenigen in der Mathematik, wo Ausdrücke auch Terme genannt werden. Es handelt sich um beliebige Verknüpfungen von Werten (Literalen, Variablen usw.), unter anderem durch:

- *Arithmetische Operationen*, das heißt im Wesentlichen die Grundrechenarten
- *Vergleichsoperationen*, die überprüfen, ob Werte identisch oder verschieden sind
- *Logische Operationen*, mit denen sich mehrere Vergleiche verknüpfen lassen
- *Bit-Operationen*, die die gespeicherten Werte direkt manipulieren
- *Methodenaufrufe*, die beliebige Anweisungen ausführen, um Werte zu verändern

An jeder Stelle in einem Skript, an der ein einzelner Wert erwartet wird, darf auch ein beliebig komplexer Ausdruck stehen, solange er einen Wert des benötigten Typs ergibt (natürlich ist es unmöglich, Texte zu subtrahieren[4] oder eine einzelne Zahl zu sortieren).

Literale

Das einfachste Element eines Ausdrucks – und gleichzeitig ein eigenständiger Ausdruck – ist das *Literal*, ein »wörtlich gemeinter« Wert. Ruby kennt verschiedene Arten von Literalen:

- *Numerische Literale*, das heißt Zahlen – unterteilt in ganze Zahlen und Fließkommazahlen
- *String-Literale*, also durch Anführungszeichen gekennzeichnete Textblöcke
- *Symbole* – eindeutige, garantiert voneinander verschiedene Objekte ohne spezifischen Wert
- *Spezialliterale* wie `true`, `false` oder `nil`

Numerische Literale

Computer bewältigen heutzutage immer mehr nichtmathematische Aufgaben. Dennoch lassen sich diese hinter den Kulissen natürlich immer auf – teils sehr komplexe – Berechnungen zurückführen. Daher sind Zahlen einer der wichtigsten Bestandteile von Computerprogrammen. Sie werden für arithmetische Berechnungen verwendet, aber beispielsweise auch zur Darstellung von Bildschirmpositionen, Farben, Datum und Uhrzeit, Ordnungskriterien oder ähnlichen Angaben.

Da der Prozessor selbst zwischen Ganzzahlen und Fließkommazahlen unterscheidet, bietet auch so gut wie jede Programmiersprache diese beiden Zahlenarten an. Es gibt also einen internen Unterschied zwischen Zahlen wie beispielsweise `4` oder `4.4`. Es geht um die Art und Weise, wie diese Zahlen gespeichert werden und wie damit gerechnet wird: Ganzzahloperationen gehen schneller und effizienter vonstatten als Fließkommaberechnungen. Wann immer keine Nachkommastellen benötigt werden, bietet sich daher die Verwendung ganzer Zahlen an.

Ganze Zahlen (Integer) sind intuitiv alle Zahlen ohne Nachkommastellen, das heißt die Reihe

..., -3, -2, -1, 0, 1, 2, ...

Während diese Reihe in der Mathematik in beide Richtungen unendlich weit fortgesetzt wird, gibt es in der traditionellen Ganzzahlarithmetik von Computern eine

4 Na ja ... in Kapitel 4 werden Sie sehen, wie weit man das Standardverhalten von Ruby »aufbohren« kann.

größte und eine kleinste darstellbare Zahl; die Bandbreite hängt dabei von der Größe des Speicherplatzes ab, der pro Zahl reserviert wird.

Hier eine gute Nachricht für Sie: In Ruby können Sie tatsächlich *beliebig große* Ganzzahlen verwenden! Hinter den Kulissen werden aus Effizienzgründen zwei verschiedene Arten von Integerzahlen verwendet; beim Überschreiten der Grenze von gut einer Milliarde (positiv wie negativ) erfolgt allerdings ein vollautomatischer Wechsel zwischen ihnen. Wenn Sie sich dennoch für die Details interessieren, können Sie den Infokasten *Ganzzahlarithmetik* lesen.

Ganzzahlarithmetik

Die beschränkten Integerzahlen als effizienteste Operanden für Berechnungen werden in Ruby `Fixnum` genannt. Prinzipiell stehen für jede `Fixnum` 32 Bit (oder 4 Byte) an Speicherplatz zur Verfügung. Im letzten Bit wird allerdings eine konstante 1 gespeichert, um klarzustellen, dass der gespeicherte Wert vom Typ `Fixnum` ist. Somit sind 31 Byte für den eigentlichen Zahlenwert verfügbar.

Wie Ihnen bekannt sein dürfte, kann jedes Bit zwei verschiedene Zustände annehmen, die traditionell als 0 oder 1 bezeichnet werden. Somit besitzt jede `Fixnum` 2^{31} verschiedene mögliche Zustände,[5] von

```
0000000000000000000000000000000
```

bis

```
1111111111111111111111111111111
```

Das vorderste Bit wird für das *Vorzeichen* verwendet; 0 steht für positive Zahlen sowie 0, während 1 negative Zahlen repräsentiert. Positive Zahlen werden dabei einfach im Dualsystem (Zweiersystem) dargestellt – die weithin bekannte Zahlenreihe von 0 bis 10 lautet beispielsweise so:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010

Negative Zahlen werden dagegen durch ein Verfahren namens *Zweierkomplement* gebildet: Zum Kehrwert der entsprechenden positiven Zahl wird 1 addiert. Zum Beispiel:

```
0000000000000000000000000001111  (+7)
1111111111111111111111111110000  (Kehrwert)
1111111111111111111111111110001  (1 addieren => -7)
```

Damit lassen sich Zahlen von -1.073.741.824 bis +1.073.741.823 darstellen, was für die allermeisten alltäglichen Berechnungen oder Zählungen ausreichen dürfte.

→

5 Auf 64-Bit-Systemen sind es 63, und die angegebenen Zahlenbereiche verzigfachen sich bis in Bereiche, in denen Sie wahrscheinlich nie rechnen werden.

Der Übersicht halber werden die weiteren Eigenschaften des Zweierkomplement-Systems im Folgenden an nur vier Bit breiten Integerwerten dargestellt. Dabei reichen die darstellbaren Zahlen von -8 bis +7: -8 berechnet sich dabei aus +8 (binär, ohne Berücksichtigung des Vorzeichens: 1000). Der Kehrwert beträgt 0111. Anschließend wird 1 addiert; dies ergibt wieder 1000.

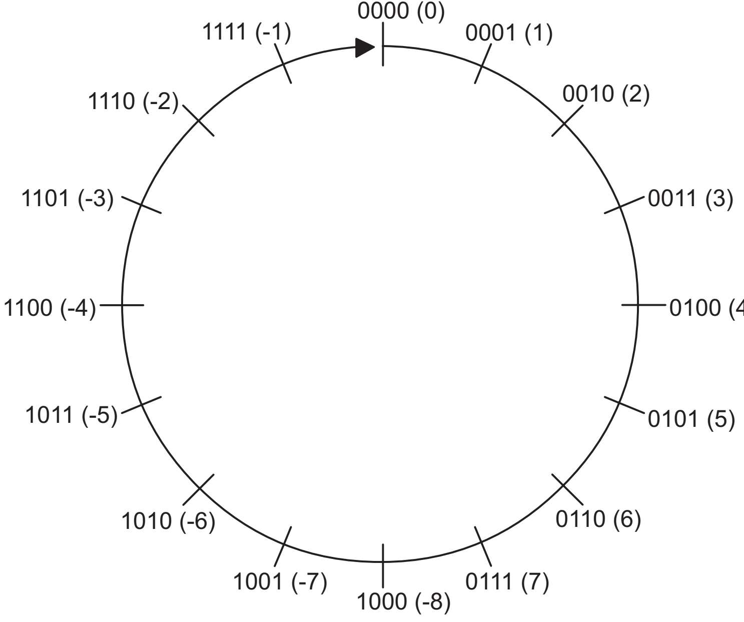

Abbildung 2-3: Darstellung der Raumfolgearithmetik am Beispiel des 4-Bit-Zweierkomplements

Der Vorteil der Zweierkomplement-Schreibweise besteht darin, dass die arithmetische Reihenfolge stimmt und dass jede Zahl – auch die größte positive – einen Nachfolger besitzt: In der Vier-Bit-Variante kommt nach 0111 (+7) der Wert 1000 (-8). Ebenso folgt auf 1111 (-1) wieder 0000 (0), da der Übertrag auf die nächste Stelle innerhalb fester Bit-Grenzen nicht mehr stattfinden kann. Das Ganze wird als *Raumfolgearithmetik* (sequence space arithmetic) bezeichnet und in Abbildung 2-3 anhand des Vier-Bit-Beispiels demonstriert.

Die Klasse der beliebig großen Ganzzahlen wird als `Bignum` bezeichnet. Selbstverständlich belegen sie mehr Speicher als `Fixnum`-Zahlen, und das Rechnen mit ihnen verlangsamt Ihre Programme. Dafür wird die Zahlenart, wie bereits erwähnt, automatisch konvertiert. Geben Sie in `irb` Folgendes ein, um das zu prüfen (der Operator ** steht für die Potenz):

```
>> a = 2 ** 30 - 1
```

→

Die Ausgabe lautet `1073741823`; das ist – wie oben hergeleitet – die größte darstellbare `Fixnum`.[6] Aufgrund der Verfügbarkeit von `Bignum` funktioniert aber auch Folgendes:

```
>> a += 1
=> 1073741824
```

(Andere Programmiersprachen würden -1073741824 liefern, sofern ihre Grenze ebenfalls bei 30 Bit läge.)

Weiter unten in diesem Kapitel erfahren Sie genauer, wie Sie mit Hilfe von `Ausdruck.class` den Datentyp von Literalen und Ausdrücken ermitteln können. Probieren Sie es hier ruhig schon einmal aus:

```
>> (2 ** 30 - 1).class
=> Fixnum
>> (2 ** 30).class
=> Bignum
```

Der Übersicht halber können Sie Unterstriche verwenden, um die Ziffern einer Zahl nach jeweils drei Stellen von rechts an zu trennen. Zum Beispiel:

```
>> 1_000_000_000
=> 1000000000
```

Es gibt aber noch mehr Möglichkeiten. Geben Sie in `irb` Folgendes ein:

```
>> 056
=> 46
```

Warum ist das Ergebnis 46 statt der zu erwartenden 56? Nun, das auch von Ruby standardmäßig verwendete Dezimalsystem ist nicht immer die ideale Darstellungsform für ganze Zahlen. Aus diesem Grund gibt es die Möglichkeit, Integer-Literale in anderen Schreibweisen anzugeben:

- Eine führende `0` betrachtet Zahlen als *oktal*, das heißt zum Achtersystem gehörend. Dabei sind Ziffern von 0 bis 7 zulässig; der Wert jeder Stelle beträgt das Achtfache der rechts daneben liegenden Stelle. Spielen Sie in `irb` einfach ein wenig mit Oktalwerten herum; ihre Werte werden jeweils dezimal angezeigt. Hier noch einige Beispiele:

```
>> 033
=> 27
>> 0777
=> 511
```

6 Wie erwähnt, müssen Sie auf einem 64-Bit-Rechner in diesem und den folgenden Beispielen stattdessen `2 ** 62` schreiben.

Wenn Sie Lust haben, können Sie nach einer führenden 0 auch einmal die unzulässigen Ziffern 8 und 9 ausprobieren. Sie erhalten natürlich eine Fehlermeldung:

```
>> 098
SyntaxError: compile error
(irb):3: Illegal octal digit
        from (irb):3
```

- Wenn Sie einer Zahl `0x` voranstellen, gilt sie als *hexadezimal*; das ist das Sechzehnersystem. Es ist ideal zur Darstellung von Byte-Inhalten, weil alle 256 möglichen Werte von 8 Bit ($2^8 = 256$) in zwei Stellen passen. Da es nur Ziffern von 0 bis 9 gibt, die für das Hexadezimalsystem nicht ausreichen, werden zur Darstellung der Ziffernwerte 10 bis 15 die Buchstaben A bis F hinzugenommen (die Sie übrigens auch kleinschreiben dürfen). Der Wert jeder Stelle beträgt das Sechzehnfache ihres rechten Nachbarn. Auch Hexadezimalwerte erkunden Sie am einfachsten, indem Sie sie in `irb` ausprobieren. Geben Sie beispielsweise folgende Zahlen ein:

```
>> 0x100
=> 256
>> 0xFF
=> 255
>> 0xABC
=> 2748
```

- Das Präfix `0b` schließlich ist für *duale* Zahlen gedacht. Die einzigen zulässigen Ziffern sind 0 und 1; eine Stelle besitzt den doppelten Wert ihres rechten Nachbarn. Probieren Sie auch das in `irb` aus:

```
>> 0b1000
8
>> 0b11111
31
>> 0b10101010
170
```

Die umgekehrte Umwandlung von Dezimalzahlen in beliebige andere Zahlensysteme ist ebenfalls möglich; diese Spezialvariante der zuständigen Methoden `to_s` und `to_i` wird weiter unten erläutert.

Natürlich kommen mathematische Anwendungen nicht immer ohne Nachkommastellen aus. Für das Rechnen mit Dezimalbrüchen stehen Prozessoren und Programmiersprachen die so genannten *Fließkommazahlen* oder auch *Gleitkommazahlen* (Floating Point Numbers) zur Verfügung – in Ruby durch die Klasse `Float` repräsentiert. Der Name besagt, dass die Anzahl der Nachkommastellen variabel sein darf.[7]

7 Im Gegensatz dazu gibt es auch *Festkommazahlen* mit einer konstanten Anzahl von Nachkommastellen; das prominenteste Beispiel sind Währungsbeträge mit zwei Dezimalstellen. Diese können Sie intern im Prinzip durch Integer darstellen und Komma oder Punkt jeweils erst bei der Ausgabe einfügen.

Beachten Sie, dass das Dezimaltrennzeichen gemäß der englischen Schreibweise kein Komma, sondern ein Punkt (.) ist. Beispiele für Fließkommazahlen sind etwa:

```
2.4, 3.1415926, 0.1, -7.5
```

Fließkomma-Arithmetik

Aufgrund der digitalen Natur des Computers müssen natürlich auch Fließkommazahlen in irgendeiner Form binär[8] repräsentiert werden. Das übliche Modell ist eine duale Form der wissenschaftlichen Schreibweise, die also x^*2^n (statt dem dezimalen x^*10^n) verwendet. In einer solchen Darstellung wird das x als *Mantisse*, die 2 (oder 10) als *Basis* und das n als *Exponent* bezeichnet; die Kommaverschiebung erfolgt durch eine Änderung des Exponenten. Negative Exponenten machen den Faktor $\text{Basis}^{\text{Exponent}}$ kleiner, positive dagegen größer als 1.

Die Mantisse wird stets auf eine Stelle vor dem Komma *normalisiert*, das heißt, $1 \leqq \text{Mantisse} < \text{Basis}$. Dadurch hat die erste Stelle für die Basis 2 *immer* den Wert 1. Das machen sich die Fließkommadarstellungen der meisten Rechner zunutze, indem sie sich das Bit zum Speichern der festen 1 sparen und stattdessen die Genauigkeit (Nachkommastellen der Mantisse) um dieses Bit verbessern.

Konkret bildet das höchste Bit das Vorzeichen der Mantisse. Darauf folgt die größte Bitgruppe, die den (Nachkomma-)Betrag der normalisierten Mantisse darstellt. Das nächste Bit ist das Vorzeichen des Exponenten, und die restlichen Bits stehen für dessen Betrag. Mögliche Exponenten reichen in Ruby von -1023 bis +1023. Das können Sie in `irb` ausprobieren (** steht dabei für die Potenz):

```
>> 1.0 * 2 ** -1023
=> 1.1125369292536e-308
>> 1.0 * 2 ** 1023
=> 8.98846567431158e+307
```

Übrigens können Sie auch in Ruby-Skripten die wissenschaftliche Schreibweise verwenden, allerdings nur die übliche dezimale. Dabei werden Mantisse und Exponent durch den Buchstaben `E` getrennt:

```
3.5E9  = 3 * 10**9  = 3500000000.0
2.0E-7 = 3 * 10**-1 = 0.0000002
```

Strings

Eine weitere wichtige Fähigkeit von Computerprogrammen und eine besondere Stärke von Ruby ist der Umgang mit Text. Textdaten werden in so genannten *Zeichenketten-* oder *String*-Literalen gespeichert. Diese werden durch Anführungs-

8 Beachten Sie den in der Umgangssprache manchmal vernachlässigten Unterschied zwischen *dual* (arithmetisches Zweiersystem) und *binär* (beliebige Codierung mit Hilfe von zwei verschiedenen Symbolen, meist 0 und 1).

zeichen gekennzeichnet. Grundsätzlich können einfache oder doppelte Anführungszeichen verwendet werden. Gültige Strings sind also beispielsweise "Hallo" oder 'Welt'.

Die einfachen Strings in Ruby enthalten nur Zeichen mit je 8 Bit. Eine Zeichensatzinformation steht nicht zur Verfügung. Die Folge: Umlaute und andere sprachliche Sonderzeichen werden oft falsch dargestellt. Die Konsolen-Anwendungen in den ersten Kapiteln dieses Buchs behelfen sich daher mit einer einfachen Umschreibung – etwa »Koeln« statt »Köln«. Wenn es in späteren Kapiteln um Webanwendungen geht, erfahren Sie, wie Webserver dem Browser Zeichensatzinformationen übermitteln können.

Die beiden Arten von Anführungszeichen haben Auswirkungen auf die Inhalte der Strings: Bei Strings in doppelten Anführungszeichen werden diverse Elemente auf spezielle Weise interpretiert; wenn Sie einfache Anführungszeichen benutzen, geschieht dies dagegen so gut wie gar nicht. Die beiden Arten von speziellen Inhalten, die bei der Verwendung doppelter Anführungszeichen ausgewertet werden, sind:

- *Escape-Sequenzen*: Einige Zeichen, denen Sie einen Backslash (\) voranstellen, haben eine besondere Bedeutung. Wichtige Beispiele sind \n für einen Zeilenumbruch oder \t für einen Tabulator. Auch wenn Sie das Anführungszeichen oder den Backslash selbst als Zeichen benötigen, kommen entsprechende Escape-Sequenzen zum Einsatz: \" wird in " umgewandelt, \\ in \. Geben Sie in irb zum Beispiel Folgendes ein, um Escape-Sequenzen zu testen:

  ```
  >> puts "\"Text\"\tmit Tab\nHier\tNoch ein Tab"
  "Text"  mit Tab
  Hier    Noch ein Tab
  => nil
  ```

  Das (in künftigen Beispielen weggelassene) Ergebnis => nil in der letzten Zeile bedeutet übrigens, dass der verwendete Ausgabebefehl puts keinen Wert zurückgibt – das weiter unten besprochene Literal nil steht für »leer« oder »nichts«. Falls Sie in irb dagegen nur den String selbst eingeben, erhalten Sie ihn mitsamt Anführungszeichen und codierten Escape-Sequenzen zurück:

  ```
  >> "\"Text\"\tmit Tab\nHier\tNoch ein Tab"
  => "\"Text\"\tmit Tab\nHier\tNoch ein Tab"
  ```

  Zum korrekten Testen von Strings müssen Sie daher stets einfache Ausgabebefehle verwenden; diese werden im nächsten Kapitel ausführlicher behandelt.
- *Eingebettete Ausdrücke*: Wenn Sie innerhalb eines Strings in doppelten Anführungszeichen eine Sequenz in geschweifte Klammern ({}) einschließen und dieser eine Raute (#) voranstellen, wird der Inhalt der Klammern als eingebetteter Ausdruck betrachtet und vor der Verwendung des Strings ausgewertet.

Der Ausdruck kann dabei beliebig komplex sein und einen Wert nahezu beliebigen Typs ergeben. Hier ein einfaches Beispiel:

```
>> puts "6 * 7 = #{6 * 7}"
6 * 7 = 42
```

Selbstverständlich können Sie einfache Zeichen, Escape-Sequenzen und eingebettete Ausdrücke innerhalb von Strings beliebig miteinander vermischen. Zum Beispiel:

```
>> puts "Das Ergebnis der Berechnung\t3 + 2\tbetraegt:\n#{3+2}"
Das Ergebnis der Berechnung     3 + 2   betraegt:
5
```

Bei Strings in einfachen Anführungszeichen werden *keine* eingebetteten Ausdrücke ausgewertet. Escape-Sequenzen im Allgemeinen auch nicht, allerdings stehen `\'` für ein literales einfaches Anführungszeichen (`'`) und `\\` für einen Backslash zur Verfügung. Probieren Sie die obigen Beispiele ruhig einmal mit einfachen Anführungszeichen aus, um den Unterschied zu erfahren:

```
>> puts '\'Text\'\tmit Tab\nHier\tNoch ein Tab'
'Text'\tmit Tab\nHier\tNoch ein Tab
>> puts '6 * 7 = #{6 * 7}'
6 * 7 = #{6 * 7}
```

Zur Verwendung von Anführungszeichen für String-Literale gibt es übrigens noch eine Alternative. Diese ist zum Beispiel nützlich, wenn Strings viele literale Anführungszeichen enthalten, da Sie diese dann nicht durch den Backslash zu schützen brauchen. Im Einzelnen geht es um folgende Optionen:

- `%q!...!` ersetzt einfache Anführungszeichen; `%q!Hallo!` ist zum Beispiel äquivalent mit `'Hallo'`.
- `%Q!...!` kann statt doppelter Anführungszeichen verwendet werden. Beispielsweise bedeutet `%Q!Ein Text!` dasselbe wie `"Ein Text"`. Das große `Q` können Sie sogar weglassen: Auch die Variante `%!...!` wertet Escape-Sequenzen und eingebettete Ausdrücke aus.

Die Ausrufezeichen (`!!`) können – je nachdem, welche Zeichen innerhalb der Strings selbst benötigt werden – durch andere Zeichen ersetzt werden: zum einen durch alle Arten von Klammern, nämlich `(...)`, `[...]`, `{...}` und sogar `<...>`, zum anderen aber auch durch zwei gleiche Zeichen wie `|...|` , `/.../` oder gar `@...@`.

Der größte Vorteil dieser Varianten besteht, wie bereits erwähnt, darin, dass Anführungszeichen ohne Escaping verwendet werden können. Ein Beispiel (hier mit Klammern statt Ausrufezeichen, weil Letztere im Text selbst vorkommen):

```
>> puts %Q("Du musst mir helfen!", rief er. "Sie sind hinter mir her!")
"Du musst mir helfen!", rief er. "Sie sind hinter mir her!"
```

An jeder Stelle, an der ein String stehen kann, besteht auch die Möglichkeit, ein so genanntes *HIER-Dokument* zu verwenden, benannt nach der Tatsache, dass es »bis HIER«, das heißt bis zu einer speziellen Endmarkierung, reicht. Jedes HIER-Doku-

ment beginnt dabei mit der Sequenz <<Name_der_Markierung und endet mit einer Zeile, in der die Markierung erneut steht. Hier zwei typische Beispiele – ein Ausgabebefehl und eine Variablen-Wertzuweisung (mehr zu Letzteren lesen Sie weiter unten):

```
>> puts <<ENDMARKE
Text
mehr Text
ENDMARKE
Text
mehr Text

>> var = <<ENDMARKE
Text
Text
ENDMARKE
=> "Text\nText\n"
```

Die ENDMARKE-Bezeichnung – traditionell (aber nicht zwingend erforderlich) komplett in Großbuchstaben – muss ganz am linken Rand einer eigenen Zeile stehen. Falls es die letzte Zeile Ihres Skripts sein sollte, ist es unabdingbar, dass Sie dahinter noch einen Zeilenumbruch einfügen, denn nur dieser macht die Markierung zur vollwertigen Zeile. Der gesamte Text zwischen den Markierungen wird als String betrachtet; die Auswertung von Escape-Sequenzen und die Variablensubstitution erfolgen wie bei Strings in doppelten Anführungszeichen. Zusätzlich wird auch jeder Zeilenumbruch innerhalb des HIER-Dokuments in den String übernommen.

Spezialliterale

Neben den bereits behandelten Zahlen und Strings kennt Ruby noch einige besondere Literale für spezielle Aufgaben. Ihre Bedeutung wird nach und nach im Kontext klar; hier werden sie zunächst nur kurz aufgezählt.

- Die Wahrheitswerte true und false sind die Ergebnisse von Vergleichen und anderen logischen Operationen. Wenn ein Vergleich zutrifft, hat er das Ergebnis true, andernfalls false. Sogar das können Sie in irb ausprobieren:

```
>> 3 == 3
=> true
>> 3 < 3
=> false
```

In den meisten Programmiersprachen gelten 0 und der leere String ("") als false, während jeder von 0 verschiedene Wert true ist. Das ist in Ruby nicht so; auch Nullwerte haben im Kontext der Logik den Wert true. false existiert tatsächlich nur als Ergebnis einer unzutreffenden Vergleichsoperation.

- Das »Leer-Literal« `nil` steht für Elemente, die *gar keinen* Wert besitzen. Auch das ist etwas anderes als die oben erwähnten Werte `0` und `""`. In Vergleichen gilt `nil` als falsch.
- *Symbole* haben die Form `:Symbolname`, wobei der Symbolname Buchstaben, Ziffern und Unterstriche enthalten, aber nicht mit einer Ziffer beginnen darf. Symbole sind ein besonders speicherschonendes Verfahren, um *unterschiedliche* Werte zur Verfügung zu haben, wenn es nicht auf einen *bestimmten* Wert ankommt.
- *Bereiche* (Ranges) sind aufsteigende Reihen von Ganzzahlen oder Zeichen. Dabei wird jeweils das erste und das letzte Element, getrennt durch Punkte, notiert. Bei zwei Punkten ist der Bereich inklusive dem letzten Element gemeint. Zum Beispiel steht `3..7` für die Zahlenreihe 3, 4, 5, 6, 7. Wenn Sie dagegen drei Punkte verwenden, wird der letzte Wert ausgeschlossen. `"a"... "f"` steht beispielsweise für `"a"`, `"b"`, `"c"`, `"d"`, `"e"`. Bereiche sind zum Beispiel praktisch, um zu überprüfen, ob sich ein bestimmter Wert darin befindet, oder um eine Schleife über alle ihre Elemente zu bilden.

Variablen

Eines der wichtigsten Konzepte höherer Programmiersprachen – und ein gefürchteter Stolperstein für Einsteiger – sind die *Variablen*. Es handelt sich im Grunde um nichts weiter als benannte Speicherplätze. Wenn Sie den Eindruck haben, der Umgang mit Variablen sei schwierig, stellen Sie sich einen Moment lang vor, es gäbe keine (wie bei den ersten Computern in den 1940er bis 50er Jahren): Sie müssten direkt auf die nummerierten Blöcke des Arbeitsspeichers zugreifen und sich merken, wo Sie bereits etwas abgelegt haben und was noch frei ist; außerdem müssten Sie selbst die Speicherbereiche für Programme und Daten auseinanderhalten.

Variablen erleichtern Ihnen das Leben ungemein: Sie brauchen sich nur noch den selbst gewählten Namen eines Speicherplatzes zu merken, und der Computer kümmert sich hinter den Kulissen darum, einen freien Platz für die enthaltenen Informationen zu finden. Auch die Speicherbereiche der verschiedenen Programme und ihrer Daten hält er in aller Regel sauber auseinander.

Möglicherweise kennen Sie den Begriff der Variablen aus der Mathematik. Sowohl in einer mathematischen Formel als auch in einem Computerprogramm ist die Variable ein Platzhalter, der einen mehr oder weniger beliebigen Wert repräsentieren kann. Sobald Sie ein Programm allerdings ausführen (dies ist die so genannte *Laufzeit*), muss die Variable jederzeit einen eindeutigen Wert besitzen, damit alles funktioniert.

In Ruby erzeugen Sie eine Variable ganz einfach, indem Sie ihr einen Anfangswert zuweisen. Hier sehen Sie ein Beispiel:

```
a = 1
```

Dies ruft eine Variable namens a ins Leben und weist ihr den Wert 1 zu. Wenn Sie danach in einem Ausdruck Ihres Skripts a benutzen, wird ebendiese 1 eingesetzt. Zum Beispiel:

```
>> puts "Der Wert von a ist #{a}."
Der Wert von a ist 1.
```

Solange Sie allerdings statische Werte für Ihre Variablen verwenden, machen Sie Ihre Skripten zwar übersichtlicher, aber noch nicht dynamisch: Im obigen Beispiel könnten Sie statt #{a} genauso gut die 1 hinschreiben und würden dasselbe Ergebnis erhalten. In der Praxis sind Variablen erst dann wirklich sinnvoll, wenn sie bei jedem Durchlauf andere Werte annehmen können. Das geschieht beispielsweise durch:

- die Speicherung von Benutzereingaben in Variablen
- mehrere Arbeitsdurchläufe mit einer Variablen als Schleifenzähler
- eine von einer Bedingung abhängige Wertzuweisung

Natürlich können Variablen nicht nur ganzzahlige Werte enthalten, sondern beliebige Objekte. Probieren Sie es in irb aus, indem Sie Variablen verschiedene Arten von Werten zuweisen und anschließend mittels variable.class ihren Typ erfragen. Hier zur Anregung einige Beispiele:

```
>> zahl = 3.456
=> 3.456
>> zahl.class
=> Float
>> text = "Hallo"
=> "Hallo"
>> text.class
=> String
>> jetzt = Time.new
=> Thu Sep 14 21:37:52 +0200 2006
>> jetzt.class
=> Time
```

Sie können Variablen statt Literalen auch beliebige Ausdrücke als Wert zuweisen. Zum Beispiel:

```
>> ergebnis = 2*7
=> 14
```

Da der Ausdruck zuerst berechnet und erst dann in der Variablen abgelegt wird, kann er sogar die Variable selbst enthalten und so ihren bisherigen Wert modifizieren. Das folgende Beispiel erhöht den Wert der Variablen zaehler um 1:

```
zaehler = zaehler + 1
```

Der Name einer Variablen, der so genannte *Bezeichner*, kann aus Buchstaben, Ziffern und Unterstrichen bestehen und beliebig lang sein. Das erste Zeichen darf keine Ziffer sein. Ruby unterscheidet bei Bezeichnern zwischen Groß- und Kleinschrei-

bung. Variablennamen beginnen standardmäßig mit einem Kleinbuchstaben. Ruby selbst verwendet für seine eigenen Bezeichner, sofern sie aus mehreren Wörtern bestehen, keinen »CamelCode« (Binnenmajuskeln), sondern trennt die einzelnen Wörter durch Unterstriche. Sie sollten sich auch bei selbst gewählten Bezeichnern an diese Konvention halten.

Hier einige Beispiele für gültige und typische Variablennamen in Ruby:

- `n`
- `zahl`
- `text1`
- `neuer_wert`

Eine Liste *ungültiger* Bezeichner wird hier aus didaktischen Gründen *nicht* veröffentlicht – denn Menschen neigen dazu, sich Gedrucktes auch dann einzuprägen, wenn es falsch ist.

Normalerweise existiert eine Variable nur innerhalb der Umgebung, in der sie definiert wurde – das wird als ihr so genannter *Gültigkeitsbereich* oder *Geltungsbereich* (Englisch *scope*) bezeichnet. Näheres dazu erfahren Sie im übernächsten Kapitel, da Methoden, Klassen und Module zahlreiche Besonderheiten mit sich bringen. Soll eine Variable unabhängig von allen Gültigkeitsbereichen im gesamten Skript gelten, müssen Sie ihrem Namen ein Dollarzeichen (`$`) voranstellen. Dieser spezielle Typ wird als *globale Variable* bezeichnet, während die Standardvariablen ohne Dollarzeichen *lokale Variablen* heißen. Beachten Sie, dass das Dollarzeichen ein Bestandteil des Namens ist; `test` und `$test` sind beispielsweise zwei verschiedene Variablen. Hier ein Definitionsbeispiel:

```
lokal =   "Gilt nur lokal."
$global = "Gilt im gesamten Skript."
```

Wie in den meisten Skriptsprachen ist eine Ruby-Variable im Lauf ihrer Existenz nicht auf einen bestimmten Datentyp festgelegt. Sie können ihr nacheinander verschiedene Wertetypen zuweisen. Diese Eigenschaft wird als *dynamische Typisierung* bezeichnet. Speziell für Ruby wurde der Begriff *Duck Typing* geprägt: »If it walks like a duck and quacks like a duck, then it must be a duck.«[9]

Probieren Sie auch dieses Feature aus, indem Sie ein und derselben Variablen nacheinander unterschiedliche Arten von Werten zuweisen:

```
var = 2
var = 2.2
var = "Hallo"
var = true
```

9 »Wenn es wie eine Ente watschelt und wie eine Ente quakt, dann muss es eine Ente sein.«

Wenn Sie nach jeder Wertzuweisung `var.class` aufrufen, erhalten Sie nacheinander folgende Typangaben: `Fixnum`, `Float`, `String` und `TrueClass`.

Eine weitere interessante Funktion, die Ruby vielen anderen Sprachen voraushat, ist die Mengen-Wertzuweisung: Vor dem Gleichheitszeichen können Sie eine durch Kommata getrennte Liste von Variablen angeben, wenn Sie dahinter genauso viele Werte angeben. Zum Beispiel:

```
>> a, b, c = 1, 2, 3
>> a
1
>> b
2
>> c
3
```

Daraus ergibt sich unter anderem, dass Sie die Werte zweier Variablen in einem Schritt vertauschen können. Das folgende Beispiel vertauscht die Werte von `zahl1` und `zahl2`:

```
>> zahl1, zahl2 = zahl2, zahl1
```

In fast jeder anderen Programmiersprache brauchen Sie für diese Aufgabe eine Hilfsvariable und drei Anweisungen. Natürlich funktioniert dieser umständlichere Weg auch in Ruby, wenn Sie das unbedingt wollen:

```
>> helper = zahl1
>> zahl1 = zahl2
>> zahl2 = helper
```

Konstanten

Wenn Sie einen Bezeichner wählen, der ausschließlich aus Großbuchstaben (und gegebenenfalls Ziffern) besteht, wird keine Variable definiert, sondern eine *Konstante*. Diese ist global gültig und behält ihren einmal festgelegten Wert im gesamten Skript bei. Ein praktisches Beispiel wäre etwa ein Umrechnungsfaktor für Währungen. Hier ein sehr kurzes Beispielskript:

```
DM = 1.95583
euro100 = 100*DM
puts "100 Euro sind #{euro100} DM."
dm100 = 100/DM
puts "100 DM sind #{dm100} Euro."
```

Das liefert die folgende Ausgabe:

```
100 Euro sind 195.583 DM.
100 DM sind 51.1291881196219 Euro.
```

Arrays

Für die automatisierte Verarbeitung größerer Datenmengen ist es sehr nützlich, dass es spezielle Variablen gibt, in denen sich Gruppen von Werten speichern lassen

– die so genannten *Arrays* (zu Deutsch etwa Anordnungen). Ein Array enthält beliebig viele Werte beliebiger Typen, die über den Variablennamen und eine in eckigen Klammern stehende laufende Nummer (den *Index*) angesprochen werden.

Auch Arrays werden am einfachsten durch eine Wertzuweisung erzeugt, dazu wird die Wertegruppe durch Kommata getrennt und in eckige Klammern gesetzt. Hier zwei Beispiele:

```
wochentage = ["Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"]
quadratzahlen = [0, 1, 4, 9, 16, 25, 36, 49, 64]
```

Sie können auch ein leeres Array erzeugen, um es später mit Werten zu füllen:

```
noch_nichts = []
```

Der Index beginnt jeweils bei 0. Um also das erste Element des jeweiligen Arrays zu erhalten, können Sie Folgendes eingeben:

```
>> wochentage [0]
=> "Mo"
>> quadratzahlen [0]
=> 0
```

Um zu ermitteln, wie viele Elemente ein Array besitzt, können Sie dessen Methode `length` aufrufen (die Feinmechanik von Methodenaufrufen wird weiter unten erläutert; hier können Sie sie einfach intuitiv verwenden):

```
>> wochentage.length
=> 7
>> quadratzahlen.length
=> 9
```

Das letzte Element eines Arrays erhalten Sie, indem Sie die um 1 verminderte Elementanzahl als Index verwenden:

```
>> wochentage [wochentage.length-1]
=> "So"
>> quadratzahlen [quadratzahlen.length-1]
=> 64
```

Dafür gibt es übrigens eine Kurzfassung: Negative Indizes liefern die Elemente des Arrays vom hinteren Ende an. Die beiden obigen Beispiele können Sie also kürzer schreiben:

```
>> wochentage [-1]
=> "So"
>> quadratzahlen [-1]
=> 64
```

Eine praktische Methode, um ein Array aus Strings zu erzeugen, besteht übrigens darin, den Quoting-Operator `%w` zu verwenden (wie bei `%q` & Co. wahlweise mit Klammern oder identischen Umschließungszeichen). Dadurch werden die durch Leerzeichen getrennten Zeichenfolgen zu Elementen des Arrays. Zum Beispiel:

```
>> jahreszeiten = %w(Fruehling Sommer Herbst Winter)
=> ["Fruehling", "Sommer", "Herbst", "Winter"]
```

Die interessantesten Array-Operationen werden durch Methoden bereitgestellt, die es Ihnen ermöglichen, Elemente hinzuzufügen oder zu entfernen, den Inhalt des Arrays zu sortieren und so weiter. Die entsprechenden Methoden werden weiter unten im Abschnitt über Methodenaufrufe vorgestellt. Wie Sie alle Elemente eines Arrays in einer Schleife durchlaufen können, erfahren Sie dagegen in den Abschnitten über einfache Schleifen beziehungsweise Iteratoren.

Hashes

Ein *Hash* ist ein sehr ähnliches Konstrukt wie ein Array. Die Besonderheit besteht darin, dass die Indizes keine fortlaufenden Nummern, sondern beliebige Objekte sind (meistens werden Strings oder Symbole verwendet); sie werden oft als *Schlüssel* (*Keys*) statt als Indizes bezeichnet. Auf diese Weise ermöglichen Hashes Zuordnungen diverser Wertepaare ohne garantierte Reihenfolge. Hash-Literale stehen in geschweiften Klammern: {}. Zwischen Index und Element wird die Zeichenfolge => verwendet. Das folgende Beispiel verwendet Wochentagsabkürzungen als Schlüssel und die vollständigen Namen als Werte:

```
wochentagsnamen = {
  "Mo" => "Montag",
  "Di" => "Dienstag",
  "Mi" => "Mittwoch",
  "Do" => "Donnerstag",
  "Fr" => "Freitag",
  "Sa" => "Samstag",
  "So" => "Sonntag"
}
```

Die Aufteilung auf mehrere Zeilen ist dabei natürlich nur eine Option, die der besseren Übersicht dient; Sie können genauso gut alles hintereinander schreiben.

Der Zugriff auf ein einzelnes Element eines Hashs erfolgt genau wie bei einem Array, indem Sie den Schlüssel in eckige Klammern setzen. Zum Beispiel:

```
>> wochentagsnamen ["Mo"]
=> "Montag"
```

Nützliche Methoden und Iteratoren für Hashes lernen Sie weiter unten in diesem Kapitel kennen, wie bereits bei den Arrays angekündigt.

Operationen

Nachdem Sie nun die wichtigsten Einzelbestandteile von Ausdrücken kennengelernt haben, ist es an der Zeit zu erfahren, wie Sie diese Elemente verknüpfen kön-

nen. Für diese Aufgabe stellt Ruby eine umfangreiche Menge von *Operatoren* zur Verfügung.

Arithmetische Operationen

Die bekanntesten Operatoren, die Ihnen aus den alltäglichen Bereichen der Mathematik geläufig sein dürften, sind die *arithmetischen Operatoren*. Sie dienen der Durchführung von Berechnungen. Zunächst einmal gibt es die vier klassischen Grundrechenarten:

- Addition (+)
- Subtraktion (-)
- Multiplikation (*)
- Division (/)

Wie bereits im vorigen Kapitel gezeigt, können Sie diese Operatoren leicht mit numerischen Literalen in `irb` ausprobieren. Hier einige Beispiele zur Anregung:

```
>> 17 + 4
=> 21
>> 93 - 51
=> 42
>> 10 * 2 + 3
=> 23
>> 17 / 2
=> 8
>> 17.0 / 2
=> 8.5
```

Das Phänomen aus dem vorletzten Beispiel wurde bereits beschrieben: Wenn beide Operanden bei der Division ganzzahlig sind, wird auch das Ergebnis als ganze Zahl ausgegeben und dabei nicht etwa gerundet, sondern abgeschnitten. Aus Sicherheitsgründen müssen Sie also mindestens einen der Operanden explizit als Fließkommazahl hinschreiben. Wenn Sie dabei keine Literale, sondern Variablen verwenden, müssen Sie die weiter unten beschriebene `Float`-Konvertierungsmethode `to_f` benutzen.

Neben den vier Standard-Grundrechenarten kennt Ruby noch zwei weitere arithmetische Operationen:

- `%` ist der so genannte *Modulo*-Operator; er liefert den Rest der ganzzahligen Division. Hier einige Beispiele:

```
>> 3 % 2
=> 1
>> 18 % 7
=> 4
>> 9 % 3
=> 0
```

- `**` ist der Potenz-Operator, der den ersten Operanden mit dem zweiten potenziert. Einige Beispiele:

```
>> 2 ** 10
=> 1024
>> 7 ** 3
=> 343
>> 5.0 * 10 ** -4
=> 0.0005
```

Im weiteren Sinne sind auch die Vorzeichen `+` und `-` als arithmetische Operationen zu verstehen, wenngleich sie nicht zwei Werte miteinander verknüpfen, sondern den unmittelbar nachfolgenden Ausdruck modifizieren.[10]

Vergleichsoperationen

Eine häufige Aufgabe von Computerprogrammen besteht darin, Vergleiche durchzuführen und aufgrund der Ergebnisse dieser Vergleiche Entscheidungen zu treffen, sprich bestimmte Anweisungen auszuführen oder nicht auszuführen. Die Fallentscheidungen selbst werden im nächsten Abschnitt behandelt; hier geht es zunächst um ihre Grundlage im Bereich der Ausdrücke.

Jeder *Vergleichsoperator* dient dazu, zwei beliebige Werte miteinander zu vergleichen. Der zurückgelieferte Wert ist in der Regel `true` oder `false` (in einigen Ausnahmefällen auch `nil` oder etwas anderes).

Neben dem naheliegenden Vergleich von Zahlen lassen sich auch Strings miteinander vergleichen, das Kriterium ist dabei die Position des ersten unterschiedlichen Zeichens im Zeichensatz. Alle Großbuchstaben sind dabei »kleiner als« alle Kleinbuchstaben. Während Ganz- und Fließkommazahlen zum Vergleich automatisch ineinander konvertiert werden, findet *keine* Umwandlung von Zahlen in Strings oder umgekehrt statt.

Hier die Liste der grundlegenden Vergleichsoperatoren mit einer Beschreibung ihres Verhaltens und `irb`-Beispielen:

- `==` (Gleichheit). Liefert `true`, wenn die beiden Operanden denselben Wert besitzen, ansonsten `false`.

```
>> 4 == 4
=> true
>> 3.0 == 3
=> true
>> 3.0 == 3.1
=> false
>> 3 == "3"
=> false
```

---

10 Noch genauer gesagt modifiziert natürlich nur das Vorzeichen - einen Ausdruck (d.h., es kehrt sein bisheriges Vorzeichen um), während + dessen bisherigen Wert beibehält und daher jederzeit weggelassen werden kann.

```
>> "hello" == "hello"
=> true
>> "hello" == "helloworld"
=> false
>> "a" == "A"
=> false
```

- `!=` (Ungleichheit). Das Gegenstück zu `==`; das Ergebnis ist `true`, wenn die beiden Operanden verschieden sind, und `false`, wenn sie gleich sind.

```
>> 4 != 4
=> false
>> 3.0 != 3
=> false
>> 3.0 != 3.1
=> true
>> 3 != "3"
=> true
>> "hello" != "hello"
=> false
>> "hello" != "helloworld"
=> true
>> "a" != "A"
=> true
```

- `<` (kleiner als). `true`, wenn der linke Operand einen niedrigeren Wert besitzt als der rechte, ansonsten `false`. Beachten Sie, dass dieser und die nachfolgenden »gerichteten« Vergleichsoperatoren nicht für den Vergleich von Zahlen mit Strings verwendet werden dürfen (siehe das folgende Beispiel).

```
>> 2 < 3
=> true
>> 2 < 2
=> false
>> 3 < 2
=> false
>> 3 < "3"
ArgumentError: comparison of Fixnum with String failed
        from (irb):21:in `<'
        from (irb):21
>> "hello" < "world"
=> true
>> "hello" < "helloworld"
=> true
>> "hello" < "hello"
=> false
>> "a" < "A"
=> false
```

- `>` (größer als). Diese Operation liefert `true`, wenn der linke Operand einen höheren Wert besitzt als der rechte, sonst `false`.

```
>> 2 > 3
=> false
>> 2 > 2
```

```
=> false
>> 3 > 2
=> true
>> "helloworld" > "hello"
=> true
>> "hello" > "hello"
=> false
>> "hello" > "go"
=> true
>> "a" > "A"
=> true
```

- `<=` (kleiner oder gleich). Das Gegenteil von `>`: Das Ergebnis ist `true`, wenn der linke Operand *maximal so groß* wie der rechte ist (aber eben nicht größer).

```
>> 2 <= 3
=> true
>> 2 <= 2
=> true
>> 3 <= 2
=> false
>> "helloworld" <= "hello"
=> false
>> "hello" <= "hello"
=> true
>> "go" <= "hello"
=> true
```

- `>=` (größer oder gleich). Entsprechend das Gegenteil von `<`, so dass die Operation `true` ergibt, wenn der linke Operand mindestens so groß ist wie der rechte (aber nicht kleiner).

```
>> 2 >= 3
=> false
>> 2 >= 2
=> true
>> 3 >= 2
=> true
>> "helloworld" >= "hello"
=> true
>> "hello" >= "hello"
=> true
>> "hello" >= "go"
=> true
```

Ein speziellerer Vergleichsoperator ist `===`. Er erwartet auf der linken Seite einen Bereich wie `(0...10)` oder `('a'..'z')` und auf der rechten Seite einen einzelnen Wert. Der Rückgabewert ist `true`, wenn der Einzelwert in dem Bereich enthalten ist, und ansonsten `false`. Hier ein paar Beispiele:

```
>> (0..10) === 10
=> true
>> (0...10) === 10
=> false
```

```
>> ('a'..'z') === 'b'
=> true
>> ('a'..'z') === 'B'
=> false
```

Zu guter Letzt gibt es noch einen besonderen Operator, dessen Rückgabewert nicht `true` oder `false` ist: `<=>` liefert -1, wenn der erste Operand kleiner ist als der zweite, 0, wenn die beiden Operanden gleich groß sind, und +1, wenn der erste Operand größer ist:

```
>> 1 <=> 2
=> -1
>> 1 <=> 1
=> 0
>> 2 <=> 1
=> 1
```

Dieser Operator eignet sich hervorragend zur Definition von Sortierreihenfolgen eigener Klassen; dies wird in Kapitel 4 an einem Beispiel demonstriert.

Logische Operationen

Logische Aussagen werden, wie bereits erwähnt, vornehmlich durch Vergleichsoperatoren getroffen. Dabei kommt es recht häufig vor, dass die Ausführung eines Programmteils von mehreren Bedingungen abhängt. Deshalb gibt es die Möglichkeit, mehrere Vergleiche durch *logische Operatoren* miteinander zu verknüpfen.

Im Einzelnen stehen die folgenden logischen Operatoren zur Verfügung:

- `||` (alternative Schreibweise: `or`) – *logisches Oder*. Wenn Sie zwei Aussagen damit verknüpfen, ist die Gesamtaussage wahr, sobald *mindestens eine* der beiden Teilaussagen wahr ist. Das folgende Beispiel ergibt also eine wahre Aussage, nämlich dass mindestens einmal der Wert 3 gefunden wurde:

  ```
  >> a = 3
  >> b = 4
  >> a == 3 || b == 3
  => true
  ```

- `&&` (weitere Formulierung: `and`) – *logisches Und*. Hier ist die Gesamtaussage nur dann wahr, wenn *beide* Teilaussagen wahr sind. Das folgende Beispiel liefert daher eine falsche Aussage (`false`), da eben nicht *beide* Variablen den Wert 3 besitzen:

  ```
  >> a = 3
  >> b = 4
  >> a == 3 && b == 3
  => false
  ```

- `!` (oder `not`) – *logisches Nicht*. Dieser Operator dient nicht dazu, zwei Ausdrücke miteinander zu verknüpfen, sondern wird einem Ausdruck vorangestellt und kehrt dessen Wahrheitswert um. Hier einige Beispiele:

```
>> !1
=> false
>> !0
=> false
>> !false
=> true
>> 1 == 1 && 0 == 1
=> false
>> !(1 == 1 && 0 == 1)
=> true
```

Beachten Sie, dass `&&` und `||` das so genannte *Short-Circuit-Verfahren* (zu Deutsch »Kurzschluss«) verwenden: Sobald das Ergebnis feststeht, wird der Ausdruck nicht weiter ausgewertet. Das ist der Fall, wenn bei `||` der erste Operand wahr ist und so das Gesamtergebnis wahr macht – denn es reicht schließlich aus, dass ein Teilausdruck zutrifft. Umgekehrt ist der Fall bei `&&` klar, wenn der erste Teilausdruck falsch ist – da beide wahr sein müssen, macht dies bereits den gesamten Ausdruck falsch.

Bei der Überprüfung von Ausdrücken hat dies keine besonderen praktischen Auswirkungen, aber eine Besonderheit von Ruby (die es mit einigen UNIX-Shells sowie mit Perl teilt) besteht darin, dass sich mit Hilfe der logischen Operatoren nicht nur Ausdrücke, sondern auch Anweisungen verknüpfen lassen – dabei sorgt das Short-Circuit-Verfahren für die bedingte Ausführung. Eine genaue Erklärung und Beispiele finden Sie weiter unten im Abschnitt über Fallentscheidungen.

Bit-Operationen

Eine spezielle Gruppe von Operatoren bietet Zugriff auf die einzelnen Bits, aus denen gespeicherte Werte zusammengesetzt sind. Diese lassen sich auf verschiedene Arten verknüpfen oder verschieben. Hier zunächst alle Bit-Operatoren im Überblick:

- `|` (bitweises Oder); setzt alle Bits auf 1, die in *mindestens einem* der beiden Operanden 1 sind. Zum Beispiel:

```
>> 21 | 43
=> 63
```

Erläuterung (Binärform):

```
  010101
| 101011
--------
  111111
```

- `&` (bitweises Und); setzt alle Bits auf 1, die in *beiden* Operanden 1 sind:

```
>> 21 & 43
=> 1
```

Erläuterung:

```
  010101
& 101011
--------
  000001
```

- ^ (bitweises Exklusiv-Oder); setzt alle Bits auf 1, die in *genau einem* der beiden Operanden 1 sind:

```
>> 21 ^ 43
=> 62
```

Erläuterung:

```
  010101
^ 101011
--------
  111110
```

- << (Linksverschiebung; left shift); verschiebt den ersten Operanden um so viele Bits nach links, wie der zweite Operand angibt. Die leeren Stellen werden mit Nullen aufgefüllt. Zum Beispiel:

```
>> 21 << 2
=> 84
```

Erläuterung:

```
  10101   << 2
= 1010100
```

- >> (Rechtsverschiebung; right shift); verschiebt den ersten Operanden um so viele Bits nach rechts, wie der zweite Operand angibt. Rechts wegfallende Stellen werden ignoriert. Zum Beispiel:

```
>> 43 >> 2
=> 10
```

Erläuterung:

```
  101011 >> 2
= 1010
```

Die Bit-Operatoren werden häufig in der Kryptografie eingesetzt. Eine sehr einfache, aber recht effiziente Methode zum Codieren einer Wertefolge besteht beispielsweise darin, jeden Wert per Exklusiv-Oder mit derselben Konstante zu verknüpfen. Dabei bleibt nämlich kein Wert gleich, aber es passiert auch nicht, dass zwei unterschiedliche Ausgangswerte denselben Code erhalten. Zudem erfolgt die Entschlüsselung durch eine Wiederholung derselben Operation. Viele mathematisch hochkomplizierte Kryptoverfahren setzen Exklusiv-Oder daher als *einen von vielen* Schritten ein. Weiter unten in diesem Kapitel finden Sie ein entsprechendes Beispielprogramm.

Hier zunächst ein einfacheres Beispiel, das sich die Bit-Operatoren zunutze macht: Es geht um die Umrechnung von RGB-Farbwerten. Das sind die Lichtfarben, die durch eine additive Mischung des Rot-, Grün- und Blau-Anteils beschrieben wer-

den. Wie Sie vielleicht wissen, werden diese in HTML hexadezimal in der Form #RRGGBB (zwei Stellen für Rot, zwei für Grün und zwei für Blau) angegeben. Das Programm in Beispiel 2-3 erwartet die Eingabe der drei Komponenten Rot, Grün und Blau mit Werten von 0 (aus) bis 255 (Maximum); in derselben Form werden sie beispielsweise auch in Bildbearbeitungsprogrammen wie Photoshop verwendet. Daraufhin liefert das Skript den zugehörigen Hexadezimalwert sowie die nächstgelegene Webfarbe, das heißt eine Kombination aus Vielfachen von 51 (die hexadezimale Reihe 00, 33, 66, 99, CC, FF).

Beispiel 2-3: Der Farbumrechner, rgb.rb

```
 1  puts "RGB-Farbumrechner"
 2  puts
 3  puts "Bitte geben Sie die Komponenten der Farbe ein"
 4  print "Rot   (0-255): "
 5  # Rotwert einlesen und in Ganzzahl umwandeln
 6  r = gets.to_i
 7  # Unzulaessige Rotwerte auf 0 setzen
 8  r = 0 if r < 0 || r > 255

 9  print "Gruen (0-255): "
10  g = gets.to_i
11  g = 0 if g < 0 || g > 255

12  print "Blau  (0-255): "
13  b = gets.to_i
14  b = 0 if b < 0 || b > 255

15  # Den Gesamtwert berechnen
16  farbe = r << 16 | g << 8 | b

17  puts "Gesamtwert:       #{farbe}"

18  # Hexadezimalwert berechnen
19  hexfarbe = farbe.to_s(16)
20  # Auf 6 Stellen aufstocken
21  while hexfarbe.length < 6
22   hexfarbe = "0" + f
23  end
24  # In Grossbuchstaben umwandeln
25  hexfarbe.upcase!

26  puts "Hexadezimal:      ##{hexfarbe}"

27  # Naechstgelegene Web-Farbwerte berechnen
28  wr = (r + 25) / 51 * 51
29  wg = (g + 25) / 51 * 51
30  wb = (b + 25) / 51 * 51

31  # Gesamtwert der Webfarbe, hexadezimal usw.
32  webfarbe = wr << 16 | wg << 8 | wb
33  hexwebfarbe = webfarbe.to_s(16)
```

Beispiel 2-3: Der Farbumrechner, rgb.rb (Fortsetzung)

```
34   while hexwebfarbe.length < 6
35     hexwebfarbe = "0" + f
36   end
37   hexwebfarbe.upcase!

38   puts "Naechste Webfarbe: ##{hexwebfarbe}"
```

Speichern Sie das Skript unter dem Namen *rgb.rb* und starten Sie es wie üblich. Hier eine komplette Beispielausführung:

```
> ruby rgb.rb
Rot   (0-255): 217
Gruen (0-255): 81
Blau  (0-255): 20
Gesamtwert:        14242068
Hexadezimal:       #D95114
Naechste Webfarbe: #CC6600
```

Als Erstes werden die Zahlen eingegeben, mit Hilfe der Methode `to_i` in Integer umgewandelt und in Variablen gespeichert – in Zeile 6 beispielsweise `r` für den Rotwert. `to_i` setzt unzulässige Werte automatisch auf 0; in einem zweiten Schritt (zum Beispiel Zeile 8) werden unerwünschte, das heißt zu große oder zu kleine Eingaben ebenfalls auf 0 gesetzt. Das nachgestellte `if` als Fallentscheidung gilt dabei nur für die vorstehende Anweisung in derselben Zeile.

Die entscheidenden Zeilen für den Einsatz der Bit-Operatoren sind 16 und 32; sehen Sie sich als Beispiel die Berechnung des Gesamtwertes aus den ursprünglichen Eingaben an:

```
farbe = r << 16 | g << 8 | b
```

Der Rotwert wird um 16 Bit nach links verschoben, der Grünwert um 8 Bit und der Blauwert gar nicht. Verknüpft werden sie mittels bitweisem Oder. `+` wäre ebenfalls möglich, würde aber mehr Zeit für die Berechnung erfordern. Im Grunde könnten Sie dieselbe Operation – umständlicher und schwerer verständlich – auch mit arithmetischen Operatoren durchführen:

```
farbe = r * 65536 + g * 256 + b
```

Der fertig berechnete Wert wird anschließend in eine Hexadezimalzahl umgewandelt (Zeile 19 beziehungsweise 33):

```
hexfarbe = farbe.to_s(16)
```

Die Methode `to_s` zur Umwandlung in einen String kennt dazu praktischerweise die Angabe einer Basis (in diesem Fall 16); wenn Sie sie weglassen, wird automatisch 10 gewählt.

Danach wird eine `while`-Schleife ausgeführt: Solange die Hexadezimalzahl noch weniger als 6 Stellen hat, wird jeweils eine weitere 0 davorgesetzt. Der anschlie-

ßende `upcase!`-Aufruf (Zeile 25/37) wandelt die Buchstaben in der Hexadezimalzahl in Großbuchstaben um. Das Ausrufezeichen hinter einem Methodennamen steht – wie hier – in der Regel dafür, dass der geänderte Wert nicht nur zurückgeliefert, sondern auch in der Variablen selbst gespeichert wird.

Zu guter Letzt ist noch interessant, wie die drei Web-Farbwerte berechnet werden. Hier als Beispiel der Rotwert (Zeile 28):

```
wr = (r + 25) / 51 * 51
```

Gesucht sind, wie erwähnt, ganzzahlige Vielfache von 51. Diese sind leicht zu berechnen, indem der bisherige (ganzzahlige) Wert durch 51 geteilt wird – die Nachkommastellen fallen dadurch weg. Anschließend wird wieder mit 51 multipliziert. Zuvor wird allerdings noch 25, also knapp die Hälfte, addiert, damit tatsächlich der *nächstgelegene* und nicht einfach der *nächstniedrigere* Wert zurückgeliefert wird (dieses Verfahren entspricht dem mathematisch korrekten Runden).

Weitere Operationen

Ruby kennt noch einige weitere Operationen, die hier kurz vorgestellt werden.

Die Wertzuweisung an Variablen war bereits Thema. Dazu wird der Operator `=` verwendet. Da der Ausdruck hinter dem Gleichheitszeichen zuerst komplett ausgewertet wird, kann auch die Variable selbst darin vorkommen – in diesem Fall wird ihr bisheriger Wert in den Ausdruck eingesetzt. Zum Beispiel:

```
>> zahl = 3
=> 3
>> zahl = zahl + 1
=> 4
```

Genau für diese direkte Manipulation des bisherigen Variablenwerts gibt es Kurzschreibweisen, die aus dem jeweiligen Operator und einem angehängten Gleichheitszeichen gebildet werden. Das naheliegendste Beispiel sind die arithmetischen Operatoren. Statt

```
zahl = zahl + 1
```

können Sie auch Folgendes schreiben:

```
zahl += 1
```

Probieren Sie (mit einer bereits definierten Variablen!) zum Beispiel auch folgende Varianten aus:

```
zahl -= 7     # bisherigen Wert um 7 vermindern
zahl *= 3     # mit 3 multiplizieren
zahl /= 2     # durch 2 teilen
```

Dasselbe funktioniert auch mit den Bit-Operatoren `&`, `|`, `^`, `<<` und `>>`.

Die normalerweise arithmetischen Operatoren + und * besitzen nebenbei noch eine besondere Bedeutung für Strings. Mit + können Sie Strings aneinanderfügen. Zum Beispiel:

```
>> "Ruby " + "on Rails"
=> "Ruby on Rails"
```

Dasselbe geht natürlich auch mit Variablen, die Strings enthalten:

```
>> ruby = "Ruby"
>> leer = " "
>> on_rails = "on Rails"
>> ruby + leer + on_rails
"Ruby on Rails"
```

Beachten Sie, dass das nur funktioniert, wenn alle Komponenten bereits Strings sind; es findet *keine* automatische Typumwandlung statt wie etwa in JavaScript. Der Versuch liefert eine Fehlermeldung. Wenn Sie sich also bei irgendeinem Element der Verknüpfung nicht sicher sind, hängen Sie `.to_s` an, den Aufruf der weiter unten beschriebenen String-Konvertierungs-Methode. Hier ein Beispiel:

```
>> "Die Antwort auf die grosse Frage ist " + 42.to_s
=> "Die Antwort auf die grosse Frage ist 42"
```

Der Operator * kann einen String vervielfältigen, sofern der vordere Operand der String und der hintere eine ganze Zahl ist. Zum Beispiel:

```
>> "*" * 40
=> "****************************************"
>> "Hallo " * 4
=> "Hallo Hallo Hallo Hallo "
```

Ein letzter Operator, der Ihnen vielleicht manchmal begegnet, ist der *ternäre* (dreigliedrige, also aus drei Operanden bestehende) Fallentscheidungsoperator. Er hat die Form:

```
Bedingung ? Dann-Wert : Sonst-Wert
```

Die Bedingung ist üblicherweise eine Vergleichsoperation oder ein anderer logischer Ausdruck. Wenn sie wahr ist, wird der Dann-Wert als Wert des Gesamtausdrucks gewählt, andernfalls der Sonst-Wert. Zum Beispiel:

```
>> a = 10
>> a < 10 ? "kleiner" : "nicht kleiner"
=> "nicht kleiner"
>> a == 10 ? "zehn" : "nicht zehn"
=> "zehn"
```

Rangfolge der Operatoren

Die Regel »Punkt- vor Strichrechnung« kennen Sie wahrscheinlich noch aus der Schule. `4 + 5 * 6` ist nach dieser Regel 34 und nicht etwa 54. In einer Programmiersprache gibt es ebenfalls eine Rangfolge für Operatoren. Die folgende Liste zeigt diese Reihenfolge für die wichtigsten Ruby-Operatoren:

- `[]` (Menge)
- `**` (Potenz)
- `!`, `+` (Vorzeichen), `-` (Vorzeichen)
- `*`, `/`, `%`
- `>>` (Bitverschiebung links), `<<` (Bitverschiebung rechts)
- `&` (bitweise Und)
- `^` (bitweise Exklusiv-Oder), `|` (bitweise Oder)
- `<=`, `<`, `>`, `>=`
- `<=>`, `==`, `===`, `!=`, `=~` (entspricht Muster, s.u.), `!~` (entspricht Muster nicht)
- `&&`
- `||`
- `..`, `...`
- `?:`
- `=`, `+=`, `-=`, `*=`, `/=` usw.
- `not`
- `or`, `and`

Je weiter oben ein Operator in dieser Liste steht, desto stärker bindet er und desto früher wird er ausgeführt. Natürlich können Sie beliebig tief verschachtelte Klammern verwenden, um die Rangfolge nach Ihren Wünschen zu ändern. Beispielsweise ergibt `(4 + 5) * 6` tatsächlich 54.

Methodenaufrufe

Viele weitere Ausdrücke entstehen durch den Aufruf von Methoden. Das können sowohl vordefinierte Methoden sein als auch die (weiter unten behandelten) selbst geschriebenen. Eine Methode ist nichts weiter als eine benannte Abfolge von Anweisungen. Manche Methoden erwarten Eingabewerte (so genannte *Argumente*); andere geben wiederum Werte zurück – genau diese Sorte kann in Ausdrücken verwendet werden.

Da Ruby objektorientiert ist und alle Elemente (Literale, Variablen usw.) Objekte irgendeines Typs sind, gibt es für jedes von ihnen einige vordefinierte Methoden der Form

```
Element.Methode
```

Eine wichtige Methode, die *jedes* Ruby-Objekt besitzt, ist `class` – sie liefert die Klasse, das heißt den Datentyp des Objekts, zurück. Probieren Sie es mit beliebigen Variablen oder Literalen aus:

```
>> 2.class
=> Fixnum
>> 2.2.class
=> Float
>> "2".class
=> String
>> true.class
=> TrueClass
```

Daneben gibt es auch eine Reihe *globaler* Methoden in der Form

```
Methode(Argument, ...)
```

Bei vielen dieser Methoden können Sie die Klammern um die Argumente *noch* weglassen – wenn Warnungen aktiviert sind, erhalten Sie die Mitteilung:

```
warning: parenthesize argument(s) for future version
```
[11]

Wenn Sie die Klammern setzen, müssen Sie auch darauf achten, dass zwischen dem Methodennamen und der öffnenden Klammer *kein* Leerzeichen stehen darf.

Ein Beispiel für eine nützliche globale Methode ist `rand` – sie liefert eine zufällige Fließkommazahl zwischen 0 und 1. Probieren Sie es einige Male aus (natürlich werden Sie andere Werte erhalten als die hier abgedruckten):

```
>> rand
=> 0.451022176722365
>> rand
=> 0.827507832034271
```

Optional können Sie `rand` eine Ganzzahl als Argument übergeben. Wenn Sie das tun, erhalten Sie eine zufällige Ganzzahl zwischen 0 und dem Argument -1. Wenn Sie also beispielsweise einen Würfel simulieren möchten, können Sie Folgendes eingeben:

```
>> rand(6) + 1
=> 4
>> rand(6) + 1
=> 2
>> rand(6) + 1
=> 5
```

Falls Sie sich hier gegen Argumentklammern entscheiden sollten, also

```
rand 6
```

schreiben, müssen Sie den gesamten `rand`-Aufruf in Klammern setzen:

```
(rand 6) + 1
```

11 Übersetzung: »Warnung: Setzen Sie Argument(e) für zukünftige Versionen in Klammern«

Andernfalls wird die Addition zuerst ausgeführt, und `rand(7)` liefert nicht, wie gewünscht, eine Zahl zwischen 1 und 6, sondern eine zwischen 0 und 6.

Es folgen einige weitere Beispiele für bekannte Methoden, geordnet nach Kategorien.

Mathematische Methoden

Viele Methoden stammen aus dem Bereich der Mathematik. Hier nur einige wichtige im Überblick; im Verlauf des Buchs lernen Sie noch weitere kennen.

`Zahl.abs` liefert den absoluten Wert (Betrag) einer Zahl, das heißt ihren Wert ohne Vorzeichen.

```
>> 3.abs
=> 3
>> -3.abs
=> 3
```

`Zahl.ceil` gibt die nächsthöhere ganze Zahl zurück (oder die Zahl selbst, falls es sich bereits um eine ganze Zahl handelt):

```
>> 2.ceil
=> 2
>> 2.2.ceil
=> 3
>> 2.7.ceil
=> 3
```

`Zahl.floor` liefert entsprechend die nächstniedrigere Ganzzahl:

```
>> 2.floor
=> 2
>> 2.2.floor
=> 2
>> 2.7.floor
=> 2
```

`Zahl.round` rundet mathematisch, das heißt unter 0.5 wird ab-, ansonsten aufgerundet:

```
>> 2.round
=> 2
>> 2.49.round
=> 2
>> 2.5.round
=> 3
```

Es gibt keine eingebaute Funktion, die auf eine bestimmte Anzahl von Stellen hinter dem Komma rundet. Wenn Sie diese Möglichkeit benötigen, multiplizieren Sie die Fließkommazahl einfach mit der Zehnerpotenz, die der Anzahl der gewünschten Nachkommastellen entspricht. Das Ergebnis können Sie mit `round` runden. Anschließend müssen Sie wieder durch dieselbe Zehnerpotenz teilen, aber diesmal als Fließkommazahl. Das folgende Beispiel rundet eine Zahl nach diesem Verfahren auf zwei Stellen hinter dem Komma:

```
betrag = 19.4879381
betrag = (betrag * 100).round / 100.0
puts betrag
```

Das Ergebnis lautet erwartungsgemäß 19.49. Im übernächsten Kapitel erfahren Sie, wie Sie die Standardmethode `round` der Fließkommazahlen um eine optionale Nachkommastellenzahl erweitern können.

Auf viele weitere mathematische Methoden können Sie über die Standardklasse `Math` zugreifen. `Math.sqrt(Zahl)` liefert beispielsweise die Quadratwurzel, `Math.log(Zahl)` den natürlichen Logarithmus und `Math.exp(Zahl)` das Gegenstück dazu, nämlich e^x. Zum Beispiel:

```
>> Math.sqrt(9)
=> 3.0
>> Math.sqrt(2)
=> 1.4142135623731
>> Math.log(10)
=> 2.30258509299405
>> Math.exp(1)
=> 2.71828182845905
```

`Math.sin(Zahl)`, `Math.cos(Zahl)` und `Math.tan(Zahl)` sind die trigonometrischen Funktionen Sinus, Kosinus und Tangens. Beachten Sie, dass die Winkel im Bogenmaß angegeben werden müssen. Da 360° Winkelmaß im Bogenmaß 2π entsprechen, können Sie eine Grad-Angabe in das Bogenmaß umrechnen, indem Sie sie durch 180 teilen und mit π multiplizieren. Einen einigermaßen genauen Näherungswert der Kreiszahl liefert die eingebaute Konstante `Math::PI`. Die folgenden Beispiele berechnen den Sinus von 90° und den Kosinus von 0°:

```
>> Math.sin(90 / 180 * Math::PI)
=> 0.0
>> Math.cos(0)
=> 1.0
>> Math::PI
=> 3.14159265358979
```

String-Methoden

Auch zur Bearbeitung von Strings gibt es eine Reihe von Methoden; einige von ihnen wurden bereits in Beispielen eingesetzt. Die Methode `String.length` gibt etwa die Länge des Strings in Zeichen zurück:

```
>> "Hallo Welt".length
=> 10
>> "".length
=> 0
```

Andere Methoden beschäftigen sich mit dem Inhalt eines Strings. Hier einige im Überblick:

- `String.upcase` wandelt alle enthaltenen Buchstaben in Großbuchstaben um:

  ```
  >> "Hallo Welt".upcase
  => "HALLO WELT"
  ```

- `String.downcase` macht dagegen alle Buchstaben zu Kleinbuchstaben:

  ```
  >> "Hallo Welt".downcase
  => "hallo welt"
  ```

- `String.swapcase` kehrt die Groß- und Kleinschreibung jedes Buchstaben um:

  ```
  >> "Hallo Welt".swapcase
  => "hALLO wELT"
  ```

- `String.reverse` dreht den ganzen String herum:

  ```
  >> "Hallo Welt".reverse
  => "tleW ollaH"
  ```

- `String.chop` liefert den String ohne sein letztes Zeichen zurück (wird manchmal zum Entfernen des abschließenden Zeilenumbruchs verwendet):

  ```
  >> "Hallo Welt".chop
  => "Hallo Wel"
  >> "Hallo Welt\n".chop
  => "Hallo Welt"
  ```

- `String.chomp` entfernt das letzte Zeichen nur dann, wenn es tatsächlich ein Zeilenumbruch ist, und ist damit sicherer als `chop`:

  ```
  >> "Hallo Welt".chomp
  => "Hallo Welt"
  >> "Hallo Welt\n".chomp
  => "Hallo Welt"
  ```

All diese Umwandlungsmethoden haben noch eine zweite Variante mit angehängtem Ausrufezeichen. Diese Methoden können nicht auf Literale, sondern nur auf String-Variablen angewendet werden. Sie liefern nicht einfach den entsprechenden Wert zurück, sondern verändern den Inhalt der Variablen dauerhaft. Probieren Sie es aus:

```
>> text = "We apologize for the inconvenience."
>> text.upcase
=> "WE APOLOGIZE FOR THE INCONVENIENCE."
>> text
```

```
=> "We apologize for the inconvenience."
>> text.upcase!
=> "WE APOLOGIZE FOR THE INCONVENIENCE."
>> text
=> "WE APOLOGIZE FOR THE INCONVENIENCE."
```

Interessant ist schließlich der Zugriff auf die einzelnen Zeichen eines Strings, indem Sie die Nummer des gewünschten Zeichens (bei 0 beginnend) in eckigen Klammern dahintersetzen (hinter den Kulissen gibt es tatsächlich eine Methode namens []):

```
>> "Hallo Welt"[0]
=> 72
```

Allerdings bekommen Sie nicht das erwartete Zeichen "H" zurück, sondern eine Zahl. Es handelt sich dabei um den numerischen Zeichencode. Diesen können Sie mit Hilfe der Methode chr wieder in das entsprechende Zeichen umwandeln:

```
>> 72.chr
=> "H"
>> "Hallo Welt"[1].chr
=> "a"
```

Wenn Sie zwei durch Komma getrennte Ganzzahlen in die eckigen Klammern setzen, erhalten Sie dagegen einen Teilstring, wobei die erste Zahl den Startpunkt und die zweite die maximale Länge angibt. Beispiele:

```
>> "Hallo Welt"[1, 2]
=> "al"
>> "Hallo Welt"[6, 1]
=> "W"
>> "Hallo Welt"[9, 3]
=> "t"
```

Wie bei einem Hash können Sie sogar einen String als Index verwenden. Rückgabewert ist der entsprechende Teilstring selbst, falls er im durchsuchten String vorkommt, ansonsten nil:

```
>> "Koeln"["oel"]
=> "oel"
>> "Duesseldorf"["Dom"]
=> nil
```

Am praktischsten ist, dass Sie diese Indizes auf String-Variablen anwenden können, um die Werte der gefundenen Teile zu ändern:

```
>> satz = "PHP ist die beste Sprache der Welt."
>> satz["PHP"] = "Ruby"
=> "Ruby"
>> satz
=> "Ruby ist die beste Sprache der Welt."

>> halbsatz = "Dieser Satz kein Verb."
>> halbsatz[11, 2] = " hat "
>> halbsatz
=> "Dieser Satz hat ein Verb."
```

Um den numerischen Code eines einzelnen literalen Zeichens zu erhalten, genügt es übrigens auch, diesem Zeichen ein Fragezeichen voranzustellen:

```
>> ?A
=> 65
>> ?a
=> 97
>> ?0
=> 48
```

Innerhalb von String-Literalen können Sie einzelne Zeichen ebenfalls durch ihren Code angeben: `\nnn` erwartet eine Oktalzahl zwischen 0 und 176 (dezimal 126). Hier ein extremes und praxisfernes Beispiel:

```
>> "\110\101\114\114\117"
=> "HALLO"
```

Mit `\xNN` können Sie den Zeichencode auch hexadezimal angeben, diesmal von `"\x0"` bis `"\x7E"`. Hier das obige Beispiel in dieser Schreibweise:

```
>> "\x48\x41\x4C\x4C\x4F"
=> "HALLO"
```

In der Praxis wird dieses Feature eher für Steuerzeichen wie etwa `"\12"` beziehungsweise `"\xA"` (den Zeilenumbruch, auch `"\n"`) verwendet.

Array-Methoden

Genau wie Strings verfügen auch Arrays (wie bereits erwähnt) über die Methode `length`, die in diesem Fall die Anzahl der enthaltenen Elemente zurückgibt:

```
>> [1, 2, 3].length
=> 3
```

Da Arrays in Ruby eine variable Länge besitzen, gibt es eine Reihe interessanter Methoden, mit denen Sie Elemente an verschiedenen Stellen hinzufügen oder entfernen können:

- `array.push(Wert, ...)` fügt einen oder mehrere Werte am hinteren Ende hinzu. Zum Beispiel:

  ```
  >> zwei_hoch_n = [1, 2, 4, 8]
  => [1, 2, 4, 8]
  >> zwei_hoch_n.push(16)
  => [1, 2, 4, 8, 16]
  >> zwei_hoch_n.push(32, 64)
  => [1, 2, 4, 8, 16, 32, 64]
  ```

- `array.pop` entfernt das letzte Element aus dem Array und gibt es zurück:

  ```
  >> wert = zwei_hoch_n.pop
  => 64
  >> zwei_hoch_n
  => [1, 2, 4, 8, 16, 32]
  ```

- `array.unshift(Wert, ...)` fügt am Anfang des Arrays einen oder mehrere Werte ein:

```
>> monate = %w(April Mai Juni)
=> ["April", "Mai", "Juni"]
>> monate.unshift("Maerz")
=> ["Maerz", "April", "Mai", "Juni"]
>> monate.unshift("Januar", "Februar")
=> ["Januar", "Februar", "Maerz", "April", "Mai", "Juni"]
```

- `array.shift` ist das Gegenstück zu `unshift`; das erste Element wird entfernt und zurückgeliefert:

```
>> todo = %w(Analyse Entwurf Implementierung Test Dokumentation)
=> ["Analyse", "Entwurf", "Implementierung", "Test", "Dokumentation"]
>> programmierschritt = todo.shift
=> "Analyse"
>> todo
=> ["Entwurf", "Implementierung", "Test", "Dokumentation"]
```

- `array.slice(Index)` liefert das Element des Arrays an der gewünschten Stelle zurück, genau wie ein einfaches `array[Index]`. Die spezielle Form `array.slice!(Index)` entfernt das entsprechende Element dauerhaft aus dem Array:

```
>> kaese = %w(Camembert Brie Gouda Edamer Emmentaler)
=> ["Camembert", "Brie", "Gouda", "Edamer", "Emmentaler"]
>> essen = kaese.slice!(1)
=> "Brie"
>> kaese
=> ["Camembert", "Gouda", "Edamer", "Emmentaler"]
```

Einige weitere Methoden manipulieren die Reihenfolge der Elemente. Mit `sort` werden die Elemente in Zeichensatzreihenfolge sortiert (siehe die Beschreibung der Vergleichsoperatoren weiter oben). Hier ein Beispiel:

```
>> tiere = %w(Loewe Zebra Affe Nashorn Elefant Bison)
=> ["Loewe", "Zebra", "Affe", "Nashorn", "Elefant", "Bison"]
>> tiere.sort
=> ["Affe", "Bison", "Elefant", "Loewe", "Nashorn", "Zebra"]
```

Auch hier gibt es die Variante `sort!`, die das ursprüngliche Array dauerhaft sortiert abspeichert, während das normale `sort` nur eine sortierte Kopie des Arrays als Wert zurückgibt.

`reverse` beziehungsweise `reverse!` kehrt ein Array um, ähnlich wie der gleichnamige String-Operator. Das folgende Beispiel speichert die Tierliste sortiert ab und dreht sie dann ebenso dauerhaft herum:

```
>> tiere.sort!
["Affe", "Bison", "Elefant", "Loewe", "Nashorn", "Zebra"]
>> tiere.reverse!
["Zebra", "Nashorn", "Loewe", "Elefant", "Bison", "Affe"]
```

Methoden zur Typumwandlung

Am Rande wurde bereits erwähnt, dass Sie die verschiedenen einfachen Ruby-Datentypen bis zu einem gewissen Grad ineinander konvertieren können. Dazu stehen diverse Standardmethoden zur Verfügung.

Die Methode `to_s` wandelt ein beinahe beliebiges Objekt in einen String um; der Inhalt dieses Strings hängt dabei von der Art des Objekts ab. Besonders häufig kommt es vor, dass Zahlen in Strings konvertiert werden sollen. Das funktioniert ohne Probleme, denn sowohl Ganz- als auch Fließkommazahlen verfügen über diese Methode:

```
>> 23.to_s
=> "23"
>> Math::PI.to_s
=> "3.14159265358979"
```

Einen Unterschied gibt es allerdings: Bei den Ganzzahlen können Sie als optionalen Parameter das Zahlensystem des Zielstrings angeben – mögliche Werte reichen von 2 (Dualsystem) bis 36. Die Höchstgrenze ergibt sich nicht aus irgendeiner mathematischen Gegebenheit (es sind beliebige Zahlensysteme denkbar), sondern weil das 36er-System bereits die Ziffern 0-9 und die Buchstaben A-Z verbraucht; für alles darüber Hinausgehende gibt es keine allgemeingültige Vereinbarung. Wenn Sie die Basis weglassen, wird 10 angenommen, wie das obige Beispiel zeigt.

Ein erstes Beispiel für die Angabe einer Basis haben Sie bereits weiter oben im Farbkonverter-Programm gesehen, in dem Zahlen ins Hexadezimalsystem umgerechnet wurden. Hier noch einige weitere Beispiele mit häufig genutzten Zahlensystemen:

```
>> 100.to_s(2)
=> "1100100"
>> 1000.to_s(8)
=> "1750"
>> 100000.to_s(16)
=> "186a0"
```

(Beispiele mit einer abwegigeren Basis können Sie bei Bedarf gern selbst ausprobieren.)

Wenn Sie versuchen, bei der Umwandlung einer Fließkommazahl die Basis anzugeben, erhalten Sie dagegen eine Fehlermeldung (`Math::E` ist die Euler'sche Zahl e=2. 71828182845905..., also die Basis des natürlichen Logarithmus):

```
>> Math::E.to_s(16)
ArgumentError: wrong number of arguments (1 for 0)
```

Umgekehrt können Sie auch Strings (und andere Objekte) in Zahlen umwandeln, indem Sie ihre Methode `to_i` für Ganzzahlen beziehungsweise `to_f` für Fließkommazahlen aufrufen. Diese Methoden sind überaus fehlertolerant: Beim ersten Zei-

chen, das nicht zum gewünschten Zahlentyp gehören kann, brechen sie ab, und bei völlig ungültigen Strings liefern sie einfach 0. Hier einige Beispiele:

```
>> "42".to_i
=> 42
>> "42".to_f
=> 42.0
>> "3 Maenner im Schnee".to_i
=> 3
>> "Die 3 von der Tankstelle".to_f
=> 0.0
>> "3.9".to_i
=> 3
```

Bei `to_i` gibt es wiederum eine Besonderheit: Diesmal können Sie optional das ursprüngliche Zahlensystem angeben, aus dem der String als Zahl interpretiert werden soll (der Standardwert ist wieder 10). Das führt natürlich zu unterschiedlichen Werten für denselben String:

```
>> "1000".to_i
=> 1000
>> "1000".to_i(2)
=> 8
>> "1000".to_i(8)
=> 512
=> "1000".to_i(16)
=> 4096
```

Wenn Sie beide Umwandlungsoperationen kombinieren, haben Sie das nötige Handwerkszeug, um Zahlen *beliebiger* Systeme – innerhalb des zulässigen Rahmens – ineinander zu konvertieren. Zum Beispiel (die Kommentare können Sie natürlich weglassen):

```
>> "1000".to_i(8).to_s(16)        # Oktal nach Hexadezimal
=> "200"
>> "ABC".to_i(16).to_s(2)         # Hexadezimal nach Dual
=> "101010111100"
```

Interessant sind auch die Methoden, mit denen Sie Strings und Arrays ineinander umwandeln können:

```
string.split
```

teilt einen String an einem beliebigen Trennzeichen in Elemente eines Arrays auf, und

```
array.join
```

verbindet die Elemente des Arrays zu einem String, getrennt durch eine frei wählbare Zeichensequenz.

`split` verwendet standardmäßig Whitespace (beliebig viele Leerzeichen, Tabulatoren und Zeilenumbrüche) als Trennzeichen. Hier zwei Beispiele:

```
>> "Dies ist ein Satz".split
=> ["Dies", "ist", "ein", "Satz"]
>> "Dies\nist                    noch\tein   \n     Satz".split
=> ["Dies", "ist", "noch", "ein", "Satz"]
```

Wenn Sie einzelne Zeichen haben möchten, geben Sie einfach den leeren String ("") als Trennzeichen an:

```
>> "Alle Zeichen".split("")
=> ["A", "l", "l", "e", " ", "Z", "e", "i", "c", "h", "e", "n"]
```

Sie können aber auch eine beliebige andere Zeichensequenz angeben. Auch dafür sehen Sie hier ein Beispiel:

```
>> ice_strecke="Hannover - Dortmund - Essen - Duesseldorf - Koeln"
>> ice_strecke.split(" - ")
=> ["Hannover", "Dortmund", "Essen", "Duesseldorf", "Koeln"]
```

Statt eines literalen Strings können Sie auch einen regulären Ausdruck, also ein Suchmuster, als Trennzeichen verwenden. Das wird weiter unten im entsprechenden Abschnitt besprochen.

Bei `join` können Sie das Trennzeichen ebenfalls weglassen. In diesem Fall werden die Array-Elemente aber einfach ohne Abstand zusammengefügt, was meistens nicht erwünscht ist. Hier ein Beispiel für dieses Umwandlungsverfahren:

```
>> wochentage = %w(Mo Di Mi Do Fr Sa So)
=> ["Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"]
>> wochentage.join(", ")
=> "Mo, Di, Mi, Do, Fr, Sa, So"
```

Interessant ist schließlich noch die Möglichkeit, Bereiche mittels `to_a` in Arrays umzuwandeln. Zum Beispiel:

```
>> (1..10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> ('A'...'G').to_a
=> ["A", "B", "C", "D", "E", "F"]
```

Kontrollstrukturen

Wie Sie inzwischen wissen, ist ein Ruby-Skript im einfachsten Fall eine Abfolge von Anweisungen, die der Reihe nach abgearbeitet werden. Wenn dies unverrückbar so bleiben müsste, könnten Sie mit Ruby allerdings kaum irgendwelche realen Probleme lösen. Eine vollwertige Programmiersprache braucht Hilfsmittel, um den geradlinigen Programmfluss zu unterbrechen und an andere Stellen zu verzweigen – entweder abhängig von Bedingungen oder einfach mehrmals hintereinander. Genau zu diesem Zweck gibt es die in diesem Abschnitt beschriebenen *Kontrollstrukturen*. Es gibt sie in zwei Geschmacksrichtungen: *Fallentscheidungen* führen Anweisungen abhängig von Bedingungen aus, und *Schleifen* erledigen dieselbe Aufgabe mehrmals hintereinander.

Fallentscheidungen

Eine Fallentscheidung sorgt dafür, dass bestimmte Programmschritte nur ausgeführt werden, wenn eine Bedingung zutrifft. Ruby kennt eine Reihe unterschiedlicher Arten von Fallentscheidungen.

if, elsif, else und unless

Die einfachste Variante prüft die Bedingung und führt eine oder mehrere Anweisungen aus, wenn diese zutrifft. Die Syntax lautet:

```
if Bedingung
  Anweisung(en)
  ...
end
```

Das folgende kurze Beispiel fragt den Benutzer nach seinem Alter. Wenn es größer oder gleich 18 ist, wird eine Erfolgsmeldung ausgegeben:

```
puts "Zutritt nur fuer Erwachsene."
print "Bitte geben Sie Ihr Alter ein: "
# Alter einlesen und in Ganzzahl umwandeln
alter = gets.to_i
if alter >= 18
  puts "Herzlich willkommen!"
end
```

Es wäre natürlich wünschenswert, wenn auch bei »Minderjährigen« eine Meldung erscheinen würde. Zu diesem Zweck können Sie einen `else`-Block (»andernfalls«) hinzufügen. Dieser wird genau dann ausgeführt, wenn die Bedingung *nicht* zutrifft. Eine solche Fallentscheidung hat folgendes Schema:

```
if Bedingung
  "Dann"-Anweisung(en)
  ...
else
  "Sonst"-Anweisung(en)
  ...
end
```

Damit können Sie die Altersüberprüfung wie folgt ergänzen:

```
if alter >= 18
  puts "Herzlich willkommen!"
else
  puts "Fuer Minderjaehrige ist der Zutritt leider verboten."
  puts "Bitte kommen Sie in #{18-alter} Jahren wieder!"
end
```

Probieren Sie verschiedene Fälle aus, wie in Abbildung 2-4 gezeigt.

Sie können zwischen der Bedingung und der ersten von ihr abhängenden Anweisung optional das Schlüsselwort `then` schreiben, das in einigen älteren Programmiersprachen Pflicht ist. Zum Beispiel:

```
if alter >= 18 then
  puts "Herzlich willkommen!"
end
```

In diesem Fall können Sie die Bedingung und die erste Anweisung auch ohne Zeilenumbruch schreiben:

```
if alter >= 18 then puts "Herzlich willkommen!"; end
```

Eine zulässige Kurzschreibweise für `then` ist der Doppelpunkt:

```
if alter >= 18 : puts "Herzlich willkommen!"; end
```

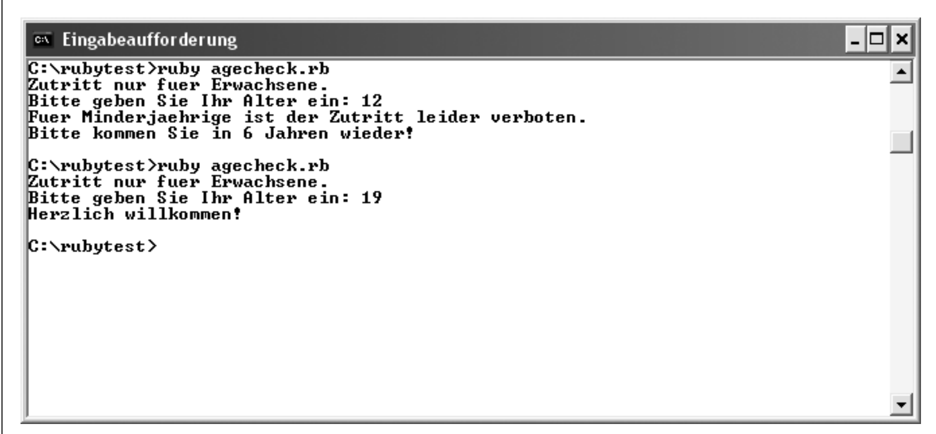

Abbildung 2-4: Die Altersüberprüfung im Einsatz

Manchmal kommt es vor, dass Sie im Sonst-Fall eine weitere Bedingung prüfen müssen, in deren Sonst-Fall wieder eine und so weiter. Für diese Aufgabe stellt Ruby das spezielle Schlüsselwort `elsif` zur Verfügung. Die allgemeine Syntax einer solchen verschachtelten Fallentscheidung sieht so aus:

```
if Bedingung
  Anweisung(en)
  ...
elsif Bedingung
  Anweisung(en)
  ...
[elsif ...]
else
  Anweisung(en)
end
```

Dieses Verfahren wird oft für eine Art »stufenweise Filterung« verwendet. Das folgende Beispiel ordnet verschiedenen Punktzahlbereichen in einer Prüfung nacheinander die unterschiedlichen Noten zu:

```
print "Bitte Punktzahl eingeben: "
punkte = gets.to_i
if punkte < 30
  puts "Ungenuegend"
elsif punkte < 50
  puts "Mangelhaft"
elsif punkte < 67
  puts "Ausreichend"
elsif punkte < 80
  puts "Befriedigend"
elsif punkte < 92
  puts "Gut"
else
  puts "Sehr gut"
end
```

Beachten Sie, dass der letzte Fall durch ein einfaches `else` vollständig abgedeckt wird – denn selbst wenn jemand mehr als 100 Punkte (von 100 möglichen!) erreichen sollte, gibt es keine bessere Note als »Sehr gut«.

Wenn Sie nur eine einzige Anweisung von einer Bedingung abhängig machen wollen und auch keinen `else`-Teil brauchen, gibt es eine kürzere Alternative zu

```
if Bedingung
  Anweisung
end
```

Es handelt sich um ein nachgestelltes `if` in der Form

```
Anweisung if Bedingung
```

Hier ein Beispiel, das »Geschafft!« ausgibt, wenn die Variable `punkte` einen Wert über 100 hat:

```
puts "Geschafft!" if punkte > 100
```

Ruby kennt sogar eine Umkehrung von `if`: `unless` überprüft eine Bedingung und führt die entsprechenden Anweisungen nur dann aus, wenn diese nicht zutrifft. Das folgende Beispiel gibt eine Fehlermeldung aus, wenn die Eingabe *nicht* 1 oder 2 lautet:

```
puts "Moechten Sie (1) noch mal oder (2) beenden?"
print "=> "
wahl = gets.to_i
unless wahl == 1 || wahl == 2
  puts "Ungueltige Eingabe!"
end
```

Für solche einzelnen Anweisungen gibt es `unless` auch als nachgestellte Fallentscheidung:

```
puts "Ungueltige Eingabe!" unless wahl == 1 || wahl == 2
```

Natürlich lässt sich jede Bedingung durch Verneinung so umformulieren, dass Sie statt `unless` auch einfach `if` schreiben können. Beachten Sie dabei aber, dass Sie dann Und-Verknüpfungen bei der Auflösung gegebenenfalls durch Oder ersetzen müssen und umgekehrt. Beispielsweise lautet das Gegenteil von

```
wahl == 1 || wahl == 2
```

also

```
!(wahl == 1 || wahl == 2)
```

und ohne die Klammern:

```
wahl != 1 && wahl != 2
```

Die ursprüngliche Oder-Bedingung ist nämlich erst dann falsch, wenn *beide* Einzelbedingungen falsch sind. Das Ganze lässt sich formal so ausdrücken und wird als De Morgan-Theorem[12] bezeichnet:

```
!(a || b) == (!a && !b)
!(a && b) == (!a || !b)
```

Dabei können `a` und `b` zwei Vergleiche oder sonstige Operationen sein, die `true` oder `false` ergeben.

case-when-Fallentscheidungen

Manchmal müssen Sie eine Variable mit verschiedenen Einzelwerten vergleichen und je nach Ergebnis unterschiedliche Anweisungen ausführen. Das funktioniert prinzipiell mit `if-elsif-else`-Strukturen, aber speziell für diese Aufgabe gibt es auch eine besondere Struktur. Sie hat folgende Form:

```
case Variable
when Wert1, ...
  Anweisung(en)
  ...
when Wert2, ...
  Anweisung(en)
  ...
...
else
  # Kein Wert trifft zu
  Anweisung(en)
  ...
end
```

Je nachdem, welchen Wert die Variable zurzeit besitzt, werden die Anweisungen im entsprechenden `when`-Teil ausgeführt. Jedes `when` kann einen oder optional mehrere durch Kommata getrennte Werte überprüfen.

---

12 Nach Augustus De Morgan (1806-1871), einem schottischen Mathematiker.

Der else-Teil ist optional. Gerade bei der Prüfung von Benutzereingaben, die besonders häufig mit case-when erfolgt, sollten Sie ihn aber verwenden, um unzulässige Werte abzufangen.

Das folgende Beispiel fragt zuerst nach einer Längenangabe in Metern. Danach erwartet es die Eingabe des Anfangsbuchstaben (oder der offiziellen Abkürzung) einer Maßeinheit, in die umgerechnet werden soll. Falls ein unzulässiger Buchstabe eingegeben wird, erfolgt eine Fehlermeldung über den else-Teil:

```
print "Laenge in Metern: "
laenge = gets.to_f
print "Umrechnen in: (M)illimeter, (C)entimeter, (K)ilometer? "
einheit = gets.downcase
case einheit
when "m", "mm"
  puts "#{laenge}m = #{laenge * 1000}mm"
when "c", "cm"
  puts "#{laenge}m = #{laenge * 100}cm"
when "k", "km"
  puts "#{laenge}m = #{laenge / 1000}km"
else
  puts "Ungueltige Masseinheit!"
end
```

Speichern Sie das Skript und probieren Sie verschiedene Fälle aus:

```
>ruby umrechnung.rb
Laenge in Metern: 726
Umrechnen in: (M)illimeter, (C)entimeter, (K)ilometer? c
726.0m = 72600.0cm

>ruby umrechnung.rb
Laenge in Metern: 726
Umrechnen in: (M)illimeter, (C)entimeter, (K)ilometer? k
726.0m = 0.726km

>ruby umrechnung.rb
Laenge in Metern: 726
Umrechnen in: (M)illimeter, (C)entimeter, (K)ilometer? d
Ungueltige Masseinheit!
```

Logische Operatoren als Fallentscheidungen

Im Abschnitt über logische Operationen wurde bereits erwähnt, dass Sie diese manchmal auch als praktischen Ersatz für Fallentscheidungen verwenden können. Beachten Sie, dass Sie dafür in Ruby die Textvarianten and und or benutzen müssen; && und || funktionieren dort nicht.

Das Ganze funktioniert aufgrund der bereits beschriebenen Short-Circuit-Funktionsweise dieser Operatoren: Sobald das Ergebnis feststeht, wird die Operation nicht weiterberechnet. Das ist bei and der Fall, wenn der erste Operand false ist, denn das

macht den Gesamtausdruck `false`. Bei `or` erfolgt der Abbruch dagegen, wenn der erste Operand `true` ist, weil der Gesamtausdruck bereits `true` sein muss.

Auf diese Weise wird `and` zu einem `if`-Ersatz, der die Anweisung nur dann ausführt, wenn die Bedingung zutrifft. Zum Beispiel:

```
punkte > 100 and puts "Sie haben gewonnen!"
```

Entsprechend kann `or` als `unless`-Stellvertreter fungieren, denn die Anweisung hinter dem Operator wird ausgeführt, wenn die Bedingung nicht wahr ist:

```
punkte > 100 or puts "Das war wohl nichts!"
```

Fallentscheidungen in Ausdrücken

Interessanterweise können Sie nicht nur logische Operatoren als Fallentscheidungen verwenden, sondern umgekehrt auch Fallentscheidungen direkt an Stellen schreiben, wo Ausdrücke erwartet werden. Wichtig ist nur, dass Sie für die einzelnen Fälle keine Anweisungen, sondern eben Ausdrücke angeben.

Beachten Sie, dass auch dies eine Ruby-Spezialität ist. Eine Übertragung auf andere Programmiersprachen liefert höchstwahrscheinlich nur eine Fehlermeldung.

Das folgende `if`-Beispiel wiederholt die weiter oben vorgestellte Zuordnung von Punkten zu Noten auf kompaktere Weise:

```
print "Bitte Punktzahl eingeben: "
punkte = gets.to_i
note = if punkte < 30
  6
elsif punkte < 50
  5
elsif punkte < 67
  4
elsif punkte < 80
  3
elsif punkte < 92
  2
else
  1
end
puts "Ihre Note: #{note}"
```

Hier ein weiteres Beispiel, das `case`/`when` einsetzt, um die obigen Noten in die Textform zu übertragen:

```
notentext = case note
when 1
  "Sehr gut"
when 2
  "Gut"
```

```
when 3
  "Befriedigend"
when 4
  "Ausreichend"
when 5
  "Mangelhaft"
when 6
  "Ungenuegend"
else
  "Ungueltige Note"
end
puts "Note in Textschreibweise: #{notentext}"
```

Einfache Schleifen

In Programmen müssen Sie bestimmte Schritte sehr oft mehrmals durchführen – sei es nacheinander mit unterschiedlichen Werten oder solange eine bestimmte Bedingung erfüllt ist. Für diese Aufgabe bieten Programmiersprachen das Konzept der *Schleife*. In diesem Abschnitt werden die verschiedenen Schleifenvarianten von Ruby vorgestellt.

Daneben kennt Ruby allerdings noch das Konzept des *Iterators*, der für viele Anwendungen nützlicher ist als eine Schleife. Es handelt sich dabei um die automatische Ausführung von Anweisungen für alle Elemente einer Menge. Aufgrund ihrer zentralen Bedeutung werden Iteratoren weiter unten in einem eigenen Abschnitt vorgestellt.

while- und until-Schleifen

Die bekannteste Schleife, die mit beinahe identischer Syntax auch in vielen anderen Programmiersprachen verfügbar ist, hat folgende Syntax:

```
while Bedingung
  Anweisung(en)
  ...
end
```

Genau wie bei einer `if`-Fallentscheidung wird zuerst eine Bedingung geprüft. Trifft diese zu, werden die nachfolgenden Anweisungen bis zum `end` ausgeführt. Danach wird die Bedingung allerdings erneut geprüft, und *solange* (englisch *while*) sie zutrifft, werden die verschachtelten Anweisungen immer wieder ausgeführt.

Das folgende Beispiel fragt nach einer Eingabe, solange diese noch nicht `"j"` oder `"n"` lautet:

```
wahl = ""
while wahl != "j" && wahl != "n"
  print "Moechten Sie noch mal (j/n)? "
  wahl = gets.chomp.downcase
end
```

Ausführungsbeispiel:

```
Moechten Sie noch mal (j/n)? a
Moechten Sie noch mal (j/n)? b
Moechten Sie noch mal (j/n)? n
```

Auf dieselbe Weise können Sie sich beispielsweise auch die Quadrate der Zahlen 1 bis 10 anzeigen lassen:

```
i = 1
while i <= 10
  puts "#{i}^2 = #{i ** 2}"
  i += 1
end
```

Die Ausgabe sieht erwartungsgemäß so aus (Ausschnitt):

```
1^2 = 1
2^2 = 4
3^2 = 9
...
10^2 = 100
```

Schleifen können beliebig tief ineinander verschachtelt werden. Um aus den Zahlen von 1 bis 100 alle Primzahlen herauszusuchen, brauchen Sie bereits zwei verschachtelte Schleifen – eine für die Zahlen selbst und eine für ihre potenziellen Teiler. Das hier verwendete Verfahren ist ziemlich langsam und daher keinesfalls zu empfehlen. Bei der verschwindend geringen Anzahl von nur 100 Kandidaten fällt das jedoch nicht ins Gewicht.

```
i = 1
while i <= 100
  # Annahme: i ist eine Primzahl
  prim = true
  j = 2
  # Potenzielle Teiler bis i/2 testen
  while j <= i/2
    # i durch j teilbar?
    if i % j == 0
      # Keine Primzahl!
      prim = false
      # Pruefschleife (j) vorzeitig verlassen
      break
    end
    j += 1
  end
  # Ausgabe, falls Primzahl
  print "#{i}  " if prim
  i += 1
end
```

Wenn Sie das Skript eingeben, speichern und ausführen, erhalten Sie folgende Liste:

```
1  2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53  59
61  67  71  73  79  83  89  97
```

Die einzige wirklich neue Anweisung in diesem Skript ist `break`. Diese verlässt sofort die Schleife, die gerade ausgeführt wird, und macht mit der ersten Zeile nach dem `end` weiter. Bei verschachtelten Schleifen – wie hier – wird jeweils die innerste verlassen.

Daneben sollten Sie sich das Konzept merken, der Zustandsvariablen `prim` zuerst »auf Verdacht« den Wert `true` zuzuweisen: Auf diese Weise ist ein schleifenbasierter Test möglich, bei dem sich nachträglich das Gegenteil der Behauptung herausstellen könnte. Das Ganze erinnert an mathematische Beweise durch Widerlegen des Gegenteils.

Genau wie `if` das Gegenstück `unless` besitzt, gibt es auch zu `while` eine Umkehrung: `until` führt die Schleifenanweisungen aus, *bis* die Bedingung zutrifft. Damit lässt sich etwa das obige Eingabebeispiel logischer formulieren (beachten Sie wieder die Vertauschung von Und und Oder bei der Verneinung):

```
wahl = ""
until wahl == "j" || wahl == "n"
  print "Moechten Sie noch mal (j/n)? "
  wahl = gets.chomp.downcase
end
```

Endlosschleifen

Eine echte »Endlosschleife« ist eigentlich eine Situation, die durch Programmierfehler entsteht und unerwünscht ist. In Ruby können Sie sie allerdings manchmal als Hilfsmittel verwenden, wenn die Bedingung zu Beginn des ersten Schleifendurchlaufs noch gar nicht geprüft werden kann. Innerhalb der Schleife können Sie dann ein per `if`-Fallentscheidung geprüftes `break` verwenden, um diese abzubrechen.

Die grundlegende Syntax der Endlosschleife sieht so aus:

```
loop do
  Anweisung(en)
  ...
end
```

Das folgende Beispiel gibt immer wieder neue »Würfelwürfe« aus, bis der Benutzer die Frage nach einem weiteren Wurf verneint:

```
loop do
  wurf = rand(6) + 1
  puts "Gewuerfelt: #{wurf}"
  print "Noch einmal wuerfeln (j/n)? "
  wahl = gets.chomp.downcase
  # Schleife bei "n" verlassen
  break if wahl == "n"
end
```

Ausführungsbeispiel:

```
Gewuerfelt: 6
Noch einmal wuerfeln (j/n)? j
```

```
Gewuerfelt: 3
Noch einmal wuerfeln (j/n)? n
```

Diese Schleifenkonstruktion wurde übrigens bereits für das Einführungsbeispiel dieses Kapitels, den Taschenrechner, verwendet.

Es folgt ein weiteres Beispielprogramm. Der Computer »denkt sich« per Zufallsgenerator eine Zahl zwischen 1 und 100, die der Benutzer erraten muss. Nach jeder Eingabe erhält er die Information, ob die Eingabe zu groß, zu klein oder genau richtig ist. Geben Sie das Skript zunächst ein:

```
puts "Zahlenraten"
puts "==========="
puts
puts "Raten Sie eine Zahl zwischen 1 und 100."
puts

# Computer "denkt sich" Zahl
zufallszahl = rand(100) + 1
# Versuche werden separat gezaehlt
versuch = 0
# Endlosschleife
loop do
  versuch += 1
  print "#{versuch}. Versuch: "
  zahl = gets.to_i
  if zahl < zufallszahl
    puts "Diese Zahl ist zu klein."
  elsif zahl > zufallszahl
    puts "Diese Zahl ist zu gross."
  else
    # Zahl ist korrekt -- Schleife verlassen
    puts "Volltreffer!"
    break
  end
end
```

Geben Sie das Skript ein und probieren Sie es aus. Um die Zahl möglichst schnell zu erraten, brauchen Sie nur jeweils den übrig gebliebenen Bereich zu halbieren. Zum Beispiel:

```
Zahlenraten
===========

Raten Sie eine Zahl zwischen 1 und 100.

1. Versuch: 50
Diese Zahl ist zu gross.
2. Versuch: 25
Diese Zahl ist zu klein.
3. Versuch: 38
Diese Zahl ist zu klein.
4. Versuch: 44
```

```
Diese Zahl ist zu klein.
5. Versuch: 47
Diese Zahl ist zu gross.
6. Versuch: 46
Volltreffer!
```

for-Schleifen

Die dritte Schleifenart verwendet das Schlüsselwort `for` und ermöglicht die Durchführung von Anweisungen für alle Elemente eines Bereichs, eines Arrays oder einer anderen Art von Menge. Die Syntax lautet:

```
for Variable in Menge
  Anweisung(en)
  ...
end
```

Innerhalb der Schleifenanweisungen steht das jeweilige Element der Menge unter dem Namen der angegebenen Variablen zur Verfügung.

Auf diese Weise lässt sich beispielsweise das Programm mit den Primzahlen wesentlich prägnanter schreiben:

```
for i in 1..100
  # Annahme: i ist eine Primzahl
  prim = true
  # Potenzielle Teiler bis i/2 testen
  for j in 2..i/2
    # i durch j teilbar?
    if i % j == 0
      # Keine Primzahl!
      prim = false
      # Pruefschleife (j) vorzeitig verlassen
      break
    end
  end
  # Ausgabe, falls Primzahl
  print "#{i}  " if prim
end
```

Wie Sie sehen, sind die Zuweisung des jeweiligen Anfangswerts und die Erhöhung um 1 verschwunden. Beides wird automatisch durch die `for`-Anweisung geregelt.

Die `for`-Schleife eignet sich auch hervorragend zur Bearbeitung aller Elemente eines Arrays. Hier ein einfaches Beispiel:

```
sprachen = %w(Ruby Perl PHP Java)
for s in sprachen
  puts "Ich programmiere in #{s}."
end
```

Das ergibt natürlich folgende Ausgabe:

```
Ich programmiere in Ruby.
Ich programmiere in Perl.
```

```
Ich programmiere in PHP.
Ich programmiere in Java.
```

Wenn Sie die Elemente eines Hashes entsprechend bearbeiten möchten, können Sie dessen Eigenschaft keys zu Hilfe nehmen, die nacheinander alle Schlüssel liefert:

```
tlds = {
  "de" => "Deutschland",
  "at" => "Oesterreich",
  "ch" => "Schweiz"
}
for t in tlds.keys
  puts "Die TLD #{t} gehoert zu #{tlds[t]}"
end
```

Hier der Vollständigkeit halber das zu erwartende Ergebnis:

```
Die TLD de gehoert zu Deutschland
Die TLD ch gehoert zu Schweiz
Die TLD at gehoert zu Oesterreich
```

Als etwas praktischeres for-Schleifen-Beispiel sehen Sie hier den weiter oben angekündigten Codierer, der bitweises Exklusiv-Oder verwendet, um jeweils dieselben Bits der Zeichencodes zu maskieren. Nach dem Start wird der Benutzer aufgefordert, den Text einzugeben, und danach eine Zahl zwischen 1 und 255 als Schlüssel. Die for-Schleife geht vom ersten Zeichen (0) bis zum letzten (die um 1 verminderte Länge, da ... den Endwert ausschließt). Die Anwendung eines Index in eckigen Klammern auf einen String liefert, wie bereits erwähnt, den Zeichencode. Dieser wird jeweils mittels Exklusiv-Oder mit dem Schlüssel maskiert. Der neue Wert wird per chr wieder in ein Zeichen umgewandelt und schließlich ausgegeben.

Geben Sie zunächst den Code ein:

```
print "Text: "
# Text einlesen, Zeilenumbruch abschneiden
text = gets.chomp
print "Schluessel (1-255): "
key = gets.to_i

# Fehlermeldung & Ende bei unzulaessigem Schluessel
if key < 1 or key > 255
  puts "Ungueltiger Schluessel!"
  exit
end

# Verschluesselter Text ist zunaechst leer
encrypt = ''

# Schleife ueber alle Zeichen des Textes
for i in 0...text.length
  # Durch Exklusiv-Oder maskierten Zeichencode wieder in Zeichen umwandeln
```

```
    # und an verschluesselten Text anfuegen
    encrypt += ((text[i]^key).chr)
  end
  print "Verschluesselter Text: #{encrypt}"
```

Speichern Sie das kurze Skript als *crypt.rb* und starten Sie es. Hier ein Ausführungsbeispiel:

```
> ruby crypt.rb
Text: Dieser Text soll verschluesselt werden.
Schluessel (1-255): 9
Verschluesselter Text: M`lzl{)]lq})zfee)¦l{zjae|lzzle})~l{mlg'
```

Aufgrund der Funktionsweise der Exklusiv-Oder-Operation kann der verschlüsselte Text durch die erneute Ausführung mit demselben Schlüssel wieder entschlüsselt werden:

```
> ruby crypt.rb
Text: M`lzl{)]lq})zfee)¦l{zjae|lzzle})~l{mlg'
Schluessel (1-255): 9
Verschluesselter Text: Dieser Text soll verschluesselt werden.
```

Die einzige neue Anweisung in diesem Skript ist `exit`. Sie verlässt das Programm sofort. Bei kurzen Konsolenprogrammen können Sie schwerwiegende Eingabefehler am effektivsten auf diese Weise beheben.

Sie brauchen den oft mit merkwürdigen Sonderzeichen gespickten verschlüsselten Text zur Entschlüsselung nicht selbst einzugeben, sondern können ihn kopieren und einfügen. Unter Windows müssen Sie dazu die Titelleiste des Konsolenfensters mit der rechten Maustaste anklicken und die Untermenüpunkte des Befehls *Bearbeiten* wählen: zuerst *Markieren* – was Sie daraufhin mit der Maus tun können –, dann *Kopieren* (oder einfach **Enter**) und schließlich *Einfügen*. Noch viel einfacher geht es in den Terminalfenstern der meisten UNIX/Linux-Varianten: Fahren Sie mit gedrückter linker Maustaste über den zu kopierenden Text. Drücken Sie anschließend die mittlere Maustaste (oder das Scrollrad oder notfalls beide Maustasten zusammen), um den kopierten Text an der aktuellen Textcursor-Position einzufügen.

Mustervergleiche mit regulären Ausdrücken

Ähnlich wie Perl ist Ruby beliebt für seine fortgeschrittenen Textverarbeitungsmöglichkeiten. Das liegt vor allem an der umfassenden Unterstützung *regulärer Ausdrücke* (englisch *regular expressions* oder kurz *Regexp*, manchmal sogar *Regex*). Es handelt sich dabei um Suchmuster, mit denen sich die inhaltlichen Eigenschaften von Strings beschreiben lassen. Sie können sie sowohl zur Formulierung von Bedingungen in Fallentscheidungen verwenden als auch zum Ersetzen bestimmter Gruppen von Zeichenfolgen in bestehenden Strings.

Einführung

Was Einsteiger als Allererstes über reguläre Ausdrücke wissen sollten, ist, dass diese normalerweise keinen ganzen String beschreiben. Ein *Match* (Treffer) ist bereits dann gegeben, wenn der reguläre Ausdruck einen beliebigen Teil des untersuchten Strings beschreibt. Ein ganz einfaches Beispiel: Lautet der reguläre Ausdruck einfach nur `a`, so ist er erfüllt, wenn der String an einer beliebigen Stelle mindestens ein `a` enthält.

Formal werden reguläre Ausdrücke in Ruby entweder zwischen zwei Slashes (`/`) gesetzt oder aber durch den speziellen Quoting-Operator `%r` gekennzeichnet (und dann wie bei `%q` und ähnlichen Quoting-Helfern in diverse Klammern oder andere Zeichen eingeschlossen). Um also das besagte `a` als regulären Ausdruck zu verwenden, können Sie es als `/a/` oder beispielsweise als `%r(a)` schreiben.

Um nun einen String mit einem regulären Ausdruck zu vergleichen (der einfachste Anwendungsfall), wird der *Matching-Operator* `=~` verwendet. Probieren Sie in `irb` einmal den regulären Ausdruck `/a/` mit verschiedenen Strings (mit und ohne enthaltenes »a«) aus und betrachten Sie die Ergebnisse:

```
>> "aber" =~ /a/
=> 0
>> "hallo" =~ /a/
=> 1
>> "Variable" =~ /a/
=> 1
>> "Regexp" =~ /a/
=> nil
>> "Abend" =~ /a/
=> nil
```

Wie Sie sehen, lautet der Rückgabewert `nil`, wenn der reguläre Ausdruck nicht zutrifft – ein großes »A« ist demzufolge kein »a«, weil reguläre Ausdrücke normalerweise zwischen Groß- und Kleinschreibung unterscheiden (weiter unten lernen Sie einen Modifier dafür kennen). Trifft der reguläre Ausdruck dagegen zu, ist das Ergebnis eine ganze Zahl – die Position des Zeichens im String, bei dem das erste Vorkommen des regulären Ausdrucks beginnt.

Da gemäß den weiter oben beschriebenen Regeln für Wahrheitswerte *jede* Zahl (auch die 0) als wahr gilt und `nil` als falsch, können Sie die Matching-Operation ohne Weiteres als Kriterium für eine `if`-Fallentscheidung verwenden. Das folgende Beispiel gibt bekannt, ob die Benutzereingabe mindestens ein `e` enthält oder nicht:

```
eingabe = gets
if eingabe =~ /e/
  puts "e gefunden"
else
  puts "Kein e gefunden"
end
```

Speichern Sie diesen Codeblock als Skript ab und führen Sie ihn aus. Geben Sie beim ersten Mal einen Text mit `e` ein, danach einen ohne:

```
Hello
e gefunden
Hallo
Kein e gefunden
```

Die Syntax der regulären Ausdrücke

Reguläre Ausdrücke stellen eine ganz eigene Sprache dar, die Ihnen nach dem Erlernen übrigens nicht nur in Ruby von Nutzen ist, sondern auch in vielen anderen Programmiersprachen, in der UNIX-Shell oder in zahlreichen Texteditoren. Das grundlegende Konzept stammt nicht aus der Informatik, sondern wurde unter anderem von dem amerikanischen Sprachwissenschaftler *Noam Chomsky* zur Beschreibung der Syntax und Grammatik natürlicher Sprachen mit mathematischen Mitteln entwickelt. Die Regexp-Implementierungen in Programmiersprachen und anderen Computerprogrammen besitzen zwar nur unterschiedlich große Teilmengen der allgemeinen Fähigkeiten regulärer Ausdrücke, sind aber dennoch überaus leistungsfähig.

Die Regexp-Implementierung in Ruby ist besonders umfangreich; noch mehr bietet nur noch die entsprechende Bibliothek von Perl. Deshalb ist es leider unmöglich, hier sämtliche Regexp-Fähigkeiten von Ruby darzustellen. Betrachten Sie diesen Abschnitt vielmehr als solides Tutorial für die Grundlagen; wenn Sie damit zurechtkommen, können Sie sich später leicht zusätzliches Wissen aneignen. Als weiterführende Lektüre empfehle ich Ihnen das hervorragende Buch *Reguläre Ausdrücke* von Jeffrey E. F. Friedl (O'Reilly Verlag), in dem das Thema umfassend, mit wissenschaftlicher Präzision und zugleich äußerst unterhaltsam präsentiert wird.[13]

Zeichen und Zeichenfolgen

Der einfachste Fall eines regulären Ausdrucks – ein einzelnes Zeichen – wurde bereits in der Einleitung vorgestellt. Ein Treffer ist immer dann gegeben, wenn das entsprechende Zeichen mindestens einmal im untersuchten String enthalten ist. Beispielsweise liefert

```
"Hallo" =~ /l/
```

den Wert 2 (erstes `"l"` in `"Hallo"` an Position 2, wie üblich ab 0 gezählt). Der Fall

```
"Hallo" =~ /L/
```

liefert dagegen keinen Treffer, das heißt den Wert `nil`, weil `"Hallo"` kein großes `"L"` enthält.

13 Die englische Originalversion *Mastering Regular Expressions* ist 2006 bereits in der 3. Auflage erschienen.

Besonders nützlich sind einzelne Zeichen für die Gegenprobe, das heißt für das Nichtzutreffen eines regulären Ausdrucks. Eine Operation in der Form

```
String !~ Regexp
```

liefert `true`, wenn der reguläre Ausdruck auf *keinen* Teil des Strings zutrifft, und `false`, wenn er eben doch darin vorkommt. Das folgende Beispiel verbietet die Ziffer 0 im gesamten String `eingabe`:

```
if eingabe !~ /0/
  puts "Korrekte Eingabe!"
else
  puts "Bitte keine 0 eingeben!"
end
```

Wenn Sie innerhalb des regulären Ausdrucks mehrere Zeichen hintereinander setzen, müssen diese für einen Treffer genau in der angegebenen Reihenfolge und direkt hintereinander im String vorkommen. Hier einige Beispiele, die Sie in `irb` nachvollziehen können:

```
>> "Hallo" =~ /al/
=> 1
>> "Hallo" =~ /la/
=> nil
>> "Hammel" =~ /al/
=> nil
```

Der erste Versuch liefert einen Treffer, denn die Zeichenfolge `al` kommt an Position 1 in `"Hallo"` vor. Der Test auf `la` scheitert dagegen, denn die Reihenfolge ist falsch. Auch die Untersuchung auf `al` in `"Hammel"` liefert keinen Treffer, da `a` und `l` in diesem Wort nicht direkt aufeinanderfolgen.

Wenn Sie Zeichen verwenden möchten, die innerhalb regulärer Ausdrücke eine besondere Bedeutung haben, müssen Sie diesen einen Backslash (\) voranstellen. Es handelt sich insbesondere um folgende Zeichen: `/ . ? * + - | ( ) [ ] ^ $`

Zeichengruppen und -bereiche

Das bisher gezeigte Matching einzelner Zeichen und Zeichenfolgen ist noch nichts besonders Aufregendes und lässt sich auch ohne reguläre Ausdrücke erledigen. Viel interessanter ist die Möglichkeit, eine Auswahl passender Zeichen anzugeben. Solche Zeichenmengen werden grundsätzlich in eckige Klammern geschrieben.

Die einfachste Variante ist eine simple Aufzählung einzelner Zeichen als Alternativen. Beispielsweise steht `[aeiou]` für einen beliebigen kleingeschriebenen Vokal, nämlich eines der Zeichen `a`, `e`, `i`, `o` oder `u`. Probieren Sie es ruhig einmal aus:

```
>> "hello" =~ /[aeiou]/
=> 1
>> "42" =~ /[aeiou]/
=> nil
```

Wenn Sie denselben (längeren) regulären Ausdruck mehrfach in einem Skript verwenden oder in `irb` ausprobieren möchten, können Sie ihn problemlos in einer Variablen speichern wie jedes andere Literal. Anschließend wird hinter dem `=~` (oder an anderen Stellen, wo reguläre Ausdrücke erlaubt sind) einfach die Variable statt dem literalen Regexp eingesetzt. Zum Beispiel:

```
>> re = /[aeiou]/
=> /[aeiou]/
>> "hallo" =~ re
=> 1
>> "grrrr" =~ re
=> nil
```

Die Zeichengruppe braucht keineswegs in alphabetischer beziehungsweise Zeichensatz-Reihenfolge geschrieben zu werden; jedes Zeichen in der Gruppe ist gleichberechtigt. Die Gruppenangaben `[ab]` und `[ba]` sind also beispielsweise identisch: An der entsprechenden Stelle darf entweder ein `a` oder ein `b` stehen.

Statt einer einfachen Zeichengruppe können Sie auch einen Zeichenbereich angeben. Dabei wird das erste und das letzte Zeichen einer aufeinanderfolgenden Reihe zulässiger Zeichen angegeben, getrennt durch einen `-`. Einige Beispiele:

- `[a-z]` – ein beliebiger Kleinbuchstabe
- `[A-Z]` – ein beliebiger Großbuchstabe
- `[0-9]` – eine beliebige Ziffer
- `[f-m]` – einer der Buchstaben f, g, h, i, j, k, l oder m

Sie können einzelne Zeichen und Zeichenbereiche innerhalb der eckigen Klammern beliebig mischen. Die letzte Ziffer, das heißt die Prüfziffer einer ISBN (internationale Buchnummer; das vorliegende Buch hat die 3-89721-478-4), kann beispielsweise 0 bis 9 oder X lauten.[14] Das können Sie wie folgt in einem regulären Ausdruck angeben:

```
[0-9X]
```

Ein weiteres Beispiel ist die folgende Verknüpfung mehrerer Bereiche, die alle zulässigen Ziffern einer Hexadezimalzahl umfasst, wobei die Buchstaben A bis F vorsichtshalber sowohl in Großschreibung als auch in Kleinschreibung vorkommen:

```
[0-9A-Fa-f]
```

Wenn Sie innerhalb der eckigen Klammern als erstes Zeichen ein `^` setzen, bedeutet das, dass an der entsprechenden Stelle ein beliebiges Zeichen *außer* den angegebe-

14 Das liegt daran, dass die Prüfziffer durch eine Modulo-11-Operation, das heißt den Rest der Division durch 11, gebildet wird. Der mögliche Rest 10 wird dabei durch das X dargestellt. Seit Anfang 2007 gibt es übrigens neue, dreizehnstellige ISBN-Nummern, aber noch werden beide Systeme nebeneinander verwendet.

nen stehen soll. Das Folgende steht beispielsweise für »ein Zeichen, aber keine Ziffer«:

```
[^0-9]
```

Wenn an irgendeiner Stelle ein Zeichen stehen soll, das kein Buchstabe und keine Ziffer ist, können Sie Folgendes schreiben:

```
[^0-9A-Za-z]
```

Natürlich können Sie Zeichenmengen mit beliebigen anderen Regex-Konstrukten mischen, insbesondere mit den bereits gezeigten literalen Zeichenfolgen. Angenommen, Sie möchten zwei Stellen nach einem `0x` als Hexadezimalzahl interpretieren. Dann lautet der reguläre Ausdruck zur Prüfung der Gültigkeit der beiden hexadezimalen Ziffern (mit den bisher erläuterten Mitteln; weiter unten lernen Sie verschiedene Verkürzungen kennen):

```
/0x[0-9A-Fa-f][0-9A-Fa-f]/
```

Die allgemeinste aller Zeichengruppen schließlich ist ein einfacher Punkt (`.`). Dieser steht für *genau ein beliebiges* Zeichen. Der folgende Ausdruck passt etwa auf `"Ball"`, `"Bill"` und `"Bell"`, aber auch auf vollkommen sinnlose Zeichenfolgen wie `"Bxll"`, `"BGll"`, `"B3ll"` oder `"B%ll"`:

```
/B.ll/
```

Quantifizierer

Die bisher gezeigten Optionen machen reguläre Ausdrücke noch nicht sonderlich interessant. Nützlicher werden sie erst durch weitere Features. Beispielsweise können Sie durch so genannte *Quantifizierer* (englisch *quantifiers*) genau angeben, wie oft bestimmte Zeichen, Zeichenmengen und beliebige Teilausdrücke im String vorkommen sollen oder dürfen.

Der einfachste Quantifizierer ist ein Fragezeichen (`?`): Wenn Sie es hinter ein Element setzen, darf dieses Element an der entsprechenden Stelle einmal oder auch keinmal vorkommen. Das folgende Beispiel passt auf die String-Variable `str`, wenn darin entweder `"eis"` oder `"es"` (aber beispielsweise nicht `"ens"`) vorkommt, weil das `i` wortwörtlich in Frage gestellt wird:

```
str =~ /ei?s/
```

Beachten Sie, dass `/ei?s/` etwas völlig anderes ist als `/e[^i]s/`: `[^i]` steht für *genau ein Zeichen, das kein i ist*, während `i?` *das Zeichen i oder gar kein Zeichen* repräsentiert. So passt `/ei?s/` wie gesagt auf `"eis"` und `"es"`, während `/e[^i]s/` beispielsweise auf `"ems"`, `"ens"` oder `"ess"` zutrifft.

Auch das Auftreten von Zeichenmengen lässt sich per Quantifizierer genauer beschreiben. Das folgende Beispiel beschreibt eine ein- bis zweistellige Zahl, denn die zweite Ziffer `[0-9]` wird durch das Fragezeichen optional gesetzt:

```
[0-9][0-9]?
```

Sehr wichtig für die Arbeit mit Quantifizierern ist die Möglichkeit, Teile regulärer Ausdrücke zu *gruppieren*. Das geschieht durch runde Klammern: Wenn Sie einen Quantifizierer hinter einen geklammerten Teilausdruck setzen, wird der gesamte Teilausdruck modifiziert. Der folgende Teilausdruck findet etwa sowohl den `"Administrator"` als auch die gängige Abkürzung `"Admin"`:

```
Admin(istrator)?
```

Die Klammerung lässt sich sogar beliebig ineinander verschachteln. Hier sehen Sie einen Teilausdruck, der neben Administrator und Admin auch den Systemadministrator und den Sysadmin findet:

```
(Sys(tem)?)?[Aa]dmin(istrator)?
```

Die Bestandteile dieses regulären Ausdrucks sollte man sich genauer ansehen:

- `(Sys(tem)?)?` wird durch die äußeren Klammern und das Fragezeichen insgesamt optional gesetzt und enthält zusätzlich den verschachtelten Teilausdruck `(tem)?`, der seinerseits optional ist. Das Ganze passt mit anderen Worten auf `"System"`, `"Sys"` oder gar nichts. Da ein regulärer Ausdruck aber auf einen beliebigen Teil eines Strings zutreffen kann, bedeutet dieses »gar nichts«, dass vor dem nachfolgenden »Admin« auch ein ganz anderer Text stehen darf, so dass der gesamte Ausdruck beispielsweise auch auf `"Netzwerkadministrator"` passt.
- `[Aa]` dürfte klar sein: An dieser Stelle wird entweder ein großes oder ein kleines A erwartet, aber auf keinen Fall ein anderes Zeichen.
- `dmin` ist eine einfache Zeichenfolge, die genau in dieser Reihenfolge und ohne Unterbrechung im String vorkommen *muss*.
- `(istrator)?` schließlich wird durch das Fragezeichen wiederum zum optionalen Bestandteil.

Wenn Sie den Teilausdruck

```
(Sys(tem)?)?[Aa]dmin(istrator)?
```

vollständig auswerten, passt er auch auf einige unwahrscheinliche Wörter wie »SystemAdmin« oder »Sysadministrator«. Es bleibt Ihnen selbst überlassen, *wie* genau Sie Ihre regulären Ausdrücke schreiben – in den meisten Fällen genügt es, wie hier, dass alle gewünschten Treffer enthalten sind, und Sie brauchen sich keine Gedanken über nicht existierende Wörter zu machen, auf die das Muster zufällig ebenfalls passt. Wenn es um Sicherheitsaspekte geht, etwa bei der Eingabe einer Kreditkartennummer, müssen Sie solche Fälle jedoch explizit ausschließen.

Hier ein kleines Beispielskript, mit dem Sie den obigen Ausdruck testen können. Geben Sie zuerst die folgenden Zeilen ein:

```
re = /((Sys(tem)?)?[Aa]dmin(istrator)?)/

loop do
  print "> "
  ein = gets.chomp
  break if ein == "q"
  if ein =~ re
    puts "  \"#{$1}\" passt!"
  else
    puts "  Passt nicht."
  end
end
```

Speichern Sie das Ganze, beispielsweise als *admin.rb*, und führen Sie es aus. Nun können Sie beliebig viele Zeilen eingeben, und es wird jeweils geprüft, ob der komplizierte reguläre Ausdruck auf Ihre Eingabe passt. Sobald Sie ein einzelnes q eingeben, wird das Programm beendet. Hier eine Beispielausführung:

```
> ruby admin.rb
> Systemadministrator
  "Systemadministrator" passt!
> Systemadministratorin
  "Systemadministrator" passt!
> Netzwerkadministrator
  "administrator" passt!
> Der Admin ist in Urlaub
  "Admin" passt!
> Sysad
  Passt nicht.
> q
```

Beachten Sie, dass der gesamte Ausdruck noch einmal zusätzlich in runden Klammern steht (hier zur Verdeutlichung fett gesetzt und durch ^ markiert):

```
re = /((Sys(tem)?)?[Aa]dmin(istrator)?)/
     ^                                ^
```

Diese äußersten Klammern werden nicht verwendet, um den Ausdruck zu quantifizieren, sondern für ein zusätzliches, weiter unten genauer erläutertes Feature: Diejenigen Teile des Strings, die auf geklammerte Ausdrücke passen, werden der Reihe nach (das heißt von außen nach innen sowie von links nach rechts) automatisch in den Variablen `$1`, `$2` und so weiter gespeichert. Das Skript kann dadurch im Fall eines Treffers ausgeben, auf *welchen* Teil der Eingabe der reguläre Ausdruck passt. Bei der "`Systemadministratorin`" ist es beispielsweise der Teilstring "`Systemadministrator`", beim "`Netzwerkadministrator`" dagegen nur "`administrator`".

Der nächste – und allgemeinste – Quantifizierer ist das Sternchen (*). Es bedeutet, dass der entsprechende Teilausdruck *beliebig oft* im untersuchten String vorkom-

men darf, das heißt keinmal, einmal oder öfter. Das folgende Beispiel passt auf eine 1, gefolgt von *beliebig vielen* Nullen:

```
10*
```

Hier noch ein Beispiel, das ein Kleiner-Zeichen, beliebig viele zufällige Zeichen und ein anschließendes Größer-Zeichen findet – eine vermeintlich perfekte Beschreibung von HTML- oder XML-Tags:

```
/<.*>/
```

Dabei ergibt sich allerdings ein Problem, das sich erst offenbart, wenn man den gesamten Ausdruck in runde Klammern setzt, auf einen String mit mehreren solchen Tags anwendet und anschließend den Inhalt von $1 kontrolliert. Hier ein entsprechender irb-Test:

```
>> html = "<html><body><h1>Hi!</h1></body></html>"
=> "<html><body><h1>Hi!</h1></body></html>"
>> if html =~ /(<.*>)/; puts $1; end
<html><body><h1>Hi!</h1></body></html>
```

Die Ausgabe des *gesamten* Strings zeigt, dass hier etwas nicht in Ordnung sein kann. Denken Sie noch einmal genau darüber nach, was die einzelnen Bestandteile von

```
<.*>
```

bedeuten: ein Kleiner-Zeichen, beliebig viele zufällige Zeichen, ein Größer-Zeichen. Nun, auch die verschiedenen Größer-Zeichen innerhalb des Strings sind beliebige Zeichen, das heißt, .* trifft genauso gut auch auf sie zu. Erst das allerletzte mögliche Größer-Zeichen wird als Treffer für > geliefert. Das Kernproblem: Der Quantifizierer * ist »gierig« (englisch *greedy*), das heißt, er passt auf so viel Text wie möglich.

Wenn Sie die Gier aufheben möchten, können Sie *hinter* das Sternchen zusätzlich ein Fragezeichen setzen. Der HTML-Ausdruck lautet dann also <.*?>. Probieren Sie es selbst aus:

```
>> html = "<html><body><h1>Hi!</h1></body></html>"
=> "<html><body><h1>Hi!</h1></body></html>"
>> if html =~ /(<.*?>)/; puts $1; end
<html>
```

Es gibt noch eine andere Lösung für die HTML-Tags. Der folgende Ausdruck passt auf ein Kleiner-Zeichen, beliebig viele Zeichen, die *kein* Größer-Zeichen sind, und ein Größer-Zeichen:

```
<[^>]*>
```

Der letzte allgemeine Quantifizierer ist das Pluszeichen (+). Es bedeutet, dass der betrachtete Teilausdruck *mindestens einmal* im String vorkommen muss. Das folgende Beispiel findet in Strings gepackte Fließkommazahlen – mindestens eine Zif-

fer vor dem Dezimalpunkt und mindestens eine dahinter (denken Sie an das weiter oben erwähnte Escaping des literalen Punkts durch den Backslash):

```
[0-9]+\.[0-9]+
```

Das Pluszeichen ist ebenso greedy wie das Sternchen; auch hier können Sie diese Wirkung mit einem nachgestellten Fragezeichen aufheben. So passt `/<.+>/` wieder auf einen beliebig langen HTML-Block (und zwar noch genauer als `/<.*>/`, weil zwischen den Klammern mindestens ein Zeichen erwartet wird), während `/<.+?>/` einzelne Tags findet.

Einen etwas anderen Weg gehen die numerischen Quantifizierer, die als Zahlen in geschweiften Klammern notiert werden. Dabei gibt es drei Möglichkeiten, die schematisch so aussehen:

- `{n}` bedeutet, dass das vorstehende Element genau *n*-mal vorkommen muss.
- `{m, n}` steht für ein Element, das an der entsprechenden Stelle mindestens *m*- und höchstens *n*-mal auftreten darf.
- `{n,}` schließlich bedeutet, dass das Element *mindestens n*-mal im String stehen muss. So ist `{0,}` etwa eine andere Schreibweise für `*`, während `{1,}` einem `+` entspricht.

Die Varianten `{m, n}` und `{n,}` sind wieder einmal gierig; auch sie lassen sich durch ein nachfolgendes Fragezeichen bescheidener machen.

Das folgende Beispiel findet eine ISBN als Teil des beliebigen Strings `str`:

```
if str =~ /([0-9]\-[0-9]{5}\-[0-9]{3}\-[0-9X])/
  puts "ISBN #{$1} gefunden."
end
```

Da manche Websites – zum Beispiel *amazon.de* – die stets an denselben Stellen vorkommenden Striche weglassen, sollten Sie diese per Fragezeichen optional setzen:

```
/([0-9]\-?[0-9]{5}\-?[0-9]{3}\-?[0-9X])/
```

Bereichsmarkierungen

Alle bisher vorgestellten regulären Ausdrücke passten stets auf ein beliebiges Teilstück eines Strings. Das ist prima, um bestimmte Muster in einem Text zu suchen, aber etwa für die Sicherheitsüberprüfung einer Eingabe katastrophal. Um die Gültigkeit regulärer Ausdrücke auf bestimmte Teile von Strings einzuschränken, stehen diverse Bereichsmarkierungen zur Verfügung.

Die wichtigsten sind `^` für den Anfang und `$` für das Ende des untersuchten Strings beziehungsweise einer Zeile. Der reguläre Ausdruck

```
/0x[0-9a-fA-F]+/
```

passt auf alle Strings, die an beliebiger Stelle eine Hexadezimalzahl enthalten. Also zum Beispiel auf die Strings `"Der Wert ist 0xABC"` oder `"0xFF0000 ist sattes Rot"`.

Wenn Sie jedoch

```
/^0x[0-9a-fA-F]+$/
```

schreiben, sind plötzlich nur noch ganze Zeilen erlaubt, die ausschließlich aus einer Hexadezimalzahl bestehen.

Auf ähnlich strenge Weise könnten Sie etwa überprüfen, ob eine Zeile ein gültiger Ruby-Bezeichner ist (die Kriterien dafür wurden bereits weiter oben in diesem Kapitel erläutert):

```
/^[A-Za-z_][0-9A-Za-z_]+$/
```

In Zeile 46 unseres Taschenrechner-Beispiels wurde die »Rechenschleife« unterbrochen, wenn die Antwort auf die Frage nach einem neuen Durchgang mit `n` beginnt:

```
break if nochmal =~ /^n/i
```

Das `i` hinter dem letzten Slash ist übrigens ein Modifier, der Groß- und Kleinschreibung ignoriert. Diesen und andere Modifier lernen Sie weiter unten genauer kennen.

Noch strenger (und in dieser Form eine Besonderheit von Ruby) sind `\A` und `\Z`, die auf jeden Fall den absoluten Anfang beziehungsweise das absolute Ende des jeweiligen Strings kennzeichnen. Um das zu verdeutlichen, sollten Sie in `irb` einen String mit enthaltenem Zeilenumbruch untersuchen:

```
>> "Hallo\nWelt" =~ /^W/
=> 6
>> "Hallo\nWelt" =~ /\AW/
=> nil
>> "Hallo\nWelt" =~ /\AH/
=> 0
```

Der reguläre Ausdruck `/^W/` trifft zu, denn es gibt ein W an einem Zeilenanfang. `/\AW/` passt dagegen nicht, weil das W nicht am Anfang des Strings selbst steht. `/\AH/` wird dagegen gefunden, weil der String mit H anfängt.

Eine weitere Bereichsmarkierung ist die durch `\b` gekennzeichnete *Wortgrenze*. Dabei kann es sich um Whitespace, um String-Anfang oder -Ende sowie um Satzzeichen handeln. So passt der Teilausdruck `\bals` beispielsweise auf `"Regexp als Suchmuster"`, aber nicht auf `"Ich habe Halsschmerzen"`. Ebenso ist `Tag\b` für `"Guten Tag."` geeignet, aber nicht für `"Tageszeitung"`.

Das Gegenteil von einer Wortgrenze können Sie mit `\B` ausdrücken: Dabei muss sich der entsprechende Teilausdruck innerhalb eines Wortes befinden. So passt `\Beis` beispielsweise auf `"Erdbeereis"`, aber nicht auf `"Das Wetter ist eisig"`.

Modifizierer

Mit Hilfe bestimmter Zeichen, die hinter den letzten Slash eines regulären Ausdrucks gesetzt werden, lässt sich dessen Wirkung modifizieren. In Ruby sind drei solcher *Modifizierer* (Modifier) wichtig.

- `i` steht für »ignorecase« und bedeutet, dass die Groß- und Kleinschreibung nicht mehr beachtet werden soll. So liefert `"HALLO" =~ /a/` beispielsweise keinen Treffer, während `"HALLO" =~ /a/i` den Wert 1 (Position des Buchstaben »A«) zurückgibt.
- `m` bedeutet »multiline«. Normalerweise werden reguläre Ausdrücke nur auf einzelne Zeilen von Strings angewendet. Zum Beispiel passt `/o.W/` (o, beliebiges Zeichen, W) nicht auf `"Hallo\nWelt"`, weil das »beliebige Zeichen« (hier `\n`) auf einer Zeilengrenze liegt. Wenn Sie stattdessen den modifizierten Ausdruck `/o.W/m` verwenden, erhalten Sie einen Treffer, da der komplette String betrachtet wird.
- `x` ist der »Extended«-Modus. Er ermöglicht Ihnen, beliebigen Whitespace in Ihre regulären Ausdrücke einzufügen, um sie übersichtlicher zu gestalten. Hier zum Beispiel ein strenger, mehrzeilig geschriebener ISBN-Regexp:

```
/\A
 \d\-?
 \d{5}\-?
 \d{3}\-?
 [0-9X]
\Z/x
```

Sie können die Modifizierer beliebig miteinander kombinieren. `/[a-z]+/im` steht zum Beispiel für einen oder mehrere Groß- oder Kleinbuchstaben (`i`) im Multiline-Modus (`m`).

Wenn Sie einen regulären Ausdruck als `%r(...)` statt als `/.../` schreiben, werden die Modifizierer ebenfalls ohne Abstand dahintergesetzt. Das nachfolgende Beispiel findet beliebig lange Hexadezimalzahlen:

```
%r([0-9a-f]+)i
```

Weitere Konstrukte

Man könnte endlos weitermachen und ein Kapitel oder gar ein ganzes Buch über die Ruby-Regexp-Syntax schreiben, aber im Rahmen dieser Einführung muss irgendwann Schluss sein. Deshalb finden Sie hier nur noch eine kurze Aufzählung weiterer Elemente, die teilweise in späteren Kapiteln zum Einsatz kommen. Am besten probieren Sie sie am sprichwörtlichen verregneten Sonntagnachmittag alle einmal aus.

Das Pipe-Zeichen (`|`) repräsentiert alternative Bereiche, meist innerhalb runder Klammern, um sie vom Rest abzugrenzen. `(Netzwerk|System)administrator` steht beispielsweise für `"Netzwerkadministrator"` oder `"Systemadministrator"`.

Zu den bereits genannten Elementen, die mit `\` und einem Buchstaben gebildet werden, kommen noch folgende hinzu:

- `\d` steht für eine beliebige Ziffer, ist also eine Kurzfassung für `[0-9]`. Das Gegenstück `\D` stellt ein Zeichen dar, das keine Ziffer ist, genau wie `[^0-9]`. Damit lässt sich etwa die weiter oben vorgestellte ISBN verkürzt als `\d\-?\d{5}\-?\d{3}\-?[\dX]` schreiben.
- `\w` trifft auf jedes Zeichen zu, das in einem gültigen Ruby-Bezeichner stehen darf: Buchstabe, Ziffer oder Unterstrich, also `[0-9A-Za-z_]`. `\W` passt entsprechend auf ein Zeichen, das nicht zu dieser Gruppe gehört.
- `\s` passt auf ein beliebiges Whitespace-Zeichen, also auf ein Leerzeichen, einem Tabulator oder einen Zeilenumbruch. Jedes Nicht-Whitespace-Zeichen wird dagegen durch `\S` beschrieben.

Daneben gibt es einige Konstrukte in der Form `[[:Klasse:]]`. Es handelt sich dabei um die sogenannten *POSIX-Zeichenklassen*, die zum Regex-Vokabular klassischer UNIX-Kommandos wie `grep` gehören. Unter anderem sind folgende definiert:

- `[[:alnum:]]` – alphanumerisches Zeichen (Ziffer oder Buchstabe), entspricht `[\dA-Za-z]`
- `[[:alpha:]]` – Buchstabe, entspricht `[A-Za-z]`
- `[[:digit:]]` – Ziffer, entspricht `\d`
- `[[:space:]]` – Whitespace, entspricht `\s`

Reguläre Ausdrücke im Einsatz

Nachdem Ihnen nun einigermaßen ausführlich die Regex-Syntax vorgestellt wurde, sollten Sie auch wissen, was man mit regulären Ausdrücken alles anstellen kann. In diesem Abschnitt erhalten Sie einen kurzen Überblick; in späteren Kapiteln werden einige Anwendungsmöglichkeiten vertieft.

Das einfache *Matching* – das Finden von Suchmustern – ist die häufigste Anwendung für reguläre Ausdrücke. Die beiden zuständigen Operatoren `=~` und `!~` haben Sie bereits kennengelernt. Hier zur Wiederholung noch einmal ihre Funktionsweise in Kurzform:

```
String =~ Regexp
```

liefert bei Erfolg die Anfangsposition des ersten Treffers im String und bei Misserfolg `nil`.

```
String !~ Regexp
```

ergibt dagegen `true`, wenn es keinen Treffer gibt, und andernfalls `false`.

Ein weiteres, besonders nützliches Einsatzgebiet regulärer Ausdrücke ist das Ersetzen der Treffer durch neuen Text. Dafür sind die String-Methoden `sub` und `gsub` zuständig:

```
String.sub(Regex, NeuStr)
```

ersetzt den ersten Treffer im untersuchten String durch den String *NeuStr*.

```
String.gsub(Regex, NeuStr)
```

ersetzt dagegen *alle* Treffer. Ein Beispiel für Letzteres haben Sie bereits in Zeile 26 des Textmanipulierer-Beispiels gesehen:

```
@txt.gsub(/[aeiou]/i, "*")
```

In der String-Instanzvariablen `@txt` wird jeder – wegen dem Modifizierer `i` sowohl groß- als auch kleingeschriebene – Vokal durch ein Sternchen ersetzt.

Hier ein weiteres Beispiel, das die Fließkommazahl π in einen String umwandelt und den Punkt durch das in Kontinentaleuropa gebräuchliche Komma ersetzt:

```
zahl = Math::PI
ausgabe = zahl.to_s.sub(/\./, ",")
puts "Pi hat den Wert #{ausgabe}"
```

Das Beispiel gibt folgende Zeile aus:

```
Pi hat den Wert 3,14159265358979
```

Besonders praktisch ist, dass Sie innerhalb des Ersetzungs-Strings auf Treffer-Teilstrings zurückgreifen können, wenn Sie diese in runde Klammern setzen. In der bisher gezeigten Schreibweise funktioniert das allerdings noch nicht mit Hilfe der bereits erwähnten Variablen `$1`, `$2` und so weiter. Diese stehen nämlich erst *nach* der Ausführung einer Regexp-Methode zur Verfügung. Stattdessen gibt es hier die spezielle Variante `\1`, `\2` usw. Einziges Problem: In einem String in doppelten Anführungszeichen werden sie als Zeichencodes missverstanden, und sie lassen sich auch nicht als eingebettete Ausdrücke schreiben. Abhilfe schaffen hier entweder einfache Anführungszeichen, deren Nachteile zu Beginn dieses Kapitels bereits aufgezählt wurden, oder das Escaping des jeweiligen Backslashs: schreiben Sie mit anderen Worten entweder `'\1'` oder `"\\1"`.

Hier ein Beispiel, das einen String der Form `"Wort1 Wort2"` durch `"Wort2, Wort1"` ersetzt – also beispielsweise Namen für eine alphabetische Liste umkehrt:

```
name = "Peter Schmitz"
puts name.sub(/(\w+)\s+(\w+)/, '\2, \1')
```

Ergebnis:

```
Schmitz, Peter
```

Hier ein weiteres Beispiel, das eine ISBN mit Strichen versieht, falls sie noch keine hat, und sie ansonsten unangetastet lässt:

```
isbn1 = "3897214784"
isbn2 = "3-89721-403-2"
puts isbn1.sub(/(\d)\-?(\d{5})\-?(\d{3})\-?([\dX])/, '\1-\2-\3-\4')
puts isbn2.sub(/(\d)\-?(\d{5})\-?(\d{3})\-?([\dX])/, '\1-\2-\3-\4')
```

Die Ausgabe lautet:

```
3-89721-478-4
3-89721-403-2
```

Erfreulicherweise gibt es eine zweite Möglichkeit: Sie können den zweiten Parameter, das heißt den Ersetzungs-String, weglassen und stattdessen einen Block benutzen. Die `$`-Treffervariablen stehen darin zur Verfügung. Die Ersetzungsmethode `gsub` funktioniert sogar als vollwertiger Iterator; innerhalb des Blocks kann der jeweils aktuelle Treffer als `|Variable|` abgefangen werden (Näheres über Blöcke und Iteratoren erfahren Sie im nächsten Abschnitt). Nebenbei können Sie beliebige Fallentscheidungen und andere Anweisungen verwenden, um den Ersetzungswert zu erzeugen.

Das ISBN-Beispiel lässt sich auf diese Weise übersichtlicher schreiben:

```
isbn = "3897214784"
puts isbn.sub(/(\d)\-?(\d{5})\-?(\d{3})\-?([\dX])/) {
  "#{$1}-#{$2}-#{$3}-#{$4}"
}
```

Hier noch ein sehr unterhaltsames `gsub`-Beispiel. Es liest eine Eingabezeile ein, ersetzt jeden Vokal durch einen anderen und gibt den veränderten String aus:

```
eingabe = gets.chomp
puts eingabe.gsub(/[aeiou]/i) { |vok|
  case vok
  when "a"
    "e"
  when "e"
    "i"
  when "i"
    "o"
  when "o"
    "u"
  when "u"
    "a"
  when "A"
    "U"
  when "E"
    "A"
  when "I"
    "E"
  when "O"
    "I"
  when "U"
    "O"
  end
}
```

Wenn Sie das Beispiel speichern und ausführen, sieht es beispielsweise so aus:

```
Es ist eine Freude, den Vokalverschieber bei der Arbeit zu sehen
As ost ioni Friadi, din Vukelvirschoibir bio dir Urbiot za sihin
```

Der jeweilige Treffer wird dabei in der Variablen `vok` gespeichert. Diese wird per case-Fallentscheidung untersucht, und je nach Inhalt wird der Ersetzungswert festgelegt.

Natürlich sind auch sinnvollere Anwendungen möglich. Das folgende Beispiel kommt sogar ohne Auswertung der Treffer selbst aus – es ersetzt einfache Striche (-) am Anfang von Zeilen durch fortlaufende Zahlen, macht also aus einer einfachen Aufzählung eine nummerierte Liste:

```
text = <<ENDE
Ruby ist
- flexibel
- objektorientiert
- leicht zu erlernen
ENDE
i = 0
puts text.gsub(/^\-\s+/) {
  i += 1
  "#{i}. "
}
```

Hier die entsprechende Ausgabe:

```
Ruby ist
1. flexibel
2. objektorientiert
3. leicht zu erlernen
```

Innerhalb des Blocks im obigen Beispiel ist die Reihenfolge wichtig, weil der *letzte* Ausdruck zurückgeliefert wird. Falls Sie `i` also erst nach dem String-Ausdruck `"#{i}.  "` erhöhen (und dafür vor dem Block auf 0 setzen), erhalten Sie Zeilen wie `"1flexibel"`.

Wie bei vielen bereits vorgestellten Methoden gibt es übrigens auch für die Ersetzungsmethoden die Varianten `sub!` und `gsub!`, die den Inhalt der behandelten String-Variablen selbst modifizieren. Das Ausprobieren überlasse ich Ihnen als zusätzliche Übung ;-).

Zuletzt sollte an dieser Stelle noch erwähnt werden, dass die bereits vorgestellte String-Methode `split` als Trennzeichen nicht nur eine einfache Zeichenfolge, sondern auch einen regulären Ausdruck akzeptiert. Das folgende Beispiel wandelt einen beliebigen Text in ein Array aus Wörtern um, indem es beliebig lange Folgen von Nicht-Wort-Zeichen als Trennzeichen akzeptiert:

```
>> text = <<ENDE
Saetze, die (verschiedene) Satzzeichen enthalten,
sogar Zeilenumbrueche -- sie sollen von diesem Ballast
```

```
befreit werden: als Array!
ENDE
>> text.split(/\W+/)
=> ["Saetze", "die", "verschiedene", "Satzzeichen", "enthalten",
"sogar", "Zeilenumbrueche", "sie", "sollen", "von", "diesem",
"Ballast", "befreit", "werden", "als", "Array"]
```

Neben den hier gezeigten Möglichkeiten der Regex-Verwendung gibt es auch noch einen explizit objektorientierten Ansatz für den Umgang mit regulären Ausdrücken. Das wird passenderweise im nächsten Kapitel besprochen.

Iteratoren und Blöcke

Neben den weiter oben behandelten Schleifen hat Ruby noch ein weiteres interessantes Konzept zu bieten: *Iteratoren*. Es handelt sich dabei um eine Konstruktion, die bestimmte Anweisungen für alle Elemente einer Menge durchführt. Mit ähnlich praktischer Syntax gibt es Iteratoren sonst nur noch in der Programmiersprache Smalltalk, einem leider nicht allzu verbreiteten Klassiker der Objektorientierung.

Alle Iteratoren besitzen die folgende grundsätzliche Syntax:[15]

```
Menge.Methode {
  Anweisung(en)
  ...
}
```

Die alternative Schreibweise lautet:

```
Menge.Methode do
  Anweisung(en)
  ...
end
```

Die *Menge* ist irgendeine Aufzählung mit mehreren Elementen, normalerweise ein Bereich wie `1..10` oder ein Array. Der Bereich zwischen den geschweiften Klammern beziehungsweise zwischen `do` und `end` wird als *Block* bezeichnet. Die *Methode* ist der eigentliche Iterator und kann sowohl eine Ruby-Standardmethode als auch eine selbst geschriebene Methode (siehe nächstes Kapitel) sein; Voraussetzung ist nur, dass sie keinen einzelnen Wert, sondern einen Block als Parameter akzeptiert.

Die einfachste derartige Methode ist `each`. Wie der Name schon sagt, führt sie die Anweisungen im Block einfach für jedes Element der Menge aus. Das folgende (zugegeben recht sinnfreie) Beispiel gibt zehnmal `"Hallo"` aus:

```
(1..10).each {
  puts "Hallo"
}
```

---

15 Die geschweiften Klammern – die Sie in Ruby seltener brauchen als in vielen anderen Sprachen – werden auf einem PC mit **Alt Gr + 7** bzw. **Alt Gr + 0** erzeugt.

Das ist aber erst die halbe Wahrheit über Iteratoren und Blöcke. Der besondere Clou besteht darin, dass die Werte der Elemente in den Block weitergegeben werden. Um Gebrauch davon zu machen, müssen Sie am Anfang des Blocks eine Variable (oder gegebenenfalls auch mehrere) benennen, die diese Werte aufnehmen soll. Diese wird zwischen zwei |-Zeichen gesetzt. Hier ein Beispiel, das den Bereich von 1 bis 10 unter dem Variablennamen `i` durchzählt und die entsprechenden Zahlen ausgibt:

```
(1..10).each { |i|
  puts i
}
```

Die andere Schreibweise sieht so aus:

```
(1..10).each do |i|
  puts i
end
```

Natürlich können Sie mit den Elementen auch viel kompliziertere Aufgaben erledigen. Das folgende Beispiel gibt zusätzlich zu den Zahlen der Menge – durch Tabulatoren getrennt – auch ihr Quadrat sowie ihre Quadratwurzel aus:

```
puts "Zahl\tQuadrat\tQuadratwurzel"
(1..10).each do |i|
  quadrat = i ** 2
  wurzel = Math.sqrt(i)
  puts "#{i}\t#{quadrat}\t#{wurzel}"
end
```

Damit Sie einen direkten Vergleich haben, sehen Sie hier eine letzte Neufassung des Primzahlen-Beispiels aus dem Abschnitt über Schleifen, diesmal mit der Iterator-Methode each:

```
(1..100).each { |i|
  # Annahme: i ist eine Primzahl
  prim = true
  # Potenzielle Teiler bis i/2 testen
  (2..i/2).each { |j|
    # i durch j teilbar?
    if i % j == 0
      # Keine Primzahl!
      prim = false
      # Pruefschleife (j) vorzeitig verlassen
      break
    end
  }
  # Ausgabe, falls Primzahl
  print "#{i}  " if prim
}
```

Sie können auch die Elemente eines Arrays mit `each` verarbeiten und innerhalb des Blocks als `|Variable|` ansprechen. Das folgende Beispiel gibt alle Elemente des Arrays untereinander aus:

```
programmiersprachen = %w(Ruby Perl Smalltalk Java)
programmiersprachen.each { |sprache|
  puts sprache
}
```

Wenn Sie den Iterator each auf einen Hash anwenden, werden die Schlüssel und ihre Werte gleichberechtigt nacheinander ausgewertet wie die Elemente eines Arrays:

```
geschlecht = {"f" => "weiblich", "m" => "maennlich"}
geschlecht.each { |g|
  puts g
}
```

ergibt

```
m
maennlich
f
weiblich
```

Da dies meist nicht das gewünschte Ergebnis ist, gibt es speziell für Hashes den Iterator each_key, der nur die Schlüssel nacheinander durchgeht. Damit lässt sich der Hash-Inhalt sinngemäß verarbeiten:

```
geschlecht.each_key { |k|
  puts "#{k}: #{geschlecht[k]}"
}
```

liefert die korrektere Ausgabe:

```
m: maennlich
f: weiblich
```

Falls Sie jedoch auf die Schlüssel verzichten können und nur die Werte des Hashs benötigen, können Sie alternativ den Iterator each_value verwenden:

```
geschlecht.each_value { |v|
  puts v
}
```

Die Ausgabe lautet hier einfach:

```
maennlich
weiblich
```

Schließlich gibt es auch noch den Hash-Iterator each_pair, der das jeweilige Schlüssel-Wert-Paar bereitstellt. Sie können innerhalb des Blocks zwei Variablen verwenden, um diese Daten aufzunehmen:

```
sprachen = {
  "de" => "Deutsch",
  "en" => "Englisch",
  "fr" => "Franzoesisch"
}
```

```
sprachen.each_pair { |kurz, lang|
  puts "#{kurz} steht fuer #{lang}"
}
```

Hier die Ausgabe dieses Beispiels:

```
fr steht fuer Franzoesisch
de steht fuer Deutsch
en steht fuer Englisch
```

Auch andere Datentypen besitzen spezielle Iteratoren. Ein sehr nützlicher steht für Ganzzahlen zur Verfügung und heißt `times`. Er führt die im Block verschachtelte Aufgabe so oft aus, wie es die jeweilige Zahl angibt. Das folgende Beispiel liefert 5 Würfe eines Würfels (ideal etwa für das beliebte Spiel »Kniffel«):

```
5.times {
  wurf = rand(6) + 1
  print "#{wurf}  "
}
```

Beispielausgabe:

```
4  3  5  1  5
```

Weitere wichtige Iteratoren für Integerwerte sind:

- `n1.upto(n2)` zählt aufwärts von `n1` bis einschließlich `n2`:

  ```
  >> 3.upto(7) { |i| print "#{i}  "}
  3  4  5  6  7
  ```

- `n1.downto(n2)` zählt abwärts von `n1` bis `n2`, wobei `n1` natürlich größer sein muss als `n2`:

  ```
  >> 10.downto(5) { |i| print "#{i}  "}
  10  9  8  7  6  5
  ```

- `n1.step(n2, n3)` zählt in Schritten der Größe `n3` von `n1` bis `n2`:

  ```
  >> 8.step(64, 8) { |i| print "#{i}  "}
  8  16  24  32  40  48  56  64
  >> 49.step(7, -7) { |i| print "#{i}  "}
  49  42  35  28  21  14  7
  ```

Strings dagegen sind beispielsweise mit dem Iterator `each_byte` ausgestattet, der nacheinander die Codes der einzelnen Zeichen liefert. Damit lässt sich das oben vorgestellte Exklusiv-Oder-Verschlüsselungsprogramm etwas kompakter schreiben:

```
print "Text: "
text = gets.chomp
print "Schluessel (1-255): "
key = gets.to_i

if key < 1 or key > 255
  puts "Ungueltiger Schluessel!"
  exit
end
```

```
encrypt = ""

# Iterator ueber alle Zeichen des Textes
text.each_byte { |ch|
  # Durch Exklusiv-Oder maskierten Zeichencode wieder in Zeichen umwandeln
  # und an verschluesselten Text anfuegen
  encrypt += (ch^key).chr
}
print "Verschluesselter Text: #{encrypt}"
```

Zum einen können Sie so die umständliche Bereichskonstruktion `0...text.length` vermeiden, und zum anderen liegt das jeweilige Zeichen innerhalb des Blocks bereits als Codenummer vor, was die eigentliche Verschlüsselungszeile verkürzt.

Ein weiterer interessanter Iterator für Bereiche, Arrays und andere Mengen ist `collect`: Das Ergebnis jedes Blockdurchlaufs wird wieder als neues Element zu einem Array hinzugefügt. Auf diese Weise lässt sich beispielsweise ganz schnell ein Pool mit allen 49 Lottozahlen bilden, aus denen später »gezogen« werden kann:

```
lottozahlen = (1..49).collect { |z|
  z
}
```

Wenn Sie das in `irb` eingeben, erhalten Sie sofort das gesamte Array als Ausgabe:

```
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

Aber anstatt den Wert des jeweiligen Elements aus der ursprünglichen Menge einfach unverändert zu übernehmen, können Sie natürlich auch beliebige Ausdrücke daraus bilden. Das folgende Beispiel speichert alle Quadrate der Zahlen 1 bis 20 in einem Array (hier einmal in einer einzelnen `irb`-Zeile und gleich mit Ausgabe):

```
>> quadrate = (1..20).collect { |i| i**2 }
=> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225,
256, 289, 324, 361, 400]
```

Aus dem Lottozahlen-Beispiel lässt sich ganz leicht ein Ziehungssimulator bauen. Denn es genügt natürlich nicht, einfach sieben Zufallszahlen zwischen 1 und 49 zu wählen; dabei kann schließlich leicht mehrmals dieselbe Zahl vorkommen. Die »Kugeln« werden daher – wie oben gezeigt – mittels `collect` in einem Array gespeichert. Anschließend werden per `slice!` sieben zufällig gewählte Glückszahlen ausgeschnitten und angezeigt. Hier der komplette Code, den Sie nach allen Erklärungen in diesem Kapitel ohne Weiteres verstehen müssten:

```
# Zahlen von 1 bis 49 in Array speichern
lottozahlen = (1..49).collect { |z|
  z
}
```

```
# Schleife ueber 7 Durchgaenge
for i in 1..7
  # Zufaellig gewaehlte Zahl ausschneiden
  zahl = lottozahlen.slice!(rand(lottozahlen.length))

  # Ausgeben
  if i == 7
    puts "Zusatzzahl: #{zahl}"
  else
    puts "#{i}. #{zahl}"
  end
end
```

Eine Beispielausführung sieht so aus:

```
> ruby lotto.rb
1. 39
2. 24
3. 26
4. 47
5. 27
6. 6
Zusatzzahl: 15
```

Noch hübscher ist es natürlich, wenn die ersten sechs Zahlen sortiert ausgegeben werden. Dazu können Sie sie wieder mit Hilfe von `collect` in einem Array sammeln und anschließend mit dessen Methode `sort!` dauerhaft sortieren und ausgeben. Die Zusatzzahl kann danach einfach separat gezogen werden. Hier der gesamte abgewandelte Code:

```
# Zahlen von 1 bis 49 in Array speichern
lottozahlen = (1..49).collect { |z|
  z
}

# Die sechs regulaeren Zahlen sammeln
ziehung = (1..6).collect {
  # Zufaellig gewaehlte Zahl ausschneiden
  lottozahlen.slice!(rand(lottozahlen.length))
}

# Zusatzzahl ziehen
zusatz = lottozahlen.slice!(rand(lottozahlen.length))

# Gezogene Zahlen sortieren
ziehung.sort!

# Ausgabe
print ziehung.join(", ")
puts "   Zusatzzahl: #{zusatz}"
```

Beachten Sie, dass der `collect`-Block für die Ziehung ohne Parametervariable auskommt, da die Zählungswerte selbst nicht benötigt werden. Zudem wird die Zie-

hung selbst einfach als Ausdruck hingeschrieben, da `collect` sie automatisch im Ergebnis-Array sammelt.

Die Ausgabe könnte zum Beispiel so aussehen:

```
> ruby lotto2.rb
18, 20, 25, 36, 40, 49   Zusatzzahl: 2
```

Im Verlauf dieses Buchs werden Sie noch weitere Beispiele für Iteratormethoden kennenlernen, zum Beispiel für das zeilen- oder zeichenweise Auslesen von Dateien. Zudem erfahren Sie im nächsten Kapitel, wie Sie selbst welche schreiben können.

Zusammenfassung

Herzlichen Glückwunsch! Sie haben bis hierhin durchgehalten und sind damit für alle künftigen Ruby-Abenteuer bestens gerüstet. Sie haben mit anderen Worten alle wichtigen Werkzeuge und Materialien kennengelernt. Im nächsten Kapitel erfahren Sie dann, wie Sie daraus größere Anwendungen konstruieren können.

Was Sie bereits alles gelernt haben, können Sie feststellen, indem Sie sich nochmals den zu Beginn dieses Kapitels vorgestellten Taschenrechner ansehen. In Beispiel 2-4 sehen Sie ihn der Übersicht halber noch einmal.

Beispiel 2-4: Der Ruby-Taschenrechner, rechner.rb – nun kennen Sie alle Anweisungen

```
 1  # Ueberschrift ausgeben
 2  puts "Ruby-Rechner"
 3  puts "============"
 4  puts

 5  # Endlosschleife starten
 6  loop do
 7    print "Bitte die erste Zahl:      "
 8    # Eingabe direkt in Fliesskommazahl umwandeln
 9    # und in z1 speichern
10    z1 = gets.to_f
11    print "Bitte die zweite Zahl:     "
12    # Eingabe direkt in Fliesskommazahl umwandeln
13    # und in z2 speichern
14    z2 = gets.to_f

15    print "Rechenoperation (+|-|*|/)? "
16    # Operator einlesen und anschliessenden Zeilenumbruch entfernen
17    op = gets.chomp

18    # Gueltigkeit des Operators pruefen
19    if op !~ /^[\+\-\*\/]$/
20      puts "Ungueltige Operation: #{op}"
21      puts
22      next
23    end
```

Beispiel 2-4: Der Ruby-Taschenrechner, rechner.rb – nun kennen Sie alle Anweisungen

```
24      # Bei Division 0 als zweiten Operanden ausschliessen
25      if op == "/" && z2 == 0
26        puts "Division durch 0 ist verboten"
27        puts
28        next
29      end

30      # Ergebnis je nach Operator berechnen
31      case op
32        when "+"
33          ergebnis = z1 + z2
34        when "-"
35          ergebnis = z1 - z2
36        when "*"
37          ergebnis = z1 * z2
38        when "/"
39          ergebnis = z1 / z2
40      end

41      # Ausgabe des Ergebnisses
42      puts "Ergebnis: #{z1} #{op} #{z2} = #{ergebnis}"
43      print "Noch eine Berechnung (j/n)? "
44      nochmal = gets.chomp
45      puts
46      break if nochmal =~ /^n/i
47    end
```

Damit Sie noch einmal rekapitulieren können, was Sie in diesem Kapitel (unter anderem) gelernt haben, folgen hier noch einmal die Erläuterungen zum Skript – allerdings mit allen Fachbegriffen, die zu Beginn des Kapitels vermieden wurden und die Sie nun kennengelernt haben.

Nach Ausgabe der Überschrift (Zeile 2 bis 4) läuft der eigentliche Rechner in einer Endlosschleife, wird also immer wieder ausgeführt. Dazu dient der folgende Block (Zeile 6 bis 47):

```
loop do
  ...
end
```

Innerhalb der Schleife werden zunächst die beiden Zahlen vom Benutzer erfragt (Zeile 7 bis 14). Das Ergebnis der Eingabemethode `gets` wird mit Hilfe der Methode `to_f` sofort in eine Fließkommazahl umgewandelt und dann in der Variable `z1` beziehungsweise `z2` abgelegt. Beim eingegebenen Operator wird dagegen mittels `chomp` der Zeilenumbruch entfernt.

In Zeile 19 bis 29 erfolgen zwei Tests: Wenn das »Anti-Matching« per `!~` ergibt, dass der eingegebene Operator keines der vier zulässigen Zeichen `+`, `-`, `*` oder `/` ist (Zeile 19), wird eine Fehlermeldung ausgegeben; anschließend wird mittels `next`

(Zeile 22) der nächste Schleifendurchgang gestartet. Als Nächstes wird für den Fall, dass die Operation eine Division ist, die unzulässige 0 als zweiter Operand ausgeschlossen.

Das Ergebnis wird in Zeile 31 bis 40 mit Hilfe einer `case`/`when`-Struktur berechnet. Die `when`-Fälle prüfen nacheinander verschiedene Einzelwerte und führen die Operation für den passenden Fall durch. In Zeile 42 wird das auf diese Weise berechnete Ergebnis ausgegeben.

Am Ende des übernächsten Kapitels erhalten Sie eine ähnliche Rekapitulation für das Textmanipulierer-Beispiel.

Die Zeilen 42 bis 46 kümmern sich um die Frage, ob der Benutzer eine weitere Berechnung wünscht. Wenn seine Antwort mit `n` beginnt, das heißt auf den regulären Ausdruck `/^n/` passt, wird die Schleife mit `break` verlassen (Zeile 46). Da nach dem `end` in Zeile 47 keine weitere Anweisung folgt, ist damit auch das gesamte Programm beendet.

In diesem Kapitel:

- Was ist Objektorientierung?
- Ein- und Ausgabe
- Datum und Uhrzeit
- Einige weitere Klassen
- Die Ruby-Hilfe ri

KAPITEL 3
Ruby-Klassen

Nirgends kann man den Grad der Kultur einer Stadt
und überhaupt den Geist ihres herrschenden
Geschmacks schneller und doch zugleich richtiger
erkennen als – in den Lesebibliotheken.

– Heinrich von Kleist

Im vorigen Kapitel haben Sie die meisten wichtigen Einzelbausteine für Ihre Arbeit mit Ruby kennengelernt. In diesem Kapitel kommen die Hilfsmittel hinzu, mit denen Sie längere, strukturierte Programme daraus konstruieren können: Sie erfahren, wie Objektorientierung in der Praxis funktioniert. In diesem Kapitel werden Sie nach einem allgemeinen Einstieg einige der wichtigsten vorgefertigten Klassen und Objekte von Ruby verwenden: Klassen zur Ein- und Ausgabe, für Datum und Uhrzeit sowie für einige andere Einsatzgebiete. Im nächsten Kapitel erfahren Sie dann, wie leicht es ist, Ihre eigenen Programmentwürfe in Klassen und Objekten abzubilden.

Was ist Objektorientierung?

Einen ausführlichen Einstieg in den Entwurf von Ruby-Klassen erhalten Sie im nächsten Kapitel; in diesem geht es zunächst um die Nutzung vorgefertigter Ruby-Elemente, die als Klassen vorliegen. Die wichtigsten Begriffe aus dem Bereich der Objektorientierung sollten Sie allerdings auch dafür bereits kennen.

Deshalb sehen Sie in Beispiel 3-1 ein Listing, in dem Klassen und die Arbeit mit Instanzen dieser Klassen demonstriert werden. Anhand dieses Beispiels werden anschließend die wichtigsten Begriffe der objektorientierten Programmierung erläutert.

Beispiel 3-1: Ein kurzes Beispiel zur Erläuterung der OO-Grundbegriffe, rechteck.rb

```
 1  # Klassendefinition: Rechteck
 2  class Rechteck
 3    # Konstruktor
 4    def initialize(b, h)
 5      @breite = b
 6      @hoehe = h
 7    end

 8    # Methoden
 9    def get_breite
10      @breite
11    end

12    def get_hoehe
13      @hoehe
14    end

15    def get_flaeche
16      @breite * @hoehe
17    end

18    def get_diagonale
19      Math.sqrt(@breite**2 + @hoehe**2)
20    end
21  end

22  # Klassendefinition: Quadrat,
23  # abgeleitet von Rechteck
24  class Quadrat < Rechteck
25    # Geaenderter Konstruktor
26    def initialize(b)
27      super(b, b)
28    end
29  end

30  # Anwendungsbeispiele

31  r = Rechteck.new(20, 10)
32  printf "Rechteck, Breite:         %d\n", r.get_breite
33  printf "          Hoehe:          %d\n", r.get_hoehe
34  printf "          Flaecheninhalt: %d\n", r.get_flaeche
35  printf "          Diagonale:      %.3f\n", r.get_diagonale

36  puts

37  q = Quadrat.new(20)
38  printf "Quadrat,  Kantenlaenge:   %d\n", q.get_breite
39  printf "          Flaecheninhalt: %d\n", q.get_flaeche
40  printf "          Diagonale:      %.3f\n", q.get_diagonale
```

Wenn Sie dieses Skript ausführen, erhalten Sie folgende Ausgabe:

```
> ruby rechteck.rb
Rechteck, Breite:         20
          Hoehe:          10
          Flaecheninhalt: 200
          Diagonale:      22.361

Quadrat,  Kantenlaenge:   20
          Flaecheninhalt: 400
          Diagonale:      28.284
```

In diesem Beispiel sind `Rechteck` und `Quadrat` zwei *Klassen*. Sie legen die Datenstruktur und das Verhalten von Objekten fest, die diese Formen beschreiben. Diese Verknüpfung von Daten und ihren Verarbeitungsmethoden ist eines der wichtigsten Ziele der Objektorientierung, das als *Kapselung* bezeichnet wird.

Die Klasse `Rechteck` (Zeile 2-21) besitzt fünf *Methoden*, das heißt Funktionen zur Verarbeitung von Rechteck-Objekten. Die erste Methode, `initialize`, ist der so genannte *Konstruktor* der Klasse. Er wird aufgerufen, sobald ein neues Objekt der Klasse erzeugt wird, und dient in der Regel dazu, der Datenstruktur des Objekts ihre Anfangswerte zuzuweisen. Beim Rechteck sind die relevanten Daten Breite und Höhe, die bei der Objekterzeugung übergeben werden. Sie werden in den *Instanzvariablen* (auch *Attribute* oder *Eigenschaften* genannt) `@breite` und `@hoehe` gespeichert.

Der Typ der restlichen vier Methoden wird *Getter* genannt. Die Namen solcher Methoden beginnen traditionell mit dem Wort `get`, und sie geben den Wert einer Instanzvariablen oder einen daraus berechneten Ausdruck zurück. Konkret werden folgende Werte zurückgeliefert beziehungsweise berechnet:

- `get_breite` (Zeile 9-11) liefert die Breite `@breite` zurück
- `get_hoehe` (Zeile 12-14) gibt die Höhe `@hoehe` zurück
- `get_flaeche` (Zeile 15-17) berechnet den Flächeninhalt – das Produkt aus den beiden Instanzvariablen
- `get_diagonale` (Zeile 18-20) zieht die Wurzel aus dem Satz des Pythagoras, um die Länge der Diagonale zu ermitteln

Anhand der Klasse `Quadrat` wird das Konzept der *Vererbung* demonstriert. Die Einleitungszeile (24)

```
class Quadrat < Rechteck
```

besagt, dass `Quadrat` sämtliche Eigenschaften und Methoden von `Rechteck` übernimmt. Durch die Vererbung brauchen Sie nur diejenigen Aspekte zu ändern (Fachbegriff: *überschreiben*), die die *abgeleitete Klasse* (hier `Quadrat`) von der *Elternklasse* (`Rechteck`) unterscheiden oder die hinzukommen.

Der einzige Unterschied zwischen Rechtecken und Quadraten besteht darin, dass Breite und Höhe bei letzteren identisch sind. Somit braucht lediglich der Konstruktor überschrieben zu werden: Derjenige der Klasse `Quadrat` erhält nur einen Wert. Er wird einfach zweimal an einen Aufruf des Konstruktors der Elternklasse – Schlüsselwort `super` – übergeben.

In Zeile 31 wird `Rechteck.new` aufgerufen, um ein Objekt oder eine *Instanz* der Klasse `Rechteck` zu erzeugen. Anschließend werden die vier Methoden der Instanz aufgerufen, um sie zu testen. Dieselben Schritte werden in Zeile 37-40 mit `Quadrat` durchgeführt. Die Syntax der überwiegend eingesetzten Ausgabemethode `printf` wird im nächsten Abschnitt behandelt.

Ein- und Ausgabe

In den bisherigen Kapiteln wurden einige Anweisungen zur Ein- und Ausgabe (auf Englisch *Input/Output* oder kurz *I/O*) ohne nähere Erklärung verwendet. In diesem Unterabschnitt werden die wichtigsten von ihnen systematischer erläutert. Los geht es mit der Zeilenausgabe auf der Konsole sowie der zugehörigen Benutzereingabe. Anschließend werden Ein- und Ausgabemethoden für Dateien und Verzeichnisse vorgestellt, wobei Sie feststellen werden, dass diese aufgrund der inzwischen in allen Betriebssystemen üblichen Abstraktion beinahe genauso funktionieren wie die entsprechenden Konsolenbefehle. Im übernächsten Kapitel werden Netzwerk-I/O-Operationen behandelt, und sogar diese sind fast identisch.

Konsolen-Ein- und -Ausgabe

Einfache Skripten kommunizieren normalerweise zeilenbasiert über die Konsole mit dem Benutzer. Einige der dafür zuständigen Anweisungen wie `puts` oder `gets` haben Sie bereits gesehen; hier werden sie genauer erläutert.

Die Standard-I/O-Kanäle

Offene Ein- und Ausgabeverbindungen werden als *Datenströme* (data streams) oder *Kanäle* (channels) bezeichnet. In den meisten modernen Betriebssystemen, darunter Windows und alle UNIX-Varianten, gibt es drei Standardkanäle für Konsolen-I/O. Das liegt daran, dass diese Betriebssysteme in C oder C++ programmiert wurden und die drei Kanäle der C-Standardbibliothek benutzen. Im Einzelnen handelt es sich um:

- `STDIN`, die *Standardeingabe*. Hier stammt die Eingabe von der Tastatur, solange sie nicht umgeleitet wird.

→

- `STDOUT`, die *Standardausgabe*. Der Text wird an die aktuelle Position des jeweiligen Terminalfensters geschrieben, falls keine Ausgabeumleitung stattfindet.
- `STDERR`, die *Standardfehlerausgabe*. Zunächst besteht scheinbar kein Unterschied zur Standardausgabe, aber der Nutzen dieses zusätzlichen Ausgabekanals besteht darin, dass Sie Fehlermeldungen oder Warnungen auch dann noch auf der Konsole ausgeben können, wenn die Standardausgabe umgeleitet wird.

Die besagte Umleitung funktioniert in ihrer einfachsten Form unter Windows und UNIX gleich: Wenn Sie an einen Konsolenbefehl (zum Beispiel den Aufruf eines Ruby-Skripts) `>DATEINAME` anhängen, wird die Standardausgabe in die angegebene Datei umgeleitet. `>>DATEINAME` hängt die Ausgabe an den bisherigen Inhalt der Datei an. `<DATEINAME` schließlich liest die Eingabe aus der angegebenen Datei.

Das folgende Beispiel schreibt den Inhalt des aktuellen Verzeichnisses in die Datei *inhalt.txt*:

```
> dir >inhalt.txt                              (Windows)
$ ls >inhalt.txt                               (UNIX)
```

Eine besondere Form der I/O-Umleitung ist die *Pipe*: Wenn Sie zwei Anweisungen mit | verketten, wird die Ausgabe der ersten als Eingabe an die zweite weitergeleitet. Das folgende Beispiel leitet den Verzeichnisinhalt an ein Suchprogramm weiter und gibt so nur diejenigen Dateien oder Verzeichnisse aus, deren Name ein *a* enthält:

```
> dir | find "a"                               (Windows)
$ ls | grep "a"                                (UNIX)
```

Vielleicht fragen Sie sich, warum die Ein- und Ausgabebefehle überhaupt in diesem Kapitel zur Objektorientierung behandelt werden – schließlich wurden Anweisungen wie `printf` oder `gets` bisher in diesem Buch ohne Objektbezug (`"irgendetwas".gets`) verwendet. Das liegt allerdings nur daran, dass Ruby dies in manchen Fällen als Abkürzung akzeptiert, wenn die Standardeingabe (bei Eingabeanweisungen wie `gets`) beziehungsweise die Standardausgabe (bei Ausgabemethoden wie `print` oder `puts`) betroffen ist. Die vollständige Schreibweise dieser Methoden folgt eigentlich dem Schema `I/O-Objekt.Methode`.

Konkrete I/O-Objekte werden in der Regel nicht aus der allgemeinen Oberklasse `I/O` gebildet, sondern aus konkreteren Unterklassen wie etwa `File` für Dateien (siehe unten). Für die Konsolen-I/O stehen die im Kasten »Die Standard-I/O-Kanäle« vorgestellten Kanäle als konstante I/O-Objekte `STDIN`, `STDOUT` und `STDERR` zur Verfügung. Ein einfaches `print` ist daher lediglich eine kompakte Schreibweise für `STDOUT.print`, und `gets` entsprechend eine Kurzfassung von `STDIN.gets`. Probieren Sie es einfach in `irb` aus:

```
>> puts "Einfache Ausgabe"
Einfache Ausgabe
>> STDOUT.puts "Einfache Ausgabe auf STDOUT"
Einfache Ausgabe auf STDOUT
>> a = gets
Hallo
=> "Hallo\n"
>> a = STDIN.gets
Hallo STDIN!
=> "Hallo STDIN!\n"
```

Für die Konsolenausgabe können Sie als Alternative zu `STDOUT` auch `STDERR` wählen, was Sie immer dann tun sollten, wenn Fehlermeldungen oder Warnungen auszugeben sind. Geben Sie zum Testen folgendes Beispielprogramm in Ihren Editor ein und speichern Sie es als `ausgabe.rb`:

```
STDOUT.puts "Dies ist eine normale Ausgabe."
STDERR.puts "Dies ist eine Warnung."
```

Führen Sie es zunächst normal aus – beide Zeilen werden untereinander ausgegeben; es geschieht nichts Besonderes:

```
> ruby ausgabe.rb
Dies ist eine normale Ausgabe.
Dies ist eine Warnung.
```

Leiten Sie nun die Standardausgabe um, damit Sie den Unterschied erkennen:

```
> ruby ausgabe.rb >ausgabe.txt
Dies ist eine Warnung.
```

Wie Sie sehen, wird nur noch die auf `STDERR` geschriebene »Warnung« auf der Konsole ausgegeben; `STDOUT` wurde dagegen in die Datei `ausgabe.txt` umgeleitet, was Sie überprüfen können, indem Sie sich deren Inhalt anzeigen lassen:

```
> type ausgabe.txt                                    (Windows)
Dies ist eine normale Ausgabe.
$ cat ausgabe.txt                                     (UNIX)
Dies ist eine normale Ausgabe.
```

Die wichtigsten Ausgabemethoden, die Sie auf `STDOUT`, `STDERR` und jedem anderen für die Ausgabe geöffneten I/O-Kanal verwenden können, sind `print`, `puts` und `printf`. Wenn Sie ihnen keinen Kanal voranstellen, wird automatisch `STDOUT` verwendet.

`print` gibt einen oder mehrere (durch Kommata getrennte) Ausdrücke aus. Weder zwischen den Ausdrücken noch am Ende wird ein Zeilenumbruch oder sonstiger Abstand eingefügt. Probieren Sie es in `irb` aus:

```
>> print "2", 3
23
```

`puts` gibt die Werte der einzelnen Ausdrücke dagegen zeilenweise aus und fügt auch hinter dem letzten einen Zeilenumbruch ein. Zum Beispiel:

```
>> puts "2", 3
2
3
```

Bereits im vorigen Kapitel wurde der Aufbau von String-Literalen besprochen – wenn Sie diese in doppelte Anführungszeichen setzen, werden Escape-Sequenzen und eingebettete Ausdrücke ausgewertet. Zum Beispiel:

```
>> puts "Naeherungswert von Pi: #{Math::PI}"
Naeherungswert von Pi: 3.14159265358979
```

Einen etwas anderen Weg geht die aus der C-Standardbibliothek stammende Methode `printf`. Das `f` steht für »Format«, denn das erste Argument ist ein Formatstring. Er kann neben beliebigen Zeichen spezielle Platzhalter enthalten, die mit `%` beginnen. Diese bestimmen, wie die nachfolgenden Argumente dargestellt werden sollen. Bevor die Details beschrieben werden, hier zunächst ein Beispiel:

```
>> printf "%d + %d = %d\n", 2, 2, 2+2
2 + 2 = 4
```

`%d` steht für eine Formatierung als Ganzzahl. Die einzelnen Platzhalter werden von links nach rechts durch die jeweiligen Argumente ersetzt. Sie müssen darauf achten, dass mindestens so viele zusätzliche Argumente wie Platzhalter vorhanden sind, ansonsten erhalten Sie eine Fehlermeldung:

```
>> printf "Nicht vorhandene Zahl: %d\n"
ArgumentError: too few arguments
```

Die wichtigsten Formatplatzhalter sind:

- `%s` – ein String. Wenn Sie eine Zahl dazwischensetzen, etwa `%10s`, wird der String auf die entsprechende Anzahl von Zeichen ergänzt und rechtsbündig im Gesamtfeld ausgerichtet. Zum Beispiel:

  ```
  printf "1.%9s\n", "Hallo"
  printf "2.%9s\n", "du liebe"
  printf "3.%9s\n", "Welt!"
  ```

  Das ergibt folgende Ausgabe:

  ```
  1.    Hallo
  2. du liebe
  3.    Welt!
  ```

  Ist der entsprechende String dagegen länger als die angegebene Zeichenanzahl, wird die Formatierung zugunsten der vollständigen Ausgabe aufgehoben.

- `%d` – eine Ganzzahl. Wenn Sie an der entsprechenden Position eine Fließkommazahl angeben, wird der Nachkommateil ohne Rundung abgeschnitten:

  ```
  >> printf "%d\n", 3
  3
  >> printf "%d\n", 3.2
  3
  >> printf "%d\n", 3.9
  3
  ```

Auch %d können Sie durch die Angabe einer Gesamtlänge rechtsbündig ausrichten. Zum Beispiel:

```
>> printf "Viel Platz:%10d\n", 7
Viel Platz:         7
```

- %f – eine Fließkommazahl. Auch hier können Sie wieder eine Gesamtlänge angeben, die für die gesamte Zahl einschließlich dem Dezimalpunkt gilt. Daneben ist es auch möglich, zusätzlich oder ausschließlich die Anzahl der Nachkommastellen anzugeben, und zwar hinter einem Punkt. Um die passenden Längen zu erreichen, wird entweder gerundet oder mit Nullen aufgefüllt. Wenn Sie sowohl die Gesamtlänge als auch eine Genauigkeit angeben, wird die Zahl gegebenenfalls wieder rechtsbündig ausgerichtet. Hier drei Beispiele:

```
>> printf "%8f\n", 3.5
3.500000
>> printf "%.2f\n", 3.5
3.50
>> printf "Wert:%8.2f\n", 3.5
Wert:    3.50
```

In Tabelle 3-1 finden Sie eine Übersicht aller zulässigen Platzhalter mit Erläuterungen und Beispielen.

Tabelle 3-1: Übersicht über die printf-Formatplatzhalter

Platzhalter	Bedeutung	Beispiel[a]
%b	(Numerisches) Argument als Dualzahl	"%b", 13 => 1101
%c	Zeichencode in Zeichen konvertieren	"%c", 97 => a
%d	Ganzzahl; Nachkommastellen werden abgeschnitten	"%d", 10 => 10 "%d", 3.9 => 3
%e	Fließkommazahlen in wissenschaftliche Schreibweise konvertieren	"%.2e", 0.0001 => 1.00e-004
%E	Wie %e, aber mit großem E	"%.2E", 1000000 => 1.00E+006
%f	Fließkommazahl; bei Nachkommastellen wird gerundet/ergänzt	"%.2f", 3.3 => 3.30 "%.2f", 4.777 => 4.78
%g	Wissenschaftliche Schreibweise, falls Exponent ≧ Nachkommastellenanzahl	"%.2g", 0.00001 => 1e-005 "%.2g", 0.1 => 0.1
%G	Wie %g, aber mit großem G	"%.2G", 0.00001 => 1E-005
%i	Entspricht %d	"%i", 17 => 17
%o	In Oktalzahl konvertieren	"%o", 100 => 144
%p	Ergebnis von Ausdruck.inspect (im Wesentlichen String-Darstellung)	"%p", [1, 2, 3] => [1, 2, 3]
%s	String; mit Genauigkeitsangabe werden maximal entsprechend viele Zeichen dargestellt	"%s", "Hallo" => Hallo "%.2s", "Hallo" => Ha
%u	Vorzeichenlose Ganzzahl	"%u", 10 => 10 "%u", -10 => ..4294967286[b]

Tabelle 3-1: Übersicht über die printf-Formatplatzhalter (Fortsetzung)

Platzhalter	Bedeutung	Beispiel[a]
%x	In Hexadezimalzahl konvertieren	"%x", 65534 => fffe
%X	Wie %x, aber mit Großbuchstaben	"%X", 65534 => FFFE
%%	Literales Prozentzeichen	"%d%%", 100 => 100%

a Die Anweisung `printf` selbst wurde aus Platzgründen jeweils weggelassen.

b Zur Interpretation dieses seltsamen (vorn abgeschnittenen) Ergebnisses lesen Sie bitte den Infokasten zur Ganzzahlarithmetik im vorigen Kapitel.

Es gibt noch einige interessante Ergänzungen für die Formatplatzhalter, die die Feinheiten der Formatierung regeln. Hier einige von ihnen im Überblick:

- `#`: präzisere Schreibweise. Fügt vor Oktalzahlen `0` und vor Hexadezimalzahlen `0x` ein; bei den wissenschaftlichen Schreibweisen werden ein Dezimalpunkt und Nachkommastellen eingefügt, auch wenn keine nötig wären. Zum Beispiel:

```
>> printf "%#x", 3333
0xd05
```

- `+`: fügt vor positiven Zahlen ein Pluszeichen ein. Zum Beispiel:

```
>> printf "%+d", 1200
+1200
```

- `-`: Das Ergebnis linksbündig statt rechtsbündig formatieren (funktioniert nur zusammen mit einer Gesamtbreite). Beispiele zum Vergleich:

```
>> printf "Vortext %10d Nachtext", 3
Vortext          3 Nachtext
>> printf "Vortext %-10d Nachtext", 3
Vortext 3          Nachtext
```

- `0`: Den freien Platz mit Nullen statt mit Leerzeichen auffüllen. Dazu wird die 0 vor die Angabe der Gesamtbreite gesetzt. Zum Beispiel:

```
>> printf "%7d", 23
     23
>> printf "%07d", 42
0000042
```

Die Fähigkeiten von `printf` können Sie auch einsetzen, um formatierte String-Ausdrücke zu bilden. Dazu steht die globale Methode `sprintf` zur Verfügung, deren Ergebnis ein String mit den entsprechenden Formatierungen ist. Hier als Beispiel eine Variablen-Wertzuweisung:

```
>> text = sprintf "%.1f / %.1f = %.1f", 5, 2, 5.0 / 2
=> "5.0 / 2.0 = 2.5"
```

Die wichtigste Methode für die Konsoleneingabe ist `gets`. Sie liest eine Zeile samt Zeilenumbruch ein, was Sie leicht in `irb` ausprobieren können:

```
>> text = gets
Hallo
=> "Hallo\n"
```

Da der abschließende Zeilenumbruch in der Regel nicht gebraucht wird, sondern stört, wird er meist sofort mit chomp entfernt:

```
>> text = gets.chomp
Hallo
=> "Hallo"
```

Sie können gets als Bedingung einer while-Schleife verwenden. Die Schleife wird dann ausgeführt, bis das Zeichen »End of File« (EOF) eingegeben wird. Während es bei Dateien – wie der Name vermuten lässt – am Dateiende automatisch erzeugt wird, muss es bei Tastatureingaben künstlich erzeugt werden – unter Windows mit **Strg + Z**, bei UNIX-Systemen mit **Strg + D**. Hier ein Beispiel, das jede eingegebene Zeile umgekehrt ausgibt:

```
while line = gets
  line.chomp!
  puts line.reverse
end
```

Speichern Sie das kurze Skript und führen Sie es aus. Geben Sie zum Abbrechen die EOF-Tastenkombination Ihrer Plattform und dann **Enter** ein. Zum Beispiel:

```
> ruby textrev.rb
Ich kehre alles um
mu sella erhek hcI
Dieser Text soll andersherum stehen.
.nehets murehsredna llos txeT reseiD
^Z
```

Versuchen Sie auch, das Skript per Eingabeumleitung auf eine Datei anzuwenden – zum Beispiel auf sich selbst:

```
> ruby textrev.rb <textrev.rb
steg = enil elihw
!pmohc.enil
esrever.enil stup
dne
```

Es gibt auch die Möglichkeit, ein einzelnes Zeichen einzulesen. Dazu wird die Methode getc verwendet, die eine explizite Angabe des I/O-Kanals benötigt – im Fall der Tastatur STDIN. Zum Beispiel:

```
>> zeichen = STDIN.getc
a
=> 97
```

Beachten Sie, dass das entsprechende Zeichen als numerischer Code zurückgeliefert wird (Sie können es wie üblich mit chr umwandeln). Noch wichtiger ist, dass Ihre Eingabe gepuffert wird – das Zeichen steht erst zur Verfügung, nachdem die **Enter**-Taste gedrückt wurde. Das ungepufferte Auslesen eines Zeichens ist leider betriebssystemabhängig und steht daher in einer plattformneutralen Skriptsprache nicht

zur Verfügung. Es gibt allerdings Erweiterungen wie die mit Ruby gelieferte Bibliothek `Curses`, die dies leisten. Quellen für weitere Informationen zu diesem Thema finden Sie in Anhang B.

Das andere Extrem ist die Methode `read`. Sie liest beliebig viel Text bis EOF und speichert ihn in einem einzigen String. Speichern Sie dazu das folgende kurze Skript unter dem Namen *read.rb*:

```
txt = STDIN.read
puts "Die Eingabe:"
puts txt
```

Führen Sie das Skript aus, geben Sie beliebig viele Zeilen ein und drücken Sie dann das EOF-Tastenkürzel. Zum Beispiel:

```
> ruby read.rb
Dies ist
ein Test
^Z
Die Eingabe:
Dies ist
ein Test
```

Dass tatsächlich eine beliebig lange Eingabe verarbeitet wird, sehen Sie, wenn Sie eine längere Textdatei an die Standardeingabe umleiten. Eine solche können Sie sich zunächst mit folgendem Skript erstellen, das die Zahlen von 1 bis 1000 ausgibt:

```
(1..1000).each { |i|
  puts i
}
```

Speichern Sie dieses Skript als *bis1000.rb* und führen Sie es aus, wobei Sie die Ausgabe in eine Datei umleiten:

```
> ruby bis1000.rb >bis1000.txt
```

Nun können Sie die neue Datei *bis1000.txt* als Eingabe für *read.rb* verwenden:

```
> ruby read.rb <bis1000.txt
Die Eingabe:
1
2
[...]
999
1000
```

Natürlich werden Sie nur die letzten Zeilen der Ausgabe sehen – die Windows-Eingabeaufforderung und moderne Terminalfenster verfügen zwar über einen Ausgabepuffer mit Scrollbalken, aber dieser ist in der Regel keine 1001 Zeilen lang.

Wenn Sie misstrauisch sind und es wirklich auf die Spitze treiben möchten, können Sie die Ausgabe von *read.rb* auch wieder in eine Datei umleiten und diese danach in Ruhe in einem Editor lesen:

```
> ruby read.rb <bis1000.txt >readtest.txt
```

Kommandozeilen-Argumente des Ruby-Skripts

Bei einfachen Konsolenprogrammen ist es oft praktischer, die zu verarbeitenden Werte gleich beim Start als Parameter einzulesen, als sie nachträglich per zeilenbasierter Eingabe anzufordern. Stellen Sie sich etwa vor, der Konsolenbefehl zum Löschen (Windows: `del`, UNIX: `rm`) müsste zunächst ohne Dateimuster eingegeben werden und würde anschließend fragen: »Welche Datei(en) möchten Sie löschen?« Umständlicher ginge es kaum – gerade die Kommandozeilen-Argumente machen die Konsole zum praktischen Arbeitsumfeld.

Um einem Ruby-Skript beim Aufruf Argumente zu übergeben, werden diese – durch Leerzeichen getrennt – hinter dessen Dateinamen (und vor eine eventuelle Ein- oder Ausgabeumleitung) geschrieben, und zwar nach dem Schema:

```
ruby Dateinam arg0 arg1 ... argn
```

Innerhalb Ihres Skripts stehen die Parameter dann im Array `ARGV` zur Verfügung; der erste befindet sich in `ARGV[0]`, der zweite in `ARGV[1]` und so weiter. Bevor Sie die Parameter auf Verdacht verarbeiten, sollten Sie zunächst mittels `ARGV.length` ihre Anzahl ermitteln. Speichern Sie dazu das folgende einzeilige Skript in einer Datei:

```
printf "Sie haben %d Argumente eingegeben.\n", ARGV.length
```

Führen Sie das Skript mehrfach mit einer unterschiedlichen Anzahl von Argumenten aus:

```
> ruby argnum.rb
Sie haben 0 Argumente eingegeben.
> ruby argnum.rb eins zwei drei
Sie haben 3 Argumente eingegeben.
```

Wenn Sie nur wenige Argumente benötigen, können Sie `ARGV[0]`, `ARGV[1]` und so weiter auch direkt testen – nicht vorhandene Parameter haben den Wert `nil`. Hier ein entsprechendes Beispiel mit nur einem Argument:

```
if ARGV[0] != nil
  puts ARGV[0]
else
  puts "Kein Argument"
end
```

Das Array `ARGV` lässt sich zudem sehr praktisch mit Hilfe des Iterators `each` bearbeiten. Das folgende Beispielskript gibt die Argumente nummeriert aus:

```
i = 1
ARGV.each { |arg|
  printf "%2d. %s\n", i, arg
  i += 1
}
```

Speichern Sie es als *args.rb* und führen Sie es mit einigen Parametern aus:

```
> ruby args.rb Ruby rocks "da House"
 1. Ruby
```

```
2. rocks
3. da House
```

Wie Sie sehen, können Sie auch einzelne Argumente mit enthaltenen Leerzeichen eingeben, indem Sie sie in Anführungszeichen setzen.

In Beispiel 3-2 sehen Sie ein längeres Skript, das einen Taschenrechner ähnlich dem Einführungsbeispiel aus dem vorigen Kapitel realisiert. Allerdings werden die beiden Zahlen und der Operator diesmal als Kommandozeilenparameter erwartet. Geben Sie das Programm zunächst ein; die Erläuterungen folgen später.

Beispiel 3-2: Taschenrechner mit Kommandozeilenargumenten, argrechner.rb

```
 1  # String-Variable mit Anleitung
 2  usage = <<ENDUSAGE
 3  Verwendung: ruby argrechner.rb -h | <num1> <op> <num2>
 4    -h:             Nur Hilfe ausgeben und beenden
 5    <num1>, <num2>: Beliebige Fliesskommazahlen
 6    <op>:           +, -, x(!), /
 7  ENDUSAGE

 8  # Bei -h nur Hilfe ausgeben und beenden
 9  if ARGV[0] == "-h"
10    STDERR.puts usage
11    exit(0)
12  end

13  # Fehler & Abbruch, wenn weniger als drei Argumente
14  if ARGV.length < 3
15    STDERR.puts "FEHLER: Nicht genug Eingabewerte"
16    STDERR.puts usage
17    exit(1)
18  end

19  # Zahlen und Operand extrahieren
20  num1 = ARGV[0].to_f
21  num2 = ARGV[2].to_f
22  op = ARGV[1]

23  # Ungueltige Operatoren aussortieren
24  if op !~ /^[\+\-x\/]$/
25    STDERR.puts "FEHLER: Ungueltiger Operator"
26    STDERR.puts usage
27    exit(1)
28  end

29  # Division durch 0 ausschliessen
30  if op == "/" && num2 == 0
31    STDERR.puts "FEHLER: Illegale Division durch 0"
32    STDERR.puts usage
33    exit(1)
34  end
```

Beispiel 3-2: Taschenrechner mit Kommandozeilenargumenten, argrechner.rb (Fortsetzung)

```
35  # Ergebnis je nach Operator berechnen
36  result = case op
37  when "+"
38    num1 + num2
39  when "-"
40    num1 - num2
41  when "x"
42    num1 * num2
43  when "/"
44    num1 / num2
45  end

46  # Ausgabe
47  printf "%.2f %s %.2f = %.2f\n", num1, op, num2, result
```

Speichern Sie das Skript unter dem Dateinamen *argrechner.rb* und führen Sie es aus. Es erwartet die Parameter in der Reihenfolge: erste Zahl, Operator, zweite Zahl. Da das Skript verschiedene Fehler abfängt, sollten Sie es mit einigen falschen Eingaben auf die Probe stellen. Hier einige Beispiele (der Verwendungshinweis wird ab dem zweiten Versuch aus Platzgründen weggelassen):

```
> ruby argrechner.rb 1
FEHLER: Nicht genug Eingabewerte
Verwendung: ruby argrechner.rb -h | <num1> <op> <num2>
  -h:            Nur Hilfe ausgeben und beenden
  <num1>, <num2>: Beliebige Fliesskommazahlen
  <op>:          +, -, x(!), /

> ruby argrechner.rb 2 % 1
FEHLER: Ungueltiger Operator

> ruby argrechner.rb 2 / 0
FEHLER: Illegale Division durch 0
```

Aber Sie können natürlich auch erfolgreiche Berechnungen durchführen:

```
> ruby argrechner.rb 17 + 4
17.00 + 4.00 = 21.00
```

Hier noch einige systematische Erläuterungen zum Skript:

- Zeile 2-7: Der Variablen `usage` wird per HIER-Dokument ein mehrzeiliger Erklärungstext über die Verwendung des Skripts zugewiesen.
- Zeile 9-12: Wenn das erste Argument (ungeachtet der Anzahl) den Wert `"-h"` hat, wird der Inhalt von `usage` ausgegeben. Obwohl dies keine Fehlermeldung ist, erfolgt die Ausgabe auf `STDERR`. Anschließend wird das Skript mittels `exit` verlassen – das Argument 0 wird dabei an das Betriebssystem zurückgegeben und bedeutet »kein Fehler«.

- Zeile 14-18: Falls die Anzahl der Argumente (`ARGV.length`) kleiner als drei ist, werden eine entsprechende Fehlermeldung und `usage` ausgegeben. Der Abbruch erfolgt dabei mittels `exit(1)` – der von 0 verschiedene Rückgabewert signalisiert dem Betriebssystem, dass etwas nicht in Ordnung ist.
- Zeile 20-22: Die beiden Zahlen und der Operator werden aus den Kommandozeilenargumenten gelesen und in anderen Variablen gespeichert. Auf die Zahlen wird die bereits bekannte Float-Umwandlungsmethode `to_f` angewendet; der Operator bleibt ein String.
- Zeile 24-28: Der eingegebene Operator wird mit dem regulären Ausdruck `/^[\+\-x\/]$/` verglichen. Dieser steht für: Ausdrucksbeginn, eines der Zeichen (`+`, `-`, `x` oder `/`), Ausdrucksende. Wie Sie bereits wissen, müssen die Sonderzeichen `+`, `-` und `/` durch einen Backslash geschützt werden, weil sie in regulären Ausdrücken selbst eine besondere Bedeutung haben. Wenn die Eingabe diesem Schema nicht entspricht (Operator `!~`), erfolgen wieder Fehlermeldung und Abbruch.

Wahrscheinlich fragen Sie sich an dieser Stelle, warum für die Multiplikation ein `x` und nicht das übliche `*` verwendet wird. Das liegt daran, dass das * für das Betriebssystem als Platzhalter gilt – beim Ruby-Skript kommt dieses Zeichen gar nicht erst an, sondern stattdessen der erste Verzeichniseintrag (UNIX) beziehungsweise der erste Verzeichniseintrag *ohne Endung* (Windows).

Genauere Details über Datei-Platzhalter finden Sie im Abschnitt Zu Unrecht gefürchtet: Arbeiten mit der Konsole in Kapitel 1.

- Zeile 30-34: Wenn der Operator `"/"` und der zweite Operand 0 ist, wird dies per Fehlermeldung und Notbremse als versuchte Division durch 0 ausgeschlossen.
- Zeile 36-44: Das Ergebnis wird mit Hilfe einer `case-when`-Fallentscheidung berechnet. Wie im vorigen Kapitel erläutert, besteht eine der nützlichen Besonderheiten von Ruby darin, dass Sie Fallentscheidungen als Ausdrücke einsetzen können, wenn Sie für den jeweiligen Fall einfach einen Wert hinschreiben.
- Zeile 47: Mit Hilfe der oben ausführlich besprochenen Methode `printf` erfolgt die Ausgabe – diesmal auf `STDOUT`. Alle drei Zahlen erhalten dabei zwei Nachkommastellen.

Datei-Ein- und -Ausgabe

Mit Hilfe der Ein- und Ausgabeumleitung können Sie aus Dateien lesen und in sie schreiben. Das ist allerdings ein interessantes Zusatzfeature für Poweruser und nicht die Art und Weise, wie man im Alltag auf Dateien zugreift. Im Kasten Die

Standard-I/O-Kanäle wurde beispielsweise gezeigt, wie Sie eine Pipe verwenden können, um die Verzeichnisausgabe mit einem Suchbefehl zu kombinieren.

Selbstverständlich besitzt Ruby auch eingebaute Methoden, um explizit auf Dateien zugreifen zu können. Diese werden hier gezeigt.

Grundlagen

Um eine Datei zur Verarbeitung zu öffnen, muss ein Objekt der Klasse `File` erzeugt werden. Das geschieht, wie beim Erstellen von Objekten üblich, mit Hilfe des *Konstruktors* `new`. Schematisch sieht das so aus:

```
var = File.new(Dateipfad, Modus)
```

Das Ergebnis von `new` wird fast immer einer Variablen zugewiesen. Diese Variable wird zu einer *Referenz* (einem Verweis) auf das Objekt, die notwendig ist, um danach weiter mit dem Objekt arbeiten zu können. Wenn Sie nämlich einfach

```
File.new(Dateipfad, Modus)
```

schreiben, wird die Datei zwar ebenfalls geöffnet und ist ein Ruby-Objekt, aber dieses befindet sich dann an einer unbekannten Stelle im Arbeitsspeicher (RAM) des Rechners und kann nicht eingesetzt werden.

Der Aufruf erwartet zwei bis drei Argumente:

- Der *Dateipfad* gibt in Form eines Strings an, wo sich die zu öffnende Datei befindet. Der Pfad kann entweder absolut oder aber relativ zum Verzeichnis des Skripts selbst angegeben werden. Der Aufbau von Pfaden wurde bereits in Kapitel 1 beschrieben. Ein absoluter Pfad könnte unter Windows zum Beispiel `"C:\Rubyskripten\test.txt"` lauten, unter UNIX dagegen `"/home/username/rubyskripten/test.txt"`. Bei Skripten zur allgemeineren Verwendung sollten aber eher relative Pfade gewählt werden – ein Beispiel wäre `"test.txt"` für die gleichnamige Datei im aktuellen Verzeichnis.
- Der *Modus* gibt an, für welche Operation die Datei geöffnet werden soll. Im einfachsten Fall handelt es sich um einen String mit einem der Werte `"r"` (read) für Lesen, `"w"` (write) für Schreiben und `"a"` (append) für Anhängen an die bestehende Datei. Es gibt noch eine andere Syntax, die den Modus mit Hilfe einiger Integer-Konstanten genauer beschreibt (zum Beispiel die Frage, ob die Datei neu erstellt werden soll, wenn sie noch nicht existiert); diese Konstanten sind allerdings stark plattformabhängig und werden deshalb in diesem Buch nicht weiter behandelt.
- Optional können Sie als drittes Argument die *Rechte* der Datei angeben (nur unter UNIX). Dateirechte bestehen aus einer dreistelligen Oktalzahl; die Stellen beschreiben von links nach rechts die Rechte des Dateieigentümers, der Dateigruppe und aller anderen Benutzer. Jede Stelle setzt sich dabei aus einer Summe von Bits mit folgenden Bedeutungen zusammen: 4 – lesen, 2 – schrei-

ben, 1 – ausführen. 0755 ist zum Beispiel typisch für Programme und Skripten, denn es bedeutet, dass der Eigentümer alles mit der Datei tun darf, während die restlichen User sie lesen und ausführen dürfen.

Das folgende Beispiel öffnet die Datei *test.txt* im aktuellen Verzeichnis zum Lesen und verwendet dafür eine Referenzvariable namens `file`:

```
file = File.new("test.txt", "r")
```

Ein gleichwertiges Synonym für `File.new` ist `File.open` – diese Schreibweise macht es gerade bei Datei-Leseoperationen klarer, dass die Datei nicht *neu* erzeugt, sondern lediglich *geöffnet* wird. Wenn Sie `open` verwenden, können Sie sogar das `File` weglassen, weil die globale Methode `open` ebenfalls eine Datei öffnet.

Somit bewirken die drei folgenden Anweisungen genau dasselbe – sie öffnen die Datei *test.txt* zum Lesen:

```
f = File.new("test.txt", "r")
f = File.open("test.txt", "r")
f = open("test.txt", "r")
```

Nachdem das Dateiobjekt zur Verfügung steht, können Sie im Prinzip – je nach Modus – die bereits bekannten Ein- oder Ausgabemethoden darauf anwenden. Um aus der soeben zum Lesen geöffneten Datei *test.txt* eine Zeile zu lesen, wird zum Beispiel folgende Anweisung verwendet:

```
line = file.gets
```

Wenn Sie mit dem Bearbeiten einer Datei fertig sind, können Sie sie mit Hilfe der Methode `close` wieder schließen:

```
file.close
```

Beim Versuch, eine nicht vorhandene Datei zum Lesen zu öffnen, erhalten Sie eine Fehlermeldung. Probieren Sie es einfach in `irb`:

```
>> file = File.new("gibtsnicht.txt", "r")
Errno::ENOENT: No such file or directory - gibtsnicht.txt
```

Es ist dagegen kein Problem, eine nicht vorhandene Datei zum Schreiben zu öffnen – in diesem Fall wird sie neu angelegt:

```
>> file = File.new("neu.txt", "w")
=> #<File:neu.txt>
```

Um in eine Datei zu schreiben, können Sie alle bereits besprochenen Ausgabemethoden darauf anwenden. Das folgende Beispiel schreibt eine Zeile in die soeben geöffnete Datei:

```
>> file.puts "Test"
=> nil
```

Um auszuprobieren, ob es funktioniert hat, müssen Sie die Datei zuerst schließen, anschließend zum Lesen öffnen und dann beispielsweise mittels `read` ihren gesamten Inhalt auslesen:

```
>> file.close
=> nil
>> file = open("x.txt", "r")
=> #<File:x.txt>
>> file.read
=> "Test\n"
```

Es ist übrigens gefährlich, eine bereits vorhandene Datei zum Schreiben zu öffnen, denn sie wird einfach *überschrieben*! Daher ist es in vielen Fällen wichtig, dass Sie vorher überprüfen, ob die Datei schon existiert. Zu diesem Zweck besitzt die Klasse `File` eine *Klassenmethode* (das ist eine Methode, die ohne konkretes Objekt funktioniert) namens `exists?`, die einen String daraufhin überprüft, ob er ein existierender Verzeichniseintrag (Datei, Unterverzeichnis oder Ähnliches) ist.

Das folgende Beispiel verlässt das Programm mit einer Fehlermeldung, falls die Datei *x.txt* bereits existiert, und öffnet diese ansonsten zum Schreiben:

```
if File.exists?("x.txt")
  STDERR.puts "FEHLER: Ausgabedatei existiert bereits!"
  exit(1)
end
# Wenn das Programm hier ankommt, existiert x.txt noch nicht,
# daher zum Schreiben oeffnen:
file = File.new("x.txt", "w")
```

Der Modus `"a"` ist interessant, wenn Sie regelmäßig Daten zur späteren Auswertung speichern möchten. Serverdienste und andere Programme, die ohne sichtbare Ausgabe im Hintergrund ausgeführt werden, führen beispielsweise so genannte *Logdateien* (auch *Protokolldateien* genannt), in denen alle wichtigen oder auch nur alle fehlerhaften Operationen festgehalten werden. Logdateien werden üblicherweise bei jedem Start des entsprechenden Programms zum Anhängen geöffnet.

Wenn Sie eine noch nicht vorhandene Datei mit dem Modus `"a"` öffnen, wird sie bequemerweise neu angelegt, genau wie bei `"w"`. Bei jedem weiteren Durchgang werden neue Inhalte an das Ende der Datei angehängt. Geben Sie zum Testen zunächst das folgende kleine Skript ein und speichern Sie es als *log.rb*:

```
log = File.new("log.txt", "a")
log.puts Time.new
if ARGV[0]
  log.printf "Eingabe: %s\n", ARGV[0]
end
log.puts "---------------------------"
log.close
```

Was dieses Skript tut, dürfte relativ offensichtlich sein: Es öffnet die Datei *log.txt* zum Anhängen und schreibt Datum und Uhrzeit hinein (siehe den nächsten Abschnitt). Falls das erste Kommandozeilenargument existiert, wird es ebenfalls hinzugefügt, und zum Schluss folgt eine Trennlinie. Führen Sie das Programm zunächst mehrmals mit wechselnden Kommandozeilenargumenten aus. Zum Beispiel:

```
> ruby log.rb "Test des Log-Skripts"
> ruby log.rb "Noch ein Test"
> ruby log.rb "Der letzte Test"
```

Schauen Sie sich danach den Inhalt von *log.txt* an; er sollte beispielsweise so aussehen:

```
Tue Nov 07 21:42:53 +0100 2006
Eingabe: Test des Log-Skripts
---------------------------
Tue Nov 07 21:43:00 +0100 2006
Eingabe: Noch ein Test
---------------------------
Tue Nov 07 21:43:06 +0100 2006
Eingabe: Der letzte Test
---------------------------
```

Wie im Zusammenhang mit der Konsoleneingabe bereits beschrieben wurde, können Sie Lese-Methoden wie `gets` als Bedingung einer `while`-Schleife verwenden, um die Zeilen bis zum Dateiende einzulesen.

Das folgende kleine Beispielskript liest die gerade angelegte Textdatei *log.txt* ein und gibt jede ihrer Zeilen aus (Ergebnis siehe oben):

```
file = File.new("log.txt", "r")
while line = file.gets
  line.chomp!
  puts line
end
file.close
```

Alternativ können Sie im Lesemodus auch den bereits besprochenen Iterator `each` auf die geöffnete Datei anwenden; dieser stellt die einzelnen Zeilen der Datei zur Verfügung. Sie können das obige Beispiel also wie folgt noch kürzer und übersichtlicher schreiben:

```
file = File.open("log.txt", "r").each { |line|
  line.chomp!
  puts line
}
```

Anwendungsbeispiel: Ein Text-»Blog«

Weblogs oder kurz Blogs sind heutzutage die einfachste Möglichkeit, um eine private Homepage zu betreiben. Wenn das Layout einmal feststeht, können Sie auf Knopfdruck einen neuen Eintrag erstellen, der automatisch ganz oben erscheint.

Ein ähnliches Konzept eignet sich auch als persönliches Notizbuch auf der Konsole. Das vorliegende Beispiel realisiert ein solches »Offline-Blog«. Auf dem Bildschirm werden jeweils fünf vorhandene Einträge angezeigt (der neueste ganz oben). Der Benutzer kann blättern, falls insgesamt mehr als fünf vorhanden sind, oder einen neuen Eintrag verfassen. Die einzelnen Postings werden in durchnummerierten Dateien namens *post1.txt*, *post2.txt* und so weiter gespeichert; die zusätzliche Datei *postings.txt* merkt sich deren bisherige Anzahl.

Geben Sie das Skript aus Beispiel 3-3 zunächst ein und speichern Sie es unter dem Namen *textblog.rb* – vorzugsweise in einem eigenen Unterordner, da dieser auch alle Eintragsdateien enthalten wird. Anschließend können Sie das Programm ausführen und testen. Nach dem Beispiel folgen einige Erläuterungen.

Beispiel 3-3: Das Konsolen-Blog, textblog.txt

```
 1  # Gesamtzahl der Postings zunaechst auf 0 setzen
 2  postings = 0
 3  # Aktuelle Position auf 0 setzen
 4  pos = 0

 5  # Anzahldatei vorhanden?
 6  if File.file?("postings.txt")
 7    # Anzahl auslesen
 8    f = File.open("postings.txt", "r")
 9    postings = f.gets.to_i
10    f.close
11    # Aktuelle Position entspricht zunaechst Anzahl
12    pos = postings
13  end

14  # Hauptschleife
15  loop do

16    # Bildschirm loeschen -- plattformabhaengig
17    if RUBY_PLATFORM =~ /win/
18      system "cls"
19    else
20      system "clear"
21    end

22    # Fuenf Eintraege ab pos ausgeben, falls vorhanden
23    if postings > 0
24      puts "Bisherige Eintraege"
25      puts "=================="
26      puts
27      # Letzter anzuzeigender Eintrag
```

Beispiel 3-3: Das Konsolen-Blog, textblog.txt (Fortsetzung)

```
28        last = pos - 4
29        last = 1 if last < 1
30        # Iterator: umgekehrte Reihenfolge
31        pos.downto(last) { |i|
32          printf "%d. -- ", i
33          # Aktuelle Datei oeffnen
34          fname = "post#{i}.txt"
35          f = open(fname, "r")
36          # Gesamten Inhalt auslesen und ausgeben
37          post = f.read
38          puts post
39          puts "-------------------"
40          f.close
41        }
42      else
43        puts "Noch keine Eintraege."
44      end

45      # Menue anzeigen
46      puts
47      print "AUSWAHL: "
48      print "(N)euere " if pos < postings
49      print "(A)eltere " if pos > 5
50      print "(S)chreiben (E)nde\n"
51      print "===> "

52      # Menueauswahl
53      wahl = gets.chomp

54      # Beenden
55      break if wahl =~ /^e/i

56      # Neuere Postings
57      if wahl =~ /^n/i
58        pos += 5
59        pos = postings if pos > postings
60        next
61      end

62      # Aeltere Postings
63      if wahl =~ /^a/i
64        pos -= 5
65        pos = 1 if pos < 1
66        next
67      end

68      # Eingabe eines neuen Eintrags
69      puts "Neuen Eintrag zeilenweise eingeben,
              leere Zeile zum Fertigstellen"
70      # Anzahl und Position erhoehen
71      postings += 1
```

Beispiel 3-3: Das Konsolen-Blog, textblog.txt (Fortsetzung)

```
72      pos = postings
73      # Neue Datei zum Schreiben oeffnen
74      fname = "post#{postings}.txt"
75      f = File.open(fname, "w")
76      # Datum/Uhrzeit als erste Zeile erzeugen
77      t = Time.new
78      eintrag = t.strftime("%d.%m.%Y, %H:%M")
79      # Schleife, bis Eintrag leerer String ist
80      while eintrag != ""
81        # Eintrag in die Datei schreiben
82        f.puts eintrag
83        # Neue Zeile einlesen
84        eintrag = gets.chomp
85      end
86      # Datei schliessen
87      f.close

88      # Neue Anzahl eintragen
89      f = File.open("postings.txt", "w")
90      f.puts postings
91      f.close
92    end
```

Hier die Beschreibung der einzelnen Teile des Skripts:

- Zeile 2-4: Zwei Variablen werden initialisiert. `postings` speichert die Anzahl der vorhandenen Einträge und `pos` die Nummer des ersten Eintrags, der auf der aktuellen Seite angezeigt wird. Da sich erst aus dem Inhalt der anschließend untersuchten, eventuell noch gar nicht vorhandenen Datei *postings.txt* ergibt, wie viele Postings vorhanden sind, ist der Anfangswert beider Variablen 0.
- Zeile 6-13: Wenn *postings.txt* existiert und eine gewöhnliche Datei ist (Methode `File.file?`), wird sie zum Lesen geöffnet. Der ausgelesene Wert wird mittels `to_i` in eine Ganzzahl umgewandelt und in `postings` gespeichert. Da die Anzeige beim neuesten Eintrag mit der höchsten Nummer beginnen soll, erhält `pos` denselben Wert.
- Zeile 15-92: Der Rest des Skripts wird in einer Endlosschleife ausgeführt, die später bei einer bestimmten Benutzereingabe verlassen wird. Innerhalb dieser Schleife kann der Benutzer in den Postings blättern oder neue verfassen.
- Zeile 17-21: Vor der Ausgabe wird der Bildschirm gelöscht. Dazu wird die Anweisung `system` verwendet, die das Betriebssystem beauftragt, den übergebenen String als Konsolenbefehl auszuführen. Das Problem ist nur, dass der entsprechende Befehl unter Windows `cls` heißt, aber in UNIX-Systemen `clear`. Glücklicherweise liefert die globale Konstante `RUBY_PLATFORM` den Namen der Systemplattform, für die die laufende Ruby-Version kompiliert wurde. Der Wert wird gegen den regulären Ausdruck `/win/` getestet, weil davon auszugehen ist, dass alle Windows-Varianten von Ruby diesen Teilstring enthalten. Im

else-Fall gilt die Plattform – etwas ungenau, aber in der Praxis wohl ausreichend – als UNIX-Variante.

- Zeile 23-42: Durch eine Überprüfung der Variable postings wird ermittelt, ob Einträge vorhanden sind. Ist das der Fall, so werden sie angezeigt, ansonsten erscheint eine Meldung, dass es noch keine gibt.
- Zeile 28-29: Da bis zu fünf Postings angezeigt werden sollen, wird hier der letzte Eintrag ermittelt: entweder pos - 4 oder einfach 1, falls auf der Seite mit den ältesten Einträgen weniger als fünf stehen.
- Zeile 31-41: Zum Auslesen der Dateien wird der Iterator downto mit den Werten von pos bis last verwendet.
- Zeile 32: Zunächst wird die aktuelle Posting-Nummer ausgegeben.
- Zeile 34: Der Dateiname des aktuellen Eintrags wird konstruiert.
- Zeile 35-40: Die Datei wird geöffnet, ihr Inhalt wird ausgelesen und angezeigt und sie wird wieder geschlossen.
- Zeile 46-51: Das Menü wird angezeigt. Das ist nur erwähnenswert, weil die Menüpunkte »(N)euere« und »(A)eltere« davon abhängig gemacht werden, ob das Blättern in die entsprechende Richtung noch funktioniert.
- Zeile 55: Wenn die Eingabe mit (großem oder kleinem) e wie »Ende« beginnt, wird die Schleife mittels break verlassen. Damit ist das Programm beendet.
- Zeile 57-61: Beginnt die Eingabe mit n, so wird pos um 5 erhöht, um die fünf nächstneueren Einträge anzuzeigen. Als Höchstgrenze wird der Wert von postings festgelegt. Dadurch passiert auch kein Fehler, falls der Benutzer auf der ersten Seite nochmals n eingibt. Nach der Wertänderung startet next einen neuen Schleifendurchlauf und springt so direkt zur Ausgabe.
- Zeile 63-67: Hier wird pos entsprechend um 5 verringert (Mindestwert 1), um ältere Postings anzuzeigen.
- Zeile 70-71: Wenn das Skript hier ankommt, wurde die Schleife noch nicht verlassen oder neu gestartet. Es findet also die Eingabe eines neuen Eintrags statt. Dazu wird postings zuerst um 1 erhöht; auch pos erhält den neuen Wert dieser Variablen.
- Zeile 74-75: Wie in Zeile 34 wird der Name der Posting-Datei generiert, danach wird die Datei geöffnet, diesmal allerdings zum Lesen.
- Zeile 77: Ein Time-Objekt wird erzeugt; es enthält automatisch das Systemdatum und die -uhrzeit.
- Zeile 78: In der Variablen eintrag werden als Anfangswert Datum und Uhrzeit in einem Format wie 22.10.2006, 18:14 eingetragen. Damit wird als erste Zeile eines jeden Eintrags auf jeden Fall ein Zeitstempel eingetragen. Die Funktionsweise der Formatierungsmethode strftime wird weiter unten im Abschnitt »Datum und Uhrzeit« beschrieben.

- Zeile 80-85: Die Eingabeschleife läuft, solange `eintrag` kein leerer String ist. Beim ersten Durchlauf ist diese Bedingung erfüllt, weil Datum und Uhrzeit darin gespeichert wurden. Innerhalb der Schleife wird der aktuelle Wert von `eintrag` zuerst in die Datei geschrieben, erst danach erfolgt die Eingabe der nächsten Zeile. Diese etwas eigenwillige Reihenfolge stellt sicher, dass die als Abbruchbedingung einzugebende Leerzeile (einfach **Enter**) nicht in der Datei landet.
- Zeile 87: Die Eingabe ist beendet und die Datei wird geschlossen.
- Zeile 89-91: Abschließend wird die neue Anzahl in *postings.txt* geschrieben. Danach erfolgt automatisch ein neuer Durchlauf der Hauptschleife; da `pos` bereits in Zeile 72 auf den neuen Wert von `postings` gesetzt wurde, kann der Benutzer seinen neuen Eintrag sofort lesen.

In Abbildung 3-1 sehen Sie das Textblog nach der Eingabe des sechsten Eintrags.

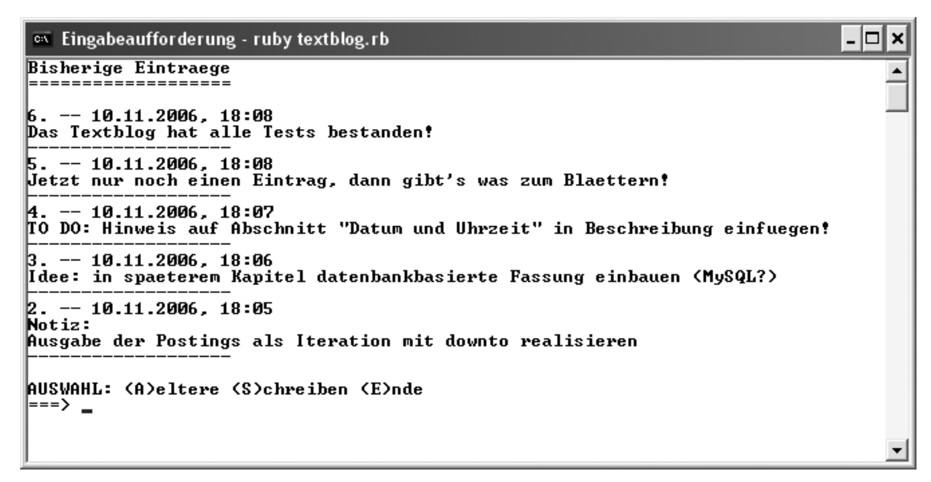

Abbildung 3-1: Das Textblog im Einsatz

Erweiterte Dateizugriffe

Sie können Dateien übrigens nicht nur sequenziell (Zeile für Zeile) bearbeiten. Der *Dateizeiger*, das heißt die aktuelle Schreib- oder Leseposition in der Datei, kann verschoben werden, und Sie können Dateien auch zum gemischten Lesen und Schreiben öffnen. Damit lassen sich gezielt bestimmte Stellen einer Datei einlesen oder sogar ersetzen. Es wäre vorstellbar, auf diese Weise eine Art eigener Datenbank zu programmieren.[1]

---

1 In der Praxis sind echte Datenbanken immer vorzuziehen, weil sie stabiler sind und einen gewissen Schutz vor Datenverlust bieten. Andererseits benötigt ein User in diesem Fall die entsprechende Datenbank-Software, so dass man für kleinere Anwendungen überlegen könnte, diese Funktionalität selbst ins Programm einzubauen.

Die File-Methode seek dient dazu, den Dateizeiger auf ein bestimmtes Byte zu setzen. Stellen Sie sich vor, die Datei *0bis9.txt* enthält nur eine Zeile mit den Ziffern von 0 bis 9:

```
0123456789
```

Sie können diese Datei entweder in einem Editor anlegen oder mit Hilfe folgender Ruby-Anweisungen:

```
f = File.new("0bis9.txt", "w")
f.print "0123456789"
f.close
```

Öffnen Sie diese Datei nun zum Lesen, um seek auszuprobieren:

```
>> f = File.open("0bis9.txt", "r")
=> #<File:0bis9.txt>
```

Setzen Sie den Dateizeiger jetzt auf Byte Nummer 5:

```
>> f.seek(5)
=> 0
```

Nun können Sie die Zeile einlesen:

```
>> f.gets
=> "56789"
```

Wie Sie sehen, werden die Bytes – wie üblich – von 0 an gezählt.

Ein weiterer Versuch, aus der Datei zu lesen, liefert keinen Wert mehr, da der Dateizeiger am Dateiende angekommen ist:

```
>> f.gets
=> nil
```

Um danach wieder weiter vorn in der Datei zu lesen, können Sie erneut seek verwenden. Um den Dateizeiger wieder ganz an den Anfang zu setzen, können Sie statt f.seek(0) auch folgende Methode aufrufen:

```
>> f.rewind
=> 0
```

Lesen Sie nochmals eine Zeile, um zu testen, ob es geklappt hat:

```
>> f.gets
=> "0123456789"
```

Die Methode seek besitzt einen optionalen zweiten Parameter, der angibt, wie der erste interpretiert werden soll. Als mögliche Werte stehen drei Konstanten der Klasse IO zur Verfügung:

- file.seek(n, IO::SEEK_SET) ist das Standardverhalten: Der Dateizeiger wird zur absoluten Byteposition n bewegt.
- file.seek(n, IO::SEEK_CUR) bewegt den Dateizeiger um n Bytes von der aktuellen Position an. Sie können auch negative Werte angeben, um ihn weiter in Richtung Dateianfang zu verschieben.

- `file.seek(n, IO::SEEK_END)` interpretiert `n` relativ zum Dateiende. Dabei muss `n` natürlich einen negativen Wert erhalten.

Um eine Datei zum gemischten Lesen und Schreiben zu öffnen, wird der spezielle Modus `"r+"` verwendet. Öffnen Sie die Datei *0bis9.txt* auf diese Weise, um es auszuprobieren:[2]

```
>> f = File.open("0bis9.txt", "r+")
=> #<File:0bis9.txt>
```

Analog zu einem bekannten Kinderspiel sollen nun alle Vielfachen von 3 in der Liste durch ein +-Zeichen ersetzt werden. Das ist mit `seek` und `putc` sehr einfach, denn eine Schreiboperation in einer zum Lesen und Schreiben geöffneten Datei ersetzt die Zeichen an der entsprechenden Position. Verwenden Sie am einfachsten folgende Iteration:

```
>> 3.step(9,3) { |i| f.seek(i); f.putc "+"}
=> 3
```

Nun können Sie den Dateizeiger zurücksetzen und eine Zeile lesen, um herauszufinden, ob alles funktioniert hat:

```
>> f.rewind
=> 0
>> f.gets
=> "012+45+78+"
```

Eine weitere interessante Dateimethode ist `sort`. Sie liest sämtliche Zeilen einer Datei aus und gibt sie als sortiertes Array zurück, was bei der Ausgabe in Dateien gespeicherter Listen manchmal ein zusätzliches Sortieren erspart. Erzeugen Sie zum Testen eine Textdatei namens *unsort.txt* mit nicht sortierten Zeilen, beispielsweise:

```
C
A
D
E
B
```

Öffnen Sie diese Datei anschließend zum Lesen und rufen Sie ihre Methode `sort` auf:

```
>> f = File.open("unsort.txt", "r")
=> #<File:unsort.txt>
>> f.sort
=> ["A\n", "B\n", "C\n", "D\n", "E\n"]
```

Verzeichnisse lesen

Ruby kann nicht nur auf Dateien zugreifen, sondern auch auf Verzeichnisse. Das wird beispielsweise erforderlich, sobald Sie Benutzern Ihrer Skripten die Möglichkeit einräumen, selbst über Dateinamen zum Speichern zu bestimmen. Sie könnten

2 Vorher müssen Sie die Datei natürlich schließen, wenn Sie die vorige Übung mitgemacht haben: `f.close`.

dann etwa eine Liste der vorhandenen Dateien anzeigen, wenn es darum geht, sie wieder zu laden oder beim Speichern nicht noch einmal denselben Dateinamen zu verwenden.

Um ein Verzeichnis auszulesen, müssen Sie ein Objekt der Klasse `Dir` erzeugen; als Argument benötigt es den relativen oder absoluten Pfad des gewünschten Verzeichnisses im Stringformat:

```
dir = Dir.new(Verzeichnispfad)
```

Wenn Sie das aktuelle Verzeichnis öffnen möchten, in dem sich Ihr Skript befindet, lautet der relative Pfad `"."` – in beiden Systemwelten das Kürzel für »aktuelles Arbeitsverzeichnis«.

Nachdem das Verzeichnis geöffnet ist, können Sie seine Methode `read` verwenden, um einen Eintrag auszulesen. Wenn Sie alle Einträge brauchen, können Sie `read` in der Bedingung einer `while`-Schleife einsetzen. Die folgende Schleife gibt die Verzeichniseinträge einfach untereinander aus:

```
while entry = dir.read
  puts entry
end
```

Zum Schluss sollten Sie die Methode `close` aufrufen, um das Verzeichnisobjekt freizugeben:

```
dir.close
```

Die ausgelesenen Verzeichniseinträge sind einfache Datei- oder Unterverzeichnisnamen ohne Pfadangabe. Die Spezialeinträge `.` und `..` (Verweis auf das aktuelle beziehungsweise übergeordnete Verzeichnis) werden stets mitgeliefert. Hier als Beispiel der auf obige gezeigte Weise ausgelesene Inhalt eines Verzeichnisses mit Ruby-Skripten (Ausschnitt) – einige Dateinamen kommen Ihnen wahrscheinlich bekannt vor:

```
.
..
agecheck.rb
arg.rb
argnum.rb
argrechner.rb
args.rb
argvt.rb
ausgabe.rb
ausgabe.txt
bis1000.rb
bis1000.txt
[...]
```

Wenn es darum geht, die Einträge weiterzuverarbeiten, sollten Sie zunächst einmal feststellen, ob es sich um gewöhnliche Dateien, Unterverzeichnisse oder etwas

anderes handelt. Dazu können unter anderem folgende Klassenmethoden von `File` verwendet werden:

- `File.exists?(Pfad)`wurde bereits erwähnt. Die Methode liefert `true` zurück, wenn der angegebene Pfad ein existierender Verzeichniseintrag ist, und `false`, falls er nicht existiert.
- `File.file?(Pfad)` hat dagegen nur dann das Ergebnis `true`, wenn der untersuchte Pfad eine reguläre Datei ist, ansonsten erhalten Sie das Resultat `false`.
- `File.directory?(Pfad)` ergibt `true`, wenn der Pfad ein Verzeichnis ist, und `false`, wenn er kein Verzeichnis ist.

Diese Testmethoden helfen im nächsten Kapitel bei einem ehrgeizigeren Projekt: Dort sollen nicht nur die Inhalte eines Verzeichnisses ausgelesen werden, sondern auch die Inhalte aller ineinander verschachtelten Unterverzeichnisse sollen eingerückt dargestellt werden.

Datum und Uhrzeit

Jeder Rechner enthält heutzutage eine Uhr, die durch eine Batterie auch im ausgeschalteten Zustand am Laufen gehalten wird. Daher ist der Umgang mit Datums- und Zeitinformationen zu einer der wichtigsten Fähigkeiten von Computerprogrammen geworden. Zu den bekanntesten Anwendungen gehören die typischen Zeitstempel für Dateizugriffe und natürlich die allgegenwärtigen Terminkalender-Anwendungen.

Alle modernen Programmiersprachen sind mit Funktionen ausgestattet, um auf Systemdatum und -uhrzeit zuzugreifen. In Ruby funktioniert das Ganze mit Hilfe verschiedener Klassen, deren Objekte etwa einen bestimmten Zeitpunkt repräsentieren. Verschiedene Methoden ermöglichen dann den Zugriff auf die einzelnen Bestandteile von Datum und Uhrzeit. In diesem Abschnitt werden verschiedene Aspekte der wichtigsten Datums- und Uhrzeitklasse `Time` vorgestellt.

Die Klasse Time

Die einfachste Möglichkeit, die aktuelle Systemzeit auszulesen, besteht darin, ein Objekt der Klasse `Time` ohne Argumente zu erzeugen. Das geht mittels

```
var = Time.new
```

oder dem Synonym

```
var = Time.now
```

Das Objekt, auf das wie üblich über die Referenzvariable zugegriffen wird, speichert Datum und Uhrzeit im Augenblick seiner Erzeugung. Beachten Sie, dass es seiner-

seits *nicht* wie eine Uhr weiterläuft – wenn Sie eine aktualisierte Uhrzeit benötigen, müssen Sie auch ein neues Objekt erzeugen.

Wenn Sie Datum und Uhrzeit nur kurz ausgeben möchten, brauchen Sie das Objekt noch nicht einmal in einer Referenzvariablen zu speichern. Die zu Beginn dieses Kapitels beschriebenen Ausgabemethoden sorgen automatisch dafür, dass ein `Time.new`-Aufruf als String ausgegeben wird. Versuchen Sie es in `irb`:

```
>> puts Time.new
Fri Oct 20 22:40:47 +0200 2006
```

Das Format können Sie sich so allerdings nicht aussuchen. Es handelt sich um das RFC 1123-Format[3], das beispielsweise auch für viele Server-Logdateien verwendet wird. In Ruby 1.8.5 wurde es übrigens leicht modifiziert: Statt dem Namen der Zeitzone wird als vorletzte Komponente nun die Differenz zur UTC (Weltstandardzeit, Greenwich Mean Time ohne Sommerzeit) angezeigt. Bis Version 1.8.4 lautete die Ausgabe dagegen zum Beispiel:

```
Fri Oct 20 22:40:47 Westeuropaeische Normalzeit 2006
```

Für Endanwenderprogramme ist dieses Format in jedem Fall nicht empfehlenswert, da die merkwürdige Reihenfolge der Datums- und Uhrzeitbestandteile weder kontinentaleuropäischen noch britisch-amerikanischen Standards entspricht. Daher sollten Sie sich mit den vielen Methoden der Klasse `Time` vertraut machen, die die einzelnen Elemente von Datum und Uhrzeit extrahieren und diese selbstständig zusammensetzen. Hier die wichtigsten im Überblick (die Referenzvariable für das `Time`-Objekt heißt jeweils `t`):

- `t.year` – die vierstellige Jahreszahl, zum Beispiel 2006
- `t.month` – der Monat in numerischer Darstellung (1-12)
- `t.day` – der Tag im Monat (1-31)
- `t.wday` – der numerisch codierte Wochentag (0=So., 1=Mo., ..., 6=Sa.)
- `t.hour` – die Stunde (0-23)
- `t.min` – die Minute (0-59)
- `t.sec` – die Sekunde (0-59)

Damit können Sie sich eine für deutschsprachige User gut lesbare Datums- und Uhrzeitanzeige zusammenbasteln. Zum Beispiel:

```
t = Time.new
printf "%02d.%02d.%4d, %02d:%02d:%02d\n", t.day, t.month, t.year,
       t.hour, t.min, z.sec
```

---

3 RFC heißt »Request For Comments« (Bitte um Kommentare). Es handelt sich um die öffentlich verfügbare Dokumentation von Protokollen, Diensten und Techniken des Internets und seiner Vorläufernetze, die seit 1969 gesammelt werden. Sie können die inzwischen gut 4.600 meist sehr technischen und trockenen (Ausnahme: einige Spaß-RFCs vom 1. April, z.B. RFC 2324) Dokumente zum Beispiel unter *http://www.ietf.org/rfc.html* abrufen.

Die Ausgabe sieht beispielsweise so aus:

```
21.10.2006, 10:32:07
```

Zur Erinnerung: Der Formatplatzhalter `%02d` bedeutet, dass die entsprechende Ganzzahl auf mindestens zwei Stellen ergänzt werden soll und dass als eventuelles Füllzeichen kein Leerzeichen, sondern eine 0 verwendet wird.

Ein wenig mehr Arbeit ist es, den Wochentag dazu als Text auszugeben, und nach Möglichkeit auch noch den Monat. Sie lernen zwar als Nächstes die Formatierungsmethode `strftime` für `Time`-Objekte kennen, die auch diese Werte liefern kann, das allerdings nur auf Englisch. Deshalb hier zunächst eine Anleitung für die Extrahierung deutscher Wochentags- und Monatsnamen.

Wenn Sie sich den Wertebereich der Wochentagsnummern ansehen, ist der Fall eigentlich klar: Die möglichen Werte 0 bis 6 lassen sich ohne Weiteres als Index auf ein Array mit Wochentagsnamen anwenden. Wichtig ist nur, dass Sie die Reihenfolge beachten – in der englischsprachigen Welt beginnt die Woche mit dem Sonntag. Das Array muss also folgendermaßen aussehen:

```
wtage = %w(Sonntag Montag Dienstag Mittwoch
           Donnerstag Freitag Samstag)
```

Beim Monat wird es etwas schwieriger: Der Wertebereich ist 1 bis 12, wie in der normalen Kalenderschreibweise. Für den Zugriff auf ein Array müssen Sie 1 davon abziehen. Das Array selbst können Sie aber erst einmal definieren:

```
monate = %w(Januar Februar Maerz April Mai Juni Juli August
            September Oktober November Dezember)
```

Nun können Sie den Rückgabewert der Methode `wday` als Index auf `wtage` und den um 1 verminderten Wert von `month` als Index auf `monate` anwenden:

```
t = Time.new
wtag = wtage[t.wday]
monat = monate[t.month - 1]
```

Nun können Sie Datum und Uhrzeit vollständig ausgeben:

```
printf "%s, %02d. %s %4d, %02d:%02d:%02d\n", wtag, t.day, monat,
       t.year, t.hour, t.min, t.sec
```

Ein Ausgabebeispiel könnte so aussehen:

```
Samstag, 21. Oktober 2006, 10:51:46
```

Um direkt aus dem `Time`-Objekt ein nach Ihren Wünschen formatiertes Ergebnis zu extrahieren, können Sie seine Methode `strftime` aufrufen. Der Name ist eine Abkürzung für »String Format Time«; eine gleichnamige Funktion mit praktisch identischer Formatsyntax steht seit vielen Jahren in der C-Standardbibliothek zur Verfügung. Die Ruby-Variante besitzt folgende Grundsyntax, wobei `t` wieder ein `Time`-Objekt ist:

```
t.strftime(Formatstring)
```

Der Formatstring ist ein beliebiger String, in dem – ähnlich wie bei `printf` – spezielle `%`-Zeichen-Sequenzen für die einzelnen Zeit-Aspekte stehen. In Tabelle 3-2 finden Sie eine Übersicht über alle verfügbaren Formate.

Tabelle 3-2: Formatplatzhalter für die Methode Time.strftime

Platzhalter	Bedeutung	Ausgabebeispiel
`%a`	Wochentag (Abkürzung)	`Fri`
`%A`	Wochentag ausgeschrieben	`Friday`
`%b`	Monat (Abkürzung)	`Oct`
`%B`	Monat ausgeschrieben	`October`
`%c`	Datum und Uhrzeit	`Sat Oct 21 10:07:01 2006`[a]
`%d`	Tag im Monat (1-31)	`20`
`%H`	Stunde (00-23)	`21`
`%I`	Stunde (00-12)	`09`
`%j`	Tag im Jahr	`293`
`%m`	Numerischer Monat	`10`
`%M`	Minute (00-59)	`27`
`%p`	AM oder PM für 12-Stunden-Anzeige	`PM`
`%S`	Sekunde (00-59)	`15`
`%U`	Kalenderwoche (beginnend mit Sonntag)	`42`
`%w`	Numerischer Wochentag (0=So., 1=Mo., ..., 6=Sa.)	`5`
`%W`	Kalenderwoche (beginnend mit Montag)	`42`
`%x`	Datum (englisch, MM/TT/JJ)	`10/20/06`
`%X`	Uhrzeit	`21:27:15`
`%y`	Jahr (zweistellig)	`06`
`%Y`	Jahr (vierstellig)	`2006`
`%z`	Zeitzonendifferenz zu GMT	`+1:00`[b]
`%Z`	Zeitzone	`GMT+1:00 (UNIX)` `Westeuropaeische Normalzeit (Windows)`
`%%`	Literales Prozentzeichen	`%`

a Unter Windows ist das Format MM/TT/JJ hh:mm:ss (z.B. 10/20/06 21:27:15).

b Unter Windows wird hier ebenfalls die Zeitzone angezeigt.

Hier zwei `irb`-Beispiele – einmal das rein numerische Datum in deutscher Reihenfolge und einmal ein sehr ausführliches englisches Datum:

```
>> t.strftime("%d.%m.%Y, %H:%M")
=> "21.10.2006, 12:16"
>> t.strftime "%A, %B %d, %Y, %I:%M %p"
=> "Saturday, October 21, 2006, 12:16 PM"
```

Wenn Sie nicht die aktuelle Systemzeit, sondern ein anderes Datum und/oder eine andere Uhrzeit speichern möchten, können Sie die `Time`-Methode `parse` aufrufen. Das Ergebnis ist ein neues `Time`-Objekt, in dem der angegebene Zeitpunkt codiert ist. `parse` ist sehr tolerant bezüglich der möglichen Formate. Nehmen Sie für einige Tests an, die aktuelle Zeit sei

```
Sat Oct 21 13:44:29 Westeuropaeische Normalzeit 2006
```

Prinzipiell können Sie Datum und Uhrzeit, nur ein Datum oder nur eine Uhrzeit angeben. Eine vollständige Angabe machen Sie am einfachsten als String im Format `"JJJJ/MM/TT hh:mm:ss"`. Zum Beispiel:

```
>> d = Time.parse("1999/09/13 17:59:00")
=> Mon Sep 13 17:59:00 Westeuropaeische Normalzeit 1999
```

Wenn Sie nur ein Datum ohne Uhrzeit angeben, wird 00:00 Uhr eingestellt:

```
>> d = Time.parse("1999/09/13")
=> Mon Sep 13 00:00:00 Westeuropaeische Normalzeit 1999
```

Eine Uhrzeit ohne Datum verwendet dagegen das Systemdatum mit der angegebenen Uhrzeit:

```
>> d = Time.parse("9:13")
=> Sat Oct 21 09:13:00 Westeuropaeische Normalzeit 2006
```

Sie können auf diese Weise keinen Zeitpunkt vor dem 01.01.1970, 00:00 Uhr speichern. Dieses spezielle Datum – Spitzname EPOCH – gilt als UNIX-Erfindungsdatum. Die C-Standardbibliothek, auf der alle wichtigen Betriebssysteme und Programmiersprachen (wie Ruby) basieren, speichert alle Zeitangaben in Sekunden – oder in neueren Implementierungen Millisekunden – seit EPOCH.

Um von einem bestimmten Datum zu einem anderen gelangen, können Sie eine Sekundenzahl zu einem Datum addieren oder von diesem abziehen. Die folgenden Beispiele addieren nacheinander eine Stunde, einen Tag und eine Woche zur aktuellen Systemzeit:

```
>> t = Time.new
=> Sun Oct 22 12:11:31 +0200 2006
>> t + 3600
=> Sun Oct 22 13:11:31 +0200 2006
>> t + 86400
=> Mon Oct 23 12:11:31 +0200 2006
>> t + 86400 * 7
=> Sun Oct 29 11:11:31 +0100 2006
```

Wichtig ist in diesem Fall, dass Sie hinter dem `+` beziehungsweise `-`, anders als bei den gleichnamigen arithmetischen Operationen, ein Leerzeichen setzen. Andernfalls erhalten Sie eine Fehlermeldung.

Der »Zeitsprung« im letzten Beispiel liegt daran, dass die Uhr am 29.Oktober 2006 um 03:00 Uhr morgens von Sommer- auf Winterzeit umgestellt wurde. Das sehen Sie auch an der geänderten UTC-Differenz (vorher zwei Stunden, danach eine Stunde).

Anwendungsbeispiel: Ein kleiner Konsolenkalender

Das kleine Skript in diesem Unterabschnitt gibt einen Monatskalender aus, wobei Monat und Jahr aus dem Systemdatum ermittelt werden. Tippen Sie das Listing aus Beispiel 3-4 zunächst ein und speichern Sie es als *kalender.rb*. Danach erhalten Sie die übliche Beschreibung der Programmzeilen und ein Ausgabebeispiel.

Beispiel 3-4: Der Konsolen-Monatskalender, kalender.rb

```
 1   # Datum ermitteln
 2   jetzt = Time.new
 3   d = jetzt.day
 4   m = jetzt.month
 5   y = jetzt.year

 6   # Laengen aller Monate
 7   monate = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
 8   # Schaltjahr?
 9   if ((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0)
10     monate[1] = 29
11   end

12   # Laenge des aktuellen Monats
13   mlaenge = monate[m - 1]

14   # Wochentag des Monatsersten
15   erster = Time.parse("#{y}/#{m}/01")
16   wt = erster.wday
17   # Sonntag => 7 (europ. Woche)
18   wt = 7 if wt == 0

19   puts "#{m}/#{y}"
20   puts
21   puts " MO  DI  MI  DO  FR  SA  SO"

22   # Leerzeichen von Montag bis Wochentag des Ersten
23   leer = (wt - 1) * 4
24   print " " * leer

25   1.upto(mlaenge) { |tag|
26     # Wochenwechsel?
27     if wt > 7
28       wt = 1
29       print "\n"
30     end
31     # Aktueller Tag?
```

Beispiel 3-4: Der Konsolen-Monatskalender, kalender.rb (Fortsetzung)

```
32      if tag == d
33        printf "[%2d]", tag
34      else
35        printf "%3d ", tag
36      end
37      wt += 1
38    }
```

Wenn Sie das Skript ausführen, erhalten Sie eine Ausgabe wie in Abbildung 3-2.

```
Eingabeaufforderung

C:\rubytest>ruby kalender.rb
10/2006

 MO  DI  MI  DO  FR  SA  SO
                          1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21 [22]
 23  24  25  26  27  28  29
 30  31
C:\rubytest>_
```

Abbildung 3-2: Ausgabe des Konsolenkalenders am 22.10.2006

Hier die wichtigsten Programmschritte im Überblick:

- Zeile 2-5: Das aktuelle Systemdatum wird ermittelt, anschließend werden die Bestandteile Tag, Monat und Jahr extrahiert.
- Zeile 7-11: Die Längen der verschiedenen Monate werden in einem Array gespeichert. Zeile 9 prüft, ob das aktuelle Jahr ein Schaltjahr ist: durch 4, aber nicht durch 100 teilbar, oder aber durch 400 teilbar.
- Zeile 13: Aus dem Array wird die Länge des aktuellen Monats ermittelt.
- Zeile 15-18: Mittels `Time.parse` wird ein neues Datumsobjekt erzeugt. Monat und Jahr stimmen mit dem Systemdatum überein, aber als Tag wird der Monatserste eingetragen, um dessen Wochentag zu ermitteln. Wenn es sich um den Sonntag (Wert 0) handelt, wird er auf 7 geändert, weil der Kalender kontinentaleuropäisch sortiert werden soll.
- Zeile 23-24: Die erste Woche wird mit Leerzeichen für die nicht vorhandenen Wochentage aufgefüllt – vier Zeichen für jeden Tag.
- Zeile 25-38: Der Operator `upto` zählt von 1 bis zum Monatsletzten.

- Zeile 27-30: Wenn der Wochentag größer als 7 (Sonntag) ist, wird er auf 1 zurückgesetzt, und es erfolgt ein Zeilenumbruch.
- Zeile 32-36: Der Tag wird ausgegeben. Falls es sich um den heutigen Tag handelt, wird er von eckigen Klammern umgeben.

Anregung: Denken Sie darüber nach, wie Sie das Skript durch zwei optionale Parameter (Monat und Jahr) erweitern könnten, um alternativ auch einen Kalender für einen anderen Monat ausgeben zu können. Einen Lösungsvorschlag finden Sie auf der Webseite zum Buch.

Einige weitere Klassen

Nachdem Sie nun Klassen für zwei wichtige Anwendungsgebiete kennengelernt haben, wissen Sie bereits gut, wie man mit vorgefertigten Klassen arbeitet. Deshalb sollen hier in diesem kurzen Abschnitt noch einige weitere interessante Klassen erwähnt werden.

Bruchrechnen mit Rational

Im vorigen Kapitel haben Sie ausführliche Informationen über die Arbeit mit Ganzzahlen und Fließkommazahlen erhalten. Aber wussten Sie, dass Ruby auch mit echten Brüchen arbeiten kann? Dafür zuständig ist die Klasse `Rational`, die Zähler und Nenner eines Bruchs speichert. Sie enthält auch alle wichtigen Operatoren, um mit Brüchen zu rechnen – einschließlich dem dafür notwendigen Erweitern und Kürzen. Mit echten Brüchen lässt sich oftmals exakter und auch angenehmer rechnen als mit Fließkommazahlen.

Ein `Rational`-Objekt wird folgendermaßen erzeugt (ein `new`-Aufruf ist bei dieser Klasse nicht möglich):

```
>> bruch = Rational(2, 3)
=> Rational(2, 3)
```

Wenn Sie einen Bruch erzeugen, der sich noch kürzen lässt, geschieht dies automatisch. Zum Beispiel:

```
>> bruch2 = Rational(5, 10)
=> Rational(1, 2)
```

Arithmetische Operationen sind sowohl mit anderen Brüchen als auch mit Ganz- oder Fließkommazahlen möglich:

```
>> bruch = Rational(1, 2)
=> Rational(1, 2)
>> bruch / 2
=> Rational(1, 4)
>> bruch + Rational(1, 3)
=> Rational(5, 6)
```

Zähler und Nenner des Bruchs erhalten Sie mit Hilfe der Methoden `numerator` beziehungsweise `denominator`:

```
>> printf "%d/%d", zweidrittel.numerator, zweidrittel.denominator
2/3
```

Schließlich können Sie den Bruch auch mit Hilfe der wohlbekannten Methode `to_f` in eine Fließkommazahl umwandeln:

```
>> zweidrittel.to_f
=> 0.666666666666667
```

Objektorientierte reguläre Ausdrücke

Auch die im letzten Kapitel gründlich erläuterten regulären Ausdrücke besitzen einen klassenbasierten Zugang, und zwar über die Klassen `Regexp` und `MatchData`. Für den Alltagsgebrauch sind diese oft zu umständlich, aber sie haben einige interessante zusätzliche Features, die hier gezeigt werden.

Ein `Regexp`-Objekt speichert einen regulären Ausdruck. Es wird mit Hilfe von `new` oder dem Synonym `compile` erzeugt. Für die Argumente gibt es zwei Schreibweisen. Die eine Möglichkeit ist ein literaler regulärer Ausdruck – hier beispielsweise ein oder mehrere Buchstaben ohne Berücksichtigung der Groß- und Kleinschreibung:

```
r = Regexp.new(/[a-z]+/i)
```

Die Alternative besteht darin, einen String zu verwenden, der in einen regulären Ausdruck umgewandelt wird, und die Modifizierer als zweites Argument anzugeben. Hier das obige Beispiel in dieser Syntax:

```
>> r = Regexp.new("[a-z]+", "i")
=> /[a-z]+/i
```

Die Optionen müssen Sie in diesem Fall nicht als Strings angeben, sondern können sie auch durch eine bitweise Oder-Kombination aus folgenden Konstanten ersetzen:

- `Regexp::EXTENDED` entspricht dem Modifier `x`: Zur Formatierung verwendeter Whitespace wird ignoriert.
- `Regexp::IGNORECASE` steht für `i`: In den untersuchten Strings spielen Groß- und Kleinschreibung keine Rolle mehr.
- `Regexp::MULTILINE` ersetzt das `m`, also die Suche über Zeilengrenzen hinweg.

Damit sieht das Beispiel wie folgt aus:

```
>> r = Regexp.new("[a-z]+", Regexp::IGNORECASE)
=> /[a-z]+/i
```

Denken Sie bei der String-Schreibweise daran, dass Sie Escape-Sequenzen berücksichtigen müssen – ein Backslash (\) muss beispielsweise durch zwei (\\) ersetzt werden. Hier als Beispiel ein regulärer Ausdruck für die Suche nach beliebig viel Whitespace, einmal in Regexp- und einmal in String-Schreibweise:

```
r1 = Regexp.new(/\s+/)
r2 = Regexp.new("\\s+")
```

Um einen String mit einem auf diese Weise konstruierten regulären Ausdruck zu vergleichen, können Sie entweder den bereits bekannten Operator `=~` oder aber die Methode `match` des `Regexp`-Objekts verwenden. Zum Testen wird der folgende reguläre Ausdruck zur Suche nach einer deutschen Postleitzahl[4] konstruiert:

```
plz = Regexp.new("\\d{5}")
```

Hier eine Beispieladresse, die untersucht werden soll:

```
adr = "O'Reilly Verlag GmbH & Co. KG, Balthasarstr. 81, 50670 Koeln"
```

Die bereits bekannte Schreibweise für den Mustervergleich lautet:

```
>> adr =~ plz
49
```

Der Wert 49 ist die Position im String, an der die Postleitzahl beginnt. Übrigens ist die umgekehrte Schreibrichtung

```
plz =~ adr
```

absolut gleichwertig und liefert daher dasselbe Ergebnis.

Auch bei `match` kann die Reihenfolge vertauscht werden, da sowohl reguläre Ausdrücke als auch Strings diese Methode besitzen:

```
>> plz.match(adr)
=> #<MatchData:0x2affaa8>
>> adr.match(plz)
=> #<MatchData:0x2afbd18>
```

Mit beiden Ergebnissen können Sie so nichts anfangen. Es sind `MatchData`-Objekte, die Sie speichern müssen, um anschließend ihre Methoden aufzurufen (der Hexadezimalwert codiert jeweils die eindeutige Objekt-ID). Sie brauchen also eine Zeile wie diese:

```
m = plz.match(adr)
```

`m` speichert die relevanten Daten in einem Array. `m[0]` ist der gefundene Teilstring selbst:

```
>> m[0]
=> "50670"
```

4 Oder nach einer fünfstelligen Hausnummer, aber die gibt es in Deutschland wohl recht selten.

`m[1]`, `m[2]` und so weiter entsprechen den im vorigen Kapitel beschriebenen geklammerten Teilausdrücken `$1`, `$2` usw.

Interessant sind noch folgende `MatchData`-Methoden (alle Beispiele beziehen sich weiterhin auf die Postleitzahl):

- `m.begin(n)` liefert die Startposition des angegebenen Elements aus dem Match-Array. `m.begin(0)` ist also mit dem Ergebnis der Operation `=~` identisch:

  ```
  >> m.begin(0)
  => 49
  ```

- `m.end(n)` ist entsprechend die Position des ersten Zeichens im String, das nicht mehr zum Match-Bestandteil `n` gehört:

  ```
  >> m.end(0)
  => 54
  ```

- `m.offset(n)` liefert die Ergebnisse von `m.begin(n)` und `m.end(n)` zusammengefasst als Array:

  ```
  >> m.offset(0)
  => [49, 54]
  ```

- `m.pre_match` gibt alle Zeichen des Strings zurück, die vor dem Treffer stehen:

  ```
  >> m.pre_match
  => "O'Reilly Verlag GmbH & Co. KG, Balthasarstr. 81, "
  ```

- `m.post_match` liefert entsprechend die Zeichen hinter dem Match:

  ```
  >> m.post_match
  => " Koeln"
  ```

Die Ruby-Hilfe ri

Wenn Sie Informationen über Ruby-Klassen, -Methoden oder -Eigenschaften suchen, können Sie die Ruby-Konsolendokumentation `ri` verwenden (die Referenz in Anhang A dieses Buches ist für einen ersten Überblick auch nützlich, aber aufgrund des beschränkten Buchumfangs leider nicht einmal ansatzweise vollständig). Die `ri`-Beschreibungen sind englischsprachig. `ri` befindet sich im *bin*-Verzeichnis Ihrer Ruby-Installation, sollte also in jedem beliebigen Arbeitsverzeichnis funktionieren, wenn Sie Installationsanleitung in Kapitel 1 gefolgt sind.

`ri` ist sehr leicht zu bedienen: Geben Sie einfach

```
ri Schluesselwort
```

ein, um Informationen zum gewünschten Thema zu erhalten. An dieser Stelle werden exemplarisch die Schritte beschrieben, um zu den Beschreibungen der Anweisungen des allerersten Beispiels aus diesem Buch zu gelangen:

```
puts "Hallo Welt!"
jetzt = Time.new
puts "Es ist jetzt #{jetzt}."
puts "Wie heissen Sie?"
```

```
name = gets
name.chomp!
puts "Hallo #{name}!"
```

Die Vorgehensweise mag anfangs ein wenig verworren erscheinen, aber mit etwas Übung wird sie klarer.

Die erste Anweisung ist puts. Versuchen Sie also Ihr Glück mit:

```
> ri puts
```

Die Antwort besagt, dass mehrere Klassen Methoden namens puts besitzen:

```
More than one method matched your request. You can refine
your search by asking for information on one of:

     IO#puts, Kernel#puts, StringIO#puts, Zlib::GzipWriter#puts
```

Da puts im Beispiel nicht als Klasse.puts oder Instanz.puts, sondern alleinstehend aufgerufen wurde, dürfte es sich um Kernel#puts handeln. Die spezielle Klasse Kernel enthält die Basismethoden und -eigenschaften des Ruby-Sprachkerns. (Sie könnten in Ihren Skripten sogar Kernel.puts statt einfachem puts schreiben.) Geben Sie daher Folgendes ein, um die Beschreibung der gewünschten Methode zu lesen:

```
> ri Kernel#puts
```

Die Ausgabe sieht so aus:

```
------------------------------------------------------------
Kernel#puts
     puts(obj, ...)    => nil
------------------------------------------------------------
     Equivalent to

         $stdout.puts(obj, ...)
```

Der Kopfbereich zwischen den Linien zeigt das Syntaxschema der Methode an: puts mit beliebig vielen durch Komma getrennten Objekten, kein Rückgabewert (nil ist ein spezieller Wert, der »nichts« bedeutet).

Die Beschreibung selbst ist in diesem Fall allerdings ein wenig mager; man erfährt lediglich, dass die Methode ein Äquivalent zu $stdout.puts ist. $stdout ist ein Synonym für STDOUT. Um noch mehr zu erfahren, müssen Sie nun herausfinden, welcher Klasse diese Variable angehört. Geben Sie dazu diese Anweisung ein:

```
> ruby -e "puts $stdout.class"
```

Die Antwort lautet IO, so dass Sie ausführlichere Hilfe zu puts letztendlich mit Hilfe dieser Eingabe finden:

```
> ri IO#puts
```

Beim nächsten Befehl ist der Fall klar: Time.new nennt Klasse und Methode. Geben Sie also Folgendes ein:

```
> ri Time.new
```

Der Unterschied zwischen # und . in diesen Eingaben ist wichtig: Der Punkt (oder wahlweise ::) beschreibt Klassenmethoden, die ohne Existenz einer Instanz, das heißt eines konkreten Objekts der Klasse, aufgerufen werden. Die Raute (#) ist dagegen für Instanzmethoden zuständig, für die ein Objekt existieren muss. Die Methode new (die eine neue Instanz erzeugt) ist natürlich stets eine Klassenmethode, da die Instanz zum Zeitpunkt ihres Aufrufs eben noch nicht existiert.

Die nächste Methode ist gets; es erscheint logisch, dass sie sich analog zu puts verhält. Geben Sie also auf Verdacht

```
> ri IO#gets
```

ein. Volltreffer! (Allerdings liefert auch Kernel#gets, im Gegensatz zu Kernel#puts, eine ausführliche Antwort.)

Die letzte neue Methode ist chomp!. Hier ist die Klasse noch unbekannt, so dass Sie zuerst einfach

```
> ri chomp!
```

eingeben sollten. Die beiden Alternativen sind Kernel#chomp! und String#chomp!. Da chomp! im Beispielskript zweifellos als Methode eines Objekts (name) aufgerufen wurde, erscheint es logisch, dass String#chomp! die richtige Wahl ist. Wenn Sie

```
> ri String#chomp!
```

eingeben, werden Sie feststellen, dass die Vermutung korrekt ist.

Umgekehrt können Sie ri Klassenname eingeben, um zunächst eine Liste aller Methoden der jeweiligen Klasse (manchmal auch eine allgemeine Beschreibung) zu erhalten. Anschließend können Sie sich nach dem soeben erläuterten Verfahren die Beschreibungen der jeweiligen Methoden anzeigen lassen. Beispiel:

```
> ri String
```

Zusammenfassung

Nach den »geradlinigen« Sprachgrundlagen aus Kapitel 2 haben Sie in diesem Kapitel auch den vollständig objektorientierten Ansatz der Sprache Ruby kennengelernt, zumindest den passiven, das heißt den Einsatz vorhandener Klassen.

Im ersten Hauptabschnitt drehte sich alles um die Ein- und Ausgabe. Die grundlegenden Methoden wurden zunächst am Beispiel der Konsole vorgestellt. Wichtig ist in diesem Zusammenhang, dass Sie die drei Standard-I/O-Kanäle STDIN, STDOUT und STDERR kennen. Auf diese können Sie Eingabemethoden wie gets und read beziehungsweise Ausgabemethoden wie print, puts und printf anwenden. Die Methode printf besitzt eine praktische Syntax zur String-Formatierung mehrerer Ausdrücke; die Methode sprintf stellt dieselbe Syntax für String-Ausdrücke zur Verfügung.

Nach der ausführlichen Erläuterung der Konsolen-I/O haben Sie festgestellt, wie leicht sich das Erlernte auf Dateien übertragen lässt. Dabei kommt nur die spezifische Syntax von `File.new` beziehungsweise `File.open` hinzu. Achten Sie jeweils auf den richtigen Modus (Lesen, Schreiben, Anhängen usw.) sowie darauf, dass Sie Dateien nach Gebrauch mittels `datei.close` wieder schließen sollten.

Zum Durchsuchen von Verzeichnissen gibt es eine eigene Klasse namens `Dir`. Ihre Methode `read` liest jeweils den nächsten Verzeichniseintrag. Mit Hilfe von statischen Methoden wie `File.file?` oder `File.directory?` können Sie ermitteln, ob es sich bei einem Eintrag um eine gewöhnliche Datei oder ein Verzeichnis handelt.

Ein weiteres wichtiges Element einer modernen Programmiersprache ist der Umgang mit Datum und Uhrzeit. Ruby stellt dafür die Klasse `Time` zur Verfügung. Methoden wie `month` oder `hour` stellen die einzelnen Bestandteile eines Datumsobjekts zur Verfügung, während `strftime` eine Reihe von Platzhaltern für Formatstrings bereitstellt.

Abgerundet wurde dieses Kapitel durch die Klassen `Rational` für die Bruchrechnung sowie `Regexp` und `MatchData` für den objektorientierten Zugang zu regulären Ausdrücken. Und zum Schluss wurde noch kurz auf die Verwendung der Ruby-Hilfe `ri` eingegangen, so dass Sie nun leicht weitere eingebaute Klassen und Methoden erkunden können.

In späteren Kapiteln werden Sie eine Reihe weiterer Klassen aus dem Sprachkern sowie aus den mit Ruby gelieferten Bibliotheken kennenlernen. Im nächsten Kapitel erfahren Sie außerdem ausführlich, wie Sie Ihre eigenen Klassen entwickeln können, und sogar, wie sich die Ruby-Standardklassen durch eigene Methoden erweitern lassen.

In diesem Kapitel:

- Objektorientierte Programmierung – Eine praktische Einführung
- Klassen entwerfen und implementieren
- Weitere objektorientierte Konstrukte

KAPITEL 4

Eigene Klassen und Objekte erstellen

Die Ordnung der Ideen muss fortschreiten nach der Ordnung der Gegenstände.
– Giambattista Vico

Nachdem Sie im vorigen Kapitel mit diversen eingebauten Klassen gearbeitet haben, werden Sie nun erfahren, wie leicht es ist, Ihre eigenen Klassen zu schreiben. Dies ist überaus nützlich, um übersichtlichere und weniger fehlerträchtige Programme zu schreiben. Zudem sind Klassen leichter für neue Projekte wiederverwendbar als andere Arten von Codeblöcken.

Objektorientierte Programmierung – Eine praktische Einführung

Da Sie gerade dieses Buch lesen, in dem eine der wichtigsten objektorientierten Programmiersprachen beschrieben wird, ist davon auszugehen, dass Sie den Begriff schon einmal gehört haben. Wahrscheinlich haben Sie auch eine ungefähre Vorstellung davon, was er bedeutet. Hier erfahren Sie Genaueres.

Der Ansatz der objektorientierten Programmierung (auch OOP abgekürzt) wurde bereits in den frühen 1970er Jahren entwickelt, und zwar als Reaktion auf die so genannte *Softwarekrise* der späten 60er: Zum ersten Mal wurde Softwareentwicklung damals teurer als Hardwarebeschaffung – und das, obwohl die damaligen Rechner nicht viel weniger kosteten als ein Haus. Wie viele Errungenschaften der modernen IT wurde auch die OOP vor allem im Forschungszentrum Xerox PARC

(Palo Alto Research Center) ersonnen. Das Ergebnis der dortigen Arbeiten war *Smalltalk*, die erste konsequent objektorientierte Sprache. Interessanterweise hat sie viele Ähnlichkeiten mit Ruby, während andere Sprachen wie C++ oder Java mehr Kompromisse mit Nicht-OOP-Techniken eingehen.

Bei der OOP geht es kurz gesagt darum, Datenstrukturen – das heißt Einzelvariablen, Arrays und so weiter – zu einer Einheit mit den Funktionen zu verbinden, die der Verarbeitung dieser Strukturen dienen. Dieses Konzept heißt *Kapselung*. Der Code wird dadurch übersichtlicher und leichter zu warten, weil die Datenstrukturen nicht mehr über das ganze Programm verstreut manipuliert werden können.

Am leichtesten lässt sich die objektorientierte Programmierung im direkten Vergleich mit einem nicht objektorientierten Ansatz erläutern, der in Ruby zwar nicht empfehlenswert, aber doch möglich ist. Die Aufgabenstellung: Ein Güterzug[1] soll simuliert werden. Die Lokomotiven (1-2 Stück) können je nach Antriebstechnik (elektrisch, Diesel, Dampf) eine bestimmte Anzahl von Waggons ziehen; diese können an- oder abgehängt werden, sofern die Loks mit der betreffenden Anzahl zurechtkommen.

In Beispiel 4-1 sehen Sie zunächst den klassischen Ansatz: Alle Informationen über den Zug werden in globalen Variablen gespeichert. Die Werte dieser Variablen werden von beliebigen, über das Skript verstreuten Methoden modifiziert. Eine solches Programmierverfahren wird als *imperativ* bezeichnet. Probieren Sie diese Variante zunächst aus; die Erläuterungen folgen.

Beispiel 4-1: Die Güterzug-Anwendung ohne Objektorientierung, zug_imp.rb

```
 1  # Konstanten fuer die maximale Anzahl an Waggons
 2  OHNE = 0
 3  DAMPF = 30
 4  STROM = 40
 5  DIESEL = 50

 6  # Datenstruktur -- globale Variablen
 7  $loks = [STROM, DIESEL]
 8  $waggons = 0

 9  # (Globale) Methoden

10  # Eine bestimmte Anzahl Waggons anhaengen
11  def anhaengen(waggons)
12    if $waggons + waggons <= $loks[0] + $loks[1]
13      $waggons += waggons
14      # Erfolg melden
15      true
16    else
17      # Misserfolg melden
18      false
```

1 Zumindest im Wortsinn ein kleiner Vorgeschmack auf »Ruby on Rails« (Kapitel 7) … ;-)

Beispiel 4-1: Die Güterzug-Anwendung ohne Objektorientierung, zug_imp.rb (Fortsetzung)

```
19     end
20   end

21   # Eine bestimmte Anzahl Waggons abhaengen
22   def abhaengen(waggons)
23     if $waggons - waggons >= 0
24       $waggons -= waggons
25       # Erfolg melden
26       true
27     else
28       # Misserfolg melden
29       false
30     end
31   end

32   # Eine der Loks ersetzen
33   def lok_aendern(loknr, loktyp)
34     $loks[loknr] = loktyp
35   end

36   # Differenz Waggonkapazitaet/-anzahl ermitteln
37   def waggontest
38     return $waggons - ($loks[0] + $loks[1])
39   end

40   # Infostring ueber eine der Loks
41   def lokinfo(loknr)
42     case $loks[loknr]
43     when OHNE
44       "keine"
45     when DAMPF
46       "Dampflok"
47     when STROM
48       "Elektrolok"
49     when DIESEL
50       "Diesellok"
51     end
52   end

53   # Informationen ueber den Zug ausgeben
54   def info
55     printf "Lokomotive 1:        %s\n", lokinfo(0)
56     printf "Lokomotive 2:        %s\n", lokinfo(1)
57     printf "Waggonkapazitaet:    %d\n", $loks[0] + $loks[1]
58     printf "Aktuelle Waggonzahl: %d\n", $waggons
59   end

60   # Globaler Code -- einen Zug testen

61   # Informationen ueber den Grundzustand
62   info
63   puts
```

Beispiel 4-1: Die Güterzug-Anwendung ohne Objektorientierung, zug_imp.rb (Fortsetzung)

```
64   # 50 Waggons anhaengen
65   anhaengen(50)
66   info
67   puts

68   # Versuchen, weitere 50 Waggons anzuhaengen
69   if anhaengen(50)
70     puts "50 weitere Waggons angehaengt."
71   else
72     puts "Konnte keine 50 Waggons mehr anhaengen."
73   end
74   puts

75   # Lok 2 entfernen
76   lok_aendern(1, OHNE)
77   info
78   puts

79   # Waggonanzahl testen
80   printf "Zug kann nicht fahren. %d Waggons zu viel.\n", waggontest
```

Wenn Sie das Skript ausführen, erhalten Sie folgende Ausgabe:

```
> ruby zug_imp.rb
Lokomotive 1:        Elektrolok
Lokomotive 2:        Diesellok
Waggonkapazitaet:    90
Aktuelle Waggonzahl: 0

Lokomotive 1:        Elektrolok
Lokomotive 2:        Diesellok
Waggonkapazitaet:    90
Aktuelle Waggonzahl: 50

Konnte keine 50 Waggons mehr anhaengen.

Lokomotive 1:        Elektrolok
Lokomotive 2:        keine
Waggonkapazitaet:    40
Aktuelle Waggonzahl: 50

Zug kann nicht fahren. 10 Waggons zu viel.
```

Die einzelnen Anweisungen im Skript dürften weitgehend bekannt sein, so dass die Beschreibung hier recht kurz ausfallen kann:

- Zeile 2-5: In vier Konstanten wird die Anzahl der Waggons gespeichert, die der jeweilige Lokomotiventyp ziehen kann. Wenn eine der beiden Lokomotiven nicht dabei ist, hat diese Position den Typ `OHNE` mit 0 möglichen Waggons.
- Zeile 7-8: Die beiden globalen Variablen werden definiert und erhalten ihre Anfangswerte. `$loks` enthält eine Elektro- und eine Diesellok, die insgesamt 90

Waggons schaffen. `$waggons`, die Anzahl der bereits am Zug hängenden Waggons, ist anfangs 0.

- Zeile 11-20: Die Methode `anhaengen` versucht, die globale Variable `$waggons` um die als Parameter übergebene Anzahl `waggons` zu erhöhen. Dazu wird die gesamte Anzahl mit der Gesamtkapazität beider Lokomotiven verglichen. Wenn der Wert im Rahmen liegt, wird `$waggons` erhöht, und die Methode gibt `true` zurück. Andernfalls geschieht nichts, und der Rückgabewert ist `false`.
- Zeile 22-31: `abhaengen` funktioniert genau wie `anhaengen`, nur dass `$waggons` diesmal vermindert wird, sofern eine entsprechende Mindestanzahl an Waggons vorhanden ist.
- Zeile 33-35: Die Methode `lok_aendern` erwartet eine Lokomotivnummer, das heißt 0 oder 1, sowie einen Loktyp (eine der Konstanten). Das entsprechende Element von `$loks` wird auf den angegebenen Wert gesetzt.

  Anregung: Diese Methode implementiert keinerlei Fehlerkontrolle. Wie könnte man sie so erweitern, dass sie als Loknummer wirklich nur 0 oder 1 und als Typ nur eine der Waggonanzahl-Konstanten akzeptiert? Was sollte die Methode zurückgeben, wenn die übergebenen Werte ungültig sind?
- Zeile 37-39: `waggontest` liefert einfach die Differenz zwischen der aktuellen Waggonanzahl und der Kapazität der beiden Lokomotiven.
- Zeile 41-52: In der Methode `lokinfo` wird die Kapazität der angegebenen Lok per `case`/`when`-Fallentscheidung untersucht. Diese liefert eine String-Entsprechung des Lokomotiventyps.
- Zeile 54-59: Die Methode `info` gibt alle relevanten Informationen über den Güterzug aus. Sie ruft zunächst für beide Lokomotiven `lokinfo` auf; anschließend werden die Waggonkapazität und die aktuelle Waggonzahl des Zugs ausgegeben.
- Zeile 62: Hier beginnt der globale, das heißt nicht mehr in Methodendefinitionen stehende Code (abgesehen von den Konstantendefinitionen zu Beginn des Skripts). Die erste Anweisung, die tatsächlich ausgeführt wird, ist dieser Aufruf von `info`, um den Grundzustand des Zugs auszugeben.
- Zeile 65-66: 50 Waggons werden angehängt. Danach wird erneut `info` aufgerufen, um den aktuellen Zustand zu ermitteln.
- Zeile 69-74: Ein erneuter Aufruf von `anhaengen` macht sich dessen Rückgabewert zunutze, um zu überprüfen, ob sich weitere 50 Waggons hinzufügen lassen. Die `if`-Abfrage sorgt für die entsprechende Erfolgs- oder Misserfolgsmeldung.
- Zeile 76: Die zweite Lokomotive, also Nummer 1, wird »entfernt«, indem ihr Typ auf die Konstante `OHNE` gesetzt wird.
- Zeile 80: Ein Aufruf von `waggontest` bringt an den Tag, dass für die verbliebene Lokomotive zu viele Waggons vorhanden sind, so dass der Zug nicht mehr fahren kann.

Wie Sie sehen, können die in den globalen Variablen `$loks` und `$waggons` gespeicherten Eigenschaften des Zugs an jeder Stelle des Skripts modifiziert werden. Genau darin besteht das Problem dieses Ansatzes: Das Skript wird unübersichtlich, und bei Fehlern lässt sich kaum noch erkennen, an welcher Stelle sie auftraten. Viel besser wäre ein Güterzug, der sich intern um seine »eigenen Angelegenheiten« kümmert und Zugriffe auf seine Datenstrukturen nur über bestimmte Methoden erlaubt.

Genau diesen Anforderungen wird die objektorientierte Lösung gerecht: Lokomotive und Zug werden als Klassen definiert, die ihre Daten selbst speichern und deren Änderung mit Hilfe von Objektmethoden ermöglichen. Die Zug-Klasse enthält zwei Lok-Objekte, während der globale Code wiederum ein Zug-Objekt erzeugt. Geben Sie zunächst Beispiel 4-2 ein und führen Sie es aus. Die Ausgabe sollte exakt der Nicht-OO-Variante entsprechen. Die entsprechenden Erläuterungen folgen nach dem Listing.

Beispiel 4-2: Der objektorientierte Güterzug, zug_oo.rb

```
 1  # Datenstrukturen/Methoden als Klassen
 2  class Lok
 3    # Konstanten fuer die maximale Anzahl von Waggons
 4    OHNE = 0
 5    DAMPF = 30
 6    STROM = 40
 7    DIESEL = 50

 8    # Grundzustand
 9    def initialize(typ=OHNE)
10      @typ = typ
11    end

12    # Typinformation als String
13    def get_typ
14      case @typ
15      when OHNE
16        "keine"
17      when DAMPF
18        "Dampflok"
19      when STROM
20        "Elektrolok"
21      when DIESEL
22        "Diesellok"
23      end
24    end

25    # Waggonkapazitaet
26    def get_kapazitaet
27      @typ
28    end

29    # Stringdarstellung
30    def to_s
```

Beispiel 4-2: Der objektorientierte Güterzug, zug_oo.rb (Fortsetzung)

```
31      "#{get_typ} -- Kapazitaet #{@typ} Waggons"
32    end
33  end

34  class Gueterzug
35    # Grundzustand
36    def initialize(lok0=Lok::OHNE, lok1=Lok::OHNE, waggons=0)
37      @loks = Array.new
38      @loks[0] = Lok.new(lok0)
39      @loks[1] = Lok.new(lok1)
40      @waggons = waggons
41    end

42    # Eine bestimmte Anzahl Waggons anhaengen
43    def anhaengen(waggons)
44      if @waggons + waggons <= @loks[0].get_kapazitaet +
                                 @loks[1].get_kapazitaet
45        @waggons += waggons
46        # Erfolg melden
47        true
48      else
49        # Misserfolg melden
50        false
51      end
52    end

53    # Eine bestimmte Anzahl Waggons abhaengen
54    def abhaengen(waggons)
55      if @waggons - waggons >= 0
56        @waggons -= waggons
57        # Erfolg melden
58        true
59      else
60        # Misserfolg melden
61        false
62      end
63    end

64    # Eine der Loks ersetzen
65    def lok_aendern(loknr, loktyp)
66      @loks[loknr] = Lok.new(loktyp)
67    end

68    # Differenz Waggonkapazitaet/-anzahl ermitteln
69    def waggontest
70      return @waggons - (@loks[0].get_kapazitaet +
                          @loks[1].get_kapazitaet)
71    end

72    # Infostring ueber eine der Loks
73    def lokinfo(loknr)
74      @loks[loknr].get_typ
75    end
```

Beispiel 4-2: Der objektorientierte Güterzug, zug_oo.rb (Fortsetzung)

```
 76     # Informationen ueber den Zug ausgeben
 77     def info
 78       printf "Lokomotive 1:        %s\n", lokinfo(0)
 79       printf "Lokomotive 2:        %s\n", lokinfo(1)
 80       printf "Waggonkapazitaet:    %d\n",
                  @loks[0].get_kapazitaet + @loks[1].get_kapazitaet
 81       printf "Aktuelle Waggonzahl: %d\n", @waggons
 82     end

 83   end

 84   # Globaler Code -- einen Zug testen

 85   # Neuen Zug erzeugen
 86   zug = Gueterzug.new(Lok::STROM, Lok::DIESEL)

 87   # Informationen ueber den Grundzustand
 88   zug.info
 89   puts

 90   # 50 Waggons anhaengen
 91   zug.anhaengen(50)
 92   zug.info
 93   puts

 94   # Versuchen, weitere 50 Waggons anzuhaengen
 95   if zug.anhaengen(50)
 96     puts "50 weitere Waggons angehaengt."
 97   else
 98     puts "Konnte keine 50 Waggons mehr anhaengen."
 99   end
100   puts

101   # Lok 2 entfernen
102   zug.lok_aendern(1, Lok::OHNE)
103   zug.info
104   puts

105   # Waggonanzahl testen
106   printf "Zug kann nicht fahren. %d Waggons zu viel.\n", zug.waggontest
```

In den einzelnen Teilen des objektorientierten Güterzug-Skripts geschieht Folgendes:

- Zeile 2-33: Definition der Klasse `Lok`, deren Objekte Informationen über eine einzelne Lokomotive speichern können.
- Zeile 4-7: Da die Typ- und Kapazitätskonstanten nur Lokomotiven betreffen, ist es sinnvoll, sie nun innerhalb der Klasse `Lok` zu definieren. Außerhalb dieser Klasse stehen die Konstanten als `Lok::KONSTANTE` zur Verfügung, zum Beispiel

`Lok::DAMPF` – ähnlich wie die im vorigen Kapitel vorgestellten Konstanten der Klasse `IO`, etwa `IO::SEEK_SET`.

- Zeile 9-11: Eine Methode namens `initialize` heißt *Konstruktor* und wird automatisch aufgerufen, wenn Sie mit `new` eine Instanz, das heißt ein neues Objekt der Klasse erzeugen. Der Konstruktor wird vor allem verwendet, um der Datenstruktur eines Objekts sinnvolle Anfangswerte zuzuweisen. Bei der Lokomotive ist dies nur der Typ, der beim Aufruf übergeben werden kann oder ansonsten auf `Lok::OHNE` gesetzt wird. Gespeichert wird der Wert in der Instanzvariablen `@typ`.
- Zeile 13-24: Die `Lok`-Methode `get_typ` untersucht per `case`/`when`-Fallentscheidung den Wert von `@typ` und gibt einen entsprechenden String mit dem Lokomotiventyp zurück.
- Zeile 26-28: `get_kapazitaet` gibt Auskunft darüber, wie viele Waggons die Lokomotive ziehen kann. Dazu genügt es, den ohnehin numerischen Wert von `@typ` zurückzuliefern.
- Zeile 30-32: Zur Fehlersuche und zur bequemen Verarbeitung ist es praktisch, wenn eine Klasse eine Methode namens `to_s` besitzt, die Informationen über alle wichtigen Daten einer Instanz als String zurückgibt. Das Praktische an einer Methode dieses Namens ist, dass sie automatisch aufgerufen wird, wenn die Instanz selbst im String-Kontext verwendet wird. Ein anschauliches Beispiel erhalten Sie, wenn Sie im aktuellen Verzeichnis `irb` starten, darin zunächst mit Hilfe der Anweisung `require` das Skript `zug_oo.rb` importieren und anschließend per `puts` die Eigenschaften eines neuen `Lok`-Objekts ausgeben:

```
>> require "zug_oo.rb"
[Ausgabe siehe oben]
>> puts Lok.new(Lok::DIESEL)
=> Diesellok -- Kapazitaet 50 Waggons
```

Eine Methode zum nachträglichen *Ändern* des Loktyps gibt es nicht, weil auch in der Realität beispielsweise eine Dampflokomotive nicht zu einer E-Lok mutieren kann. Auch bei der objektorientierten Fassung können Sie lediglich die an den Zug gehängten Waggons wechseln. Eines der Ziele der Objektorientierung ist es schließlich, Elemente und Sachverhalte aus der Realität möglichst genau abzubilden.

Die eigentlich auch etwas realitätsfremde Entscheidung, ein `Lok`-Objekt mit dem Typ `Lok::OHNE` zuzulassen – also gewissermaßen eine »Lokomotive, die nicht da ist« –, folgt praktischen Überlegungen: Der Fall, dass eine der Loks eines Güterzugs nicht vorhanden ist, kann auf diese Weise genauso behandelt werden wie jeder Loktyp. Andernfalls müsste die Klasse `Gueterzug` diese beiden Fälle selbst unterscheiden, was umständlicher wäre.

- Zeile 34-83: Die Klasse `Gueterzug` ist die Vorlage für Güterzüge mit ihrer Datenstruktur (Lokomotiven und Waggons) sowie diversen Änderungs- und Auskunftsmethoden.
- Zeile 36-41: Der Konstruktor von `Gueterzug` hat etwas mehr zu tun als der von `Lok`, weil die zu verwaltende Datenstruktur umfangreicher ist. Das Array `@loks` enthält diesmal keine Rohdaten wie bei der nicht objektorientierten Version, sondern zwei `Lok`-Instanzen für die beiden Lokomotiven. `@waggons` speichert dagegen weiterhin einfach die aktuelle Lokomotivenanzahl. Beim Aufruf des Konstruktors können Sie die Typen der beiden Lokomotiven sowie die Anfangs-Waggonzahl (von rechts an) weglassen. Standardmäßig besitzt der Zug keine Loks (Typ `Lok::OHNE`) und keine Waggons.
- Zeile 43-52: Die Methode `anhaengen` entspricht fast genau ihrem imperativen Pendant aus dem vorigen Beispiel. Der einzige Unterschied befindet sich in Zeile 44: Um die maximale Waggonanzahl des Zugs zu ermitteln, wird die Methode `get_kapazitaet` der beiden `Lok`-Instanzen aufgerufen; die Ergebnisse werden dann wie gehabt addiert.
- Zeile 54-63: Die Methode `abhaengen` besitzt sogar keinen einzigen Unterschied zur imperativen Variante, da der Grenzwert für ein erfolgreiches Entfernen von Waggons hier wie dort 0 ist.
- Zeile 65-67: `lok_aendern` funktioniert in der objektorientierten Fassung etwas anders: Da der Typ einer bestehenden Lok aus den oben erwähnten Gründen nicht nachträglich geändert werden kann, wird mit Hilfe der übergebenen Parameter eine neue `Lok`-Instanz erzeugt.
- Zeile 69-71: Die Änderung in `waggontest` erfolgt analog zu derjenigen in `anhaengen` – die Zugkraft einer Lokomotive wird nun mit Hilfe ihrer Methode `get_kapazitaet` ermittelt.
- Zeile 73-75: Die Methode `lokinfo` ist nur noch ganz kurz, weil ihre eigentliche Aufgabe in der Klasse `Lok` erledigt wird (Zeile 13-24). Deshalb genügt es, an dieser Stelle die Methode `get_typ` der betroffenen Lok aufzurufen.
- Zeile 77-82: In der Methode `info` haben sich nur zwei Zeilen geändert: In Zeile 80 wird, wie bereits beschrieben, mit `get_kapazitaet` gearbeitet. In Zeile 81 wird statt der früheren globalen Variablen nun der Wert der Instanzvariablen `@waggons` ausgegeben.
- Zeile 86: Hier wird eine Instanz der Klasse `Gueterzug` erzeugt; die Referenzvariable `zug` verweist darauf. Es werden zwei der drei möglichen Parameter übergeben, nämlich die Loktypen als Konstanten der Klasse `Lok`. Der dritte Parameter, die Anfangswaggonzahl, wird weggelassen, so dass er den Standardwert 0 erhält (siehe Zeile 36).
- Zeile 88-106: Dieser Teil des globalen Codes entspricht im Ablauf genau den Zeilen 62-80 der imperativen Lösung. Der Unterschied, der sich durch alle

Codezeilen zieht, ist die Syntax der Methodenaufrufe: Wo bei der imperativen Fassung ein einfaches `info` oder `anhaengen(50)` steht, wird in der OO-Version spezifiziert, um welchen konkreten Zug es geht, indem den Methodenaufrufen die Referenzvariable `zug` vorangestellt wird: `zug.info` beziehungsweise `zug.anhaengen(50)`.

Anregung: Auf den ersten Blick wirkt die objektorientierte Fassung umständlicher – schon allein, weil sie mit 106 Zeilen deutlich umfangreicher ist als die 80 Zeilen des imperativen Skripts. Machen Sie sich einmal Gedanken darüber, wie man beide Skripten umschreiben müsste, um die Daten von *fünf* verschiedenen Zügen zu verwalten. Glauben Sie, dass die imperative Schreibweise dann immer noch schlanker wäre?

Klassen entwerfen und implementieren

Nach dem Einführungsbeispiel sind Sie nun bereit für einen systematischen Überblick über den Entwurf und die Programmierung von Klassen und das Arbeiten mit ihren Instanzen und Methoden.

Jede Ruby-Klasse wird in einem Codeblock mit folgender Syntax definiert:

```
class Klassenname
  # Konstanten, Methoden usw.
end
```

Zwischen `class` und `end` werden dann sämtliche Elemente der Klasse definiert, die Sie im Folgenden kennenlernen – Konstruktor, Methoden und so weiter. Interessanterweise besteht die kürzeste gültige Klasse aber aus genau diesen beiden Zeilen, wenngleich sie dann keinen besonderen Nutzen hat. Probieren Sie es trotzdem in `irb` aus:

```
>> class Test
>> end
=> nil
>> t = Test.new
=> #<Test:0x2b1ac7c>
```

Sie haben damit erfolgreich eine Klasse definiert und eine Instanz davon erzeugt. Da diese keinerlei Daten enthält, zeigt `irb` die Objekt-ID an. Das ist eine eindeutige Nummer, unter der sich Ruby selbst jedes einzelne Objekt merkt; sie wird weiter unten noch näher besprochen.

Der objektorientierte Entwurf

Bei jedem Softwareprojekt, das über ein paar Zeilen hinausgeht, sollten Sie nicht einfach anfangen, Code einzutippen, sondern zunächst einen Entwurf anfertigen. Der hier besprochene *objektorientierte Entwurf* (auf Englisch *Object Oriented*

Design oder kurz OOD) gehört zum Handwerkszeug der modernen Softwareentwicklung. Dazu müssen Sie auch ein wenig mehr über den theoretischen Hintergrund und damit über das Warum der Objektorientierung wissen.

Zunächst wird es Ihnen helfen, nach dem intuitiven Zugang im vorigen Kapitel genaue Definitionen der »Mitspieler« objektorientierter Software zu erhalten:

- Eine *Klasse* (englisch class) ist zwar das wichtigste Element eines OO-Programms, aber selbst noch kein Objekt. Es handelt sich vielmehr um einen *Datentyp* – eine Art Vorlage, die die *Eigenschaften* (properties) und das *Verhalten* (behavior) beliebig vieler konkreter Objekte dieses Typs festlegt. Die Eigenschaften werden als *Instanzvariablen* (instance variables) gespeichert, während das Verhalten in so genannten *Instanzmethoden* (instance methods) festgelegt wird. Diese Zusammenfassung von Datenstruktur und Programmcode zu dessen Manipulation heißt *Kapselung* (encapsulation). In vollwertigen objektorientierten Sprachen hat ein Entwickler von außen in der Regel nur Zugriff auf die Methoden, aber nicht auf die Eigenschaften. Die Methoden bilden somit die offiziellen Schnittstellen für den Zugriff auf die Daten, so dass die Zulässigkeit jeder Änderung kontrolliert werden kann. Das macht objektorientierte Programme weniger fehleranfällig als imperative.
- Eine *Instanz* (instance) – allgemeiner auch als *Objekt* (object) bezeichnet – ist ein konkretes Element, dessen Eigenschaften und Verhalten durch eine bestimmte Klasse festgelegt werden. Die Instanz wird erzeugt, indem mittels `Klasse.new` der *Konstruktor* (constructor) einer Klasse aufgerufen wird – eine Art spezieller Methode, deren Rückgabewert eben die neue Instanz ist. Sobald die Instanz existiert, können Sie ihre Methoden aufrufen, um auf ihr Verhalten zuzugreifen. Aufrufe von Instanzmethoden werden auch als *Nachrichten* (messages) bezeichnet, die verschiedene Objekte einander senden. Diese Interpretation der Objektorientierung unterstreicht die Autonomie der verschiedenen Objekte: Sie führen bestimmte »Handlungen« durch, sobald sie eine entsprechende Nachricht erhalten.
- Eine *Methode* (method) ist ein benanntes Stück Programmcode. Sie wird zwar in einer Klasse definiert, kann aber in der Regel nur für eine konkrete Instanz der Klasse aufgerufen werden (übliche Syntax, nicht nur in Ruby: `Instanz.Methodenname`). Eine Ausnahme sind *Klassenmethoden* (class methods), die für allgemeine und nicht auf ein bestimmtes Objekt bezogene Aufgaben verwendet werden – im vorigen Kapitel haben Sie beispielsweise die Instanzmethode `File-Instanz.gets` (eine Zeile aus einer konkreten Datei lesen) im Gegensatz zur Klassenmethode `File.exists?` (prüfen, ob unter einem bestimmten Namen überhaupt eine Datei existiert) kennengelernt.

Eine zentrale Rolle beim Objektorientierten Entwurf spielen die standardisierten Diagramme der *UML* (Unified Modeling Language). Für die Entwicklung komplexer Systeme gibt es eine Reihe verschiedener Diagrammtypen, die nicht nur den

geplanten Programmcode, sondern beispielsweise auch Geschäftsabläufe, beteiligte Personen und Geräte modellieren. Die wichtigste (und hier ausschließlich besprochene) Diagrammsorte sind die *Klassendiagramme*, die die Bestandteile von Klassen und die Beziehungen zwischen ihnen illustrieren.

In Abbildung 4-1 sehen Sie ein vereinfachtes Klassendiagramm für die Klasse `Lok` aus dem Einführungsbeispiel. Ganz oben steht der Name der Klasse (`Lok`), unter dem ersten Trennstrich finden Sie die Datenstruktur – die Liste der Instanzvariablen – und nach einer weiteren Trennlinie folgen die Methoden.

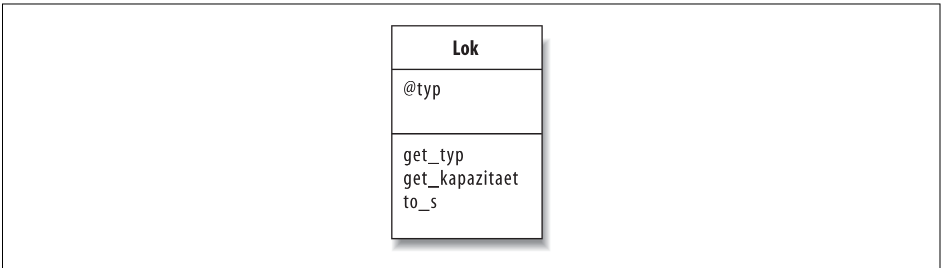

Abbildung 4-1: Vereinfachtes UML-Klassendiagramm der Klasse Lok

Vereinfacht ist dieses Klassendiagramm, weil es keine Angaben zum Datentyp der Instanzvariablen sowie zu Parametern und Rückgabetypen der Methoden macht.

Die verschiedenen Arten von Beziehungen zwischen Klassen werden durch Linien dargestellt, gegebenenfalls mit Pfeilen für eine bestimmte Richtung. Am wichtigsten sind zwei Arten von Beziehungen:

- Wenn die Datenstruktur einer Klasse eine Instanz einer anderen Klasse enthält, wird das als *HAS-A-Beziehung* bezeichnet. Zum Beispiel: Ein Güterzug *hat eine* `Lok` (genauer gesagt zwei). Bei der voll objektorientierten Sprache Ruby stellen *alle* Instanzvariablen HAS-A-Beziehungen dar, weil es keine Nicht-OO-Datentypen (so genannte einfache oder primitive Typen wie etwa in Java) gibt. In Abbildung 4-2 sehen Sie die HAS-A-Beziehung zwischen den Klassen `Gueterzug` und `Lok`. Die Ziffern am Pfeil besagen, dass ein Güterzug zwei Loks besitzt.
- Wenn eine Klasse von einer allgemeineren Klasse abgeleitet wird, spricht man von *Vererbung* (inheritance). Im Klassendiagramm ist das eine *IS-A-Beziehung*, weil eine Instanz der abgeleiteten Klasse auch eine spezielle Instanz der Elternklasse ist. Beispiele: Jedes Auto *ist ein* (spezielles) Fahrzeug, und jede Fließkommazahl (`Float`) *ist eine* Zahl (`Numeric`). Ruby kennt sogar die eingebaute Methode `is_a?`, die diese Beziehung überprüft – siehe den Abschnitt über Introspektion gegen Ende dieses Kapitels. In Abbildung 4-2 wird die Vererbungsbeziehung der Klassen `Rechteck` und `Quadrat` aus dem Einführungsbeispiel des vorigen Kapitels gezeigt.

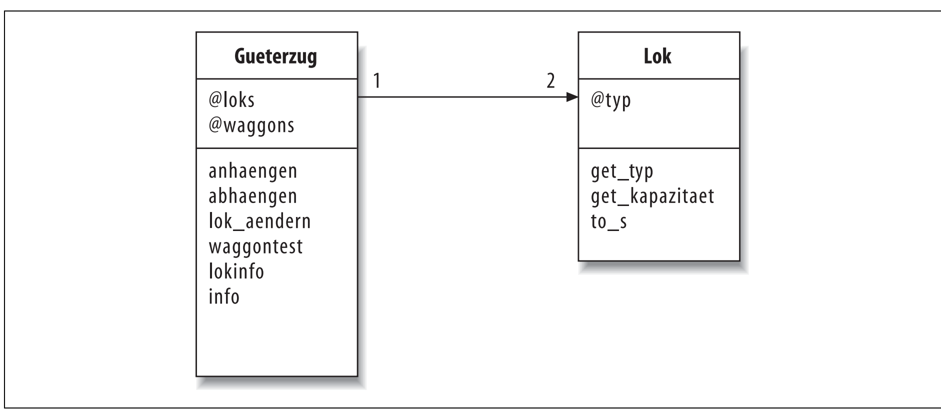

Abbildung 4-2: Vereinfachtes UML-Klassendiagramm, das die HAS-A-Beziehung zwischen Gueterzug und Lok darstellt

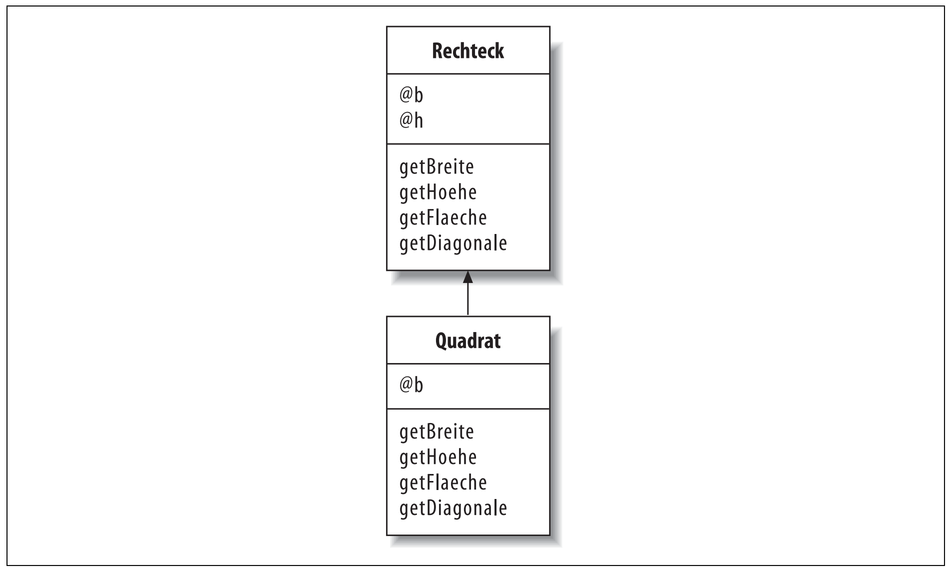

Abbildung 4-3: Vereinfachtes UML-Klassendiagramm, das die IS-A-Beziehung (Vererbung) zwischen Rechteck und Quadrat darstellt

Diese wenigen Beispiele reichen natürlich nicht als Einführung in das reichhaltige Diagrammvokabular der UML, und noch nicht einmal in Klassendiagramme. Dennoch sollten Sie sich selbst einen Gefallen tun und Ihre Programme mit Hilfe dieser oder ähnlicher Visualisierungen planen. Es gibt sehr nützliche, teilweise sogar kostenlose Software dafür; in Anhang B finden Sie einige entsprechende Links sowie OOP- und UML-Buchempfehlungen. Manchmal ist es aber nützlicher, vom Rech-

ner aufzustehen und die Entwürfe mit einem Stift auf ein großes leeres Blatt zu zeichnen. Auf diese Weise kommen Sie auch nicht so leicht in Versuchung, vor dem Entwurf mit der Implementierung zu beginnen ;-).

So, nun aber fürs Erste genug codefreie Theorie. Die nachfolgenden Abschnitte beschreiben die Ruby-Syntax der diversen Bestandteile von Klassen und Objekten.

Der Konstruktor

Eines der wichtigsten Elemente einer Klasse ist der *Konstruktor*. Sobald mit `Klassenname.new` eine neue Instanz einer Klasse erzeugt wird, wird der Konstruktor automatisch aufgerufen, so dass er sinnvollerweise verwendet wird, um der Datenstruktur der Instanz ihre Anfangswerte zuzuweisen. Es handelt sich dabei um eine spezielle Methode mit dem vorgegebenen, zu dieser Aufgabe passenden Namen `initialize`.

Die Datenstruktur einer Instanz wird in so genannten *Instanzvariablen* gespeichert – bekannte Alternativnamen sind *Eigenschaften* oder *Attribute*. Diese speziellen Variablen werden durch ein einleitendes `@`-Zeichen gekennzeichnet.[2] Sie werden in aller Regel innerhalb des Konstruktors ins Leben gerufen und stehen während der gesamten Lebensdauer einer Instanz in allen ihren Methoden zur Verfügung.

Es gibt noch eine weitere besondere Variablenart, die Sie in Klassen einsetzen können: *Klassenvariablen*. Diese beginnen mit zwei `@`-Zeichen (Beispiel: `@@classvar`) und stehen in jeder Methode zur Verfügung – auch in den weiter unten besprochenen Klassenmethoden. Sie nehmen innerhalb einer Klasse die Position von globalen Variablen ein.

In diesem Buch kommt nur ein Beispiel vor, das Klassenvariablen verwendet; auch in der Praxis werden sie eher selten eingesetzt. Da eine Klasse als Vorlage ein statisches Gebilde ist, sind Konstanten (wie etwa die Loktypen in der Klasse `Lok`) meist nützlicher, zumal Sie auf diese auch von außen zugreifen können. Veränderliche Daten gehören dagegen eher zu einer konkreten Instanz und werden demzufolge eben in Instanzvariablen gespeichert.

Die allgemeine Syntax für eine Klasse mit Konstruktor sieht so aus:

```
class Klassenname
  def initialize(parameter1, parameter2, ...)
    @instanzvariable1 = parameter1
    # ...
  end
end
```

2 Viele andere Programmiersprachen verwenden dafür das umständlichere Konstrukt `this.Variablenname`.

Eine Instanz einer solchen Klasse wird mit Hilfe des folgenden Aufrufs erzeugt (den Sie bereits im vorigen Kapitel für Ruby-Standardklassen kennengelernt haben):

```
instanz = Klassenname.new(argument1, argument2, ...)
```

Betrachten Sie beispielsweise noch einmal den Konstruktor der Klasse `Lok` aus dem Einführungsbeispiel:

```
class Lok
  # Konstanten
  # ...

  # Grundzustand: Konstruktor
  def initialize(typ=OHNE)
    @typ = typ
  end

  # Methoden
  # ...
end
```

Wie Sie sehen, enthält eine `Lok`-Instanz nur eine Instanzvariable namens `@typ`, die angibt, wie viele Waggons die Lokomotive ziehen kann. Der Anfangswert wird aus der Parametervariablen `typ` des Konstruktors gelesen. Wenn die Konstruktordefinition einfach

```
def initialize(typ)
  @typ = typ
end
```

lauten würde, wären Sie gezwungen, bei der späteren Instanzerzeugung (oder *Instanziierung*) auf jeden Fall einen Wert anzugeben. Der vorliegende Konstruktor nutzt aber eine nützliche Eigenschaft von Ruby-Methoden: einen Standardwert für einen Parameter. Dieser wird per normaler Wertzuweisung (`=`) notiert und kommt zum Einsatz, wenn der entsprechende Wert beim Aufruf der Methode (beziehungsweise in diesem Fall des Konstruktors) *nicht* angegeben wird.

Die folgende Instanziierung erzeugt also eine Elektrolok mit einer Zugkraft von 50 Waggons:

```
lok = Lok.new(Lok::STROM)
```

Das nächste Beispiel erzeugt dagegen eine »leere Lok« – der nicht angegebene Parameter erhält den Standardwert `Lok::OHNE` (0 Waggons):

```
lok = Lok.new
```

Mehr über Standardwerte erfahren Sie im nachfolgenden Abschnitt, der sich mit Methoden beschäftigt, da dieses Verfahren nicht nur für Konstruktoren, sondern auch für alle Arten von Methoden angewandt werden kann.

Wenn Sie keinen Konstruktor schreiben, übernimmt Ihre Klasse automatisch denjenigen der Klasse, von der sie abgeleitet ist, also den ihrer Elternklasse. Explizite Vererbung wird weiter unten behandelt, aber jede Klasse wird automatisch von der

Ruby-Standardklasse `Object` abgeleitet. Diese stellt einen leeren Konstruktor bereit, der nichts Besonderes tut, aber eben einen `new`-Aufruf zur Instanziierung ermöglicht.

In der Praxis ist eine selbst geschriebene Klasse ohne eigenen Konstruktor normalerweise wenig hilfreich. Sie ist meist nur bei der Vererbung sinnvoll, wenn sich der Konstruktor einer abgeleiteten Klasse nicht von demjenigen der Elternklasse unterscheidet – aber auch das kommt selten vor, weil sich abgeleitete Klassen meist gerade durch eine erweiterte Datenstruktur von ihren Vorfahren unterscheiden.

Methoden

Die *Methoden*, genauer gesagt *Instanzmethoden*, einer Klasse sind der zweite Bestandteil der Kapselung neben den Instanzvariablen: Sie haben die Aufgabe, die in den Instanzvariablen gespeicherte Datenstruktur einer Instanz zu manipulieren und zu veröffentlichen. Sie bilden mit anderen Worten die Schnittstelle zwischen einer Instanz und der »Außenwelt«, das heißt anderen Programmteilen.

Jede Methodendefinition steht in einem Codeblock mit folgender Struktur:

```
def Methodenname
  ...
end
```

Die Zeile `def Methodenname(...)` wird dabei als *Methodenkopf* bezeichnet; die Codezeilen zwischen `def` und `end` heißen *Methodenrumpf*.

Um die Methode später aufzurufen, brauchen Sie nur ihren Namen, gefolgt von eventuellen Argumenten, anzugeben. Bei den hier erläuterten Instanzmethoden muss allerdings zuerst eine Instanz existieren. Betrachten Sie als Beispiel eine der Methoden der Klasse `Gueterzug` aus dem Einführungsbeispiel und ihren späteren Aufruf im globalen Code:

```
class Gueterzug
  # Konstruktor ...

  # Eine bestimmte Anzahl Waggons anhaengen
  def anhaengen(waggons)
    if @waggons + waggons <= @loks[0].get_kapazitaet +
                             @loks[1].get_kapazitaet
      @waggons += waggons
      # Erfolg melden
      true
    else
      # Misserfolg melden
      false
    end
  end
```

```
  # Weitere Methoden ...

end

# ...

# Neuen Zug erzeugen
zug = Gueterzug.new(Lok::STROM, Lok::DIESEL)

# ...

# 50 Waggons anhaengen
zug.anhaengen(50)
```

Die Methode `anhaengen` erwartet genau ein Argument, nämlich die Anzahl anzuhängender Waggons. Diese wird in der *Parametervariablen* `waggons` gespeichert. Innerhalb der Methode ist eine Parametervariable eine gewöhnliche lokale Variable, die in Ausdrücken verwendet und deren Wert auch geändert werden kann.

`anhaengen` überprüft zuerst, ob die vorhandenen Lokomotiven die neue Gesamtzahl von Waggons ziehen könnten. Wenn das der Fall ist, wird die Instanzvariable `@waggons` um den Übergabewert `waggons` erhöht. Um der Außenwelt mitzuteilen, dass das Anhängen erfolgreich war, enthält die nächste Zeile das Literal `true`. Die zuletzt ausgeführte Zeile einer Methode liefert nämlich den so genannten *Rückgabewert* der Funktion, der sich von außen abfragen lässt. Im `else`-Fall (es wären zu viele Waggons) wird die Anzahl dagegen nicht verändert, und der Rückgabewert ist `false`.

Beim späteren Aufruf der Methode können Sie den Rückgabewert ermitteln, müssen es aber nicht unbedingt – die Methode erledigt in beiden Fällen ihre Aufgabe. Der globale Code des Güterzug-Beispiels enthält beide Varianten:

```
# Ohne Beruecksichtigung des Rueckgabewerts
zug.anhaengen(50)
# ...
# Aufruf im Rahmen eines Ausdrucks (if-Bedingung)
# zur Nutzung des Rueckgabewerts
if zug.anhaengen(50)
  puts "50 weitere Waggons angehaengt."
else
  puts "Konnte keine 50 Waggons mehr anhaengen."
end
```

Parameter

Wie bereits mehrfach erwähnt, kann der Methodenkopf eine oder mehrere *Parametervariablen* enthalten. Diese werden ohne trennendes Leerzeichen in runde Klammern hinter den Methodennamen geschrieben und beim Aufruf mit Werten gefüllt, den *Argumenten*.

Im einfachsten Fall sind die Parameter normale Variablen – wie in dieser Methode, die den größeren von zwei Werten zurückliefert:

```
def groessere(z1, z2)
  if z1 > z2
    z1
  else
    z2
  end
end
```

Beim Aufruf *müssen* Sie genau zwei Argumente angeben, sonst erhalten Sie eine Fehlermeldung. Zum Ausprobieren können Sie die Datei mit dieser alleinstehenden Methode einfach um zwei Testzeilen erweitern, eine gültige und eine unvollständige: Zum Beispiel:

```
puts groessere(2, 7)
puts groessere(2)
```

Die Ausgabe lautet:

```
7
gr.rb:10:in `groessere': wrong number of arguments (1 for 2)
(ArgumentError)
```

Im ersten Fall wird also das korrekte Ergebnis 7 ausgegeben, im zweiten Fall kommt es zum erwarteten Fehler.

Eine praktische Möglichkeit, die Anzahl der Argumente variabel zu machen, ist der Einsatz von *Standardwerten*. Wichtig: Ab dem ersten Parameter mit Standardwert benötigen alle nachfolgenden ebenfalls Standardwerte, damit Ruby beim Aufruf die Reihenfolge der Argumente versteht. Bei der Klasse `Lok` besitzt beispielsweise der einzige Parameter des Konstruktors, der Loktyp, einen Standardwert:

```
def initialize(typ=OHNE)
  @typ = typ
end
```

Sie können eine `Lok` mit oder ohne Typangabe erzeugen; im letzteren Fall wird der Standardwert gewählt:

```
lok1 = Lok.new(Lok::DIESEL)
lok2 = Lok.new
```

Die Methode `get_typ` zeigt jeweils, welche Art von Lokomotive erzeugt wurde:

```
lok1.get_typ            # "Diesellok"
lok2.get_typ            # "keine"
```

Beim Konstruktor von `Gueterzug` besitzen sogar alle drei Parameter Standardwerte:

```
def initialize(lok0=Lok::OHNE, lok1=Lok::OHNE, waggons=0)
  @loks = Array.new
  @loks[0] = Lok.new(lok0)
```

```
  @loks[1] = Lok.new(lok1)
  @waggons = waggons
end
```

Diesen Konstruktor können Sie mit null bis drei Werten aufrufen, wobei Ihre Argumente der Reihe nach den Parametervariablen zugewiesen werden. Wenn Sie weniger als drei Argumente übergeben, erhalten die restlichen Parameter den jeweiligen Standardwert. Zum Beispiel:

```
gz1 = Gueterzug.new                  # Lok::OHNE, Lok::OHNE, 0
gz2 = Gueterzug.new(Lok::STROM)      # Lok::STROM, Lok::OHNE, 0
```

Eine andere Möglichkeit, die Standardwerte ausschließt, besteht darin, als letzten Parameter ein »Rest-Array« anzugeben. Diese Variable wird in der Parameterliste durch ein vorangestelltes Sternchen (*) gekennzeichnet, das jedoch nicht Teil des Variablennamens ist. In diesem Array werden beliebig viele zusätzliche Argumente gespeichert.

Die folgende Methode verwendet ein solches Array als einzigen Parameter und gibt dessen Elemente nummeriert aus:

```
def argliste(*args)
  i = 1
  args.each { |arg|
    printf "%2d. %s\n", i, arg
    i += 1
  }
}
```

Hier ein Aufrufbeispiel:

```
argliste("hallo", "was", "ist?", 1)
```

Die Ausgabe lautet:

```
 1. hallo
 2. was
 3. ist?
 4. 1
```

Eine prominente Ruby-Standardmethode, die sich diesen Mechanismus zunutze macht, ist `printf`: Sie nimmt genau einen Formatstring und beliebig viele optionale Ausdrücke entgegen.

Hier sehen Sie ein weiteres Beispiel – es berechnet die Summe aller (numerischen) Elemente eines Arrays:

```
def summe(*args)
  s = 0
  args.each { |z|
    s += z.to_f
  }
  s
end
```

Probieren Sie es aus. Beispielsweise ergibt

```
puts summe(2, 4, 6, 8)
```

die Ausgabe:

```
20.0
```

Eine weitere interessante Parameter-Lösung besteht darin, nur eine einzige Parametervariable entgegenzunehmen, die als Hash behandelt wird – vorzugsweise mit Symbolen als Schlüssel. Auf diese Weise können Sie leicht *benannte Parameter* nach dem Schema `:parameter => Wert` verwenden. Beispielsweise könnten Sie den Konstruktor von `Gueterzug` so umschreiben, dass er im übergebenen Hash nach den Schlüsseln `:zug0`, `:zug1` und `:waggons` sucht und andernfalls Standardwerte verwendet:

```
class Gueterzug
  # Grundzustand
  def initialize(hash)
    # Erst einmal Standardwerte setzen, dann gegebenenfalls ueberschreiben
    lok0 = Lok::OHNE
    lok1 = Lok::OHNE
    waggons = 0
    # Argument nur auswerten, falls Hash
    if hash.class == Hash
      if(hash[:lok0])
        lok0 = hash[:lok0]
      end
      if(hash[:lok1])
        lok1 = hash[:lok1]
      end
      if(hash[:waggons])
        waggons = hash[:waggons]
      end
    end
    # Instanzvariablen initialisieren
    @loks = Array.new
    @loks[0] = Lok.new(lok0)
    @loks[1] = Lok.new(lok1)
    @waggons = waggons
  end

  # Methoden ...
end
```

Mit diesem neuen Konstruktor können Sie nun flexibel selbst entscheiden, welche Werte Sie angeben möchten, und sogar die Reihenfolge spielt keine Rolle mehr. Zum Beispiel:

```
zug1 = Gueterzug.new(:waggons => 10, :lok0 => Lok::DIESEL)
zug2 = Gueterzug.new(:lok1 => Lok::STROM)
```

Innerhalb des Konstruktorrumpfes wurde die Reihenfolge ein wenig geändert, weil die Standardwerte nun nicht mehr aus der Parameterliste stammen, sondern nach-

träglich festgelegt werden. In diesem Fall ist es am praktischsten, einen Satz lokaler Variablen zuerst mit diesen Standardwerten zu initialisieren und ihn dann gegebenenfalls mit den Hashwerten zu überschreiben.

Bevor der Hash ausgelesen wird, überprüft eine Fallentscheidung aus Sicherheitsgründen mittels `class`, ob der übergebene Parameter überhaupt ein Hash ist. Anschließend wird jeder mögliche Schlüssel ausprobiert; falls er vorhanden ist, wird die entsprechende lokale Variable mit dessen Wert geändert.

Erst ganz zum Schluss werden Instanzvariablen mit den nun feststehenden Werten initialisiert.

Sie können sogar noch einen Schritt weitergehen und beide Konstruktorvarianten miteinander verknüpfen. Die folgende Variante des `Gueterzug`-Konstruktors verarbeitet sowohl die bisherigen 0 bis 3 Einzelwerte als auch den Hash:

```
class Gueterzug
  # Grundzustand
  def initialize(lok0=Lok::OHNE, lok1=Lok::OHNE, waggons=0)
    @loks = Array.new
    # Ist das erste Argument (lok0) ein Hash?
    if lok0.class == Hash
      # Ja, ein Hash: Benannte Argumente verarbeiten
      if(lok0[:lok0])
        @loks[0] = Lok.new(lok0[:lok0])
      else
        @loks[0] = Lok.new(Lok::OHNE)
      end
      if(lok0[:lok1])
        @loks[1] = Lok.new(lok0[:lok1])
      else
        @loks[1] = Lok.new(Lok::OHNE)
      end
      if(lok0[:waggons])
        @waggons = lok0[:waggons]
      else
        @waggons = 0
      end
    else
      # Nein, kein Hash: Klassische Argumente verarbeiten
      @loks[0] = Lok.new(lok0)
      @loks[1] = Lok.new(lok1)
      @waggons = waggons
    end
  end

  # Methoden ...
end
```

Dieser Konstruktor kann unter anderem wie folgt aufgerufen werden:

```
zug1 = Gueterzug.new                        # Standardwerte
zug2 = Gueterzug.new(Lok::DIESEL)           # nur eine Diesellok
```

```
zug3 = Gueterzug.new(:lok0 => Lok::STROM, :waggons => 10)
                                              # E-Lok, 10 Waggons
```

Die Auswertungsreihenfolge musste hier abermals etwas geändert werden, weil die Standardwerte nun wieder aus der Parameterübergabe stammen. Wenn ein Hash übergeben wird, befindet er sich automatisch in der Parametervariablen `lok0`; die beiden anderen Argumente erhalten zwar ihre Standardwerte, werden aber in diesem Fall nicht weiter beachtet. Wenn kein Hash übergeben wird, werden die Argumente dagegen wie gehabt von links nach rechts in den Variablen gesammelt.

Ein noch extremeres Beispiel, das die Datentypen der Argumente unterscheidet, wenn kein Hash übergeben wird, finden Sie im Abschnitt über Klassenmethoden.

Rückgabewerte

Dass Methoden einen Wert zurückgeben können, haben Sie bereits erfahren und praktisch eingesetzt. Interessant ist aber, dass Ruby zwei Möglichkeiten der Wertrückgabe kennt: alleinstehende Ausdrücke oder `return`.

Wenn Sie einen Ausdruck in eine einzelne Zeile schreiben, wird dieser zum Rückgabewert, sofern diese Zeile innerhalb der Methode als letzte ausgeführt wird, bevor der Rücksprung an die aufrufende Stelle erfolgt. Hier ein kleines `irb`-Beispiel, das den Namen und den Slogan eines bekannten Online-Auktionshauses variiert:

```
>> def xbay; 3; 2; 1; "deins"; end
>> xbay
=> "deins"
```

Der Rückgabewert dieser Methode ist der String `"deins"`, weil dieser Ausdruck die letzte Anweisung innerhalb der Methode ist.

Das explizite `return` funktioniert ein wenig anders. Probieren Sie es aus:

```
>> def xbay2; return 3; return 2; return 1; return "deins"; end
>> xbay2
=> 3
```

Der Rückgabewert ist 3. Woran liegt das? Nun, `return` legt nicht den Rückgabewert für den Fall eines eventuellen Rücksprungs fest, sondern sorgt selbst für den sofortigen Rücksprung mit dem angegebenen Wert. Die Ausführung der Methode ist beim ersten `return` beendet. Das können Sie überprüfen, indem Sie davor und dahinter Ausgabebefehle einbauen:

```
>> def zurueck; puts "vorher"; return true; puts "nachher"; end
>> zurueck
vorher
=> true
```

Sie sehen, dass der Text `"vorher"` ausgegeben wird; anschließend folgt der Rücksprung mit dem Rückgabewert `true`. Die Anweisung `puts "nachher"` wird niemals ausgeführt.

Mit Hilfe der beiden Wertrückgabeverfahren können Sie etwa die beiden folgenden Fassungen einer Methode schreiben, die überprüft, ob eine Zahl gerade (d.h. durch 2 teilbar) ist:

```
def gerade1(zahl)
  if zahl % 2 == 0
    true
  else
    false
  end
end

def gerade2(zahl)
  if zahl % 2 == 0
    return true
  end
  return false
end
```

Für welche der beiden Varianten Sie sich entscheiden, ist letztlich Geschmackssache. Wie Sie sehen, benötigt `gerade2` keinen `else`-Teil, denn wenn die Funktion an dieser Stelle noch nicht verlassen wurde, ist klar, dass die untersuchte Zahl ungerade sein muss. Möglicherweise schätzen Sie das `else` aber, weil es die Struktur der Methode deutlicher macht. In diesem Fall hält Sie auch nichts davon ab, die `return`-Fassung mit einem (eigentlich überflüssigen) `else` zu schreiben.

Iteratoren definieren

Bereits in Kapitel 2 haben Sie die Nützlichkeit der eingebauten Ruby-Iteratoren kennengelernt. Noch erfreulicher ist, dass Sie auch eigene Iteratoren implementieren können. Das geht erstaunlich einfach: Sobald Sie innerhalb einer Methode die Anweisung `yield` aufrufen, wird der Code innerhalb des übergebenen Blocks ausgeführt. Hier ein erster kleiner Iterator, der die im Block befindlichen Anweisungen unverändert ausführt, und zwar mit einer zufälligen Anzahl von Durchläufen zwischen 1 und 10:

```
def zufaellig_oft
  (rand(10)+1).times {
    yield
  }
end
```

Hier ein Praxistest zur Vorweihnachtszeit:

```
>> zufaellig_oft { print "Ho! " }
Ho! Ho! Ho! Ho! Ho! => 5
>> zufaellig_oft { print "Ho! " }
Ho! => 1
>> zufaellig_oft { print "Ho! " }
Ho! Ho! Ho! Ho! Ho! Ho! => 6
```

Eine solche Methode kann auch normale Parameter entgegennehmen. Hier eine Variante von `zufaellig_oft`, die statt der `10` ein Argument als Höchstwert für `rand` verwendet:

```
def zufaellig_oft(n)
  (rand(n)+1).times {
    yield
  }
end
```

Diese geänderte Version wird dann etwa wie folgt verwendet:

```
>> zufaellig_oft(3) { print "Ho! " }
Ho! Ho! Ho! => 3
>> zufaellig_oft(3) { print "Ho! " }
Ho! Ho! => 2
```

Wenn Sie `yield` mit Argumenten aufrufen, werden diese an den Block übergeben und können mit Hilfe des bekannten Konstrukts `|Variable, ...|` zu Beginn des Blocks in Empfang genommen werden. Das folgende überaus nützliche Beispiel erweitert die Standardklasse `Array` um einen Iterator namens `search` (Grundsätzliches zur Erweiterung von Standardklassen lesen Sie im Abschnitt »Globale Methoden und neue Methoden für Standardklassen« auf Seite 184). Er durchsucht das Array nach einem Wert, der als konventioneller Parameter angegeben wird, und liefert die Array-Positionen aller Fundstellen an den Block zurück. Der Rückgabewert ist schließlich die Anzahl der gefundenen Treffer:

```
class Array
  def search(wert)
    # Startposition 0
    pos = 0
    # Anfangs 0 Treffer
    count = 0
    # Alle Elemente durchgehen
    self.each { |w|
      # Uebereinstimmung mit dem Suchwert?
      if w == wert
        # Trefferzaehler erhoehen
        count += 1
        # Position an Block liefern
        yield pos
      end
      # Position erhoehen
      pos += 1
    }
    # Trefferzahl zurueckliefern
    count
  end
end
```

Probieren Sie es nun aus. Als Code innerhalb des Blocks wird die jeweils gelieferte Position hier einfach mit einem kleinen Text ausgegeben:

```
>> a = %w(Hallo Welt hallo Mars hallo Jupiter hallo Galaxis)
>> z = a.search("hallo") { |pos| puts "Hallo an Position #{pos}" }
Hallo an Position 2
Hallo an Position 4
Hallo an Position 6
=> 3
>> puts "#{z} Treffer"
3 Treffer
```

Das einzig Ärgerliche ist, dass die Methode `search` nun unverrückbar ein Iterator ist und somit eine Fehlermeldung ausgibt, wenn ihr kein Block übergeben wird:

```
>> a.search("hallo")
LocalJumpError: no block given
```

Um das zu verhindern, können Sie auch »Hybrid-Methoden« schreiben, die nur dann als Iteratoren fungieren, wenn ein Block übergeben wird, und ansonsten als brave Standardmethode. Zur Entscheidungsfindung wird die Methode `block_given?` verwendet. Die folgende Erweiterung von `Array.search` liefert bei einem Aufruf ohne Block die Position des ersten Treffers; wenn ein Block vorhanden ist, erfüllt sie dagegen ihre bisherige Aufgabe:

```
class Array
  def search(wert)
    pos = 0
    count = 0
    self.each { |w|
      if w == wert
        # Block vorhanden?
        if block_given?
          # Verfahren wie bisher
          count += 1
          yield pos
        else
          # Einfach erste Fundstelle liefern
          return pos
        end
      end
      pos += 1
    }
    count
  end
end
```

Rufen Sie diese neue Variante nun einmal mit und einmal ohne Block auf, um den Unterschied zu testen. Mit dem obigen Beispiel-Array sähe dies beispielsweise so aus:

```
>> a.search("hallo")
=> 2
>> a.search("hallo") { |pos| puts "Fundstelle: #{pos}" }
Fundstelle: 2
```

```
Fundstelle: 4
Fundstelle: 6
=> 3
```

Anregung: Schreiben Sie eine zusätzliche `Array`-Erweiterungsmethode namens `reg_search`, die nach demselben Schema keine festen Werte, sondern reguläre Ausdrücke sucht.

Klassenmethoden

Die Methoden, die in den bisherigen Beispielen innerhalb von Klassen definiert wurden, waren Instanzmethoden. Diese existieren nur und können nur aufgerufen werden, wenn Sie zuvor eine Instanz der entsprechenden Klasse erzeugen. Im vorigen Kapitel haben Sie dagegen auch Methoden wie `File.exists?` kennengelernt, die ohne Existenz einer Instanz als Bestandteil der Klasse selbst aufgerufen werden. Solche Methoden heißen deshalb *Klassenmethoden*.

Klassenmethoden sind immer dann nützlich, wenn eine Aufgabe durchgeführt werden soll, die mit der Klasse in lockerem Zusammenhang steht. Um etwa zu überprüfen, ob eine Datei existiert, ist es nicht nötig, sie zu öffnen.[3] Instanzen der Klasse `File` sind aber grundsätzlich geöffnete Dateien, so dass für die Überprüfung nur eine Klassenmethode in Frage kommt. Dasselbe gilt etwa für

```
File.rename(AlterName, NeuerName)
```

Geöffnete Dateien können nämlich nicht umbenannt werden.

Eine Klassenmethode wird implementiert, indem Sie bei ihrer Definition den durch Punkt getrennten Klassennamen vor den Methodennamen setzen. Das folgende Beispiel kapselt einen Termin für einen elektronischen Terminkalender. Die Klassenmethode `Termin.datum` formatiert ein übergebenes `Time`-Objekt mittels `strftime` in deutscher Schreibweise. Hier der Code mitsamt Anwendungsbeispielen:

```
class Termin

  # Prioritaetskonstanten
  NIEDRIG  = -1
  NORMAL   = 0
  DRINGEND = 1

  def initialize(datum, eintrag, prioritaet=NORMAL)
    @datum = Time.parse(datum)
    @eintrag = eintrag
    @prioritaet = prioritaet
  end
```

3 Und wenn die Datei nicht existiert, ist es nicht einmal möglich, sie zu öffnen, sondern führt zu einer Fehlermeldung, die die Benutzer Ihres Programms irritiert oder nervt.

```
  # Klassenmethode: deutsches Datumsformat
  def Termin.datum(dat)
    dat.strftime("%d.%m.%Y, %H:%M")
  end

  # Prioritaet als String zurueckgeben
  def get_prioritaet
    case @prioritaet
    when NIEDRIG
      "Niedrig"
    when NORMAL
      "Normal"
    when DRINGEND
      "Wichtig!"
    end
  end

  # Formatierten Termin zurueckgeben
  def get_termin
    ausgabe = sprintf("%s\n%s\nPrioritaet: %s\n",
              Termin.datum(@datum), @eintrag,
              get_prioritaet)
    ausgabe
  end

end

t1 = Termin.new("18:00", "Backup-Tapes wechseln")
t2 = Termin.new("2006/11/08 9:00",
                "Kundenberatung Linux-Migration", Termin::DRINGEND)
t3 = Termin.new("2006/11/09", "Spam loeschen", Termin::NIEDRIG)

# Aktuelles Datum
d = Time.new
printf "Es ist jetzt %s\n\n", Termin.datum(d)
puts "TO DO:"
puts
puts t1.get_termin
puts
puts t2.get_termin
puts
puts t3.get_termin
```

Hier ist die Ausgabe des Skripts nach dessen Ausführung:

```
> ruby termine.rb
Es ist jetzt 07.11.2006, 16:48

TO DO:

07.11.2006, 18:00
Backup-Tapes wechseln
Prioritaet: Normal
```

```
08.11.2006, 09:00
Kundenberatung Linux-Migration
Prioritaet: Wichtig!

09.11.2006, 00:00
Spam loeschen
Prioritaet: Niedrig
```

Den Code müssten Sie mittlerweile ohne längere Erklärungen verstehen. Die Klassenmethode, um die es hier vor allem geht, besitzt folgende Definition:

```
def Termin.datum(dat)
  dat.strftime("%d.%m.%Y, %H:%M")
end
```

Die Methode wird einmal innerhalb und einmal außerhalb der Klasse aufgerufen. Der erste Aufruf steht in der Instanzmethode `get_termin`; er formatiert das in der Instanzvariablen `@datum` gespeicherte Datum:

```
ausgabe = sprintf("%s\n%s\nPrioritaet: %s\n",
          Termin.datum(@datum), @eintrag,
          get_prioritaet)
```

Im globalen Code wird die Methode dagegen zur Formatierung des aktuellen Datums aufgerufen:

```
d = Time.new
printf "Es ist jetzt %s\n\n", Termin.datum(d)
```

Als umfangreicheres Beispiel, das auch eine Klassenmethode als Rechenhelfer verwendet, sehen Sie in Beispiel 4-3 einen klassenbasierten RGB-Farbumrechner. Die meisten wichtigen Programmschritte wurden bereits für ein ähnliches Beispiel in Kapitel 2 erläutert.

Beispiel 4-3: RGB-Farbumwandlung als Klasse, rgb_farbe.rb

```
 1  class RGBFarbe

 2    # Konstruktor
 3    def initialize(arg1=0, arg2=0, arg3=0)
 4      # Absolute Standardwerte setzen -- spart Arbeit!
 5      @rot = 0
 6      @gruen = 0
 7      @blau = 0
 8      if arg1.class == Hash
 9        # Hash? Komponenten auslesen
10        if arg1[:rot]
11          @rot = arg1[:rot]
12        end
13        if arg1[:gruen]
14          @gruen = arg1[:gruen]
15        end
16        if arg1[:blau]
17          @blau = arg1[:blau]
18        end
```

Beispiel 4-3: RGB-Farbumwandlung als Klasse, rgb_farbe.rb (Fortsetzung)

```
19        elsif arg1.class == String
20          # String? Als HTML-Farbstring behandeln und umrechnen
21          # Zunaechst eventuelles #-Zeichen entfernen
22          arg1.sub!(/^#/, "")
23          # Auf sechs Zeichen verlaengern
24          while arg1.length < 6
25            arg1 += "0"
26          end
27          # RGB-Komponenten berechnen
28          @rot = arg1[0, 2].to_i(16)
29          @gruen = arg1[2, 2].to_i(16)
30          @blau = arg1[4, 2].to_i(16)
31        else
32          # Ganzzahlen erzwingen
33          @rot = arg1.to_i
34          @blau = arg2.to_i
35          @gruen = arg3.to_i
36        end
37      end

38      # Klassenmethode: Gegebene Werte in Hex-RGB umrechnen
39      def RGBFarbe.hex(r, g, b)
40        # Gesamtwert berechnen
41        farbe = r << 16 | g << 8 | b
42        # In Hex umrechnen und auf 6 Stellen aufstocken
43        hexfarbe = farbe.to_s(16)
44        while hexfarbe.length < 6
45          hexfarbe = "0" + hexfarbe
46        end
47        "#" + hexfarbe.upcase
48      end

49      # Rotwert zurueckgeben
50      def get_rot
51        @rot
52      end

53      # Gruenwert zurueckgeben
54      def get_gruen
55        @gruen
56      end

57      # Blauwert zurueckgeben
58      def get_blau
59        @blau
60      end

61      # Alle drei Einzelwerte als Array zurueckgeben
62      def get_rgb_array
63        [@rot, @gruen, @blau]
64      end
```

Beispiel 4-3: RGB-Farbumwandlung als Klasse, rgb_farbe.rb (Fortsetzung)

```
65     # Alle drei Einzelwerte als Hash zurueckgeben
66     def get_rgb_hash
67       {:rot => @rot, :gruen => @gruen, :blau => @blau}
68     end

69     # HTML-Farbcode zurueckgeben
70     def get_html_farbe
71       RGBFarbe.hex(@rot, @gruen, @blau)
72     end

73     # Die naechstgelegene Farbe der Webpalette ermitteln
74     def get_naechste_web_farbe
75       # Naechstgelegene Web-Farbwerte berechnen
76       wr = (@rot + 25) / 51 * 51
77       wg = (@gruen + 25) / 51 * 51
78       wb = (@blau + 25) / 51 * 51
79       RGBFarbe.hex(wr, wg, wb)
80     end

81   end
```

Beachten Sie besonders den Konstruktor (Zeile 3-37). Er ermöglicht die Angabe eines Hashes, einzelner Rot-, Grün- und Blauwerte oder eines HTML-Farbstrings mit oder ohne Raute (etwa `"#FF0000"` oder `"99CC00"`).

Die Klassenmethode `hex` steht in Zeile 39-48. Sie rechnet die drei übergebenen Integerwerte (für Rot, Grün und Blau) in einen HTML-Farbstring um.

Speichern Sie die Klasse in einer eigenen Datei namens `rgb_farbe.rb` und importieren Sie sie wie folgt in `irb`:

```
>> require "rgb_farbe.rb"
```

Nun können Sie die verschiedenen Aufrufvarianten des Konstruktors und die Methoden der Klasse `RGBFarbe` aufrufen. Zum Beispiel:

```
>> f1 = RGBFarbe.new(50, 100, 200)
=> #<RGBFarbe:0x2b17464 @blau=100, @gruen=200, @rot=50>
>> f1.get_html_farbe
=> "#32C864"
>> f1.get_naechste_web_farbe
=> "#33CC66"
>> f2 = RGBFarbe.new("#FF0000")
=> #<RGBFarbe:0x2b0bda8 @blau=0, @gruen=0, @rot=255>
>> f2.get_rgb_hash
=> {:rot=>255, :gruen=>0, :blau=>0}
>> f3 = RGBFarbe.new(:gruen => 200, :rot => 100)
=> #<RGBFarbe:0x28d1d94 @blau=0, @gruen=200, @rot=100>
>> f3.get_html_farbe
=> "#64C800"
```

Wie Sie sehen, liefert Ihnen ein Konstruktoraufruf in `irb` jeweils interessante Informationen über die neue Instanz – hinter dem Klassennamen stehen die Objekt-ID und die Anfangswerte der Instanzvariablen. Das ist zum Ausprobieren neuer Klassen sehr hilfreich.

Zum Schluss können Sie auch die Klassenmethode ausprobieren, etwa wie folgt:

```
>> RGBFarbe.hex(255, 153, 51)
=> "#FF9933"
```

»Globale« Methoden und neue Methoden für Standardklassen

Übrigens wurden auch in der nicht objektorientierten Lösung des einführenden Eisenbahn-Beispiels und in anderen Skripten in diesem Buch Methoden definiert. Es handelt sich dabei scheinbar um globale Methoden (in anderen Sprachen *Funktionen* genannt), da sie sich ohne Instanz aufrufen lassen – in dem imperativen Skript heißt es etwa

```
anhaengen(50)
```

statt

```
zug.anhaengen(50)
```

Wie passt dies zu der Tatsache, dass Ruby eine konsequent objektorientierte Programmiersprache ist? Ganz einfach: Formal betrachtet ist *jede* Methode, die Sie außerhalb einer Klasse schreiben, trotzdem eine Instanzmethode! Der Trick besteht darin, dass der Ruby-Ausführungskontext, also beispielsweise das aktuelle Skript oder die aktuelle `irb`-Sitzung, eine Instanz der allgemeinen Klasse `Object` ist. Das können Sie leicht herausfinden, indem Sie in `irb` Folgendes eingeben:

```
>> self.class
=> Object
```

Das Schlüsselwort `self` dient dem Selbstbezug einer Instanz. Die Methode `class` wurde bereits in Kapitel 2 beschrieben; sie liefert die Klasse, zu der irgendein Objekt gehört – und wie Sie wissen, ist in Ruby *alles* ein Objekt.

Da `Object` der Urahn sämtlicher Ruby-Klassen ist, können Sie Ihre »globalen« Methoden in `irb` nicht nur alleinstehend, sondern auch als Instanzmethoden jedes beliebigen Objekts aufrufen. Probieren Sie es aus, indem Sie folgende kurze Methode definieren:

```
>> def hallo; puts "Hallo!"; end
```

Rufen Sie die Methode zunächst wie gehabt ohne Objektbezug auf:

```
>> hallo
Hallo!
```

Ein Synonym für diesen Aufruf ist, wie eben ausgeführt:

```
self.hallo
```

Probieren Sie die Methode nun für beliebige Literale und andere Objekte aus, z.B.:

```
1.hallo              # Fixnum
"hallo".hallo        # String
[1, 2, 3].hallo      # Array
```

Es funktioniert, denn das Ergebnis ist immer dasselbe: die Ausgabe des Textes »Hallo!«.

Dieses interessante Phänomen können Sie sich zunutze machen, indem Sie innerhalb des Methodenrumpfes mittels `self` Bezug auf die Instanz nehmen, für die die Methode aufgerufen wurde. Hier ein sehr einfaches Beispiel:

```
>> def verdoppeln; self * 2; end
```

Rufen Sie diese neue Methode nun für verschiedene Objekte auf, von denen Sie wissen, dass man sie mit einer Ganzzahl multiplizieren kann:

```
>> 7.verdoppeln
=> 14
>> "Hallo".verdoppeln
=> "HalloHallo"
>> [1, 2, 3].verdoppeln
=> [1, 2, 3, 1, 2, 3]
```

Natürlich gibt es auch Objekte, die keine Multiplikation kennen. In diesem Fall erhalten Sie eine Fehlermeldung. Hier zum Beispiel ein Hash-Literal:

```
>> {:name => "Schmitz", :vorname => "Jupp"}.verdoppeln
NoMethodError: undefined method `*' for {:name=>"Schmitz", :vorname=>"Jupp"}:Hash
```

In einer gespeicherten Skriptdatei funktioniert es nicht ganz so einfach. Um dort eine Methode auf beliebige Objekte anwenden zu können,[4] müssen Sie sie als Erweiterung der Klasse `Object` schreiben, indem Sie sie einfach in den Block

```
class Object
  ...
end
```

setzen. Dies ersetzt die Standardklasse `Object` nicht etwa, sondern erweitert sie um die entsprechende Methode. Hier eine erweiterte Fassung von `verdoppeln` innerhalb eines Skriptbeispiels, die für alle Werte außer Zahlen, Strings und Arrays `nil` zurückgibt. Dazu untersucht sie mittels `class.to_s` den in Strings umgewandelten Typ von `self` und reagiert entsprechend:

```
class Object
  def verdoppeln
    case self.class.to_s
    when "Fixnum", "Bignum", "Float", "String", "Array"
      self * 2
    else
      nil
```

4 Es geht hier wohlgemerkt nicht um Methoden ohne expliziten Objektbezug wie im imperativen Güterzug-Beispiel (diese funktionieren völlig problemlos), sondern eben nur um solche, die als `Instanz.Methode` aufgerufen werden sollen.

```
    end
  end
end

# Passende Klassen
puts (2.verdoppeln)
puts 7.1.verdoppeln
puts "Hallo".verdoppeln
puts ["a", "b", "c"].verdoppeln
# Unpassende Klassen
puts true.verdoppeln
puts verdoppeln   # Skript ist Instanz von Object
```

Wenn Sie dieses Skript speichern und ausführen, erhalten Sie folgende Ausgabe:

```
> ruby verdoppeln.rb
4
14.2
HalloHallo
a, b, c, a, b, c
nil
nil
```

Interessanterweise können Sie aber nicht nur die allgemeine Klasse `Object`, sondern auch gezielt einzelne Klassen erweitern. Es wäre beispielsweise sehr nützlich, die in Kapitel 2 besprochene Vorgehensweise zum Runden von Fließkommazahlen gleich in die Klasse `Float` einzubauen. Das funktioniert eigentlich sehr einfach. Als zusätzliche Anforderung soll die Methode aber nach wie vor `round` heißen und die neue Funktionalität nur dann besitzen, wenn eine Nachkommastellenanzahl angegeben wird. Hilfreich ist dafür die Ruby-Standardmethode `alias`, die es Ihnen ermöglicht, einer bestehenden Methode einen Ersatznamen zuzuweisen und den bisherigen Namen anderweitig zu verwenden. Die Syntax lautet:

```
alias NeuerMethodenname AlterMethodenname
```

Hier zunächst der vollständige Code der `Float`-Erweiterung nebst einigen Testfällen – Erläuterungen folgen:

```
# Aenderungen in der Standardklasse Float
class Float

  # Traditionelle Round-Methode umbenennen
  alias oldround round

  # Neue, angepasste Round-Methode
  def round(stellen=0)
    if stellen == 0
      # Bisherige Methode aufrufen bei 0 Nachkommastellen
      self.oldround
    else
      # Auf die gewuenschte Anzahl Stellen runden
      (self * 10**stellen).oldround.to_f / 10**stellen
    end
```

```
  end

end

puts 4.44444.round
puts 4.99999.round
puts 4.44444.round(2)
puts 4.99999.round(2)
puts 4444.444444.round(-2)
```

Wenn Sie das Skript ausführen, erhalten Sie folgende Ausgabe:

```
4
5
4.44
5.0
4400.0
```

Was passiert in diesem Programm? Zunächst findet eine Erweiterung der Klasse `Float` statt, so dass der relevante Code zwischen `class Float` und `end` steht. Dort erhält die Methode `round` als Erstes einen Alternativnamen:

```
alias oldround round
```

Beachten Sie, dass der Methodenname `round` an dieser Stelle noch funktioniert; der Aliasname wird zusätzlich vergeben. Als Nächstes wird unter dem Namen `round` allerdings eine neue Methode vereinbart, so dass die klassische nur noch als `old-round` zur Verfügung steht.

Die neue Methode `round` nimmt eine Nachkommastellenanzahl als optionales Argument mit dem Standardwert 0 entgegen. Wenn `stellen` (explizit oder automatisch) 0 ist, wird für `self` – den Wert der Fließkommazahl – einfach `oldround` aufgerufen. Andernfalls muss eine etwas komplexere Berechnung durchgeführt werden: Die Zahl wird mit der Zehnerpotenz der Nachkommastellenanzahl multipliziert und dann mittels `oldround` ganzzahlig gerundet. Das Ergebnis wird per `to_f` in eine Fließkommazahl umgewandelt und wieder durch den Faktor der ursprünglichen Multiplikation geteilt. So bleibt genau die gewünschte Anzahl von Nachkommastellen übrig. Und wie das letzte Beispiel zeigt, können Sie mit Hilfe eines negativen Arguments sogar vor dem Komma runden.

Diese Erweiterung ist am nützlichsten, wenn sie in einer externen, importierbaren Bibliotheksdatei zur Verfügung steht. Zu diesem Zweck können Sie den Teil von `def Float` bis zum äußersten `end` in einer separaten Datei speichern (beispielsweise unter dem Namen *round.rb*) und dann wie folgt am Anfang eines neuen Programms im selben Verzeichnis importieren:

```
require "round.rb"
```

Diese flexible Erweiterbarkeit der vorgegebenen Elemente ist wieder einmal eines jener Features, die Ruby zu einer vollkommen einzigartigen Sprache machen. Weil es so überaus nützlich ist, hier gleich noch ein Beispiel – es erweitert die Klasse

Array um die Methoden sum (Summe aller Elemente) und average (Mittelwert aller Elemente):

```
# Aenderungen in der Standardklasse Array
class Array

  # Summe aller Elemente
  def sum
    s = 0
    self.each { |z|
      # In Fliesskommazahl umrechnen - 0, wenn keine Zahl
      s += z.to_f
    }
    # Summe zurueckgeben
    s
  end

  # Mittelwert
  def average
    self.sum / self.length
  end

end
```

Viel zu erklären gibt es hier eigentlich nicht mehr, da Ihnen alle Anweisungen und Konstrukte des Skripts bekannt sind. Speichern Sie es unter einem Namen wie *array_ext.rb* und importieren Sie es in ein eigenes Skript oder in irb:

```
>> require "array_ext.rb"
=> true
```

Nun können Sie die Erweiterungen testen:

```
>> arr = [1, 2, 6]
>> arr.sum
=> 9.0
>> arr.average
=> 3.0
```

Operatoren definieren und ändern

Eine der überraschendsten Eigenschaften von Ruby[5] ist vielleicht die Tatsache, dass viele Operatoren auch nichts anderes als spezielle Methoden sind. Sie werden auch auf dieselbe Weise definiert. Vielleicht möchten Sie einen praktischeren Güterzug haben, bei dem Sie Waggons einfach per += und -= anhängen beziehungsweise abhängen können? Nichts leichter als das: Definieren Sie einfach die Methoden + und -. Wichtig ist in diesem Zusammenhang, dass diese speziellen Methoden als Ergebnis die entsprechend modifizierte Instanz der Klasse zurückliefern müssen.

5 Es sei denn, Sie haben Erfahrungen mit in einer Sprache wie C++, in der sich Operatoren »überladen« lassen – allerdings umständlicher als in Ruby.

Das folgende Beispiel leitet eine Klasse von Gueterzug ab und fügt die beiden Methoden hinzu (die dazu nötige Vererbung wird weiter unten genauer erläutert):

```
# "EinfacherGueterzug" uebernimmt alle Eigenschaften
# und Methoden von "Gueterzug"
class EinfacherGueterzug < Gueterzug

  # Methode +: Waggons anhaengen
  def +(waggons)
    # Durch anhaengen testen, ob Anzahl passt
    if anhaengen(waggons)
      # Neue Instanz mit geaenderter Waggonzahl zurueckgeben
      return EinfacherGueterzug.new(@loks[0].get_kapazitaet,
                                    @loks[1].get_kapazitaet, @waggons)
    else
      # Die bisherige Instanz unveraendert zurueckgeben
      return self
    end
  end

  # Methode -: Waggons abhaengen
  def -(waggons)
    # Durch abhaengen testen, ob Anzahl passt
    if abhaengen(waggons)
      # Neue Instanz mit geaenderter Waggonzahl zurueckgeben
      return EinfacherGueterzug.new(@loks[0].get_kapazitaet,
                                    @loks[1].get_kapazitaet, @waggons)
    else
      # Die bisherige Instanz unveraendert zurueckgeben
      return self
    end
  end

end
```

Wenn Sie das ursprüngliche objektorientierte Güterzug-Skript, *zug_oo.rb*, und das Skript mit dieser Erweiterung, *einfacher_gueterzug.rb*, nacheinander in irb importieren, können Sie die neue Klasse ausprobieren. Beispielsweise so:

```
>> require "zug_oo.rb"
[Vollständige Ausgabe dieses Skripts]
=> true
>> require "einfacher_gueterzug.rb"
=> true
>> egz = EinfacherGueterzug.new(Lok::DIESEL, Lok::OHNE, 40)
=> #<EinfacherGueterzug:0x2b13710 @loks=[#<Lok:0x2b136d4 @typ=50>,
   #<Lok:0x2b136c0 @typ=0>], @waggons=40>
>> egz += 10
=> #<EinfacherGueterzug:0x2b12374 @loks=[#<Lok:0x2b12338 @typ=50>,
   #<Lok:0x2b12324 @typ=0>], @waggons=50>
>> egz += 10
=> #<EinfacherGueterzug:0x2b12374 @loks=[#<Lok:0x2b12338 @typ=50>,
   #<Lok:0x2b12324 @typ=0>], @waggons=50>
```

Dass nach dem ersten (erfolgreichen) Anhängen von 10 Waggons tatsächlich ein neues `EinfacherGueterzug`-Objekt erzeugt wurde, sehen Sie an den geänderten IDs des Zugs und seiner Loks. Der zweite Versuch hat keinen Erfolg, da die Anzahl zu hoch würde, und liefert daher die ursprüngliche Instanz zurück.

Vielleicht fragen Sie sich zudem, wo die Anzahl der Waggons im Erfolgsfall eigentlich geändert wird. Ganz einfach: Genau zu diesem Zweck wird zunächst `anhaengen` beziehungsweise `abhaengen` aufgerufen. Diese Methoden ändern den Wert der Instanzvariablen `@waggons` und geben `true` zurück, wenn die neue Anzahl im Rahmen des Möglichen liegt, und ansonsten einfach `false`. Danach kann `@waggons` einfach zur Konstruktion der neuen Instanz verwendet werden, da der Wert bereits korrekt ist. Im Grunde könnte man sich sogar das `if` sparen, würde dann aber bei einer missglückten Änderung der Waggonanzahl eine überflüssige Instanziierung durchführen.

Hier noch ein besonders interessantes Beispiel: Erinnern Sie sich noch daran, dass in Kapitel 2 die Rede davon war, man könne keine Strings subtrahieren? Nun, von Hause aus kann Ruby das auch nicht. Aber nichts hält Sie davon ab, der Klasse `String` eine Methode namens - zu spendieren, die den angegebenen Teilstring aus dem ursprünglichen Text löscht. Da man per Textindex auf Strings zugreifen kann, ist das ein Kinderspiel:

```
class String
  def -(txt)
    self[txt] = ""
    self
  end
end
```

Probieren Sie es aus:

```
>> "Ruby ist unflexibel" - "un"
=> "Ruby ist flexibel"
>> "ball" - 0
=> "all"
```

Wie Sie sehen, können Sie den »abzuziehenden« Teil sowohl als Text als auch numerisch angeben, weil beide Varianten als Index zulässig sind.

Rekursion: Unterverzeichnisse durchsuchen

Als komplexeres Beispiel für die Arbeit mit Methoden wird hier das im vorigen Kapitel angekündigte Skript gezeigt, das die Inhalte eines Verzeichnisses und seiner Unterverzeichnisse mit passender Einrückung auflistet.

Um eine beliebig tief verschachtelte Verzeichnishierarchie zu verarbeiten (oder andere verschachtelte Probleme zu lösen), kommt ein Verfahren namens *Rekursion* zum Einsatz, bei dem sich eine Methode für einen untergeordneten Bereich immer wieder selbst aufruft. Wie das funktioniert, zeigt zum Beispiel dieses bekannte Kinderlied:

> Ein Hund kam in die Küche und stahl dem Koch ein Ei
> Da nahm der Koch den Löffel und schlug den Hund entzwei
> Da kamen alle Hunde und gruben ihm ein Grab
> Und setzten ihm ’nen Grabstein, worauf geschrieben stand:
> Ein Hund kam in die Küche und stahl dem Koch ein Ei
> Da nahm der Koch den Löffel und schlug den Hund entzwei
> Da kamen alle Hunde und gruben ihm ein Grab
> Und setzten ihm ’nen Grabstein, worauf geschrieben stand:
> Ein Hund kam ...

Übertragen auf das Verzeichnisproblem, bedeutet das, dass sich die zuständige Methode jedes Mal, wenn sich ein Eintrag als Unterverzeichnis entpuppt, mit diesem Verzeichnis als Pfad selbst aufruft. Das stellt sicher, dass der gesamte untergeordnete Verzeichnisbaum verarbeitet wird.

Die Besonderheit einer rekursiven Methode besteht darin, dass Sie sie innerhalb ihres eigenen Methodenrumpfes aufrufen. Wichtig ist dabei eine *Abbruchbedingung*: Jeder Methodenaufruf belegt etwas Speicher, weil der Computer sich merken muss, an welcher Stelle das Programm nach Ausführung der Methode weitergeht (*Rücksprungadresse*). Eine unendliche Rekursion würde so irgendwann den Speicher überlaufen lassen. Beim Verzeichnisbeispiel existiert eine natürliche Abbruchbedingung, weil es auf Datenträgern mit endlicher Größe natürlich keine unendlich tief verschachtelten Verzeichnishierarchien geben kann.

Hier als kleines Beispiel ein Programm, das das obige Gedicht rekursiv ausgibt. Als Abbruchbedingung erhält die Methode die Information, wie oft der Text auf den »Grabstein« passt. Dieser Wert wird jeweils um 1 vermindert; bei 0 erfolgt kein weiterer Aufruf mehr:

```
def grabstein(anzahl=0)
  # Text ausgeben
  puts "Ein Hund kam in die Kueche und stahl dem Koch ein Ei"
  puts "Da nahm der Koch den Loeffel und schlug den Hund entzwei"
  puts "Da kamen alle Hunde und gruben ihm ein Grab"
  puts "Und setzten ihm 'nen Grabstein, worauf geschrieben stand:"
  # Rekursiver Aufruf mit anzahl-1, falls anzahl noch > 1
  grabstein(anzahl-1) if anzahl > 1
end

# Erster Aufruf, z.B. 5 Durchgaenge
grabstein(5)
```

Das zu erwartende Ergebnis dieses Programms sind fünf Ausgaben des Textes.

Nach diesen Vorab-Erläuterungen sehen Sie in Beispiel 4-4 das komplette Verzeichnisskript. Geben Sie es wie gewohnt ein; ein Ausführungsbeispiel und genauere Erläuterungen folgen.

Beispiel 4-4: Der rekursive Verzeichnisauflister, listdir.rb

```
 1  # Methode listdir
 2  #   dir    = Verzeichnispfad
 3  #   indent = aktuelle Einrueckungstiefe (Standardwert 0)
 4  def listdir(dir, indent=0)
 5    # Verzeichnis des aktuellen Pfads oeffnen
 6    d = Dir.new(dir)
 7    # Verzeichnis in Schleife auslesen
 8    while entry = d.read
 9      # Neuen Pfad konstruieren
10      path = dir + "/" + entry
11      # Ist der Pfad ein Verzeichnis?
12      if File.directory?(path)
13        # Ja, Verzeichnis
14        # Einrueckung + Verzeichnisname in eckigen Klammern ausgeben
15        puts "#{' ' * indent}[#{entry}]"
16        # Rekursiver Aufruf mit neuem Pfad und akt. Einrueckungstiefe+2,
17        # falls Eintrag nicht . oder .. ist
18        listdir(path, indent+2) if entry !~ /^\.{1,2}$/
19      else
20        # Nein, kein Verzeichnis
21        # Einrueckung + aktuellen Eintrag ausgeben
22        puts "#{' ' * indent}#{entry}"
23      end
24    end
25    # Verzeichnis schliessen
26    d.close
27  end

28  # Verzeichnis zunaechst auf aktuelles Arbeitsverzeichnis setzen
29  dir = "."
30  # Verzeichnis gegebenenfalls auf 1. Kommandozeilenargument setzen
31  dir = ARGV[0] if ARGV[0] != nil
32  # Erstaufruf mit dem aktuellen Verzeichnis
33  listdir(dir)
```

Wenn Sie das Skript ohne Kommandozeilenargument ausführen, wird das aktuelle Arbeitsverzeichnis aufgelistet. Hier ein (gekürztes) Beispiel für das Verzeichnis, in dem sich das Skript *listdir.rb* befindet:

```
> ruby listdir.rb
[.]
[..]
agecheck.rb
arg.rb
argnum.rb
listdir.rb
[old]
  [.]
  [..]
  comments.rb
  conv.rb
```

```
    crypt.rb
    doc.rb
  prim.rb
  read.rb
  rgb.rb
  x.txt
```

Wie Sie sehen, werden Verzeichnisnamen in eckigen Klammern dargestellt und die Inhalte von Unterverzeichnissen (hier nur eines) eingerückt. Nun zu den detaillierten Erläuterungen des Skripts:

- Zeile 4-27: Definition der Methode `listdir`. Beachten Sie zunächst, dass eine Methode in Ruby definiert werden muss, bevor sie zum ersten Mal aufgerufen werden kann (in manchen anderen Sprachen wird das freier gehandhabt). Deshalb steht die Methode vor dem globalen Code am Anfang des Skripts. `listdir` nimmt bis zu zwei Argumente entgegen: den Pflichtparameter `dir` (das zurzeit zu durchsuchende Verzeichnis) und die optionale Einrückungstiefe `indent`, die den Standardwert 0 (keine Einrückung) enthält, wenn sie beim Aufruf nicht angegeben wird.
- Zeile 6: Es wird ein neues `Dir`-Objekt erzeugt. Als Pfad wird die Parametervariable `dir` eingesetzt; die Referenzvariable heißt `d`.
- Zeile 8-24: In einer Schleife werden alle Einträge des aktuellen Verzeichnisses ausgelesen. Der Aufruf der Methode `read` dient dabei als Bedingung der `while`-Schleife, die dadurch automatisch ausgeführt wird, solange weitere Einträge vorhanden sind. Der jeweilige Eintrag wird in der Variablen `entry` gespeichert.
- Zeile 10: `dir` und `entry` werden zu einem neuen Pfad verknüpft, um den aktuellen Eintrag untersuchen zu können. Glücklicherweise akzeptiert auch Ruby für Windows den UNIX-Slash als Trennzeichen, so dass hier keine Plattform-Fallentscheidung nötig ist. Das Ergebnis der Operation wird in einer Variablen namens `path` abgelegt.
- Zeile 12: Die `if`-Abfrage prüft, ob der aktuelle Pfad `path` ein Verzeichnis ist. Dazu wird die oben besprochene Methode `File.directory?` auf den Pfad angewendet. Wenn der Eintrag ein Verzeichnis ist, werden die Zeilen 13-18 ausgeführt, ansonsten der `else`-Block in Zeile 20-22.
- Zeile 15: Der Verzeichnisname wird in eckigen Klammern ausgegeben. Zuvor wird das Leerzeichen mit `indent` »multipliziert«, um die korrekte Einrückungstiefe zu erzeugen.
- Zeile 18: Hier erfolgt der rekursive Aufruf von `listdir`. Als Parameter werden `path` (der komplette Pfad des aktuellen Unterverzeichnisses) und der um 2 erhöhte Wert von `indent` übergeben. Ein nachgestelltes `if` sorgt dafür, dass für die Verzeichnisse `.` und `..` kein Aufruf stattfindet – diese würden nämlich ansonsten zu einer Endlosschleife führen.

- Zeile 22: Hier wird der Name des Eintrags ohne eckige Klammern ausgegeben, wenn er kein Verzeichnis ist. Die Einrückung funktioniert wie bei den Verzeichnisnamen in Zeile 15.
- Zeile 26: Mit Hilfe der `Dir`-Methode `close` wird das Verzeichnis wieder geschlossen.
- Zeile 29: Hier ist die Methodendefinition bereits beendet; das ist tatsächlich die erste Anweisung, die ausgeführt wird. Sie weist einer Variablen namens `dir` den Wert `"."` zu, weil das aktuelle Arbeitsverzeichnis als Standardverzeichnis für die Funktion des Programms verwendet wird.
- Zeile 31: Wenn das erste Kommandozeilenargument `ARGV[0]` vorhanden ist, erhält `dir` stattdessen seinen Wert.
- Zeile 33: Hier erfolgt der Erstaufruf der Methode `listdir` mit dem Verzeichnis `dir` als Argument. Eine Einrückung wird nicht angegeben (das liefert den Standardwert 0), da die Einträge des obersten Verzeichnisses nicht eingerückt werden sollen.

Zugriff auf Attribute

In der Klasse `Gueterzug` aus dem Einführungsbeispiel oder in der Klasse `Rechteck` aus dem vorigen Kapitel haben Sie diverse Methoden gesehen, deren Name mit `get...` beginnt und die nichts weiter tun, als den Wert einer bestimmten Instanzvariablen zurückzugeben. Das widerspricht keineswegs dem Ansatz der Kapselung, da Sie frei entscheiden können, welche Eigenschaften Sie transparent machen möchten und welche nicht.

Noch erfreulicher ist aber, dass Sie diesen Zugriff über die so genannten *Getter-Methoden* in Ruby nicht unbedingt brauchen. Stattdessen gibt es eine sehr einfache Möglichkeit, die Werte bestimmter Attribute nach außen zu veröffentlichen. Schreiben Sie einfach Folgendes an den Anfang Ihrer Klassendefinition:

```
class Klassenname
  # Nach aussen lesbare Attribute
  attr_reader :attr1, :attr2, ...

  # ...
end
```

Statt `:attr1`, `:attr2` und so weiter müssen Sie die korrekten Namen Ihrer Instanzvariablen verwenden, nur eben als Symbole (mit Doppelpunkt statt mit `@` beginnend). Hier die entsprechende Kurzfassung der Klasse `Rechteck` aus dem vorigen Kapitel:

```
class EasyRechteck
  # Zu veroeffentlichende Attribute
  attr_reader :breite, :hoehe

  # Konstruktor
```

```
  def initialize(b, h)
    @breite = b
    @hoehe = h
  end

  # Methoden
  # get_breite und get_hoehe entfallen!

  # Die restlichen Methoden an die Ruby-Namenskonventionen anpassen:
  def flaeche
    @breite * @hoehe
  end

  def diagonale
    Math.sqrt(@breite**2 + @hoehe**2)
  end
end
```

In Klassen, die dieses nützliche Hilfsmittel verwenden, sollten Sie generell auf die Bezeichnung `get...` verzichten, um die Namen veröffentlichter Attribute und durch Methoden berechneter Werte zu vereinheitlichen. Im vorliegenden Beispiel wurden `get_flaeche` und `get_diagonale` daher in `flaeche` beziehungsweise `diagonale` umbenannt. Die neue Klasse können Sie beispielsweise folgendermaßen verwenden:

```
rechteck = EasyRechteck.new(20, 30)
printf "Breite:    %.3f\n", rechteck.b
printf "Hoehe:     %.3f\n", rechteck.h
printf "Flaeche:   %.3f\n", rechteck.flaeche
printf "Diagonale: %.3f\n", rechteck.diagonale
```

Das liefert folgende Ausgabe:

```
Breite:    20.000
Hoehe:     30.000
Flaeche:   600.000
Diagonale: 36.056
```

Analog lassen sich übrigens auch *Setter-Methoden* ersetzen, also Methoden, die den Wert eines bestimmten Attributs ändern. Dazu wird entsprechend die Anweisung `attr_writer` verwendet; anschließend können die freigeschalteten Attribute per einfacher Wertzuweisung mit `=` geändert werden. Probieren Sie die folgende Ableitung von `EasyRechteck` aus, bei der Breite und Höhe nachträglich änderbar sind (die Vererbung selbst wird weiter unten näher erläutert):

```
class VarEasyRechteck < Rechteck
  attr_writer :breite, :hoehe
end
```

Erzeugen Sie zum Testen eine `VarEasyRechteck`-Instanz und ändern Sie beispielsweise nachträglich ihre Breite:

```
r = VarEasyRechteck.new(10, 30)
printf "Flaeche vorher:  %d\n", r.flaeche
```

```
# Breite aendern
r.breite = 20
printf "Flaeche nachher: %d\n", r.flaeche
```

Hier das Ergebnis:

```
Flaeche vorher:  300
Flaeche nachher: 600
```

Mit `attr_writer` sollten Sie überaus vorsichtig und sparsam umgehen, denn die entsprechenden Attribute lassen sich von außen ungeprüft auf *jeden beliebigen Wert* ändern.

Eine sicherere, aber fast genauso praktische Alternative ist eine selbst definierte Methode, die den Namen eines Attributs, gefolgt von einem Gleichheitszeichen, trägt. Auf diese können Sie von außen genauso zugreifen wie auf einen automatisch generierten Setter, aber zusätzlich kann eine Wertkontrolle eingebaut werden.

Die nachfolgende Klasse kapselt ein rechtwinkliges Dreieck, indem sie einen der beiden anderen Winkel (`@alpha`) festlegt; der zweite (`beta`) ergibt sich durch die Operation `90-@alpha`. Damit dieser Winkel nicht zu groß wird, enthält der Konstruktor eine entsprechende Abfrage, wobei der Ersatzwert 89.9 willkürlich gewählt wurde – jeder Wert unter 90° ist zulässig. Die spezielle Methode `alpha=` ändert den Winkel nachträglich mit demselben Schutzmechanismus. Hier der Code:

```
class RechtwinkligesDreieck
  # Attribut lesbar machen
  attr_reader :alpha

  # Konstruktor
  def initialize(alpha=45)
    # Winkel beschraenken
    alpha = 89.9 if alpha >= 90
    @alpha = alpha
  end

  # Aenderungsmethode
  def alpha=(alpha)
    # Winkel beschraenken
    alpha = 89.9 if alpha >= 90
    @alpha = alpha
  end

  # Den anderen Winkel ermitteln
  def beta
    90 - @alpha
  end
end
```

Probieren Sie die Klasse aus, indem Sie eine Instanz erzeugen und den Winkel `alpha` nachträglich ändern. Zur Kontrolle können Sie jeweils die Methode `beta` aufrufen. Zum Beispiel:

```
d = RechtwinkligesDreieck.new(30)
printf "Alpha: %d, Beta: %d\n", d.alpha, d.beta
# Alpha aendern
d.alpha = 40
printf "Alpha: %d, Beta: %d\n", d.alpha, d.beta
```

Die Ausgabe lautet natürlich:

```
Alpha: 30, Beta: 60
Alpha: 40, Beta: 50
```

Eine Kurzfassung für alle Attribute, die sowohl lesbar als auch änderbar sein sollen, ist `attr_accessor` (*Accessor-Methoden* ist übrigens der traditionelle Oberbegriff für Getter und Setter). Hier als Beispiel eine Klasse, die alle Informationen über einen Song kapselt[6] – Interpret, Titel und Laufzeit in Sekunden. Da alle diese Daten unkritisch änderbar sind, werden sie mittels `attr_accessor` freigegeben:

```
class Song
  # Attribute veroeffentlichen
  attr_accessor :interpret, :titel, :dauer

  # Konstruktor
  def initialize(interpret="", titel="", dauer=0)
    @interpret = interpret
    @titel = titel
    @dauer = dauer.to_i
  end

  # Dauer in Minuten
  def minuten
    sprintf "%d:%02d", (@dauer / 60).floor, @dauer % 60
  end

  # Alle Infos ueber den Song als String
  def to_s
    sprintf "%s: %s (%s)", @interpret, @titel, minuten
  end
end
```

Zum einfachen Ausprobieren können Sie die Datei mit der Klasse in `irb` importieren und dann interaktiv mit der Klasse spielen. Hier ein komplettes Beispiel einer solchen »Sitzung«:

```
>> require "song.rb"
=> true
>> lied1 = Song.new("Metallica", "One", 444)
=> #<Song:0x2b18bac @dauer=444, @titel="One", @interpret="Metallica">
```

6 Inspiriert durch ein ähnliches Praxisbeispiel im »Pickaxe Book«.

```
>> puts lied1
Metallica: One (7:24)
>> lied2 = Song.new("Red Hot Chili Peppers", "Snow (Hey Oh)")
=> #<Song:0x2b1437c @dauer=0, @titel="Snow (Hey Oh)",
@interpret="Red Hot Chili Peppers">
>> puts lied2
Red Hot Chili Peppers: Snow (Hey Oh) (0:00)
>> lied2.dauer = 334
=> 334
>> puts lied2
Red Hot Chili Peppers: Snow (Hey Oh) (5:34)
```

Schauen Sie sich nebenbei noch kurz die Methode `minuten` an. Sie setzt die Dauer in Minuten aus dem ganzzahligen Anteil der durch 60 geteilten Sekunden, einem Doppelpunkt und dem Rest dieser Division zusammen. Für Letzteren wird das Format `%02d` verwendet, das zwei Stellen und eventuell eine führende Null garantiert. Als Methode wird `sprintf` und nicht `printf` verwendet, da die Methode nicht etwa eine Ausgabe durchführen, sondern einen String-Ausdruck zurückliefern soll.

Letzteres gilt analog für die Methode `to_s`. Eine Methode dieses Namens hat, wie bereits erwähnt, noch einen Vorteil: Sie wird automatisch aufgerufen, wenn die Instanz in einem String-Kontext verwendet wird – hier beispielsweise in den `irb`-Zeilen, in denen die Instanzen `lied1` und `lied2` als Argumente von `puts` fungieren.

Weitere objektorientierte Konstrukte

Nachdem Sie nun über die Definition einzelner Klassen und den Einsatz entsprechender Instanzen Bescheid wissen, können Sie sich mit der Zusammenarbeit von verschiedenen Klassen beschäftigen. In diesem Abschnitt erfahren Sie daher das Wichtigste über die Themen Vererbung, Geheimhaltungsstufen (öffentliche oder private Methoden), Module und Mixins (die gleichnamige Klassenmethoden und gemeinschaftliche Instanzmethoden verschiedener Klassen ermöglichen) sowie Introspektion (die »Live-Untersuchung« von Klassenstrukturen).

Vererbung

Die *Vererbung* (Inheritance) zählt zu den bedeutendsten Vorteilen der Objektorientierung: Indem Sie Gemeinsamkeiten in übergeordneten Klassen verallgemeinern und Unterschiede in abgeleiteten Klassen herausarbeiten, sparen Sie sich häufig doppelte Arbeit und machen Ihre Programme übersichtlicher.

Die Vererbung wird durchgeführt, indem Sie in einer Klassendefinition ein `<`-Zeichen und den Namen der gewünschten Elternklasse hinter den Klassennamen schreiben. Dabei können Sie Ihre Klassen sowohl von Ruby-Standardklassen als auch von Ihren eigenen Klassen ableiten.

Das folgende einfache Beispiel definiert eine Elternklasse namens `Parent` und leitet eine Klasse namens `Child` davon ab:

```
class Parent

  # Attribut veroeffentlichen
  attr_accessor :wert

  # Konstruktor
  def initialize(wert=0)
    @wert = wert
  end

end


Class Child < Parent

  # Zusaetzliches Attribut
  attr_accessor :wert2

  # Konstruktor
  def initialize(wert=0, wert2=0)
    # Konstruktor der Elternklasse aufrufen
    super(wert)
    @wert2 = wert2
  end

end
```

Inhaltlich sind diese beiden Klassen recht sinnlos: `Parent` kapselt einen beliebigen Wert, `Child` fügt einen zweiten hinzu. Im Konstruktor von `Child` wird der Konstruktor von `Parent` mit Hilfe des Schlüsselworts `super` aufgerufen. Zum Ausprobieren der Vererbung ist das Beispiel jedoch gut geeignet. Speichern Sie beide Klassen als *inherit1.rb*, importieren Sie sie in `irb` und spielen Sie ein bisschen damit:

```
>> require "inherit1.rb"
=> true
>> p = Parent.new(7)
=> #<Parent:0x2b19f20 @wert=7>
>> p.wert = 12
=> 12
>> c = Child.new(8, 7)
=> #<Child:0x2b1667c @wert=8, @wert2=7>
>> c.wert = 5
=> 5
>> c.wert2 = 9
=> 9
```

Wie Sie sehen, hat die `Child`-Instanz `c` sowohl Zugriff auf den in ihrer eigenen Klasse definierten Accessor `wert2` als auch auf `wert` aus der Elternklasse `Parent`. Umgekehrt besitzt ein reines `Parent`-Objekt dagegen nicht die zusätzlichen Eigenschaften, die in

der Klasse `Child` definiert werden. Ein entsprechender Zugriffsversuch liefert die Fehlermeldung, dass die betreffende Methode nicht verfügbar sei:

```
>> p.wert2
NoMethodError: undefined method `wert2' for
#<Parent:0x2b19f20 @wert=12>
```

Natürlich können Sie nicht nur von eigenen, sondern auch von eingebauten Klassen neue Klassen ableiten. Das folgende Beispiel leitet eine Klasse von der im vorigen Kapitel behandelten Standardklasse `Time` ab, um diese durch Methoden für die deutsche Datums- und Uhrzeitformatierung zu erweitern:

```
class DTime < Time

  # Konstanten-Arrays
  WTAGE = %w(Sonntag Montag Dienstag Mittwoch
             Donnerstag Freitag Samstag)
  MONATE = %w(Januar Februar Maerz April Mai Juni Juli
              August September Oktober November Dezember)

  # Deutschsprachiger Wochentag
  def wtag
    t = self.wday
    WTAGE[t]
  end

  # Deutschsprachiger Monat
  def monat
    m = self.month
    MONATE[m-1]
  end

  # Ausfuehrliches deutschsprachiges Datum
  def datum
    self.strftime("#{self.wtag}, %d. #{self.monat} %Y, %H:%M:%S")
  end

  # Kurzes "Euro-Datum"
  def kurzdatum
    self.strftime("%d.%m.%Y, %H:%M:%S")
  end

end
```

Probieren Sie die neuen Methoden in `irb` aus, nachdem Sie das Skript mit der Klassendefinition importiert haben:

```
>> require "dtime.rb"
=> true
>> DTime.new.wtag
=> "Montag"
>> DTime.new.monat
=> "November"
>> DTime.new.datum
```

```
=> "Montag, 13. November 2006, 13:53:21"
>> DTime.new.kurzdatum
=> "13.11.2006, 13:53:26"
```

Beachten Sie, dass Sie alternativ auch immer die vorhandene Klasse selbst erweitern können, indem Sie einen `class`-Block mit den gewünschten Änderungen oder Ergänzungen für sie schreiben. Das wurde weiter oben beispielsweise für die Klasse `Array` gezeigt, die um die nützlichen Methoden `sum` und `average` erweitert wurde.

Zugriffsschutz

In Ruby sind alle Methoden standardmäßig öffentlich. Das bedeutet, dass sie für jede Instanz einer Klasse von außen zugänglich sind. Das lässt sich bei Bedarf ändern, wenn das Design einer Klassenhierarchie verlangt, dass bestimmte Methoden die Öffentlichkeit nichts angehen. Zu diesem Zweck definiert Ruby drei *Geheimhaltungs-* oder *Sichtbarkeitsstufen*:

- `public` ist der Standardfall: Eine Methode mit dieser Sichtbarkeit steht innerhalb der Klassendefinition und überall dort zur Verfügung, wo eine Instanz der Klasse verwendet werden kann.
- `private` ist das andere Extrem: Methoden mit dieser Geheimhaltungsstufe sind ausschließlich innerhalb der Klassendefinition selbst verfügbar.
- `protected` ermöglicht den Zugriff für andere Instanzen derselben Klasse sowie für abgeleitete Klassen und ihre Instanzen.

Sie ändern die Sichtbarkeit von Methoden, indem Sie das entsprechende Schlüsselwort in eine eigene Zeile Ihrer Klassendefinition setzen. Von da an besitzen alle nachfolgenden Methoden diese Geheimhaltungsstufe, bis Sie sie wieder ändern.

Machen Sie zunächst den folgenden einfachen Versuch mit `private`:

```
class PrivatTest
  private
  def privat
    puts "Dies ist privat"
  end

  public
  def oeffentlich
    print "Ganz unter uns: "
    privat
  end
end
```

Die Klasse `PrivatTest` besitzt zwei Methoden. `privat` hat die Sichtbarkeit `private`, während `oeffentlich` explizit als `public` gekennzeichnet wurde. Innerhalb von `oeffentlich` wird die Methode `privat` ohne Instanzbezug aufgerufen. Erzeugen Sie nun eine Instanz der Klasse und testen Sie beide Methoden. Beim Aufruf von `privat` erhalten Sie eine Fehlermeldung:

```
>> pt = PrivatTest.new
=> #<PrivatTest:0x2b058f4>
>> pt.privat
NoMethodError: private method `privat' called for
#<PrivatTest:0x2b058f4>
```

Der Aufruf von `oeffentlich` verläuft jedoch problemlos und offenbart so indirekt auch das Verhalten von `privat`:

```
>> pt.oeffentlich
Ganz unter uns: Dies ist privat
```

Private Methoden erwecken den Eindruck, etwas Ähnliches zu sein wie Klassenmethoden. Dieser Eindruck täuscht allerdings: Klassenmethoden können durchaus von außen aufgerufen werden, beziehen sich aber nicht auf eine konkrete Instanz. Private Methoden sind dagegen Instanzmethoden, die jedoch nicht von außen aufgerufen werden können.

Es gibt noch eine andere Aufrufmöglichkeit für `public`, `private` und `protected`: Sie können ihnen ein oder mehrere Symbole als Argumente übergeben, um nur den angegebenen Methoden die entsprechende Sichtbarkeit zuzuweisen. Das ist vor allem nützlich, um die Geheimhaltungsstufe von Methoden zu modifizieren, die aus der Elternklasse übernommen wurden. Das obige Beispiel könnten Sie auf diese Weise wie folgt umschreiben:

```
class PrivatTest
  def privat
    puts "Dies ist privat"
  end

  def oeffentlich
    print "Ganz unter uns: "
    privat
  end

  private :privat

end
```

Es kann nützlich sein, private Methoden zu schreiben, um sich um die »inneren Angelegenheiten« einer Instanz zu kümmern. Stellen Sie sich beispielsweise vor, ein Händler bietet für seine Waren eine Ratenkreditfinanzierung an. Der Kredit soll gewährt werden, falls das Monatsgehalt des Kunden mindestens ein Viertel des Kaufpreises beträgt. Das genaue Gehalt geht den Verkäufer nichts an, aber dennoch muss er entscheiden können, ob der Kunde den Kredit erhält. Zu diesem Zweck könnte man eine private Methode schreiben, die das Gehalt mit vier multipliziert. Sie wird hinter den Kulissen von einer öffentlichen Methode aufgerufen, die nur `true` oder `false` zurückgibt. Die Klasse `Kunde` sähe in diesem Fall so aus:

```
class Kunde
  def initialize(gehalt)
    @gehalt = gehalt
  end

  # Die maximale Kreditsumme ist geheim
  private
  def max_kredit
    @gehalt * 4
  end

  # Oeffentlich ist nur, ob ein bestimmter Kredit gewaehrt werden kann
  public
  def kredit_ok(summe)
    if summe <= max_kredit
      true
    else
      false
    end
  end
end
```

Probieren Sie die Klasse nun aus:

```
mueller = Kunde.new(2500)
neuwagen = 26000
gebrauchtwagen = 8000
if mueller.kredit_ok(neuwagen)
  puts "Glueckwunsch! Wir koennen Ihren Neuwagen finanzieren!"
else
  puts "Der Neuwagen sprengt leider den Rahmen."
end
if mueller.kredit_ok(gebrauchtwagen)
  puts "Glueckwunsch! Wir koennen Ihren Gebrauchtwagen finanzieren!"
else
  puts "Der Gebrauchtwagen sprengt leider den Rahmen."
end
```

Die Ausgabe lautet:

```
Der Neuwagen sprengt leider den Rahmen.
Glueckwunsch! Wir koennen Ihren Gebrauchtwagen finanzieren!
```

Ein direkter Aufruf der Methode `max_kredit` von außen scheitert dagegen erwartungsgemäß:

```
>> mueller.max_kredit
NoMethodError: private method `max_kredit' called for
#<Kunde:0x2b09ea4 @gehalt=2500>
```

Die Geheimhaltungsstufe `private` wird im Übrigen sehr häufig verwendet, um die unkontrollierte Instanziierung einer Klasse zu verhindern. Bestimmte Ressourcen wie etwa eine Druckerwarteschlange oder den Zugriff auf eine Logdatei darf es nur ein einziges Mal geben, und jeder Bezug darauf muss somit dasselbe Objekt liefern.

Es folgt ein Beispiel, das ein einfaches Logbuch realisiert. Jeder neue Eintrag wird mit einem Zeitstempel versehen und mittels `unshift` an den Anfang eines Arrays gestellt. Die Instanzbildung wird verboten, indem `new` (und nicht etwa `initialize`) für privat erklärt wird. Beachten Sie in diesem Zusammenhang, dass `new` eine Klassen- und keine Instanzmethode ist (denn eine Instanz entsteht erst durch ihren Aufruf!), so dass `private_class_method` verwendet werden muss. Die einzige Instanz wird intern als Klassenvariable `@@logbuch` gespeichert und durch einen Aufruf von `instance` zurückgegeben. Hier der komplette Code für diese Klasse:

```
class Logbuch
  # Instanziierung verhindern
  private_class_method :new

  def initialize
    @log = Array.new
  end

  # Klassenvariable fuer die einzige Instanz
  @@logbuch = nil

  # Methode zur Instanzrueckgabe
  def Logbuch.instance
    if @@logbuch
      @@logbuch
    else
      @@logbuch = new
    end
  end

  def log(eintrag)
    logeintrag = Time.new.strftime("%d.%m.%Y, %H:%M:%S -- ")
    logeintrag += eintrag
    @log.unshift(logeintrag)
  end
end
```

Stellen Sie die Funktionalität auf die Probe. Importieren Sie `logbuch.rb` in `irb` und versuchen Sie zunächst wider besseres Wissen, `new` aufzurufen – dies misslingt:

```
>> log1 = Logbuch.new
NoMethodError: private method `new' called for Logbuch:Class
```

Verwenden Sie stattdessen `instance`, um Zugriff auf die Instanz zu erhalten – am besten zweimal, dann merken Sie an der Objekt-ID, dass Sie tatsächlich dieselbe Instanz erhalten:

```
>> log1 = Logbuch.instance
=> #<Logbuch:0x2b116e0 @log=[]>
>> log2 = Logbuch.instance
=> #<Logbuch:0x2b116e0 @log=[]>
```

Nun können Sie die Methode log (sowohl von log1 als auch von log2) verwenden, um Einträge vorzunehmen. Der jeweils neueste Eintrag wird an den Anfang gestellt. Zum Beispiel:

```
>> log1.log("Computerlogbuch der Enterprise, Captain Picard.")
=> ["01.12.2006, 22:39:12 -- Computerlogbuch der Enterprise,
Captain Picard."]
>> log2.log("Befinden uns im Standardorbit um den Planeten Haven.")
=> ["01.12.2006, 22:39:26 -- Befinden uns im Standardorbit um den
Planeten Haven.", "01.12.2006, 22:39:12 -- Computerlogbuch der
Enterprise, Captain Picard."]
>> log1.log("Ein Erkundungstrupp unter der Leitung von Lt.Cmdr. Data
wird hinuntergebeamt, um Untersuchungen vorzunehmen.")
=> ["01.12.2006, 22:39:54 -- Ein Erkundungstrupp unter der Leitung
von Lt.Cmdr. Data wird hinuntergebeamt, um Untersuchungen vorzunehmen.",
"01.12.2006, 22:39:26 -- Befinden uns im Standardorbit um den
Planeten Haven.", "01.12.2006, 22:39:12 -- Computerlogbuch der
Enterprise, Captain Picard."]
```

Dieses Verfahren, das die Existenz nur einer Instanz garantiert, wird übrigens recht häufig verwendet. Es hat sogar einen Namen: *Singleton*. Solche immer wiederkehrenden Programmierschemata werden als *Design Patterns* oder *Entwurfsmuster* bezeichnet. Sie sind sehr hilfreich, wenn es darum geht, häufig auftretende Probleme zu lösen – es besteht die Hoffnung, dass Sie das Rad nicht neu zu erfinden brauchen, weil sich bereits jemand darum gekümmert und seine Erfahrungen in einem Musterkatalog abgelegt hat. Wenn Sie die Arbeit mit Design Patterns erlernen möchten,[7] sollten Sie sich das hervorragende Buch *Entwurfsmuster von Kopf bis Fuß* (O'Reilly Verlag) besorgen. Dort wird das Thema gründlich und überaus unterhaltsam präsentiert.

Ein wenig schwieriger sind Funktionsweise und Bedeutung von protected zu vermitteln, denn Methoden mit dieser Geheimhaltungsstufe sind für *beliebige* Instanzen der eigenen Klasse und abgeleiteter Instanzen sichtbar. Sie sind also immer dann sinnvoll, wenn innerhalb einer Klassendefinition auf eine Instanzmethode einer anderen Instanz derselben Klasse zugegriffen werden muss, ohne dass die restliche »Öffentlichkeit« das auch dürfte.

Module und Mixins

Wenn Sie eine rein statische Funktionalität unter einem gemeinsamen Dach zusammenfassen möchten, steht es Ihnen frei, eine Klasse mit Konstanten und Klassenmethoden zu definieren. Allerdings könnte jemand auf die Idee kommen, (nutzlose) Instanzen von dieser Klasse zu bilden oder Kindklassen von ihr abzuleiten. Deshalb

7 Das kann man allen, die objektorientiert programmieren, nur empfehlen, weil es jede Menge Arbeit spart.

bietet Ruby auch ein offizielles Hilfsmittel für diese Aufgabe: *Module*. Ein Modul ähnelt einer Klasse – außer dass es weder instanziiert noch vererbt werden kann. Eines der bekanntesten Module aus dem Lieferumfang von Ruby ist `Math` (siehe Kapitel 2).

Das folgende Modul implementiert eine Konstante und zwei Klassenmethoden zum Umrechnen von DM in Euro und umgekehrt – nützlich für die kurz vor Weihnachten noch immer beliebten »Zahl mit DM«-Aktionen des Einzelhandels:

```
module Waehrung
  DM = 1.95583

  def Waehrung.dm_in_euro(dm)
    dm / DM
  end

  def Waehrung.euro_in_dm(euro)
    euro * DM
  end
end
```

Importieren Sie die Datei mit dem Modul, *waehrung.rb*, in `irb` und probieren Sie es aus:

```
>> require "waehrung.rb"
=> true
>> Waehrung::DM
=> 1.95583
>> Waehrung.dm_in_euro(7)
=> 3.57904316837353
>> Waehrung.euro_in_dm(9)
=> 17.60247
```

Ein nützlicher Nebeneffekt von Modulen ist, dass sie einen *Namensraum* definieren – das Voranstellen des Modulnamens ermöglicht allgemeine Methoden, die den gleichen Namen tragen, aber unterschiedliche Aufgaben erfüllen. Hier ein plakatives Beispiel, das mit der Doppeldeutigkeit des Wortes »Platz« spielt:

```
module Hund
  def Hund.platz(name)
    puts "#{name} hat sich hingelegt."
  end
end

module Tuete
  def Tuete.platz
    puts "*Peng*"
  end
end

Hund.platz("Bello")
Tuete.platz
```

Das liefert die Ausgabe:

```
Bello hat sich hingelegt.
*Peng*
```

Hintergrundwissen: Typisierte und untypisierte Sprachen

Nur wenige objektorientierte Programmiersprachen, beispielsweise C++, bieten die Möglichkeit der *Mehrfachvererbung*, das heißt die Ableitung einer Klasse von mehr als einer Elternklasse gleichzeitig. Da dies zu Problemen führen kann, etwa bei gleichnamigen Methoden, wurde dieses Feature gerade bei modernen OO-Sprachen weggelassen oder durch andere Konzepte ersetzt. Java verwendet beispielsweise *Interfaces*, die eine Klasse einem zusätzlichen Typ zuordnen, dafür aber verlangen, dass sie alle ihre Methoden implementiert (denn die Methodendeklarationen eines Java-Interfaces sind leer).

Das Hauptproblem, das die C++-Mehrfachvererbung oder die Interfaces in Java lösen, ist die Festlegung eines *gemeinsamen Datentyps* für Instanzen unterschiedlicher Klassen, damit diese Instanzen beispielsweise von ein und derselben Methode als Argument akzeptiert werden. Hier zur Veranschaulichung eine typische Java-Methode:

```
public String info(Auto a) {
  return "Das Auto ist " + a.getFarbe() + " und faehrt maximal " +
         a.getHoechstgeschwindigkeit() + " km/h."
}
```

Wie Sie sehen, besitzt in typisierten Sprachen wie Java alles einen festgelegten Datentyp: Die Methode `info()` liefert einen String zurück, also ist ihr Typ `String`, und sie liefert Informationen über ein Auto, so dass ihr Parameter eine Instanz von `Auto` sein muss. Soll `info()` nun auch Informationen über Boote liefern, so müssen `Auto` und `Boot` entweder eine gemeinsame Elternklasse haben oder eben ein Interface implementieren, das die gemeinsamen Methoden `getFarbe()` und `getHoechstgeschwindigkeit()` vorgibt. Der Datentyp des Parameters von `info()` wird dann auf diesen gemeinsamen Typ geändert, und es funktioniert wieder.

Ein anderer Nebeneffekt der Typisierung ist übrigens die so genannte *Polymorphie*: Eine Methode mit demselben Namen kann mehrfach definiert werden, wenn die Anzahl oder Datentypen der Parameter unterschiedlich sind.

In untypisierten Sprachen wie Ruby ist das alles nicht nötig: Da Variablen keinen ein für alle Mal festgelegten Datentyp haben, kann eine Methode einfach dynamisch ermitteln, wie viele Argumente sie erhalten hat (das wurde im Abschnitt »Parameter« auf Seite 170 erläutert), und auch für die Wertrückgabe ist kein bestimmter Typ vorgegeben.

Ruby kennt keine Mehrfachvererbung (siehe den Kasten »Hintergrundwissen: Typisierte und untypisierte Sprachen« auf Seite 207). Trotzdem kann es manchmal nützlich sein, dieselbe Methode für verschiedene, nicht verwandte Klassen bereitzustellen. Ruby kennt einen eleganten Weg dafür: den Import eines Moduls als so genanntes *Mixin*. Dazu muss das Modul eine (zunächst widersinnig erscheinende, da in sich nicht aufrufbare) Instanzmethode definieren. Nach dem Import steht diese Methode in einer Klasse zur Verfügung wie jede Ihrer eigenen Instanzmethoden.

Das folgende Beispiel definiert das Modul `Zeit` mit einer Instanzmethode namens `zeit`, die Datum und Uhrzeit in deutscher Formatierung liefert:

```
module Zeit
  def zeit
    Time.new.strftime("%d.%m.%Y, %H:%M")
  end
end
```

Dieses Modul können Sie nun mittels `include` in beliebige Klassen importieren. Zum Beispiel:

```
class LogEintrag
  include Zeit

  attr_reader :ereignis

  def initialize(text)
    @ereignis = sprintf("%s: %s", zeit, text)
  end
end
```

Testen Sie es – die importierte Methode wird sowohl intern als auch extern korrekt aufgerufen:

```
>> l = LogEintrag.new("Festplatte partitionieren")
=> #<LogEintrag:0x2b86d50
     @ereignis="14.11.2006, 14:38: Festplatte partitionieren">
>> l.ereignis
=> "14.11.2006, 14:38: Festplatte partitionieren"
>> l.zeit
=> "14.11.2006, 14:39"
```

Natürlich steht die Mixin-Methode auch in abgeleiteten Klassen zur Verfügung. Hier sehen Sie auch dafür ein Beispiel:

```
class LogToDo < LogEintrag
  attr_accessor :status

  def initialize(text, status="erledigt")
    super(text)
    @status = status
  end
end
```

Hier das Ergebnis des entsprechenden irb-Tests:

```
>> l2 = LogToDo.new("Backup einspielen", "nach Erledigung pruefen")
=> #<LogToDo:0x2b4641c
     @ereignis="14.11.2006, 14:47: Backup einspielen",
     @status="nach Erledigung pruefen">
>> l2.zeit
=> "28.11.2006, 14:47"
```

Neben selbst geschriebenen Modulen bietet Ruby auch einige nützliche Standardmodule, die Sie importieren können. Eines der praktischsten ist `Comparable`: Wenn Sie es importieren und in der Methode <=> die Ordnung von Objekten Ihrer Klasse festlegen, stehen auf einen Schlag sämtliche Vergleichsoperatoren wie `==`, `!=`, `<` und > zur Verfügung. Das folgende Beispiel erweitert eine Ableitung der Klasse `Lok` aus dem Einführungsbeispiel um eine solche Vergleichsmöglichkeit, wobei natürlich die Anzahl der möglichen Waggons als Kriterium dient:

```
class BessereLok < Lok
  # Das Modul Comparable importieren
  include Comparable

  # Die Methode <=> zur Verfuegung stellen
  def <=>(andereLok)
    @typ <=> andereLok.get_kapazitaet
  end
end
```

Auch das können Sie nun testen:

```
>> BessereLok.new(Lok::STROM) > BessereLok.new(Lok::DAMPF)
=> true
>> BessereLok.new(Lok::STROM) > BessereLok.new(Lok::DIESEL)
=> false
```

Exceptions

In objektorientierten Programmen mit ihren ineinander verschachtelten Klassen- und Objekthierarchien ist es nützlich, wenn sich Programmfehler bis zu ihrem Ursprung zurückverfolgen lassen. Dafür steht in Ruby – wie in verschiedenen anderen Sprachen – das System der *Exceptions* (Ausnahmen) zur Verfügung. Sie können solche speziellen Fehler in Ihren Klassen sowohl auslösen als auch abfangen.

Hier zunächst ein einfaches Beispiel für das Abfangen einer Exception: Eine Datei soll zum Lesen geöffnet werden, anschließend wird eine Zeile daraus gelesen, und die Datei wird wieder geschlossen. Falls die Datei nicht vorhanden ist, soll eine benutzerfreundliche Fehlermeldung ausgegeben werden. Der entsprechende Code sieht so aus:

```
filename = "nichtda.txt"
begin
  f = File.open(filename, "r")
```

```
  text = f.gets
  f.close
rescue
  STDERR.puts "Datei #{filename} kann leider nicht geoeffnet werden."
end
```

Das Schema ist immer dasselbe: Setzen Sie die Codezeilen mit Fehlermöglichkeit in einen Block, der mit `begin` anfängt. Hinter dem Schlüsselwort `rescue` steht der Code, der beim Auftreten eines Fehlers ausgeführt werden soll. Wenn nötig, können Sie vor dem abschließenden `end` noch das Schlüsselwort `ensure` einfügen. Der darin befindliche Code wird auf jeden Fall ausgeführt – ob der Fehler auftritt oder nicht.

Hinter dem Schlüsselwort `rescue` können Sie optional eine Fehlerklasse nennen. Dadurch können sogar mehrere `rescue`-Abschnitte angegeben werden, um verschiedene Arten von Fehlern unterschiedlich zu behandeln. Das folgende Beispiel unterscheidet zwischen einer nicht vorhandenen Methode (`NoMethodError`) und allen anderen Arten von Fehlern (`StandardError`):

```
begin
  pots "Hallo!"         # Absichtlich falsch!
  f = File.new          # Ebenfalls absichtlich falsch!
rescue NoMethodError
  puts "Unbekannte Methode: #{$!}"
rescue StandardError
  puts "Anderer Fehler: #{$!}"
end
```

Bei der ersten Ausführung lautet die Fehlermeldung:

```
Unbekannte Methode: undefined method `pots' for main:Object
```

Wenn Sie `pots` durch die korrekte Schreibweise `puts` ersetzen, erhalten Sie jedoch diese Fehlermeldung:

```
Anderer Fehler: wrong number of arguments (0 for 1)
```

Die spezielle Variable `$!` enthält jeweils den Text der offiziellen Fehlermeldung.

Wenn Sie derartigen Code in eine Klasse einbauen, können sogar diejenigen Arten von Fehlern, um die Sie sich nicht selbst kümmern möchten oder können, können Sie als letzte Anweisung vor `ensure` beziehungsweise `end` ein `raise` hinzufügen. Es reicht alle anderen Fehlerarten nach außen weiter, das heißt an die Stelle, wo eine Instanz der Klasse verwendet wird.

Um in Ihren Klassen eigene Arten von Fehlern zu erzeugen, erweitern Sie einfach `Exception`, den Urahn aller Fehlerklassen (oder gegebenenfalls eine speziellere Klasse, die besser passt). Eine Meldung, die in `$!` übernommen wird, können Sie

einbauen, indem Sie eine to_s-Methode schreiben. Ausgeworfen wird Ihre Exception mit raise.

Betrachten Sie beispielsweise die folgende Ableitung der Klasse Lok, die keine beliebigen maximalen Waggonzahlen mehr akzeptiert, sondern nur noch die offiziellen Loktypen. Andernfalls erzeugt sie bei der Instanziierung eine IllegaleLok-Exception, die zuvor definiert wird:

```
# Einen benutzerdefinierten Fehler definieren
class IllegaleLok < Exception
  # Die Fehlermeldung festlegen
  def to_s
    "Dieser Loktyp existiert nicht"
  end
end

# Erweiterung der Klasse Lok
class ExakteLok < Lok
  # Konstruktor ueberschreiben
  def initialize(typ = OHNE)
    # Nur offizielle Typen zulassen
    if typ != OHNE && typ != DAMPF && typ != DIESEL && typ != STROM
      raise IllegaleLok
    else
      @typ = typ
    end
  end
end
```

Testen Sie die neue Klasse wie folgt:

```
# Erster Versuch: passender Typ
begin
  e1 = ExakteLok.new(ExakteLok::STROM)
  puts e1.get_typ
rescue IllegaleLok
  puts $!
end

# Zweiter Versuch: unzulaessiger Typ
begin
  e2 = ExakteLok.new(35)
  puts e2.get_typ
rescue IllegaleLok
  puts $!
end
```

Während bei e1 der Typ »Elektrolok« ausgegeben wird, erhalten Sie für e2 die folgende Fehlermeldung:

```
Dieser Loktyp existiert nicht
```

Introspektion

Ein interessantes Feature vieler objektorientierter Programmiersprachen ist die Möglichkeit, *zur Laufzeit* (d.h. während ein Skript ausgeführt wird) die Bestandteile von Klassen und Objekten zu untersuchen. Gerade für eine untypisierte Sprache wie Ruby ist es praktisch, dass Sie jederzeit ermitteln können, welcher Klasse ein Objekt angehört, auf welche Methoden es reagiert oder wie seine Ahnenreihe bis hinauf zu `Object` aussieht. Eine solche Fähigkeit wird als *Introspektion* oder *Reflexion* bezeichnet – auf Deutsch könnte man auch »Nabelschau« dazu sagen.

Die einfachste Introspektionsmethode haben Sie bereits kennengelernt:

```
Objekt.class
```

gibt die Klasse zurück, der ein beliebiges Objekt – ein Literal, ein Ausdruck oder eine Variable – angehört. Entsprechende Beispiele haben Sie schon in Kapitel 2 gesehen.

Interessant ist das Modul `ObjectSpace`, dessen Methoden Auskunft über sämtliche Objekte des aktuellen Ausführungskontextes, das heißt der Ruby-Umgebung plus Ihrem Programm, geben. Ein Beispiel ist der Iterator `each_object`, der nacheinander jedes einzelne Objekt bereitstellt. Er nimmt optional einen Klassennamen als Argument entgegen und iteriert dann nur über die Instanzen der angegebenen Klasse sowie der von ihr abgeleiteten Klassen. Nach der Ausführung wird die Anzahl der passenden Objekte als Ergebnis zurückgeliefert.

Das folgende Beispiel liefert eine formatierte Liste aller Objekte vom Typ `Numeric`, ihrer konkreten Klassen (`n.class`) sowie ihrer eindeutigen Objekt-IDs (`n.object_id`) und gibt anschließend deren Gesamtzahl aus:

```
puts " Objekt               | Klasse           | Objekt-ID"
puts "----------------------+------------------+-----------"
arr = [0.2, 2.2, 3.2, 4.2, 5.2]
num = ObjectSpace.each_object(Numeric) { |n|
  printf " %-20g | %-16s | %d\n", n, n.class, n.object_id
}
puts
puts "Es gibt #{num} numerische Objekte."
```

Hier die Ausgabe des Beispiels:

```
 Objekt               | Klasse           | Objekt-ID
----------------------+------------------+-----------
 2.71828              | Float            | 21237020
 3.14159              | Float            | 21237030
 2.22045e-016         | Float            | 21257670
 1.79769e+308         | Float            | 21257680
 2.22507e-308         | Float            | 21257690
 5.2                  | Float            | 21313340
 4.2                  | Float            | 21313370
 3.2                  | Float            | 21313400
```

```
2.2                  | Float              | 21313430
0.2                  | Float              | 21313480
100                  | Float              | 21341780

Es gibt 11 numerische Objekte.
```

Die Syntax des verwendeten `printf`-Formatstrings können Sie übrigens in Kapitel 2 nachschlagen; der Rest des Codes müsste ohne Weiteres verständlich sein.

Neben den selbst definierten Fließkommazahlen aus dem Array `arr` werden einige vordefinierte Zahlen aufgelistet: π (3.14159), e (2.71828), die kleinste von 0 verschiedene Zahl (2.22045e-016) sowie die absolut größte (1.79769e+308) und absolut kleinste (2.22507e-308) Fließkommazahl. Wie Sie vielleicht bemerkt haben, werden keine Instanzen der Ganzzahl-Klasse `Fixnum` aufgelistet. Diese sind – genau wie `true`, `false` und `nil` – statische Objekte, über die `ObjectSpace` keine Informationen enthält. Das merken Sie daran, dass die Objekt-ID beim selben Wert stets identisch ist. Testen Sie es in `irb`:

```
>> a = 2
>> b = 2
>> a.object_id
=> 5
>> b.object_id
=> 5
```

Machen Sie auch die Gegenprobe mit zwei gleichwertigen Fließkommazahlen – hier sind die IDs unterschiedlich:

```
>> c = 4.5
>> d = 4.5
>> c.object_id
=> 22570900
>> d.object_id
=> 22567520
```

Sie können auch ermitteln, ob ein Objekt zu einer bestimmten Klasse gehört. Dafür besitzen alle Instanzen von `Object` – das heißt sämtliche Ruby-Objekte – die beiden Methoden `instance_of?` und `kind_of?`. Der kleine, aber manchmal wichtige Unterschied: `instance_of?` liefert nur dann `true`, wenn das untersuchte Objekt eine Instanz von genau der angegebenen Klasse ist, während `kind_of?` auch auf die Elternklasse und beliebige weitere übergeordnete Klassen zutrifft. Ein Synonym für `kind_of?` ist übrigens `is_a?`, was der Bezeichnungsweise im zu Beginn dieses Kapitels besprochenen objektorientierten Design näher kommt.

Die folgenden Beispiele untersuchen das Ganzzahl-Literal 3:

```
>> 3.instance_of?(Fixnum)
=> true
>> 3.instance_of?(Numeric)
=> false
>> 3.kind_of?(Fixnum)
```

```
=> true
>> 3.kind_of?(Numeric)
=> true
>> 3.is_a?(Object)
=> true
```

Wie Sie sehen, liefert `3.instance_of?(Fixnum)` den Wert `true`, weil »kleine« Ganzzahlen direkte Instanzen der Klasse `Fixnum` sind. Für `Numeric`, den gemeinsamen Vorfahren aller Zahlenarten, liefert die Untersuchung `instance_of?` jedoch `false`. `kind_of?` ist dagegen sowohl für `Fixnum` selbst als auch für `Numeric` und sogar für `Object`, die allgemeinste aller Oberklassen, erfüllt.

Interessant ist auch die Frage, ob ein Objekt eine bestimmte Methode unterstützt. Verwenden Sie für diese Prüfung die Methode `respond_to?`, wobei Sie den Methodennamen entweder als String oder als Symbol angeben können. Hier einige Beispiele:

```
>> arr = [1, 2, 3]
=> [1, 2, 3]
>> arr.class
Array
>> arr.respond_to?("sort")
=> true
>> arr.respond_to?(:test)
=> false
>> arr.respond_to?("respond_to?")
=> true
```

Wie Sie sehen, besitzt ein `Array`-Objekt die Methoden `sort` und `respond_to?`, aber keine Methode namens `test`. Dabei ist es egal, ob die Methoden in der fraglichen Klasse selbst, in einer übergeordneten Klasse oder in einem Mixin-Modul definiert wurden.

Sie können auch eine Liste aller Methoden eines Objekts oder einer Klasse als Array erhalten, indem Sie die Methode `methods` aufrufen. Zum Beispiel:

```
>> z = Gueterzug.new(Lok::DIESEL, Lok::OHNE)
=> #<Gueterzug:0x2b7ef38 @loks=[#<Lok:0x2b7eefc @typ=50>,
#<Lok:0x2b7eee8 @typ=0>], @waggons=0>
>> z.methods
=> ["methods", "instance_eval", "dup", "instance_variables",
"lok_aendern", "instance_of?", "extend", "eql?", "info", "hash",
"id", "singleton_methods", "taint", "frozen?",
"instance_variable_get", "kind_of?", "to_a", "type",
"protected_methods", "instance_variable_set", "method", "is_a?",
"to_s", "respond_to?", "class", "require_gem", "tainted?", "==",
"private_methods", "anhaengen", "===", "__id__", "nil?", "untaint",
"waggontest", "gem", "send", "display", "inspect", "clone", "=~",
"object_id", "abhaengen", "require", "public_methods", "__send__",
"equal?", "freeze", "lokinfo"]
```

Auch hier werden nicht nur die in der Klasse selbst definierten Methoden, sondern auch diejenigen aus übergeordneten Klassen und Mixins angezeigt.

Das war nur ein kurzer Überblick über die wichtigsten Introspektions-Hilfsmittel. Weitere Informationen finden Sie in den Büchern und Online-Ressourcen, die in Anhang B vorgestellt werden.

Zusammenfassung

Zugegeben: In diesem Kapitel wurden eine Menge Themen abgehandelt. Aber nun verfügen Sie über solides und vor allem praktisches Grundlagenwissen über die wichtigsten Aspekte der Objektorientierung. Wenn Sie bereits andere objektorientierte Sprachen kennen, sind Sie sicherlich beeindruckt von der Einfachheit und Eleganz, mit der Ruby auch komplexe Konzepte handhabt. Wenn Ruby dagegen Ihre erste OO-Sprache ist, haben Sie Glück gehabt. So leicht lernen Sie diese Konzepte nämlich sonst nirgends, denn alle anderen Sprachen (mit Ausnahme von Smalltalk) implementieren sie umständlicher.

Das Grundkonstrukt der Objektorientierung ist eine Klasse; ihre Definition wird durch `class` Klassenname eingeleitet und mit `end` abgeschlossen. Dazwischen können beliebig viele Methoden definiert werden, die die Datenstruktur eines Objekts der Klasse manipulieren oder einfach zurückliefern. Die feste Verknüpfung von Daten und Code innerhalb jeder Instanz einer Klasse (Kapselung) sorgt für wartungsfreundlicheren und leichter wiederverwendbaren Code. Die Datenstruktur selbst wird in sogenannten Instanzvariablen oder Attributen gespeichert, die Ruby durch das führende `@` kennzeichnet. Eine besondere Methode ist der Konstruktor, dessen Name `initialize` lautet. Er wird in der Regel genutzt, um die Instanzvariablen zu initialisieren.

Mit Hilfe des Operators < können Sie bei einer Klassendefinition angeben, dass die neue Klasse alle Eigenschaften und Methoden von einer anderen Klasse übernehmen soll. Danach genügt es, Ergänzungen und Unterschiede neu zu schreiben. Dieses Konzept heißt Vererbung und ermöglicht den Aufbau umfangreicher Klassenhierarchien, in denen jegliche Codeverdopplung – eine häufige Fehlerquelle – verhindert wird.

Einige zusätzliche Features ergänzen das OOP-Standardrepertoire in Ruby. Ein wichtiges Beispiele sind die Module, die sich als Mixins in beliebige Klassen importieren lassen, so dass in verschiedenen Klassen dieselben Methoden zur Verfügung stehen, obwohl diese Klassen nicht durch Vererbung voneinander abgeleitet wurden. Auch der umfangreiche Werkzeugkasten zur Introspektion, also zur Untersuchung von Klassen, Instanzen und Methoden im laufenden Betrieb, erleichtert Ihnen das Leben als OO-Entwickler enorm.

Da Sie nun das Wichtigste über Objektorientierung wissen, wird hier noch einmal das Textmanipulierer-Beispiel aus Kapitel 2 aufgegriffen, wobei die Beschreibung nun alle in diesem Kapitel eingeführten Fachbegriffe enthält.

Beispiel 4-5: modtext.rb, der objektorientierte Textmanipulierer

```
 1  class ModText

 2    # Konstruktor: wird bei Objekterzeugung
 3    # mit new aufgerufen
 4    def initialize(txt = "")
 5      @txt = txt
 6    end

 7    # Enthaltenen Text nachtraeglich aendern
 8    def set_text(txt = "")
 9      @txt = txt
10    end

11    # Enthaltenen Text zurueckliefern
12    def get_text
13      @txt
14    end

15    # Als Text-Entsprechung des Objekts
16    # ebenfalls den Text zurueckliefern
17    def to_s
18      get_text
19    end

20    # Den Text rueckwaerts zurueckliefern
21    def turn
22      @txt.reverse
23    end

24    # Den Text mit * statt Vokalen zurueckliefern
25    def hide_vowels
26      @txt.gsub(/[aeiou]/i, "*")
27    end

28    # Den Text in "Caesar-Code" zurueckliefern
29    def rot13
30      @txt.tr("[A-Z][a-z]", "[N-ZA-M][m-za-m]")
31    end

32  end

33  # Neues ModText-Objekt mit Inhalt erzeugen
34  mtext = ModText.new("Hallo, meine liebe Welt!")

35  # Ausgabe der verschiedenen Methoden
36  printf "Originaltext:      %s\n", mtext.get_text
37  printf "Umgekehrt:         %s\n", mtext.turn
```

Beispiel 4-5: modtext.rb, der objektorientierte Textmanipulierer (Fortsetzung)

```
38   printf "Versteckte Vokale: %s\n", mtext.hide_vowels
39   printf "ROT13:             %s\n", mtext.rot13

40   # Text aendern
41   mtext.set_text "Diese Worte sind neu!"

42   # Ausgabe des Objekts als Text
43   # (ruft automatisch to_s auf)
44   printf "Neuer Text:        %s\n", mtext
```

Das Programm besteht aus drei logischen Teilen. Der erste Teil (Zeile 1 bis 32) ist die Definition der Klasse `ModText`:

```
class ModText
  ...
end
```

Die einzelnen `def`-Blöcke definieren die Methoden der Klasse, die im letzten Teil des Skripts aufgerufen werden. Außer `initialize` und `set_text` liefern sie alle einen Wert zurück – in Ruby genügt es, den entsprechenden Wert als einzelne Anweisung hinzuschreiben.

Hier eine kurze Übersicht über die Aufgaben der einzelnen Methoden:

- `initialize` (Zeile 4 bis 6) wird bei der Instanziierung aufgerufen und weist der Instanzvariablen `@txt` ihren Anfangswert zu – entweder den übergebenen Inhalt oder einen leeren Text.
- `set_text` (Zeile 8 bis 10) ändert den enthaltenen Text des Objekts nachträglich.
- `get_text` (Zeile 12 bis 14) liefert den Text zurück.
- `to_s` (Zeile 17 bis 19) ruft `get_text` auf, da beide Methoden dieselbe Aufgabe erfüllen. Beachten Sie aber: Wenn ein Objekt in einem String-Kontext eingesetzt werden soll, wird seine Methode `to_s` – falls vorhanden – automatisch aufgerufen.
- `turn` (Zeile 21 bis 23) dreht den Text mit Hilfe der Ruby-Methode `reverse` herum und liefert das Ergebnis zurück.
- `hide_vowels` (Zeile 25 bis 27) verwendet den regulären Ausdruck `[aeiou]`, der auf alle Vokale im Text zutrifft, und ersetzt diese durch Sternchen. Auch hier wird das Endergebnis zurückgeliefert.
- `rot13` (Zeile 29 bis 31) wendet den so genannten »Cäsar-Code« auf den Text an: Jeder Buchstabe wird um 13 Zeichen verschoben; da das Alphabet 26 Buchstaben besitzt, sind die Vorgänge der Codierung und Decodierung identisch.

Der zweite Teil (ausschließlich Zeile 34) ist die Erzeugung einer konkreten `ModText`-Instanz mit Textinhalt:

```
mtext = ModText.new("Hallo, meine liebe Welt!")
```

Dies ruft automatisch die Methode `initialize` auf und speichert den übergebenen Text dauerhaft in der Instanzvariablen `@txt` des neuen Objekts `mtext`.

Im dritten Teil (ab Zeile 36) werden schließlich die verschiedenen Methoden des Objekts aufgerufen; außer `set_text` zum Wechseln des Inhalts dienen sie alle dem Auslesen des (meist manipulierten) Textinhalts von `mtext`. Die Anweisung `printf` ersetzt die im Text enthaltenen `%`-Platzhalter übrigens der Reihe nach durch die nachfolgenden Werte; `%s` steht dabei für einen Textwert (*String*).

In diesem Kapitel:

- Kurze Einführung in TCP/IP
- Sockets
- Web-Clients mit Net::HTTP
- Prozesse und Threads

KAPITEL 5

Netzwerkanwendungen

The Network is the Computer.
– Werbeslogan von Sun Microsystems

Programmierung wird noch interessanter, wenn sie über die Grenzen eines einzelnen Rechners hinausgeht. *Netzwerkfähigkeit* ist eine zentrale Forderung an moderne Software. Noch vor wenigen Jahren hätte man an dieser Stelle fragen müssen, welche Netzwerkprotokolle denn zum Einsatz kommen sollen – Microsoft, Apple und die UNIX-Welt kochten jeweils ihr eigenes Süppchen mit zueinander inkompatiblen Netzwerken.[1] Heute kann man glücklicherweise praktisch sagen:

Netzwerk = Internet

Das heißt, dass so gut wie jedes lokale oder standortübergreifende Computernetzwerk Internetstandards verwendet. Diese sind frei, offen und für jedes Betriebssystem und beliebige Netzwerkhardware verfügbar.

In diesem Kapitel erfahren Sie, wie man Netzwerk- und Internetanwendungen zu Fuß schreibt, während es im nächsten Kapitel um Webanwendungen mit fertig eingerichtetem Webserver geht.

Kurze Einführung in TCP/IP

Bevor Sie erfolgreich in die Netzwerkprogrammierung einsteigen können, brauchen Sie ein wenig technisches Hintergrundwissen über die Internet-Kommunikation.

1 Die meisten dieser proprietären Protokolle existieren bis heute, und selbst Ruby kann darauf zugreifen, falls das jeweilige Betriebssystem das unterstützt. Die praktische Bedeutung wird allerdings von Tag zu Tag geringer, so dass dies hier nicht weiter thematisiert werden soll.

Das Internet und die meisten anderen modernen Netzwerke setzen einen seit 1969 entwickelten Kommunikationsstandard ein, der als TCP/IP (Transmission Control Protocol/Internet Protocol) bezeichnet wird. Der Begriff *Protokoll* steht in diesem Zusammenhang für einen Standard zur Datenübertragung, wobei TCP und IP nur zwei wichtige Teilprotokolle einer ziemlich großen Familie sind.

Schematischer Überblick

Um Netzwerke zu analysieren und zu verstehen, bedient man sich so genannter *Schichtenmodelle*. Sie beschreiben die verschiedenen Funktionsebenen eines Netzwerks. Betrachten Sie zum Vergleich die »Schichten« der Kommunikation per klassischem Brief:

1. Ganz unten befindet sich die »Hardware« – bedrucktes Papier.
2. Wer die entsprechende Schrift und Sprache beherrscht, erkennt in den gedruckten Zeichen auf dem Papier einen Text.
3. Aus den Wörtern und Sätzen des Textes ergibt sich ein Inhalt, den möglicherweise nur spezielle Personengruppen verstehen. Mit einem Brief, den ein Arzt an einen Fachkollegen schreibt, hätte ein Programmierer beispielsweise Schwierigkeiten.
4. Damit der Brief einen ganz bestimmten Empfänger erreichen kann, wird er in einen Umschlag verpackt, der mit einer Anschrift versehen wird.

Ein Merkmal von Schichtenmodellen ist, dass beim Sender und Empfänger dieselben Schichten existieren, aber umgekehrt abgearbeitet werden. Um beim Alltagsbeispiel zu bleiben: Der Absender schreibt den Brief, verpackt ihn in den Umschlag und adressiert ihn. Der Empfänger packt ihn dagegen zuerst aus und liest ihn dann. Diese beiden Schichten werden in Abbildung 5-1 dargestellt.

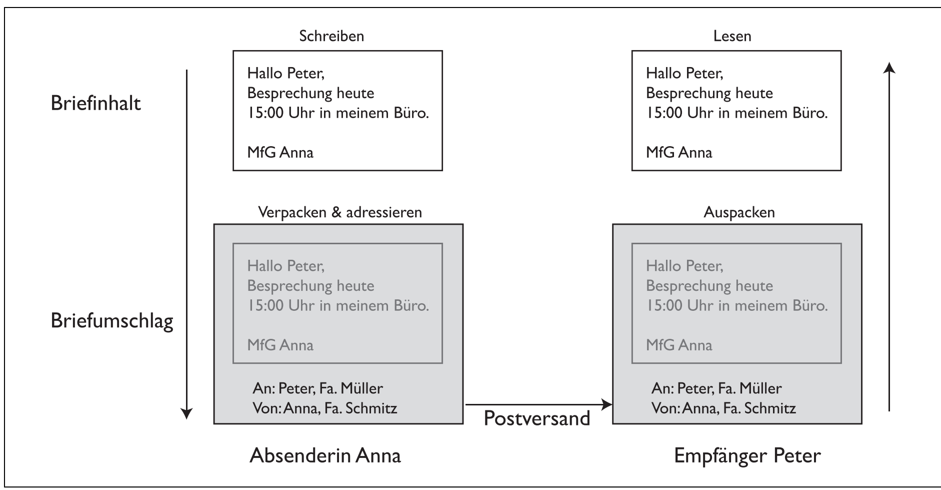

Abbildung 5-1: Ablauf der Kommunikation im einfachen Schichtenmodell »Briefversand«

Der Klassiker unter den Schichtenmodellen, das so genannte *OSI-Referenzmodell*, besitzt sieben Schichten. Es wurde vor allem im Hinblick auf die Vergleichbarkeit völlig unterschiedlicher Netzwerkarten entworfen. In einer Infrastruktur, in der ausschließlich die Internetprotokolle zum Einsatz kommen, ist es irrelevant und wird daher hier nicht weiter behandelt.

Für das Internet gibt es ein eigenes Schichtenmodell mit nur vier Schichten:

1. Auf der *Netzzugangsschicht* arbeitet die Netzwerkhardware. Dort wird geklärt, wie die zu übertragenden Daten beschaffen sind, wer wann senden darf und so weiter.
2. Die *Internetschicht* kümmert sich zum einen um die eindeutige Adressierung jedes einzelnen Rechners und zum anderen um die Weiterleitung von Daten zwischen verschiedenen Teilnetzen (*Routing*). Die Bezeichnung ist übrigens älter als *das Internet*; sie besagt, dass diese Schicht die Kommunikation zwischen mehreren Netzen ermöglicht.
3. Die *Host-zu-Host-Transportschicht* zerlegt die Daten in kleinere Einheiten (*Datenpakete*). Sie sorgt dafür, dass sie auf dem Zielrechner wieder korrekt zusammengesetzt und an das richtige Programm ausgeliefert werden. Der Begriff *Host* (Englisch für Gastgeber) bezeichnet jeden Rechner, der an der Netzwerkkommunikation als aktiver Sender oder Empfänger mitwirkt (im Gegensatz zu den Routern, die nur als Vermittler zwischen verschiedenen Netzwerken agieren).
4. Auf der *Anwendungsschicht* schließlich werden die eigentlichen Nutzdaten ausgetauscht. Dort gibt es eine Reihe standardisierter Anwendungsprotokolle wie HTTP (Web), FTP und E-Mail. In diesem Kapitel werden Sie allerdings auch ein eigenes Anwendungsprotokoll implementieren.

Auch diese Schichten werden beim Sender von oben nach unten und beim Empfänger in umgekehrter Reihenfolge abgearbeitet. Nehmen Sie beispielsweise an, jemand möchte die Website *www.ruby-lang.org* besuchen. Dazu muss der Webbrowser (Client) auf seinem Rechner eine Verbindung mit dem Webserver auf dem Zielrechner herstellen. Diese Verbindung wird durch ein Paar eindeutiger Nummern gekennzeichnet, die so genannten *Portnummern*. Der Client verwendet dabei eine zufällige Nummer, der Server in der Regel eine je nach Aufgabe festgelegte Nummer – ein Webserver hat beispielsweise meistens die Portnummer 80.

Sobald die Verbindung steht, sendet der Client eine *Anfrage* (request), deren Format durch ein *Anwendungsprotokoll* festgelegt wird. Für Webseiten wird das Anwendungsprotokoll *HTTP* verwendet; die Anforderung der Startseite einer Website lautet in dieser Sprache:

```
GET / HTTP/1.1
[... weitere Anfragedaten ...]
```

Die Anfrage wird in ein *Datenpaket* verpackt. Die Nutzdaten (die dem Briefbogen im Postbeispiel entsprechen) werden also um Versandinformationen (vergleichbar mit einem Briefumschlag) ergänzt. Der »innere Umschlag« der Host-zu-Host-Transportschicht nummeriert die Pakete einer Übertragungssequenz und fügt die beiden Portnummern hinzu. Damit weiß der Empfängerrechner, an welches konkrete Programm er den Inhalt aushändigen soll und von welchem Programm auf dem Absenderrechner die Daten stammen. Als Nächstes wird das Paket der Transportschicht in einen weiteren »Umschlag« verpackt: Die Internetschicht fügt die Adresse des Absenderrechners und die des Empfängerrechners hinzu. Das Format dieser so genannten *IP-Adressen* wird weiter unten erläutert. Die Pakete der Internetschicht werden schließlich über die Netzwerkhardware (zum Beispiel Ethernet, WLAN, DSL oder eine Modem-Verbindung) ins eigentliche Netz geschickt, wobei eine letzte Ebene von Versandinformationen hinzukommt.

Nun müssen die Daten zum Zielrechner transportiert werden. Wenn sich Sender und Empfänger im gleichen physischen Netzwerk befinden, werden die Pakete ohne Umweg zugestellt – etwa über einen Ethernet-Switch oder einen einzelnen WLAN-Hotspot. Bei Internetverbindungen ist das jedoch so gut wie nie der Fall. Hier kommt das Routing ins Spiel: Die Router sind spezielle Rechner, die jeweils mit mehr als einem Teilnetz verbunden sind und Daten zwischen diesen Netzen weiterleiten. Das Internet ist ein komplexes Geflecht aus Teilnetzen, die durch Router miteinander verbunden sind.

Ein Router betrachtet Datenpakete nur auf der Internetschicht; er bestimmt anhand der Absender- und Zieladresse, an welches Teilnetz er sie am besten weiterleitet. Damit ähnelt er gewissenhaften Postmitarbeitern, die nur die Adressangaben auf dem Umschlag lesen, um Briefe weiterzuleiten.[2]

Sobald die Daten beim endgültigen Empfänger ankommen, reicht dieser sie nach oben bis zur Anwendung weiter: Die Empfangssoftware der Internetschicht entfernt die für sie bestimmten Zusatzinformationen und liefert das enthaltene Paket an die Transportschicht aus. Diese bestimmt anhand der Empfänger-Portnummer, für welches Programm der Inhalt bestimmt ist, und gibt diesen weiter.

Das Empfängerprogramm – im vorliegenden Beispiel der Webserver – interpretiert die Anfrage. Anschließend verschickt er eine passende *Antwort* (response) und wird auf diese Weise seinerseits zum Sender. Angenommen, der Webserver hat die angeforderte Startseite gefunden, dann antwortet er wie folgt:

---

2 Wobei es prinzipiell möglich ist, einen Router so zu programmieren, dass er den Inhalt der Pakete liest oder kopiert – genau wie unseriöse Zusteller heimlich Briefe öffnen können, um beispielsweise Schecks zu finden. Inzwischen werden Netzbetreiber sogar staatlicherseits zu derartigen Maßnahmen gezwungen, um im Rahmen der vermeintlichen »Terrorabwehr« die Kommunikation zu überwachen. Aus diesen Gründen ist es wichtig, die Nutzdaten zu verschlüsseln, wenn etwa eine Kreditkartennummer oder andere vertrauliche Daten übermittelt werden.

```
HTTP/1.1 200 OK
[... weitere Antwortdaten ...]

<html><head><title> ...
```

Die erste Zeile enthält einen Statuscode (hier 200 für eine erfolgreich beantwortete Anfrage); nach einigen weiteren technischen Daten, die weiter unten behandelt werden, folgt der Inhalt der angeforderten Datei – hier durch `<html>...` angedeutet.

Der Weg durch die Schichten verläuft nun nach demselben Schema, wie oben erläutert, aber genau umgekehrt. Zu guter Letzt nimmt der Browser den eventuell aus mehreren Paketen zusammengesetzten HTML-Code in Empfang und stellt nach den darin formulierten Regeln die gewünschte Webseite dar. In der Regel sendet er dabei weitere Anfragen an den Server, um eingebettete Bilder zu laden.

In Abbildung 5-2 sehen Sie den gesamten hier geschilderten Ablauf noch einmal schematisch dargestellt. Einige Details – etwa IP-Adressen und Portnummern – werden im Folgenden noch genauer erklärt. Die hier verwendeten Adressen sind natürlich fiktiv.

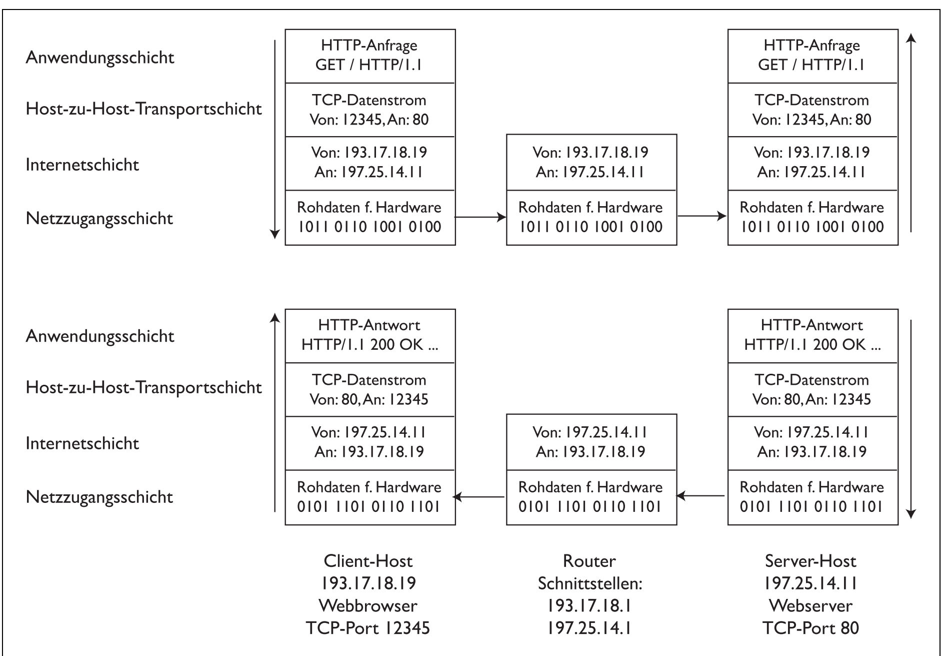

Abbildung 5-2: Schematischer Ablauf der Anforderung und Lieferung einer Webseite

IP-Adressen und Routing

Ein besonders wichtiges Detail der TCP/IP-Kommunikation sind die Vorgänge auf der Internetschicht. Dort arbeitet das *Internet Protocol* (IP), einer der beiden Namensgeber der Protokollfamilie. Wie bereits erwähnt, werden die Datenpakete dieser Schicht – *Datagramme* genannt – mit der Adresse des Absenders und derjenigen des Empfängers versehen. Dafür werden so genannte *IP-Adressen* verwendet. Es handelt sich um 32 Bit lange Zahlen, die jeden im Internet oder einem anderen TCP/IP-Netzwerk vorhandenen Rechner (Host oder Router) eindeutig kennzeichnen.[3] Üblicherweise besteht eine IP-Adresse aus vier durch Punkte getrennten Dezimalzahlen zwischen 0 und 255. Beispiele: 129.17.21.56 oder 83.12.51.126.

Der Anfangsteil einer IP-Adresse gibt das Netzwerk an, zu dem der betreffende Rechner gehört, der Rest steht für den Rechner selbst. Der Übergang zwischen den beiden Teilen ist nicht immer an derselben Stelle. Traditionell gibt es die folgenden *Klassen* von IP-Adressen. Zu welcher Klasse eine Adresse gehört, richtet sich nach dem ersten ihrer vier Zahlenblöcke:

- *Klasse A* (von 0 bis 127): 8 Bit kennzeichnen das Netzwerk, die restlichen 24 den Rechner. Dies ermöglicht 16.777.216 Adressen pro Netz. 12.13.14.15 und 12.14.15.16 gehören beispielsweise zum gleichen Netz, während 13.14.15.16 einem anderen angehört.
- *Klasse B* (von 128 bis 191): 16 Bit für das Netzwerk und 16 für den Rechner. In jedem Netz stehen 65.536 Adressen zur Verfügung. Somit gehören 130.10.22.23 und 130.10.23.24 zum selben Netz, aber 130.11.23.24 nicht.
- *Klasse C* (von 192 bis 223): 24 Bit für das Netzwerk und 8 für den Rechner, so dass jedes Netz nur 256 Adressen enthält. 194.10.11.12 und 194.10.11.13 gehören daher zum gleichen Netz, aber 194.11.12.13 zu einem anderen.
- *Klasse D* (von 224 bis 239) ist der Bereich der so genannten *Multicast-Adressen*. Sie werden verwendet, um dieselben Daten ressourcenschonend an mehrere Hosts weiterzuleiten.

Übrigens müssen Sie von der Anzahl der Adressen in einem Teilnetz noch zwei abziehen, um die Anzahl der möglichen Hosts zu erhalten. Die erste verfügbare Adresse kennzeichnet nämlich das Netzwerk selbst, während die letzte als so genannte *Broadcast-Adresse* dient, die empfangene Pakete an alle Hosts im gleichen Netz weiterleitet. So können im Klasse-C-Netz mit dem Adressbereich 193.17.18.x beispielsweise nur die 254 Adressen von 193.17.18.1 bis 193.17.18.254 für Hosts benutzt werden. 193.17.18.0 ist die Netzwerk-Adresse und 193.17.18.255 die Broadcast-Adresse.

3 Dies ist die klassische Version, *IPv4*. Da der Adressraum allmählich knapp wird, gibt es eine Neufassung des Protokolls namens *IPv6*, die 128 Bit lange Adressen verwendet, aber noch nicht universell verbreitet ist.

Das IP-Protokoll wurde zu einer Zeit erfunden, als es nur wenige hundert Hosts gab und man sich vielleicht einen Zuwachs auf einige Tausend vorstellen konnte. Heute gibt es allerdings viele Millionen Hosts, so dass sich das Klassensystem längst als zu unflexibel erwiesen hat. Deshalb kann die Grenze zwischen Netz- und Hostteil heute flexibel gesetzt werden. Dazu wird die Anzahl der Netzwerk-Bits, durch einen Slash getrennt, hinter der IP-Adresse notiert. Dieses Verfahren wird als *CIDR* (Classless Inter-Domain Routing) bezeichnet.

Eine andere Schreibweise ist die *Teilnetzmaske* (subnet mask); sie verwendet dieselben vier Blöcke wie die Adressdarstellung selbst, wobei Netzwerk-Bits auf 1 und die Host-Bits auf 0 gesetzt werden. Hier als Beispiel die Teilnetzmasken für die drei Standardklassen:

- Klasse A: 255.0.0.0
- Klasse B: 255.255.0.0
- Klasse C: 255.255.255.0

168.10.17.4 ist beispielsweise eine Klasse-B-Adresse, die zum Adressbereich 168.10.0.0 bis 168.10.255.255 gehört. 168.10.17.4/24 (Teilnetzmaske 255.255.255.0) gehört dagegen zu einer CIDR-Teilmenge dieses Netzes, in der 24 Bit das Netzwerk festlegen – es beinhaltet also nur den Adressbereich 168.10.17.0 bis 168.10.17.255.[4]

Viele Hosts besitzen keine festen IP-Adressen, sondern dynamisch zugewiesene. In lokalen Netzwerken werden sie beim Booten über ein Protokoll namens *DHCP* (Dynamic Host Configuration Protocol) zugewiesen, in Fernverbindungen (etwa DSL) bei der Einwahl. Eine weitere Adress-Sparmaßnahme besteht darin, dass einige Adressbereiche für private Netzwerke reserviert sind. Die Daten dieser Adressen werden nicht ins Internet geroutet, so dass ein und derselbe Bereich in beliebig vielen Organisationen verwendet werden kann. Im Einzelnen sind das folgende Adressen:

- das Klasse-A-Netz 10.0.0.0/8
- die sechzehn Klasse-B-Netze 172.16.0.0/16 bis 172.31.0.0/16
- die 256 Klasse-C-Netze 192.168.0.0/24 bis 192.168.255.0/24

Es gibt eine indirekte Möglichkeit, solche Privatnetze mit dem Internet zu verbinden – ein Verfahren namens *NAT* (Network Address Translation). Selbst die kleinsten Geräte, etwa jede beliebige DSL-Routerbox, unterstützen dieses Protokoll, was Ihnen den Aufwand und die Kosten erspart, offizielle IP-Adressen für Ihre Privatrechner zu beantragen.

---

4 Es gibt sogar CIDR-Adressen, in denen die Trennlinie zwischen Netz- und Hostteil nicht an einer Byte-Grenze verläuft. 192.168.17.4/26 (Teilnetzmaske 255.255.255.192) gehört beispielsweise zu einem Netz mit den Adressen 192.168.17.0 bis 192.168.17.63. Dieser komplexere Fall wird hier allerdings nicht näher betrachtet.

Die zweite Aufgabe des IP-Protokolls neben der Adressierung ist das *Routing*, das heißt die Weiterleitung von Daten in andere Teilnetze. Wie sich das Teilnetz aus der Adresse ergibt, wurde soeben erläutert. Ein Router besitzt mehrere Netzwerkschnittstellen (Ethernet, WLAN, DSL usw.). Jede von ihnen besitzt eine IP-Adresse aus einem anderen Netz. Der Router leitet Daten, die er über eine seiner Schnittstellen empfängt, über eine andere in ein anderes Netz weiter. Sie können sich sogar die einzelnen Stationen des Routings ansehen. Beinahe jeder TCP/IP-fähige Rechner enthält ein Dienstprogramm, das diesen Weg beschreiben kann. Hier die Route von meinem Rechner (DSL bei NetCologne) zu *www.rubygarden.org*, einer empfehlenswerten Ruby-Ressourcen-Site:

```
> tracert www.rubygarden.org

Routenverfolgung zu rubygarden.org [216.133.73.195]  ueber maximal 30 Abschnitte:


  1  60 ms  50 ms  54 ms  erx-maw1.netcologne.de [195.14.247.95]
  2  46 ms  49 ms  47 ms  swrt-maw1-g34.netcologne.de
[213.196.239.169]
  3  47 ms  45 ms  47 ms  core-maw1-vl200.netcologne.de
[195.14.195.145]
  4  47 ms  47 ms  46 ms  core-sto2-vl911.netcologne.de
[195.14.215.249]
  5 176 ms 143 ms 142 ms  rteq-g03.netcologne.de [81.173.194.169]
  6 143 ms 143 ms 143 ms  equinix-ash.epoch.net [206.223.115.52]
  7 143 ms 145 ms 144 ms  ge-0-0-0.c00.ash.megapath.net
[155.229.70.85]
  8 143 ms 144 ms 142 ms  ge-0-2-0.c01.dcp.megapath.net
[155.229.70.38]
  9 148 ms 148 ms 149 ms  pos1-0.btm-m200.gw.epoch.net
[155.229.123.213]
 10 146 ms 149 ms 149 ms  206-135-244-114.btm-m200.cust.gw.epoch.net
[206.135.244.114]
 11 148 ms 149 ms 148 ms  ns1.chadfowler.com [216.133.73.195]

Ablaufverfolgung beendet.
```

Beachten Sie, dass das Programm auf UNIX-Systemen `traceroute` statt `tracert` heißt. Wie Sie sehen, werden für jede Station drei Übertragungszeiten angegeben, weil jede dreimal befragt wird.

Ein noch einfacheres Dienstprogramm ist übrigens `ping`. Es prüft, ob ein bestimmter Host überhaupt erreichbar ist – hier etwa *www.ruby-lang.org*:

```
> ping www.ruby-lang.org

Ping carbon.ruby-lang.org [210.163.138.100] mit 32 Bytes an Daten:

Antwort von 210.163.138.100: Bytes=32 Zeit=331ms TTL=42
Antwort von 210.163.138.100: Bytes=32 Zeit=331ms TTL=42
Antwort von 210.163.138.100: Bytes=32 Zeit=332ms TTL=42
Antwort von 210.163.138.100: Bytes=32 Zeit=346ms TTL=40
```

```
Ping-Statistik fuer 210.163.138.100:
    Pakete: Gesendet = 4, Empfangen = 4, Verloren = 0 (0% Verlust),
Ca. Zeitangaben in Millisek.:
    Minimum = 331ms, Maximum = 346ms, Mittelwert = 335ms
```

Die hier gezeigte Windows-Variante von `ping` sendet automatisch vier Testpakete und zeigt dann eine Statistik an. Auf UNIX-Systemen sendet das Programm dagegen so lange, bis Sie **Strg + C** drücken.

Übrigens verwenden Menschen im Internet in der Regel keine IP-Adressen, sondern Hostnamen wie *www.oreilly.de* oder *www.rubygarden.org*. Intern müssen diese allerdings stets in die zugehörigen IP-Adressen umgewandelt werden. Diese Dienstleistung erbringt ein Dienst namens *DNS* (Domain Name System), eine weltweit verteilte Datenbank so genannter *Nameserver*. Wie das funktioniert, können Sie auf der Kommandozeile mit Hilfe des Dienstprogramms `nslookup` ermitteln. Es benötigt den aufzulösenden Hostnamen sowie die IP-Adresse eines Nameservers. Zum Beispiel:

```
> nslookup www.oreilly.de 194.8.194.70
Server:  ns1.netcologne.de
Address:  194.8.194.70

Nicht autorisierte Antwort:
Name:    norawww.oreilly.de
Address:  62.206.71.33
Aliases:  www.oreilly.de
```

Der in diesem Beispiel befragte Nameserver stammt von NetCologne – verwenden Sie entsprechend einen Nameserver Ihres eigenen Providers. Der Ausdruck »Nicht autorisierte Antwort« besagt nicht etwa, dass die Antwort falsch ist, sondern dass der Nameserver sie aus seinem Cache geliefert hat, ohne den eigentlich zuständigen Nameserver erneut zu fragen – gewissermaßen eine Antwort »ohne Gewähr«.

Aufgaben der Host-zu-Host-Transportschicht

Während das IP-Protokoll durch Routing grundsätzlich dafür sorgt, dass überhaupt Datenpakete von einem bestimmten Host zu einem beliebigen anderen gesendet werden können, ist es die Aufgabe der Host-zu-Host-Transportschicht, dies für den eigentlichen Datentransport zu nutzen. Dafür gibt es zwei verschiedene Transportprotokolle, die für unterschiedliche Anwendungsfälle geeignet sind: das *Transmission Control Protocol* (TCP) und das *User Datagram Protocol* (UDP). TCP stellt einen verlässlichen Datenstrom zwischen den beiden Hosts zur Verfügung, während UDP zum schnellen, einfachen Versand einzelner Nachrichten dient.

Bei TCP wird vor dem eigentlichen Datenversand eine Verbindung zwischen den Hosts aufgebaut. Da die zugrunde liegenden Datagramme der Internetschicht nach wie vor durch Routing transportiert werden, handelt es sich nicht um eine echte

Punkt-zu-Punkt-Verbindung (die etwa bei klassischen Telefondiensten zustande kommt), sondern um eine virtuelle: Die Datenpakete werden dazu mit einer Sequenznummer versehen, und der Sender erwartet für jedes Paket eine Bestätigung vom Empfänger. Bleibt diese aus, sendet er das betreffende Paket nach einer gewissen Wartezeit erneut. Das Ergebnis dieses Verfahrens ist ein absolut zuverlässiger Datenstrom mit definierter Reihenfolge, so dass sich über eine TCP-Verbindung beliebig große Datenmengen übertragen lassen.

UDP-Pakete werden dagegen stets einzeln gesendet, und der Sender kümmert sich nicht darum, ob sie ihr Ziel erreichen. Sie haben allerdings auch einen wichtigen Vorteil gegenüber TCP: Da sie keine Sequenznummern, Empfangsbestätigungen und weitere Datenstromlogik benötigen, ist der Anteil von Verwaltungsinformationen an den zu sendenden Gesamtdaten erheblich geringer, so dass sich in derselben Zeit wesentlich mehr Informationen übertragen lassen, dies allerdings nicht zuverlässig.

Gemeinsam haben die beiden Protokolle das bereits erwähnte Konzept der Portnummern: Das Portnummern-Paar eines Datenpakets kennzeichnet eindeutig einen bestimmten Kommunikationskanal zwischen Absender und Empfänger. Stellen Sie sich zur Verdeutlichung vor, ein und derselbe Browser fordert in zwei Fenstern unterschiedliche Seiten vom gleichen Server an. Der Webserver besitzt in beiden Fällen die Portnummer 80; aber die Verwendung von zwei verschiedenen Client-Ports ermöglicht eine exakte Zuordnung.

Tabelle 5-1 zeigt einige Portnummern, die üblicherweise für gängige Serverdienste verwendet werden. Die Spalte *Transportprotokoll* gibt Auskunft darüber, mit welchem der beiden Transportdienste (TCP oder UDP) die jeweiligen Anwendungsdaten übertragen werden können. Die vollständige Liste aller offiziell registrierten TCP- und UDP-Portnummern – der so genannten *Well-Known Ports* – finden Sie unter *http://www.iana.org/assignments/port-numbers*. Wenn Sie ein UNIX-System verwenden, besitzen Sie zusätzlich die Datei */etc/services* mit einer brauchbaren Teilmenge dieser Liste.

Tabelle 5-1: Einige verbreitete TCP- und UDP-Portnummern

Portnummer	Transportprotokoll	Anwendungsprotokoll	Erläuterung
20	TCP	FTP-Daten	Datei-Upload/Download
21	TCP	FTP-Steuerung	Datei-Upload/Download
22	TCP	SSH	Sichere Terminalverbindung
23	TCP	Telnet	Klassische Terminalverbindung
25	TCP	SMTP	E-Mail-Versand
53	TCP, UDP	Domain	Nameserver-Abfragen
80	TCP	HTTP	Webserver
110	TCP	POP3	E-Mail-Postfach

Tabelle 5-1: Einige verbreitete TCP- und UDP-Portnummern (Fortsetzung)

Portnummer	Transportprotokoll	Anwendungsprotokoll	Erläuterung
123	UDP	NTP	Uhrzeitsynchronisation
443	TCP, UDP	HTTPS	Verschlüsselte Webverbindung
3306	TCP, UDP	MySQL	Beliebter Open-Source-Datenbankserver

Auf UNIX-Systemen gibt es eine wichtige Einschränkung: Verbindungen, bei denen die lokale Portnummer kleiner als 1024 ist (klassische Serverdienste), dürfen nur von `root` initiiert werden. In der »guten alten Zeit«, als die Universitätsnetze praktisch unter sich waren, diente dies der Sicherheit: Auf einem allgemein zugänglichen Host konnte nicht jeder Hinz und Kunz einen offiziellen Server starten.

Inzwischen ist diese Einschränkung sogar eher ein Sicherheitsrisiko, da ein erfolgreicher externer Angriff auf den Server so einen `root`-Prozess unter seine Kontrolle bringen und erheblich mehr Schaden anrichten könnte. Professionelle Server wie der Webserver Apache (siehe nächstes Kapitel) umgehen dieses Problem, indem sie zwar die Verbindung über einen `root`-Prozess herstellen, die eigentliche Kommunikation mit den Clients aber nicht privilegierten Prozessen überlassen.

Sockets

Wenn Sie beim Programmieren auf Netzwerkverbindungen zugreifen möchten, verwenden Sie so genannte *Sockets* dafür. Ein Socket ist einer der beiden Endpunkte der Netzwerkkommunikation. Im Gegensatz zu Dateihandles oder Pipes sind Sockets *bidirektional*, können also sowohl zum Senden als auch zum Empfangen von Daten eingesetzt werden.

Als Grundlage der Socket-Programmierung dient bei allen UNIX-Derivaten die *Berkeley Socket API*[5], die ursprünglich für BSD UNIX entwickelt wurde. Windows verwendet von Haus aus eine leicht unterschiedliche Bibliothek namens *Windows Socket Library* (kurz Winsock). Viele moderne Programmiersprachen – auch Ruby – maskieren die geringfügigen Unterschiede zwischen diesen Ansätzen und stellen für alle unterstützten Plattformen dieselbe Netzwerkunterstützung zur Verfügung: eine meist um nützliche Features erweiterte Variante der Berkeley Sockets. In Ruby ist der Zugang aufgrund der Objektorientierung besonders komfortabel.

Um Sockets zu verwenden, müssen Sie zunächst die Include-Datei *socket* aus der Ruby-Standardbibliothek importieren, und zwar am besten zu Beginn Ihres Skripts:

```
require "socket"
```

5 API steht für »Application Programming Interface«, also eine Schnittstelle zur Anwendungsprogrammierung.

Danach steht Ihnen eine Reihe verschiedener Socket-Klassen zur Verfügung. Sie alle werden von der Klasse `BasicSocket` abgeleitet. Diese Basisklasse enthält die gemeinsame Funktionalität aller Socket-Typen, aber es lassen sich keine Instanzen von ihr bilden. Die verschiedenen Unterklassen lösen dafür jeweils spezielle Aufgaben. `BasicSocket` stammt wiederum von `IO` ab, so dass Sie sämtliche in Kapitel 3 vorgestellten Schreib- und Lesemethoden auch auf Sockets anwenden können (mit einigen Besonderheiten, die im Folgenden näher erläutert werden).

In diesem Kapitel werden vor allem einige komfortable Ruby-Erweiterungen der Socket-API behandelt. Der mehr oder weniger direkte Zugriff auf die Socket-Bibliothek des Betriebssystems ist ebenfalls möglich, und zwar über die auch von `BasicSocket` abgeleitete Klasse `Socket`. Diese Methode ist allerdings viel komplizierter und plattformabhängig, so dass sie hier nicht weiter behandelt wird.

Bevor einige Socket-Klassen und ihre Fähigkeiten im Detail erläutert werden, hier zunächst ein einfaches Beispiel. Das folgende kleine Skript lädt die Startseite von *www.oreilly.de* herunter, entfernt sämtliche HTML-Tags und zeigt den übrig gebliebenen Text an:

```
require "socket"

sock = TCPSocket.new("www.oreilly.de", 80)
sock.puts "GET / HTTP/1.1
Host: www.oreilly.de"
sock.puts
inhalt = sock.read
inhalt.gsub!(/<.*?>/m, "")
inhalt.gsub!(/\s{2,}/, "\n")
puts inhalt
```

Wenn Sie das Skript starten, erhalten Sie eine Ausgabe über mehrere Fensterhöhen (leiten Sie sie gegebenenfalls in eine Datei um, wenn Sie sie vollständig lesen möchten), die wie folgt beginnt:

```
> ruby easy_httpclient.rb
HTTP/1.1 200 OK
Date: Sun, 17 Dec 2006 09:58:26 GMT
Server: Apache/1.3.28 (Linux/SuSE) mod_ssl/2.8.15 OpenSSL/0.9.7b PHP/4.3.3 mod_
perl/1.28
Last-Modified: Sun, 17 Dec 2006 02:36:15 GMT
ETag: "211b7-6ca0-4584ad1f"
Accept-Ranges: bytes
Content-Length: 27808
Connection: close
Content-Type: text/html
oreilly.de -- Willkommen beim O'Reilly Verlag
O&#8217;Reilly Verlag
```

Was geschieht hier? Zuerst wird eine Instanz der Klasse `TCPSocket` erzeugt. Es handelt sich um einen TCP-Client-Socket, der eine Anfrage an einen TCP-Server senden kann. Die nötigen Argumente sind Hostname oder IP-Adresse sowie Port des gewünschten Servers, in diesem Fall `"www.oreilly.de"` sowie `80`. Bei Standard-Serverports können Sie auch deren bekannten Namen verwenden, hier also `"http"`.

Je nach angesprochenem Server können Sie nach dem Verbindungsaufbau unmittelbar lesen, bei anderen müssen Sie zuerst einen Befehl senden, der näher bestimmt, welche Information oder Dienstleistung Sie von dem Server wünschen. Ein klassischer `daytime`-Server (Port 13) sendet beispielsweise sofort Datum und Uhrzeit, während der im vorliegenden Beispiel angesprochene HTTP-Server zunächst eine Anfrage erwartet. Diese besteht hier aus drei Elementen:

- Der eigentliche Befehl `GET / HTTP/1.1` – eine einfache Dateianforderung (`GET`) für die Startseite `/`, Protokollversion `HTTP/1.1`.
- Die zusätzliche Header-Zeile `Host: www.oreilly.de` – der `Host`-Header ist in der HTTP-Version 1.1 Pflicht, weil unter derselben IP-Adresse mehrere Hostnamen mit unterschiedlichen Websites (technisch ausgedrückt: *namensbasierte virtuelle Hosts*) liegen können. Dieser Header gibt an, welche konkrete Site auf dem angesprochenen Rechner angefordert wird. Beim TCP-Verbindungsaufbau wird nämlich nur noch die zuvor über einen Nameserver aufgelöste IP-Adresse verwendet, so dass der konkrete Hostname zur Unterscheidung nochmals innerhalb der Nutzdaten des TCP-Pakets stehen muss.
- Eine Leerzeile (erzeugt durch `sock.puts`), die anzeigt, dass die Header vorbei sind. Eine `GET`-Anfrage ist damit vollständig, während andere HTTP-Anfragearten sowie die meisten HTTP-Antworten nach dieser Leerzeile noch einen Body enthalten – bei der Antwort beispielsweise den Inhalt der übertragenen Datei.

Nachdem die Anfrage vollständig ist, versendet der Server die Antwort. Sie enthält folgende Komponenten:

- Die Statuszeile `HTTP/1.1 200 OK` – bestehend aus Protokollversion (hier wiederum `HTTP/1.1`), Statuscode und -text. Der Status `200 OK` besagt, dass das angeforderte Dokument vorhanden ist und geliefert wird. Ein anderer bekannter Status ist beispielsweise `404 Not Found`; er tritt auf, wenn die gewünschte Seite nicht vorhanden ist.
- Diverse Header-Zeilen. Die wichtigste ist `Content-type`; sie teilt dem Browser den Datentyp der Antwort mit (im vorliegenden Fall `text/html` für ein HTML-Dokument). Die wichtigsten HTTP-Header werden weiter unten in diesem Kapitel erläutert.
- Der Body[6] – er wird durch eine Leerzeile von den Headern getrennt und enthält entweder das angeforderte Dokument oder gegebenenfalls eine Fehlermittei-

6 Wie Sie vielleicht wissen, besteht auch ein HTML-Dokument wiederum aus einem Head- und einem Body-Bereich. Aus der Sicht des HTTP-Protokolls bildet allerdings das *gesamte* HTML-Dokument den Body der Antwort. Eine sehr kurze HTML-Übersicht finden Sie weiter unten in diesem Kapitel.

lung. Im obigen Beispiel ist die erste vom HTML-Code befreite Body-Zeile der Titel `oreilly.de -- Willkommen beim O'Reilly Verlag`. In Wirklichkeit steht er zwischen den Tags `<title>` und `</title>`.

Der Client liest die Antwort mittels `sock.read` vollständig aus seinem Socket. Zum Schluss werden noch zwei Musterersetzungen durchgeführt: Alles, was auch mehrzeilig zwischen spitzen Klammern steht (`/<.*?>/m`), verschwindet, und zwei oder mehr Whitespace-Zeichen (`/\s{2,}/`) werden durch genau einen Zeilenumbruch ersetzt, um größere Lücken zu vermeiden.

Für den praktischen Gebrauch ist die hier verwendete Methode der HTML-Entfernung viel zu ungenau. Man müsste nämlich beispielsweise Text innerhalb von Anführungszeichen oder Kommentaren anders behandeln, das heißt unbehelligt stehen lassen.

Zum Schluss wird der gelesene und umgewandelte Inhalt mittels `puts` auf die Standardausgabe geschrieben.

TCP-Server und -Clients

Für die beiden Transportverfahren TCP und UDP existieren jeweils eigene Socket-Klassen namens `TCPSocket` und `UDPSocket`. Beide stammen von der gemeinsamen, nicht instanziierbaren Elternklasse `IPSocket` ab, in der ihre (beträchtliche) gemeinsame Funktionalität festgelegt wird. Von `TCPSocket` wird wiederum die Klasse `TCPServer` abgeleitet, die nicht selbstständig eine Verbindung initiiert, sondern an einem bestimmten Port auf Anfragen lauscht.

Dieser Abschnitt konzentriert sich auf das wichtigere und häufiger eingesetzte TCP; Informationen über den Einsatz von `UDPSocket` erhalten Sie in einigen der Ressourcen, die in Anhang B empfohlen werden. Zudem finden Sie auf der Website zu diesem Buch eine UDP-Version der weiter unten vorgestellten ECHO-Anwendung (Client und Server).

Grundlagen

Ein TCP-Server-Socket wird erzeugt, indem `TCPServer.new` mit einem oder zwei Parametern aufgerufen wird: Host und Portnummer oder nur Portnummer. Als Host können Sie einen Hostnamen oder eine IP-Adresse des lokalen Rechners angeben. In diesem Fall reagiert der Server nur auf Anfragen über diejenige Netzwerkschnittstelle, zu der die angegebene Adresse gehört. Das ist beispielsweise nützlich, wenn Ihr Server nur im lokalen Netzwerk, aber nicht über eine Fernverbindung erreichbar sein soll. Das folgende Beispiel erzeugt einen TCP-Server-Socket, der auf Verbindungen über Port 8000 mit der lokalen IP-Adresse 192.168.0.4 wartet:

```
server = TCPSocket.new("192.168.0.4", 8000)
```

Wenn Sie keinen Host angeben, lauscht der Server am angegebenen Port auf allen verfügbaren Schnittstellen. Hier ein Beispiel für einen Server, der an Port 8888 jeder Netzwerkverbindung lauscht:

```
server = TCPSocket.new(8888)
```

Statt der Portnummer können Sie auch den Namen eines Well-Known Ports verwenden. Im Zweifelsfall können Sie den betreffenden Port zunächst mit Hilfe der `Socket`-Klassenmethode `getservbyname` testen. Zum Beispiel:

```
>> Socket.getservbyname("ftp")
=> 21
```

Nachdem Sie einen Server-Socket erzeugt haben, lauscht er auf Client-Verbindungen. Wenn eine eintrifft, müssen Sie die Methode `accept` des Sockets aufrufen. Daraufhin erhalten Sie einen *zusätzlichen* Socket zur Kommunikation mit dem Client; bei der Ruby-Implementierung gehört er bequemerweise der Klasse `TCPSocket` an. Der Server-Socket lauscht dagegen weiter auf neue Verbindungen. Damit diese nacheinander verarbeitet werden können, führt der typische Server eine so genannte `accept`-Schleife aus: Jede neue Client-Verbindung wird akzeptiert, verarbeitet und wieder geschlossen.

Eine solche Schleife sieht beispielsweise so aus:

```
while (client = server.accept)
  # client ist der Kommunikations-Socket;
  # hier alle Lese- und Schreiboperationen abwickeln,
  # dann den Client-Socket schliessen:
  client.close
end
```

Dieses einfachste Modell eines TCP-Servers bekommt allerdings immer dann ein Problem, wenn die Verarbeitung der Client-Anfragen länger dauert oder gar bestehen bleiben soll. Der nächste Client kann nämlich wie an einer Supermarktkasse immer erst dann bedient werden, wenn einer fertig ist.

Die passende Lösung wurde erst vor kurzem aus der IT-Welt in die reale Welt übernommen: Alle Clients (Kunden!) warten an einer einzigen Warteschlange (dem lauschenden Socket), und sobald sie an der Reihe sind, verteilen sie sich auf verschiedene Schalter (Server-Instanzen). Dies wird im Abschnitt »Prozesse und Threads« auf Seite 256, erläutert. Hier wird dagegen zunächst ein einfacher Server gezeigt, der eine einzelne Anfrage schnell beantwortet und die Verbindung dann sofort beendet, um den nächsten wartenden Client zu bedienen.

Der aus dem `accept`-Aufruf empfangene `TCPSocket` des Servers sowie der `TCPSocket` des Clients dienen anschließend der direkten Kommunikation zwischen den beiden Stationen. Jeder Partner kann sowohl lesen als auch schreiben. Dafür kommen die in Kapitel 2 besprochenen Ein- und Ausgabemethoden zum Einsatz.

Ein TCP-Client-Socket wird übrigens direkt mit Hilfe des Konstruktors der Klasse TCPSocket erzeugt. Die beiden Parameter sind:

- Hostname oder IP-Adresse des gewünschten Servers
- Portnummer oder Name eines Well-Known Ports auf dem Server

Der Client selbst erhält übrigens eine zufällige Portnummer (»ephemeral port«). Das folgende Beispiel erzeugt einen Client-Socket für den FTP-Server der Universität zu Köln:

```
client = TCPSocket.new("ftp.uni-koeln.de", "ftp")
```

Hier ein weiteres Beispiel, das eine Verbindung zu dem nicht näher spezifizierten Server auf Port 8888 des lokalen Rechners herstellt:

```
client = TCPSocket.new("localhost", 8888)
```

Bei Sockets im Standardmodus gibt es allerdings ein spezifisches Problem: Wenn Sie versuchen, aus einem Socket zu lesen, der noch keine Daten bereitstellt, blockiert Ihre Netzwerkanwendung, bis das erste Byte ankommt (ähnlich wie beim Warten auf direkte Benutzereingaben). Anders als beim Einlesen von Dateien gibt es auch kein End-of-File, das den Abschluss einer Übertragungssequenz markiert. Sie müssen also genau wissen, wann Sie auf welche Datenmenge warten, und sollten die Lese- und Schreibaufgaben Ihrer TCP-basierten Skripten zudem jeweils als eigenständige Threads oder Prozesse ausführen.

Alternativ besteht die Möglichkeit, einen Socket in den so genannten *Nonblocking*-Modus zu versetzen, so dass er jederzeit sende- und empfangsbereit ist. Dies geht allerdings über den Themenkreis dieses Kapitels hinaus; konsultieren Sie dazu die in Anhang B empfohlenen Quellen.

Praxisbeispiel: ECHO-Server und -Client

Um das Zusammenspiel zwischen TCP-Server und -Client zu verdeutlichen, wird hier ein kleiner Server und der zugehörige Client präsentiert. Es handelt sich um einen ECHO-Server, der die an ihn gesendeten Daten einfach zurücksendet. Dieser Dienst verwendet traditionell den TCP- und UDP-Port 7 und kann eingesetzt werden, um die grundsätzliche Verfügbarkeit von Netzwerkverbindungen zu testen. Die hier vorgestellte ECHO-Version wurde allerdings leicht erweitert; das Ergebnis ist ein echtes kleines *Anwendungsprotokoll*, das zusätzlich zur bloßen Textrückgabe auf vier Administrationskommandos des Clients reagiert:

- MODE_NORMAL versetzt den Server in den Standardmodus, in dem der Text der Client-Anfrage originalgetreu zurückgesendet wird.
- MODE_REVERSE stellt den Reverse-Modus ein: Ab sofort wird der Inhalt der Client-Anfrage rückwärts zurückgesendet.

- `MODE_ROT13` beantwortet alle künftigen Anfragen mit der `ROT13`-Codierung des Anfragetextes.
- `EXIT` fordert den Server auf, sich zu beenden. Es ist wichtig, dass Sie ein solches Kommando zum Beenden in einen TCP-Server mit `accept`-Schleife einbauen, weil er sonst nur von außen mittels `kill` (UNIX) oder über den Taskmanager (Windows) beendet werden kann.

Die drei `MODE_*`-Kommandos liefern übrigens auch kein Textecho zurück, sondern eine administrative Antwort.

In Beispiel 5-1 sehen Sie zunächst den kompletten Quellcode des Servers, der im Anschluss erläutert wird.

Beispiel 5-1: Der erweiterte TCP-ECHO-Server, echoserver.rb

```
 1  require "socket"

 2  # Modus-Konstanten
 3  NORMAL = 0
 4  REVERSE = 1
 5  ROT13 = 2

 6  # Anfangswert fuer Modus festlegen
 7  mode = NORMAL

 8  # Port von der Kommandozeile lesen oder auf 7 setzen
 9  if(ARGV[0])
10    port = ARGV[0].to_i
11  else
12    port = 7
13  end

14  # Lauschenden Socket erzeugen
15  server = TCPServer.new(port)
16  # Infozeile ausgeben
17  puts "ECHO Server listening on port #{port} ..."

18  # Accept-Schleife
19  while (client = server.accept)
20    # Client-Anfrage auslesen ...
21    anfrage = client.gets.strip
22    # ... und auf die Konsole protokollieren
23    puts "Anfrage: #{anfrage}"
24    # Bei EXIT beenden
25    break if anfrage == "EXIT"
26    # Modus-Befehle behandeln
27    case anfrage
28    when "MODE_NORMAL"
29      mode = NORMAL
30      client.print "[Modus auf Normal gesetzt]\r\n"
31    when "MODE_REVERSE"
```

Beispiel 5-1: Der erweiterte TCP-ECHO-Server, echoserver.rb (Fortsetzung)

```
32        mode = REVERSE
33        client.print "[Modus auf Reverse gesetzt]\r\n"
34      when "MODE_ROT13"
35        mode = ROT13
36        client.print "[Modus auf ROT13 gesetzt]\r\n"
37      else
38        # Standardanfrage behandeln
39        case mode
40        when NORMAL
41          client.printf "%s\r\n", anfrage
42        when REVERSE
43          client.printf "%s\r\n", anfrage.reverse
44        when ROT13
45          client.printf "%s\r\n", anfrage.tr("[a-m][A-M][n-z][N-Z]",
                                             "[n-z][N-Z][a-m][A-M]")
46        end
47      end
48      # Client-Verbindung schliessen
49      client.close
50    end
51    # Lauschenden Socket schliessen
52    server.close
```

Die einzelnen Teile des ECHO-Server-Codes haben folgende Aufgaben:

- Zeile 1: Import der Socket-Bibliothek.
- Zeile 3-5: Definition einiger Konstanten für die verschiedenen ECHO-Modi.
- Zeile 7: Der Modus wird zunächst standardmäßig auf `NORMAL` gesetzt.
- Zeile 9-13: Wenn ein Kommandozeilenargument existiert, wird die Server-Portnummer auf den entsprechenden Wert gesetzt, andernfalls wird der Standardport 7 gewählt.[7]
- Zeile 15: Der lauschende Socket wird erzeugt. Wenn Sie eine sichere Variante bevorzugen, die nur Clients auf dem lokalen Rechner selbst zulässt, können Sie `TCPServer.new(port)` durch `TCPServer.new("localhost", port)` ersetzen.
- Zeile 19-50: Die komplette `accept`-Schleife, in der eine Client-Anfrage nach der anderen verarbeitet wird.
- Zeile 19: In der `while`-Bedingung wird die Client-Anfrage akzeptiert. Dadurch steht innerhalb des Schleifenrumpfs eine `TCPSocket`-Instanz namens `client` zur Verfügung, die die Kommunikation mit dem Client ermöglicht.

---

7 Achten Sie darauf, dass Sie das Skript auf einem UNIX-System in diesem Fall als `root` ausführen müssen. Das sollten Sie aus Sicherheitsgründen nur bei geschlossener Internetverbindung oder hinter einer Firewall tun. Außerdem wird der Start des Servers fehlschlagen, wenn Port 7 bereits durch einen »offiziellen« ECHO-Server belegt ist.

- Zeile 21: Der Text der Client-Anfrage wird ausgelesen. Der Server verlässt sich darauf, dass der Client nicht nur die Verbindung herstellt, sondern auch Text sendet. Die String-Methode `strip`, die auf `gets` angewendet wird, entfernt sämtlichen Whitespace vor und hinter dem Text. Die explizite Erwähnung von `STDIN` schließlich ist sicherer, damit das Skript nicht versehentlich aus dem Socket liest – unter Umständen könnte die zuletzt geöffnete `IO`-Instanz mit Lesebereitschaft nämlich als Standard betrachtet werden.
- Zeile 23: Der Server gibt den Text der Anfrage zur Kontrolle aus. Bessere Server verwenden dafür eine Logdatei und fügen mindestens die IP-Adresse des Clients sowie Datum und Uhrzeit hinzu. Wie dies gemacht wird, erfahren Sie im letzten Abschnitt dieses Kapitels, wo ein richtiger kleiner Webserver vorgestellt wird.
- Zeile 25: Wenn der »ausgeschnittene« Textinhalt der Anfrage `"EXIT"` lautet, beendet der Server die `accept`-Schleife per `break`-Anweisung. Damit lässt sich der Server von einem entfernten Rechner aus beenden, wie Sie weiter unten im Praxistest sehen werden. In der Praxis sollten daher höhere Ansprüche an Server-Administrationsbefehle gestellt werden – etwa die Kenntnis eines speziellen, konfigurierbaren Passworts.
- Zeile 27-47: In einer `case-when`-Fallentscheidung wird geprüft, ob ein Client einen der drei Moduswechsel-Befehle gesendet hat.
- Zeile 28-30: Wenn der Befehl `"MODE_NORMAL"` empfangen wurde, wird die Variable `mode` auf `NORMAL` gesetzt. Der Client erhält eine entsprechende administrative Meldung als Antwort.

Der hier gezeigte Zeilenabschluss mit `"\r\n"` ist für Netzwerkanwendungen sehr wichtig. Wie Ihnen vielleicht bekannt ist, verwenden die verschiedenen Betriebssysteme unterschiedliche Zeichen zur Markierung eines Zeilenumbruchs: Alle UNIX-Varianten benutzen den einfachen Zeilenvorschub (line feed, ASCII-Code 10) oder `"\n"`. Windows benutzt Wagenrücklauf (carriage return, ASCII 13) *und* Zeilenvorschub, also die Sequenz `"\r\n"`. Das klassische Mac OS bis Version 9 schließlich verwendet nur einen Wagenrücklauf.

Für TCP/IP-Netzwerkanwendungen wird erstaunlicherweise die »Windows-Sequenz« eingesetzt, obwohl diese Protokolle aus der UNIX-Welt stammen. Deshalb müssen Sie `"\r\n"` in TCP-Clients und -Servern explizit hinschreiben, damit sie auf jeder Plattform funktionieren. In lokalen Anwendungen wird ein einfaches `"\n"` dagegen automatisch in das Zeilenende des jeweiligen Systems konvertiert.

- Zeile 31-36: Die Modus-Wechselbefehle `"MODE_REVERSE"` beziehungsweise `"MODE_ROT13"` werden auf dieselbe Weise verarbeitet. Beachten Sie, dass der Moduswechsel, den ein Client übermittelt, ab sofort für *alle* Clients gilt.
- Zeile 39-46: Wenn keiner der speziellen Befehle empfangen wurde, kann der Inhalt der Client-Anfrage als einfacher Text für die ECHO-Verarbeitung interpretiert werden. Eine verschachtelte `case-when`-Struktur erzeugt die zum aktuellen Modus passende Antwort; die jeweilige Vorgehensweise entspricht dem objektorientierten Textmanipulierer aus Kapitel 2 und 4.
- Zeile 52: Nachdem die `accept`-Schleife per `break` verlassen wurde, wird der Server-Socket geschlossen. Dies geschieht am Skriptende zwar ohnehin automatisch, aber da Sie das Skript jederzeit verlängern könnten, sollten Sie unbenötigte Ressourcen der Ordnung halber stets schließen.

In Beispiel 5-2 sehen Sie den Code des entsprechenden Clients. Anschließend folgt die übliche stufenweise Erläuterung, und danach wird die Zusammenarbeit der beiden Komponenten im praktischen Einsatz gezeigt.

Beispiel 5-2: Der TCP-ECHO-Client, echoclient.rb

```
 1  require "socket"

 2  # Host und Port von der Kommandozeile lesen
 3  # oder auf localhost und 7 setzen
 4  if(ARGV[0])
 5    host = ARGV[0]
 6  else
 7    host = "localhost"
 8  end
 9  if(ARGV[1])
10    port = ARGV[1].to_i
11  else
12    port = 7
13  end

14  # Konfiguration ausgeben
15  printf "Server: %s, Port: %d\n\n", host, port

16  # Zunaechst Endlosschleife
17  loop do
18    print "Text: "
19    # Text von STDIN lesen, Zeilenumbruch entfernen
20    text = STDIN.gets.chomp
21    # Client beenden, falls "quit" eingegeben wurde
22    break if text == "quit"
23    # Client-Socket mit Host und Portnummer erzeugen
24    conn = TCPSocket.new(host, port)
25    # Anfrage an den Server senden
26    conn.printf "%s\r\n", text
```

Beispiel 5-2: Der TCP-ECHO-Client, echoclient.rb (Fortsetzung)

```
27     # Bei "EXIT" auch den Client beenden
28     break if text == "EXIT"
29     # Server-Antwort empfangen
30     antwort = conn.read
31     # Socket schliessen
32     conn.close
33     # Antwort ausgeben
34     puts "Antwort: #{antwort}"
35   end
```

Der Client besteht aus folgenden Arbeitsschritten:

- Zeile 1: Auch hier muss die Bibliothek *socket* importiert werden.
- Zeile 4-13: Hostname und Server-Portnummer werden aus den ersten beiden Kommandozeilenargumenten gelesen oder behelfsweise auf `"localhost"` beziehungsweise Port 7 gesetzt.
- Zeile 15: Zur Kontrolle werden Host und Port ausgegeben.
- Zeile 17-35: Der Client nimmt nun in einer Schleife Benutzereingaben entgegen, sendet sie an den Server und empfängt dessen jeweilige Antwort.
- Zeile 20: Einlesen der Benutzereingabe von `STDIN`.
- Zeile 22: Wenn die Eingabe `"quit"` lautet, wird nur der Client beendet, indem die Schleife per `break` verlassen wird. Der Server läuft dagegen unbehelligt weiter.
- Zeile 24: Ein `TCPSocket` wird erzeugt, um eine Verbindung zum Server herzustellen.
- Zeile 26: Der Text der Client-Anfrage wird an den Server gesendet, und zwar auch ein eventuelles Kommando.
- Zeile 28: Falls der eingegebene Text `"EXIT"` war, wird der Client nun ebenfalls beendet – im Unterschied zu `"quit"` aber erst, nachdem dieser Befehl an den Server geschickt wurde.
- Zeile 30: Mit Hilfe von `read` wird eine beliebig lange Server-Antwort ausgelesen. Auch der Client verlässt sich an dieser Stelle darauf, dass der Server tatsächlich Text liefert – falls dies nicht der Fall sein sollte, blockiert der Client. Sie müssen sich bei solchen einfachen Client-Server-Anwendungen deshalb stets genau über das Protokoll und dabei speziell über die Sende- und Empfangs-Reihenfolge im Klaren sein.
- Zeile 32: Der Socket wird geschlossen, weil dieser Durchgang beendet ist.
- Zeile 34: Zum Schluss wird die Antwort des Servers einfach ausgegeben.

Nur die spezielle Architektur der hier gezeigten einfachen Netzwerkanwendung ermöglicht die »gleichzeitige« Verarbeitung mehrerer Client-Verbindungen: Der Client baut die Verbindung nämlich erst *nach* erfolgter Benutzereingabe auf. Daraufhin sendet und empfängt er schnell und beendet die Verbindung sofort wieder.

Wenn Sie stattdessen eine einzelne, dauerhafte Verbindung bevorzugen, müssen weitere Clients entweder warten, oder der Server muss so umgebaut werden, dass er für jeden Client eine neue Instanz seiner selbst zur Verfügung stellt. Dies ist Gegenstand des übernächsten Abschnitts.

Nun ist es Zeit, diese Client-Server-Anwendung zu testen. Öffnen Sie dazu drei Terminalfenster und navigieren Sie jeweils in das Verzeichnis, in dem sich die Anwendung befindet. Starten Sie im ersten Terminal den Server mit einem beliebigen (verfügbaren) Port. Zum Beispiel:

```
> ruby echoserver.rb 7007
ECHO Server listening on port 7007 ...
```

Wechseln Sie in das zweite Terminalfenster und starten Sie darin den Client mit dem Hostnamen `localhost` und derselben Portnummer (oder einfach ohne Parameter, wenn der Server auf Port 7 lauscht):

```
> ruby echoclient.rb localhost 7007
Server: localhost, Port: 7007
```

Starten Sie im dritten Fenster ebenfalls den Client mit denselben Host- und Port-Daten. Wenn Sie Zugriff auf einen weiteren Rechner im lokalen Netz haben, der Ruby enthält, können Sie den Client dorthin kopieren und ebenfalls starten. Beachten Sie nur, dass der Server-Hostname auf einem entfernten Client natürlich nicht `localhost` lautet. Stattdessen müssen Sie die IP-Adresse des Servers eingeben (in manchen LANs sind auch Hostnamen definiert):

```
> ruby echoclient.rb 192.168.0.10 7007
Server: 192.168.0.10, Port: 7007
```

Geben Sie nun in jeden Client einen beliebigen Text ein. Wie Sie sehen, wird er im Server-Fenster protokolliert und unverändert an den jeweiligen Client zurückgesendet. Probieren Sie als Nächstes die drei Befehle `MODE_REVERSE`, `MODE_ROT13` und `MODE_NORMAL` aus. Sie werden – wie oben beschrieben – sehen, dass die Änderung stets alle Clients betrifft.

Beenden Sie zuletzt einen der Clients mit `quit`. Geben Sie im anderen `EXIT` ein, um den Server aus der Ferne zu beenden.

In Abbildung 5-3 sehen Sie den hier geschilderten Ablauf (ohne den externen Client) im Überblick.

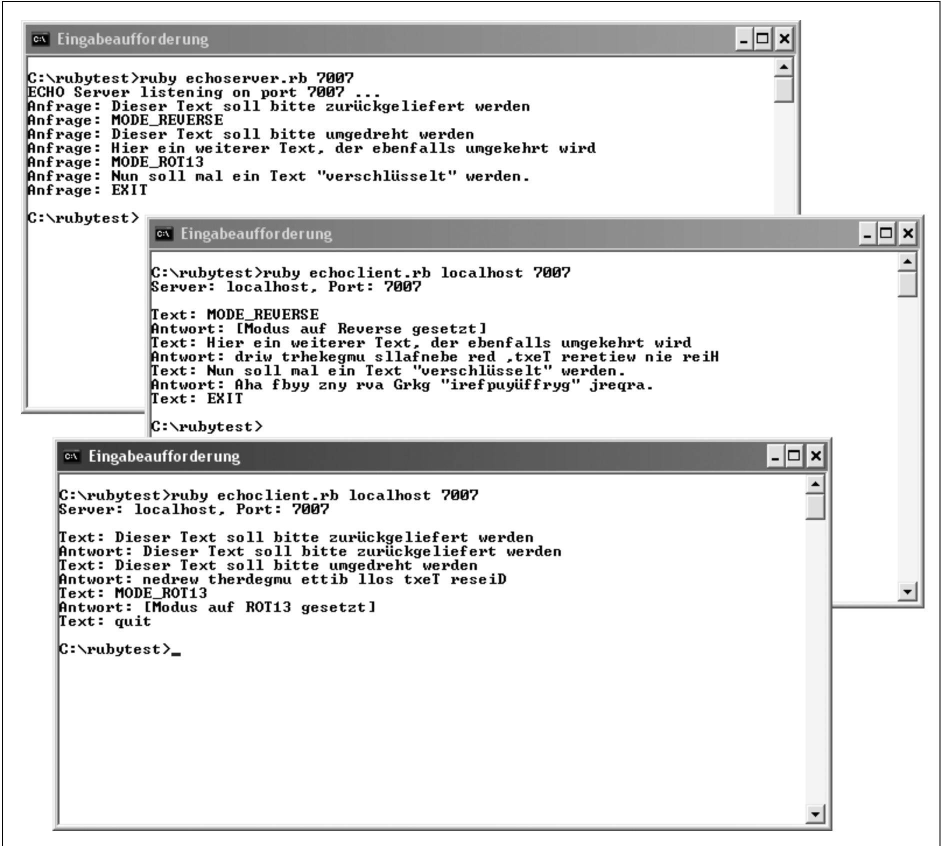

Abbildung 5-3: Die Zusammenarbeit des ECHO-Servers (oberstes Fenster) mit zwei Clients

Web-Clients mit Net::HTTP

Wenn es darum geht, Clients für Internet-Standardprotokolle zu schreiben, brauchen Sie sich in der Regel nicht die Mühe zu machen, sie auf der Basis von `TCPSocket` manuell zu implementieren. Ruby enthält nämlich bereits ab Werk Bibliotheken für den Zugriff auf die wichtigsten dieser Dienste. Sie befinden sich im Unterverzeichnis *net* der Ruby-Standardbibliothek und werden daher als *net/Bibliothek* eingebunden, während die entsprechenden Klassen *Net::Protokoll* heißen. Als Beispiel wird hier der Einsatz von `Net::HTTP` für den Zugriff auf Webserver erläutert; andere wichtige Bibliotheken sind `Net::FTP`, `Net::SMTP` (E-Mail-Versand) oder `Net::Telnet` (Terminal-Emulation).

Net::HTTP-Grundlagen

Um `Net::HTTP` zu verwenden, müssen Sie die Bibliothek zunächst importieren:

```
require "net/http"
```

Danach können Sie eine `Net::HTTP`-Instanz erzeugen und mit ihrer Hilfe eine Verbindung zu einem Webserver herstellen. Zum Beispiel:

```
webclient = Net::HTTP.new("buecher.lingoworld.de")
```

Als optionales zweites Argument können Sie eine Portnummer angeben; standardmäßig – wie bei Webservern üblich – 80.

Die neue Client-Instanz `webclient` besitzt unter anderem verschiedene Methoden, die den diversen HTTP-Anfragemethoden entsprechen – unter anderem `get`, `post` und `head` (Einzelheiten finden Sie im nächsten Kapitel). Die wichtigste dieser Methoden ist `get`: Sie fordert eine Ressource vom Server an, holt die Antwort ab und speichert sie in einer komplexen Datenstruktur. Das folgende Beispiel lädt die Startseite der Website zum vorliegenden Buch:

```
response = webclient.get("/ruby/")
```

Die Datenstruktur `response` besitzt je nach Antwort des Servers einen leicht unterschiedlichen Datentyp, den Sie mit `response.class` ermitteln können. Wenn eine Seite erfolgreich geladen werden konnte, erhalten Sie:

```
>> response.class
=> Net::HTTPOK
```

Wichtige Eigenschaften von `response` (unabhängig vom konkreten Typ) sind vor allem folgende Felder:

- `response.code` (String): der Statuscode der Antwort – etwa 200 für eine korrekt geladene Seite, 404 für eine nicht verfügbare Seite oder 301 für eine Weiterleitung.
- `response.message` (String): die passende Textmeldung zur Statusnummer – zum Beispiel `"OK"` für den Status 200.
- `response.body` enthält den Body, das heißt den eigentlichen Dateninhalt der Antwort.
- Zusätzlich enthält `response` eine hashartige Struktur mit sämtlichen Headern der HTTP-Antwort, in der die Schlüssel die Header-Namen bilden. Sie können sie beispielsweise wie folgt auslesen:

  ```
  response.each_key { |h|
    printf "%s: %s\n", h, response[h]
  }
  ```

  Für unser Beispiel *http://buecher.lingoworld.de/ruby/* liefert dieser Code folgende Header (einige werden im nächsten Kapitel genauer erklärt):

  ```
  last-modified: Sat, 04 Nov 2006 11:43:39 GMT
  ```

```
connection: close
content-type: text/html
etag: "15dc535-1582-454c7ceb"
date: Mon, 18 Dec 2006 22:16:02 GMT
server: Apache/1.3.33 (Unix)
content-length: 5506
accept-ranges: bytes
```

In `response.body` befindet sich – wie erwähnt – der Dateninhalt der gelieferten Ressource – bei `200 OK` die gewünschte Datei, bei den meisten anderen Statuscodes dagegen eine kurze Fehlermeldung im HTML-Format. Bei einem Status wie 404 können Sie nichts weiter unternehmen; die Seite ist nicht vorhanden – möglicherweise haben Sie sich beim Aufruf verschrieben, oder der Betreiber hat die Seite aus dem Netz genommen.

Interessant sind dagegen die 30x-Statuscodes: Der Client erhält die Mitteilung, dass sich die Seite an einem anderen Ort befindet; die neue URL steht im Header `Location`. Browser verfolgen solche Weiterleitungen automatisch, während Sie das Ihren eigenen Clients erst beibringen müssen. Eine Möglichkeit wäre eine Schleife wie diese, die auch mehreren Weiterleitungen automatisch folgt:

```
# Bei 30x-Statuscodes der Weiterleitung folgen
while response.code =~ /^30\d$/
  # Neue URL aus dem Header Location lesen
  url = response['location']
  puts "Verfolge Weiterleitung nach #{url}"
  # Neue URL anfordern
  response = webclient.get(url)
end
```

Sie können dies leicht ausprobieren, indem Sie die obige Site ohne den abschließenden Slash aufrufen, also */ruby* statt */ruby/* schreiben:

```
response = webclient.get("/ruby")
```

Der angesprochene Apache-Webserver sendet daraufhin eine Weiterleitungsmitteilung, und die obige Schleife liefert folgende Meldung:

```
Verfolge Weiterleitung nach http://buecher.lingoworld.de/ruby/
```

HTML-Einstieg in zwei Minuten

In diesem Abschnitt und in den nachfolgenden Kapiteln kommt relativ viel HTML zum Einsatz. Falls Sie sich noch nicht damit auskennen, sollten Sie das brillante Online-Tutorial *SelfHTML* von Stefan Münz (*http://de.selfhtml.org*) konsultieren oder ein Buch wie *HTML & XHTML – Das umfassende Referenzwerk* von Cuck Musciano und Bill Kennedy (O'Reilly Verlag, ISBN 978-3-89721-494-1) zu Rate ziehen. In diesem Kasten erfahren Sie nur das Allerwichtigste über HTML.

→

HTML ist eine klartextbasierte Auszeichnungssprache, in der Formatierungen und funktionale Elemente durch so genannte *Tags* gekennzeichnet werden. Ein Tag steht in `<spitzen Klammern>` und markiert den Beginn und das Ende einer bestimmten Eigenschaft der darin verschachtelten Elemente. Jedes Tag besteht aus einem öffnenden und einem schließenden Teil mit folgender Schreibweise:

```
<tag [attribut="wert" ...]>
   [...]
</tag>
```

Die eckigen Klammern stehen hier, wie in Syntaxschemata üblich, für optionale Teile. Das folgende Beispiel zeigt einen Textabsatz (*p* ist die Abkürzung für *paragraph*), in den ein wenig Text mit einem kursiven Wort (*i* für *italic*) verschachtelt ist:

```
<p>Eine kurze Einf&uuml;hrung in <i>HTML</i>.</p>
```

Die merkwürdige Zeichensequenz `ü` steht für den Umlaut ü – Sonderzeichen, die in HTML eine besondere Bedeutung haben (zum Beispiel `<` und `>`), sowie Nicht-ASCII-Zeichen werden durch diese so genannten *Entity-Referenzen* dargestellt. Auf der Website zum Buch finden Sie eine entsprechende Liste.

Für HTML-Tags, die keine weiteren Inhalte mehr umfassen, existiert eine spezielle Kurzschreibweise. Statt

```
<tag [attribut="wert" ...]></tag>
```

können Sie auch einfach Folgendes schreiben:

```
<tag [attribut="wert" ...] />
```

Das bekannteste Beispiel ist der einfache Zeilenumbruch:

```
<br />
```

Übrigens werden Sie im Web zahlreiche ältere HTML-Dokumente antreffen, die solche »Einfach-Tags« ohne den abschließenden Slash (`/`) verwenden. Das liegt daran, dass 1999 die HTML-Neufassung *XHTML* eingeführt wurde. Sie besitzt eine strengere Syntax als ältere HTML-Versionen, weil sie *XML*-kompatibel ist (Informationen über XML erhalten Sie im nächsten Kapitel). Ohne hier näher ins Detail zu gehen: Gewöhnen Sie sich den End-Slash für leere Tags grundsätzlich an, damit Ihre HTML-Dokumente zukunftssicher sind.

Jedes HTML-Dokument besitzt die folgende Grundstruktur:

```
<html>
  <head>
    <title>Titel des Dokuments</title>
  </head>
  <body>
    Sichtbarer Inhalt des Dokuments
  </body>
</html>
```

→

Einrückungen und Zeilenumbrüche spielen übrigens keine Rolle; zur besseren Lesbarkeit ist es aber ratsam, sie auf die hier gezeigte Weise anzuwenden.

Das Tag-Paar `<html>` und `</html>` umschließt den gesamten Inhalt; es handelt sich um das so genannte *Wurzelelement* des Dokuments. Innerhalb dieses Rahmens besteht das HTML-Dokument aus zwei Teilen: *Header* und *Body*. Der Header kann allerlei technische Informationen über das Dokument enthalten, die für Webserver, Proxies oder Browser interessant sind. Die einzige verbindliche Angabe ist der hier gezeigte Dokumenttitel `<title>...</title>`; er erscheint unter anderem in der Titelleiste des Browserfensters. Der Body enthält dagegen den sichtbaren Inhalt des Dokuments, der im Browser angezeigt wird.

Anwendungsbeispiel: Ein kleiner Textbrowser

Die Implementierung eines ausgewachsenen Webbrowsers ist äußerst kompliziert – Sie benötigen eine grafische Oberfläche und vor allem leistungsfähiges HTML-Rendering, das selbst aus zweifelhaftem Quellcode die bestmögliche Webseite zurechtformatiert. Dennoch ist es ein spannendes Projekt, einen eigenen Mini-Browser zu schreiben. Das hier vorgestellte Skript besitzt folgende Eigenschaften:

- Nach Eingabe einer URL wird diese angefordert. Weiterleitungen werden verfolgt, wie im vorigen Abschnitt gezeigt.
- Der erhaltene HTML-Code wird einigen »Säuberungs«- und Formatierungsschritten unterzogen.
- Sämtliche Hyperlinks werden extrahiert und mit einer Nummerierung versehen. Nach Abschluss einer Seite können Sie entweder eine völlig neue URL eingeben, oder Sie tippen eine der Nummern ein, um einem Link zu folgen.
- Die Ausgabe erfolgt seitenweise und mit Wortumbruch. Dazu wird zunächst die unabhängig einsetzbare Klasse `WrapPager` entwickelt.

Das Programm hat mithin nicht einmal den Komfort »richtiger« Textbrowser wie Lynx (*http://lynx.browser.org/*). Diese haben nämlich den Vorteil, dass sie nicht zeilenorientiert arbeiten, sondern den Terminalbildschirm kontrollieren. Auf diese Weise können Sie zum Beispiel beliebig zurück- und vorblättern oder Links mit der Tastatur ansteuern.

Einen Vorteil hat die hier vorgestellte Lösung aber doch: Dieser kleine Browser ist wahrscheinlich schneller als alle, die Sie jemals ausprobiert haben!

In Beispiel 5-3 sehen Sie zunächst den Quellcode von *wrappager.rb*, also die Klasse zur formatierten, seitenweisen Ausgabe.

Beispiel 5-3: Eine Klasse für Wortumbruch und seitenweise Ausgabe, wrappager.rb

```
 1  class WrapPager

 2    def to_page(text)

 3      # Die Hauptersetzung: nach max. 70 Zeichen
 4      # an einer Wortgrenze umbrechen
 5      text.gsub!(/(.{1,70})(\s+|\Z)/, "\\1\n")
 6      # Die einzelnen Zeilen in einem Array ablegen
 7      textlines = text.split("\n")
 8      # Aktuelle Bildschirmzeilennummer
 9      line = 0
10      # Solange noch Zeilen vorhanden sind ...
11      while (textlines != [])
12        # Solange der Bildschirm nicht voll ist ...
13        while (line < 22)
14          # Aktuelle Zeile holen
15          l = textlines.shift
16          # Ende, falls keine Zeile mehr vorhanden
17          if l == nil
18            ende = true
19            break
20          end
21          # Zeile ausgeben
22          puts l
23          # Bildschirmzeilenzaehler erhoehen
24          line += 1
25        end
26        # Abbruch, falls Textende gefunden wurde
27        break if ende
28        # Ausgabe der Fusszeile
29        print "\n\t----- WEITER: [ENTER]; ENDE: [Q][ENTER] -----"
30        weiter = STDIN.gets.chomp
31        # Abbruch, falls q eingegeben wurde
32        break if weiter == "q"
33        # Zaehler zuruecksetzen
34        line = 0
35      end
36    end

37  end
```

Eine zeilenweise Beschreibung entfällt an dieser Stelle ausnahmsweise. Lesen Sie den Quellcode in Ruhe; im Prinzip müssten Ihnen die einzelnen Anweisungen des Skripts bekannt sein. Beachten Sie besonders den eigentlichen Wortumbruch (Zeile 5):

```
text.gsub!(/(.{1,70})(\s+|\Z)/, "\\1\n")
```

Der erste geklammerte Ausdruck besagt, dass 1 bis 70 beliebige Zeichen gewählt werden sollen. Danach folgt entweder beliebig viel Whitespace (`\s+`) oder das Textende (`\Z`). Als Ersatztext dient der Inhalt des ersten Klammerausdrucks, `\\1`, gefolgt von einem Zeilenumbruch. Das Verfahren funktioniert aufgrund der in Kapitel 2

beschriebenen »Greediness« der Regexp-Quantifizierer: Der Teilausdruck `.{1,70}` macht nicht beim ersten, sondern beim letzten möglichen Auftreten von Whitespace halt.

Wenn Sie diese Klasse unabhängig ausprobieren möchten, können Sie das folgende kurze Skript (*wraptest.rb*) verwenden, das eine auf der Kommandozeile angegebene Datei nach dem beschriebenen Schema anzeigt:

```
require "wrappager.rb"

if !ARGV[0]
  STDERR.puts "Verwendung: #{$0} Datei"
  exit(1)
end
filename = ARGV[0]
file = File.new(filename, "r")
text = file.read
file.close

wp = WrapPager.new
wp.to_page(text)
```

Anregung: Schreiben Sie eine erweiterte Fassung von *wrappager.rb*, in der sich die maximalen Zeilen- und Spaltenanzahlen als Parameter angeben lassen.

Nachdem die gut formatierte Anzeige funktioniert, wird es Zeit für den eigentlichen Textbrowser. In Beispiel 5-4 sehen Sie zunächst den gesamten Quellcode.

Beispiel 5-4: Der Ruby-Textbrowser, httpclient.rb

```
 1  require "net/http"
 2  require "wrappager.rb"

 3  # URL in Host, Port und Ressource zerlegen
 4  def parse_url(url)
 5    if url.match(%r|([^/]+)(/.*)|)
 6      host = $1
 7      uri = $2
 8    end

 9    # Gegebenenfalls Host und Portnummer trennen
10    if host =~ /:/
11      (host, port) = host.split(":")
12      port = port.to_i
13    else
14      port = 80
15    end

16    # Alle drei Komponenten zurueckgeben
17    [host, port, uri]
18  end
```

Beispiel 5-4: Der Ruby-Textbrowser, httpclient.rb (Fortsetzung)

```
19  # Einen WrapPager erzeugen
20  wrappager = WrapPager.new

21  # Hash zur Umsetzung wichtiger Entity-Referenzen
22  entities = {" " => " ", "&lt;" => "<", "&gt;" => ">",
23              "&quot;" => "\"", "&amp;" => "&",
24              "&auml;" => "ae", "&ouml;" => "oe", "&uuml;" => "ue",
25              "&Auml;" => "Ae", "&Ouml;" => "Oe", "&Uuml;" => "Ue",
26              "&szlig;" => "ss"}

27  begin
28    print "URL => "
29    url = gets.chomp
30  end while url == ""

31  # Hauptschleife
32  loop do
33    # / anfuegen, falls URL keinen enthaelt
34    if url !~ %r|/|
35      url += "/"
36    end

37    # URL zerlegen
38    (host, port, uri) = parse_url(url)

39    puts "Hole #{uri} von #{host}:#{port}"

40    begin
41      # Client erzeugen
42      httpclient = Net::HTTP.new(host, port)
43      # Server-Antwort holen
44      response = httpclient.get(uri)

45      # Bei 30x-Statuscodes der Weiterleitung folgen
46      while response.code =~ /^30\d$/
47        # Neue URL aus dem Header Location lesen
48        location = response['location']
49        # Absolute URL gesondert behandeln
50        if location =~ %r|^http://(.*)|
51          location = $1
52          (host, port, uri) = parse_url(location)
53          httpclient = Net::HTTP.new(host, port)
54        else
55          uri = location
56        end
57        puts "Verfolge Weiterleitung nach #{uri}"
58        # Neue URL anfordern
59        response = httpclient.get(uri)
60      end
```

Beispiel 5-4: Der Ruby-Textbrowser, httpclient.rb (Fortsetzung)

```
61        # Eventuell Fehlermeldung ausgeben
62        if response.code != "200"
63          printf "FEHLER: %s %s\n", response.code, response.message
64          puts
65        end

66        # Linkzaehler und -liste zuruecksetzen
67        linkcounter = 0
68        linksammlung = []

69        # Es koennen nur Text-Seiten angezeigt werden
70        ctype = response['content-type']
71        if  ctype !~ %r|text/|
72          puts "RESSOURCE NICHT DARSTELLBAR"
73          puts
74          puts "Der Typ dieser Ressource ist #{ctype}."
75          puts "Dieser Browser kann leider nur Text anzeigen."
76          puts
77          print "URL => "

78        else

79          body = response.body

80          # Seiteninhalt weiterverarbeiten (7 Schritte)

81          # 1. Hyperlinks in Linksammlung aufnehmen
82          body.gsub!(%r|<a href="([^"]+)">(.*?)</a>|mi) {
83            linksammlung.push($1)
84            linkcounter += 1
85            "=> #{$2} [#{linkcounter}]"
86          }

87          # 2. Wichtige Entity-Referenzen ersetzen
88          entities.each_pair { |ent, repl|
89            body.gsub!(ent, repl)
90          }

91          # 3. Ueberschriften grossschreiben
92          body.gsub!(/<h[1-6]>(.*?)<\/h[1-6]>/mi) {
93            $1.upcase + "\n"
94          }

95          # 4. Zeilenumbrueche bei <br />, <p>, <div> und <tr> setzen
96          body.gsub!(%r|<br\s*/?>|i, "\n")
97          body.gsub!(/<p.*?>/i, "\n")
98          body.gsub!(/<div.*?>/i, "\n")
99          body.gsub!(/<tr.*?>/i, "\n")
```

```
100         # 5. Tabellenzelle => Tabulator
101         body.gsub!(/<td.*?>/i, "\t")

102         # 6. Sonstiges HTML entfernen
103         body.gsub!(/<.*?>/m, "")

104         # 7. Whitespace zusammenfassen
105         body.gsub!(/^\s*\n$/, "\n")
106         body.gsub!(/\n{3,}/m, "\n\n")
107         body.gsub!(/ {3,}/, "  ")
108         body.gsub!(/\t+/, "\t")

109         # Linkliste anfuegen
110         if linksammlung.length > 0
111           body += "\nENTHALTENE LINKS:\n\n"
112           linkcounter = 0
113           linksammlung.each { |link|
114             linkcounter += 1
115             body += "#{linkcounter}. #{link}\n"
116           }
117         end

118         # Body ausgeben
119         wrappager.to_page(body)

120         # Neue Eingabeaufforderung
121         if linksammlung.length > 0
122           print "LINK# oder URL => "
123         else
124           print "URL => "
125         end
126       end
127     rescue
128       puts "Fehler: #{$!}"
129       puts "(Pruefen Sie den eingegebenen Hostnamen)"
130       puts
131       print "URL => "
132     end

133     begin
134       newurl = gets.chomp
135     end while newurl == ""

136     # Programmende bei Eingabe Q
137     break if newurl == "q"

138     if newurl =~ /^[0-9]+$/
139       # URL bilden, falls Linknummer eingegeben
140       newurl = linksammlung[newurl.to_i - 1]
```

Beispiel 5-4: Der Ruby-Textbrowser, httpclient.rb (Fortsetzung)

```
141        if newurl =~ /^#/
142          # Nichts tun, URL bleibt
143        elsif newurl =~ %r|^http://(.*)|
144          # http:// entfernen
145          url = $1
146        elsif newurl =~ %r|^/|
147          # Host und absoluten Pfad kombinieren
148          url = host + newurl
149        else
150          # Host und relativen Pfad kombinieren
151          url.gsub!(%r|(.*)/.*|, "\\1")
152          url += "/" + newurl
153        end
154      else
155        # Normale URL-Eingabe uebernehmen
156        url = newurl
157      end

158    end
```

Starten Sie den kleinen Browser und geben Sie eine beliebige URL (ohne *http://*) ein. Bei einem reinen Hostnamen ohne Ressourcenteil – etwa *www.rubygarden.org* – wird automatisch ein / für die absolute Startseite angefügt. Nun wird die Seite geladen, und Sie können mit **Enter** seitenweise vorwärts blättern oder die Anzeige mit **Q** und **Enter** vorzeitig beenden. Nach dem eigentlichen Seitentext wird eine Liste aller verfügbaren Links angezeigt. An dieser Stelle haben Sie drei verschiedene Eingabemöglichkeiten:

- Die Nummer eines der Hyperlinks
- Eine neue URL
- **Q** zum Beenden des Programms

In Abbildung 5-4 sehen Sie als Beispiel die erste und die letzte Bildschirmseite (das Ende der Linkliste) nach Eingabe der URL *www.oreilly.de*.

Hier die übliche Beschreibung der einzelnen Arbeitsschritte des Skripts:

- Zeile 1-2: Import der benötigten Bibliotheken. Neben *net/http* aus der Standardbibliothek wird der weiter oben beschriebene Pager *wrappager.rb* importiert; diese Datei muss sich dazu im selben Verzeichnis befinden wie *httpclient.rb* selbst.
- Zeile 4-18: Die Methode `parse_url` wird zum Zerlegen von Anfrage-URLs bereitgestellt. Sie erwartet als Argument einen String mit folgendem Schema:

  ```
  Host[:Port]/Pfad/der/Ressource
  ```

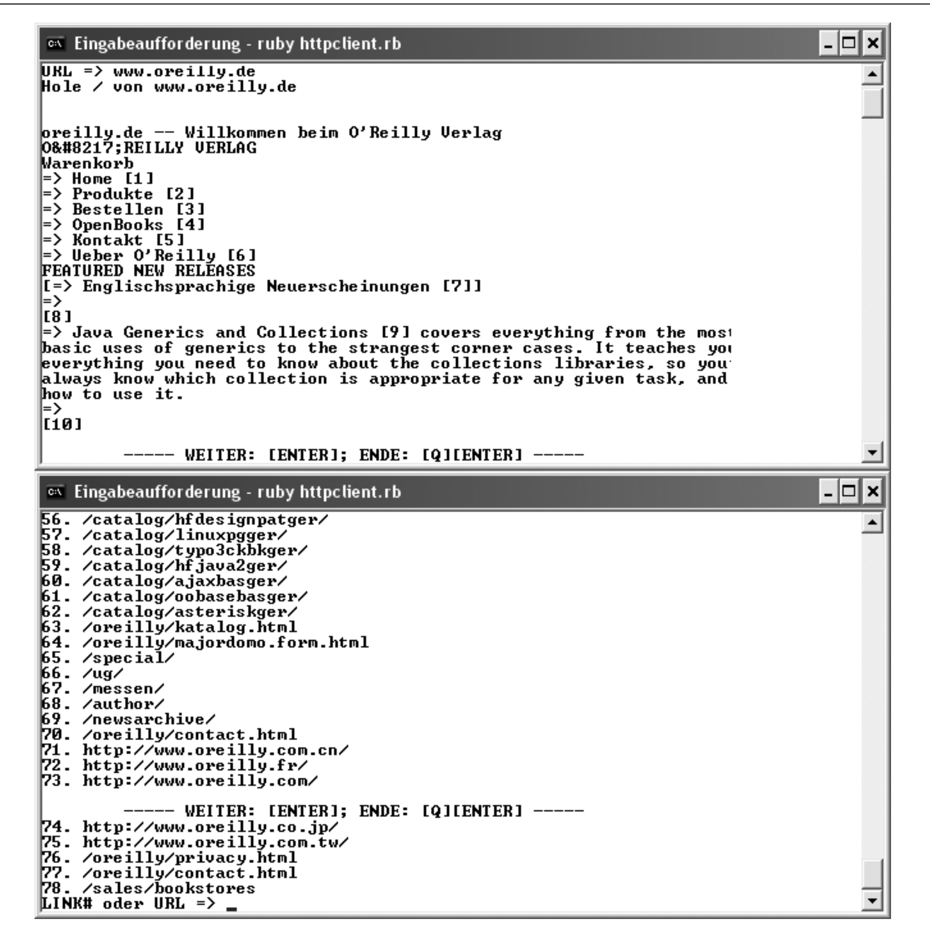

Abbildung 5-4: Der kleine Ruby-Textbrowser im Einsatz (erster und letzter Bildschirm von www.oreilly.de)

Gültige Beispiele wären etwa `buecher.lingoworld.de/ruby/` oder `localhost:8000/seite.html`.

- Zeile 5-8: Aus der URL werden der Host, eventuell mit Portnummer (alle Zeichen vor dem ersten `/`), und der Pfad (der gesamte Rest des Strings) extrahiert.
- Zeile 10-15: Wenn der Hostname einen Doppelpunkt enthält, wird die dahinter befindliche Portnummer abgetrennt. Ansonsten wird der HTTP-Standardport 80 festgelegt.
- Zeile 17: Die drei Werte Host, Port und Pfad werden zum Schluss als Array zurückgegeben.
- Zeile 20: Da der `WrapPager` zur Anzeige jeder geladenen Webseite benötigt wird, wird an dieser Stelle ein für alle Mal eine Instanz dieser Klasse erzeugt.

- Zeile 22-26: Der hier definierte Hash `entities` enthält die wichtigsten Sonderzeichen, die während der Verarbeitung des HTML-Codes ersetzt werden. Aufgrund der in Kapitel 2 beschriebenen Zeichensatzinkompatibilität zwischen Windows und seiner Eingabeaufforderung werden für die deutschen Umlaute die Ersetzungen *ae*, *oe* und so weiter festgelegt.
- Zeile 27-30: Eingabe der ersten URL. Um völlig leere Eingaben abzufangen, wurde der Vorgang in eine Schleife eingebettet. Dieser spezielle Schleifentyp – eine *fußgesteuerte Schleife* – prüft die Bedingung erst nach dem Schleifenrumpf, so dass zumindest der erste Durchgang stattfinden muss. In Ruby erreichen Sie das durch die Kombination eines `begin`-`end`-Blocks mit einem nachgestellten `while`.
- Zeile 32-158: Die Hauptschleife des Programms, in der eine Seite nach der anderen angefordert werden kann. Sie ist als Endlosschleife konzipiert und kann später durch die Eingabe von **Q** verlassen werden.
- Zeile 34-36: Sollte die Eingabe gar keinen Slash (`/`) enthalten, ist davon auszugehen, dass es sich um einen reinen Hostnamen handelt. Da dieser keine Seite zum Anfordern enthält, wird `/` als URL-Pfad der Startseite angehängt. Der reguläre Ausdruck selbst wird – wie noch oft in diesem Skript – als `%r|...|` statt `/.../` geschrieben, um das Escaping des Zeichens `/` zu vermeiden.
- Zeile 38: Ein Aufruf von parse_url zerlegt die URL in die drei Komponenten Host, Port und Pfad.
- Zeile 40-132: Anforderung, Auslesen und Verarbeiten der Antwort werden in einen Block verpackt, um einen eventuellen Fehler abzufangen – insbesondere einen nicht existierenden Host.
- Zeile 42-44: Eine neue `Net::HTTP`-Instanz wird erzeugt; als Argumente dienen der extrahierte Hostname und die Portnummer. Anschließend wird per `GET`-Anfrage die gewünschte Ressource geladen.
- Zeile 46-60: Solange die Server-Antwort einen 30x-Statuscode besitzt, folgt der Browser automatisch der Weiterleitung.
- Zeile 48: Aus dem Header `Location` wird die URL des Weiterleitungsziels ausgelesen.
- Zeile 50-53: Manche Weiterleitungen besitzen eine absolute URL wie *http://andere-site.de/seite*. Wenn dies der Fall ist, muss diese URL erneut zerlegt werden. Anschließend wird aus der neuen URL ein `Net::HTTP`-Objekt mit Host und Port erzeugt.
- Zeile 55: Sollte das Weiterleitungsziel dagegen nur ein URL-Pfad wie */seite* sein, kann dieser Pfad sofort verwendet werden.
- Zeile 59: In jedem Fall wird nun das Weiterleitungsziel geladen. Falls es sich erneut um eine Weiterleitung handelt, findet der nächste Schleifendurchlauf statt.

- Zeile 62-65: Falls der Statuscode – gegebenenfalls nach der letzten Weiterleitung – nicht 200 sein sollte, ist offensichtlich ein Fehler aufgetreten. An dieser Stelle werden Statuscode und -meldung dieses Fehlers ausgegeben. Beachten Sie, dass die nachfolgende normale Verarbeitung der Ressource nicht in einem `else`-Block steht. Eine Fehlerseite wird somit genauso verarbeitet wie eine korrekte Lieferung; dies ist passend, weil die Fehlermeldungen der meisten Webserver ein HTML-Dokument mit einer Fehlerbeschreibung mitliefern.
- Zeile 67-68: An dieser Stelle werden ein ganzzahliger Zähler und ein leeres Array für die Linkliste initialisiert.
- Zeile 70-77: Als Nächstes wird der Header `Content-type` der HTTP-Antwort ausgewertet. Dieser gibt den Datentyp der gelieferten Ressource an. Da es sich bei diesem Skript um einen Textbrowser handelt, können nur Daten mit `text/*`-Typen verarbeitet werden, beispielsweise `text/html` (HTML-Code; der Standard bei Webseiten), `text/plain` (einfacher Text ohne Formatbefehle) oder `text/xml` (XML-Code). Sollte die Ressource einen anderen Typ haben – etwa `image/gif` für ein GIF-Bild oder `application/pdf` für ein PDF-Dokument –, dann wird an dieser Stelle eine Fehlermeldung angezeigt.
- Zeile 78: Der Rest des Codes zur Ressourcenverarbeitung wird nun in einem `else`-Block ausgeführt, weil Nicht-Text-Ressourcen nicht weiterverarbeitet werden können.
- Zeile 79: Zur weiteren Verarbeitung wird der Body der Anfrage, also das gelieferte Textdokument, in einer Variablen gespeichert.
- Zeile 81-108: Hier wird eine Reihe von Transformationen durchgeführt, um Links zu extrahieren, den Text anhand bestimmter Begrenzer zu formatieren und die restlichen HTML-Tags zu entfernen. Die grundlegende Aufgabe jedes Arbeitsschritts wird in einem Kommentar zusammengefasst. Alle diese Operationen basieren auf regulären Ausdrücken, und wenn Sie Kapitel 2 aufmerksam durchgearbeitet haben, sollten Sie keine Schwierigkeiten haben, sie zu verstehen – gegebenenfalls können Sie dort noch einmal nachschlagen.

Die hier gezeigten Verarbeitungsschritte reichen »für den Hausgebrauch« – aus den meisten Webseiten sollten Sie einigermaßen vernünftig formatierten Text erhalten (falls diese überhaupt Text enthalten und nicht ganz auf Flash oder Bildern basieren). Es werden allerdings oft Stylesheet- und JavaScript-Code sowie einige unaufgelöste Entity-Referenzen übrig bleiben.

Ein professioneller Textbrowser verlässt sich deshalb zur Formatierung des Inhalts nicht auf ein paar reguläre Ausdrücke, sondern verwendet einen *HTML-Parser*, der das Dokument zunächst vollständig zerlegt. Mit der aus dieser Vorverarbeitung erhaltenen Datenstruktur kann man danach erheblich leichter arbeiten.

- Zeile 110-117: Nachdem der Dokumentinhalt verarbeitet wurde, wird die Linkliste zeilenweise nummeriert und an den Body angefügt. Vielleicht fragen Sie sich, warum sie nicht einfach ausgegeben wird? Das liegt daran, dass sie mit dem Body zusammen Teil der seitenweisen Ausgabe sein soll.
- Zeile 119: Der fertig verarbeitete Body samt eventueller Linkliste wird an den `WrapPager` übergeben, der ihn nach dem weiter oben erläuterten Verfahren seitenweise ausgibt. Der Benutzer kann entweder **Enter** drücken, um die jeweils nächste Seite zu lesen, oder **Q Enter** eingeben, um die Ausgabe vorzeitig abzubrechen.
- Zeile 121-125: Nach dem Ende der Ausgabe wird eine neue Eingabeaufforderung angezeigt. Falls Links vorhanden sind, wird nach einer Linknummer oder einer neuen URL gefragt, andernfalls nur nach einer URL.
- Zeile 127-131: Wenn beim Laden und bei der Verarbeitung der Webseite irgendein schwerwiegender Fehler auftritt – etwa wenn der angesprochene Host nicht erreichbar ist –, werden automatisch diese Anweisungen ausgeführt. Der User erfährt den Inhalt der Fehlermeldung (`$!`) und wird nach einer neuen URL gefragt.
- Zeile 133-135: Schleife zur Eingabe einer neuen URL oder einer Linknummer.
- Zeile 137: Falls die Eingabe `"q"` lautete, wird die Hauptschleife verlassen. Damit ist das Programm beendet.
- Zeile 138-152: Wenn es sich bei der Eingabe um eine Linknummer handelt, ist eine Reihe von Verarbeitungsschritten nötig.
- Zeile 140: Zunächst wird der passende Eintrag aus der Linkliste ausgelesen; da Arrays ab 0 zählen, muss die eingegebene Zahl um 1 vermindert werden.
- Zeile 141: Unterschätzen Sie das »leere« (anweisungslose) `if` nicht! Eine URL, die mit `#` beginnt, verweist auf eine Markierung innerhalb der aktuellen Seite; ein grafischer Browser scrollt an die entsprechende Stelle, wenn ein solcher Link angeklickt wird. Für den vorliegenden zeilenorientierten Browser hat diese Linksorte dagegen keine Bedeutung, so dass sie durch dieses `if` aus dem letzten `else`-Teil (URL relativ zum aktuellen Verzeichnis) herausgehalten werden muss.
- Zeile 143-145: Bei einer absoluten URL muss nur das führende *http://* entfernt werden.
- Zeile 146-148: Eine URL, die mit `/` beginnt, verweist auf die Wurzel des aktuellen Hosts. Deshalb wird dieser wieder vor dem Pfad eingefügt.
- Zeile 149-152: Alle anderen URLs werden als relativ zum aktuellen Verzeichnis betrachtet. Deshalb wird der Bereich ab dem letzten Slash aus der vorherigen URL entfernt; anschließend wird die Link-URL angefügt. Wenn die aktuelle

Seite etwa *www.site.de/info/index.html* ist und der Verweis *katalog.html* lautet, wird daraus auf diese Weise die vollständige neue URL *www.site.de/info/katalog.html*.

Alle Spezial-URLs, die dieser Browser gar nicht verarbeiten kann – etwa auf FTP-Servern oder E-Mail-Adressen –, fallen auch unter dieses letzte `else`. Sie werden automatisch durch das `rescue` im nächsten Schleifendurchlauf abgefangen.

- Zeile 154-156: Wenn die Eingabe keine Zahl ist, wird sie als normale URL übernommen und nach dem oben beschriebenen Schema im nachfolgenden Schleifendurchlauf verarbeitet.

Anregung: Spendieren Sie dem Textbrowser eine Bookmark-Verwaltung. Durch Eingabe von `b Stichwort` soll die zuletzt geladene Seite unter dem gewünschten Stichwort in der Bookmark-Datei gespeichert werden. Mit `g Stichwort` soll das betreffende Bookmark später wieder aufgerufen werden. l schließlich soll eine (seitenweise) Liste der verfügbaren Bookmarks in der Form `Stichwort: URL` anzeigen.

Prozesse und Threads

Bereits in der Socket-Einführung wurde darauf hingewiesen, dass praxistaugliche TCP-Server einen Mechanismus benötigen, um mehrere Client-Anfragen gleichzeitig zu bearbeiten. Auch andere Aufgaben lassen sich hervorragend parallelisieren – beispielsweise könnten Sie im Hintergrund automatisch einen Index der Texte erstellen, die Sie im Vordergrund eingeben. In diesem Abschnitt erfahren Sie, wie Sie Ruby dazu bringen, mehrere Arbeitsabläufe im schnellen Wechsel zu erledigen. Der Fachbegriff für Anwendungen, die dies können, heißt *Nebenläufigkeit* oder auch *Gleichzeitigkeit* (auf Englisch concurrency).

Neue Prozesse erzeugen und steuern

Beachten Sie zunächst, dass ein Computer Aufgaben nur dann *wirklich gleichzeitig* erledigen kann, wenn er mehrere Prozessoren besitzt. Dennoch sind Sie es von einem modernen PC gewohnt, dass viele Programme nebeneinander ausgeführt werden – Sie können problemlos zur selben Zeit eine Datei herunterladen, eine E-Mail schreiben und natürlich diverse Serverdienste zur Verfügung stellen.

Wenn Sie sich einen Überblick verschaffen möchten, was auf Ihrem Computer alles los ist, können Sie unter Windows den Task-Manager aufrufen. Klicken Sie dazu mit der rechten Maustaste auf die Taskleiste und wählen Sie *Task-Manager*. Auf der Registerkarte *Anwendungen* werden nur wenige Desktop-Programme angezeigt; eine Vorstellung von den wirklichen Verhältnissen liefert erst die Registerkarte

Prozesse. Ein *Prozess* ist eine von vielen Aufgaben, zwischen denen Ihr Betriebssystem permanent hin- und herwechselt.

Auf einem Linux-System können Sie sich die Prozessliste anzeigen lassen, indem Sie auf der Konsole

```
$ ps aux
```

eingeben. Senden Sie die Ausgabe am besten durch eine Pipe an `less`, da die Ausgabe höchstwahrscheinlich mehrere Seiten umfasst.

Bei anderen UNIX-Varianten lautet das korrekte Argument für die vollständige Prozessliste möglicherweise nicht `aux`. Geben Sie evenatuell `man ps` ein, um nachzulesen, wie Ihre lokale `ps`-Variante funktioniert.

Der Ruby-Interpreter, der eines Ihrer Skripten ausführt, ist selbstverständlich auch ein Prozess und sollte in der erwähnten Prozessliste auftauchen. Interessanterweise können Sie ein Skript dazu bringen, zur Laufzeit einen zusätzlichen Prozess zu erzeugen.

Das in diesem Abschnitt beschriebene Verfahren funktioniert nur auf UNIX-Systemen. Wenn Sie Ruby auf einem Windows-Rechner verwenden, stehen Ihnen nur die weiter unten beschriebenen Threads zur Verfügung.

Forking-Grundwissen

Alle UNIX-Systeme erzeugen neue Prozesse durch den Systemaufruf `fork`. Dieser fertigt eine exakte Kopie des laufenden Prozesses an, so dass nach diesem Vorgang zwei Prozesse mit denselben Variablen, geöffneten Dateien und standardmäßig auch mit demselben Code existieren. Letzteres ist in der Regel unerwünscht, zeigt aber am deutlichsten, wie das Verfahren funktioniert. Probieren Sie dazu das folgende kleine Skript aus:

```
puts "Noch bin ich der einzige Prozess."
# Neuen Prozess erzeugen
fork
puts "Was glauben Sie, wie viele Prozesse jetzt da sind?"
```

Die Ausgabe dieses Skripts sieht so aus:

```
Noch bin ich der einzige Prozess.
Was glauben Sie, wie viele Prozesse jetzt da sind?
Was glauben Sie, wie viele Prozesse jetzt da sind?
```

Da die zweite Ausgabe nach dem `fork` ausgeführt wurde, sehen Sie sie zweimal, da alle nachfolgenden Anweisungen nun von zwei Prozessen ausgeführt werden. Die beiden werden als *Parent-* und *Child-Prozess* bezeichnet.

In der Regel soll der Child-Prozess natürlich eine andere Aufgabe erledigen als der Parent-Prozess. Bei einem typischen TCP-Server stellt der Parent beispielsweise den lauschenden Socket zur Verfügung, während für jede Client-Verbindung ein Child erzeugt wird. Dieses klassische Modell heißt *Forking-Server*.

Damit Parent und Child unterschiedlichen Aufgaben nachgehen können, brauchen Sie eine Möglichkeit, sie zu unterscheiden. Der Schlüssel ist der Rückgabewert von `fork`. Im Parent-Prozess enthält er die Prozess-ID (PID) des Childs, eine positive Ganzzahl, und im Child `nil`. Somit ermöglicht ein einfaches `if`/`else` die Unterscheidung zwischen den beiden Prozessen. Hier ein einfaches Beispiel:

```
f = fork
if f
  puts "Hi! Ich bin der Parent, und mein Child heisst #{f}."
else
  puts "Ich bin das Child."
end
```

Etwas ungewöhnlich an dieser Fallentscheidung ist, dass beide Zweige ausgeführt werden – der `if`-Teil eben vom Parent und der `else`-Teil vom Child. Die Ausgabe lautet also beispielsweise so:

```
Ich bin das Child.
Hi! Ich bin der Parent, und mein Child heisst 4778.
```

Es wurde bereits erwähnt, dass Parent und Child nach dem `fork` dieselben Variablen besitzen. Allerdings erhält jeder Prozess seine eigene Kopie davon, was Sie sehen können, wenn Sie den Wert individuell ändern. Zum Beispiel:

```
a = 42
f = fork
if f
  # Parent
  # Wert von a ausgeben
  puts "Im Parent besitzt a den Wert #{a}."
  # a aendern
  a = 23
  puts "Im Parent wurde a nun auf #{a} geaendert."
else
  # Child
  puts "Im Child behaelt a den Wert #{a}."
end
# a in beiden Prozessen ausgeben
puts "Nun hat a den Wert #{a}."
```

Hier die zugehörige Ausgabe. Zur Unterscheidung wurden die Parent-Zeilen fett und die Child-Zeilen kursiv gesetzt:

Im Child behaelt a den Wert 42.
Nun hat a den Wert 42.
Im Parent besitzt a den Wert 42.
Im Parent wurde a nun auf 23 geaendert.
Nun hat a den Wert 23.

Beim Einsatz von fork müssen Sie ein spezielles Problem bedenken: Wenn ein Child – etwa zur Bedienung von Clients in einem Forking-Server – seine Arbeit erledigt hat und Sie exit aufrufen, um auch den Prozess selbst zu beenden, gelingt dies nicht vollständig. Der Child-Prozess bleibt als so genannter *Zombie-Prozess* bestehen. Wenn nur wenige Child-Prozesse existieren, ist das nicht weiter schlimm. Aber stellen Sie sich zum Beispiel einen Webserver vor, der monatelang läuft und dabei Tausende von Clients bedient. Dabei könnten Zombies nach und nach den Speicher verstopfen.

Ein Zombie erfüllt keine besondere Aufgabe mehr, sondern steht nur zum Abruf bereit, damit der Parent seinen Exit-Status auslesen kann. Dies geschieht durch ein Verfahren namens Reaping (»Ernte«). Zuständig ist der Systemaufruf wait4, der in Ruby als Klassenmethode namens Process.wait zur Verfügung steht. In Forking-Anwendungen sollte der Parent diese Anweisung regelmäßig aufrufen.

Hier ein interessantes Zombie-Testprogramm:

```
f = fork
if f
  puts "Ich bin der Parent; mein Child heisst #{f}."
  puts "Hier mein Child in der Prozessliste:"
  ps = `ps aux`
  puts ps.match(/^\w+\s+#{f}.*/)
  puts "Bitte ENTER druecken."
  gets
  Process.wait
  puts "Reaping erledigt. Ist mein Child noch in der Prozessliste?"
  ps = `ps aux`
  puts ps.match(/^\w+\s+#{f}.*/)
else
  puts "Ich bin das Child und bin schon fertig."
  exit
end
```

Was macht dieses Skript? Der Child-Prozess jedenfalls macht nicht viel. Er gibt eine Zeile Text aus und beendet sich danach sofort.

Die Arbeitsschritte des Parents sind interessanter: Er gibt zunächst die PID des Child-Prozesses (*seinen* Rückgabewert von fork) aus. Anschließend sucht er genau diesen Wert in der Prozessliste. Dazu ruft er das bereits erwähnte Shell-Kommando ps aux auf.

Wenn Sie ein Shell-Kommando wie hier in `` `Backticks` `` setzen, führt Ruby es als eigenständigen Prozess[8] aus und liefert seine gesamte Ausgabe zurück.

Als Nächstes wird die Zeile herausgesucht, die mit beliebig vielen Wortzeichen (User, der den Prozess ausführt), beliebig viel Whitespace und der Prozess-ID beginnt. Beachten Sie in der ausgegebenen Zeile den Status Z+ (Zombie) sowie die Bemerkung <defunct> (außer Betrieb).

8 Im Gegensatz zu fork funktioniert dies auch unter Windows.

Nun wird der User aufgefordert, **Enter** zu drücken (damit etwas Zeit vergeht). Danach wird `Process.wait` aufgerufen.

Zur Kontrolle wird am Ende nochmals die Prozessliste ausgelesen und durchsucht: Wenn das Reaping erfolgreich war, dürfte der Child-Prozess nicht mehr gefunden werden. In Abbildung 5-5 ist das jedenfalls so – die zweite Ausgabe lautet `nil`.

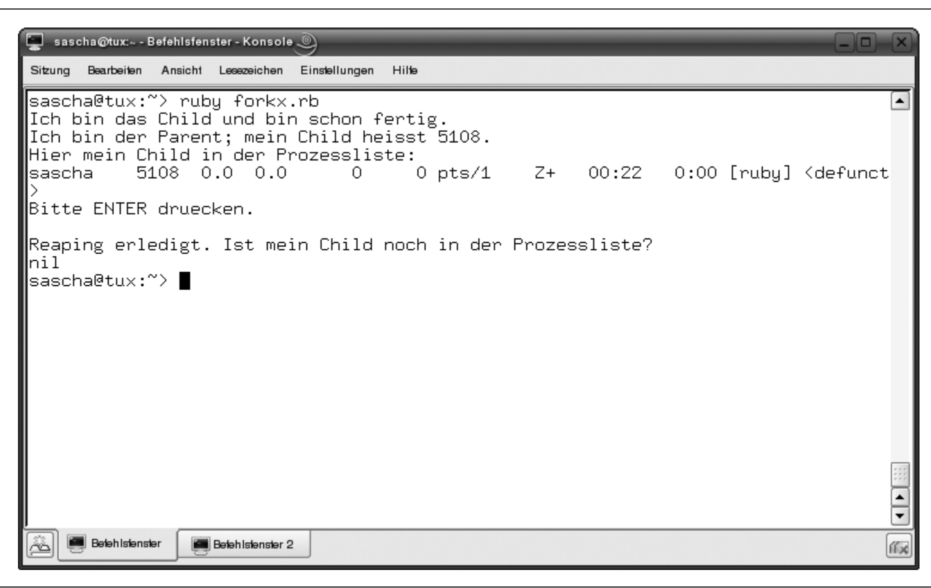

Abbildung 5-5: Ausgabe des Zombie-Testprogramms: Vor dem Reaping befindet sich der Child-Prozess noch in der Prozessliste, danach nicht mehr.

In der Praxis genügt dieses Vorgehen meistens nicht. Das Problem ist nämlich, dass ein `Process.wait`-Aufruf blockiert, bis der erste Child-Prozess beendet wurde. Wenn Sie etwa einen TCP-Server schreiben, der im Parent-Prozess die `accept`-Schleife ausführt und im Child-Prozess die Client-Verbindungen verarbeitet, können Sie diese Anweisung nicht einfach in den Parent-Teil schreiben. Denn dann lässt der Server nur genau einen Client zu, wartet auf das Ende von dessen Verbindung und akzeptiert erst dann den nächsten.

`wait` darf mit anderen Worten nur dann aufgerufen werden, wenn feststeht, dass ein Child beendet wurde. Glücklicherweise wird dieses Ereignis dem Parent automatisch durch ein *Signal* mitgeteilt. Signale sind die einfachste Möglichkeit der Kommunikation zwischen Prozessen (Interprozesskommunikation oder kurz IPC) – umfangreiche Inhalte lassen sich damit nicht austauschen, aber immerhin ein vordefinierter Satz von Konstanten.

Das Signal, mit dem das Ende eines Child-Prozesses bekannt gemacht wird, heißt `SIGCLD` (dies ist die Ruby-Bezeichnung; der POSIX-Name[9] lautet `SIGCHLD`). Um in Ruby ein Signal abzufangen, wird ein Konstrukt mit folgender Syntax verwendet:

```
trap(Signalstring) { ... }
```

Der Signalstring ist ein String, der einen der zulässigen Signalnamen angibt – entweder mit oder ohne `"SIG"`-Präfix. Welche Signale möglich sind, variiert leicht zwischen den UNIX-Varianten. Der Mindestsatz ist im POSIX-Standard festgelegt – eine Liste mit einer kurzen Beschreibung jedes Signals finden Sie beispielsweise unter *http://en.wikipedia.org/wiki/Signal_(computing)*.

Der Block von `trap` ist also die passende Stelle, an der Sie das Reaping ohne die Gefahr einer Blockade durchführen können. Das Schema sieht so aus:

```
trap("SIGCLD") {
  # Reaping, hier ohne Untersuchung des Child-Exit-Status
  Process.wait
}
```

Sie können eine `trap` gleich zu Beginn des gewünschten Prozesses platzieren; aufgerufen wird sie erst später, wenn das betreffende Signal eintrifft.

Um diese Variante zu testen, können Sie das folgende kurze Skript ausprobieren:

```
f = fork
if f
  puts "Ich bin der Parent; mein Child heisst #{f}."
  # Ende des Child-Prozesses abfangen
  trap("SIGCLD") {
    # Reaping
    Process.wait
    # Meldung
    puts "Child-Prozess beendet."
    # Parent beenden
    exit
  }
  # Endlosschleife, damit der Parent weiterlaeuft
  loop do
  end
else
  puts "Ich bin das Child. Druecken Sie ENTER, um mich zu beenden."
  STDIN.gets
  exit
end
```

9 POSIX ist ein recht alter Standard, der den »kleinsten gemeinsamen Nenner« dessen festlegt, was ein UNIX-artiges Betriebssystem können muss. Linux, alle BSDs einschließlich Mac OS X, Sun Solaris und viele andere Systeme genügen diesem Standard; aufgrund neuerer Entwicklungen gehen ihre Gemeinsamkeiten sogar noch weit über POSIX hinaus.

Wie Sie sehen, handelt es sich um eine leicht modifizierte Version des obigen Beispiels. Hier bittet der Child-Prozess darum, Enter zu drücken, und beendet sich erst danach. In diesem Moment sendet er `SIGCLD` an den Parent. Die `trap` fängt dieses Signal ab, ruft Process.wait auf und beendet dann auch den Parent-Prozess.

Wenn Sie ein Signal manuell versenden möchten, können Sie den Systemaufruf `kill` dafür verwenden. Der etwas martialische Name stammt daher, dass das Standardsignal einen Prozess auffordert, sich zu beenden. In Ruby wird `Process.kill` verwendet, um ein Signal zu senden. Die Syntax lautet:

```
Process.kill(Signalstring, PID)
```

Bei Forking-Anwendungen ermittelt der Parent-Prozess die ID eines Child-Prozesses wie beschrieben aus dem Rückgabewert des `fork`-Aufrufs. Ein Child kann dagegen die PID seines Parents herausfinden, indem er `Process.ppid` aufruft.

Wichtig ist, dass Sie für diese Art der Interprozesskommunikation Signale auswählen, die standardmäßig einfach ignoriert werden. Wenn Sie beispielsweise das Signal `SIGTERM` senden, wird der angesprochene Prozess automatisch beendet (was auch nützlich sein kann, wie Sie im nächsten Beispiel erfahren werden). Für benutzerdefinierte Nachrichten sind dagegen die Signale `SIGUSR1` und `SIGUSR2` interessant, auf die ein Prozess normalerweise nicht reagiert.

Das folgende Beispiel stellt im Child-Prozess ein Menü dar, um verschiedene Signale an den Parent zu senden. Dieser reagiert auf `SIGUSR1` und `SIGUSR2` mit verschiedenen Meldungen, während er bei `SIGTERM` automatisch nicht weiter ausgeführt wird:

```
f = fork
if f
  # Parent
  trap("SIGUSR1") {
    puts "Child sendet SIGUSR1"
  }
  trap("SIGUSR2") {
    puts "Child sendet SIGUSR2"
  }
  loop do
  end
else
  # Child
  puts "1. SIGUSR1 senden"
  puts "2. SIGUSR2 senden"
  puts "3. SIGTERM senden"
  loop do
    wahl = STDIN.gets.chomp
    case wahl
    when "1"
      Process.kill("SIGUSR1", Process.ppid)
    when "2"
      Process.kill("SIGUSR2", Process.ppid)
    when "3"
```

```
          Process.kill("SIGTERM", Process.ppid)
          break
        else
          STDOUT.puts "Unbekannter Befehl"
        end
      end
    end
```

Beachten Sie den `break`-Aufruf nach dem Senden von `SIGTERM`. Ein Child-Prozess wird nämlich keineswegs ebenfalls beendet, wenn sein Parent fertig ist. Stattdessen erhält er den Urprozess `init` (PID 1) als »Adoptiv-Parent«.

Forking-Server

Mit Hilfe von `fork` können Sie TCP-Server schreiben, die mehrere dauerhafte Client-Verbindungen zur selben Zeit bedienen können. Hier wird eine entsprechend modifizierte Version des ECHO-Servers präsentiert. Auch der Client muss leicht geändert werden, denn er soll nicht mehr bei jeder Anfrage eine neue Verbindung zum Server herstellen. Stattdessen wird das neue Steuerkommando `QUIT` eingeführt, das den Server anweist, den Child-Prozess *einer* Client-Verbindung zu beenden. Eine weitere Besonderheit dieses neuen Servers ist, dass die Modus-Steuerbefehle nun für jeden Client separat gelten.

In Beispiel 5-5 sehen Sie zunächst den Server; in Beispiel 5-6 folgt der angepasste Client. Wenn Sie dem Kapitel bisher aufmerksam gefolgt sind, müssten Sie die Skripten – mitsamt den kurzen Erläuterungen in den Kommentaren – problemlos verstehen.

Beispiel 5-5: Die Forking-Version des ECHO-Servers, echoforkserver.rb

```
 1   require "socket"

 2   # Modus-Konstanten
 3   NORMAL = 0
 4   REVERSE = 1
 5   ROT13 = 2

 6   # Anfangswert fuer Modus festlegen
 7   mode = NORMAL

 8   # Port von der Kommandozeile lesen oder auf 7 setzen
 9   if(ARGV[0])
10     port = ARGV[0].to_i
11   else
12     port = 7
13   end

14   # Lauschenden Socket erzeugen
15   server = TCPServer.new(port)
16   # Infozeile ausgeben
17   puts "ECHO Server listening on port #{port} ..."
```

Beispiel 5-5: Die Forking-Version des ECHO-Servers, echoforkserver.rb (Fortsetzung)

```
18   # Accept-Schleife
19   loop do
20     next unless client = server.accept
21     # Forking
22     f = fork
23     # Im Child-Prozess Client-Verbindung verarbeiten
24     if f == nil
25       STDOUT.puts "CLIENT-VERBINDUNG HERGESTELLT"
26       loop do
27         # Client-Anfragen auslesen ...
28         anfrage = client.gets.strip
29         # ... und auf die Konsole protokollieren
30         STDOUT.puts "Anfrage: #{anfrage}"
31         # Bei QUIT Client-Verbindung beenden
32         break if anfrage == "QUIT"
33         # Bei EXIT Server beenden und Schluss
34         if anfrage == "EXIT"
35           Process.kill("SIGTERM", Process.ppid)
36           exit
37         end
38         # Modus-Befehle behandeln
39         case anfrage
40         when "MODE_NORMAL"
41           mode = NORMAL
42           client.print "[Modus auf Normal gesetzt]\r\n"
43         when "MODE_REVERSE"
44           mode = REVERSE
45           client.print "[Modus auf Reverse gesetzt]\r\n"
46         when "MODE_ROT13"
47           mode = ROT13
48           client.print "[Modus auf ROT13 gesetzt]\r\n"
49         else
50           # Standardanfrage behandeln
51           case mode
52           when NORMAL
53             client.printf "%s\r\n", anfrage
54           when REVERSE
55             client.printf "%s\r\n", anfrage.reverse
56           when ROT13
57             client.printf "%s\r\n", anfrage.tr("[a-m][A-M][n-z][N-Z]",
                                                   "[n-z][N-Z][a-m][A-M]")
58           end
59         end
60       end
61       # Client-Verbindung schliessen
62       client.close
63       # Child-Prozess beenden
64       exit
65     else
66       # Im Parent trap fuer Ende eines Childs
67       trap("SIGCLD") {
```

Beispiel 5-5: Die Forking-Version des ECHO-Servers, echoforkserver.rb (Fortsetzung)

```
68          # Reaping
69          Process.wait
70          # Meldung
71          STDOUT.puts "CLIENT-VERBINDUNG BEENDET"
72        }
73      end
74    end
75    # Lauschenden Socket schliessen
76    server.close
```

Die einzige kleine Änderung im Server, die noch erwähnt werden sollte, ist die leicht geänderte Struktur der accept-Schleife (Zeile 19-20): Sie wird nun als Endlosschleife ausgeführt, und falls im jeweiligen Durchlauf keine Client-Verbindungsanfrage eingeht, wird per next sofort der nächste Durchlauf eingeleitet. Letztlich ist dies aber Geschmackssache.

Beispiel 5-6: Der passende Client zum Forking-ECHO-Server, echoforkclient.rb

```
 1  require "socket"

 2  # Host und Port von der Kommandozeile lesen
 3  # oder auf localhost und 7 setzen
 4  if(ARGV[0])
 5    host = ARGV[0]
 6  else
 7    host = "localhost"
 8  end
 9  if(ARGV[1])
10    port = ARGV[1].to_i
11  else
12    port = 7
13  end

14  # Konfiguration ausgeben
15  printf "Server: %s, Port: %d\n\n", host, port

16  # Client-Socket mit Host und Portnummer erzeugen
17  conn = TCPSocket.new(host, port)

18  # Zunaechst Endlosschleife
19  loop do
20    print "Text: "
21    # Text von STDIN lesen, Zeilenumbruch entfernen
22    text = STDIN.gets.chomp
23    # Anfrage an den Server senden
24    conn.printf "%s\r\n", text
25    # Bei "EXIT" oder "QUIT" den Client beenden
26    break if text == "EXIT" || text == "QUIT"
27    # Server-Antwort empfangen
28    antwort = conn.gets
29    # Antwort ausgeben
30    puts "Antwort: #{antwort}"
31  end
```

Mit Threads arbeiten

Seit ein paar Jahren unterstützen immer mehr Betriebssysteme, Programmiersprachen und Bibliotheken eine »leichtgewichtige« Alternative zu Prozessen, die so genannten *Threads*. Innerhalb ein und desselben Prozesses können mehrere Threads ausgeführt werden, und auch zwischen ihnen wird automatisch hin- und hergewechselt.

Ruby besitzt eine völlig eigenständige Thread-Implementierung, die nicht von den Fähigkeiten des Betriebssystems abhängt. Damit können Sie nebenläufige Anwendungen schreiben, die auf jedem beliebigen System funktionieren – sowohl auf Windows-Rechnern, auf denen `fork` nicht zur Verfügung steht, als auch auf der UNIX-Plattform, wo sich systemeigene Threads erst nach und nach durchsetzen.

Thread-Grundlagen

Um Code in einem separaten `Thread` auszuführen, müssen Sie eine Instanz der Klasse `Thread` erzeugen. Der Konstruktor benötigt einen Block mit den Anweisungen, die in diesem Thread laufen sollen. Im folgenden Beispiel durchsucht ein separater Thread ein Array mit Zufallszahlen nach einem bestimmten Wert (hier 13), während das Hauptprogramm oder der Haupt-Thread eine Schleife mit 10.001 Durchgängen ausführt und alle durch 1.000 teilbaren Werte ausgibt:

```
# Suchergebnis ausgeben
def suchergebnis(index, wert)
  puts "    #{wert} gefunden an Position #{index}"
end

# Ein Array nach einem bestimmten Wert durchsuchen
def suchen(array, wert)
  array.each_with_index { |a_wert, index|
    if a_wert == wert
      suchergebnis(index, wert)
    end
  }
end

# Array mit 10.000 Zufallszahlen (1-100) erzeugen
arr = (1..10000).collect {
  rand(100) + 1
}

# Thread erzeugen und mit Suche nach 13 beauftragen
suchthread = Thread.new {
  suchen(arr, 13)
}

# Haupt-Thread: in 1000er-Schritten bis 10.000 zaehlen
10001.times { |x|
  if x % 1000 == 0
```

```
    puts "Haupt-Thread ist bei #{x}"
  end
}
```

Leiten Sie die Ausgabe dieses Beispiels, *such.rb*, am besten in eine Datei um, damit Sie einen Eindruck von der Ausführungsreihenfolge der Threads bekommen. Sie werden feststellen, dass sich die Ausgabe der Hauptschleife und der Suchergebnisse blockweise abwechselt; damit Sie dies schneller erkennen können, wurden die Ausgabezeilen des Such-Threads etwas eingerückt.

Der in der Suchen-Funktion verwendete Iterator `each_with_index` wurde bisher noch nicht besprochen. Wie Sie sehen, liefert er für jedes Array-Element zwei Werte in den Block, nämlich den Wert des aktuellen Elements sowie dessen Index.

Der Thread wird mit Hilfe folgender Zeilen erzeugt:

```
suchthread = Thread.new {
  suchen(arr, 13)
}
```

Die Ausführung beginnt unmittelbar nach der Erzeugung des Threads – abwechselnd mit den anderen Threads des Programms, wobei auch das Hauptprogramm einen Thread darstellt. Letzteres hat als Haupt-Thread allerdings eine Besonderheit: Wenn es beendet wird, enden automatisch auch alle anderen Threads. Wenn es erforderlich ist, dass ein Thread seine Arbeit am Programmende noch abschließt, müssen Sie seine Methode `join` aufrufen – am Ende des obigen Programmbeispiels könnten Sie beispielsweise folgende Zeile hinzufügen:

```
suchthread.join
```

Ein Thread lässt sich bei Bedarf auch verlangsamen. Die zuständige Methode heißt `sleep`; sie kann sowohl innerhalb eines Threads (ohne Objektbezug) als auch für eine benannte `Thread`-Instanz von außen aufgerufen werden. Das Argument ist die gewünschte Pause in Sekunden – ein `Float`, damit auch Sekundenbruchteile möglich sind.

Das folgende Beispiel führt im Haupt-Thread einen Countdown von zehn Sekunden durch:

```
10.downto(0) { |n|
  puts n
  sleep(1)
}
```

Im Gegensatz zu Prozessen besitzen Threads keinen separaten Speicherbereich, so dass sie sich alle vor der Thread-Erzeugung definierten Variablen teilen. Der Vorteil ist, dass dabei keine speziellen Kommunikationsverfahren zum Einsatz kommen müssen. Das folgende Beispiel definiert eine Variable namens `a` und erzeugt dann drei Threads. Jeder Thread erhöht `a` um 1 und gibt dann ihren Wert aus:

```
threads = []
a = 10
puts "Anfangswert: #{a}"

3.times {
  threads << Thread.new {
    a += 1
    puts "Wert im Thread: #{a}"
  }
}
```

Die Ausgabe lautet:

```
Anfangswert: 10
Wert im Thread: 11
Wert im Thread: 12
Wert im Thread: 13
```

Der Operator `<<` ist übrigens eine Kurzschreibweise für die `Array`-Methode `push` (siehe Kapitel 2).

Gegenüber Thread-Implementierungen vieler anderer Programmiersprachen haben Ruby-Threads zusätzlich den Vorteil, dass lokal definierte Variablen Ihre Privatangelegenheit sind. Das liegt daran, dass Blöcke einen in sich abgeschlossenen Geltungsbereich bilden.

Es gibt einen speziellen Trick, eine Variable in jeden neu erzeugten Thread hineinzukopieren, aber innerhalb der Threads unabhängig zu behandeln: Sie brauchen die Variable nur als Parameter von `Thread.new` einzusetzen und dann innerhalb des Thread-Blocks als `|Variable|` abzuholen. Hier ein Beispiel, das auf diese Weise die Threads zählt:

```
threads = []

1.upto(4) { |i|
  threads << Thread.new(i) { |n|
    puts "Ich bin Thread Nr. #{n}."
  }
}
```

Was dieses Beispiel ausgibt, können Sie sich wahrscheinlich denken. Wenn nicht, probieren Sie es einfach aus.

Threading-Server

Mit den Socket-Informationen in diesem Kapitel und den Thread-Grundlagen aus dem vorigen Abschnitt dürfte es nicht schwerfallen, eine Thread-Variante des ECHO-Servers zu erstellen. In Beispiel 5-7 wird der vollständige Code dieser letzten Version gezeigt. Anders als der Forking-Server kann diese Version auch unter Windows verwendet werden.

Beispiel 5-7: Die Threading-Version des ECHO-Servers, echothreadserver.rb

```
1   require "socket"

2   # Modus-Konstanten
3   NORMAL = 0
4   REVERSE = 1
5   ROT13 = 2

6   # Port von der Kommandozeile lesen oder auf 7 setzen
7   if(ARGV[0])
8     port = ARGV[0].to_i
9   else
10    port = 7
11  end

12  # Lauschenden Socket erzeugen
13  server = TCPServer.new(port)
14  # Infozeile ausgeben
15  puts "ECHO Server listening on port #{port} ..."

16  # Thread-Array initialisieren
17  threads = []

18  # Accept-Schleife
19  loop do
20    next unless newclient = server.accept
21    # Thread erzeugen und darin Client-Verbindung verarbeiten
22    threads << Thread.new(newclient) { |client|
23      # Modus initialisieren (nun pro Thread lokal!)
24      mode = NORMAL
25      STDOUT.puts "CLIENT-VERBINDUNG HERGESTELLT"
26      loop do
27        # Client-Anfragen auslesen ...
28        anfrage = client.gets.strip
29        # ... und auf die Konsole protokollieren
30        STDOUT.puts "Anfrage: #{anfrage}"
31        # Bei QUIT Client-Verbindung beenden
32        break if anfrage == "QUIT"
33        # Bei EXIT Server beenden und Schluss
34        if anfrage == "EXIT"
35          exit
36        end
37        # Modus-Befehle behandeln
38        case anfrage
39        when "MODE_NORMAL"
40          mode = NORMAL
41          client.print "[Modus auf Normal gesetzt]\r\n"
42        when "MODE_REVERSE"
43          mode = REVERSE
44          client.print "[Modus auf Reverse gesetzt]\r\n"
45        when "MODE_ROT13"
```

Beispiel 5-7: Die Threading-Version des ECHO-Servers, echothreadserver.rb (Fortsetzung)

```
46          mode = ROT13
47          client.print "[Modus auf ROT13 gesetzt]\r\n"
48        else
49          # Standardanfrage behandeln
50          case mode
51          when NORMAL
52            client.printf "%s\r\n", anfrage
53          when REVERSE
54            client.printf "%s\r\n", anfrage.reverse
55          when ROT13
56            client.printf "%s\r\n", anfrage.tr("[a-m][A-M][n-z][N-Z]",
                                                 "[n-z][N-Z][a-m][A-M]")
57          end
58        end
59      end
60      # Client-Verbindung schliessen
61      client.close
62    }
63  end
64  # Lauschenden Socket schliessen
65  server.close
```

In Zeile 22 wird der durch `accept` erzeugte Client-Socket nach dem bereits gezeigten Schema an den zuständigen Thread weitergereicht. Weiterhin müssen Sie noch beachten, dass die Variable `mode` nun lokal pro Client-Thread vereinbart werden muss (Zeile 24). Andernfalls würden sich Modus-Änderungen wieder auf *alle* Client-Verbindungen auswirken – wie im ursprünglichen, nicht nebenläufigen Beispiel.

Wenn Sie diesen neuen Server ausprobieren möchten, können Sie als Client die Version *echoforkclient.rb* aus dem vorigen Abschnitt verwenden. Sie funktioniert auch mit dem Threading-Server genauso unproblematisch.

Sie können sich das Schreiben Thread-basierter Server noch einfacher machen, indem Sie Ihren Server von der Klasse `GServer` ableiten. Dazu müssen Sie zunächst mittels

```
require "gserver"
```

die passende Bibliothek importieren; anschließend können Sie loslegen. Das Grundgerüst eines solchen Servers lautet:

```
class MeinServer < GServer
  def initialize(port)
    super(port)
  end
end
```

Auf der Website zum Buch finden Sie eine `GServer`-Version des ECHO-Servers sowie einen einfachen Webserver.

Zusammenfassung

In diesem Kapitel haben Sie zum ersten Mal die Grenzen eines einzelnen Rechners verlassen und sich ins Netzwerk vorgetastet. Erfreulicherweise funktionieren heutzutage fast alle Netzwerke genauso wie das Internet, so dass jede Netzwerkanwendung zugleich eine Internetanwendung ist und umgekehrt.

Im ersten Abschnitt wurden einige wichtige Grundbegriffe der Netzwerktheorie eingeführt. Diese sollten Sie verstanden und verinnerlicht haben, um erfolgreich Netzwerkanwendungen programmieren zu können. TCP/IP-Netzwerke bestehen aus vier *Schichten* oder Funktionsebenen: Auf die Netzwerkhardware baut die logische Adressierung durch IP-Adressen auf. Diese wird von den Transportprotokollen TCP und UDP verwendet, um Verbindungen zwischen bestimmten Anwendungen bereitzustellen. Auf der obersten Ebene befinden sich schließlich die Anwendungsprotokolle, das heißt die »Sprachen«, in denen Clients und Server miteinander kommunizieren. Die meisten Internet-Anwendungsprotokolle sind klartextbasiert, was Entwicklern und Administratoren die Fehlersuche enorm vereinfacht.

Der zweite Abschnitt führte das grundlegende Handwerkszeug zum Schreiben von Netzwerkanwendungen ein: die Sockets, genauer gesagt die verschiedenen von `BasicSocket` abgeleiteten Klassen. In der Praxis sind die wichtigsten `TCPSocket` für allgemeine TCP-Verbindungen (in der Regel Clients) sowie TCPServer für eigene Serverdienste. Beide wurden anhand eines einfachen Beispiels – einem erweiterten ECHO-Server, der lediglich die Client-Anfrage zurücksendet, und dem zugehörigen Client – erläutert.

Neben den einfachen Sockets stellt Ruby eine Reihe weiterer Bibliotheken zur Verfügung, mit denen Sie bequemer auf verschiedene Anwendungsprotokolle zugreifen können. Als besonders prominenter Vertreter wurde `Net::HTTP` vorgestellt, und zwar zur Implementierung eines richtigen kleinen (zeilenbasierten) Webbrowsers.

Im letzten Abschnitt schließlich drehte sich alles um das Thema Nebenläufigkeit: In der Praxis müssen TCP-Server in der Lage sein, mehrere Client-Verbindungen gleichzeitig zu verarbeiten. Zu diesem Zweck wurde zunächst der (auf UNIX-Systeme beschränkte) Umgang mit Prozessen vorgestellt. Wenn Sie Windows verwenden, stehen Ihnen dagegen nur die im Anschluss erläuterten Threads zur Verfügung. Beide Varianten wurden durch eine Neuimplementierung des ECHO-Servers mit der jeweiligen Technik veranschaulicht.

Die nächsten beiden Kapitel beschäftigen sich weiterhin mit Netzwerkthemen, allerdings auf einer höheren Ebene: Zunächst erfahren Sie, wie Sie in Ruby klassische Webanwendungen (mit einem externen Webserver) schreiben können; anschließend wird das moderne Web-Framework Ruby on Rails vorgestellt.

In diesem Kapitel:

- Den Webserver Apache 2 installieren
- CGI-Skripten mit Ruby
- Zugriff auf Datenbanken

KAPITEL 6
Klassische Webanwendungen

We are in great haste to construct a magnetic telegraph from Maine to Texas; but Maine and Texas, it may be, have nothing important to communicate.[1]
– Henry David Thoreau

Schon bevor das im nächsten Kapitel besprochene Framework Ruby on Rails entwickelt wurde, konnte man mit Ruby Webanwendungen schreiben. Bei einer Webanwendung liefert ein Webserver keine statischen Dokumente aus, sondern startet nach dem Empfang einer HTTP-Anfrage ein Programm oder Skript. Seine Eingabe erhält dieses Programm aus Benutzereingaben in ein Webformular. Die Ausgabe des Programms – meist HTML-Code, manchmal auch ein dynamisch erzeugtes Bild oder eine andere Datei – wird an den Browser des Benutzers zurückgeliefert.

Viele Webanwendungen lesen zusätzlich Daten aus einer Datenbank und schreiben wieder neue Daten hinein, damit sie dauerhaft gespeichert bleiben. Das ist die übliche Art und Weise, größere Websites zu betreiben, weil sie effizient und wenig störanfällig ist. Die Zusammenarbeit mit einer Datenbank wird in diesem Kapitel anhand des beliebten Open Source-Datenbankservers MySQL erläutert.

1 Übersetzung: Wir beeilen uns, eine Magnettelegrafenleitung von Maine nach Texas zu bauen, aber Maine und Texas haben sich vielleicht gar nichts Wichtiges mitzuteilen.

Den Webserver Apache 2 installieren

Bevor Sie Webanwendungen schreiben können, benötigen Sie einen Webserver. Der bekannteste und verbreitetste von allen ist der *Apache HTTP Server*. Da es sich dabei um frei verfügbare Open Source Software handelt, können Sie diesen Industriestandard auch als Entwicklungsserver für Ihre eigenen Websites einsetzen. Nach einigen Worten zu den Fähigkeiten dieses Servers wird die Installation unter Windows und UNIX beschrieben; danach erhalten Sie noch einige Informationen zur Apache-Grundkonfiguration.

Eine schnelle und einfache Alternative für einen reinen Entwicklerrechner ist die Installation von XAMPP. Es handelt sich dabei um ein integriertes und weitgehend fertig konfiguriertes Paket mit Apache, MySQL-Datenbank, PHP und weiteren Serverkomponenten. Unter den diversen Erweiterungspaketen ist inzwischen auch `mod_ruby` (siehe den Kasten am Ende dieses Kapitels) zu finden. Downloads für Windows, Linux und Mac OS X sowie ausführliche Installationsanleitungen finden Sie unter *http://www.apachefriends.org/de/xampp.html*.

Eigenschaften von Apache 2

Apache wurde seit 1995 entwickelt, zunächst als Weiterführung des universitären *NCSA HTTPd* und danach als eigenständiger, wesentlich leistungsfähigerer Webserver. Die *Apache Software Foundation*, in der sich seine Entwickler organisiert haben, betreut neben diesem Server längst eine große Anzahl verschiedener renommierter Open Source-Projekte, beispielsweise den Java-Webanwendungs-Server *Tomcat*, den Spam-Filter *SpamAssassin* oder unzählige Java- und XML-Tools.

Die aktuelle Apache-Version ist 2.2; die vorherigen Versionen 2.0 und 1.3 sind aber noch immer weitverbreitet, und es gibt noch regelmäßige Bugfixes dafür. Wenn Sie eine Neuinstallation von Apache planen, sollten Sie allerdings auf jeden Fall die neueste Version wählen.

Sie können die passende Apache-Distribution – den Binär-Installer für Windows oder das Quellcode-Paket für UNIX – unter *http://httpd.apache.org* herunterladen.

Apache ist modular aufgebaut. Viele Funktionen sind also kein unverzichtbarer Bestandteil des Webservers, sondern können je nach Bedarf hinzugefügt oder entfernt werden. Das inzwischen bevorzugt eingesetzte Verfahren seit Version 2.0 besteht sogar darin, die gewünschten Module nicht ein für alle Mal in Apache einzukompilieren, sondern sie als DSO-Dateien (*Dynamic Shared Objects*) zu laden. Somit müssen Sie nur noch einen bestimmten Eintrag zur Webserver-Konfigurationsdatei hinzufügen und Apache neu starten, um das entsprechende Modul einzubinden.

Neben den zahlreichen mitgelieferten Modulen gibt es auch noch Unmengen von Drittanbieter-Modulen. Viele von ihnen sind auf der Site *http://modules.apache.org* registriert. Für Ruby-Programmierer ist das weiter unten angesprochene Modul `mod_ruby` interessant, das den Ruby-Interpreter direkt in den Webserver einbettet. In Tabelle 6-1 finden Sie einen Überblick über einige der wichtigsten Module.

Tabelle 6-1: Die wichtigsten Apache-Module im Überblick

Modul	Bedeutung	mitgeliefert	aktiviert
`mod_alias`	Um- und Weiterleitung	ja	ja
`mod_auth_basic`	Klartextbasierte Benutzeranmeldung (Authentifizierung)	ja	ja
`mod_auth_digest`	Verschlüsselte Authentifizierung	ja	nein
`mod_authn_file`	Anmeldedaten aus Textdateien	ja	ja
`mod_authn_dbm`	Anmeldedaten aus datenbankähnlichen DBM-Dateien	ja	nein
`mod_authz_host`[a]	Host-basierte Zugriffskontrolle (Order/Allow/Deny, siehe unten)	ja	ja
`mod_autoindex`	Automatisch erstellter Verzeichnisindex	ja	ja
`mod_cgi`	CGI-Skripten (klassische Webserver-Anwendungen)	ja	ja
`mod_dir`	Definition der Indexseite	ja	ja
`mod_log_config`	Protokollierung (Logdateien)	ja	ja
`mod_mime`	Zuweisungen von Dateityp, Zeichensatz und Sprache	ja	ja
`mod_negotiation`	Content-Negotiation (MIME-Varianten je nach Client-Vorgabe)	ja	ja
`mod_perl`	In Apache integrierter Perl-Interpreter	nein	–
`mod_ruby`	In Apache integrierter Ruby-Interpreter (siehe unten)	nein	–
`mod_ssl`	Verschlüsselte Verbindungen (https)	ja[b]	nein

a Bis Apache 2.0 lautete der Name dieses Moduls `mod_access`.

b In der offiziellen Binärversion für Windows wird `mod_ssl` nicht mitgeliefert. Lesen Sie *http://buecher.lingoworld.de/apache2/mod_ssl.html* für Bezugsquellen und eine Installationsanleitung.

Eine weitere interessante Neuerung von Apache 2 gegenüber der alten Version 1.3 ist die Einführung der so genannten *Multi-Processing-Module* (MPM): Wie Sie aus dem vorigen Kapitel wissen, gibt es verschiedene Möglichkeiten, nebenläufige Server zu schreiben. Durch MPM können Sie die jeweils passende Variante für Ihr System und Ihre Bedürfnisse auswählen. Die beiden wichtigsten Beispiele sind:

- `prefork` – die klassische, rein prozessbasierte UNIX-Lösung. *Pre*fork bedeutet, dass neue Prozesse nicht erst beim Eintreffen von Client-Anfragen, sondern bereits vorher auf Vorrat erzeugt werden.
- `worker` – ein Thread-basiertes Modell. Wenn Ihr System über eine moderne Thread-Implementierung verfügt (zum Beispiel Linux ab Kernel 2.4), arbeitet diese Variante speicherschonender als `prefork`.

Daneben gibt es einige gemischte Prozess/Thread-Modelle. Für Windows und einige andere Nicht-UNIX-Plattformen werden jeweils eigene, angepasste MPM verwendet.

Die Einführung in diesem Abschnitt genügt, um mit Apache eigene Webanwendungen zu entwickeln. Wenn Sie eine Website auf einem eigenen Server im Internet veröffentlichen möchten, sind zusätzliche Überlegungen erforderlich, vor allem im Hinblick auf die Sicherheit. Wenn Sie weitere Informationen über Apache benötigen, empfehle ich Ihnen mein Buch *Apache 2*, 2. Auflage, Bonn 2006, Galileo Press. Auch auf der Website zu diesem Buch unter *http://buecher.lingo-world.de/apache2/* finden Sie zahlreiche Konfigurationstipps.

Installation unter Windows

Für Windows bietet die Apache Software Foundation einen bequemen Binär-Installer im MSI-Format (Windows-Installer) an. Starten Sie das Paket *apache_2.2.4-win32-x86-no_ssl.msi* per Doppelklick. Folgen Sie dann Schritt für Schritt den Installationsanweisungen auf den folgenden Dialogseiten:

1. Information, dass Apache 2.2 installiert wird. Klicken Sie hier – wie auch auf allen folgenden Bildschirmen – *Next* an, um fortzufahren.
2. Anzeige der Apache-Lizenz. Es handelt sich um eine GPL-ähnliche Open Source-Lizenz; im Wesentlichen erlaubt sie den freien Einsatz und die Veränderung von Apache und schützt die Entwickler vor der Vereinnahmung durch kommerzielle Unternehmen oder gar Softwarepatente. Wählen Sie »I accept the terms in the license agreement«.
3. Kurze Übersicht über die Features von Apache 2.2 und weitere Informationsquellen.
4. Eingabe einiger Grundeinstellungen: *Network Domain* ist Ihr Domainname; für einen lokalen Testserver können Sie zum Beispiel *test.local* eingeben. *Server Name* ist der Name des Webservers, zum Beispiel *www.test.local*. Die *Administrator's E-Mail Address* wird angegeben, damit Benutzer Ihnen bei Fehlern eine entsprechende Mitteilung senden können; normalerweise wird dafür die Adresse *webmaster@<Ihre-Domain>* verwendet. Als Letztes müssen Sie auswählen, ob Apache als automatisch startender Produktions-Server (*For All Users, on Port 80, as a Service*) oder als manuell zu startendes Testprogramm (*only for the Current User, on Port 8080, when started Manually*) installiert wer-

den soll. Selbst für lokale Testzwecke lohnt sich Ersteres; der Dienst verbraucht kaum Ressourcen und kann leicht gegen Zugriffe von außen geschützt werden.

5. Nun müssen Sie die Vollständigkeit der Installation auswählen: *Typical* installiert die wichtigsten Komponenten; mit *Custom* können Sie weitere Elemente auswählen. Sie sollten auf jeden Fall *Custom* einstellen, da Sie nur so den Installationsort bestimmen können.
6. Diese Dialogseite wird nur angezeigt, wenn Sie im vorigen Schritt *Custom* gewählt haben. Oben sehen Sie in einer Baumansicht die Bestandteile der Installation. Wenn genügend Festplattenspeicher zur Verfügung steht (gut 50 MByte), sollten Sie einfach den obersten Eintrag anklicken und *This Feature, and all subfeatures, will be installed on local hard drive* wählen; andernfalls können Sie beispielsweise die Dokumentation weglassen. Unten können Sie das Installationsverzeichnis wählen – voreingestellt ist *Apache Software Foundation* im *Programme*-Verzeichnis.
7. Klicken Sie auf dem letzten Bildschirm auf *Install*, um die Installation mit den gewählten Einstellungen zu starten, oder auf *Back*, falls Sie doch noch etwas ändern möchten.

Nach der Installation als Dienst wird Apache automatisch gestartet. Im Systray (rechter Rand der Taskleiste) erscheint das kleine Feder-Icon des *Apache Service Monitors*. Klicken Sie es mit der rechten Maustaste an, um den Monitor zu öffnen. Dort können Sie Apache steuern – ihn zum Beispiel nach einer Konfigurationsänderung neu starten.

Installation unter UNIX

In vielen Linux- und BSD-Distributionen ist der Apache-Webserver bereits ab Werk enthalten oder kann bei der Installation des Systems optional ausgewählt werden. Auch Mac OS X liefert ihn mit. Falls dies auf Ihrem System der Fall ist, können Sie diesen Abschnitt überspringen – Sie sollten lediglich sicherstellen, dass mindestens Apache 2.0 installiert ist und keine 1.3-Version. Zu Ihrer eigenen Sicherheit sollten Sie außerdem überprüfen, ob die installierte Version aktuell ist – ältere Releases enthalten oft bekannte Sicherheitslücken, die Cracker sich zunutze machen.

Auf UNIX-Systemen wird Apache fast immer aus dem Quellcode kompiliert; selbst die Apache Software Foundation bietet für diese Plattform keine Binärpakete an. Zum Kompilieren benötigen Sie einen modernen ANSI-C-Compiler, vorzugsweise die *GNU Compiler Collection* (GCC).

Der erste Installationsschritt besteht darin, dass Sie das Quellcode-Paket durch folgende Anweisung entpacken:

```
# tar -xzvf httpd-2.2.4.tar.gz
```

Wechseln Sie anschließend in das neu entstandene Verzeichnis, in das die Archivdateien entpackt wurden:

```
# cd httpd-2.2.4
```

Nun muss das Skript `configure` im aktuellen Verzeichnis aufgerufen werden, um die Quelldateien vor der eigentlichen Kompilierung an Ihre Bedürfnisse anzupassen. Das Skript kennt unzählige Optionen; geben Sie Folgendes ein, wenn Sie sie alle studieren möchten:

```
# ./configure --help |less
```

Hier nur die allerwichtigsten Optionen im Überblick:

- `--prefix=/Verzeichnispfad` gibt das übergeordnete Verzeichnis für die Apache-Installation an; standardmäßig */usr/local/apache2*.
- `--enable-layout=Layoutname` ermöglicht alternativ (oder zusätzlich) zur Angabe eines Verzeichnisses die Auswahl eines Installationslayouts, das die Verzeichnisse für die verschiedenen Komponenten festlegt. Die Datei *config.layout* enthält die Definitionen der verschiedenen Layouts. Sie können sie mit einem Texteditor öffnen, um das für Sie passende Layout zu finden (oder nach dem Schema in der Datei selbst zu erstellen).
- `--with-mpm=MPM-Modul` wählt eines der oben angesprochenen Multi-Processing-Module (MPM) aus. Unter UNIX wird standardmäßig das MPM `prefork` installiert. Wenn Sie keine exotischen Drittanbieter-Module verwenden, kann sich auch die Einstellung `--with-mpm=worker` lohnen – sie installiert das Thread-basierte Worker-MPM.
- `--enable-so` ist eine der wichtigsten Optionen: Sie schaltet die Unterstützung für Dynamic Shared Objects (DSOs) ein, so dass Module dynamisch hinzugefügt werden können. Da Apache fast alle seine Aufgaben durch einzelne Module löst, ist dies praktisch unabdingbar. Inzwischen ist diese Option glücklicherweise Standard.
- `--enable-modules="modul1 modul2 ..."` kompiliert die angegebenen Module fest ein. In der Liste müssen Sie das Kürzel `mod_` vor dem Modulnamen weglassen – also zum Beispiel `"autoindex log_config"` statt `"mod_autoindex mod_log_config"`. Für manche häufig genutzten Module steht übrigens auch die Kurzfassung `enable-modulname` zur Verfügung.
- `--enable-mods-shared="modul1 modul2 ..."` kompiliert die angegebenen Module als DSOs, was den praktischen Vorteil hat, dass sie jederzeit (auch vorübergehend) wieder entfernt werden können. Für prominente Module gibt es auch die Schreibweise `enable-modulname=shared`.
- `--enable-modules=most` beziehungsweise `enable-mods-shared=most` kompiliert *fast alle* Module. Je nach Auswahl werden sie statisch einkompiliert oder als DSOs erstellt. `most` ist in den meisten Fällen gegenüber `all` vorzuziehen, weil unter anderem alle Module weggelassen werden, die beim Kompilieren Probleme bereiten würden.

- `--disable-modules="modul1 modul2 ..."` lässt die angegebenen Module ausdrücklich weg. Das ist besonders beim Einsatz von `enable-modules=all` sehr nützlich.

Das folgende Beispiel bereitet Apache für die Kompilierung und Installation im Standardverzeichnis */usr/local/apache2* und zur Installation der meisten Module als DSOs vor:

```
# ./configure --enable-layout=Apache --enable-so \
--enable-mods-shared=most
```

Nachdem `configure` seine Arbeit beendet hat, die viele Minuten dauern kann, werden die beiden Befehle zur Kompilierung und Installation aufgerufen; spätestens für Letzteres benötigen Sie `root`-Rechte:

```
# make
# make install
```

Nach der Installation können Sie Apache starten. Dafür ist das Skript `apachectl` im *bin*-Verzeichnis Ihrer Installation (Standard: */usr/local/apache2/bin*) zuständig:

```
# apachectl start
```

Dieses Skript dient auch dazu, Apache nach einer Konfigurationsänderung neu zu starten. Geben Sie dazu Folgendes ein:

```
# apachectl restart
```

Bei Servern im Praxiseinsatz sollten Sie laufende Client-Verbindungen nicht durch einen Neustart abbrechen, sondern zunächst abwarten, bis sie abgeschlossen sind. Das erledigt die Option `graceful` (»elegant«):

```
# apachectl graceful
```

Sie können den Webserver mit Hilfe dieses Tools auch beenden, indem Sie den Befehl `stop` beziehungsweise `graceful-stop` verwenden.

Zu guter Letzt kann `apachectl` dazu genutzt werden, um Apache 2 beim Booten automatisch zu starten. Dazu müssen Sie in Ihrem Startskripten-Verzeichnis einen Link darauf erzeugen und dieses Startskript aktivieren. Bei den meisten Linux-Systemen funktionieren diese beiden Schritte beispielsweise so:

```
# ln -s /usr/local/apache2/bin/apachectl /etc/init.d/apache2
# chkconfig -a apache2
```

Basiskonfiguration

Nun ist Apache installiert, und Sie können sich um die Basiskonfiguration kümmern, bevor Sie ihn starten. Bei einer Apache-Standardinstallation befinden sich die wichtigsten Einstellungen in einer einzigen Konfigurationsdatei namens *httpd.conf*, die sich im *conf*-Verzeichnis des Servers befindet; das Unterverzeichnis *extra* enthält einige optionale Konfigurationsdateien für diverse Zusatzmodule. In welchem

Verzeichnis sich die Konfigurationsdateien unter UNIX bei Ihrem Installationslayout befinden, können Sie dem Eintrag *sysconfdir* in der Datei *config.layout* aus dem Quellcode-Paket entnehmen. Bei der Installation in */usr/local/apache2* mit dem Standardlayout *Apache* handelt es sich um das Verzeichnis */usr/local/apache2/conf*. Unter Windows befindet sich die Datei im Verzeichnis *<Apache-Basisverzeichnis>/conf*, bei einer Standardinstallation also unter *C:/Programme/Apache Software Foundation/Apache2.2/conf*.

Die Konfigurationsdatei besteht aus zahlreichen Konfigurationseinstellungen, den *Direktiven*. Die meisten von ihnen werden in der Datei selbst durch Kommentare beschrieben – ein Kommentar ist jede Zeile, die mit # anfängt. Eine deutschsprachige Referenz aller Direktiven finden Sie im offiziellen Apache Manual (*http://httpd.apache.org/docs-2.2/de/*), das normalerweise mit Apache auf Ihrem Rechner installiert wird.

Im Folgenden werden nur einige der wichtigsten Direktiven im Überblick vorgestellt, und zwar in der Reihenfolge, in der sie in der Standard-Konfigurationsdatei einer Apache-Neuinstallation auftreten. Beachten Sie, dass es noch einige weitere wichtige Einstellungen gibt; diese betreffen allerdings die Sicherheit und Stabilität des Servers, so dass Sie die Voreinstellungen beibehalten sollten, falls Sie nicht ganz genau wissen, was Sie tun.

`ServerRoot`

Diese Direktive gibt das Apache-Installationsverzeichnis an; in der Regel wurde es beim Installationsprozess bereits richtig eingestellt. Zum Beispiel:

```
ServerRoot /usr/local/apache2
```

`Listen`

Diese Direktive stellt den TCP-Port ein, an dem Apache lauscht. Wie im vorigen Kapitel beschrieben, hat das HTTP-Protokoll den Standardport 80; in der Regel steht hier also Folgendes:

```
Listen 80
```

Falls Apache noch an anderen Ports lauschen soll (etwa 443 für gesicherte SSL-Verbindungen), werden weitere `Listen`-Direktiven benötigt.

`LoadModule`

Zum dynamischen Laden gewünschter DSO-Module müssen Sie `LoadModule` verwenden. Die Syntax lautet:

```
LoadModule xxx_module modules/mod_xxx.so
```

Das erste Argument ist die allgemeine Modulbezeichnung, das zweite der Pfad der Moduldatei. Letzterer kann – wie hier – relativ zur `ServerRoot` oder auch absolut sein. Das folgende Beispiel lädt das Modul `mod_autoindex` zur automatischen Erzeugung von Verzeichnis-Indizes:

```
LoadModule autoindex_module modules/mod_autoindex.so
```

`ServerAdmin`

Hier sollte bei öffentlichen Webservern die E-Mail-Adresse des Administrators (meist *webmaster@<Ihre-Domain>*) angegeben werden. Wenn der Apache automatisch Seiten erzeugt – vor allem für Fehlermeldungen wie »Seite nicht gefunden« –, kann er einen Link auf diese Adresse anzeigen, damit Benutzer entsprechende Fehlermitteilungen versenden können. Zum Beispiel:

```
ServerAdmin webmaster@test.local
```

`ServerName`

Mit Hilfe von `ServerName` wird der Netzwerkname des Webservers angegeben. Zum Beispiel:

```
ServerName www.test.local
```

Beachten Sie, dass diese Einstellung nur der Selbstidentifikation des Servers dient. Da mit er tatsächlich unter diesem Namen im Netzwerk erreichbar ist, müssen Sie einen entsprechenden Eintrag auf einem Nameserver vornehmen. Für den Hausgebrauch genügt es auch, eine Zeile wie die folgende in die Datei */etc/hosts* (Windows: *<Windows-Verzeichnis>/System32/drivers/etc/hosts*) aller beteiligten Rechner zu schreiben:

```
192.168.0.2 www.test.local
```

Statt 192.168.0.2 müssen Sie natürlich die richtige IP-Adresse Ihres Webserver-Rechners angeben.

Wenn Sie Webserver und Browser auf demselben Rechner ausführen, ist der Server stets auch unter der speziellen IP-Adresse 127.0.0.1 (die *Loopback-Adresse* für »Selbstgespräche«) erreichbar; meist ist ihr auch der besondere Hostname *localhost* zugeordnet. Geben Sie also im Browser *http://127.0.0.1* oder *http://localhost* ein.

`DocumentRoot`

Hier wird das Stammverzeichnis der Website angegeben. Bei einer UNIX-Standardinstallation sieht der Eintrag beispielsweise so aus:

```
DocumentRoot /usr/local/apache2/htdocs
```

Eine Client-Anfrage mit der URL *http://www.test.local/products/info.html* würde in diesem Fall auf die Datei */usr/local/apache2/htdocs/products/info.html* verweisen. Hier zum Vergleich ein Windows-Beispiel:

```
DocumentRoot "C:/Programme/Apache Software Foundation/Apache2.2/htdocs"
```

Beachten Sie in diesem Zusammenhang die Anführungszeichen – sie sind bei Apache-Direktiven zwingend erforderlich, sobald ein Parameter Leerzeichen enthält.

`<Directory> ... </Directory>`

Dies ist eine von mehreren so genannten *Container-Direktiven*: Die darin enthaltenen Einstellungen beziehen sich nur auf das angegebene Verzeichnis und seine Unterverzeichnisse. Das wichtigste Beispiel sind die weiter unten gezeigten Voreinstellungen für alle Verzeichnisse und für die `DocumentRoot`; Letzteres sieht bei einer UNIX-Standardinstallation schematisch so aus:

```
<Directory /usr/local/apache2/htdocs>
   # Voreinstellungen fuer die DocumentRoot ...
</Directory>
```

`<Location> ... </Location>`

Ähnelt `<Directory>` sehr, mit dem Unterschied, dass sich die Angabe auf einen URL-Pfad bezieht. Betrachten Sie dazu das folgende Beispiel:

```
<Location /info>
   # Einstellungen fuer URL-Pfad /test
</Location>
```

Die Direktiven in diesem Container betreffen also (auf das obige Beispiel bezogen) alle URLs, die mit *http://www.test.local/info/...* beginnen.

`Options`

`Options` stellt verschiedene Einstellungen für das Verhalten eines Verzeichnisses zur Verfügung. Beispiele: `Indexes` generiert automatisch eine Liste mit Hyperlinks auf die Dateien im Verzeichnis, wenn keine Indexseite (`DirectoryIndex`, siehe unten) gefunden wird; `Includes` aktiviert die Ausführung von SSI (Server Side Includes); `FollowSymLinks` löst symbolische Links (Verweise auf Dateien in anderen Verzeichnissen) auf. Zum Beispiel:

```
Options Indexes FollowSymLinks
```

`AllowOverride`

Innerhalb der Website selbst können zusätzliche Dateien mit Konfigurationsdirektiven stehen, die nur das jeweilige Verzeichnis betreffen. Der Standardname einer solchen Datei lautet *.htaccess*; dies kann auf Wunsch mit Hilfe der (hier nicht näher beschriebenen) Direktive `AccessFileName` geändert werden. Die Direktive `AllowOverride` legt fest, ob – und welche – Direktiven in *.htaccess*-Dateien innerhalb der Verzeichnisse der Website selbst überschrieben werden dürfen. Neben `All` (jede grundsätzlich verzeichnisfähige Direktive) und `None` (keine) können einige Gruppen angegeben werden, zum Beispiel `FileInfo` (Datei-Einstellungen) oder `AuthConfig` (Authentifizierung). Zum Beispiel:

```
AllowOverride FileInfo
```

Wenn Sie eine einzelne Website betreiben, die nur von Ihnen administriert wird, sollten Sie *.htaccess*-Dateien mittels `AllowOverride None` abschalten und sämtliche Konfigurationseinstellungen in die *httpd.conf* schreiben. Diese Dateien sind nur nützlich, wenn Sie die Verantwortung für einzelne Verzeichnisse an Dritte delegieren möchten.

`Order`, `Allow`, `Deny`

Diese drei Direktiven bestimmen, welche Hosts oder Netzwerke Zugriff auf ein bestimmtes Verzeichnis erhalten sollen. Bei `Allow` können Sie Hosts oder Adressbereiche auflisten, denen der Zugriff explizit gewährt werden soll; `Deny` verbietet ihn grundsätzlich. `Order` kann die Werte `Allow,Deny` oder `Deny,Allow` (jeweils ohne Leerzeichen!) annehmen und bestimmt so, in welcher Reihenfolge `Allow`- und `Deny`-Angaben ausgewertet werden. In aller Regel wird eine der beiden Direktiven für alle Hosts gesetzt (`Deny from all` beziehungsweise `Allow from all`); die jeweils andere definiert dann Ausnahmen. Hier ein Beispiel:

```
# URL-Pfad /lokal nur fuer das lokale Netz
<Location /lokal>
   Order Deny,Allow
   Deny from all
   Allow from 192.168.0
</Location>
```

Natürlich müssen Sie 192.168.0 durch den Netzwerkteil der IP-Adressen Ihres eigenen Netzes ersetzen. Beachten Sie im Übrigen, dass diese Direktiven normalerweise nichts bringen, wenn Sie bestimmte externe Benutzer ausschließen wollen, da die meisten IP-Adressen dynamisch von den Internet-Providern zugewiesen werden.

In den Verzeichniseinstellungen der `DocumentRoot` befinden sich bei öffentlichen Webservern folgende Einträge, die allen Hosts den Zugriff gewähren:

```
Order Allow,Deny
Allow from all
```

In einer reinen Testumgebung sollten Sie die Zugriffe allerdings auf den Rechner selbst (127.0.0.1) beziehungsweise auf das lokale Netzwerk beschränken.

`DirectoryIndex`

Diese Direktive bestimmt, welche Dateien als Indexdokumente (oder Startseiten) gelten sollen. Wenn ein Benutzer ein Verzeichnis, aber keine bestimmte Datei anfordert, sucht Apache automatisch nach Dateien mit den hier angegebenen Namen; die erste gefundene Datei wird ausgeliefert. Die Voreinstellung ist *index.html*; auf einem Ruby-fähigen Webserver empfiehlt sich dagegen:

```
DirectoryIndex index.html index.rb index.rbx
```

Was passiert, wenn im angesprochenen Verzeichnis gar keine Indexseite gefunden wird, hängt von weiteren Einstellungen ab: Sofern die Option `Indexes` (siehe oben) gesetzt und das Modul `mod_autoindex` geladen ist, generiert Apache selbst einen Verzeichnisindex. Andernfalls wird die Statusmeldung 404 (Seite nicht gefunden) an den Client gesendet.

`Alias`

Die Direktive `Alias` ordnet einem Verzeichnis außerhalb der `DocumentRoot` einen URL-Pfad zu. Auf diese Weise können Sie über den Webserver auch Dateien

veröffentlichen, die aus organisatorischen Gründen oder aus Sicherheitserwägungen nicht im eigentlichen Website-Verzeichnis liegen. Betrachten Sie dazu folgendes Beispiel:

```
Alias /test/ /usr/local/testverzeichnis/
```

Das Verzeichnis `testverzeichnis`, das in diesem Beispiel in Wirklichkeit im Verzeichnis `/usr/local/` liegt, wird unter dem URL-Pfad `/test/` eingebunden. Es kann also unter der URL *http://www.test.local/test/* aufgerufen werden.

HTTP-Grundwissen

Webserver wie Apache kommunizieren über das bereits im vorigen Kapitel erwähnte Anwendungsprotokoll *HTTP* (HyperText Transfer Protocol) mit Client-Programmen – die meisten von ihnen sind Webbrowser wie Firefox oder der Internet Explorer. Die genaue Definition der aktuellen HTTP-Version steht in RFC 2616.

Üblicherweise sendet der Client eine Anfrage an den Server, sobald Sie eine URL in die Adresszeile Ihres Browsers eintippen oder einen Link anklicken. Eine HTTP-Anfrage sieht beispielsweise folgendermaßen aus:

```
GET /seiten/info.html HTTP/1.1
Accept: */*
Accept-Language: de, en-US
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.8.1.1) Gecko/20061204
Firefox/2.0.0.1
Host: www.test.local
Connection: Keep-Alive
```

Die erste Zeile ist der eigentliche Befehl. Er besteht aus der *HTTP-Methode* – hier `GET` –, dem Pfad der angeforderten Ressource (*info.html* im Website-Unterverzeichnis */seiten*) sowie der HTTP-Protokollversion (HTTP/1.1 ist der Standard). Die Methode `GET` fordert eine Ressource vom Server an. Eine andere bekannte HTTP-Methode ist zum Beispiel `POST`; sie kann zusätzlich größere Datenmengen – meist aus Webformularen – an den Webserver senden.

Alle anderen Zeilen sind HTTP-Header, die dem Server weitere Aspekte über den Client und die Anfrage mitteilen:

- `Accept` gibt eine Liste von MIME-Typen der Dokumentarten an, die der Client akzeptiert. MIME-Typen haben das Format Haupttyp/Untertyp – etwa `text/html` für HTML-Dokumente oder `image/jpeg` für JPEG-Bilder. `*/*` bedeutet, dass der Client alle Arten von Dokumenten annimmt (was noch lange nicht heißt, dass er sie auch darstellen kann).
- `Accept-Language` listet die ISO-Kürzel der bevorzugten Sprachen des Clients auf. Webserver wie Apache beherrschen eine Technik namens *Content-Negoti-*

ation, die die Accept*-Header auswertet und Dokumente in der bevorzugten Sprache, dem bevorzugten Dateityp oder dem gewünschten Zeichensatz an Clients ausliefern kann. Im Beispiel werden Deutsch und US-Englisch (in dieser Reihenfolge) gewünscht.

- Accept-Encoding gibt an, dass der Browser komprimierte Ressourcen verarbeiten kann, und listet die einzelnen Kompressionsformate auf. Der Browser im Beispiel versteht GNU-Zip (gzip) und ZIP (deflate).
- User-Agent ist die Selbstidentifikation des Clients – hier handelt es sich um den Browser Firefox 2.0 unter Windows XP.
- Host ist der wichtigste Anfrage-Header; in HTTP/1.1 ist er vorgeschrieben. Auf einem Serverrechner können mehrere Websites mit eigenen Domainnamen liegen, die als *virtuelle Hosts* bezeichnet werden. Damit der Server weiß, welche Site angefordert wird, benötigt er diesen Header.
- Connection gibt an, ob die Verbindung zwischen Client und Server bestehen bleiben soll (Keep-Alive) oder geschlossen wird (close). Das Offenhalten der Verbindung ermöglicht das schnellere Nachladen verknüpfter Dateien – wie beispielsweise Bilder, die in eine Webseite eingebettet wurden.

Auf eine solche Anfrage antwortet der Webserver mit einer HTTP-Antwort (Response). Diese besteht aus einer Statuszeile, mehreren Antwort-Headern, einer Leerzeile und schließlich dem Body, der die eigentliche, vom Client angeforderte Ressource liefert. Hier ein Beispiel für die HTTP-Server-Antwort auf eine GET-Anfrage:

```
HTTP/1.1 200 OK
Date: Wed, 27 Dec 2005 19:37:42 GMT
Server: Apache/2.2.3 (Unix)
Last-Modified: Thu, 10 Feb 2005 07:56:38 GMT
ETag: "39a4cb-5a3-6ef34d10"
Accept-Ranges: bytes
Content-Length: 2092
Connection: Keep-Alive
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//Transitional//EN">
<html>
<head>
[...]
```

Die Statuszeile gibt Auskunft über die HTTP-Version (hier 1.1) sowie über Erfolg oder Misserfolg der Anfrage. Der Status 200 OK bedeutet, dass die Anfrage mit der Lieferung der gewünschten Ressource beantwortet werden kann. Andere wichtige Statusmeldungen sind etwa 404 Not Found (Ressource nicht gefunden; meist bei fehlerhaften URL-Eingaben oder veralteten Links), 500 Internal Server Error (könnte Ihnen bei Fehlern in der Webprogrammierung begegnen) oder die diversen 3xx-Weiterleitungen.

Auch bei der Antwort sind die restlichen Zeilen Header-Felder; die hier verwendeten bedeuten Folgendes:

- `Date` gibt Datum und Uhrzeit der Lieferung an.
- `Server` ist die Selbstidentifikation der Webserver-Software; bei Apache wird ihre Ausführlichkeit durch die weiter unten beschriebene Direktive `ServerTokens` gesteuert.
- `Last-Modified` enthält Datum und Uhrzeit der letzten Änderung. Browser- und Proxy-Caches nutzen diese Angabe, um zu entscheiden, ob die zwischengespeicherte Version noch gültig ist. Für solche Prüfungen existiert übrigens die spezielle HTTP-Anfragemethode `HEAD`, die nur die Header, aber nicht die eigentliche Ressource anfordert.
- `ETag` (Entity-Tag) ist ein aus verschiedenen Angaben berechneter Hashwert, der die Identität eines Dokuments über die URL oder den Dateinamen hinaus überprüft. Auch dies ist für Caching-Zwecke wichtig.
- `Accept-Ranges: bytes` gibt an, dass der Server Anfragen nach Teilbereichen einer Ressource akzeptiert. Das macht sich beispielsweise der Adobe (Acrobat) Reader zunutze, um umfangreiche PDF-Dokumente stückweise anzufordern.
- `Content-Length` ist die Größe des Bodys (also der gelieferten Datei) in Byte.
- `Connection` entspricht dem gleichnamigen Anfrage-Header. Der Server gibt an, ob er das Offenhalten der Verbindung bestätigt (`Keep-Alive`) oder ablehnt (`close`).
- `Content-Type` ist der wichtigste Antwort-Header. Er gibt den Datentyp (MIME-Typ) der gelieferten Ressource an, damit der Client weiß, wie er die empfangenen Daten behandeln soll.

Nach einer Leerzeile folgt der Body, also der eigentliche Inhalt der Antwort. Im Beispiel wurde er auf die ersten drei Zeilen gekürzt.

CGI-Skripten mit Ruby

Der klassische Ansatz für Webanwendungen ist die CGI-Schnittstelle (Common Gateway Interface). Der Webserver startet kurz gesagt einen neuen Prozess und darin die Webanwendung als externes Programm – im Fall einer interpretierten Skriptsprache wie Ruby wird zuerst der Interpreter selbst gestartet. Damit ist CGI problemlos für kleine bis mittlere Websites geeignet; wenn Sie eine extrem gut besuchte Site betreiben, sollten Sie dagegen eine andere Lösung wählen.

Apache ermöglicht den Zugriff auf CGI-Anwendungen über das Modul `mod_cgi`. Standardmäßig können Sie Ihre CGI-Skripten im Verzeichnis *cgi-bin* unter Ihrer `ServerRoot` (dem Apache-Installationsverzeichnis) speichern. Dieses Verzeichnis wird automatisch mit Hilfe einer `ScriptAlias`-Direktive als URL-Pfad eingebunden, unter Windows beispielsweise so:

```
ScriptAlias /cgi-bin/ \
   "C:/Programme/Apache Software Foundation/Apache2.2/cgi-bin/"
```

`ScriptAlias` funktioniert im Prinzip wie die im ersten Abschnitt besprochene Direktive `Alias` – mit dem Unterschied, dass *alle* Dateien im eingebundenen Verzeichnis als CGI-Skripten betrachtet werden.

Alternativ können Sie folgende Sektion in Ihre *httpd.conf*-Datei einfügen (am besten unter den `<Directory>...</Directory>`-Block der `DocumentRoot`), um *alle* Dateien mit der Endung *.rb* serverweit als CGI-Skripten zu betrachten:

```
<Files *.rb>
  Options +ExecCGI
  SetHandler cgi-script
</Files>
```

Ein `<Files>`-Container wird auf alle Dateien im aktuellen Kontext angewendet, die dem angegebenen Dateimuster entsprechen. `Options +ExecCGI` fügt die CGI-Ausführung zu den bestehenden Optionen hinzu. Die `AddHandler`-Direktive sorgt schließlich dafür, dass die betreffenden Dateien als CGI-Skripten betrachtet werden.

Wenn Sie diese (oder irgendeine andere) Konfigurationsänderung durchführen möchten, müssen Sie daran denken, Apache anschließend neu zu starten.

Die ersten CGI-Beispiele

Ein Ruby-CGI-Skript ist im Prinzip ein gewöhnliches Ruby-Skript, dessen Ausgabe als Webseite an den Browser eines Users ausgeliefert wird. Dazu müssen solche Skripten allerdings zwei Bedingungen erfüllen:

- Sie müssen dem Rechner mitteilen, dass es sich um ein Ruby-Skript handelt. Dafür wird die in Kapitel 2 erwähnte Shebang-Zeile als erste Zeile des Skripts verwendet. Unter Windows sieht sie beispielsweise so aus:

  ```
  #!C:/ruby/bin/ruby.exe -w
  ```

  Die entsprechende UNIX-Variante lautet:

  ```
  #!/usr/bin/ruby -w
  ```

  Den Pfad müssen Sie jeweils an Ihr eigenes System anpassen.
- Damit der Browser weiß, um welche Art von Daten es sich bei der Ausgabe Ihres CGI-Skripts handelt, müssen Sie ihm zuerst einen `Content-Type`-Header senden. In den meisten Fällen erzeugen CGIs HTML-Code, so dass die betreffende Zeile so lautet:

  ```
  puts "Content-type: text/html"
  puts
  ```

  Die Leerzeile nach dem Header signalisiert dem Browser, dass nun der Body folgt.

Mit diesem Vorwissen können Sie leicht Ihr erstes CGI-Skript schreiben. Hier der vollständige Code:

```
#!/usr/bin/ruby -w

puts "Content-type: text/html; charset=iso-8859-1"
puts
puts "<html>"
puts "<head>"
puts "<title>Ruby-CGI-Skript</title>"
puts "</head>"
puts "<body>"
puts "<h1>HALLO, WELT!</h1>"
puts "Ein CGI-Skript, erzeugt #{Time.new}"
puts "</body>"
puts "</html>"
```

Vergessen Sie nicht, gegebenenfalls die Shebang-Zeile anzupassen. Speichern Sie das Skript danach je nach Konfiguration im Verzeichnis *cgi-bin* oder irgendwo in Ihrem Website-Verzeichnis *htdocs*. Öffnen Sie dann einen beliebigen Webbrowser und geben Sie die passende URL (zum Beispiel *localhost/cgi-bin/hello.rb*) ein. Abbildung 6-1 zeigt das Ergebnis in dem kleinen Textbrowser aus dem vorigen Kapitel.

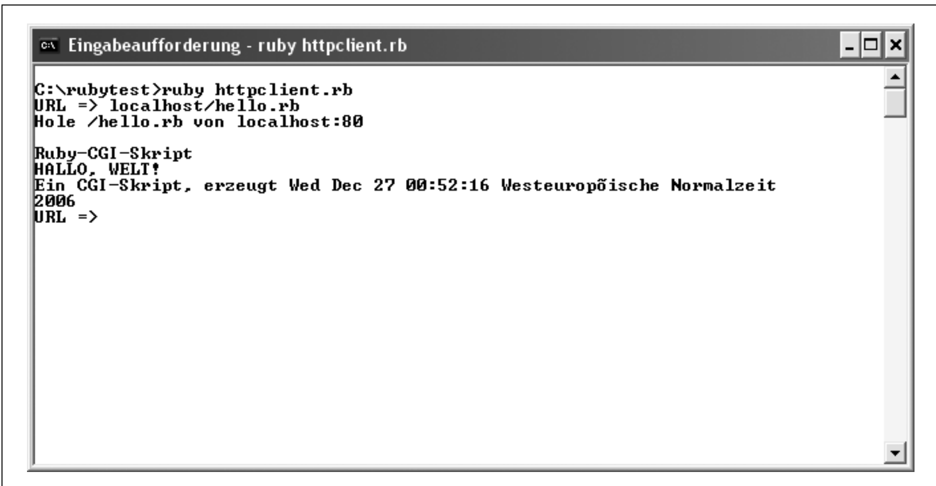

Abbildung 6-1: Ausgabe des ersten Ruby-CGI-Skripts im Textbrowser aus Kapitel 5

Das nächste Beispiel ist bereits viel nützlicher: Es gibt eine HTML-Tabelle mit den Namen und Werten aller Umgebungsvariablen aus. *Umgebungsvariablen* sind Voreinstellungen im Ausführungskontext eines Programms oder Skripts. Eine der bekanntesten Konsolen-Umgebungsvariablen ist `PATH`; sie legt die Verzeichnisse fest, in denen bei Eingabe eines Kommandos nach ausführbaren Programmen gesucht wird. Um ihren Inhalt zu lesen, können Sie auf der Windows-Konsole Folgendes eingeben:

```
> echo %PATH%
```

Auf UNIX-Systemen lautet die Syntax dagegen:

```
$ echo $PATH
```

Ruby stellt die Umgebungsvariablen in einem Hash namens `ENV` zur Verfügung; die Variablennamen sind die Schlüssel und ihre Werte die Inhalte. Eine einfache Iteration zur Ausgabe aller Umgebungsvariablen lautet also:

```
ENV.each_pair { |var, val|
  puts "#{var} = #{val}"
}
```

Beachten Sie, dass sich der CGI-Ausführungskontext stark von der Konsole unterscheidet. Deshalb gibt es dort auch völlig andere Umgebungsvariablen. Hier das CGI-Skript, mit dem Sie sie lesen können:

```
#!C:/Ruby/bin/ruby.exe -w

puts <<"ENDE_KOPF"
Content-type: text/html

<html>
<head>
<title>CGI-Umgebungsvariablen</title>
</head>
<body>
<h1>CGI-Umgebungsvariablen mit Ruby</h1>
Dieses Skript wurde mit Hilfe von Ruby erzeugt. Hier eine Tabelle mit allen
Umgebungsvariablen:
<table border="2" cellpadding="8">
<tr>
<th>Umgebungsvariable</th>
<th>Wert
</tr>
ENDE_KOPF

ENV.each_pair { |var, val|
  puts "<tr>"
  puts "<td valign=\"top\"><tt>#{var}</tt></td>"
  print "<td valign=\"top\">"
  print val == "" ? " " : val
  puts "</td>"
  puts "</tr>"
}

puts <<"ENDE_FUSS"
</table>
</body>
</html>
ENDE_FUSS
```

Wie Sie sehen, bieten sich HIER-Dokumente für die Ausgabe längerer HTML-Passagen an. Der eigentliche Funktionskern des Skripts steht innerhalb des Iterators. Name und Wert jeder Variablen werden in Tabellenzellen geschrieben. Die Zeile

```
print val == "" ? " " : val
```

sorgt zusätzlich dafür, dass leere Werte durch ein geschütztes Leerzeichen (` `) ersetzt werden. Eine ganz leere Tabellenzelle würde nämlich ohne Ränder angezeigt, was recht hässlich aussehen würde.

In Abbildung 6-2 sehen Sie den Beginn der Ausgabe dieses Skripts im Browser Mozilla Firefox.

Abbildung 6-2: Ausgabe der CGI-Umgebungsvariablen per CGI-Skript

Hintergrundwissen über Formulardaten

Wenn Sie unbedingt möchten, können Sie sogar Webformulare »zu Fuß« einlesen. Bei `GET`-Anfragen stehen die Daten in der Umgebungsvariablen `QUERY_STRING` zur Verfügung, auf die Sie mittels `ENV['QUERY_STRING']` zugreifen können. Bei `POST`-Anfragen werden sie dagegen über die Standardeingabe bereitgestellt und können mit den üblichen Eingabeoperationen eingelesen werden. Welche HTTP-Methode verwendet wurde, steht im Übrigen in `ENV['REQUEST_METHOD']`.

Das Format der Daten ist für die manuelle Verarbeitung allerdings recht unpraktisch. Angenommen, eine Webseite enthält folgendes Formular mit zwei Textfeldern namens `ort` und `telefon`:

```
<form action="auswertung.rb" method="GET">
  Ort: <input type="text" name="ort" /><br />
  Telefon: <input type="text" name="telefon" /><br />
  <input type="submit" value="Abschicken" />
</form>
```

Sie füllen die beiden Felder mit *Köln* beziehungsweise *0221/123456* aus und klicken auf den mit *Abschicken* beschrifteten Absende-Button an. Die Daten werden mit der Methode `GET` an das Ruby-CGI-Skript *auswertung.rb* versendet. `GET` hängt sie – durch ein Fragezeichen getrennt – als Query-String an die Anfrage-URL. Diese lautet daher: `auswertung.rb?ort=K%F6ln&telefon=0221%2F123456`.

Wenn das Ruby-Skript `ENV['QUERY_STRING']` ausliest, erhält es:

```
ort=K%F6ln&telefon=0221%2F123456
```

An dieser Stelle müssten die einzelnen `Name=Wert`-Paare an den `&`-Zeichen getrennt werden; anschließend könnten Sie diese im erhaltenen Array nochmals am Gleichheitszeichen in Feldname und Wert aufteilen. Zuletzt ist eine Umwandlung der speziell codierten Sonderzeichen wie `%F6` für `ö` oder `%2F` für `/` erforderlich. Zu diesem Zweck enthält die Klasse `CGI` aus der Ruby-Standardbibliothek die Klassenmethode `unescape`. Der Code zum Zerlegen der genannten Felder sähe also wie folgt aus:

```
require "cgi"
formdata = ENV['QUERY_STRING']
formvars = formdata.split("&")
formvars.each { |feld|
  (name, wert) = feld.split("=")
  printf "%s = %s<br />\n", name, CGI.unescape(wert)
end
```

Wenn Sie dies ausprobieren möchten, können Sie es leicht auf eigene Faust tun. Allerdings ist diese Vorgehensweise in der Praxis weder erforderlich noch empfehlenswert. Denn sobald Sie die Bibliothek *cgi.rb* importieren, können Sie CGI-Parameter viel leichter auslesen – und noch viele andere Webaufgaben erheblich eleganter erledigen.

CGI-Fehler finden

CGI-Skripten haben einen großen Nachteil gegenüber Ruby-Skripten, die auf der Konsole ausgeführt werden: Wenn ein Fehler auftritt, erhalten Sie keine detaillierte Fehlermeldung an Ort und Stelle, das heißt in diesem Fall im Browser. Dieser gibt bei schwerwiegenden Fehlern (besonders, wenn keine Header vorhanden sind) eine allgemeine Fehlerseite mit dem Status *500 Internal Server Error* zurück. Bei geringfügigen Fehlern wird sogar einfach nur eine leere Seite angezeigt.

Ruby gibt die Fehlermeldungen wie gehabt aus – allerdings nicht auf `STDOUT` wie die eigentliche Webseite, sondern auf `STDERR`. Apache wiederum schreibt alle `STDERR`-

Ausgaben in seine `ErrorLog`-Datei. Wo sich diese befindet, steht in der gleichnamigen Apache-Konfigurationsdirektive – in der Regel handelt es sich um die Datei *error.log* oder *error_log* im Unterverzeichnis *logs* der Apache-Konfiguration. Ein typischer Windows-Pfad wäre also beispielsweise *C:\Programme\Apache Software Foundation\Apache2.2\logs\error.log*; unter UNIX könnte er dagegen etwa */usr/local/apache2/logs/error_log* lauten.

Um sich einen solchen Logeintrag anzusehen, können Sie zum Beispiel folgendes CGI-Skript mit einem kleinen Fehler im Browser testen:

```
#!/usr/bin/ruby -w
puts "Content-type: text/html"
puts
pits "<html></html>"
```

Öffnen Sie Ihre `ErrorLog`-Datei zum Lesen und betrachten Sie die letzte Zeile. Sie dürfte ungefähr wie folgt lauten:

```
[Thu Dec 28 19:14:35 2006] [error] [client 127.0.0.1]
C:/Programme/Apache Software Foundation/Apache2.2/htdocs/fehler.rb:4:
undefined method `pits' for main:Object (NoMethodError)\r
```

Einen schwerwiegenden Fehler mit Status 500 erhalten Sie beispielsweise durch dieses Skript:

```
#!/usr/bin/ruby -w
puts "<html><head><title>Test</title></head>"
puts "<body>Test</body></html>"
```

Wie Sie sehen, fehlen in diesem Skript die Header. Die `ErrorLog`-Datei meldet dies wie folgt:

```
[Thu Dec 28 19:20:44 2006] [error] [client 127.0.0.1] malformed
header from script. Bad header=<html><head><title>Test</title:
fatal.rb
```

Die Ruby-CGI-Bibliothek verwenden

Die weiter oben erwähnte Bibliothek *cgi.rb* vereinfacht das Schreiben von CGI-Skripten in Ruby. Ihre wichtigsten Fähigkeiten sind:

- Einfaches Lesen von Formulardaten
- Erzeugung von HTML-Code durch Ruby-Methoden
- Setzen und Auslesen von Cookies
- Session-Tracking

Bevor diese Themen in den nachfolgenden Abschnitten vertieft werden, hier zunächst ein einfaches Beispiel. Es handelt sich um einen Service für Reisende in die englischsprachige Welt, denn er ermöglicht die Eingabe einer Temperatur und die Auswahl, ob von Celsius in Fahrenheit umgerechnet werden soll oder umgekehrt.

In Beispiel 6-1 sehen Sie zunächst den Quellcode. Diese Datei, *temperatur.rb*, muss wieder je nach Konfiguration in das *cgi-bin*-Verzeichnis oder in ein beliebiges Verzeichnis unterhalb Ihrer Apache-`DocumentRoot` kopiert werden. Auch die Shebang-Zeile müssen Sie gegebenenfalls wieder anpassen.

Beispiel 6-1: Der CGI-Temperaturumrechner, temperatur.rb

```
 1  #!C:/ruby/bin/ruby.exe -w
 2  require "cgi"
 3  # CGI-Instanz mit HTML-Erzeugungsmethoden
 4  cgi = CGI.new("html4")
 5  # HTTP-Header ausgeben
 6  cgi.header
 7  # HTTP-Body ausgeben
 8  cgi.out {
 9    # HTML-Dokument
10    cgi.html {
11      # Kopf des HTML-Dokuments
12      cgi.head {
13        cgi.title { "Ruby-Temperaturumrechner" }
14      } +
15      # Body des HTML-Dokuments
16      cgi.body {
17        # Hauptueberschrift
18        cgi.h1 { "Temperaturumrechner" } +
19        if cgi.has_key?('temp') && cgi.has_key?('rechnung')
20          # Wenn Formulardaten vorhanden sind, umrechnen
21          temp = cgi['temp'].to_i
22          rechnung = cgi['rechnung']
23          if rechnung == "cf"
24            # Celsius in Fahrenheit
25            ergebnis = temp * 1.8 + 32
26            ausgabe = sprintf("%.2f", ergebnis)
27            "#{temp}&deg;C = #{ausgabe}&deg;F"
28          else
29            # Fahrenheit in Celsius
30            ergebnis = (temp - 32) / 1.8
31            ausgabe = sprintf("%.2f", ergebnis)
32            "#{temp}&deg;F = #{ausgabe}&deg;C"
33          end
34        else
35          # Nichts ausgeben, falls keine Formulardaten
36          ""
37        end +
38        # Formular: Methode GET, Ziel aktuelles Skript (Standard)
39        cgi.form("get") {
40          # Tabelle
41          cgi.table('border' => "2", 'cellpadding' => "8") {
42            cgi.tr {
43              cgi.td { "Temperaturwert" } +
44              cgi.td {
45                # Textfeld zur Eingabe des Temperaturwerts
```

Beispiel 6-1: Der CGI-Temperaturumrechner, temperatur.rb (Fortsetzung)

```
46                cgi.text_field("temp")
47              }
48            } +
49            cgi.tr {
50              cgi.td { "Umrechnung" } +
51              cgi.td {
52                # Radio-Buttons fuer Umrechnungsrichtung
53                cgi.radio_group("rechnung",
54                                ["cf", "&deg;C -> &deg;F  ", true],
55                                ["fc", "&deg;F -> &deg;C"])
56              }
57            } +
58            cgi.tr {
59              cgi.td('colspan' => "2") {
60                # Absende-Button
61                cgi.submit("Umrechnen")
62              }
63            }
64          }
65        }
66      }
67    }
68  }
```

Öffnen Sie einen Webbrowser und geben Sie die URL des Skripts ein – zum Beispiel *http://localhost/temperatur.rb*, falls es sich direkt im äußersten Website-Verzeichnis befindet. Geben Sie einen Temperaturwert ein und wählen Sie die gewünschte Umrechnungsrichtung. Klicken Sie dann auf *Umrechnen*. Abbildung 6-3 zeigt das Ergebnis im Firefox, nachdem 30°C in Fahrenheit umgerechnet wurde.

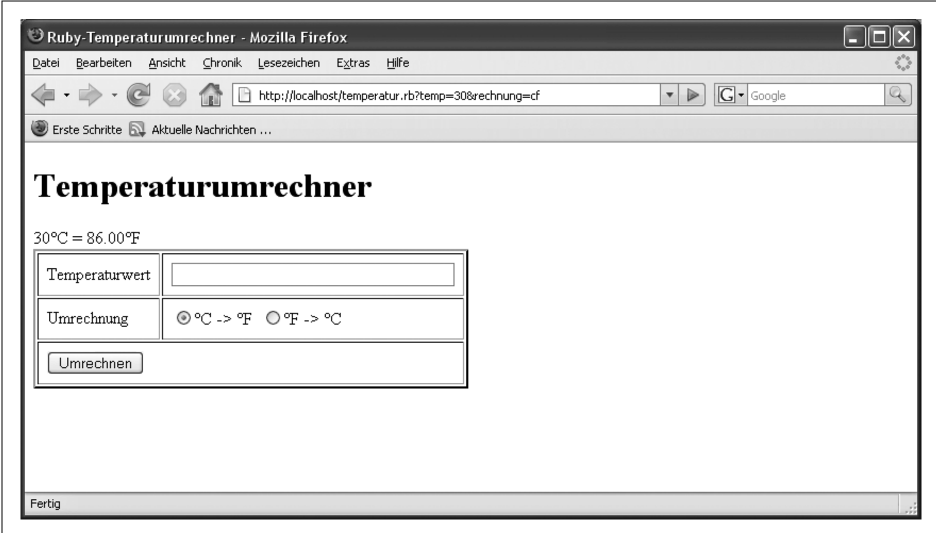

Abbildung 6-3: Der CGI-Temperaturumrechner im Einsatz

Die verschiedenen Teile dieses Skripts erledigen folgende Aufgaben:

- Zeile 2: Import der CGI-Bibliothek.
- Zeile 4: Eine neue `CGI`-Instanz wird erzeugt. Das Argument `"html4"` sorgt dafür, dass die HTML-Erzeugungsmethoden aus dem Mixin-Modul `CGI::HtmlExtension` eingefügt werden, und zwar für HTML 4.0.
- Zeile 6: `cgi.header` erzeugt den HTTP-Header – wenn Sie keine Argumente angeben, wird der minimal erforderliche Header erzeugt:

  ```
  Content-type: text/html
  ```

- Zeile 8-68: `cgi.out` nimmt einen Block entgegen, dessen Inhalt als Body der HTTP-Antwort dient – standardmäßig also als HTML-Dokument.
- Zeile 12-14: `cgi.head` erzeugt den `<head>`-Block des HTML-Dokuments. Das Pluszeichen in Zeile 14 muss gesetzt werden, weil die Ausgabe von `cgi.head`, genau wie der gesamte in `cgi.out` verschachtelte Inhalt, ein String ist. Diese String-Werte werden auf dieser sowie auf allen untergeordneten Ebenen mittels `+` verkettet.
- Zeile 13: `cgi.title` generiert das `<title>`-Tag; der Block-Inhalt ist der enthaltene Text.
- Zeile 16-66: `cgi.body` umschließt den `<body>`-Block des Dokuments.
- Zeile 19-37: Der `if-else`-Abschnitt prüft, ob Werte für die beiden Formularfelder `temp` und `rechnung` vorliegen. Wenn Sie Formulare über die Ruby-Klasse `CGI` erzeugen, werden diese standardmäßig wieder an das aktuelle Skript selbst versandt, so dass diese Daten vorhanden sind, wenn das Formular im vorigen Durchlauf abgeschickt wurde. Zum Testen wird die Methode `has_key?` des `CGI`-Objekts verwendet.
- Zeile 21: Das Formularfeld `temp` wird ausgelesen und in eine Ganzzahl umgewandelt. Zum Lesen von CGI-Parametern können Sie den jeweiligen Feldnamen einfach als Index auf die `CGI`-Instanz anwenden.
- Zeile 22: Auch der Inhalt des Feldes `rechnung` wird ermittelt. Es handelt sich um zwei Radio-Buttons, von denen nur einer ausgewählt werden kann. Je nachdem, welchen Sie anklicken, kommt entweder `"cf"` (Umrechnung von Celsius nach Fahrenheit) oder `"fc"` dabei heraus.
- Zeile 23-27: Im Modus `"cf"` wird der eingegebene Temperaturwert hier von °C in °F umgewandelt. In Zeile 25 findet die eigentliche Umrechnung statt. In Zeile 26 wird der Wert auf zwei Nachkommastellen gekürzt, und Zeile 27 übernimmt die Ausgabe. Beachten Sie, dass das Sonderzeichen `°` nicht websicher ist und daher wie hier als Entity-Referenz `°` geschrieben werden sollte.
- Zeile 28-32: In diesem `else`-Block findet entsprechend die Umrechnung von Fahrenheit in Celsius statt.

- Zeile 34-36: Wenn keine Formulardaten vorliegen, wird ein leerer String als Ausgabe erzeugt. Dieser `else`-Teil ist absolut notwendig, weil die `if-else`-Sequenz auf jeden Fall einen String zurückliefern muss. Andernfalls funktioniert die String-Verkettung mit + nicht mehr.
- Zeile 39-65: Dieser Bereich des Skripts erzeugt den `<form>...</form>`-Block. Der einzige Formularparameter ist hier die Methode, `"get"`. Da das zweite Argument – die Action-URL – weggelassen wurde, setzt die CGI-Bibliothek automatisch das aktuelle Skript selbst ein. Der erzeugte HTML-Code sieht so aus:

  ```
  <FORM METHOD="get" ENCTYPE="application/x-www-form-urlencoded"
  ACTION="/temperatur.rb"> ... </FORM>
  ```

  Beim Klicken auf den Absende-Button wird also dieselbe Datei vom Server angefordert, erhält aber zusätzlich Formulardaten. Die Verarbeitung dieser Daten findet in Zeile 19-33 statt und wurde bereits beschrieben.
- Zeile 41-64: Damit Formulare vernünftig formatiert werden, sollten Sie ihre Inhalte wie hier in einer Tabelle unterbringen. Die zahlreichen zulässigen Attribute des `<table>`-Tags können Sie als Hash in die Argumentklammern der Methode `cgi.table` aufnehmen. Aus

  ```
  cgi.table('border' => "2", 'cellpadding' => "8") { ... }
  ```

  wird so beispielsweise:

  ```
  <TABLE border="2" cellpadding="8"> ... </TABLE>
  ```

  `border="2"` sorgt dabei für einen sichtbaren Rahmen und Zellränder mit einer Stärke von zwei Pixeln, während `cellpadding="8"` einen Zellinnenabstand von 8 Pixeln festlegt, damit die Inhalte nicht zu eng an den Rändern kleben.
- Zeile 46: Ein einfaches Textfeld innerhalb eines Formulars können Sie mittels `cgi.text_field` erzeugen. Der einzige notwendige Parameter ist der Feldname, hier `"temp"`. Das HTML-Ergebnis ist:

  ```
  <INPUT NAME="temp" SIZE="40" TYPE="text">
  ```

- Zeile 53-55: Hier wird mit Hilfe der Methode `cgi.radio_group` eine Gruppe von *Radio-Buttons* erzeugt. Der Name dieser Steuerelemente stammt daher, dass Sie immer nur eines aus der Gruppe auswählen können – genau wie bei den Frequenzbandwählern eines alten Radios. Das erste Argument ist ein String mit dem gemeinsamen Gruppennamen, danach folgt für jeden Button ein Array mit zu übertragendem Formularwert und Beschriftung. Einem Ihrer Buttons können Sie als drittes Argument den Wert true mitgeben – dieser ist dann automatisch vorausgewählt. Aus dem vorliegenden Methodenaufruf entsteht so folgender HTML-Code:

  ```
  <INPUT CHECKED NAME="rechnung" TYPE="radio" VALUE="cf">
  &deg;C -> &deg;F  
  <INPUT NAME="rechnung" TYPE="radio" VALUE="fc">
  &deg;F -> &deg;
  ```

- Zeile 62: Zuletzt benötigt das Formular noch eine Schaltfläche zum Abschicken. Als Argument können Sie ihm die gewünschte Beschriftung übergeben; andernfalls legt der jeweilige Browser einen Standardtext fest.

Formulardaten auslesen

Wie Sie im *cgi.rb*-Einführungsbeispiel bereits gesehen haben, gibt es eine sehr einfache Möglichkeit, mit Hilfe der CGI-Bibliothek Formulardaten zu ermitteln: Sie können den jeweiligen Feldnamen als Index auf die `CGI`-Instanz selbst anwenden, um das gewünschte Feld auszulesen. Das Ergebnis ist ein String mit dem fraglichen Wert. Zuvor kann die Methode `cgi.has_key?('Feldname')` eingesetzt werden, um zu prüfen, ob der Benutzer das entsprechende Feld überhaupt ausgefüllt hat.

Das folgende Beispiel prüft, ob ein Feld namens `kundennr` vorhanden ist, und speichert seinen Wert in der gleichnamigen Variablen. Falls das Feld nicht existiert, erhält die Variable dagegen den Wert `"NEUKUNDE"`:

```
if cgi.has_key?('kundennr')
  kundennr = cgi['kundennr']
else
  kundennr = "NEUKUNDE"
end
```

In den meisten Fällen funktioniert dieser einfache Zugriff auf Formularfelder problemlos. Es gibt allerdings zwei Arten von Steuerelementen, die mehrere Werte zurückliefern können: *Checkboxen* und *multiple Auswahlmenüs*. Bei ihnen liefert dieses Zugriffsverfahren jeweils nur den ersten Wert als String. Wenn Sie alle angegebenen Werte auslesen möchten, müssen Sie die Methode `cgi.param` aufrufen. Auch sie liefert einen Hash, in dem die Feldnamen die Schlüssel bilden, aber im Unterschied zum Direktzugriff auf die `CGI`-Instanz ist der jeweilige Wert ein Array mit einem oder mehreren Strings.

Betrachten Sie als Beispiel die folgende Gruppe von Checkboxen:

```
Welche Programmiersprachen beherrschen Sie?<br />
<input type="checkbox" name="sprachen" value="ruby" /> Ruby<br />
<input type="checkbox" name="sprachen" value="perl" /> Perl<br />
<input type="checkbox" name="sprachen" value="php" /> PHP<br />
<input type="checkbox" name="sprachen" value="c" /> C<br />
<input type="checkbox" name="sprachen" value="java" /> Java
```

Angenommen, ein User füllt das Formular mit dieser Checkbox-Gruppe aus und kreuzt Ruby, PHP und Java an. Der einfache Zugriff über

```
cgi['sprachen']
```

liefert in diesem Fall nur den String `"ruby"`. Um alle Sprachen auszulesen, müssen Sie dagegen über

```
cgi.param['sprachen']
```

zugreifen. Dann erhalten Sie das vollständige Ergebnis `["ruby", "php", "java"]`.

Dieselbe Auswahl könnten Sie übrigens auch als Menü schreiben, in dem der User mehrere Werte mit gedrückter **Strg**-Taste an- und wieder abwählen kann. Der entsprechende HTML-Code sähe so aus:

```
<select name="sprachen" size="5" multiple="multiple">
<option value="ruby">Ruby</option>
<option value="perl">Perl</option>
<option value="php">PHP</option>
<option value="c">C</option>
<option value="java">Java</option>
```

HTML-Code erzeugen

Es ist nicht sonderlich elegant, in ein CGI-Skript endlose Ketten von Ausgabekommandos wie

```
puts "<h1>Haupt&uuml;berschrift</h1>"
```

einzufügen. Eine Möglichkeit, längere HTML-Sequenzen auszugeben, sind HIER-Dokumente. Zum Beispiel:

```
# Logo anzeigen, Tagline, Hauptueberschrift
puts <<"ENDE_KOPF"
<img src="logo.gif" /><br/>
<i>Wir bauen Ihre Site</i>
<h1>Willkommen bei der Sitebuilder GmbH & Co. KG</h1>
ENDE_KOPF
```

Noch viel praktischer ist es meistens, die HTML-Erzeugungsmethoden der CGI-Bibliothek einzusetzen. Dazu benötigt der Konstruktoraufruf für die CGI-Instanz die gewünschte HTML-Version als Stringargument. Zulässig sind unter anderem:

- `"html3"` – HTML 3.2, passend für veraltete 3.x-Browser
- `"html4"` – HTML 4.0 Strict, die aktuelle Fassung des klassischen HTML-Standards in ihrer strengen (nicht abwärtskompatiblen) Form; wird von allen gängigen Browsern ab Version 4.x (ca. 1998) verarbeitet
- `"html4Tr"` – HTML 4.0 Transitional, eine etwas tolerantere Version von HTML 4, die zum Beispiel das veraltete `<font>`-Tag zur Schriftformatierung erlaubt

Eine Version der CGI-Bibliothek, die auch den XML-basierten Standard XHTML anbietet, ist in Arbeit. Sie wird voraussichtlich mit Ruby 1.9 in die Standardbibliothek aufgenommen.

Das folgende Beispiel verwendet HTML 4.0:

```
cgi = CGI.new("html4")
```

Die gesamte HTML-Ausgabe muss von

```
cgi.out { ... }
```

umschlossen werden. Diese Methode erzeugt einen HTTP-Header, falls Sie nicht zuvor `cgi.header` eingesetzt haben, sowie einen Body. Der Body-Inhalt wird dabei als String in den Block geschrieben.

Innerhalb des Body wird das HTML-Dokument mit Header und Body erzeugt; diese Struktur sieht so aus:

```
cgi.html {
  cgi.head { ... } +
  cgi.body { ... }
}
```

Da die Ergebnisse aller HTML-Erzeugungsmethoden Strings sind, werden sie auf jeder Verschachtelungsebene mit Hilfe des Operators + verkettet. Sie können auch jederzeit literale Strings einfügen, einschließlich solcher, die HTML-Code als einfachen Text enthalten. Das folgende Beispiel fügt ohne weitere Formalitäten kursiven Text in den Body ein:

```
cgi.body { "<i>Das ist kursiv</i>" }
```

Da praktisch alle gängigen HTML-Tags als CGI-Methoden zur Verfügung stehen, können Sie dies genauso gut als

```
cgi.body { cgi.i { "Das ist kursiv" } }
```

schreiben.

Auch das Einfügen von Zeilenumbrüchen ist auf diese Weise möglich. Sie machen den erzeugten HTML-Code lesbarer und damit wartungsfreundlicher. Das folgende Beispiel trennt Head und Body durch einen Zeilenumbruch:

```
cgi.html {
  cgi.head { ... } +
  "\n" +
  cgi.body { ... }
}
```

Verwechseln Sie einen einfachen Zeilenumbruch zur HTML-Strukturierung aber nicht mit einem HTML-Zeilenumbruch. Auf einer Webseite wird beliebig viel Whitespace nämlich stets zu genau einem Leerzeichen (abgesehen vom Wortumbruch) zusammengefasst. Wenn Sie einen expliziten Zeilenumbruch benötigen, müssen Sie das HTML-Tag `<br />` verwenden, das Sie bei Bedarf auch mittels `cgi.br` erzeugen können.

Das wichtigste Element innerhalb des Heads ist der Dokumenttitel, der mit Hilfe der Methode `cgi.title` angegeben wird. Zum Beispiel:

```
cgi.head {
  cgi.title { "Titel des Dokuments" }
}
```

Bei vielen HTML-Erzeugungsmethoden können Sie die zulässigen Attribute als Hashargument einfügen. Das folgende Beispiel erzeugt einen Body mit schwarzem Hintergrund und weißer Standardschriftfarbe:

```
cgi.body('bgcolor' => "#000000", 'text' => "#FFFFFF") { ... }
```

Sie sollten wissen, dass moderne Webseiten HTML ausschließlich zur Strukturierung des Inhalts verwenden. HTML-Tags, die für Layout und Design zuständig sind, gelten als veraltet; die meisten sind in strengem HTML 4.0 sowie in XHTML unzulässig, wenngleich sie von den meisten Browsern noch verarbeitet werden. Um das Design kümmert sich exklusiv die Sprache *CSS* (*Cascading Style Sheets*).

Wenn Sie eine externe Stylesheet-Datei einbetten möchten, funktioniert dies mit Hilfe eines `<link>`-Tags im Head. Hier ein vollständiger Head mit Titel und der Einbettung einer CSS-Datei namens *layout.css*, die sich im selben Verzeichnis befindet wie das Ruby-Skript selbst:

```
cgi.head {
  cgi.title { "Eine Seite mit CSS-Layout" } +
  cgi.link(
    'rel' => "stylesheet",
    'type' => "text/css",
    'href' => "layout.css"
  )
}
```

Wenn auch der CSS-Code in dieser Datei den Seitenhintergrund schwarz und den Text weiß darstellen soll, muss er wie folgt lauten:

```
body {
  background-color: #000000;
  color: #FFFFFF
}
```

Wenn Sie weitere Informationen über Stylesheets benötigen, halten Sie sich an die Weblinks und Buchempfehlungen in Anhang.

Es wäre sinnlos, an dieser Stelle die CGI-Methoden zur Erzeugung sämtlicher HTML-Tags aufzuführen. Wenn Sie HTML beherrschen, können Sie davon ausgehen, dass sie alle wie erwartet funktionieren. Nähere Angaben stehen in der `ri`-Dokumentation zu `CGI::HtmlExtension`.

Besonders interessant sind allerdings einige Methoden zur Erzeugung von Formularelementen, da sie eben nicht exakt den zugehörigen HTML-Tags entsprechen. Bereits die Methode `form` selbst funktioniert etwas anders als andere Tag-Methoden, weil die Attribute als Strings und nicht als benannter Hash angegeben werden. Es handelt sich von links nach rechts um folgende Werte:

- HTTP-Methode – Originalattribut `method` – mit den möglichen Werten `"post"` oder `"get"` (ihre Bedeutung wurde bereits besprochen). Der Standardwert ist `"post"`.
- Action-URL – Originalattribut `action`. Die Formulardaten werden an die angegebene URL geschickt, sobald die Absende-Schaltfläche betätigt wird. Wenn Sie diesen Parameter weglassen, wird das aktuelle Skript selbst als Versandziel gewählt.
- Datentyp – Originalattribut `enctype`. Damit geben Sie das Format an, in dem die Formulardaten abgeschickt werden. Der Standardwert ist `application/x-www-form-urlencoded`. Dieses Format ersetzt alle Sonderzeichen, die in URLs nicht zulässig sind, durch passende Maskierungen; weiter oben haben Sie bereits ein Beispiel gesehen. Wenn Sie die Versandmethode `POST` wählen, können Sie auch einen anderen Datentyp einstellen, weil der Browser in diesem Fall einen `Content-type`-Header sendet. Ein wichtiges Beispiel ist der Typ `multipart/form-data`, der für den Upload von Dateien verwendet wird. Genau für diesen Typ steht die Kurzform `cgi.multipart_form("URL")` zur Verfügung.

Das folgende Beispiel erzeugt ein Formular, das die Daten mit der Methode `GET` an die relative URL *pruefung.rb* verschickt:

```
cgi.form("get", "pruefung.rb") { ... }
```

Innerhalb des Formulars können Sie weiterhin beliebige HTML-Inhalte und Text unterbringen, vor allem aber die diversen Eingabeelemente. Die wichtigsten sind:

- Einfache Textfelder. In HTML werden sie mittels

  ```
  <input type="text" name="Feldname" />
  ```

  erzeugt. In Ruby können Sie entweder `cgi.input` mit einem Hash benannter Attribute verwenden, oder aber die elegantere Kurzfassung `text_field`:

  ```
  cgi.text_field("Feldname")
  ```

  Auch dieses Element kann benannte Parameter entgegennehmen. Die wichtigsten sind `name` (Feldname), `size` (Breite in Zeichen), `maxlength` (maximal zulässige Zeichenanzahl) und `value` (Vorgabetext). Betrachten Sie etwa dieses Beispiel:

  ```
  cgi.text_field('name' => "geburtsdatum", 'size' => "10",
                 'maxlength' => "10", 'value' => "TT.MM.JJJJ")
  ```

  Das Ergebnis ist folgender HTML-Code:

  ```
  <INPUT name="geburtsdatum" size="10" TYPE="text"
  value="TT.MM.JJJJ" maxlength="10">
  ```

- Mehrzeilige Textfelder. Für größere Textbereiche ist die Methode `cgi.textarea` zuständig. Sie erzeugt ein `<textarea></textarea>`-Tag-Paar. Beliebige Attribute können als Hash eingefügt werden. Die wichtigsten sind `name` (oder ein einfacher String) für den Namen des Feldes, `cols` für die Breite in Zeichen, `rows` für die Höhe in Zeichen und `wrap` für die Art des Zeilenumbruchs – meist `vir-`

tual, um die Umbrüche zwar darzustellen, aber nicht in den Feldtext zu übernehmen. Das folgende Beispiel erzeugt ein 40 Zeichen breites und sieben Zeichen hohes Feld namens meinung:

```
cgi.textarea("meinung", 'cols' => "40", 'rows' => "7",
             'wrap' => "virtual")
```

Falls Sie einen Block mit Text einfügen, wird dieser zwischen <textarea> und </textarea> gesetzt und somit als Vorgabetext in den Textbereich eingetragen. Zum Beispiel:

```
cgi.textarea("meinung", 'cols' => "40", 'rows' => "7",
             'wrap' => "virtual") { "Ihre Meinung hier!" }
```

- Radio-Button-Gruppen. Diese Steuerelemente haben Sie bereits im Einführungsbeispiel gesehen. Es handelt sich um mehrere Optionsschalter, von denen Sie genau einen auswählen können. Die allgemeine CGI-Syntax dafür lautet:

```
cgi.radio_group("Gruppenname",
                ["Feldwert1", "Beschriftung1"],
                ["Feldwert2", "Beschriftung2"]...)
```

  Wenn Sie die Beschriftung eines Buttons (den zweiten Parameter im jeweiligen Array) weglassen, wird der interne Wert auch als Beschriftungstext verwendet. Optional können Sie bei einem der Buttons als dritten Parameter true einsetzen, damit dieser Button vorausgewählt wird.

  Das folgende Beispiel erzeugt eine Gruppe von Radio-Buttons zur Auswahl der Lieblingsprogrammiersprache, wobei Ruby natürlich vorausgewählt ist:

```
cgi.radio_group("lieblingssprache",
                ["ruby", "Ruby<br />", true],
                ["perl", "Perl<br />"],
                ["java", "Java<br />"],
                ["php", "PHP<br />"],
                ["c", "C"])
```

  Das <br /> in den ersten vier Elementen erzeugt je einen Zeilenumbruch und sorgt so dafür, dass die Buttons untereinander statt nebeneinander dargestellt werden. Alternativ – und passender zur CGI-Bibliothek – könnten Sie "Ruby"+cgi.br und so weiter schreiben.

- Checkbox-Gruppen. Bei dieser Variante können Sie beliebig viele Kontrollkästchen ankreuzen und wieder abwählen – ideal immer dann, wenn mehrere Antworten möglich sind. Die Syntax von cgi.checkbox_group entspricht exakt derjenigen von cgi.radio_group – mit der Ausnahme, dass Sie bei Bedarf mehr als ein Feld vorab ankreuzen können. Hier ein Beispiel zur Auswahl der vom User eingesetzten Betriebssysteme:

```
cgi.checkbox_group("betriebssysteme",
                   ["linux", "Linux"+cgi.br, true],
                   ["win", "Windows"+cgi.br, true],
                   ["mac", "Mac OS X"+cgi.br],
                   ["bsd", "Free/Open/Net/*BSD"+cgi.br],
                   ["x", "sonstige"])
```

Denken Sie daran, dass Sie die Eingaben aus Checkbox-Gruppen mittels `cgi.param` ermitteln müssen, weil Sie nur so mehrere Werte auslesen können.

- Absende-Button. Natürlich brauchen Sie eine Möglichkeit, ein Formular nach dem Ausfüllen abzuschicken. Dafür ist ein Submit-Button zuständig, den Sie wie folgt erzeugen können:

  ```
  cgi.submit("Beschriftung")
  ```

- Reset-Button. Viele Formulare enthalten eine Schaltfläche, die alle Eingaben löscht und den Ausgangszustand wiederherstellt. Sie erstellen dieses Element bei Bedarf so:

  ```
  cgi.reset("Beschriftung")
  ```

Datei-Uploads

Wenn Sie ein Multipart-Formular erstellen, besteht – wie bereits erwähnt – die Möglichkeit, Dateien auf den Server hochzuladen. Für das entsprechende Upload-Feld ist die `CGI`-Methode `file_field` zuständig. Die wichtigsten Parameter, die wieder als Hash gesetzt werden können, sind der bekannte `name` sowie `accept` für die MIME-Typen, die das Feld akzeptiert. Zum Beispiel:

```
cgi.file_field('name' => "textdatei", 'accept' => "text/plain")
```

Beachten Sie, dass die Datentypkontrolle durch das Attribut `accept` nicht zuverlässig ist. Sie müssen den Typ der hochgeladenen Dateien aus Sicherheitsgründen nochmals überprüfen.

Beachten Sie, dass in einem Multipart-Formular die Inhalte *aller* Felder in temporären Dateien gespeichert werden. Der Zugriff erfolgt weiterhin mit Hilfe von `cgi['Feldname']`, allerdings nicht mehr direkt, sondern über die folgenden Methoden:

- `read` liefert den Body, das heißt den Inhalt der jeweiligen Datei, zurück
- `local_path` ist der Pfad der temporären Datei, in der die Datei gespeichert wurde
- `original_filename` gibt den ursprünglichen Dateinamen an
- `content_type` ist der Datentyp (MIME-Typ) des Feldes beziehungsweise der hochgeladenen Datei

Das folgende Komplettbeispiel stellt ein Multipart-Formular mit einem Textfeld und einem Upload-Feld zur Verfügung. Wenn die hochgeladene Datei einen `text/*`-MIME-Typ besitzt, wird ihr Inhalt angezeigt. Die dafür verwendete statische CGI-Methode `escapeHTML` sorgt dafür, dass HTML-relevante Sonderzeichen maskiert werden. Schauen Sie sich den Code, den Sie komplett verstehen müssten, an, und probieren Sie das Beispiel (*upload.rb*) aus:

```
#!C:/ruby/bin/ruby.exe -w

require "cgi"
cgi = CGI.new("html4")

cgi.out {
  cgi.html {
    cgi.head {
      cgi.title { "Uploader" }
    } +
    cgi.body {
      cgi.h1 { "Datei-Uploads" } +
      # Datei und Kommentar vorhanden?
      if cgi.has_key?("kommentar") && cgi.has_key?("datei")
        # Kommentar als Ueberschrift ausgeben
        cgi.h2 { cgi['kommentar'].read } +
        if cgi['datei'].content_type =~ %r|^text/|
          # Datei ausgeben, falls Textdatei
          CGI.escapeHTML(cgi['datei'].read)
        else
          # Ansonsten Fehlermeldung
          "FEHLER: Unzulaessiger Dateityp!"
        end
      else
        ""
      end +
      # Multipart-Formular
      cgi.multipart_form {
        cgi.p {
          "Kommentar: " +
          cgi.text_field("kommentar")
        } +
        cgi.p {
          "Textdatei: " +
          # Datei-Upload-Feld
          cgi.file_field('name' => "datei", 'accept' => "text/plain")
        } +
        cgi.p {
          cgi.reset("Zuruecksetzen") +
          cgi.submit("Hochladen")
        }
      }
    }
  }
}
```

Cookies

Ein großes Problem für umfangreichere Webanwendungen besteht darin, dass HTTP ein *zustandsloses* Protokoll (englisch *stateless*) ist. Nach Browser-Anfrage und Webserver-Antwort ist die Kommunikation abgeschlossen, und die nächste

Anfrage »weiß« nichts mehr von der vorherigen. Webanwendungen mit Warenkorbsystem, datenbankbasierten Katalogen oder persönlicher Anmeldung müssen Zustände aber über mehrere Seitenwechsel hinweg speichern. Zu diesem Zweck wurde unter anderem das Konzept der *Cookies* eingeführt. Es handelt sich um kleine Textinformationen, die der Browser im Auftrag einer bestimmten Website in einer Datei speichert und bei der nächsten Anfrage an dieselbe Site automatisch wieder mitschickt.

Der Server setzt einen Cookie, indem er einen entsprechenden HTTP-Header mitschickt. Auch künftige Anfragen, die der Browser an den Server sendet, enthalten den Cookie als Header. Cookies verfügen über ein *Verfallsdatum*, bis zu dem der Browser sie speichert. Auf diese Weise ist es möglich, dass Besucher auch beim nächsten Besuch ihren Warenkorb oder persönliche Voreinstellungen wiederfinden.

Um mit Hilfe der CGI-Bibliothek einen Cookie zu setzen, müssen Sie zuerst eine Instanz der Klasse `CGI::Cookie` erzeugen. Der Konstruktor erwartet einen Hash mit diversen Parametern für den Cookie. Die wichtigsten sind:

- `name` – der Name, unter dem der Browser den Cookie speichert. Wenn mehrere Seiten Ihrer Website unterschiedliche Informationen als Cookies übertragen sollen, müssen Sie darauf achten, dass Sie diesen verschiedene Namen zuweisen.
- `value` – der zu speichernde Wert, ein String.
- `expires` – das erwähnte Verfallsdatum. Sie können eine `Time`-Instanz verwenden; am einfachsten ist es, die gewünschte Zeitspanne in Sekunden zu `Time.now` zu addieren. Wenn Sie kein Verfallsdatum angeben, gilt der Cookie nur für die Dauer der aktuellen Browsersitzung eines Benutzers (*Session-Cookie*).

Das folgende Beispiel erzeugt einen Cookie namens `lastvisit` mit der in einen String umgewandelten aktuellen Systemzeit und einer Gültigkeitsdauer von einer Woche:

```
cookie = Cookie.new(
  'name' => "lastvisit",
  'value' => Time.new.to_s,
  'expires' => Time.new + 86400 * 7
)
```

Um den Cookie tatsächlich zu setzen, müssen Sie im Argument-Hash von `cgi.header` oder `cgi.out` einen Header mit dem Schlüssel `cookie` hinzufügen. Der entsprechende Wert ist keine `CGI::Cookie`-Instanz, sondern ein Array solcher Instanzen, da eine HTTP-Antwort mehrere Cookies setzen kann. Hier ein Beispiel, das nur den soeben erzeugten `cookie` setzt:

```
cgi.out('cookie' => [cookie]) { ... }
```

Eine Seite, die den Cookie lesen möchte, kann auf die Methode `cookies` des CGI-Objekts zugreifen. Diese liefert einen Hash, in dem alle verfügbaren Cookie-Namen die Schlüssel bilden. Sie können auch hier `has_key?` einsetzen, um zu überprüfen, ob ein bestimmter Cookie existiert. Den Cookie `lastvisit` könnten Sie beispielsweise wie folgt auslesen:

```
if cgi.cookies.has_key?("lastvisit")
  lastvisit = cgi.cookies['lastvisit'].value[0]
else
  lastvisit = "new"
end
```

Wie Sie sehen, erfolgt der Zugriff auf den Wert eines bestimmten Cookies über die Methode `value`. Diese liefert ein Array mit einem oder mehreren Werten zurück; im vorliegenden Beispiel wird der erste Wert ausgelesen.

Im Folgenden wird ein vollständiges Beispiel präsentiert: Im CGI-Skript *farbwaehler.rb* können Sie eine von mehreren Farben auswählen. Diese wird als Cookie gespeichert. Das Skript enthält einen Link zu einem zweiten CGI-Skript namens *farbe.rb*, in dem die Farbe aus dem Cookie ausgelesen und als allgemeine Body-Textfarbe eingestellt wird.

Hier zunächst der komplette Code von *farbwaehler.rb* – nach den vorangegangenen Erläuterungen müssten Sie eigentlich alles verstehen:

```
#!C:/ruby/bin/ruby.exe -w

require "cgi"

cgi = CGI.new("html4")

# Standardwert
farbe = "#000000"

# Eventuell eingegebene Farbe auslesen
if cgi.has_key?("farbe")
  farbe = cgi['farbe']
end

# Cookie zur Speicherung der Farbe erzeugen
cookie = CGI::Cookie.new(
  'name' => "farbe",
  'value' => cgi['farbe'],
  'expires' => Time.new + 86400
)

# HTTP-Header und -Body erzeugen; Header mit Cookie
cgi.out(
    'cookie' => [cookie]
  ) {
  cgi.html {
    cgi.head {
```

```
        cgi.title { "Farbwaehler" } +
        cgi.meta(
          'http-equiv' => "Content-type",
          'content' => "text/html; charset=iso-8859-1"
        )
      } +
      cgi.body {
        cgi.h1 { "Cookie-Farbwaehler" } +
        if cookie
          cgi.p { "Farbe #{farbe} gespeichert." }
        end +
        cgi.form("get") {
          "Farbe: "
          cgi.radio_group(
            "farbe",
            ["#FF0000", "rot   ", true],
            ["#009900", "gruen   "],
            ["#0000FF", "blau"]
          ) +
          cgi.br +
          cgi.submit("Farbe speichern")
        } +
        cgi.a('href' => "farbe.rb") { "Farbe anschauen" }
      }
    }
  }
```

Das Skript *farbe.rb* enthält folgenden Code:

```
#!C:/ruby/bin/ruby.exe -w

require "cgi"

cgi = CGI.new("html4")

# Farbe aus dem Cookie lesen
if cgi.cookies.has_key?("farbe")
  cookie = cgi.cookies['farbe']
  farbe = cookie.value[0]
else
  farbe = "#000000"
end

cgi.out {
  cgi.html {
    cgi.head {
      cgi.title { "Cookie-Schreiber" } +
      cgi.meta(
        'http-equiv' => "content-type",
        'content' => "text/html; charset=iso-8859-1"
      )
    } +
    cgi.body('bgcolor' => "#FFFFFF", 'text' => farbe) {
```

```
        cgi.h1 { "Hier die gewaehlte Farbe aus dem Cookie" } +
        cgi.p { "als Textfarbe" } +
        cgi.a('href' => "farbwaehler.rb") { "Zurueck" }
      }
    }
  }
```

Wie Sie sehen, enthalten beide Skripten die Anweisung:

```
cgi.meta(
  'http-equiv' => "content-type",
  'content' => "text/html; charset=iso-8859-1"
)
```

Das `<meta>`-Tag im Head stellt zusätzliche Optionen ein, die den Browser und eventuell auch diverse Suchmaschinen interessieren. Das Attribut `http-equiv` besagt, dass die Angabe dem betreffenden HTTP-Header entspricht. Im vorliegenden Fall wird der Header `Content-type` zur Angabe des Dateityps überschrieben. Der vollständige Wert ist:

```
text/html; charset=iso-8859-1
```

Der Header wird also verwendet, um neben dem Datentyp auch den zu verwendenden Zeichensatz einzustellen. In diesem Beispiel wird der Standardzeichensatz für Westeuropa, `iso-8859-1`, gesetzt. Sofern Ihr Editor denselben Zeichensatz unterstützt, können Sie danach Umlaute und andere Standard-Sonderzeichen vieler europäischer Sprachen problemlos verwenden.

Eine andere häufig genutzte Zeichensatzeinstellung ist `utf-8`. Dies ist eine spezielle Variante des Weltzeichensatzes Unicode, in der die nur für Englisch ausreichenden ASCII-Zeichen (Code 0 bis 127) mit 8 Bit und alle anderen Zeichen mit 16 bis 32 Bit dargestellt werden. Damit ist `utf-8` im Gegensatz zu anderen Unicode-Darstellungsformen abwärtskompatibel zu ASCII. Beachten Sie aber, dass unter Umständen nicht alle Client-Rechner im Internet jedes Unicode-Zeichen darstellen können, das Sie zum Erstellen Ihrer Seiten verwenden.

Sessions

Für Websites, die Benutzerdaten über beliebig viele Einzelseiten hinweg speichern müssen, sind Cookies erst die halbe Miete. Abgesehen davon, dürfen Sie nie eine Webanwendung schreiben, die sich alternativlos auf Cookies verlässt, weil viele Benutzer ihre Browser so einstellen, dass diese die Annahme von Cookies verweigern. Zumindest die manuelle Bestätigung von Cookies ist sogar empfehlenswert, da manche Werbebanner-Vermarkter (deren Banner auf vielen unterschiedlichen Sites auftauchen) Cookies missbrauchen, um das Surfverhalten von Usern auszuspionieren und umfangreiche Profile darüber anzulegen. Die allgemeine Relativierung des Datenschutzes – angeblich zum »Kampf gegen den Terror« – lässt zudem befürchten, dass sich Konzern- und Staatsmacht solche Informationen künftig brüderlich teilen.

Erfreulicherweise gibt es eine andere Methode, um HTTP nachträglich um eine Zustandsspeicherung zu erweitern: den Einsatz so genannter *Session-Variablen*. Wenn Sie unbedingt möchten, können Sie so etwas auch manuell implementieren; das Verfahren funktioniert im Prinzip immer gleich:

1. Wenn ein Besucher die erste Seite Ihrer Site aufsucht (egal welche), muss eine *Session-ID* generiert werden, das heißt eine eindeutige Nummer, die dieser speziellen Client-Verbindung zugeordnet wird. Eine solche Zahl können Sie leicht selbst generieren, aber noch einfacher und eleganter ist es, einen Generator für UUIDs (Universally Unique IDentity numbers) einzusetzen. In Ruby gehört ein solcher Generator zwar nicht zum Lieferumfang, aber wenn Sie eine Internetverbindung haben, können Sie ihn mit Hilfe der folgenden Kommandozeilenanweisungen leicht installieren:

   ```
   > gem update
   > gem install uuid
   ```

   (Der Umgang mit dem Erweiterungsmanager `rubygems` wird weiter unten noch genauer erläutert.)

   Der Generator erzeugt eine UUID danach aus der weltweit einmaligen MAC-Adresse einer Netzwerkkarte, der Systemzeit und einem Zufallsfaktor. Wann immer Sie eine solche ID benötigen, können Sie die Bibliothek importieren und den Konstruktor aufrufen. Zum Beispiel:

   ```
   require "rubygems"
   require "uuid"
   uuid = UUID.new
   ```

   Die erzeugte UUID ist 128 Bit lang und wird als String aus Hexadezimalziffern und Bindestrichen notiert, zum Beispiel: `"c9ce3001-7bbf-0129-373e-005056c00008"`.

2. Die erzeugte Session-ID muss auf Dauer mit den Anfragen des fraglichen Clients verknüpft werden. Sie können zunächst versuchen, sie als Cookie zu speichern. Falls dies aber aus den oben angesprochenen Gründen scheitert, bleibt Ihnen nur noch die Möglichkeit, die ID in den Query-String oder die `POST`-Daten *jeder* internen Verknüpfung (Hyperlinks, Formularversand usw.) aufzunehmen. In Formulare können Sie dazu einfach ein *Hidden-Feld* (unsichtbares Feld mit einem vorgegebenen Wert) einbetten. Zum Beispiel:

   ```
   cgi.form(...) {
     cgi.hidden('name' => "id", 'value' => uuid)
     ...
   }
   ```

   Bei Hyperlinks müssen Sie die ID jedoch manuell anhängen, zum Beispiel so:

   ```
   cgi.a('href' => "bestellung.rb?id=#{uuid}") { "Bestellen" }
   ```

3. Unter der ID können Sie nun Daten speichern, die jeweils für eine spezielle Client-Session relevant sind. Zwei Speichermöglichkeiten für Session-Daten wären Textdateien, deren Name die jeweilige Session-ID ist, oder eine Datenbankta-

belle. Wie Sie auf Textdateien zugreifen können, wurde bereits in Kapitel 3 erläutert, und Genaueres über die Ruby-Datenbankanbindung erfahren Sie im nächsten Abschnitt.

4. Wenn Sie auf einer neuen Seite ankommen, können Sie die Session-Daten wieder auslesen, indem Sie zunächst nach dem weiter oben beschriebenen Verfahren den Wert des Parameters `id` ermitteln und anschließend auf die betreffende Datei oder Datenbanktabelle zugreifen.

Von der Implementierung einer eigenen Session-Lösung nach dem soeben erläuterten Schema soll an dieser Stelle abgesehen werden (wenn Sie möchten, können Sie es an einem verregneten Sonntagnachmittag selbst probieren). Denn erfreulicherweise enthält Ruby eine eingebaute Bibliothek für Session-Tracking. Um diese zu nutzen, müssen Sie zunächst die entsprechenden Klassen importieren:

```
require "cgi/session"
```

Nun können Sie unter Verwendung Ihres bereits existierenden `CGI`-Objekts (hier wie üblich `cgi` genannt) eine `Session`-Instanz erzeugen:

```
session = CGI::Session.new(cgi)
```

Diese Instanz kann als Hash verwendet werden, um beliebige Werte als Session-Daten zu speichern. Dies muss vor dem Senden der Header, also vor `cgi.header` beziehungsweise `cgi.out`, geschehen. Dieses Beispiel speichert den Wert eines Formularfeldes in der Session:

```
session['username'] = cgi['username']
```

Um Session-Variablen auf einer anderen Seite wieder auszulesen, brauchen Sie nur wieder ein Session-Objekt zu erzeugen und den betreffenden Hashschlüssel darauf anzuwenden. Dazwischen dürfen beliebig viele CGI-Skripten oder gar einfache HTML-Seiten liegen, die keine Session-Daten verwenden – die Ruby-Session-Bibliothek kümmert sich im Hintergrund automatisch um die Session-Verwaltung.

Durch zusätzliche Parameter und Unterklassen gibt es diverse Möglichkeiten, die genaue Arbeitsweise von Sessions zu beeinflussen. Sie können zum Beispiel wählen, ob die Daten in Textdateien (Standard), im Arbeitsspeicher (besonders schnell, aber nur für wenig datenintensive Anwendungen geeignet) oder in Datenbanken (auch komplexe Datenstrukturen möglich) abgelegt werden sollen. Die Betrachtung solcher spezieller Fähigkeiten würde an dieser Stelle allerdings zu weit führen. Lesen Sie beispielsweise die `ri`-Dokumentation zu `CGI::Session`, wenn Sie Näheres wissen möchten.

Es folgt – wie bei den Cookies – ein Beispiel aus zwei Skripten. Im ersten Skript werden die aus einem Formular gelesenen Schrifteinstellungen in der Session gespeichert. Das zweite Skript liest den Wert aus und setzt ihn per CSS im Seitentext um. Sie erfahren hier nebenbei, wie Sie CSS-Code direkt in Ihr CGI-Skript einbetten können, ohne eine externe Datei zu verwenden.

Hier zunächst der Code des Formularskripts, *schriftwahl.rb* – auch in diesem Skript sollte Ihnen inzwischen alles geläufig sein:

```
#!C:/ruby/bin/ruby.exe -w

require "cgi"
require "cgi/session"

# CGI- und Session-Objekt erzeugen
cgi = CGI.new("html4")
session = CGI::Session.new(cgi)

# Formulardaten auslesen oder Standardwerte
if cgi.has_key?("schrift")
  schrift = cgi['schrift']
else
  schrift = "Arial, Helvetica, sans-serif"
end

if cgi.has_key?("groesse")
  groesse = cgi['groesse']
else
  groesse = "12px"
end

if cgi.has_key?("farbe")
  farbe = cgi['farbe']
else
  farbe = "#000000"
end

# Daten in Session speichern
session['schrift'] = schrift
session['groesse'] = groesse
session['farbe'] = farbe

# Seite mit Formular anzeigen
cgi.out {
  cgi.html {
    cgi.head {
      cgi.title { "Schrifteinstellungen waehlen" } +
      cgi.meta(
        'http-equiv' => "Content-type",
        'content-type' => "text/html; charset=iso-8859-1"
      )
    } +
    cgi.body {
      cgi.h1 { "Waehlen Sie Ihre Schrifteinstellungen" } +
      cgi.form('method' => "get") {
        cgi.table('border' => "0", 'cellpadding' => "8") {
          cgi.tr {
            cgi.td { "Schriftart:" } +
```

```
              cgi.td {
                cgi.popup_menu("schrift",
                  ["Arial, Helvetica, sans-serif", "serifenlos", true],
                  ["Times New Roman, Times, serif", "Serifenschrift"],
                  ["Courier New, Courier, mono", "Festbreitenschrift"])
              }
            } +
            cgi.tr {
              cgi.td { "Schriftgroesse:" } +
              cgi.td {
                cgi.radio_group("groesse",
                  ["10px", "klein  "],
                  ["12px", "normal  ", true],
                  ["16px", "gross"])
              }
            } +
            cgi.tr {
              cgi.td { "Schriftfarbe:" } +
              cgi.td {
                cgi.popup_menu("farbe",
                  ["#000000", "schwarz", true],
                  ["#FF0000", "rot"],
                  ["#009900", "gruen"],
                  ["#0000FF", "blau"])
              }
            } +
            cgi.tr {
              cgi.td('colspan' => "2") {
                cgi.submit("Einstellungen uebernehmen")
              }
            }
          } +
          cgi.p {
            "Ihre Einstellungen " +
            cgi.a('href' => "schrift.rb") { "ansehen" }
          }
        }
      }
    }
  }
```

Das Gegenstück *schrift.rb*, in dem die Session-Variablen ausgelesen werden, sieht so aus:

```
#!C:/ruby/bin/ruby.exe -w

require "cgi"
require "cgi/session"

# CGI- und Session-Objekt erzeugen
cgi = CGI.new("html4")
session = CGI::Session.new(cgi)
```

```
# Session-Daten auslesen oder Standardwerte
if session['schrift']
  schrift = session['schrift']
else
  schrift = "Arial, Helvetica, sans-serif"
end

if session['groesse']
  groesse = session['groesse']
else
  groesse = "16px"
end

if session['farbe']
  farbe = session['farbe']
else
  farbe = "#000000"
end

#!C:/ruby/bin/ruby.exe -w
cgi.out {
  cgi.html {
    cgi.head {
      cgi.title { "Schrifteinstellungen waehlen" } +
      cgi.meta(
        'http-equiv' => "Content-type",
        'content-type' => "text/html; charset=iso-8859-1"
      ) +
      cgi.style('type' => "text/css") {
        "body {
          font-family: #{schrift};
          font-size: #{groesse};
          color: #{farbe}
        }"
      }
    } +
    cgi.body {
      cgi.p {
        "Hier sehen Sie die aktuellen
         Schrifteinstellungen im Einsatz."
      } +
      cgi.p {
        "Klicken Sie " +
        cgi.a('href' => "schriftwahl.rb") { "hier" } +
        ", um die Werte zu aendern."
      }
    }
  }
}
```

Wie Sie sehen, besitzen CGI::Session-Instanzen zwar keine has_key?-Methode wie CGI-Objekte und Cookies, aber dafür können Sie das if direkt auf session['Schluessel'] anwenden.

Zugriff auf Datenbanken

Die Mehrheit aller modernen Webanwendungen greift auf eine Datenbank zurück, um die Seitenstruktur mit dynamischen Inhalten zu füllen sowie um benutzergenerierte Inhalte dauerhaft zu speichern. Natürlich können Sie Ihre Katalogdaten, Foren, Gästebücher und andere Inhalte auch in Textdateien unterbringen. Aber bei häufigen Zugriffen verlangsamt dies Ihre Anwendungen, und Sie müssten sich selbst darum kümmern, gleichzeitige Schreibzugriffe in den Griff zu bekommen. In einer Datenbank können Sie dagegen fast beliebig umfangreiche Informationen speichern, und um den Zugriffsschutz und ähnliche Stabilitätsfragen kümmern sich gute Datenbankserver auch automatisch.

Wenn Sie Anwendungen mit dem im nächsten Kapitel exklusiv vorgestellten Web-Framework Ruby on Rails erstellen, basieren diese von vornherein auf einer Datenbankstruktur. Die eingebaute Datenbankschnittstelle *Active Record* erledigt dabei hinter den Kulissen die Umsetzung Ihrer Ruby-Objekte in Datensätze und umgekehrt. In diesem Abschnitt geht es jedoch um die traditionelle Art des Datenbankzugriffs: Ihre Anwendung sendet dem Datenbankserver SQL-Code;[2] dieser antwortet auf Änderungsbefehle mit einer Erfolgs- oder Fehlermeldung und auf Lesebefehle mit Datensätzen, die Ihren Suchkriterien entsprechen.

Datenbanksysteme gibt es heutzutage wie Sand am Meer. Die meisten von ihnen sind so genannte *relationale Datenbanken*, die ihre Daten in miteinander verknüpfbaren Tabellen speichern. Innerhalb dieser Gruppe gibt es Desktop-Datenbanken wie Microsoft Access oder OpenOffice.org Base und Datenbankserver wie MySQL oder Oracle. Für dieses Buch fiel die Wahl auf MySQL, den verbreitetsten Open Source-Datenbankserver. Bevor der Ruby-Zugriff auf diese Software erläutert wird, werden kurz die Installation und die ersten Schritte erklärt. Falls Sie weitere Informationen über MySQL benötigen, finden Sie im O'Reilly Verlag eine Reihe passender Bücher – etwa meinen in der vorliegenden Buchreihe erschienenen Band *Praktischer Einstieg in MySQL mit PHP*.

MySQL installieren

MySQL funktioniert unter Windows und allen gängigen UNIX-Systemen. Eine Besonderheit gegenüber den meisten anderen Open Source-Projekten ist, dass auch für die meisten UNIX-Plattformen fertige Binärversionen angeboten werden, die Sie nicht mehr zu kompilieren brauchen. Sie können die passende Version für Ihr System von der Website *http://www.mysql.com* herunterladen. Genaue Download-Links finden Sie auf der Site zu diesem Buch, weil die Site von MySQL selbst leider des Öfteren umstrukturiert wird.

2 *SQL* (Structured Query Language) ist eine Sprache zur Kommunikation mit Datenbanksystemen. Jede ernst zu nehmende Datenbank beherrscht diese Sprache – leider in teilweise sehr unterschiedlichen Dialekten.

Die aktuelle stabile Version von MySQL ist 5.0; das neueste Release ist zurzeit (Ende Dezember 2006) 5.0.27; besonders Letzteres wird sich voraussichtlich bald ändern. Die nachfolgenden Installationsanleitungen für Windows und für UNIX-Systeme gehen davon aus, dass Sie die entsprechende Binärversion für Ihre Plattform bereits heruntergeladen haben.

Installation unter Windows

Für Windows bieten die MySQL-Entwickler einen bequemen Binär-Installer an. Entpacken Sie als Erstes die ZIP-Datei (zurzeit *mysql-5.0.27-win32.zip*). Anschließend können Sie die enthaltene Datei *Setup.exe* per Doppelklick ausführen. Falls auf Ihrem Rechner bereits eine ältere MySQL-Version existiert, wird sie deinstalliert und durch die neue ersetzt; die vorhandenen Datenbanken bleiben erhalten. Als Erstes müssen Sie sich den Installationsumfang aussuchen: *Typical* (die wichtigsten Komponenten), *Complete* (alles) oder *Custom* (freie Auswahl). Wählen Sie Letzteres und klicken Sie auf dem nächsten Bildschirm auf die Schaltfläche *Change*, wenn Sie das Installationsverzeichnis wechseln möchten. Ein Klick auf *Next* startet anschließend die Installation.

Danach können Sie *Configure the MySQL Server now* wählen, um den Datenbankserver nach Ihren Wünschen einzurichten – beispielsweise wird er als automatisch startender Dienst installiert. Der Konfigurationsdialog ist später jederzeit über das Startmenü verfügbar. Beantworten Sie die Fragen für einen Programmierrechner wie folgt:

1. Wählen Sie den Installationstyp *Detailed Configuration*.
2. Entscheiden Sie sich für den Servertyp *Developer Machine* – MySQL wird nicht mit Priorität ausgeführt, sondern – wie auf einem Arbeitsrechner zu empfehlen – als ein Programm unter vielen.
3. Als Datenbanktyp sollten Sie *Multifunctional Database* wählen, um flexibel mit allen MySQL-Tabellentypen arbeiten zu können.
4. Unter *Tablespace* können Sie das Laufwerk und das Verzeichnis für Ihre MySQL-Datenbanken einstellen.
5. Wählen Sie *Decision Support* für maximal 20 gleichzeitige Client-Verbindungen – mehr braucht ein Entwicklungsrechner nicht.
6. Behalten Sie den Standard-TCP-Port 3306 bei; auch *Enable Strict Mode* für eine strengere Datenbankabfrage-Syntax sollten Sie eingeschaltet lassen.
7. Die beste Zeichensatzeinstellung für Webanwendungen ist *Standard Character Set* (ISO-8859-1).
8. Wählen Sie *Install As Windows Service* und *Launch the MySQL Server automatically*, um MySQL als Dienst zu installieren und beim Booten automatisch zu

starten. *Include Bin Directory in Windows PATH* sorgt dafür, dass Sie die MySQL-Kommandozeilen-Hilfsprogramme aus jedem Verzeichnis aufrufen können.

9. Aktivieren Sie *Modify Security Settings* und geben Sie ein Passwort für den MySQL-Administrator `root` ein. *Enable root access from remote machines* sollte deaktiviert bleiben – die Fernadministration von Servern sollte ein weniger privilegierter Benutzer vornehmen. *Create An Anonymous Account* ist in der Regel ebenfalls nicht empfehlenswert.
10. Wenn Sie auf *Execute* klicken, werden die gewählten Änderungen durchgeführt. Falls es nicht klappen sollte, müssen Sie überprüfen, ob eventuell bereits ein älterer MySQL-Dienst läuft oder ob Ihre lokale Firewall den Port 3306 blockiert.

Installation auf UNIX-Systemen

Das *Mac OS X*-Paket von MySQL ist ein bequemer Installer. Bevor Sie ihn ausführen, müssen Sie unter *Systemeinstellungen* → *Benutzer* überprüfen, ob bereits ein User namens *mysql* existiert, und diesen ansonsten neu anlegen. Anschließend können Sie die Image-Datei (*.dmg*) per Doppelklick mounten und den enthaltenen Installer durch einen weiteren Doppelklick starten. MySQL wird unter */usr/local/mysql-Version* installiert; zudem erstellt der Installer den Symlink */usr/local/mysql*. Anschließend wird automatisch das Skript *mysql_install_db* zur Erstellung der Verwaltungstabellen ausgeführt.

Bei der *Linux*-Binärvariante ist etwas mehr Handarbeit erforderlich. Vor der Installation sollten Sie auch hier einen Benutzer und eine Gruppe namens `mysql` einrichten, damit MySQL aus Sicherheitsgründen unter diesen IDs ausgeführt wird. Unter Linux und vielen anderen UNIX-Systemen funktioniert das mit den folgenden beiden Befehlen (als `root`):

```
# groupadd mysql
# useradd -g mysql mysql
```

Anschließend wird das Binärpaket entpackt. Zum Beispiel:

```
# tar xzvf mysql-max-5.0.27-linux-i686-glibc23.tar.gz
```

Verschieben Sie das entpackte Verzeichnis nach */usr/local* und erstellen Sie einen symbolischen Link darauf, der einfach *mysql* heißt:

```
# mv mysql-max-5.0.27-linux-i686-glibc23 /usr/local
# ln -s /usr/local/mysql-max-5.0.27-linux-i686-glibc23 \
 /usr/local/mysql
```

Danach können Sie in das Verzeichnis */usr/local/mysql* wechseln und müssen dort das Skript *mysql_install_db* im Unterverzeichnis *bin* aufrufen, das die Datenbank *mysql* mit den Tabellen für Benutzerrechte und weitere Verwaltungsaufgaben einrichtet:

```
# cd /usr/local/mysql
# bin/mysql_install_db
```

Als Nächstes müssen einige Benutzer- und Gruppenrechte angepasst werden – die Dateien in allen MySQL-Unterverzeichnissen sollen dem User *root* und der Gruppe *mysql* gehören, das Unterverzeichnis *data* mit den Datenbanken dagegen auch dem Benutzer *mysql*. Geben Sie dazu folgende Anweisungen ein:

```
# chown -R root .
# chown -R mysql data
# chgrp -R mysql .
```

Anschließend können Sie den Datenbank-Server starten:

```
# /usr/local/mysql/bin/mysqld_safe --user=mysql &
```

Wenn MySQL beim Booten automatisch gestartet werden soll, müssen Sie in Ihrem Verzeichnis für Startskripten (unter vielen Linux-Distributionen beispielsweise */etc/init.d*) einen symbolischen Link auf das Skript *support_files/mysql.server* aus dem MySQL-Verzeichnis erstellen. Zum Beispiel:

```
# ln -s /usr/local/mysql/support_files/mysql.server /etc/init.d/mysql
```

Anschließend muss dieses Skript auf eine jeweils betriebssystemspezifische Art und Weise aktiviert werden. SUSE und einige andere Linux-Distributionen bieten dafür zum Beispiel das bequeme Kommando *chkconfig* an:

```
# chkconfig -a mysql
```

Nach Installation und Start des MySQL-Servers können Sie zum Beispiel den Kommandozeilen-Client *mysql* verwenden, um Datenbanken einzurichten und zu verwalten. Er befindet sich im *bin*-Verzeichnis der Installation und kann beim ersten Start ohne Passwort ausgeführt werden:

```
# /usr/local/mysql/bin/mysql
```

Die neue Eingabeaufforderung `mysql>` zeigt, dass Sie sich nun innerhalb dieses Clients befinden. Hier sollten Sie sich aus Sicherheitsgründen zuerst darum kümmern, ein Passwort für `root@localhost` – die Administration vom lokalen Rechner aus – festzulegen:

```
mysql> SET PASSWORD FOR root@localhost = PASSWORD("meinPasswort");
```

Anschließend sollten Sie alle Zugangsberechtigungen mit leerem Benutzernamen und/oder leerem Passwort entfernen:

```
mysql> DELETE FROM mysql.user WHERE user="" OR password="";
```

Nun können Sie den Client zunächst wieder verlassen, indem Sie `exit` oder `\q` eingeben. Wenn Sie ihn das nächste Mal starten, müssen Sie die Optionen `-u Benutzername` und `-p` (Passworteingabeaufforderung) verwenden:

```
$ mysql -u root -p
[Passwort eingeben]
```

Erste Schritte mit MySQL

Wie bereits erwähnt, ist MySQL eine so genannte *relationale Datenbank*. Dieser Begriff besagt, dass die Daten in Tabellen organisiert sind, die zueinander in Beziehung (Relation) gesetzt werden können. Dies sorgt dafür, dass Daten niemals doppelt in der Datenbank gespeichert werden müssen. Denken Sie zum Beispiel an eine CD-Datenbank: Eine Tabelle enthält Informationen über die einzelnen CDs wie Interpret, Titel, Spielzeit und so weiter. Da mehrere CDs vom selben Interpreten stammen können, empfiehlt es sich, die Interpreten in einer separaten Tabelle zu speichern. Der Eintrag `Interpret` in der CD-Tabelle ist dann lediglich die Nummer des entsprechenden Interpreten aus der Interpreten-Tabelle.

Für die Arbeit mit relationalen Datenbanken sollten Sie einige Begriffe kennen: Eine Tabellenzelle, die eine Information über einen einzelnen Gegenstand enthält, heißt *Datenfeld*. Eine Zeile mit sämtlichen Informationen über einen Gegenstand wird *Datensatz* genannt (englisch *Record*). Wenn Sie aus einer Tabelle heraus auf Datensätze einer anderen Tabelle verweisen möchten, benötigen diese Datensätze je ein Feld mit einem einmaligen Wert. Dieses spezielle Feld heißt *Primärschlüssel* (primary key). Der Primärschlüssel wird oft durch einfaches Durchnummerieren gebildet. Einige Objekte besitzen dagegen eine Art »natürlichen« Primärschlüssel: Bei Autos ist dies etwa das amtliche Kennzeichen, bei Büchern dagegen die ISBN.

Fast alle relationalen Datenbanken können über eine Sprache namens *SQL* (Structured Query Language) gesteuert werden – MySQL leitet sogar seinen Namen davon ab. Es handelt sich um eine leicht zu erlernende, mächtige Sprache, in der so genannte Abfragen formuliert werden. Damit können Sie Tabellen erstellen, ändern und vor allem Informationen daraus erhalten. Selbstverständlich würde eine systematische Einführung in MySQL in diesem Kapitel zu weit führen. Hier finden Sie nur ein kurzes Beispiel, das Sie direkt eintippen können, wenn Sie den oben erwähnten `mysql`-Client starten:

```
> mysql -u root -p
[Passwort eingeben]
```

Das Beispiel erstellt die Datenbank *musik* mit zwei Tabellen: *interpreten* mit den Feldern *int_nr* und *int_name* sowie *cds* mit den Feldern *cd_nr*, *cd_interpret*, *cd_titel*, *cd_jahr* und *cd_songs* (Anzahl der Songs). Geben Sie dazu Folgendes ein:

```
mysql> CREATE DATABASE musik;
Query OK, 1 row affected (0.24 sec)
mysql> use musik
Database changed
mysql> CREATE TABLE interpreten
    -> (int_nr INT AUTO_INCREMENT PRIMARY KEY, int_name VARCHAR(40));
Query OK, 0 rows affected (0.28 sec)
mysql> CREATE TABLE cds (cd_nr INT AUTO_INCREMENT PRIMARY KEY,
    -> cd_interpret INT, cd_titel VARCHAR(40), cd_jahr YEAR,
    -> cd_songs INT);
Query OK, 0 rows affected (0.07 sec)
```

Wie Sie sehen, besitzt jedes Feld einen festgelegten Datentyp. Hier werden `INT` (Ganzzahl), `VARCHAR(n)` (Text mit einer Länge von maximal *n* Zeichen) und `YEAR` (Jahreszahl) verwendet. *int_nr* beziehungsweise *cd_nr* sind die Primärschlüssel (`PRIMARY KEY`) der beiden Tabellen; der Zusatz `AUTO_INCREMENT` besagt, dass sie automatisch durchnummeriert werden sollen.

Fügen Sie nun mit Hilfe der beiden folgenden Zeilen Beispielwerte in die Tabelle *interpreten* ein:

```
mysql> INSERT INTO interpreten (int_name) VALUES ("Led Zeppelin");
Query OK, 1 row affected (0.08 sec)
mysql> INSERT INTO interpreten (int_name) VALUES ("Iron Maiden");
Query OK, 1 row affected (0.06 sec)
```

Nun können Sie sämtliche Inhalte der Tabelle *interpreten* auswählen, damit diese angezeigt wird:

```
mysql> SELECT * FROM interpreten;
+--------+--------------+
| int_nr | int_name     |
+--------+--------------+
|      1 | Led Zeppelin |
|      2 | Iron Maiden  |
+--------+--------------+
```

Dies zeigt die automatisch eingefügten Nummern an, die Sie benötigen, um den CDs die korrekten Interpreten zuzuordnen. Mit dieser Information können Sie sich daranmachen, CDs einzugeben:

```
mysql> INSERT INTO cds (cd_interpret, cd_titel, cd_jahr, cd_songs)
    -> VALUES (2, "Seventh Son Of A Seventh Son", 1988, 8);
Query OK, 1 row affected (0.03 sec)
mysql> INSERT INTO cds (cd_interpret, cd_titel, cd_jahr, cd_songs)
    -> VALUES (2, "The Number Of The Beast", 1982, 9);
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO cds (cd_interpret, cd_titel, cd_jahr, cd_songs)
    -> VALUES (1, "Led Zeppelin IV", 1971, 8);
Query OK, 1 row affected (0.00 sec)
```

Der folgende Befehl zeigt die gesamte Tabelle *cds* an, und zwar aufsteigend (ASCending) nach Jahr sortiert (die Titel wurden aus Platzgründen leicht gekürzt):

```
mysql> SELECT * FROM CDS ORDER BY cd_jahr ASC;
+-------+--------------+--------------------+---------+----------+
| cd_nr | cd_interpret | cd_titel           | cd_jahr | cd_songs |
+-------+--------------+--------------------+---------+----------+
|     3 |            1 | Led Zeppelin IV    |    1971 |        8 |
|     2 |            2 | The Number Of T... |    1982 |        9 |
|     1 |            2 | Seventh Son Of ... |    1988 |        8 |
+-------+--------------+--------------------+---------+----------+
3 rows in set (0.07 sec)
```

Der wichtigste Bestandteil einer SELECT-Anfrage ist eine WHERE-Klausel, die Bedingungen dafür formuliert, welche Datensätze ausgewählt werden sollen. Das folgende Beispiel zeigt nur die beiden CDs von Iron Maiden an:

```
mysql> SELECT * FROM cds WHERE cd_interpret=2;
+-------+-------------+---------------------+---------+----------+
| cd_nr | cd_interpret | cd_titel           | cd_jahr | cd_songs |
+-------+-------------+---------------------+---------+----------+
|     1 |           2 | Seventh Son Of A S... |  1988 |        8 |
|     2 |           2 | The Number Of The ... |  1982 |        9 |
+-------+-------------+---------------------+---------+----------+
2 rows in set (0.00 sec)
```

Zu guter Letzt sollten Sie sich noch ein Beispiel dafür anschauen, wie Relationen in der Praxis genutzt werden. Dazu soll eine Abfrage formuliert werden, die den Interpreten und den Titel jeder CD ausgibt. Die Beziehung kann entweder über eine relativ komplizierte JOIN-Klausel oder mit Hilfe von WHERE formuliert werden; hier sehen Sie Letzteres:

```
mysql> SELECT int_name, cd_titel FROM interpreten, cds
    -> WHERE int_nr=cd_interpret;
+--------------+-----------------------------+
| int_name     | cd_titel                    |
+--------------+-----------------------------+
| Iron Maiden  | Seventh Son Of A Seventh Son |
| Iron Maiden  | The Number Of The Beast     |
| Led Zeppelin | Led Zeppelin IV             |
+--------------+-----------------------------+
3 rows in set (0.08 sec)
```

Wenn mehrere Tabellen gleichnamige Felder enthalten, die Sie ansprechen möchten, müssen Sie diese übrigens in der Form Datenbankname.Feldname notieren, etwa cds.cd_titel oder interpreten.int_nr. Das vorliegende Beispiel kommt ohne dieses Hilfsmittel aus, weil in jeder Tabelle konsequent Namenspräfixe (hier cd_ beziehungsweise int_) verwendet werden. Dies ist auch in der Praxis empfehlenswert.

Für den Zugriff durch Webanwendungen sollten Sie zu guter Letzt einen separaten MySQL-Benutzer erzeugen, der nur auf die jeweilige einzelnde Datenbank zugreifen darf. Das folgende Beispiel erzeugt im Kommandozeilen-Client einen Benutzer namens dbuser, der in Skripten auf demselben Rechner mit dem Passwort geheim[3] auf die Datenbank musik zugreifen darf:

```
mysql> CREATE USER dbuser@localhost;
mysql> GRANT ALL PRIVILEGES ON musik.* TO dbuser@localhost
    -> IDENTIFIED BY "geheim";
```

3 In der Praxis dürfen Sie natürlich kein so einfaches Passwort wählen!

Wenn dieser User sofort aktiv werden soll, ohne dass Sie den Client *mysql* beenden, müssen Sie danach die Benutzerdaten neu laden:

```
mysql> FLUSH PRIVILEGES;
```

Ruby-Zugriff auf MySQL-Datenbanken

Ruby ist mit einer Datenbankschnittstelle namens `DBI` ausgestattet, die für jede konkrete Datenbank einen Treiber (`DBD` – DataBase Driver) benötigt. Der passende Treiber für MySQL muss manuell kompiliert werden. Dies gelingt auf den meisten UNIX-Systemen problemlos, weil dort ein C-Compiler zum Standardlieferumfang gehört. Unter Windows müssten Sie zur Installation des DBD-Treibers dagegen einen teuren Microsoft-Compiler einsetzen, weil die Ruby-Makefiles standardmäßig für diese Variante konfiguriert sind. Aus diesem Grund wird DBI an dieser Stelle nicht weiter vertieft.

Glücklicherweise gibt es eine andere Lösung in Form einer direkten MySQL-Klasse. Für Windows kann sie wie folgt über den Erweiterungsmanager `rubygems` (siehe den folgenden Kasten) installiert werden:

```
> gem update
> gem install mysql
```

Um die Erweiterung unter UNIX zu installieren, müssen Sie sie dagegen aus dem Quellcode-Paket installieren. Laden Sie sie dazu von *http://tmtm.org/downloads/mysql/ruby/* herunter; die aktuelle Version ist derzeit 2.7. Entpacken Sie das Archiv wie folgt:

```
# tar xzvf mysql-ruby-2.7.tar.gz
```

Wechseln Sie danach in das neue Unterverzeichnis:

```
# cd mysql-ruby-2.7
```

Nun wird das Skript *extconf.rb* unter Angabe Ihres MySQL-Verzeichnisses aufgerufen. Zum Beispiel:

```
# ruby extconf.rb /usr/local/mysql
```

Zum Schluss werden `make` und `make install` aufgerufen:

```
# make
# make install
```

Der Ruby-Erweiterungsmanager rubygems

`rubygems` ist ein verteiltes Internetarchiv für Ruby-Erweiterungen. Ein Netzwerk von Mirror-Servern, aus denen automatisch ausgewählt wird, macht den Download von Paketen schnell und zuverlässig.

In neueren Ruby-Versionen ist die Unterstützung bereits ab Werk vorhanden. Wenn nicht, müssen Sie zunächst `rubygems` selbst installieren. Besuchen Sie dazu die Site *http://www.rubyforge.org* und verwenden Sie die Suchfunktion, um das Paket `rubygems` zu finden. Laden Sie die für UNIX geeignete *.tar.gz*- oder die zu Windows passende *.zip*-Datei herunter. Entpacken Sie das Archiv und führen Sie dann auf der Konsole innerhalb des ausgepackten Verzeichnisses folgendes Kommando aus:

```
> ruby setup.rb
```

Nachdem `rubygems` zur Verfügung steht, können Sie den Manager über das Konsolenkommando `gem` bedienen. Der erste Schritt sollte stets darin bestehen, die vorhandenen durch `rubygems` installierten Erweiterungen zu aktualisieren. Dies geschieht folgendermaßen:

```
> gem update
```

Um ein Paket zu installieren, dessen Namen Sie bereits kennen, genügt danach die Eingabe:

```
> gem install Paketname
```

Das folgende Beispiel dient der Installation des Frameworks Ruby on Rails, um das es im nächsten Kapitel geht:

```
> gem install rails
```

Wenn das zu installierende Paket von anderen Paketen abhängt, werden Sie jeweils einzeln gefragt, ob Sie diese installieren möchten. Um alle Abhängigkeiten automatisch zu erfüllen, können Sie die Option `--include-dependencies` hinzufügen. Zum Beispiel:

```
> gem install --include-dependencies
```

`gem` bietet auch die Möglichkeit, nach Paketen zu suchen, in deren Namen eine bestimmte Zeichenfolge vorkommt. Ein einfaches

```
> gem search Zeichenfolge
```

sucht dabei in den lokalen, das heißt bereits installierten *Gems*. Fügen Sie `--remote` hinzu, um das Online-Archiv zu durchsuchen. Das folgende Beispiel sucht nach der weiter oben installierten `uuid`:

```
> gem search uuid --remote

*** REMOTE GEMS ***
```

→

```
uuid (1.0.3, 1.0.2, 1.0.1, 1.0.0)
    UUID generator

uuid4r (0.1)
    This generates and parses UUID, based on OSSP uuid C library

uuidtools (1.0.0, 0.1.4, 0.1.3, 0.1.2, 0.1.1, 0.1.0)
    Generation of UUIDs.
```

Um Bibliotheken zu nutzen, die Sie über `rubygems` heruntergeladen haben, sollten Sie immer zuerst die Bibliothek `rubygems` importieren. Neuere Ruby-Versionen erledigen das zwar automatisch (über die automatisch gesetzte Umgebungsvariable `rubyopt`), aber es bringt auch dann keine Nachteile, die zusätzliche Zeile hinzuschreiben. Das folgende Beispiel importiert die `uuid`-Bibliothek auf sichere Weise:

```
require "rubygems"
require "uuid"
```

Wenn Sie selbst nützliche Ruby-Erweiterungen geschrieben haben, können Sie sie der Ruby-Gemeinde auch über `rubygems` zur Verfügung stellen. Eine ausführliche Anleitung über dieses und viele andere `rubygems`-Themen finden Sie unter *http://rubygems.org*.

Der MySQL-Zugriff über die Bibliothek `mysql` erfordert zunächst die passenden `require`-Anweisungen:

```
require "rubygems"
require "mysql"
```

Anschließend können Sie den Konstruktor der Klasse `Mysql` aufrufen, um eine Verbindung zum Datenbankserver herzustellen. Die notwendigen Parameter sind Host, Benutzername und Passwort. Das folgende Beispiel stellt eine Verbindung mit den Rechten des weiter oben erstellten Benutzers `dbuser` her:

```
conn = Mysql.new("localhost", "dbuser", "geheim")
```

Anschließend sollten Sie eine Standarddatenbank wählen, mit der gearbeitet werden soll. Andernfalls müssten Sie ständig `Datenbank.Tabelle` schreiben. Dies geschieht mit Hilfe der Methode `select_db`. Die Datenbank *musik*, auf die der Benutzer `dbuser` zugreifen darf, wird beispielsweise wie folgt ausgewählt:

```
conn.select_db("musik")
```

Nun lässt sich die Verbindung ganz einfach für Datenbankabfragen verwenden. So genannte *Auswahlabfragen* mit `SELECT`, die Datensätze zurückliefern, sollten Sie einer Variablen zuweisen. Diese hat den Datentyp `Mysql::Result` und enthält verschiedene Methoden zum Auslesen des Ergebnisses. Das folgende Beispiel ermittelt die Kombination aus Interpret, Album und Erscheinungsjahr:

```
result = conn.query(
  "SELECT int_name, cd_titel, cd_jahr
  FROM interpreten, cds
  WHERE int_nr=cd_interpret"
)
```

Die erste Methode von `result`, die Sie aufrufen sollten, ist `num_rows`. Sie liefert die Anzahl der Ergebnisdatensätze zurück. Wenn sie 0 ist, entsprach kein Datensatz Ihrer Abfrage. Zum Beispiel:

```
if result.num_rows > 0
  # Ergebnis ausgeben
else
  puts "Keine Datensaetze gefunden."
end
```

Zum Auslesen der Ergebnisse kommen vor allem zwei Methoden in Frage:

- `fetch_row` liest einen Datensatz als nummeriertes Array aus. Die Reihenfolge der Felder innerhalb einer Ergebniszeile entspricht den Angaben in Ihrer Abfrage; bei * (alle Felder) wird die Reihenfolge der Tabelle selbst gewählt.
- `fetch_hash` liest ebenfalls genau einen Datensatz aus. Der einzige Unterschied besteht darin, dass Sie einen Hash zurückerhalten, in dem die Spaltennamen aus den Datenbanktabellen die Schlüssel bilden.

Beide Methoden lassen sich idealerweise innerhalb der Bedingung einer `while`-Schleife platzieren, um alle Datensätze nacheinander auszulesen.

Hier ein Beispiel für den Einsatz von `fetch_row`:

```
while line = result.fetch_row
  printf "'%s' von %s (%i)\n", line[1], line[0], line[2]
end
```

Das müsste folgende Ausgabe liefern:

```
'Seventh Son Of A Seventh Son' von Iron Maiden (1988)
'The Number Of The Beast' von Iron Maiden (1982)
'Led Zeppelin IV' von Led Zeppelin (1971)
```

Mit `fetch_hash` funktioniert die Abfrage im Prinzip genauso, bis auf den besser lesbaren Zugriff auf die Felder:

```
while line = result.fetch_hash
  printf "'%s' von %s (%i)\n",
    line['cd_titel'], line['int_name'], line['cd_jahr']
end
```

Bei *Änderungsabfragen*, die keine Datensätze zurückliefern, brauchen Sie keine Ergebnisvariable. Stattdessen können Sie nach Durchführung der Abfrage die Eigenschaft `affected_rows` Ihres Verbindungsobjekts auslesen. Sie gibt an, wie viele Datensätze geändert wurden. Das folgende Beispiel fügt eine weitere Band zur Interpretentabelle hinzu und gibt danach an, ob es funktioniert hat:

```
conn.query
  ("INSERT INTO interpreten (int_name) VALUES ('Pink Floyd')")
if conn.affected_rows > 0
  puts "Band erfolgreich hinzugefuegt."
else
  puts "Band konnte nicht hinzugefuegt werden."
end
```

Alle MySQL-Methoden lösen Exceptions vom Typ `Mysql::Error` aus. Wenn Sie diese abfangen, können Sie nützliche Informationen über den genauen Fehler erhalten. Das folgende Beispiel versucht, Daten aus einer nicht existierenden Tabelle auszulesen:

```
begin
  result = conn.query("SELECT * FROM musiker")
rescue Mysql::Error => e
  puts "Fehlernummer:  #{e.errno}"     # Fehlercode
  puts "Fehlermeldung: #{e.error}"     # Meldungstext
  puts "SQL-Zustand:   #{e.sqlstate}"  # Fehler nach SQL-Standard
end
```

Sie erhalten folgende Ausgabe:

```
Fehlernummer:  1146
Fehlermeldung: Table 'musik.musiker' doesn't exist
SQL-Zustand:   42S02
```

Sobald Sie die MySQL-Verbindung nicht mehr benötigen, sollten Sie sie schließen:

```
conn.close
```

Beispiel: Eine ergänzbare CD-Tabelle

Als zusammenhängendes CGI-Skript, das Gebrauch von einer Datenbankverbindung macht, sehen Sie in Beispiel 6-2 eine Anwendung, die die vorhandenen Musik-CDs anzeigt und die Eingabe einer neuen CD ermöglicht. Der größte Teil des Codes wird in den Kommentaren erläutert. Abbildung 6-4 zeigt das Skript im Einsatz.

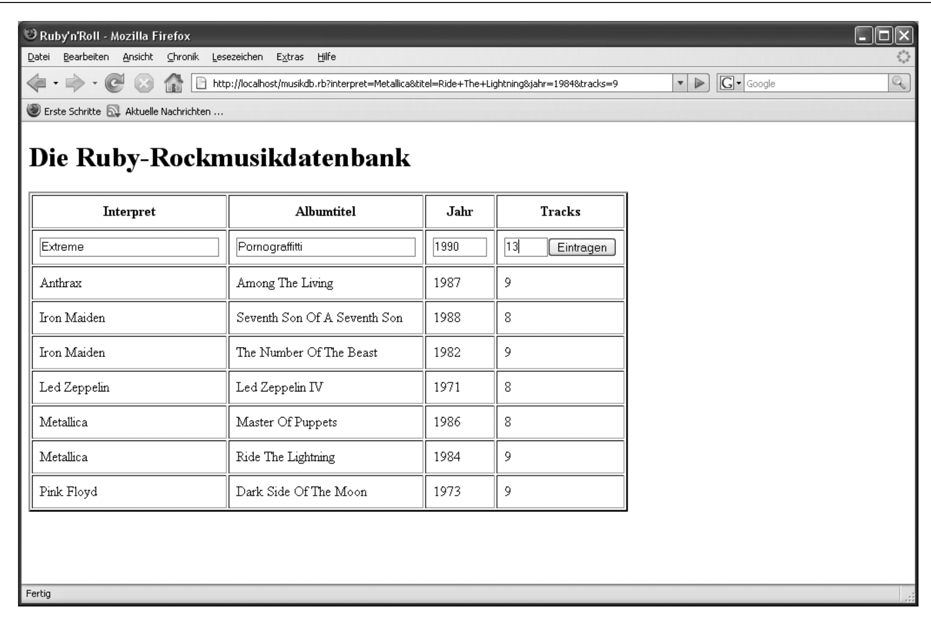

Abbildung 6-4: Datenbankbasierte CGI-Anwendung – die Rock-Tabelle

Beispiel 6-2: Die interaktive Musik-CD-Tabelle, musikdb.rb

```
 1  #!C:/ruby/bin/ruby.exe -w

 2  # Bibliotheken importieren
 3  require "cgi"
 4  require "rubygems"
 5  require "mysql"

 6  # CDs auslesen und Tabellenzeilen generieren
 7  def cd_liste(conn, cgi)
 8    # SQL-Abfrage
 9    result = conn.query(
10      "SELECT int_name, cd_titel, cd_jahr, cd_songs
11       FROM interpreten, cds
12       WHERE int_nr=cd_interpret
13       ORDER BY int_name, cd_titel"
14    )
15    ausgabe = ""
16    # Wenn Datensaetze vorhanden sind ...
17    if result.num_rows > 0
18      # ... zeilenweise hinzufuegen
19      while row = result.fetch_hash
20        ausgabe +=
21        cgi.tr {
22          cgi.td { row['int_name'] } +
23          cgi.td { row['cd_titel'] } +
24          cgi.td { row['cd_jahr'] } +
```

Beispiel 6-2: Die interaktive Musik-CD-Tabelle, musikdb.rb (Fortsetzung)

```
25          cgi.td { row['cd_songs'] }
26        }
27      end
28    end
29    # Fertigen String zurueckgeben
30    ausgabe
31  end

32  # CGI-Instanz
33  cgi = CGI.new("html4")
34  # Datenbankverbindung
35  conn = Mysql.new("localhost", "dbuser", "geheim")
36  # Standard-Datenbank
37  conn.select_db("musik")

38  # Neuer Datensatz eingegeben?
39  if cgi.has_key?("interpret") && cgi.has_key?("titel") &&
40     cgi.has_key?("jahr") && cgi.has_key?("tracks")

41    # Daten auslesen
42    interpret = cgi['interpret']
43    titel = cgi['titel']
44    jahr = cgi['jahr']
45    tracks = cgi['tracks']

46    # Den Interpreten ueberpruefen
47    result = conn.query(
48      "SELECT int_name FROM interpreten
49       WHERE int_name='#{interpret}'"
50    )
51    # Interpret neu?
52    if result.num_rows == 0
53      # Interpret einfuegen
54      conn.query(
55        "INSERT INTO interpreten (int_name)
56         VALUES ('#{interpret}')"
57      )
58    end
59    # Interpreten-ID ermitteln
60    result = conn.query(
61      "SELECT int_nr FROM interpreten
62       WHERE int_name='#{interpret}'"
63    )
64    int_nr = result.fetch_hash['int_nr']

65    # CD einfuegen
66    conn.query(
67      "INSERT INTO cds (cd_interpret, cd_titel, cd_jahr, cd_songs)
68       VALUES (#{int_nr}, '#{titel}', #{jahr}, #{tracks})"
69    )
70  end
```

Beispiel 6-2: Die interaktive Musik-CD-Tabelle, musikdb.rb (Fortsetzung)

```
 71   cgi.out {
 72     cgi.html {
 73       cgi.head {
 74         cgi.title { "Ruby'n'Roll" } +
 75         cgi.meta(
 76           'http-equiv' => "Content-type",
 77           'charset' => "iso-8859-1"
 78         )
 79       } +
 80       cgi.body {
 81         cgi.h1 { "Die Ruby-Rockmusikdatenbank" } +
 82         cgi.form("get") {
 83           cgi.table('border' => "2", 'cellpadding' => "8") {
 84             cgi.tr {
 85               cgi.th { "Interpret" } +
 86               cgi.th { "Albumtitel" } +
 87               cgi.th { "Jahr" } +
 88               cgi.th { "Tracks" }
 89             } +
 90             cgi.tr {
 91               cgi.td {
 92                 cgi.text_field(
 93                   'name' => "interpret",
 94                   'size' => "30")
 95               } +
 96               cgi.td {
 97                 cgi.text_field(
 98                   'name' => "titel",
 99                   'size' => "30")
100               } +
101               cgi.td {
102                 cgi.text_field(
103                   'name' => "jahr",
104                   'size' => "6")
105               } +
106               cgi.td {
107                 cgi.text_field(
108                   'name' => "tracks",
109                   'size' => "4") +
110                 cgi.submit("Eintragen")
111               }
112             } +
113             cd_liste(conn, cgi)
114           }
115         }
116       }
117     }
118   }

119   # Datenbankverbindung schliessen
120   conn.close
```

Beachten Sie besonders die Methode `cd_liste` (Zeile 7-31). Sie erzeugt die HTML-Tabellenzeilen mit den einzelnen CDs, falls bereits welche vorhanden sind. Datenbankverbindung und CGI-Objekt werden ihr in Form von Parametern zur Verfügung gestellt, um die Verwendung globaler Variablen zu vermeiden. Da die Ausgabe in jedem Fall ein String ist (notfalls ein leerer), kann der Methodenaufruf in Zeile 113 einfach mit + an die bisherigen Tabellenzeilen angefügt werden.

Interessant ist außerdem, dass der eingegebene Interpret nur dann in die Tabelle `interpreten` eingefügt wird, falls er noch nicht vorhanden ist. Dafür sorgen die Zeilen 47-58. Zunächst sucht eine SQL-Abfrage in der Tabelle nach einem Interpreten mit dem eingegebenen Namen. Falls dieser noch nicht vorhanden ist, findet eine `INSERT`-Abfrage statt. In Zeile 60-64 wird dann in jedem Fall die ID des Interpreten zum Eintragen in die CD-Tabelle ausgelesen.

Zwischen CGI und Rails – weitere Möglichkeiten

CGI ist, wie bereits erwähnt, die einfachste Schnittstelle für Ruby-Webanwendungen. Im nächsten Kapitel lernen Sie die derzeit perfekteste Variante kennen, Ruby on Rails. Zwischen diesen beiden Polen existieren noch zwei weitere Hilfsmittel für Ruby-Websites, die hier zumindest erwähnt werden sollen.

Wenn Sie den Apache-Webserver benutzen – was in diesem Kapitel zumindest ansatzweise erläutert wurde –, steht Ihnen das Drittanbieter-Modul `mod_ruby` zur Verfügung. Unter *http://www.modruby.org/en/* können Sie es herunterladen, und dort finden Sie auch eine Installationsanleitung. Der große Vorteil von `mod_ruby` gegenüber CGI besteht darin, dass es den Ruby-Interpreter direkt in den Webserver einbettet. Damit laufen Ihre Ruby-Webanwendungen erheblich schneller und gehen auch bei einem großen Besucheransturm nicht in die Knie. Erfreulicherweise können Sie Ihre vorhandenen Ruby-CGI-Skripten zudem einfach weiterverwenden.

Eine weitere Option ist die direkte Einbettung von Ruby in den HTML-Code mit Hilfe einer so genannten `eRuby`-Bibliothek (Embedded Ruby). Zu diesem Zweck werden die drei Pakete `eruby.cgi`, `erb` und `erubis` angeboten. `eRuby` funktioniert nur unter UNIX, während die beiden anderen plattformunabhängig sind. Funktionsweise und Lieferumfang aller drei Varianten sind dagegen praktisch identisch. Informationen und Download-Quellen finden Sie unter *http://www.eruby.info/*.

Sobald Sie eine der Einbettungs-Bibliotheken korrekt installiert und konfiguriert haben, können Sie spezielle `eRuby`-Webseiten erstellen, standardmäßig mit der Endung *.rhtml*. Diese können an beliebigen Stellen innerhalb des HTML-Codes folgende Konstrukte enthalten:

```
<% Ruby-Anweisung(en) %>
<%= Ruby-Ausdruck %>
```

→

Ersteres führt die enthaltenen Anweisungen aus; wenn Ausgabebefehle enthalten sind, wird ihr Inhalt direkt an die entsprechende Stelle der Webseite geschrieben. Das zweite Konstrukt gibt dagegen direkt an Ort und Stelle den Wert des betreffenden Ausdrucks aus. eRuby funktioniert mithin so ähnlich wie PHP, ASP oder JSP. Hier ein kleines Beispiel, das das Datum und die Uhrzeit sowie eine Schleife ausgibt:

```
<html>
<head>
<title>eRuby-Beispiel</title>
</head>
<body>
<h1>Hallo!</h1>
<p>Es ist <%= Time.new %>.</p>
<p>Hier eine Reihe von Zweierpotenzen, die &quot;Ecksteine des
Hackeruniversums&quot; (Neal Stephenson, <i>Snow Crash</i>):</p>
<%
  0.upto(16) { |i|
    puts "2<sup>#{i}</sup> = #{2**i}"

}
%>
</body>
</html>
```

Diese Art von Konstrukten wird Ihnen sehr bald wiederbegegnen, weil ein Bestandteil von Ruby on Rails – die Template-Dateien – genauso aussieht.

Zusammenfassung

Selbst wenn Sie letztlich Ruby on Rails für Ihre Webanwendungen wählen, hat Ihnen dieses Kapitel trotzdem großen Nutzen gebracht: Sie haben die hinter den Kulissen fast immer gleiche Anatomie jeder Webanwendung gesehen. Wer ein hoch integriertes Framework verwendet, vergisst nämlich beinahe, dass es immer noch Webserver und Browser sind, zu deren eigentlichen Kommunikationsfähigkeiten (HTTP) kein Framework etwas hinzufügt. So werden Sie einige merkwürdige Vorgehensweisen und Beschränkungen leichter verstehen.

Klassische Webanwendungen werden über die CGI-Schnittstelle abgewickelt: Sobald sie angefordert werden, startet der Webserver einen neuen Prozess und führt darin das CGI-Programm aus. Bei einer Skriptsprache wie Ruby wird sogar zunächst der Interpreter gestartet. Für kleine bis mittlere Websites ist das ausreichend, während Sie für größere eine schnellere Integrationstechnik wie mod_ruby einsetzen sollten.

In diesem Kapitel wurde zuerst beschrieben, wie Sie den Apache-Webserver installieren und konfigurieren können. Er bildet das Herzstück der meisten weltweiten Websites und ist daher praxiserprobt, stabil und sicher. Als Open Source-Projekt ist er zudem kostenlos und ohne Einschränkungen verwendbar.

Der Hauptteil dieses Kapitels erläuterte die CGI-Programmierung in Ruby. Zunächst wurde alles völlig zu Fuß durchgeführt, um Ihnen die Grundlagen näherzubringen. Danach haben Sie ausführlich die Arbeit mit der praktischen Bibliothek `cgi.rb` kennengelernt. Neben den CGI-Standardaufgaben – Formulardaten auslesen und HTML generieren – bietet sie die Erweiterungen Cookies und Session-Management, die ebenfalls erläutert wurden.

Zu guter Letzt wurde beschrieben, wie Sie MySQL einrichten und per Ruby darauf zugreifen können. Fast jede Webanwendung greift heutzutage nämlich auf eine Datenbank zu, um größere Inhaltsdaten in die Vorlagen zu laden oder Benutzereingaben zu speichern. MySQL ist eine der beliebtesten Open Source-Datenbanken; sie verfügt genau über die richtige Mischung aus Leistungsfähigkeit und Geschwindigkeit, um selbst große Webanwendungen erfolgreich betreiben zu können.

Nach dem soeben vorgestellten traditionellen Ansatz zur Webentwicklung lernen Sie im nächsten Kapitel das moderne Framework Ruby on Rails kennen.

In diesem Kapitel:

- Rails installieren und in Betrieb nehmen
- Die erste Rails-Anwendung
- Realistische Anwendung: Eine Online-Rock-n-Roll-Datenbank

KAPITEL 7

Ruby on Rails

Die Eisenbahn ist für mich ein Symbol des Lebens: Man sitzt ruhig und bewegt sich doch schnell vorwärts.
– Wolfgang Korruhn

Nachdem Sie Webanwendungen im vorigen Kapitel gewissermaßen zu Fuß erkundet haben, wird es nun Zeit für ein leistungsfähigeres Transportmittel. Ruby on Rails ist ein würdiger Name dafür, denn Eisenbahnen haben weltweit wesentlich zur Erschließung von unwegsamem Gelände beigetragen. Seit 1804 bewegen sie Unmengen von Personen und tonnenschwere Lasten. Genau 200 Jahre später, 2004, kam der dänische Programmierer *David Heinemeier Hansson* auf die Idee, Webanwendungen »auf die Schiene« zu setzen, und erfand das Web-Framework *Ruby on Rails*, oft einfach *Rails* genannt.

Ruby on Rails setzt das so genannte *Model-View-Controller*-Entwurfsmuster (MVC) für das Web um. Die MVC-Idee wurde ursprünglich im Zusammenhang mit Smalltalk für lokale Anwendungen mit grafischer Oberfläche geprägt. Sie teilt eine GUI-Anwendung in die folgenden drei Komponenten auf:

- Das *Model* (auf Deutsch Modell) enthält die Datenstrukturen, die der Anwendung zugrunde liegen. Damit sind nicht einfach nur ein paar Strings, Zahlen oder Arrays gemeint, sondern die softwaretechnische Abbildung der Daten für Geschäftsabläufe und andere Transaktionen (*Business Logic*). Da MVC eine Architektur für objektorientierte Sprachen ist, findet das Modell seine Ausprägung natürlich in einer Klassenstruktur. Da die konkreten Daten einer Instanz jedoch nur so lange existieren, wie ein Programm läuft, müssen sie irgendwo

dauerhaft (Fachbegriff: *persistent*) gespeichert werden – erst recht bei Webanwendungen, wo ein Skript schon wieder endet, bevor die fertige Seite an den Browser ausgeliefert wird. Ruby on Rails verwendet daher einen cleveren Trick: Die Klassenstruktur wird vollautomatisch mit passenden Datenbanktabellen verknüpft.

- Die *View* (Ansicht, Präsentation) bildet die Benutzeroberfläche der Anwendung. Sie präsentiert die Daten des Modells in frei wählbarer Form und stellt Steuerelemente zum Hinzufügen, Ändern oder Entfernen solcher Daten bereit. Rails muss in der Regel HTML-basierte Views mit variablen Daten erzeugen und verwendet deshalb die im vorigen Kapitel erwähnten `eRuby`-Vorlagedateien (*.rhtml*) als Templates, in die an geeigneter Stelle Ruby-Anweisungen und -Ausdrücke eingefügt werden können. Diese greifen auf die vom Controller bereitgestellten Methoden zurück.
- Der *Controller* (Steuerung) bildet die programmierte Verbindung zwischen Model und View: Er enthält beispielsweise Methoden zum Durchsuchen und zur Filterung der Daten aus dem Model. Umgekehrt nimmt er die Befehle aus den Steuerelementen der Benutzeroberfläche entgegen und setzt sie als Änderungen im Datenbestand des Models um. Kleinere MVC-Anwendungen kommen mitunter auch fast oder ganz ohne persistentes Model aus und wickeln die gesamte Anwendungslogik im Controller ab – das würde etwa für eine Rails-Umsetzung sämtlicher CGI-Beispiele aus dem vorigen Kapitel (bis auf das letzte, datenbankbasierte Beispiel) gelten.

In Abbildung 7-1 wird das Zusammenwirken der drei Komponenten noch einmal schematisch dargestellt. Dabei müssen Sie sich vor Augen führen, dass es sich bei Rails um eine verteilte Anwendung handelt: Model und Controller sind Server-Komponenten (die auf demselben oder auf getrennten Rechnern ausgeführt werden können), während die View zwar serverseitig erzeugt, aber letztendlich im Browser des Benutzers angezeigt wird, der umgekehrt auch die Steuerbefehle übermittelt.

Außerdem sollten Sie verstehen, dass die Interaktion zwischen Model und View im Prinzip stets indirekt über die Methoden des Controllers durchgeführt wird.

Zusätzlich zur Verwirklichung von MVC folgt Ruby on Rails zwei weiteren wichtigen Entwurfsprinzipien, die sich direkt aus der – hoffentlich auch in diesem Buch deutlich gewordenen – Ruby-Philosophie ergeben:

- *Don't Repeat Yourself* (DRY): Wenn Sie es richtig angehen, brauchen Sie bei einer Rails-Anwendung jedes Stück Ruby-Code und sogar die meisten Teile Ihrer HTML-Vorlagen nur einmal zu schreiben.
- *Convention over Configuration*: Ein klarer Satz von Konventionen bestimmt die meisten Beziehungen zwischen Model, View und Controller, so dass Sie nur wenige Informationen in Konfigurationsdateien festlegen müssen. Beispielsweise werden die Model-Klassen automatisch aus den Datenbanktabellen

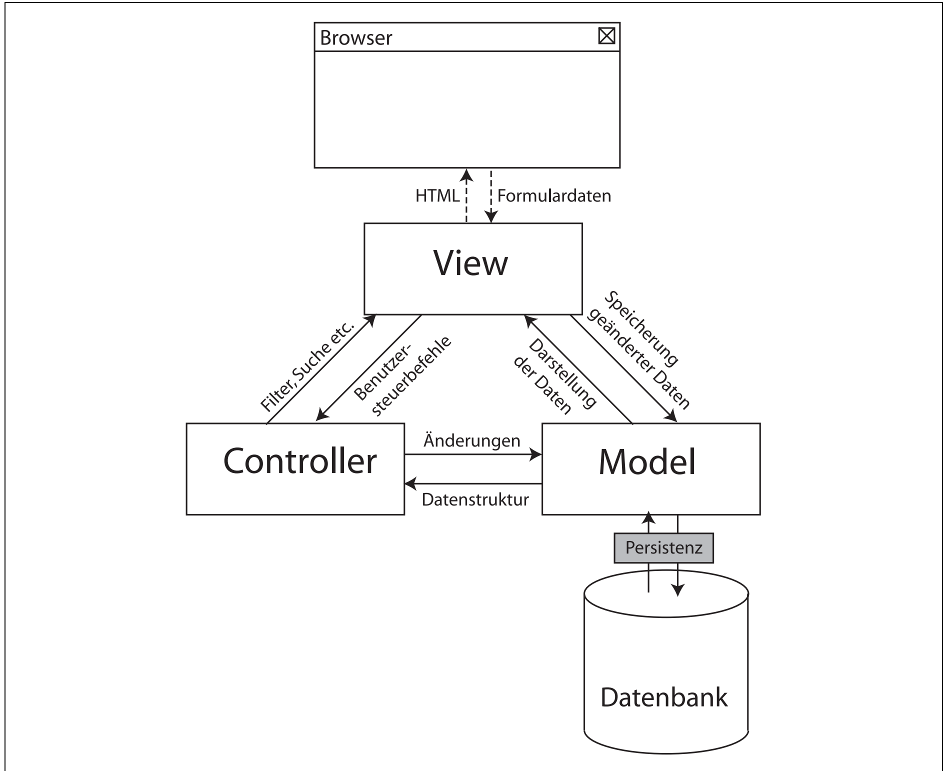

Abbildung 7-1: Das MVC-Muster mit konkreten Erweiterungen, die Ruby on Rails nutzt

generiert, und wenn an irgendeiner Stelle mehrere Instanzen einer bestimmten Klasse (Abbildung mehrerer Datensätze) gemeint sind, erzeugt Rails bei den meisten englischen Bezeichnungen automatisch den korrekten Plural.

Das Framework besteht aus den folgenden Einzelkomponenten:

- *Active Record* ist die Datenbankschnittstelle und damit der Model-Teil von MVC. Die Komponente führt ein so genanntes *objektrelationales Mapping* (ORM) durch, bildet also Ruby-Objekte auf relationale Datenbankstrukturen ab und umgekehrt. Sie brauchen also keinen SQL-Code mehr zu schreiben, sondern können Datenbankinhalte mit Ruby-Syntax auslesen und bearbeiten. Active Record arbeitet mit einer Reihe unterschiedlicher Datenbanken zusammen; Sie benötigen allerdings jeweils einen passenden Treiber oder Adapter.
- *Action Pack* besteht aus den beiden Teilen *Action Controller* und *Action View*. Action Controller liefert die Logik für die Anwendungssteuerung, während Action View die Templates für die Benutzeroberfläche bereitstellt. Es handelt sich dabei also um die Implementierung von View und Controller der MVC-Architektur.

- *Action Mailer* liefert E-Mail-Unterstützung, so dass User und/oder Administratoren automatisch per Mail über bestimmte Vorgänge in der Rails-Anwendung informiert werden können.
- *Action Web Service* ermöglicht die Rails-basierte Entwicklung von Web Services sowie entsprechenden Clients.[1]
- *Active Support* schließlich ist eine Schnittstelle für beliebige Ruby-basierte Erweiterungen des Rails-Frameworks.

Hinzu kommt die eingebaute Unterstützung für Ajax – eine JavaScript-basierte Technologie, die das Ersetzen beliebiger Einzelteile einer Webseite durch Serverdaten ohne Warten und ohne Neuladen der gesamten Seite ermöglicht. Einführungen in Action Mailer, Action Web Service und die Ajax-Komponenten sind in Vorbereitung und werden demnächst auf der Website zum Buch erscheinen.

Dieses Kapitel zeigt Schritt für Schritt, wie Sie mit Rails eine einigermaßen realistische Webanwendung erstellen können. Auf Action Mailer, Action Web Service, Active Support sowie das Deployment (die Veröffentlichung auf einem Produktions-Webserver) kann dabei aus Platzgründen leider nicht eingegangen werden. Wenn Sie nach dem Durcharbeiten dieses Kapitels auf den Geschmack gekommen sind, empfehlen wir Ihnen das Buch *Praxiswissen Ruby on Rails* von Denny Carl, das ebenfalls in der vorliegenden Reihe bei O'Reilly erschienen ist.

Rails installieren und in Betrieb nehmen

Die meisten Installationsanleitungen für Ruby on Rails beschreiben zuerst, wie Ruby installiert wird. Wenn Sie in diesem Buch bis hierher gekommen sind, dürften Sie dies allmählich hinter sich haben ;-). Falls Sie mit diesem Kapitel angefangen haben, um sich zuerst Rails anzuschauen, blättern Sie kurz zurück zu Kapitel 1 und befolgen Sie die dortige Installationsanleitung.

Als Nächstes brauchen Sie den Erweiterungsmanager `rubygems`. Falls er nicht vorhanden ist, finden Sie im vorigen Kapitel Informationen zu seiner Installation. Nun können Sie folgende Befehle eingeben, um Ruby on Rails zu installieren:

```
> gem update
> gem install rails --include-dependencies
```

Warten Sie eine Weile und verfolgen Sie dabei die verschiedenen Meldungen. Schon steht Rails zur Verfügung.

Um vollwertige MVC-Anwendungen zu erstellen, benötigen Sie zuletzt eine Datenbank sowie die entsprechenden Klassen, damit Ruby mit ihr arbeiten kann. Ruby

1 Ein Web Service ist eine Komponente, die mit Hilfe standardisierter HTTP-`POST`-Anfragen (meist in den XML-Formaten SOAP oder XML-RPC) über das Web genutzt werden kann. Das ermöglicht die plattformneutrale Zusammenarbeit verschiedener Softwarekomponenten.

on Rails kann mit vielen verschiedenen Datenbanken zusammenarbeiten. Die Empfehlung und Voreinstellung ist MySQL. Wenn Sie Kapitel 6 durchgearbeitet haben, ist dieser Datenbankserver bereits installiert, und auch die Ruby-Anbindung müsste funktionieren. Falls Sie diese Schritte noch nicht erledigt haben, lesen Sie im vorigen Kapitel nach. Anleitungen zur Verwendung anderer Datenbanksysteme gibt es dagegen auf der Website von Ruby on Rails unter *http://www.rubyonrails.org/*.

Nachdem alle Komponenten installiert sind, können Sie sofort Ihre erste Rails-Anwendung einrichten. Das Grundgerüst wird wie folgt erzeugt:

1. Wechseln Sie in ein Verzeichnis Ihrer Wahl oder erstellen Sie ein neues, das Ihre Rails-Anwendungen enthalten soll, und wechseln Sie anschließend in dieses Verzeichnis. Hier ein Windows-Beispiel:

   ```
   > C:
   > cd \
   > md railsapps
   > cd railsapps
   ```

   Unter UNIX könnten Sie hingegen folgendes Verzeichnis (als `root`) erzeugen:

   ```
   # cd /var
   # mkdir railsapps
   # cd railsapps
   ```

   Damit Sie die Rails-Entwicklungsarbeit nicht als `root` durchführen müssen, sollten Sie den Besitz an dem neuen Verzeichnis und seinen Unterverzeichnissen auf einem UNIX-System an Ihre Alltags-User-ID übertragen. Zum Beispiel:

   ```
   # chown -R sascha /var/railsapps
   ```

   Danach sollten Sie auch unter der entsprechenden User-ID und nicht als `root` weiterarbeiten. Ansonsten müssen Sie später erneut `chown` ausführen.

2. Nun kann das Skelett einer Rails-Anwendung erstellt werden. Rufen Sie dazu das Konsolenkommando `rails` mit dem Namen der gewünschten Anwendung auf. Das folgende Beispiel erzeugt eine Anwendung namens *helloworld*:

   ```
   > rails helloworld
   ```

   Dadurch entsteht ein neues Verzeichnis mit dem gewünschten Namen. Anschließend wird eine Liste der darunter erzeugten Verzeichnisstruktur angezeigt, in die zahlreiche vorgefertigte Skripten und Konfigurationsdateien kopiert werden. Die Bedeutung der diversen Verzeichnisse wird weiter unten erläutert.

3. Die neue Anwendung enthält zwar noch keinerlei spezifischen Code, ist aber prinzipiell schon bereit für einen ersten Test. Starten Sie dazu den eingebauten Ruby-Test-Webserver *WEBrick* im Unterverzeichnis *script*:

```
> ruby helloworld/script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2007-01-02 23:38:35] INFO  WEBrick 1.3.1
[2007-01-02 23:38:35] INFO  ruby 1.8.5 (2006-08-25) [i386-mswin32]
[2007-01-02 23:38:35] INFO  WEBrick::HTTPServer#start: pid=5404 port=3000
```

Öffnen Sie nun einen Browser und geben Sie die URL *http://localhost:3000/* ein.[2] Daraufhin wird die in Abbildung 7-2 dargestellte Startseite angezeigt.

Nach diesem ersten Erfolgserlebnis können Sie WEBrick mit Hilfe der Tastenkombination **Strg + C** beenden. Danach sollten Sie den nächsten Abschnitt lesen, um Ihre erste Rails-Anwendung um eigenen Code zu ergänzen.

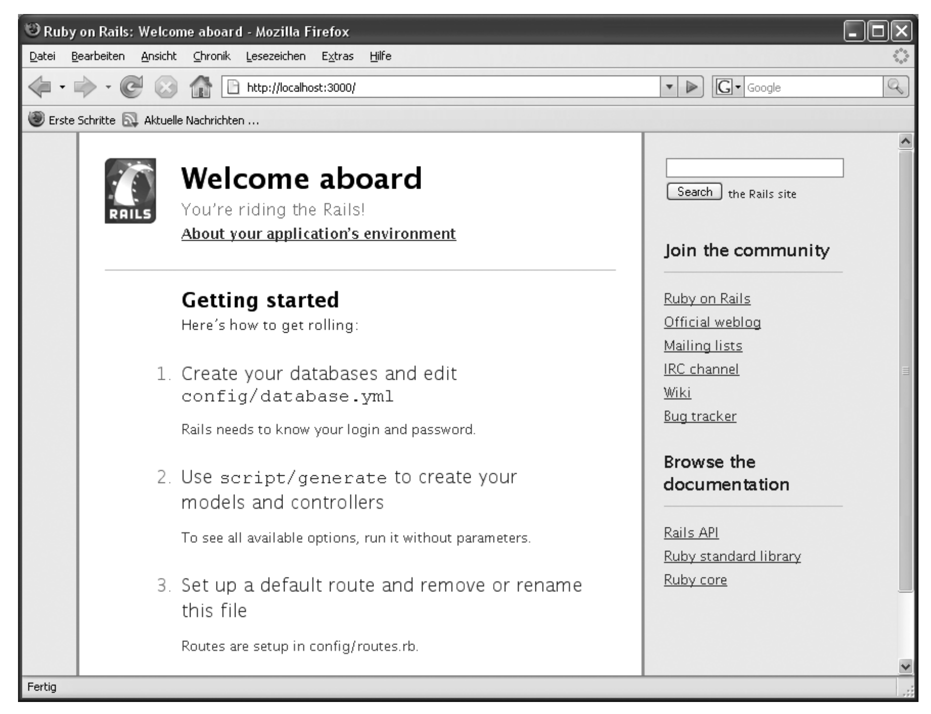

Abbildung 7-2: Die Begrüßungsseite einer neuen Rails-Anwendung im Firefox

Die erste Rails-Anwendung

Als erstes Beispiel soll eine vereinfachte Anwendung erstellt werden, die nur eine View und einen Controller, aber kein Model enthält. Der Controller stellt diverse

---

2 Wenn es sich nicht um den Standardport 80 handelt, muss das *http://* in den meisten Browsern unbedingt mit eingegeben werden.

Instanzvariablen zur Verfügung, die im `eRuby`-Template der View verwendet werden können.

Entsprechend dem gewählten Anwendungsnamen *helloworld*, sollen verschiedene Grußworte mit verschiedenen Planeten kombiniert werden. Der betreffende String soll vom Controller erstellt und dann von der View zur Ausgabe übernommen werden.

Wechseln Sie zunächst in das weiter oben erstellte Verzeichnis *helloworld*. Geben Sie dort folgende Anweisung ein, um einen Controller und eine View namens *hello* mit einer Aktion namens *index* zu erstellen:

```
> ruby script/generate controller hello index
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/hello
      exists  test/functional/
      create  app/controllers/hello_controller.rb
      create  test/functional/hello_controller_test.rb
      create  app/helpers/hello_helper.rb
      create  app/views/hello/index.rhtml
```

Der Name *index* hat wie bei normalen Websites den besonderen Vorteil, dass Sie beim späteren Aufruf im Browser keinen konkreten Dateinamen einzugeben brauchen.

Die beiden Skripten, die Sie nun um eine eigene Funktionalität erweitern können, sind der Controller *app/controllers/hello_controller.rb* und die View *app/views/hello/index.rhtml*. Die restlichen Skripten tun hier erst einmal nichts zur Sache; ihre Aufgabe wird weiter unten im Zusammenhang mit komplexeren Anwendungen erläutert.

Öffnen Sie zunächst das Skript *app/controllers/hello_controller.rb* in einem Texteditor Ihrer Wahl. Es sieht zunächst so aus:

```
class HelloController < ApplicationController

  def index

  end
end
```

Es wurde also eine leere Klassendefinition erstellt, die von der Standardklasse `ApplicationController` – der Vorlage für Rails-Controller – abgeleitet wurde. Ergänzen Sie das Skript zu folgender Fassung:

```
class HelloController < ApplicationController

  def index
    # Text lokal erzeugen
```

```
    gruesse = %w(Hallo Tag Hi Moin)
    welten = %w(Welt Mars Jupiter Saturn)
    grusszeilen = gruesse.collect { |g|
      welten.collect { |w|
        "#{g}, #{w}!"
      }
    }

    # In der View verfuegbare Instanzvariablen
    @grusstext = grusszeilen.join("<br />")
    @zeit = Time.now
  end
end
```

Die verwendeten Anweisungen brauchen hier wohl nicht weiter erläutert zu werden. In der View stehen die beiden Instanzvariablen `@grusstext` und `@zeit` zur Verfügung.

Als Nächstes muss die View *app/views/hello/index.rhtml* bearbeitet werden. Sie besteht zunächst nur aus dem folgenden unvollständigen HTML-Code:

```
<h1>Hello#index</h1>
<p>Find me in app/views/hello/index.rhtml</p>
```

Ersetzen Sie diese beiden Zeilen durch das folgende HTML-Dokument mit eingebetteten Ruby-Ausdrücken:

```
<html>
<head>
<title>Die Hallo-Seite</title>
<meta http-equiv="Content-type" content="text/html; charset="iso-8859-1" />
</head>
<body>
<h1>Planeten begr&uuml;&szlig;en</h1>
<p>Seite erzeugt um <%= @zeit %>.</p>
<p><%= @grusstext %></p>
</body>
</html>
```

Wie Sie sehen, werden die beiden Instanzvariablen aus dem Controller mittels `<%= ... %>` in den HTML-Code eingebunden.

Damit ist Ihre erste kleine Rails-Anwendung komplett. Starten Sie sie wie gehabt mit Hilfe der Anweisung:

```
> ruby script/server
```

Öffnen Sie danach einen Browser. Die neue View steht unter *http://localhost:3000/hello* zur Verfügung. In Abbildung 7-3 sehen Sie die Ausgabe des Beispiels. Da es keine Interaktivität enthält, können Sie den Server an dieser Stelle wieder beenden.

Abbildung 7-3: Die Ausgabe des ersten einfachen Rails-Beispiels im Browser Firefox

Realistische Anwendung: Eine Online-Rock-n-Roll-Datenbank

Seinen vollständigen Nutzen beginnt Rails erst dann langsam zu entfalten, wenn Sie neben der View und dem Controller auch ein Model erstellen. Deshalb wird in diesem Abschnitt die erweiterte Rails-Version des datenbankbasierten CGI-Beispiels aus dem vorigen Kapitel erstellt: eine Website, die das Betrachten, Hinzufügen, Ändern und Löschen verschiedener Rockbands und ihrer Alben ermöglicht.

Das Grundgerüst erzeugen

Der erste Schritt besteht wie beim ersten Beispiel wieder darin, das Skelett Ihrer Anwendung zu erzeugen. Wechseln Sie also erneut in Ihr Hauptverzeichnis für Webanwendungen und geben Sie Folgendes ein:

```
> rails rock_n_roll
```

Wieder wird die automatisch erzeugte Verzeichnisstruktur angezeigt. Diesmal sollten Sie sich diesen Baum näher ansehen. Die komplette Struktur sieht folgendermaßen aus:

```
app
  controllers
  helpers
  models
  views
    layouts
components
config
  environments
db
doc
lib
  tasks
log
public
  images
  javascripts
  stylesheets
script
  performance
  process
test
  fixtures
  functional
  integration
  mocks
    development
    test
  unit
tmp
  cache
  sessions
  sockets
vendor
  plugins
```

Die wichtigsten Verzeichnisse haben folgende Bedeutung:

- *app* enthält die Komponenten der eigentlichen Anwendung, unterteilt in die drei bekannten MVC-Aspekte *models*, *views* und *controllers* sowie das zusätzliche Unterverzeichnis *helpers* für gemeinsame Hilfsklassen.
- *config* enthält Konfigurationsdateien – auch wenn das Prinzip »Convention over Configuration« gilt, geht es nun einmal nicht ganz ohne. In den meisten Fällen brauchen Sie aber nur die Datei *database.yml* zu editieren, die die Namen und Zugangsdaten der verwendeten Datenbanken enthält. Das wird im nächsten Abschnitt genauer beschrieben.
- *script* enthält einige vorgefertigte Hilfsskripten, die nicht innerhalb Ihrer Rails-Anwendung ausgeführt werden, sondern Ihnen bei der Entwicklung helfen. Das Skript *generate* erzeugt beispielsweise die Grundgerüste Ihrer MVC-Komponenten, und *server* ist der WEBrick-Server, der die Anwendung zu Testzwecken ausführt.

- *public* enthält eine CGI-Konfiguration. Mit ihrer Hilfe kann die Ruby on Rails-Anwendung praktisch von jedem CGI-fähigen Webserver bereitgestellt werden. Weiter unten wird natürlich die entsprechende Konfiguration von Apache erläutert.
- *test* stellt eine Umgebung für so genannte *Unit-Tests* bereit, ein beliebtes Mittel zum automatisierten Testen objektorientierter Programme. Leider würde es den Umfang und die inhaltliche Ausrichtung dieses Buches sprengen, näher darauf einzugehen – halten Sie sich im Bedarfsfall an die Quellen aus Anhang B.

Datenbanken erzeugen und an die Rails-Anwendung binden

Die zweite Aufgabe besteht darin, das Model zu erstellen. Sie umfasst vier Einzelschritte:

- Erzeugen der Datenbanken und Tabellen
- Eintragen der Datenbanknamen und -zugriffsparameter in die Konfigurationsdatei *config/database.yml*
- Automatisches Generieren der zugehörigen Model-Klassen
- Anpassen der erzeugten Klassen durch Hinzufügen der Tabellenrelationen

Starten Sie für den ersten Schritt den Konsolen-Client `mysql`:

```
> mysql -u root -p
[Passwort]
```

Eine Rails-Anwendung benötigt bis zu drei Datenbanken, die standardmäßig mit *Anwendungsname_development*, *Anwendungsname_test* und *Anwendungsname_production* bezeichnet werden. Sie sind für die Entwicklung, die Unit-Tests beziehungsweise die Produktion (die tatsächliche Veröffentlichung der Site) zuständig. Zu Beginn benötigen Sie nur die Entwicklungs- und gegebenenfalls die Testdatenbank. Es schadet aber nichts, gleich alle drei zu erstellen. Da die neue Anwendung *rock_n_roll* heißt, können Sie dazu die folgenden drei Zeilen eingeben:

```
mysql> create database rock_n_roll_development;
mysql> create database rock_n_roll_test;
mysql> create database rock_n_roll_production;
```

Als Nächstes sollten Sie einen separaten MySQL-User erstellen, der Zugriff auf diese Datenbanken hat. Benutzername und Passwort dieses Benutzerkontos müssen Sie in die Konfiguration der Anwendung schreiben. In der Praxis verwenden Sie aus Sicherheitsgründen besser einen separaten User für die Produktionsdatenbank, aber zum Ausprobieren soll erst einmal ein gemeinsames Konto genügen. Die folgenden Zeilen legen einen User namens *rock_user* mit dem Passwort *rockonrails* an, der vollen Zugriff auf alle drei Datenbanken besitzt:

```
mysql> create user rock_user@localhost identified by "rockonrails";
mysql> grant all on rock_n_roll_development.* to rock_user@localhost;
mysql> grant all on rock_n_roll_test.* to rock_user@localhost;
mysql> grant all on rock_n_roll_production.* to rock_user@localhost;
```

Anschließend sollten Sie die MySQL-Benutzerdaten aktualisieren, damit diese Änderungen wirksam werden:

```
mysql> flush privileges;
```

Jetzt können die Tabellen erstellt werden, die das Model der Anwendung bilden sollen. Im Gegensatz zum Beispiel aus dem vorigen Kapitel werden hier konsequent englische Namen verwendet. Das liegt daran, dass Rails bei Beziehungen zwischen Tabellen automatisch den Plural beziehungsweise Singular englischer Standardwörter bildet (»one band has many albums« oder formaler: `band.has_many :albums`) – bis hin zu eingebauten Ausnahmen wie *person* <=> *people*. Wenn Sie deutsche (oder sehr seltene englische) Namen verwenden möchten, müssen Sie die Pluralregeln einzeln in *config/environment.rb* angeben.

Zunächst brauchen Sie nur in der Entwicklungs-Datenbank Tabellen. Wählen Sie diese als Standarddatenbank aus:

```
mysql> use rock_n_roll_development
```

Die beiden Tabellen sollen *bands* und *albums* heißen. Eine Band wird dabei durch ihren Namen und ihr Herkunftsland charakterisiert, ein Album durch Titel, Erscheinungsjahr und Band. Letzteres ist eine Relation mit einem Datensatz in *bands*. Da eine Band beliebig viele Alben produzieren kann, handelt es sich um eine so genannte 1:n-Relation. Die Rails-Konventionen gehen davon aus, dass jede Tabelle einen Primärschlüssel namens `id` besitzt und dass ein Feld mit Bezug auf eine andere Tabelle einen Namen nach dem Schema `andereTabelle_id` erhält.

Geben Sie kurz gesagt folgenden SQL-Code ein, um die Tabellen anzulegen:

```
mysql> create table bands (
    ->   id int auto_increment primary key,
    ->   name varchar(40),
    ->   country varchar(40)
    -> );

mysql> create table albums (
    ->   id int auto_increment primary key,
    ->   title varchar(40),
    ->   release_year date,
    ->   band_id int
    -> );
```

Verlassen Sie den MySQL-Client nach diesen Eingaben oder geben Sie vorher ein paar Testdaten ein, wenn Sie möchten. Schon bald werden Sie allerdings auch ein praktisches Web-Interface zur Datenpflege erhalten.

Nun ist es Zeit, die Datei *config/database.yml* in einem Texteditor zu öffnen und die Datenbank-Verbindungsparameter einzugeben. Die Datei ist im YAML-Format geschrieben. Es handelt sich dabei um eine sehr einfache Sprache zur Darstellung

verschiedener Datenstrukturen in Textform.[3] Das Format ist knapper und weniger komplex als etwa XML.

Die automatisch erstellte Version der Konfigurationsdatei sieht so aus (die Kommentare wurden entfernt):

```
development:
  adapter: mysql
  database: rock_n_roll_development
  username: root
  password:
  host: localhost

test:
  adapter: mysql
  database: rock_n_roll_test
  username: root
  password:
  host: localhost

production:
  adapter: mysql
  database: rock_n_roll_production
  username: root
  password:
  host: localhost
```

Wie Sie sehen, ist ein großer Teil der Arbeit bereits getan. Sogar der Datenbank-Adapter `mysql` ist bereits eingestellt, da er die Standardvorgabe für Rails ist. Ersetzen Sie den `username` `root` jeweils durch `rock_user` und fügen Sie das Passwort `rockonrails` hinzu. Sie erhalten folgende Zeilen:

```
development:
  adapter: mysql
  database: rock_n_roll_development
  username: rock_user
  password: rockonrails
  host: localhost

test:
  adapter: mysql
  database: rock_n_roll_test
  username: rock_user
  password: rockonrails
  host: localhost

production:
  adapter: mysql
  database: rock_n_roll_production
```

3 Ein solches Verfahren wird als *Serialisierung* oder *Marshaling* bezeichnet. Es wird eingesetzt, wenn komplexe Datenstrukturen seriell übertragen (zum Beispiel durch Netzwerke) oder in relationalen Datenbanken gespeichert werden sollen.

```
  username: rock_user
  password: rockonrails
  host: localhost
```

Nachdem Sie die geänderte Datei gespeichert haben, können die Model-Klassen erzeugt werden. Geben Sie dazu folgende Kommandos ein:

```
> ruby script/generate model Band
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/bands.rb
      create  test/unit/bands_test.rb
      create  test/fixtures/bands.yml
      create  db/migrate
      create  db/migrate/001_create_bands.rb
> ruby script/generate model Album
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/albums.rb
      create  test/unit/albums_test.rb
      create  test/fixtures/albums.yml
      exists  db/migrate
      create  db/migrate/002_create_albums.rb
```

Die erstellten Klassen sind so genannte *Stubs* (englisch für Stummel oder Stumpf). Sie enthalten also die Basisfunktionalität ohne jegliche Extras. Das Wichtigste, was Sie hinzufügen müssen, sind die Relationen zwischen den Daten.

Öffnen Sie also zunächst die Datei *app/models/bands.rb*. Sie besteht nur aus den folgenden beiden Zeilen:

```
class Bands < ActiveRecord::Base
end
```

Die Klasse `Bands`, die mit der gleichnamigen (kleingeschriebenen) Datenbanktabelle korrespondiert, wird also von `ActiveRecord::Base` abgeleitet, wo alles Wesentliche für die Umsetzung der Datensätze in Ruby-Instanzen bereits existiert. Ergänzen Sie die Klassendefinition wie folgt, um klarzustellen, dass eine Band mehrere Alben veröffentlichen kann:

```
class Bands < ActiveRecord::Base
  has_many :albums
end
```

Auch *app/models/albums.rb* müssen Sie entsprechend erweitern. Hier lautet die passende Beziehung `belongs_to`, weil ein Album stets einer bestimmten Band zugeordnet ist (der Sonderfall Sampler wird hier nicht betrachtet). Die ursprüngliche Fassung

```
class Albums < ActiveRecord::Base
end
```

wird somit zu

```
class Albums < ActiveRecord::Base
  belongs_to :band
end
```

Beachten Sie den Singular bei `belongs_to :band`.

Was an dieser Stelle nicht behandelt wird, ist die Möglichkeit, Validierungsanforderungen hinzuzufügen. Sie überprüfen beim Einfügen oder Ändern von Daten automatisch, ob diese beispielsweise existieren oder ein bestimmtes Format besitzen.

Automatische Skripten zur Datenbearbeitung

Eine besonders beeindruckende Fähigkeit von Ruby on Rails ist das *Scaffolding* (englisch für Gerüstbau). Mit einer einzigen Kommandozeilenanweisung erstellt es Ihnen eine Webschnittstelle zum Erstellen, Lesen, Ändern und Löschen von Datensätzen einer Tabelle (kurz *CRUD* für Create, Read, Update, Delete).

Im vorliegenden Beispiel können Sie das Scaffolding nur für die Tabelle *bands* benutzen, weil das Verfahren noch nicht »intelligent« genug ist, Tabellenrelationen automatisch zu berücksichtigen. Geben Sie also Folgendes ein:

```
> ruby script/generate scaffold band
```

Starten Sie nun den WEBrick-Server der Anwendung und geben Sie die URL *http://localhost:3000/bands* in Ihren Browser ein. Zunächst sehen Sie eine leere Liste. Klicken Sie auf *New band* und geben Sie eine Band Ihrer Wahl sowie deren Herkunftsland ein und klicken Sie auf *Create* (siehe Abbildung 7-4). Wiederholen Sie das ein paarmal, bis Sie eine Liste wie in Abbildung 7-5 erhalten.

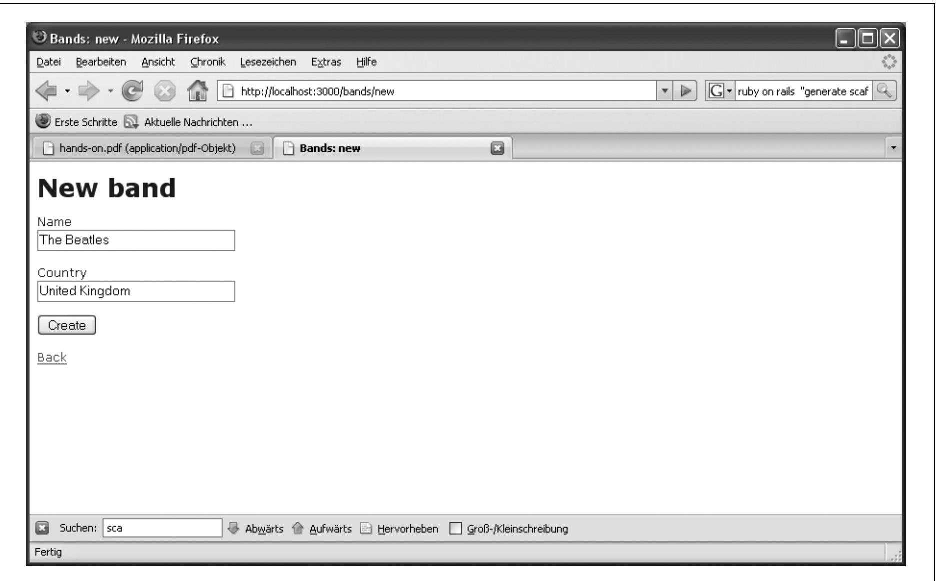

Abbildung 7-4: Eingabe einer neuen Band in die vollautomatisch erzeugte Eingabemaske

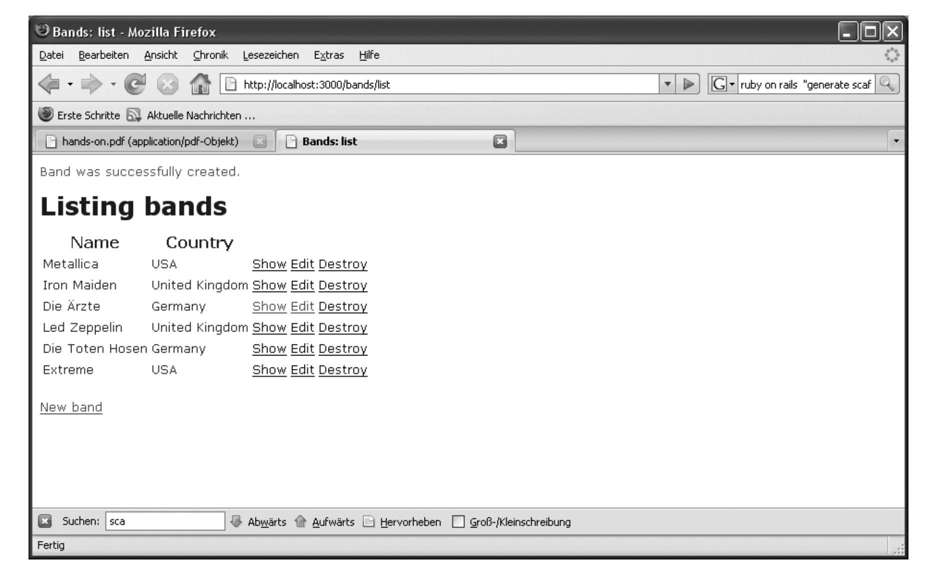

Abbildung 7-5: Die ebenfalls vollautomatisch erzeugte Bandliste

»irb on Rails« und ein Active Record-Tutorial

Im Verzeichnis *script* Ihrer Rails-Anwendung finden Sie noch zwei weitere praktische Helfer. *scripts/runner* führt einzelne Ruby-Anweisungen in der Rails-Umgebung aus, genau wie `ruby -e "Anweisung"` auf der Konsole. *scripts/console* ruft dagegen `irb --simple-prompt` auf und stellt darin sämtliche Klassen der Ruby on Rails-Anwendung zur Verfügung.

Sie sollten beides ausprobieren und nutzen, indem Sie ein wenig Active Record-Code zu Fuß eingeben. Dabei werden Sie gleichzeitig sehen, wie einfach und Ruby-gemäß der Zugriff auf die Datenstrukturen plötzlich funktioniert – vor allem verglichen mit den relativ umständlichen direkten SQL-Abfragen, die Sie im vorigen Kapitel kennengelernt haben.

Probieren Sie zuerst *scripts/runner* aus. Das folgende Beispiel liest den Namen und das Land der ersten Band aus der Tabelle *bands*:

```
> ruby script/runner "b1 = Band.find(1); puts b1.name +
' (' + b1.country + ')'"
Metallica (USA)
```

Ja, es ist wirklich *so* einfach! Die Klasse `Band` wurde automatisch aus der Tabelle *bands* generiert. Ihre Klassenmethode `find` sucht nach einer konkreten Instanz (oder einem Datensatz) mit der angegebenen ID, die in der Datenbanktabelle bekanntlich `id` heißen muss. Die verschiedenen Felder schließlich stehen einfach als Methoden der gefundenen Instanz zur Verfügung.

Starten Sie für den nächsten Versuch *scripts/console*:

```
> ruby script/console
Loading development environment.
```

Geben Sie folgenden Code ein, um eine Liste aller Bands mit ihren IDs, Namen und Ländern zu erhalten:

```
>> Band.find(:all).each { |b|
?>   puts "#{b.id}. #{b.name} (#{b.country})"
>> }
1. Metallica (USA)
2. Iron Maiden (United Kingdom)
3. Die Aerzte (Germany)
4. Led Zeppelin (United Kingdom)
5. Die Toten Hosen (Germany)
6. Extreme (USA)
```

Das Symbol `:all` kann statt einer konkreten ID verwendet werden, um alle Datensätze auszulesen.

Zum Schluss sollten Sie einige Alben anlegen. Auch für die Neuerstellung von Datensätzen gibt es eine einfache Vorgehensweise:

```
Klasse.create(:attr1 => Wert, :attr2 => Wert, ...)
```

Das Einfügen von Datensätzen ist mit anderen Worten nur noch ein Ruby-Methodenaufruf. Mit der nummerierten Bandliste ist nun auch bekannt, welche Band zu welchem Album eingegeben werden muss. Legen Sie also los. Zum Beispiel:

```
>> Album.create(:title => "Die Bestie in Menschengestalt",
?> :release_year => "1993", :band_id => 3)
```

Wenn Sie einen Fehler machen, können Sie zum Beispiel die Methode `update_attribute` eines Datensatz-Objekts aufrufen, um ihn zu beheben. Die benötigten Attribute sind `Feldname` und `neuer Wert`. Angenommen, Sie legen das Metallica-Album »Metallica« (das so genannte »Black Album«) an und geben als Jahr 1990 statt 1991 ein:

```
>> Album.create(:title => "Metallica",
?> :release_year => "1990", :band_id => 1)
```

Um die Daten eines konkreten Albums zu ändern, müssen Sie das entsprechende Album zunächst einmal finden. Statt einer bestimmten ID können Sie bei `find` auch die Kombination `:all` und `:conditions => "Bedingungen ..."` angeben. Die Bedingungen sind praktisch derjenige Teil einer SQL-Abfrage, den Sie in einem `SELECT` hinter das `WHERE` schreiben würden. Zum Beispiel:

```
>> Album.find(:all, :conditions => "title like 'Met%'").each { |a|
?>   puts a.title
>> }
Metallica
```

Die hier verwendete SQL-Klausel `LIKE` stellt eine einfache Suchmuster-Syntax zur Verfügung: Ein Prozentzeichen (%) steht für beliebig viele beliebige Zeichen und ein Unterstrich (_) für genau ein beliebiges Zeichen. Verwenden Sie dies, um das Jahr zu korrigieren:

```
>> Album.find(:all, :conditions => "title like 'Met%'").each { |a|
?>   a.update_attribute(:release_year, "1991")
>> }
=> [#<Album:0x381cccc @attributes={"title"=>"Metallica",
   "release_year"=>"1991", "id"=>"5", "band_id"=>"1"}>]
```

Geben Sie einige weitere Alben Ihrer Wahl ein, und verwenden Sie zum Schluss diesen Code, um sich eine Liste der Alben mit dem Namen ihrer jeweiligen Band anzeigen zu lassen:

```
>> Album.find(:all).each { |a|
?>   b = Band.find(a.band_id)
>>   puts "'#{a.title}' von #{b.name} (#{a.release_year})"
>> }
'Master of Puppets' von Metallica (1986)
'Ride The Lightning' von Metallica (1984)
'The Number Of The Beast' von Iron Maiden (1982)
'Die Bestie in Menschengestalt' von Die Aerzte (1993)
'Metallica' von Metallica (1991)
```

Controller und View erstellen

Mit dem Active Record-Wissen aus dem vorigen Abschnitt ist es nun überhaupt kein Problem mehr, den Controller und die View der Rock-and-Roll-Anwendung zu erstellen. Die Bands und ihre Alben sollen alternativ als verschachtelte Liste und als Tabelle angezeigt werden.

Erstellen Sie als Erstes das Grundgerüst des Album-Controllers:

```
> ruby script/generate controller Album
     exists  app/controllers/
     exists  app/helpers/
     create  app/views/album
     exists  test/functional/
     create  app/controllers/album_controller.rb
     create  test/functional/album_controller_test.rb
     create  app/helpers/album_helper.rb
```

Zuerst sollen die beiden möglichen Ansichten erstellt werden. Sie wissen bereits, wie die View Daten aus dem Controller erhalten kann – sie kann sämtliche Instanzvariablen auslesen. Eine praktische Vorgehensweise besteht also beispielsweise darin, die Ansichtsdaten komplett im Controller zu erzeugen, in Instanzvariablen zu speichern und dann in der View anzuzeigen.

Die beiden Teilansichten erstellen Sie am besten als Controller-Actions, die letztlich separate View-Templates erzeugen:

```
> ruby script/generate controller Album list table
      exists  app/controllers/
      exists  app/helpers/
      exists  app/views/album
      exists  test/functional/
overwrite app/controllers/album_controller.rb? [Ynaq]
```

Sie werden gefragt, ob die bestehende Datei überschrieben werden soll. Die Antwortmöglichkeiten sind `Y` (ja), `n` (nein), `a` (alle weiteren überschreiben) oder `q` (Erzeugung abbrechen). Im vorliegenden Fall sollten Sie `Y` wählen.

Bei späteren Erweiterungen dürfen Sie auf keinen Fall mehr mit `Y` oder gar mit a antworten, weil Sie sonst Ihren selbst erzeugten Code vernichten! Aber wenn Sie jemals eine weitere Aktion hinzufügen und die Datei *album_controllers.rb* nicht ersetzen, brauchen Sie nur die beiden Zeilen

```
def Aktionsname
end
```

selbst zu schreiben, die ansonsten automatisch erstellt würden.

Erstellen Sie nun also die Controller-Methoden, zuerst `list` für die Listenansicht. Wenn Sie die Datei *app/controllers/album_controller.rb* öffnen, finden Sie dort die Klassendefinition mit den Methoden-Stubs `list` und `table`:

```
class AlbumController < ApplicationController

  def list
  end

  def table
  end
end
```

Ergänzen Sie `list` zu folgendem Gesamtcode – die Active Record-Methodenaufrufe sollten Ihnen nach den obigen Erläuterungen bekannt sein:

```
  def list
    # Listeninhalt privat erstellen
    bandlist = "<ul>"
    # Ueber alle Bands iterieren
    Band.find(:all).each { |b|
      bandlist += "<li><b>#{b.name} (#{b.country})</b></li>"
      # Ueber alle Alben der Band iterieren
      bandlist += "<ul>"
      Album.find(:all, :conditions => "band_id=#{b.id}").each { |a|
        bandlist += "<li>#{a.title} (#{a.release_year})</li>"
      }
      bandlist += "</ul>"
    }
    bandlist += "</ul>"
```

```
    # Liste als Instanzvariable veroeffentlichen
    @bandlist = bandlist
  end
```

Nun müssen Sie nur noch die entsprechende View – *app/views/list.rhtml* – ergänzen. Sie hat momentan folgenden Inhalt:

```
<h1>Album#list</h1>
<p>Find me in app/views/album/list.rhtml</p>
```

Erstellen Sie stattdessen den folgenden HTML/eRuby-Code:

```
<html>
<head>
<title>Rock'n'Rails -- Bands und ihre Alben</title>
</head>
<body>
<h1>Alle Bands und ihre Alben</h1>
<p><%= @bandlist %></p>
</body>
</html>
```

Wie Sie sehen, brauchen Sie nicht mehr eRuby-Code als die Ausgabe der Instanzvariablen `@bandlist`. Starten Sie nun den WEBrick-Server und geben Sie die URL *http://localhost:3000/album/list* ein. Die Ausgabe sehen Sie in Abbildung 7-6.

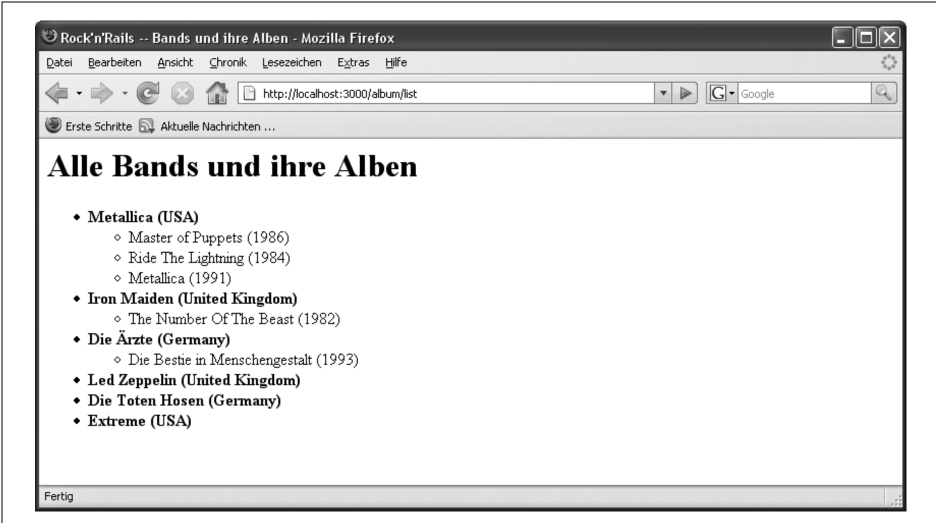

Abbildung 7-6: Die Ausgabe der Action-View album/list

Stoppen Sie WEBrick danach wieder, denn nun kommt die Tabellenansicht an die Reihe. Ergänzen Sie den Stub der Methode `table` zu:

```
  def table
    # Tabelleninhalt
    tab = "<tr>" +
        "<th>Album</th>" +
```

```
            "<th>Jahr</th>" +
            "<th>Band (Land)</th>"
        tab += "</tr>"
        Album.find(:all, :order_by => "title").each { |a|
          tab += "<tr>"
          tab += "<td>#{a.title}</td>"
          tab += "<td>#{a.release_year}</td>"
          b = Band.find(a.band_id)
          tab += "<td>#{b.name} (#{b.country})"
          tab += "</tr>"
        }

        # Veroeffentlichung
        @albumtable = tab
      end
```

Kümmern Sie sich danach um die View *table.rhtml*:

```
<html>
<head>
<title>Rails'n'Roll -- Alben-Tabelle</title>
</head>
<body>
<h1>Tabelle aller Alben</h1>
<table border="2" cellpadding="8">
<%= @albumtable %>
</table>
</body>
</html>
```

Wenn Sie WEBrick wieder starten und *http://localhost:3000/album/table* aufrufen, sollten Sie eine Ansicht wie in Abbildung 7-7 erhalten.

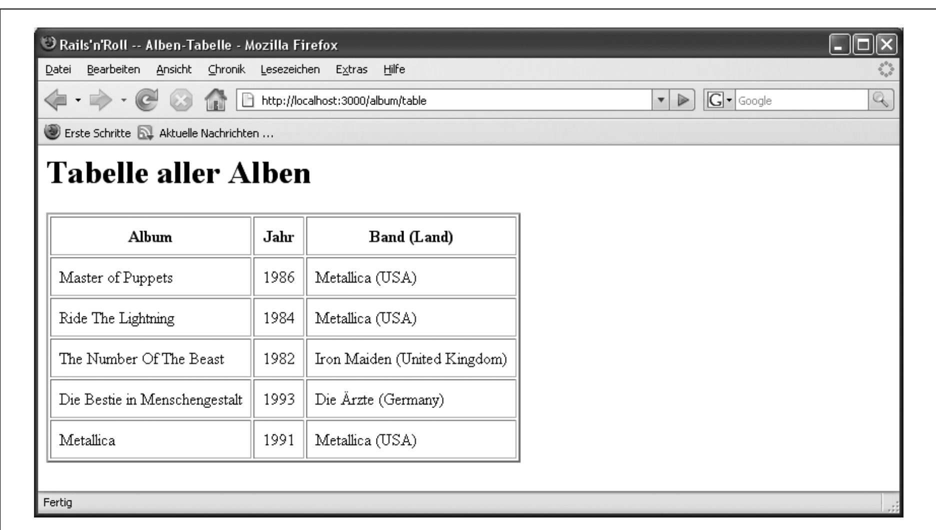

Abbildung 7-7: Die Ausgabe der Action-View album/table

Zu guter Letzt sollten Sie die beiden Ansichten der Bequemlichkeit halber per Hyperlink miteinander verknüpfen.

Bei größeren Projekten ist es ratsam, die Logik von View und Controller strikt zu trennen. Der Controller sollte dann keine HTML-Schnipsel liefern, sondern nur die Rohdaten – beispielsweise als Array. Iteration und Ausgabe kommen dann in die View.

Nichts leichter als das. Ergänzen Sie *list.rhtml* unmittelbar über `</body>` um die folgende Zeile:

```
<a href="table">Zur Tabellenansicht</a>
```

Umgekehrt können Sie an derselben Stelle in *table.rhtml* diese Ergänzung vornehmen:

```
<a href="list">Zur Listenansicht</a>
```

Was bleibt zu tun?

Der Grundstein einer ersten Rails-Webanwendung ist gelegt, und Sie haben einige der wichtigsten Komponenten des Frameworks gesehen. In der Praxis müssten nun folgende Schritte folgen:

1. Sie brauchen Controller-Actions zum Hinzufügen, Ändern und Löschen von Alben. Mit Scaffolding ist es nicht getan, weil es die Relationen nicht automatisch berücksichtigen kann. Wenn Sie aber Ihre Active Record-Kenntnisse aus diesem Kapitel und den Scaffold-Code in *app/controllers/bands_controller.rb* kombinieren, müssten Sie leicht ans Ziel gelangen.
2. Nach der Entwicklungsphase (die bereits bei mittleren Projekten immer auch Unit-Tests umfassen sollte) erfolgt der Wechsel in die Produktionsumgebung. Dazu müssen Sie die Datenbank `Anwendung_production` um dieselben Tabellenstrukturen ergänzen wie die Entwicklungsdatenbank. Anschließend müssen Sie Rails über die Umgebungsvariable `RAILS_ENV` mitteilen, dass die Produktionsumgebung verwendet werden soll. Das geht auf der Konsole wie folgt:

   ```
   > set RAILS_ENV=production              (Windows)
   $ export RAILS_ENV=production           (Linux)
   ```

   In der Regel sollten Sie diese Variablendefinition allerdings in die Konfiguration Ihres Produktions-Webservers einbauen. Apache stellt dafür beispielsweise die Direktive `SetEnv` zur Verfügung:

   ```
   SetEnv RAILS_ENV production
   ```

3. Zum Schluss erfolgt das *Deployment*, also die Veröffentlichung auf einem produktionstauglichen Webserver (WEBrick ist langsam, nicht besonders sicher und kann jeweils nur eine einzige Rails-Anwendung hosten). Auf der Website zum Buch finden Sie verschiedene Apache-Beispielkonfigurationen.

Zusammenfassung

Dies war nur ein kleiner Vorgeschmack des Leistungsumfangs von Ruby on Rails. Aber einige wichtige Punkte dürften klar geworden sein:

- Intelligente Skripten helfen Ihnen dabei, immer wieder benötigten Code der verschiedenen Grundgerüste (Anwendung, MVC-Komponenten und so weiter) automatisch zu erstellen.
- Sie brauchen keine SQL-Abfragen mehr in Ihre Webanwendungen einzubetten. Active Record kümmert sich vollautomatisch um die Umsetzung der Datenbanktabellen in Ruby-Klassen und umgekehrt. Zahlreiche praktische Methoden – von denen Sie die allerwichtigsten kennengelernt haben – ersetzen datenbankabhängiges SQL durch neutrale, anwendungsnahe Ruby-Anweisungen.
- Die Trennung in Model, View und Controller ermöglicht Ihnen eine saubere Aufgabentrennung. Sie können die zu präsentierenden Daten, die Steuerlogik und die Ansicht Ihrer Site völlig unabhängig voneinander ändern. Im Grunde macht der Einsatz von Rails Content-Management überflüssig beziehungsweise beinhaltet es bereits. Nichtsdestotrotz werden bereits diverse CMS angeboten, die wiederum auf dem Framework aufbauen.

Das Konzept von Ruby on Rails ist so erfolgreich, dass Entwickler inzwischen versuchen, ähnliche Frameworks für andere Programmiersprachen zu entwerfen, etwa PHP on Trax. Allerdings sind die meisten der zugrunde liegenden Sprachen nicht annähernd so elegant und zugleich mächtig wie Ruby, so dass diese Pakete auch nicht an Rails heranreichen.

ANHANG A

Ruby-Kurzreferenz

Der echte Schüler lernt aus dem Bekannten das Unbekannte entwickeln und nähert sich dem Meister.

– Johann Wolfgang Goethe

Syntax

- *Anweisungen* werden durch Zeilenumbruch oder innerhalb einer Zeile durch `;` getrennt
- *Bezeichner* beginnen mit Buchstabe oder Unterstrich (_), danach weitere Buchstaben, Ziffern oder Unterstriche. Groß- und Kleinschreibung werden unterschieden.
- *Kommentare*: von `#` bis Zeilenende
- *Shebang*: Erste Zeile; gibt an, wo sich Ruby befindet.
  Z.B.: `#!/usr/bin/ruby` (Unix), `#!C:/ruby/bin/ruby.exe` (Windows)

Ausdrücke

Ein Ausdruck ist eine beliebig komplexe Verknüpfung von Literalen, Variablen und Methodenaufrufen durch Operationen.

Literale

Literale sind alle wörtlich gemeinten Einzelwerte.

Strings

- "In doppelten Anfuehrungszeichen werden Escape-Sequenzen und eingebettete Ausdruecke ausgewertet"
- 'In einfachen Anführungszeichen werden nur \' und \\ umgewandelt.'
- %Q(Wie doppelte Anführungszeichen)
- %q(Wie einfache Anführungszeichen)
- HIER-Dokument:
  ```
  <<ENDMARKE
  Mehrzeiliger
  Text
  ENDMARKE
  ```
- Escape-Sequenzen: "\n" (Zeilenvorschub, auch allgemeiner Zeilenumbruch), "\r" (Wagenrücklauf), "\t" (Tabulator), "\0nnn" (oktaler Zeichencode), "\0xnn" (hexadezimaler Zeichencode) usw.
- Eingebettete Ausdrücke werden ausgewertet und in den String geschrieben: "Beliebiger Text #{beliebig komplexer Ausdruck} Text"

Zahlen

- Dezimale Ganzzahlen (Fixnum): -100, -3, 0, 1024.
- Beliebig große Ganzzahlen (Bignum): 100000000000, 5 * 10 ** 100
- Optionales Tausendertrennzeichen _: 1_000_000 = 1000000
- Oktale Ganzzahlen: 0nnn. Z.B.: 0333 = 219
- Hexadezimale Ganzzahlen: 0xnnnn. Z.B.: 0xABC = 2748
- Fließkommazahlen (Float): -3.5678, -2.0, 0.0, 1.4567
- Wissenschaftliche Schreibweise für Fließkommazahlen: +/-n.nnnnE+/-nnn = +/-n.nnnn * 10^nnn. Z.B.: 3.21e+7, 1.9876e-9
- Zeichencode: ?Zeichen. Z.B.: ?A = 65, ?! = 31, ?? = 63

Bereiche (Ranges)

- Anfang..Ende schließt Ende ein:
  0..4 = 0, 1, 2, 3, 4
  'a'..'d' = 'a', 'b', 'c', 'd'
- Anfang...Ende schließt Ende aus:
  0...4 = 0, 1, 2, 3
  'a'...'d' = 'a', 'b', 'c'

Sonstige

- true – wahre logische Aussage
- false – false logische Aussage
- nil – leeres Element
- :abc – Symbol (eindeutiges Element ohne konkreten Wert)

Variablen

Variablen sind benannte Speicherplätze. Deklaration und Wertzuweisung:

```
var = Ausdruck
```

`var` kann danach in Ausdrücken statt eines Literals eingesetzt werden.

Allgemeines

- Variablennamen beginnen mit Kleinbuchstaben.
  Z.B.: `test`, `_test`, `_test123`
- Großbuchstaben kennzeichnen *Konstanten*, deren Wert sich nach Definition nicht mehr ändert. Z.B.: `G = 9.81`, `PI = 3.1415926`
- `$var`: globale Variablen, die im gesamten Skript gelten.
  Z.B.: `$global = "Test"`
- `@var`: Instanzvariablen – nur nach `Klasse.new` verfügbar
- `@@var`: Klassenvariablen – in Klasse und allen Instanzen verfügbar

Arrays

Listen mit beliebig vielen Elementen beliebigen Typs, gekennzeichnet durch numerischen Index

- Definition (z.B.): `arr = [Wert1, Wert2, ..., WertN]`
- `arr[0]`: erstes Element
- `arr[n]`: Element Nr. n-1
- `arr[-1]`: letztes Element

Hashes

Listen mit Schlüssel-Wert-Paaren

- Definition (z.B.): `hash = {Schluessel1 => Wert1, ...}`
- Zugriff: `hash[Schluessel]`
- Typ des Schlüssels meist String oder Symbol:
  `hash["Schluessel"]` bzw. `hash[:schluessel]`

Operatoren

Die wichtigsten in absteigender Rangfolge:

- `[]` (Menge)
- `**` (Potenz)
- `!` (logisches Nicht), `+` (Vorzeichen), `-` (Vorzeichen)
- `*` (Multiplikation), `/` (Division), `%` (Modulo – Divisionsrest)
- `>>` (Bitverschiebung links), `<<` (Bitverschiebung rechts)
- `&` (bitweise Und)
- `^` (bitweise Exklusiv-Oder), `|` (bitweise Oder)

- <= (kleiner oder gleich), < (kleiner), > (größer), >= (größer oder gleich)
- <=> (Vergleich) == (ist gleich) === (in Bereich enthalten),
  != (ist ungleich), =~ (entspricht Regexp), !~ (entspricht Regexp nicht)
- && (logisches Und)
- || (logisches Oder)
- .. (Bereich incl. Ende), ... (Bereich excl. Ende)
- ?: (Fallentscheidung: Ausdruck ? Dann-Wert : Sonst-Wert)
- = (Wertzuweisung), += (bisherigen Wert erhöhen), -= (Wert vermindern)
  *= (Wert multiplizieren), /= (Wert dividieren) usw.
- not (logisches Nicht, Textform)
- or (logisches Oder, Textform), and (logisches Und, Textform)

De Morgan-Theorem

```
!(a && b) == !a || !b
!(a || b) == !a && !b
```

Kontrollstrukturen

Ändern den linearen Programmablauf

Fallentscheidungen

Führen Code abhängig von Bedingugen aus

- Einfaches if – führt Anweisungen aus, wenn Ausdruck true ist:

```
if Ausdruck
  # Anweisungen
end
```

- if/else – führt Anweisungen aus, wenn Ausdruck true ist, ansonsten andere Anweisungen:

```
if Ausdruck
  # Dann-Anweisungen
else
  # Sonst-Anweisungen
end
```

- if/elsif/else – prüft bei else weitere Bedingungen:

```
if Ausdruck1
  # Dann-Anweisungen 1
elsif Ausdruck2
  # Dann-Anweisungen 2
  ...
else
  # Sonst-Anweisungen (wenn keine Bedingung zutrifft)
end
```

- case/when – vergleicht Variable mit verschiedenen Werten:

```
case var when
```

```
Wert1:
  # Anweisungen fuer Wert1
Wert2, Wert3:
  # Anweisungen fuer Wert2 oder Wert3
  ...
else
  # Anweisungen fuer alle anderen Werte
end
```

Schleifen

Führen Code mehrmals aus

- `while` – führt Anweisungen aus, solange Ausdruck true ist:

  ```
  while Ausdruck
    # Schleifenrumpf (Anweisungen)
  end
  ```

- Fußgesteuerte Schleife – wird mindestens einmal ausgeführt, prüft erst dann Bedingung:

  ```
  begin
    # Anweisungen
  end while Ausdruck
  ```

- Endlosschleife – muss mittels `break` verlassen werden, wenn eine Bedingung zutrifft:

  ```
  loop do
    # Anweisungen
    # Abbruch:
    break if Ausdruck
  end
  ```

- `for` – Schleife über Listenelemente:

  ```
  for var in liste
    # var nimmt nacheinander den Wert jedes Elements an
  end
  ```

Reguläre Ausdrücke

Komplexe Muster zum Durchsuchen von Strings

- Darstellung als `/.../` oder `%r(...)`
- Matching: `string =~ regexp`
  Liefert bei Treffer Anfangsposition von `regexp` in `string`, ansonsten `nil`
- Umkehrung: `string !~ regexp`
  Liefert bei Treffer `false`, ansonsten `true`
- `string.sub(regexp, ersatz)`
  Ersetzt den ersten Treffer in `string` durch `ersatz`
- `string.gsub(regexp, ersatz)`
  Ersetzt alle Treffer in `string` durch `ersatz`

Konstrukte für reguläre Ausdrücke

Konstrukt	Syntax / Beispiel	Erläuterung
Einzelzeichen	`a`	Zeichen muss vorkommen
Zeichenfolge	`hallo`	Folge muss vorkommen
Zeichengruppe	`[abc]`	Eines der Zeichen muss vorkommen
Zeichenfolge	`[a-z]`	Eines der Zeichen muss vorkommen
Zeichenkombination	`[0-9a-fA-F]`	Eines der Zeichen muss vorkommen (hier: Hexadezimalziffer)
Zeichenausschluss	`[^abc]`	Ein Zeichen außer den aufgelisteten muss vorkommen
Beliebiges Zeichen	`.`	Genau ein beliebiges Zeichen muss vorkommen
Quantifier: optional	`[0-9]?`	Element kommt vor oder nicht
Quantifier: beliebig oft	`a*`	Element kommt beliebig oft vor
Quantifier: mindestens einmal	`.+`	Element kommt einmal oder öfter vor
Quantifier: genau n-mal	`[a-z]{n}`	Element kommt genau n-mal vor
Quantifier: m- bis n-mal	`[0-9a-fA-F]{m,n}`	Element kommt mindestens n- und höchstens n-mal vor
Gruppierung	`(Text)`	Fasst Gruppe zusammen, z.B. für Quantifier oder Match-Variablen
Alternative	`(abc|xyz|...)`	Eine der aufgelisteten Alternativen muss vorkommen
Bereichsmarkierung: Anfang	`^abc`	Element muss am Zeilenanfang stehen
Bereichsmarkierung: Ende	`xyz$`	Element muss am Zeilenende stehen
Wortgrenze	`\ba, n\b`	Element muss am Wortanfang bzw. Wortende stehen
Wortinneres	`\B1, 0\B`	Element darf nicht am Wortanfang bzw. Wortende stehen
Ziffer	`\d oder [0-9]`	Zeichen muss eine Ziffer sein
Keine Ziffer	`\D oder [^0-9]`	Zeichen darf keine Ziffer sein
Wortbestandteil	`\w oder [0-9a-zA-Z_]`	Zeichen muss gültiges Bezeichner-Element sein
Nicht-Wortbestandteil	`\W oder [^0-9a-zA-Z_]`	Zeichen darf kein gültiges Bezeichner-Element sein
Whitespace	`\s`	Zeichen muss Leerzeichen, Zeilenumbruch oder Tab sein
Nicht-Whitespace	`\S`	Zeichen darf kein Whitespace sein
Modifier: Case ignorieren	`/test/i`	Groß- und Kleinschreibung wird ignoriert
Modifier: Zeilenübergreifen	`/test/m`	Ausdrücke werden auch über Zeilengrenzen hinweg getestet
Modifier: Extended Mode	`/test/x`	Regexp darf durch Whitespace formatiert werden
Match-Variablen	`$1, $2, ...`	Inhalte geklammerter Ausdrücke, von außen nach innen und von links nach rechts gezählt

Klassendefinition

- Klasse:

```
class Klassenname
  # Automatische Accessor-Methoden
```

```
  # Konstruktor ...
  # Methoden ...
end
```

- Abgeleitete Klasse:

```
class Klassenname:Elternklasse
  # Zusaetze und Unterschiede programmieren
end
```

- Automatische Accessor-Methoden:

```
attr_reader :var1, ...     # fuer Instanzvar. @var1 ..., nur lesbar
attr_writer :var1, ...     # nur aenderbar
attr_accessor :var1, ...   # les- und aenderbar
```

- Konstruktor:

```
def initialize(...)
  @instanzvar1 = ...
  # ...
end
```

- Instanzmethode:

```
def methodenname(var1[=Wert], ...)
  # ...
  # Rueckgabewert:
  Ausdruck
  # Sofortiger Ruecksprung mit Rueckgabewert:
  return Ausdruck
end
```

  Aufruf von außen: `obj = Klassenname.new; obj.methodenname`

- Klassenmethode:

```
def Klassenname.methodenname(...)
  # ...
end
```

  Aufruf von außen: `Klassenname.methodenname`

Klassenreferenz

Im Folgenden werden einige besonders wichtige eingebaute Ruby-Klassen mit Konstruktor sowie häufig genutzten Klassen- und Instanzmethoden (jeweils falls vorhanden) alphabetisch aufgelistet. Beachten Sie, dass alle eingebauten und eigenen Klassen die unter `Object` aufgelisteten Methoden besitzen (es sei denn, sie setzten sie explizit `private`).

Array

Liste beliebig vieler beliebiger Objekte mit numerischem Index

Konstruktoraufrufe

- `Array.new              # ergibt []`
- `Array.new[n]           # n Elemente mit Wert nil`
- `Array.new[n, w]        # n Elemente mit Wert w`
- Implizit: `var = [...]`

Instanzmethoden

- `arr[n]` – liefert Element Nr. n-1
- `arr.length` – Anzahl der Elemente
- `arr.push(Wert, ...)` – hängt Element(e) am Ende an
- `arr << Wert` – `push`-Alternative für Einzelelement
- `arr.pop` – entfernt letztes Element und liefert es zurück
- `arr.unshift(Wert, ...)` – hängt Element(e) am Anfang an
- `arr.shift` – entfernt erstes Element und liefert es zurück
- `arr.each { ... }` – Iterator über alle Elemente

Bignum

Beliebig große Ganzzahlen

(siehe `Integer`)

Dir

Kapselt ein Verzeichnis

Konstruktoraufruf

```
Dir.new(Pfad)
```

Instanzmethoden

- `dir.read` – nächsten Eintrag lesen und zurückliefern
- `dir.rewind` – zurück zum ersten Eintrag
- `dir.close` – schließen

Siehe auch `IO`

File

Kapselt eine geöffnete Datei

Konstruktoraufruf

```
File.new(Pfad, Modus)
```

Die wichtigsten Modi:

- `r` – lesen (Datei muss existieren, sonst Fehler)
- `w` – schreiben (Datei wird neu angelegt oder überschrieben)
- `a` – anhängen (Datei wird neu angelegt, oder es wird an ihrem Ende weitergeschrieben)
- `r+` – gemischtes Lesen und Schreiben

Klassenmethoden

- `File.exists?(Pfad)` – `true`, falls Pfad existiert, ansonsten `false`
- `File.file?(Pfad)` – `true`, falls Pfad eine reguläre Datei ist, ansonsten `false`
- `File.directory?(Pfad)` – `true`, falls Pfad ein Verzeichnis ist, ansonsten `false`
- `File.open(Pfad, Modus)` – Synonym für `File.new`
- `File.rename(alt, neu)` – Datei umbenennen
- `File.delete(Pfad)` – Datei löschen

Instanzmethoden

- `file.seek(Pos)` – Dateizeiger auf die angegebene Position setzen.
- `file.rewind` – Dateizeiger zurücksetzen
- `file.sort` – Alle Zeilen auslesen und in ein Array sortieren
- `file.close` – Datei schließen

Siehe auch `IO`

Fixnum

Ganze Zahl

(Siehe `Integer`)

Float

Fließkommazahl

Instanzmethoden

- `float.round` – auf ganze Zahl runden
- `float.ceil` – nächsthöhere Ganzzahl
- `float.floor` – Nachkommastellen abschneiden
- `float.to_i` – Synomym für `float.floor`

Hash

Liste aus Schlüssel-Wert-Paaren

Konstruktoraufrufe

- `hash = Hash.new            # leeres Hash`
- Implizit: `hash = {Schluessel1 => Wert1, ...}`

Instanzmethoden

- `hash[Schluessel]` – Zugriff auf einzelnes Element
- `hash.has_key?(Schluessel)` – `true`, wenn Schlüssel existiert, ansonsten `false`

- `hash.size` – Anzahl der Paare
- `hash.invert` – Schlüssel und Werte vertauschen (Datenverlust bei vormals gleichen Werten!)
- `hash.each { ... }` – Iterator über jeden Schlüssel und jeden Wert einzeln (wie Array-Elemente)
- `hash.each_key { ... }` – Iterator über alle Schlüssel
- `hash.each_value { ... }` – Iterator über alle Werte
- `hash.each_pair { ... }` – Iterator über alle Paare

Integer

Ganzzahl; gemeinsame Elternklasse von `Fixnum` und `Bignum`

Instanzmethoden

- `int.succ` – Nachfolger
- `int.chr` – Zeichen mit dem entsprechenden Code
- `int.times { ... }` – Iterator, der `int`-mal ausgeführt wird (0 bis `int-1`)
- `int.upto(int2) { ... }` – Iterator von `int` bis `int2`, aufsteigend
- `int.downto(int2) { ... }` – Iterator von `int` bis `int2`, absteigend
- `int.step(max, schritt) { ... }` – Iterator von `int` bis `max`, Schrittweite `schritt`
- `int.to_s(basis)` – konvertiert int in das Zahlensystem `basis` (2-36) und liefert das Ergebnis als String

IO

Allgemeine Ein- und Ausgabeklasse

Instanzmethoden

- `io.print(...)` – Text ausgeben
- `io.puts(...)` – Text zeilenweise ausgeben
- `io.printf(format, ...)` – Elemente gemäß `format` ausgeben (Übersicht über Formatplatzhalter: Kapitel 3)
- `io.getc` – ein Zeichen einlesen
- `io.gets` – eine Zeile einlesen
- `io.read` – Eingabestrom bis Dateiende (EOF) einlesen

MatchData

Treffer bei Mustervergleich mit `regexp.match` (siehe Klasse `RegExp`)

Instanzmethoden

- `match[n]` – 0 ist der gefundene Text des gesamten Treffers; 1 bis n sind geklammerte Ausdrücke (entspren `$1` bis `$n`)

- `match.begin(n)` – Position des ersten Zeichens eines Treffers oder Teilausdrucks im String
- `match.end(n)` – Position des ersten Zeichens, das nicht mehr zum Treffer gehört
- `match.offset(n)` – Array mit den Werten `begin` und `end`
- `match.pre_match` – Text vor dem Treffer
- `match.post_match` – Text hinter dem Treffer

Math

Modul für komplexere mathematische Konstanten und Berechnungen

Konstanten

- `Math::PI` – π = 3.14159265358979
- `Math::E` – e = 2.71828182845905

Klassenmethoden

- `Math.sqrt(x)` – Quadratwurzel
- `Math.log(x)` – natürlicher Logarithmus (Basis e)
- `Math.exp(x)` – e^x (Umkehrfunktion des natürlichen Logarithmus')
- `Math.log10(x)` – Zehnerlogarithmus (Umkehrung: `10 ** x`)
- `Math.sin(x)`, `Math.cos(x)`, `Math.tan(x)` – trigonometrische Funktionen (Winkel im Bogenmaß, 180° = 2π)
- `Math.asin(x)`, `Math.acos(x)`, `Math.atan(x)` – Arcus- oder Umkehrfunktionen der trigonometrischen Funktionen
- `Math.hypot(k1, k2)` – Hypotenuse der beiden gegebenen Katheten gemäß Satz des Pythagoras

Object

Basisklasse für sämtliche Ruby-Klassen

Instanzmethoden

- `obj.class` – liefert die Klasse eines Objekts
- `obj.eql?(obj2)` – `true`, wenn `obj` und `obj2` ein und dasselbe Objekt sind (gleiche `object_id`), ansonsten `false`
- `obj.instance_of?(class)` – `true`, wenn `obj` direkte Instanz von `class` ist, ansonsten `false`
- `obj.kind_of?(class)` – `true`, wenn `obj` Instanz von `class` oder irgendeines Vorfahren von `class` ist, ansonsten `false`
- `obj.object_id` – die eindeutige ID des Objekts
- `obj.respond_to?(method)` – `true`, wenn `object` direkt oder indirekt die Methode `method` besitzt, ansonsten `false`
- `obj.to_s` – String-Darstellung des Objekts

Rational

Echter Bruch

Literale

```
Rational(zaehler, nenner)
```

Werden automatisch so weit wie möglich gekürzt

Instanzmethoden

- `rational.numerator` – Zähler
- `rational.denominator` – Nenner
- `rational.to_f` – in Fließkommazahl konvertieren

Regexp

Kapselt einen regulären Ausdruck

Konstruktoraufruf

- `reg = Regexp.new(/regexp/)`
- `reg = Regexp.new("regexp-string", "modifier")`
- Implizit: `reg = /regexp/`

Klassenmethoden

- `Regexp.compile(...)` – Synonym für `Regexp.new`
- `Regexp.last_match` – Treffer des letzten Regexp-Vergleichs
- `Regexp.escape(string)` – maskiert alle Zeichen, die in regulären Ausdrücken eine besondere Bedeutung haben, durch \

Instanzmethoden

- `reg.match(string)` – liefert `MatchData`-Instanz
- `reg =~ string` – liefert Trefferposition in `string` oder `nil`

String

Beliebiger Textblock (siehe auch String-Literale)

Instanzmethoden

- `str + str` – Strings verketten
- `str * int` – Inhalt `int`-mal hintereinander
- `str[n]` – Zeichen Nr. n-1
- `str[m, n]` – maximal n Zeichen ab Position n-1

- `str[str2]` – der Teilstring `str2`, falls er vorkommt, oder `nil`, falls nicht
- `str[...] = str2` – der Teilstring wird durch str2 ersetzt (Länge darf unterschiedlich sein)
- `str.capitalize`, `str.capitalize!` – jeden Anfangsbuchstaben groß schreiben (die Variante mit ! verändert – wie bei allen nachfolgenden Operationen – eine String-Variable dauerhaft)
- `str.center(n)` – zentriert den String mit Hilfe von Leerzeichen auf einer Gesamtbreite von n Zeichen
- `str.chop, str.chop!` – entfernt das letzte Zeichen
- `str.comp, str.chomp!` – entfernt das letzte Zeichen nur dann, wenn es Whitespace ist (praktisch für den Zeilenumbruch bei `gets`-Eingaben)
- `str.count(str2)` – addiert, wie oft jedes Zeichen aus `str2` in `str` vorkommt
- `str.downcase`, `str.downcase!` – wandelt alle enthalteten Buchstaben in Kleinschreibung um
- `str.gsub(regexp, ersatz)`, `str.gsub!(regexp, ersatz)` – ersetzt alle Vorkommen von `regexp` durch `ersatz`. Geklammerte Ausdrücke aus `regexp` stehen in `ersatz` als \1, \2 usw. zur Verfügung
- `str.gsub(regexp) {...}`, `str.gsub!(regexp) {...}` – leitet Treffer in den Block weiter und ermöglicht so beliebig komplexe Ersatzausdrücke
- `str.index(str2)` – liefert die Position, an der `str2` in `str` vorkommt, oder `nil`, falls `str2` gar nicht vorkommt
- `str.length` – Anzahl der Zeichen
- `str.strip, str.strip!` – entfernt sämtlichen Whitespace an Anfang und Ende
- `str.sub(regexp, ersatz)`, `str.sub!(regexp, ersatz)` – ersetzt das erste Vorkommen von `regexp` durch `ersatz`
- `str.succ` – direkter alphabetischer Nachfolger (z.B. `"abc".succ` => `"abd"`)
- `str.to_i` – wandelt möglichst viele Zeichen von links an in eine Ganzzahl um
- `str.to_i(basis)` – wandelt möglichst viele Zeichen in eine Ganzzahl um, die aus `basis` (2-36) konvertiert wird
- `str.to_f` – wandelt möglichst viele Zeichen von links an in eine Fließkommazahl um
- `str.tr(str1, str2)`, `str.tr!(str1, str2)` – ersetzt die Zeichen aus `str1` durch Zeichen aus `str2` an der entsprechenden Position
- `str.each_byte {...}` – Iterator über alle Zeichen, die als Code verfügbar sind und mittels `chr` wieder in Zeichen umgewandelt werden können

Time

Kapselt Datum und Uhrzeit

Konstruktoraufruf

`Time.new` – speichert aktuelle Systemzeit

Klassenmethoden

- `t = Time.now` – Synonym für `Time.new`
- `t = Time.parse(string)` – versucht, ein gültiges Datum aus `string` zu extrahieren

Instanzmethoden

- `t.year` – vierstellige Jahreszahl
- `t.month` – Monat, numerisch (1-12)
- `t.day` – Tag im Monat (1-31)
- `t.wday` – numerisch codierter Wochentag (0=So., 1=Mo., ..., 6=Sa.)
- `t.hour` – Stunde (0-23)
- `t.min` – Minute (0-59)
- `t.sec` – Sekunde (0-59)
- `t.strftime(format)` – formatiert Datum und Uhrzeit gemäß Formatstring. Eine Liste der Platzhalter finden Sie in Kapitel 3

ANHANG B
Ressourcen und Tools

Essen vertreibt den Hunger, Lernen vertreibt die Dummheit.
– Chinesisches Sprichwort

In diesem Anhang werden einige Bücher und Websites empfohlen, die Ihnen helfen, mehr Ruby zu lernen sowie zusätzliche Informationen über Themen zu erhalten, die in diesem Buch nebenher angesprochen wurden.

Bücher

Zuerst werden einige Bücher zu Ruby und Rails aufgelistet, danach geht es um Objektorientierung im Allgemeinen, Servertechnologien und mehr.

Ruby und Ruby on Rails

- Dave Thomas, Andy Hunt et al.: *Programming Ruby – The Pragmatic Programmers' Guide*. 2. Auflage 2004, Pragmatic Programmers. Das berühmte »Pickaxe Book« machte Ruby zum ersten Mal außerhalb Japans bekannt. Die erste Auflage ist auf Englisch und Deutsch im Web verfügbar (siehe den Abschnitt *Web-Ressourcen*). Die erweiterte 2. Auflage bietet einen umfassenden Überblick und eine ausführliche Referenz der Programmiersprache Ruby in Version 1.8. Das ideale Buch zum Weiterarbeiten, wenn Sie mit dem vorliegenden fertig sind.
- *Armin Roehrl, Stefan Schmiedl, Clemens Wyss*: Programmieren mit Ruby. Heidelberg 2002, dpunkt Verlag. Dies war das erste deutschsprachige Originalbuch zu Ruby. Es eignet sich vor allem für Umsteiger von anderen Programmiersprachen, weniger für komplette Einsteiger. Leider ist es etwas älter und behandelt keine aktuelle Ruby-Version.

- Lucas Carlson, Leonard Richardson: *Ruby Cookbook*. Sebastopol 2006, O'Reilly Media. Hier finden Sie im bewährten O'Reilly-Kochbuch-Stil zahlreiche Praxisrezepte für den Soforteinsatz im Ruby-Alltag. Jedes Rezept gliedert sich dabei in Problemstellung, Lösung und erweiterte Diskussion.
- Denny Carl: *Praxiswissen Ruby on Rails*. Köln 2007, O'Reilly Verlag. Wenn Sie Ruby gelernt haben und sich nun näher mit Rails beschäftigen möchten, dürfte dieses Buch genau das Richtige für Sie sein. Es erklärt Schritt für Schritt mit praxisnahen Beispielen, wie Sie Rails-Anwendungen programmieren und veröffentlichen können.
- Dave Thomas, David Heinemeier Hansson: *Agile Web Development with Rails*. 2. Auflage 2007, Pragmatic Programmers. Dieses Buch ist eine ausführliche Praxiseinführung und umfassende Referenz zu Ruby on Rails, geschrieben vom Rails-Erfinder David Heinemeier Hansson und dem Pragmatic Programmer Dave Thomas. Die zweite Auflage geht unter anderem stärker auf Ajax ein.
- Dave Thomas, David Heinemeier Hansson: *Agile Webentwicklung mit Rails*. München 2006, Hanser Verlag. Die erste Auflage des vorgenannten Werks in deutscher Übersetzung.
- Rob Orsini: *Rails Cookbook*. Sebastopol 2007, O'Reilly Media. Auch zu Ruby on Rails ist inzwischen ein Kochbuch mit praktischen Anleitungen erschienen.

Objektorientierung

- Bernhard Lahres, Gregor Rayman: *Praxisbuch Objektorientierung*. Bonn 2006, Galileo Computing. In diesem Buch werden alle wichtigen Konzepte der objektorientierten Programmierung ausführlich und methodisch anhand praxisnaher Beispiele erläutert. Neben C++, C# und Java kommt auch Ruby zur Sprache.
- Eric Freeman, Elizabeth Freeman, Kathy Sierra: *Entwurfsmuster von Kopf bis Fuß*. Köln 2006, O'Reilly Verlag. Der comicartige und verspielte Stil der »Von Kopf bis Fuß«-Reihe hilft Ihnen, Themen nicht nur zu lernen, sondern auch zu behalten. In diesem Band werden Entwurfsmuster (Design Patterns) behandelt, standardisierte Lösungsansätze für immer wiederkehrende Programmierprobleme. Jeder ernsthafte OO-Entwickler sollte sich eines Tages mit Entwurfsmustern beschäftigen, weil sie enorm viel Arbeit sparen.
- Christoph Kecher: *UML 2.0. Das umfassende Handbuch*. 2. Auflage, Bonn 2006, Galileo Computing. Jede OOP-Anwendung, die über ein bis zwei Klassen hinausgeht, sollte sorgfältig geplant werden. Mittel der Wahl sind meist die verschiedenen Diagramme der Unified Modeling Language. Dieses Handbuch dokumentiert ausführlich aller verfügbaren Diagrammtypen und gibt wertvolle Hinweise für ihren Einsatz.

Netzwerke, Server-Technologien, Webentwicklung

- Craig Hunt: *TCP/IP Netzwerk-Administration*. 3. Auflage, Köln 2003, O'Reilly Verlag. Dieser Klassiker bietet eine ausgewogene Mischung zwischen der Erläuterung von Protokollgrundlagen und praktischen Administrationstipps von TCP/IP-Netzwerken auf UNIX-Systemen.

- Lincoln D. Stein: *Network Programming with Perl*. Reading 2000, Addison-Wesley. Zugegeben: Dieses Buch ist einige Jahre alt, nur auf Englisch verfügbar (eine zwischenzeitlich angebotene deutsche Ausgabe ist ausverkauft) und behandelt Perl. Aber kein anderes Buch behandelt die Implementierung von TCP/IP-Servern und -Clients gründlicher, verständlicher und praxisorientierter. Da viele Konzepte der Ruby-Netzwerkprogrammierung den – zum Teil sogar gleichnamigen – Perl-Bibliotheken nachempfunden wurden, ist es auch für Ruby-Entwickler geeignet.
- Sascha Kersken: *Apache 2 – inkl. Apache 2.2*. 2. Auflage, Bonn 2006, Galileo Computing. Mein Buch über den Apache-Webserver ist ein ausführliches Handbuch zur Installation, Konfiguration und Administration.
- Mark Lubkowitz: *Webseiten programmieren und gestalten*. 3. Auflage, Bonn 2006, Galileo Press. Dieser ausführliche Band vermittelt das Wichtigste über sämtliche client- und serverseitigen Web-Technologien: HTML, CSS, JavaScript, Ajax, CGI, PHP und MySQL.
- Sascha Kersken: *Praktischer Einstieg in MySQL mit PHP*. Köln 2005, O'Reilly Verlag. Schnelleinstieg in Einrichtung, Konfiguration und praktischen Einsatz des Datenbanksystems MySQL und Datenbankprogrammierung mit PHP.

Web-Ressourcen

Nachfolgend finden Sie die Adressen einiger Websites mit Informationen zu Ruby, Software-Downloads und so weiter. Bei Drucklegung waren die URLs aktuell, aber wie Sie wissen, veralten Web-Adressverzeichnisse recht schnell. Deshalb finden Sie dieselbe Liste auf der Website zum Buch in aktueller Form. Dort können Sie zudem per Formular selbst Links vorschlagen.

Meine eigene Site zu diesem Buch hat die Adresse *http://buecher.lingoworld.de/ruby*. Bei O'Reilly hat das Buch die URL *http://www.oreilly.de/catalog/rubybasger*. Auf beiden Sites können Sie Listings herunterladen, Lob oder Kritik loswerden und mehr.

Ruby-Dokumentation

- *http://ruby-doc.org* – die Ruby-Dokumentationszentrale. Enthält sämtliche Inhalte der `ri`-Dokumentation sowie Links auf viele weitere Ressourcen.
- *http://www.rubycentral.com/book/* – die erste Auflage des Pickaxe-Buchs von Andy Hunt und Dave Thomas (siehe oben im Bereich Bücher) im Volltext.
- *http://home.vr-web.de/juergen.katins/ruby/buch/index.html* – dasselbe Buch auf Deutsch auf der Website des Übersetzers.
- *http://poignantguide.net/ruby/* – hier finden Sie *Why's (Poignant) Guide to Ruby*, ein sehr eigenwilliges Online-Buch zum Thema Ruby von why the lucky stiff (Blogger-Synonym). Es enthält eine gründliche und sehr kurzweilige Ruby-Einführung und jede Menge Cartoons mit Füchsen.
- *http://www.approximity.com/rubybuch2/rb_main.html* – noch ein deutschsprachiges Online-Buch zu Ruby. Es handelt sich um die Webausgabe des Buchs *Programmieren mit Ruby* von Armin Roehrl, Stefan Schmiedl und Clemens Wyss, das auch gedruckt erschienen ist (siehe oben).

Ruby-Community

- *http://www.ruby-lang.org/en/* – die Ruby-Homepage. Hier wird jeweils die neueste Version zum Download angeboten. Daneben finden Sie auf dieser Site zahlreiche Zusatzinformationen und Links.
- *http://www.rubygarden.org/* – umfangreiche Ruby-Ressourcen-Site von Chad Fowler.
- *http://tryruby.hobix.com/* – interaktive JavaScript-Anwendung, mit der Sie `irb` im Browser benutzen können. Inklusive kurzem Ruby-Tutorial.
- *http://www2.ruby-lang.org/en/20020104.html* – Übersicht über die wichtigsten Ruby-Mailing-Listen, sowohl für Nutzer als auch für (potentielle) Mitentwickler der Sprache.
- *http://www.rubyforen.de/* – umfangreiches deutschsprachiges Diskussionsforum zu den verschiedensten Ruby-Themen.
- *http://www.rubyonrails.org/* – die Webseite des Projekts Ruby on Rails mit Downloads, Dokumentation und allerlei nützlichen Links.
- *http://www.radrails.org/* – RadRails ist eine integrierte Entwicklungsumgebung (IDE) zum Erstellen von Ruby- und Ruby on Rails-Anwendungen. Es handelt sich um eine spezielle Ruby-Erweiterung der beliebten Open Source-IDE Eclipse.

Sonstige Webseiten

- *http://de.selfhtml.org/* – Tutorial und umfassende Referenz zu HTML und Webentwicklung von Stefan Münz
- *http://httpd.apache.org/* – die Website des Webservers Apache 2 mit Downloads und Dokumentation
- *http://www.apachefriends.org/de/xampp.html* – das integrierte Apache/PHP/MySQL-Serverpaket XAMPP
- *http://buecher.lingoworld.de/apache2/* – Website zu meinem Buch *Apache 2* (siehe oben) mit zahlreichen Konfigurationstipps inklusive Suchfunktionen und einem Diskussionsforum
- *http://www.mysql.com/* – Website der Firma MySQL AB. Hier können Sie den Datenbankserver MySQL herunterladen und seine Dokumentation lesen
- *http://www.galileocomputing/openbook/kit/* – kostenlos verfügbare Online-Version meines Buchs *Kompendium der Informationstechnik* (Bonn 2003, Galileo Computing; die Neuauflage heißt *Handbuch für Fachinformatiker*). Enthält theoretische Grundlagen und praktische Einführungen in zahlreiche IT-Themengebiete. TCP/IP und Datenbanken werden beispielsweise erheblich ausführlicher erläutert, als es im vorliegenden Buch möglich wäre.

Index

!, Operator 56
!, Suffix von Änderungsmethoden 66
!=, Operator 53
!~, Operator 88
#!, Shebang 34
#, Kommentar 33
#{...}, eingebettete Ausdrücke 42
$, Kennz. f. globale Variablen 47
$, Regexp-Anfangsmarke 94
%, Operator 51
%Q, Quoting-Operator 43
%q, Quoting-Operator 43
%r, Regexp-Operator 86
%w, Operator 49
&&, Operator 55
&, Operator 56
( ), Regexp-Gruppierung 91
**, Operator 52
*, arithmetischer Operator 51
*, Regexp-Quantifizierer 92
*, String-Operator 61
*=, Operator 60
+, arithmetischer Operator 51
+, Regexp-Quantifizierer 93
+, String-Operator 61
+, Vorzeichen 52
+=, Operator 60
-, Operator 51
-, String-Erweiterung 190
-, Vorzeichen 52
., Regexp-Platzhalter 90
.htaccess-Dateien 282
/, Operator 51
/.../, reguläre Ausdrücke 86
/=, Operator 60
/etc/hosts, Datei 281
:, Symbol-Markierung 45
<, Operator 53
<, Vererbungs-Operator 198
<<, Bit-Operator 57
<<, HIER-Dokument-Marke 44
<=, Operator 54
<=>, Operator 55
    definieren 209
<Directory> 282
<Location> 282
-=, Operator 60
=, Setter-Kennzeichen 196
==, Operator 52
===, Operator 54
=~, Operator 86, 97
>, Operator 53
>=, Operator 54
>>, Operator 57
?, Regexp-Quantifizierer 90
?, Zeichencode-Operator 68

?:, Operator 61
@, Instanzvariablen-Kennzeichen 167
@@, Klassenvariablen-Kennzeichen 167
[ ], Array-Methode 49
[ ], Hash-Methode 50
[ ], in regulären Ausdrücken 88
\, für Escape-Sequenzen 42
\A, Regexp-Anfangsmarke 95
\b, Regexp-Wortgrenze 95
\B, Regexp-Wortinneres 95
\D, Regexp-Nichtziffer 97
\d, Regexp-Ziffer 97
\S, Regexp-Nicht-Whitespace 97
\s, Regexp-Whitespace 97
\W, Regexp-Nichtwort-Zeichen 97
\w, Regexp-Wortzeichen 97
\Z, Regexp-Endmarke 95
^, Operator 57
^, Regexp-Anfangsmarke 94
^, Regexp-Verneinung 89
{ }, Regexp-Quantifizierer 94
|, Operator 56
|, Regexp-Alternativen 96
||, Operator 55
0, Oktalzahlen-Kennzeichen 39
0b, Dual-Kennzeichen 40
0x, Hexadezimal-Kennzeichen 40

A

Abgeleitete Klasse 113
abs, Methode 64
Accept, HTTP-Header 284
accept, Methode 233
Accept-Encoding
  HTTP-Header 285
Accept-Language
  HTTP-Header 284
Accept-Ranges
  HTTP-Header 286
Accessor-Methoden 197
Action Controller 335
Action Mailer 336
Action Pack 335
Action View 335
Action Web Service 336
Active Record 335
  create, Methode 349
  Datensätze ändern 350
  Datensätze einfügen 349
  Datensätze finden 348
  find, Methode 348
  Grundlagen 348
  update, Methode 350
Active Support 336
ActiveRecord::Base, Klasse 346
Ajax 336
Alias, Apache-Webserver 283
alias, Methode 186
Allow, Apache-Webserver 283
AllowOverride 282
and, Operator 55
Anfrage (HTTP) 284
Anführungszeichen, einzelne/doppelte 42
Antwort (HTTP) 285
  als Objekt 242
Anweisungen 32
  Bestandteile 32
Apache-Webserver 274
  .htaccess-Dateien 282
  <Directory> 282
  <Location> 282
  Alias 283
  Allow 283
  AllowOverride 282
  CGI-Konfiguration 287
  Deny 283
  DirectoryIndex 283
  Direktiven 280
  DocumentRoot 281
  Eigenschaften 274
  ErrorLog 291
  Installation (UNIX) 277
  Installation (Windows) 276
  Konfiguration 279
  Listen 280
  LoadModule 280
  mod_cgi 286
  mod_ruby 329
  Module 275
  Multi-Processing-Module 275
  Options 282
  Order 283
  ScriptAlias 287
  ServerAdmin 281
  ServerName 281
  ServerRoot 280
  XAMPP 274
ApplicationController, Klasse 339

argrechner.rb, Skript 123
ARGV 122
Arithmetik
  Fließkommazahlen 41
  Ganze Zahlen 37
  Raumfolgearithmetik 38
Arithmetische Operatoren 51
Array, Klasse 48
  [ ], Methode 49
  each_with_index, Iterator 267
  join, Methode 71
  length, Methode 68
  Methoden 68
  pop, Methode 68
  push, Methode 68
  Referenz 363
  Reverse, Methode 69
  search, eigene Erweiterung 177
  shift, Methode 69
  slice, Methode 69
  sort, Methode 69
  sum, Erweiterung 188
  unshift, Methode 69
Arrays 48
  als Parameter 172
  Index 49
  length, Methode 49
  Wertzuweisung 49
  zusammenfügen 71
attr_accessor 197
attr_reader 194
attr_writer 195
Attribute *siehe* Instanzvariablen
Ausdrücke
  als Rückgabewerte 175
  eingebettete 42
  Fallentscheidungen in 78
Ausgabeumleitung 115
Ausnahmen *siehe* Exceptions

B

Backticks für Shell-Kommandos 259
bash 10
BasicSocket
  Klasse 230
begin, MatchData-Methode 148
Beispiele
  argrechner.rb 123
  echoclient.rb 238
  echoforkclient.rb 265
  echoforkserver.rb 263
  echoserver.rb 235
  echothreadserver.rb 269
  hello.rb 19
  httpclient.rb 247
  kalender.rb 143
  listdir.rb 192
  mod_text.rb 216
  modtext.rb 28
  rechner.rb 24
  rechteck.rb 112
  rgb.rb 58
  rgb_farbe.rb 181
  temperatur.rb 293
  textblog.rb 130
  wrappager.rb 246
  zug_imp.rb 154
  zug_oo.rb 158
Bereiche (Ranges) 45
  in Arrays umwandeln 72
  Vergleiche mit Einzelwerten 54
Berkeley Socket API 229
Betriebssysteme, Ruby-Unterstützung in 3
Bezeichner 46
  CamelCode 47
Bignum, Klasse
  Referenz 364
Bit-Operationen 56
  Einsatzgebiete 57
block_given?, Methode 178
Blöcke 101
  definieren 176
  optional akzeptieren 178
  Variablen in 102
body, Methode 242
Bogenmaß 65
Broadcast-Adresse (IP) 224
Broser, Projekt
  Test 251
Browser, Projekt 245
  Erläuterungen 251
  Implementierung 247
Bruchrechnung 145

C

C (Programmiersprache) 2
C++ 2
CamelCode 47

Cascading Style Sheets *siehe* CSS
case-when-Fallentscheidungen 76
  else 77
cd, Konsolenbefehl 12
ceil, Methode 64
CGI
  Apache-Konfiguration 287
  Definition 286
  Erste Beispiele 287
  Fehlersuche 291
  Ruby-Bibliothek 292
  Umgebungsvariablen 289
CGI, Klasse
  checkbox_group, Methode 302
  CSS einbetten 313
  CSS einbinden 300
  file_field, Methode 303
  form, Methode 300
  Formulardaten lesen 297
  Formularelemente erzeugen 300
  Funktionsumfang 292
  has_key?, Methode 297
  hidden, Methode 309
  HTML erzeugen 298
  HTML-Versionen 298
  param, Methode 297
  radio_group, Methode 302
  reset, Methode 303
  submit, Methode 303
  text_field, Methode 301
  textarea, Methode 301
  Upload 303
  XHTML-Unterstützung 298
CGI.escapeHTML, Methode 303
CGI::Cookie 305
CGI::Cookie, Klasse 305
CGI::Session, Klasse 310
checkbox_group, Methode 302
Child-Prozess 257
chomp, Methode 66
chop, Methode 66
chr, Methode 67
CIDR 225
class, Methode 46, 212
Client 232
close, Methode 127
code, Methode 242
collect, Iterator 105
Common Gateway Interface *siehe* CGI
Comparable, Modul 209
Compiler 3
Connection, HTTP-Header 285, 286
console, Rails-Tool 348
Content-Length, HTTP-Header 286
Content-Type, HTTP-Header 286
  CGI 287
Cookies 304
  auslesen 306
  erzeugen 305
  setzen 305
cos, Math-Methode 65
CREATE TABLE, SQL-Anweisung 318
create, Active Record-Methode 349
CRUD (Datenbank-Management) 347
CSS 300
  externe Datei 300
  in CGIs einbetten 313

D

database.yml, Rails-Konfigurationsdatei 344
Datagramm (IP) 224
Date, HTTP-Header 286
Dateien
  erweiterte Funktionen 134
  Existenz testen 128, 138
  Modus 126
  öffnen 126
  schließen 127
  Typ testen 138
  umbenennen 179
Dateien verarbeiten 125
Dateisystem 11
Datei-Uploads 303
Dateizeiger 134
Datenbanken
  CRUD 347
  für Rails vorbereiten 343
Datenbankzugriff 314
Datenströme 114
Datentyp ermitteln 212
Datentypen 46
Datentyp-Umwandlung 70
  in Fließkommazahl 70
  in Ganzzahl 70
  in String 70
Datum und Uhrzeit 138
  Berechnungen 142
  bestimmter Zeitpunkt 142
  Einsatzbeispiele 143
  EPOCH 142

Formate 139
Monat, deutsch 140
RFC 1123 139
Systemzeit 138
Time, Klasse 138
Wochentag, deutsch 140
day, Methode 139
DBI 321
De Morgan-Theorem 76
def, Schlüsselwort 169
Definition von Variablen 45
denominator, Methode 146
Deny, Apache-Webserver 283
Deployment (Rails) 354
Design Patterns 205
DHCP 225
Dir, Klasse 137
read, Methode 137
Referenz 364
dir, Windows-Konsolenbefehl 13
directory?, File-Methode 138
DirectoryIndex 283
Direktiven, Apache-Webserver 280
DNS (Domain Name System) 227
DocumentRoot 281
downcase, Methode 66
downto, Iterator 104
Dualzahlen 40
Duck Typing 47
Dynamische Typisierung 47

E

each, Iterator 101
each_byte, Iterator 104
each_key, Iterator 103
each_pair, Iterator 103
each_value, Iterator 103
each_with_index, Iterator 267
ECHO-Client 238
für nebenläufige Server 265
echoclient.rb, Skript 238
echoforkclient.rb, Skript 265
echoforkserver.rb, Skript 263
ECHO-Server
Einzelberbindung 235
ECHO-Server, Forking 263
ECHO-Server, Threading 269
echoserver.rb, Skript 235
echothreadserver.rb, Skript 269
Ein- und Ausgabe 114
Dateien 125
getc 120
gets 119
Konsole 114
Nonblocking 234
Pipe 115
print 116
printf 117
puts 116
read 121
Standardkanäle 114
STDERR 115
STDIN 114
STDOUT 115
Umleitung 115
Verzeichnisse 136
Eingabeaufforderung 10
Eingabeumleitung 115
Eingebettete Ausdrücke 42
else 73
bei case-when 77
bei if 73
elsif 74
Elternklasse 113
Emacs, Editor 18
Embedded Ruby *siehe* eRuby
End Of File (EOF) 120
end, MatchData-Methode 148
Endlosschleifen (kontrollierte) 81
Entwurfsmuster *siehe* Design Patterns
ENV 289
EOF 120
EPOCH 142
erb *siehe* eRuby
ErrorLog (Apache) 291
erubis *siehe* eRuby
eRuby 329
als Rails-Template 340
Escape-Sequenzen 42
in regulären Ausdrücken 88
Zeichencode 68
ETag, HTTP-Header 286
Exceptions 209
abfangen 209
auslösen 210
eigene 211
exists?, File-Methode 128, 138
exit, Methode 125

F

Fallentscheidungen 73
 case-when 76
 durch logische Operationen 77
 if 73
 in Ausdrücken 78
 unless 75
false
 als Literal 44
fetch_hash, Methode (Mysql) 324
fetch_row, Methode 324
File, Klasse 126
 close, Methode 127
 Instanzen 126
 open, Methode 127
 Referenz 364
 rewind, Methode 135
 seek, Methode 135
 sort, Methode 136
File.directory?, Methode 138
File.exists?, Methode 128, 138
File.file?, Methode 138
File.rename, Methode 179
file?, File-Methode 138
file_field, Methode 303
find, Active Record-Methode 348
Fixnum, Klasse 37
 Referenz 365
Fließkomma-Arithmetik 41
Fließkommazahlen 40
 konvertieren in 70
Float, Klasse 40
 Referenz 365
 round, Methode (Erweiterung) 186
floor, Methode 64
for, Schleife 83
fork, Methode 257
Forking 257
Forking-Server 258, 263
form, Methode 300
Formulardaten (Web) 290
FreeRIDE, Ruby-Entwicklungsumgebung 18

G

Ganze Zahlen 36
 Dual 40
 Hexadezimal 40
 konvertieren in 70
 Oktal 39
Ganzzahlarithmetik 37
Geheimhaltungsstufen *siehe* Zugriffsschutz (OOP)
Geltungsbereich 47
gem, rubygems-Aufruf 309, 322
generate, Rails-Tool 339, 346
Geschichte von Ruby 1
GET, HTTP-Methode 284
 Query-String 291, 309
get, Methode (Net::HTTP) 242
getc, Methode 120
gets, Methode 119
getservbyname, Methode 233
Getter-Methoden 113
 automatische 194
Gier (Regexp) 93
Gleichzeitigkeit *siehe* Nebenläufigkeit
Gleitkommazahlen 40
Globale Methoden 184
Globale Variablen 47
GNOME Terminal 10
GNU General Public License 2
GPL 2
greed (Regexp) 93
Groß- und Kleinschreibung
 Dateinamen 12
Grundrechenarten 51
GServer, Klasse 270
gsub, Methode 98
Gültigkeitsbereich 47

H

Hallo Welt 17
Hansson, David Heinemeier 333
has_key?, Methode 297
HAS-A-Beziehung (OOP) 165
Hash, Klasse
 [ ], Methode 50
 each_key, Iterator 103
 each_pair, Iterator 103
 each_value, Iterator 103
 Referenz 365
Hashes 50
 als Parameter 173
 Schlüssel 50
Header (HTTP) 284

Hello World 17
hello.rb, Skript 19
Hexadezimalzahlen 40
hidden, Methode (CGI) 309
Hidden-Formularfelder 309
HIER-Dokumente 43
Hilfe (ri) 148
Host, HTTP-Header 285
hosts-Datei 281
hour, Methode 139
HTML
    erzeugen (CGI) 298
    Kurzeinführung 243
    Sonderzeichen maskieren 303
HTTP *siehe* HyperText Transfer Protocol
httpclient.rb, Skript 247
HTTP-Header
    auslesen 242
HyperText Markup Language *siehe* HTML
HyperText Transfer Protocol (HTTP)
    Anfrage 284
    Antwort 285
    Antwort (als Objekt) 242
    Client-Bibliothek 241
    Cookies 304
    GET-Anfragen 284
    Grundlagen 284
    Header 284
    Methoden 284
    Parameter 290
    POST-Anfragen 284
    Query-String 291, 309
    Sessions 308
    Statuscode 243
    Statuscodes 285
    Weiterleitungen 243

I

i, Regexp-Modifier 96
I/O *siehe* Ein- und Ausgabe
ID (Objekte) 212
if, Fallentscheidung 73
    else 73
    elsif 74
    nachgestellt 75
Imperative Programmierung 154
Import externer Skripten 187
include, Methode 208
Index
    Array 49
    Hash 50
initialize, Konstruktor-Methode 113
Input/Output *siehe* Ein- und Ausgabe
INSERT, SQL-Anweisung 319
Installation
    Ruby 4
instance_of?, Methode 213
Instanz (Begriffsdefinition) 164
Instanzen
    Erzeugen 168
    self 184
Instanziierung 168
Instanzvariablen
    Accessor-Methoden 197
    automatische Getter 194
    automatische Setter 195
    Rails-Datenaustausch mit 340
Instanzvariablen (Begriffsdefinition) 167
Integer 36
Integer, Klasse
    Referenz 366
Interactive Ruby (irb) 14
Interfaces (Java) 207
Internet Protocol (IP) 224
    Adressierung 224
    Broadcast-Adresse 224
    CIDR 225
    Datagramme 224
    DHCP 225
    IPv4 224
    IPv6 224
    Multicast 224
    NAT 225
    Routing 226
    Teilnetzmaske 225
Interpreter 3
Interprozesskommunikation (IPC) 260
Introspektion 212
    class 212
    instance_of? 213
    is_a? 213
    methods 214
    object_id 212
    ObjectSpace 212
    respond_to? 214
IO, Klasse
    Referenz 366
IP *siehe* Internet Protocol
IP-Adressen 224
    automatische Zuweisung 225
    Klassen 224
IPC *siehe* Interprozesskommunikation

irb 14
als Taschenrechner 14
Beenden 15
in Rails 348
Starten 14
Zeilennummerierung deaktivieren 23
is_a?, Methode 213
IS-A-Beziehung (OOP) 165
Iteratoren 101
Allgemeines 101
Blockvariablen 102
collect 105
downto 104
each 101
each_byte 104
each_key 103
each_pair 103
each_value 103
each_with_index 267
eigene 176
step 104
times 104
upto 104
Iteratorn
yield 176

J

Java 2
join, Methode 71
join, Methode (Thread) 267

K

kalender.rb, Skript 143
Kapselung 154
KDE Konsole 10
kill, Methode 262
kind_of?, Methode 213
Klammern, f. Priorität 62
Klasen
Time 369
Klasse ermitteln 212
Klassen 305
ActiveRecord::Base 346
ApplicationController 339
Array 48, 363
BasicSocket 230
Begriffsdefinition 164
Bignum 364
CGI 292
CGI::Session 310
Dir 137, 364
Einführung 113
Entwurf 163
erweitern 185
File 126, 364
Fixnum 37, 365
Float 40, 365
GServer 270
Hash 365
Instanziierung 168
Integer 366
IO 366
Konstruktor 167
MatchData 146, 366
Math (Modul) 65
Mysql 323
Mysql::Result 323
Net::HTTP 241
Net::HTTPOK 242
Object 367
Rational 145, 368
Regexp 146, 368
String 368
TCPServer 232
TCPSocket 231
Time 138
UUID 309
Vererbung 198
Klassen von IP-Adressen 224
Klassendiagramme (UML) 165
Klassenmethoden 128, 179
private, Abgrenzung 202
Klassenvariablen 167
Kommandozeilen-Argumente 122
Kommentare 33
RDoc 34
Konsole 10
Ein- und Ausgabe 114
Ein-/Ausgabeumleitung 115
Konsolenbefehle, Übersicht 13
Konstanten 48
Konstruktor 113
Beispiele 168
new, Aufruf 114
super (Elternklasse) 199
Syntax 167
Konstruktor (Begriffsdefinition) 167

Kontrollstrukturen 72
  Fallentscheidungen 73
  Schleifen 79
Kryptografie, Bit-Operatoren und 57

L

Last-Modified, HTTP-Header 286
Laufzeit 212
Laufzeittypüberprüfung *siehe* Introspektion
length, Array-Methode 49, 68
length, String-Methode 66
Linux
  Apache installieren 277
  Dateisystem 11
  Konsolenbefehle 13
  MySQL installieren 316
  Prompt 11
  Ruby installieren 7
  Shebang 34
  Shell 10
  Terminalfenster 10
listdir.rb, Skript 192
Listen, Apache-Direktive 280
Literale 36
  Bereiche (Ranges) 45
  nil 45
  numerische 36
  spezielle 44
  Strings 41
  Symbole 45
  true/false 44
Lizenz von Ruby 2
LoadModule 280
localhost 281
log, Math-Methode 65
Logarithmus 65
Logische Operationen 55
  als Fallentscheidungen 77
  De Morgan-Theorem 76
  Short-Circuit-Verfahren 56
loop, Schleife 81
ls, UNIX-Konsolenbefehl 13

M

m, Regexp-Modifier 96
Mac OS X
  Apache installieren 277
  Dateisystem 11
  Konsolenbefehle 13
  MySQL installieren 316
  Prompt 11
  Ruby installieren 7
  Shebang 34
  Shell 10
  Terminalfenster 10
match, Methode 147
MatchData, Klasse 146
  begin, Methode 148
  end, Methode 148
  offset, Methode 148
  post_match, Methode 148
  pre_match, Methode 148
  Referenz 366
Matching (Regexp) 86
  Einzelheiten 97
Math, Modul 65
  Referenz 367
Math.cos, Methode 65
Math.log, Methode 65
Math.sin, Methode 65
Math.sqrt, Methode 65
Math.tan, Methode 65
Mathematische Methoden 64
Matsumoto, Yukihiro 1
Matz 1
Mehrfachvererbung 207
message, Methode 242
Methoden
  [ ], Array 49
  [ ], Hash 50
  [ ], String 67
  abs 64
  accept 233
  Accessors 197
  alias 186
  Array-Parameter 172
  Arrays 68
  attr_accessor 197
  attr_reader 194
  attr_writer 195
  Aufrufe in Ausdrücken 62
  begin (MatchData) 148
  Begriffsdefinition 164, 169
  block_given? 178
  Blöcke, optionale 178
  body 242
  ceil 64

CGI.escapeHTML 303
checkbox_group 302
chomp 66
chop 66
chr 67
class 46, 212
close 127
code 242
collect (Iterator) 105
create (Active Record) 349
day 139
denominator 146
downcase 66
downto, Iterator 104
dynamische Parameter 174
each (Iterator) 101
each_byte (Iterator) 104
each_key (Iterator) 103
each_pair (Iterator) 103
each_value (Iterator) 103
each_with_index (Iterator) 267
end (MatchData) 148
exit 125
fetch_hash (Mysql) 324
fetch_row 324
File.directory? 138
File.exists? 128, 138
File.file? 138
File.rename 179
file_field 303
find (Active Record) 348
floor 64
fork 257
form 300
get (Net::HTTP) 242
getc 120
gets 119
getservbyname 233
Getter 113
Globale 184
has_key? 297
Hash-Parameter 173
hidden (CGI) 309
hour 139
include 208
initialize (Konstruktor) 113
instance_of? 213
is_a? 213
Iteratoren, eigene 176
join 71
join (Thread) 267
kind_of? 213
Klassenmethoden 179
length (Array) 49, 68
length (String) 66
match 147
Math.cos 65
Math.log 65
Math.sin 65
Math.sqrt 65
Math.tan 65
mathematische 64
message 242
methods 214
min 139
month 139
now 138
num_rows (Mysql) 324
numerator 146
object_id 212
offset 148
open 127
Operatoren als 188
param 297
Parametervariablen 170
parse (Time) 142
pop 68
post_match 148
pre_match 148
print 116
printf 117
Process.kill 262
Process.ppid 262
Process.wait 259
push 68
puts 116
query 323
radio_group 302
rand 63
read 121
read (Dir) 137
Rekursion 190
require 187
reset (CGI) 303
respond_to? 214
reverse (Array) 69
reverse (String) 66
rewind 135

round 64
Rückgabewerte 175
sec 139
seek 135
select_db 323
Setter 195
shift 69
sleep 267
slice 69
sort 69
sort (File) 136
sprintf 119
step (Iterator) 104
strftime 140
Strings 66
sub 98
submit 303
swapcase 66
text_field 301
textarea 301
times, Iterator 104
to_a 72
to_f 70
to_i 70
to_s 70
trap 261
Typumwandlung 70
unshift 69
upcase 66
update (Active Record) 350
upto (Iterator) 104
wday 139
year 139
yield 176
Methodenkopf 169
Methodenrumpf 169
methods
Methode 214
min, Methode 139
Mixins 208
include 208
mkdir, Konsolenbefehl 13
mod_ruby 329
mod_text.rb, Skript 216
Modifier (Regexp) 96
modtext.rb, Skript 28
Module 205
Comparable 209
Math 65, 206, 367
Mixins 208
Namensraum 206
ObjectSpace 212
Module (Apache-Webserver) 275
Modulo 51
Monat, deutsch 140
month, Methode 139
Multicast (IP) 224
MVC 333
MySQL 314
Daten einfügen 319
Datenbank erzeugen 318
Datensätze lesen 319
Einführung 318
Installation (Linux) 316
Installation (Mac OS X) 316
Installation (UNIX) 316
Installation (Windows) 315
Konsolen-Client 318
Ruby-Zugriff 321
Tabelle erzeugen 318
Mysql
query, Methode 323
mysql, Client 318
Mysql, Klasse 323
select_db, Methode 323
mysql, Ruby-Erweiterung 321
Mysql::result
fetch_hash, Methode 324
fetch_row, Methode 324
num_rows, Methode 324
Mysql::Result, Klasse 323

N

Namensraum (OOP) 206
Nameserver 227
NAT 225
Nebenläufigkeit 256
beim Apache-Webserver 275
Forking 257
Threads 266
Nenner (Bruch) 146
Net::HTTP, Klasse 241
Antwort 242
get, Methode 242
Net::HTTPOK, Klasse 242
body, Methode 242
code, Methode 242
message, Methode 242

Netzwerk
  Routing 222, 226
  Schichtenmodell 220
  Sockets 229
Netzwerkgrundlagen 219
new, Konstruktoraufruf 114
nil 45
Nonblocking I/O 234
not, Operator 56
Notepad, Editor 17
now, Methode 138
nslookup 227
num_rows, Methode (Mysql) 324
numerator, Methode 146
Numerische Literale 36

O

Object, Klasse
  Referenz 367
object_id, Methode 212
ObjectSpace, Modul 212
Objekt-ID 212
Objektorientierte Programmierung (OOP)
    siehe Objektorientierung
Objektorientierung
  Design Patterns 205
  Geschichte 153
  Grundbegriffe 111
  HAS-A-Beziehung 165
  Instanz (Definition) 164
  Instanziierung 168
  IS-A-Beziehung 165
  Kapselung 154
  Klasse (Definition) 164
  Klassen 113
  Klassenentwurf 163
  Konstruktor 113
  Konstruktor (Definition) 167
  Mehrfachvererbung 207
  Methoden 113
  Methoden (Definition) 164, 169
  Module 205
  Namensraum 206
  Singleton 205
  Übersicht 111
  UML-Diagramme 164
  Vererbung 113, 198
  Vergleich m. imperativer Prog. 154
  Zugriffsschutz 201
offset, Methode 148
Oktalzahlen 39
OOP *siehe* Objektorientierung
Open Source 2
open, Methode 127
Operationen 50
  arithmetische 51
  bitweise 56
  logische 55
  Rangfolge 62
  sonstige 60
  Vergleiche 52
Operatoren
  - 51
  ! 56
  != 53
  !~ 88
  %Q 43
  %q 43
  %w 49
  & 56
  && 55
  * (arithmetisch) 51
  * (Strings) 61
  ** 52
  *= 60
  + (arithmetisch) 51
  + (Strings) 61
  += 60
  / 51
  /= 60
  < 53
  << (bitweise) 57
  <<. HIER-Dokumente 44
  <= 54
  <=> 55
  -= 60
  == 52
  === 54
  =~ 86, 97
  > 53
  >= 54
  >> 57
  ?, Zeichencode 68
  ?: 61
  ^ 57
  | 56
  || 55
  als Methoden 188

and 55
eigene 188
not 56
or 55
Rangfolge 62
Operatoren, % 51
Operatoren, < (Vererbung) 198
Options, Apache-Webserver 282
or, Operator 55
Order, Apache-Webserver 283
OSI-Referenzmodell 221

P

param, Methode 297
Parameter 170
Arrays als 172
dynamische 174
Hashes als 173
Standardwerte 171
Parametervariablen *siehe* Parameter
Parent-Prozess 257
PID ermitteln 262
parse, Methode (Time) 142
Parser 254
PATH, Umgebungsvariabe
für Ruby ergänzen 9
PATHEXT, Umgebungsvariable
(Windows) 35
Patterns *siehe* Design Patterns
Perl 2
Pickaxe Book (Programming Ruby) 2
PID *siehe* Prozess-ID
ping 226
Pipe 115
pop, Methode 68
Port (TCP/UDP) 228
POSIX-Signale 261
POST, HTTP-Methode 284
post_match, Methode 148
ppid, Methode 262
pre_match, Methode 148
Principle of Least Surprise 4
print, Methode 116
printf, Methode 117
Formate 117
private 201
Beispiel 202
Klassenmethoden, Abgrenzung 202
Process.kill, Methode 262
Process.ppid, Methode 262
Process.wait, Methode 259
Prompt 11
protected 201
Prozesse verwalten 256
Forking 257
Kommunikation zwischen Prozessen 260
Reaping 259
Signale 260
Zombies 259
Prozess-ID 258
des Parents 262
public 201
push, Methode 68
puts, Methode 116
Python 2

Q

Quadratwurzel 65
Quantifizierer (Regexp) 90
query, Methode 323
Query-String 291, 309
Quoting-Operatoren 43

R

radio_group, Methode 302
Rails 333
Action Mailer 336
Action Pack 335
Action Web Service 336
Active Record 335
Active Record-Grundlagen 348
Active Support 336
ActiveRecord::Base, Basisklasse 346
Ajax-Unterstützung 336
Anwendung erzeugen 337
ApplicationController, Basisklasse 339
Architektur 333
console, Skript 348
Controller-Methoden schreiben 350
database.yml, Konfigurationsdatei 344
Datenbank vorbereiten 343
datenbankbasiertes Beispiel 341
Deployment 354
erstes Beispiel 338
eRuby-Templates 340
generate, Skript 339, 346
Installation 336

Instanzvariablen zum
Datenaustausch 340
irb einsetzen für 348
Komponenten 335
MVC 333
Scaffolding 347
Templates 340
Veröffentlichung 354
Verzeichnisstruktur einer
Anwendung 341
rails, Konsolen-Tool 337
Rails, WEBrick-Testserver 337
RAILS_ENV, Umgebungsvariable 354
raise 211
rand, Methode 63
Ranges (Bereiche) 45
Rangfolge der Operatoren 62
Rational, Klasse 145
denominator 146
numerator, Methode 146
Referenz 368
Raumfolgearithmetik 38
RDoc 34
read, Methode 121
read, Methode (Dir) 137
Reaping (Prozessverwaltung) 259
kontrolliertes 260
rechner.rb, Skript 24
rechteck.rb, Skript 112
Referenz 357
Reflexion *siehe* Introspektion
Regex 85
Regexp 85
Regexp, Klasse 146
match, Methode 147
Referenz 368
Reguläre Ausdrücke 85
$, Endmarke 94
$-Treffervariablen 98
%r, Operator 86
*, Quantifizierer 92
+, Quantifizierer 93
., Platzhalter 90
?, Quantifizierer 90
[ ] 88
\A, Anfangsmarke 95
\b, Wortgrenze 95
\B, Wortinneres 95
\D, Nichtziffer 97
\S, Nicht-Whitespace 97
\s, Whitespace 97
\W, Nichtwort-Zeichen 97
\w, Wortzeichen 97
\Z, Endmarke 95
^, Anfangsmarke 94
^, Zeichen-Ausschluss 89
{ }, Quantifizierer 94
|, Alternativen 96
Bereichsmarkierungen 94
einzelne Zeichen 86
Ersetzen 98
Escape-Sequnezen in 88
Gier (greed) 93
greed 93
Grundlagen 86
gruppieren 91
gsub, Methode 98
i, Modifier 96
in Variablen 89
m, Modifier 96
Matching 86, 97
Modifier 96
Numerische Quantifizierer 94
objektorientiert 146
POSIX-Zeichenklassen 97
Praxiseinsatz 97
Quantifizierer 90
sub, Methode 98
Syntax 87
Treffervariablen 98
x, Modifier 96
Zeichen ausschließen 89
Zeichenbereiche 88
Rekursion 190
Konzept 190
Verzeichnisse durchsuchen 192
Relationale Datenbanken 314
rename
Methode 179
Request (HTTP) 284
require, Methode 187
rescue 209
reset, Methode (CGI) 303
respond_to?, Methode 214
Response (HTTP) 285
return 175
reverse, Array-Methode 69
reverse, String-Methode 66

rewind, Methode 135
RFC 1123-Datum 139
rgb.rb, Skript 58
rgb_farbe.rb, Skript 181
RGB-Farben umrechnen 58
ri, Ruby-Hilfe 148
round, Methode 64
  Erweiterung 186
Routing 222, 226
Ruby
  Anweisungen 32
  Bezeichner 46
  Blöcke 101
  Datenbankzugriff 314
  Datum und Uhrzeit 138
  einfache Ausdrücke 14
  Einflüsse 2
  Einzelbefehle ausführen 15
  erstes Skript 17
  FreeRIDE, Entwicklungsumgebung 18
  Geschichte 1
  Installation 4
  Installation (UNIX) 7
  Installation, Windows 4
  Internationale Verbreitung 2
  Interpreter 3
  Iteratoren 101
  Kommandos ausführen 10
  Kommandozeilen-Argumente 122
  Kommentare 33
  Kontrollstrukturen 72
  Literale 36
  Lizenz 2
  Merkmale 3
  Name der Sprache 1
  PATH ergänzen 9
  Plattformen, unterstützte 3
  Principle of Least Surprise 4
  Referenz 357
  Skript ausführen 16
  Skripten ausführen ohne "ruby" 34
  Skripten importieren 187
  Syntax 31
  Variablen 45
  Verbreitungsschub durch Rails 2
Ruby on Rails 2
Ruby on Rails *siehe* Rails
ruby, Interpreter
  Direkteingabe 16
  Einzelbefehl ausführen 15
  impliziter Aufruf 34
  Skript ausführen 16
rubygems, Erweiterungsmanager 309, 322
rubygems-Erweiterungen
  mysql 321
  rails 336
  uuid 309
Ruby-Hilfe (ri) 148
Rückgabewerte 175
  Ausdrücke als 175
Rückgabwerte
  return 175
Runden auf n Nachkommastellen 65

S

Scaffolding (Rails) 347
Schichtenmodell 220
Schleifen 79
  for 83
  loop 81
  until 81
  while 79
Schlüssel (Hash) 50
SciTE, Editor 18
Scope (Gültigkeitsbereich) 47
ScriptAlias 287
search, Array-Erweiterung 177
sec, Methode 139
seek, Methode 135
SELECT, SQL-Anweisung 319
select_db, Methode 323
self 184
Server 232
Server, HTTP-Header 286
ServerAdmin 281
ServerName 281
ServerRoot 280
Sessions (HTTP) 308
  in CGI-Bibliothek 310
  manuell implementieren 309
Setter-Methoden
  Attribut= 196
Setter-Methoden (automatische) 195
Shebang 34
  bei CGI 287
Shell 10
shift, Methode 69
Short-Circuit-Verfahren 56
Sichtbarkeitsstufen *siehe* Zugriffsschutz (OOP)

SIGCLD, Signal 261
Signale 260
  abfangen 261
  senden 262
  SIGCLD 261
  SIGTERM 262
  SIGUSR1 262
  SIGUSR2 262
SIGTERM, Signal 262
SIGUSR1, Signal 262
SIGUSR2, Signal 262
sin, Math-Methode 65
Singleton 205
Skripten importieren 187
sleep, Methode 267
slice, Methode 69
Smalltalk 2
Sockets 229
  accept 233
  getservbyname 233
  Klassen 230
  lauschende (Server) 233
  Nonblocking 234
sort, Methode 69
sort, Methode (File) 136
Spezialliterale 44
Sprachreferenz 357
sprintf, Methode 119
SQL 318
  CREATE TABLE, Anweisung 318
  in Ruby-Skripten 323
  INSERT, Anweisung 319
  SELECT, Anweisung 319
sqrt, Math-Methode 65
Standardausgabe 115
Standardeingabe 114
Standardfehlerausgabe 115
Standardwerte für Parameter 171
Statuscodes (HTTP) 285
STDERR 115
STDIN 114
STDOUT 115
step, Iterator 104
strftime, Methode 140
strftime-Formate 141
String, Klasse
  -, Erweiterung 190
  [ ], Methode 67
  chomp, Methode 66
  chop, Methode 66
  chr, Methode 67
  downcase, Methode 66
  each_byte, Iterator 104
  gsub, Methode 98
  length, Methode 66
  Referenz 368
  reverse, Methode 66
  split, Methode 71
  sub, Methode 98
  swapcase, Methode 66
  upcase, Methode 66
String-Methoden 66
Strings 41
  Anführungszeichen 42
  Eingebettete Ausdrücke 42
  Escape-Sequenzen 42
  HIER-Dokumente 43
  konvertieren in 70
  Quoting-Operatoren 43
  verketten mit + 61
  vervielfachen mit * 61
  Zeilenumbruch entfernen 66
  Zerlegen 71
Stylesheets *siehe* CSS
sub, Methode 98
submit, Methode 303
Subnet Mask *siehe* Teilnetzmaske
sum, eigene Array-Erweiterung 188
super (Konstruktor d. Elternklasse) 199
swapcase, Methode 66
Symbole 45
Syntax von Ruby 31
Systemzeit auslesen 138

T

tan, Math-Methode 65
TCP *siehe* Transmission Control Protocol
TCP/IP 219
  Anwendungsprotokolle 241
  Kommunikationsablauf 221
  Nameserver 227
  NAT 225
  nslookup 227
  ping 226
  Routing 222, 226
  Schichtenmodell 220
  Sockets 229
  traceroute 226
  Transportprotokolle 227

TCPServer, Klasse 232
TCPSocket, Klasse 231
Teilnetzmaske 225
temperatur.rb, Skript 293
Templates (Rails) 340
Terminal 10
text_field, Methode 301
textarea, Methode 301
textblog.txt, Skript 130
Texteditoren, Überblick 17
TextPad, Editor 18
Thread, Klasse 266
    join, Methode 267
    sleep, Methode 267
Threading-Server 268
Threads 266
    verlangsamen 267
Time, Klasse 138
    day, Methode 139
    hour, Methode 139
    min, Methode 139
    month, Methode 139
    now, Methode 138
    parse, Methode 142
    Referenz 369
    sec, Methode 139
    strftime, Methode 140
    wday, Methode 139
    year, Methode 139
times, Iterator 104
to_a, Methode 72
to_f, Methode 70
to_i, Methode 70
to_s, Methode 70
traceroute 226
Transmission Control Protocol (TCP) 227
    Client 232
    Forking-Server 258
    Port 228
    Server 232
    Server, Forking 263
    TCPSocket 231
trap, Methode 261
true
    als Literal 44
Typisierung 207
    dynamische 47
Typumwandlung 70

U

UDP *siehe* User Datagram Protocol
Uhrzeit *siehe* Datum und Uhrzeit
Umgebungsvariablen 288
    CGI 289
    PATH 9
    PATHEXT (Windows) 35
    RAILS_ENV 354
UML (Unified Modeling Language) 164
Unified Modeling Language *siehe* UML
UNIX
    Apache installieren 277
    Dateisystem 11
    Konsolenbefehle 13
    MySQL installieren 316
    Prompt 11
    Ruby installieren 7
    Shebang 34
    Shell 10
    Terminalfenster 10
unless, Fallentscheidung 75
unshift, Methode 69
until, Schleife 81
upcase, Methode 66
update, Active Record-Methode 350
Uploads (Webformular) 303
upto, Iterator 104
User Datagram Protocol (UDP) 228
User-Agent, HTTP-Header 285
UUID 309
UUID, Klasse 309

V

Variablen 45
    als Parameter 170
    Arrays 48
    Bezeichner 46
    Datentypen 46
    Definition 45
    globale 47
    Gültigkeitsbereich 47
    Hashes 50
    in Blöcken 102
Vererbung
    Konzept 198
    private 201
    protected 201

public 201
super 199
Zugriffsschutz 201
Vererbung (OOP)
Einführung 113
Vergleichsoperationen 52
Veröffentlichung (Rails) 354
Verzeichnisse
rekursiv durchsuchen 192
Verzeichnisse auslesen 136
Verzeichnisse prüfen 138
Verzeichnisstruktur 11
vi, Editor 18
vim, Editor 18
Vorzeichen 52

W

Wahrheitswerte (true/false) 44
wait, Methode 259
wday, Methode 139
Web Services 336
Webanwedungen
mod_ruby 329
Webanwendungen
Datenbankzugriff 314
Webbrowser, Projekt 245
Erläuterungen 251
Implementierung 247
Test 251
Webformulare 290
Hidden-Felder 309
WEBrick-Webserver 337
Webserver 274
WEBrick 337
Weiterleitung (HTTP) 243
while, Schleife 79
Whitespace
in regulären Ausdrücken 97
Windows
Apache installieren 276
Dateisystem 11
Eingabeaufforderung starten 10
Konsolenbefehle 13
Laufwerk wechseln 12
MySQL installieren 315
Prompt 11
Ruby installieren 4
Winkelmaß in Bogenmaß umrechnen 65
Winsock 229
Wochentag, deutsch 140
Wortumbruch 246
wrappager.rb, Skript 246
Wrapping 246

X

x, Regexp-Modifier 96
XAMPP 274
Xerox PARC 153
XHTML, in CGI-Bibliothek 298

Y

YAML, Dateiformat 344
year, Methode 139
yield 176

Z

Zahlen 36
Basis konvertieren 70, 71
Zähler (Bruch) 146
Zeichenketten (Strings) 41
Zeilenumbruch
am String-Ende entfernen 66
Zeilenumbruch, Plattformunterschied 237
Zombie-Prozesse 259
Zufallsgenerator 63
zug_imp.rb, Skript 154
zug_oo.rb, Skript 158
Zugriffsschutz (OOP) 201
Beispiele 202
private 201
protected 201
public 201

Über den Autor

Sascha Kersken kam 1983 zum ersten Mal mit einem Computer in Berührung und hatte später das Glück, dieses langjährige Hobby zu seinem Beruf machen zu können. Er arbeitet als Fachbuchautor, Dozent und IT-Berater mit den Schwerpunkten UNIX-Serveranwendungen und Webentwicklung in Köln. Er ist Autor der Bücher *Praxiswissen Flash 8* und *Praktischer Einstieg in MySQL mit PHP* (O'Reilly) sowie *Handbuch für Fachinformatiker*, *Apache 2* und *SUSE Linux 10.x* (Galileo Press). Für den O'Reilly Verlag hat er außerdem die Titel *Java und XML*, *Java Enterprise in a Nutshell*, *DNS & BIND Kochbuch*, *Active Directory*, *Praxiswissen Dreamweaver* und *Ajax von Kopf bis Fuß* übersetzt bzw. mitübersetzt. Seine Freizeit verbringt er am liebsten mit seiner Frau und seinem Sohn oder mit guten Büchern.

Kolophon

Das Design der Reihe *O'Reillys Basics* wurde von Hanna Dyer entworfen, das Coverlayout dieses Buchs hat Henri Oreal gestaltet. Als Textschrift verwenden wir die Linotype Birka, die Überschriftenschrift ist die Adobe Myriad Condensed, und die Nichtproportionalschrift für Codes ist LucasFont's TheSansMono Condensed.