

Praxiswissen Ruby on Rails

- Schritt für Schritt zu leichtgewichtigen Web-Anwendungen
- Mashup-Programmierung mit Ruby on Rails
- Mit einer Einführung in Ruby



O'REILLY®

Denny Carl

Praxiswissen Ruby on Rails

Denny Carl

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

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

Die Darstellung eines Zylinderhuts im Zusammenhang mit dem Thema *Praxiswissen Ruby on Rails* ist ein Warenzeichen des O'Reilly Verlags.

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, Köln
Fachliche Unterstützung: Sascha Kersken, Köln
Korrektorat: Oliver Mosler, Köln
Satz: Tim Mergemeier, reemers publishing services gmbh, Krefeld; www.reemers.de
Umschlaggestaltung: Michael Oreal, Köln
Umschlagmotiv: Guido Bender, Dipl.-Designer, Wiesbaden; www.fuenfvorelf.de
Produktion: Andrea Miß, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-89721-476-7

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Damals	IX
1 Gestatten, Ruby	1
Es ist ein Rubin!	2
Warum denn ausgerechnet Ruby?	3
Programmieren kann so schön sein!	10
Ruby und Rails abfahrbereit machen	14
RadRails – der Ruby- und Rails-Hauptbahnhof	21
RubyGems, die Bahnpost	24
Zusammenfassung	26
2 Programmieren mit Ruby	27
Mit Ruby ist zu rechnen	28
Zeichen und Wunder: Strings	37
Variablen und Konstanten	48
Symbole	51
Ruby-Programm 1: Der Buchstabenzähler	52
Von von bis bis – Wertebereiche	58
Elementares über Arrays	61
Hash mich, ich bin ein assoziatives Array	67
Weichen stellen	72

Ruby-Programm 2: Der Ehe-Checker	78
Und noch 'ne Runde	82
Die Könige unter den Schleifen: Iteratoren	90
Methoden	99
Selbst gemachte Klassen	103
Ruby-Programm 3: Der Arbeitsplatzoptimierer	135
Zusammenfassung	150
3 Einsteigen, bitte	151
Ruby on Rails ist	152
Ruby und Rails auf Fachschienesisch	154
Ein Blick ins Kochbuch	164
Meine Verehrung, eure Exzellenz	171
Zusammenfassung	176
4 Ein Photoblog mit Rails	177
Photoblog? Ähm.....	178
Der Bilderrahmen	179
Der Herr sprach, es werde Picsblog	181
Hallo Datenbank, bitte melden!	182
Die Datei environment.rb	185
Beitrags-Model	186
Datenbank mit Migrationshintergrund	187
PostsController, übernehmen Sie!	192
Ansichtssache	194
Beiträge speichern	198
Rails' Kurzzeitgedächtnis: flash	201
Models erweitern	202
Alle Fotos im Überblick	207
Bessere Views durch Partials	209
Fotos seitenweise	211
Beiträge bearbeiten	213
Löschen von Beiträgen	217
Beiträge anzeigen	218
Die Startseite	219
Navigationsmenü	220
LoginGenerator: Zutritt nur bedingt gestattet	221
Kommentare erwünscht	226

Ganz frisch: die neuesten Fotos	237
Unser Picsblog soll schöner werden	239
Zusammenfassung	242
5 TVsendr – mit dem Eigenen sieht man besser	245
Erst anmelden, dann einschalten!	246
TVsendr erzeugen	248
Model Broadcast	254
Antenne ausrichten	256
Wunderwerkzeug Rails-Console	265
Informationen zur Sendung	267
Das Grundlayout	268
Auf dem Weg zum Programmchef	270
Programmvorschau	281
Eine Sendung ins Programm nehmen	285
Sendungen aus dem Programm streichen	289
Den Programmablauf festhalten	290
Feinschliff im Programmeditor	291
Wir bauen uns einen Fernseher	293
Ein TV-Gerät mit Style	297
Abspann	301
Zusammenfassung	302
6 Rails-Anwendungen veröffentlichen	303
Bevor es losgeht	304
Ruby und Rails installieren	305
FastCGI installieren	307
Eine Anwendung auf dem Server einrichten	308
Auf dem Weg zum Profi	313
Zusammenfassung	313
Anhang	315
Index	341

Damals

Ich gestehe: Wenn es in der Vergangenheit darum ging, eine dynamische Website zu erstellen, war ich nicht immer unbedingt ein Vorbild in Sachen Ordnung und Struktur. Mit Grausen schaue ich mir heute so manchen Quelltext von dynamischen Websites an, die ich in früheren Tagen und meist auf Basis von PHP entwickelt habe. Wahrlich, kein schöner Anblick. Vor allen Dingen dann, wenn ich Dinge in HTML-Dateien finde, die da einfach nicht hingehören. Der Teil der Webseite, der für das Anzeigen von Daten zuständig ist, teilt sich so manches mal die Datei mit Code, der eher für die Bereitstellung eben dieser Daten verantwortlich ist. Selbstverständlich weiß ich, dass es kein guter Stil ist, Programm- und Anzeigelogik so zu verknoten.

Es lag sicher nicht daran, dass ich es gefühlsmäßig nicht überwunden hätte, die beiden zu trennen. Die Tatsache, dass sie eng umschlungen in diversen HTML- und PHP-Dateien vertreten waren, lag meist eher darin begründet, dass ich einfach keine Zeit mehr hatte, für eine geordnete Struktur zu sorgen. Meist befand sich der vereinbarte Fertigstellungstermin bereits in unmittelbarer Nähe. Und da ich meist viele Stunden damit verbrachte, in obskuren SQL-Statements drei Datenbanktabellen left und right zu joinen, was selten im ersten Anlauf gelang, verstrich die Zeit wie im Fluge.

Und selbst wenn noch Zeit gewesen war, dann mussten erst noch Verzeichnisse und diverse Dateien erstellt werden, Dateien mussten in andere eingebunden werden, für eine spezielle Funktionalität musste das Internet erst noch nach einer passenden Bibliothek durchforstet werden, die sich am Ende als gar nicht so passend entpuppte, trotzdem man diverse Konfigurationsdateien für sie geschrieben hat und

so weiter, und so weiter. Nein, Spaß hat das nicht gemacht. Am Ende hat zwar alles immer irgendwie funktioniert, aber das Entwicklerherz blutete – spätestens bei der nächsten Erweiterung der Website, was es erforderlich machte, sich wieder in das teuflische Quelltextchaos reinzudenken.

Aber das alles ist längst vorbei. Seitdem ich Ruby on Rails benutze, läuft alles in geordneten Bahnen. Ich brauche mich nur noch auf das wirklich Wichtige beim Entwickeln von dynamischen Websites zu konzentrieren. Alles andere macht Rails selbst.

Nun werde ich fast schon dazu gezwungen, Inhalt von Layout zu trennen, denn anders geht es gar nicht mehr. Und da Ruby on Rails für mich die gesamte Verzeichnisstruktur meiner Anwendung erstellt und die nötigen Dateien mit grundlegendem Quelltext befüllt zur Verfügung stellt, ist das auch gar kein Problem. Da geht keine Zeit mehr verloren, eher im Gegenteil. Und weil Ruby absolut auf objektorientierte Programmierung setzt, ist der gesamte Quelltext in kleine, leicht zu pflegende Schäcktelchen verpackt.

Eine Verbindung zur Datenbank aufnehmen? Nicht mehr meine Baustelle! SQL-Statements formulieren? Nur noch, wenn ich es selbst möchte. Code schreiben, der das Ergebnis einer Datenbankabfrage so umwandelt, dass man es auch verwenden kann? Nein, da habe ich besseres zu tun. Zum Beispiel kann ich mit der gewonnenen Zeit eine Website mit Ajax-Funktionalität erweitern. Durch die durchdachte, wegweisende Integration dieser modernen Technologie in Ruby on Rails ist auch das im Handumdrehen erledigt - ohne dass sich mein Kopf auf JavaScript umstellen muss. Ruby on Rails übersetzt das für mich.

Und so könnte ich noch viele Vorteile aufzählen, die Ruby on Rails Ihnen und mir bietet. Das mache ich auch - allerdings erst auf den folgenden Seiten dieses Buches. Beim Kennenlernen und Ausprobieren von Ruby on Rails und bei der Arbeit mit diesem Buch wünsche ich Ihnen viel Freude und Erfolg.

Denny Carl

Zielgruppe

Dieses Buch ist grundsätzlich für alle Leser gedacht, die Spaß daran haben, selbst an der Entwicklung des Internets teilzunehmen, sich mit den unendlichen Möglichkeiten von Web 2.0 auseinanderzustzen und sich in ein aufregendes Thema stürzen möchten, der Webentwicklung mit Ruby on Rails. Klar, ein wenig Erfahrung im Erstellen von Websites und im serverseitiger Programmieren wäre wahrlich kein Nachteil für eine harmonische Leser-Buch-Beziehung. Wenn Sie beispielsweise schon einmal ein paar Zeilen PHP geschrieben haben oder mit Perl und Python experimentiert haben, ist dieses Buch genau das Richtige für Sie. Denn mit Ruby und Ruby on Rails werden Sie erfahren, wie schön serverseitige Programmierung wirklich sein kann.

Aber ich sage Ihnen auch: Selbst wenn Sie noch nie eigene Programme für Ihren Computer oder einen Webserver geschrieben haben, werden Sie mit diesem Buch zurechtkommen. Vielleicht müssen Sie den ein oder anderen Satz zweimal lesen. Doch der Erfolg wird Sie sicher belohnen. Viele sagen, Ruby ist eine gute, erste Programmiersprache. Und dem möchte ich mich anschließen.

Ich habe mich bemüht, die verwendeten Quelltexte und erforderlichen Schritte ausführlich zu erläutern und darüber hinaus so manches Mal zusätzliche Informationen zu den verwendeten Techniken zu geben.

Genau wie Ruby on Rails selbst ist auch dieses Buch grundsätzlich betriebssystemunabhängig. Allerdings sind Schwerpunktthemen, die für Einsteiger wichtig sind, so geschrieben, dass besonders Windows-Nutzer sie Schritt für Schritt nachvollziehen können. Dabei handelt es sich aber meist eh um Themen, die Mac-OS- und Linux-User ohnehin meist im Schlaf beherrschen.

Was erwartet Sie in diesem Buch?

In sechs Kapiteln, die alle aufeinander aufbauen, erlesen Sie sich zunächst solide Basiskenntnisse für das Programmieren in Ruby. Darauf aufbauend geht es dann um das Framework für Webapplikationen, Ruby on Rails. Das macht doch Sinn, oder?

Dabei, und das verspreche ich Ihnen, bleibt es nie bei seitenlangen theoretischen und langweiligen Abhandlungen. Immer wieder werden Sie gefordert, wenn es darum geht, kleine Programme oder eine ausgewachsene Anwendung mit allem modernen Schnickschnack für's Web zu erstellen – selbstverständlich mit Hilfe ausführlicher Beschreibungen und Erklärungen.

Kapitel 1, *Gestatten, Ruby!*

Sie erhalten einen Überblick darüber, was Ruby eigentlich ist und was die Sprache so besonders macht. Sie erfahren, wie Sie Ihren Rechner fit für Programmierung mit Ruby und Ruby on Rails machen und welche Software Sie einsetzen sollten.

Kapitel 2, *Programmieren mit Ruby*

Im zweiten Kapitel steigen Sie direkt ein in die Entwicklung mit Ruby. Stück für Stück erweitert sich Ihr Können im Umgang mit Variablen, Kontrollstrukturen, Objekten und Klassen. Mittendrin: Drei Projekte, die das Erlesene praktisch umsetzen und die einzelnen Komponenten im Zusammenspiel zeigen.

Kapitel 3, *Einsteigen, bitte*

Das dritte Kapitel bereitet Sie langsam vor auf Ihre ersten Schritte mit Rails. Sie erfahren, warum Rails eigentlich Rails heißt, was sich hinter diversen Fachbegriffen wie Metaprogrammierung und Objektrelationales Mapping verbirgt und blicken hinter die Kulissen einer bereits bestehenden Rails-Applikation. Am Ende des Kapitels erstellen Sie Ihre erste eigene Webanwendung mit Ruby on Rails..

Kapitel 4, *Ein Photoblog mit Rails*

Dieses Kapitel zeigt Ihnen, wie Sie mit einfachen Mitteln eine Software entwickeln, mit der Sie ein eigenes Photoblog betreiben können. Hier werden Sie sehen, wie einfach es ist, eine datenbankgestützte Anwendung mit Ruby on Rails zu schreiben – selbst wenn dabei zwei Datenbanktabellen miteinander verbunden werden.

Kapitel 5, *Mit dem Eigenen sieht man besser*

Am Ende dieses Kapitels sind Sie Programmchef Ihres eigenen, browserbasierten TV-Senders. Mit Hilfe von Ruby on Rails und einem externen Webservice, den Sie per REST kontaktieren, ist das gar kein Problem.

Kapitel 6, *Rails-Anwendungen veröffentlichen*

Das letzte Kapitel des Buches zeigt Ihnen anhand der in Kapitel 5 entwickelten Anwendung, wie Sie vorgehen müssen, damit das ganze World Wide Web Ihre Rails-Applikation bewundern kann. Dabei werden Sie erfahren, wie Sie einen herkömmlichen virtuellen Root-Server zu einer Rails-Maschine erweitern können.

Anhang

Im Anhang des Buches finden Sie viele Informationen des Buches noch einmal gesammelt und an einer zentralen Stelle. Die Ruby- und die Ruby-on-Rails-Kurzreferenzen werden Ihnen helfen, wenn Sie Ihre erste, komplett selbst geschrieben Webanwendung mit dem Rails-Framework erstellen. Außerdem werden dort interessante Links zu Rails-Hostern und weiterführenden Internetseiten gelistet.

Die Website zum Buch

Alle Beispielskripte dieses Buches zum Ausprobieren und Herunterladen finden Sie im Internet auf der Website zum Buch, die Sie unter <http://www.praxiswissen-ruby-on-rails.de> erreichen. Dort finden Sie zudem ständig Neuigkeiten rund ums Thema Ruby on Rails und kleine Tipps und Tricks, die Ihnen helfen, noch bessere Rails-Anwendungen zu schreiben.

Geklammertes Ruby

Bevor es losgeht, noch ein wichtiger Hinweis: Es gibt grundsätzlich zwei Arten, wie Sie Ruby-Code notieren können. Dabei geht es um die Angabe von Parametern zu einer Methode. Diese können Sie mit und ohne Klammern notieren. Ein Beispiel:

```
# Ohne Klammern  
obj.set_image 20, picture_id, :png  
# Mit Klammern  
obj.set_image(20, picture_id, :png)
```

Ich habe mich für dieses Buch und für meine eigenen Entwicklungen gegen die klammerlose Variante entschieden. Aus folgenden Gründen: Für mich und meine Augen ist die zweite Variante viel überschaubarer. Und vielleicht spielt auch meine PHP-Vergangenheit - siehe oben - eine Rolle. Aber es gibt noch ein weiteres Argument, das meiner Meinung nach das stärkste Gewicht hat: Es gibt einige Ausnahmen und Fälle, in denen das Setzen von Klammern keine optionale, sondern eine obligatorische Angelegenheit ist. Wenn man aber von vornherein auf Klammern setzt, braucht man sich dafür nicht zu interessieren. Wichtig für Sie ist zu wissen, dass es beide Varianten gibt und weshalb viele Beispiele an anderer Stelle ohne Klammern dargestellt werden.

Schriftkonventionen

Damit Ihnen das Lesen und Verstehen leichter fällt, werden in diesem Buch folgende typografische Konventionen verwendet:

Kursivschrift

Wird für Datei- und Verzeichnisname, E-Mail-Adressen und URLs, aber auch bei der Definition neuer Fachbegriffe und für Hervorhebungen verwendet.

Nichtproportionalschrift

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

Nichtproportionalschrift fett

Wird für Stellen eines Quelltextes verwendet, an denen neue Codezeilen hinzugefügt wurden.

Nichtproportionalschrift kursiv

Wird für Stellen eines Quelltextes verwendet, an denen Code ersetzt wurde oder noch ersetzt werden muss.



Dieses Zeichen kennzeichnet einen Tipp oder einen generellen Hinweis mit nützlichen Zusatzinformationen zum Thema.



Dieses Zeichen kennzeichnet eine Warnung oder ein Thema, bei dem man Vorsicht walten lassen sollte.

Danksagungen

Vielleicht mag es etwas ungewöhnlich sein – aber der erste Dank, den ich gern loswerden möchte, gilt mir selbst. Ich danke mir, dass ich neugierig genug war, um Ruby on Rails zu erforschen und es für mich zu entdecken. Ja, das hab' ich wirklich gut gemacht. Ich danke mir auch, weil ich es so oft geschafft habe, selbst nach 20 Stunden Arbeit an diesem Buch und trotz völliger Übermüdung ohne Auto- oder Straßenbahnberührung den Weg vom Büro ins heimische Nest zu bewältigen, um selbiges vier Stunden später wieder zu verlassen. Ich konnte schließlich nicht zulassen, dass die Tastatur im Büro erkaltet.

Viel mehr danke ich aber all den Menschen, die es erst ermöglicht haben, dass ich mir selbst danken durfte. Allen voran danke ich dem O'Reilly-Verlag, weil er sich doch tatsächlich mit mir zusammen zum wiederholten Male in das Abenteuer Buchschreiben gestürzt hat. Ein besonderes Dankeschön in der Kategorie »Nervenstärke und ständige, rückendeckende Unterstützung« geht dabei natürlich an meinen Lektor, Volker Bombien.

Ebenfalls danken möchte ich meiner Herzallerliebsten Sabrina, da sie mal wieder über viele Wochen damit leben musste, mich nur sehr selten, zu höchst sonderbaren Uhrzeiten oder friedlich schlummernd zu Gesicht zu bekommen. Ich bin sehr glücklich, dass sie trotzdem immer zu mir hält und mich stets und ständig unterstützt, bei dem was ich mache, auch wenn das für sie und uns nicht immer einfach ist.

Auch dieses Buch möchte ich jemandem widmen. Genau genommen handelt es sich dabei um zwei Menschen, die sehr wichtig sind in meinem Leben, meine Eltern. Ihr habt mich während der ganzen Zeit, in der dieses Buch entstand, stets unterstützt, aufgemuntert und motiviert, obwohl ich dadurch wenig Zeit für euch hatte. Mein Dank dafür ist um so größer, wenn ich bedenke, dass ihr gerade wahrlich keine leichte Zeit durchlebt. Dass ihr trotzdem nicht den Mut verliert und stets versucht, das Beste daraus zu machen, dafür bewundere ich euch sehr. Und nicht zuletzt deshalb soll dieses Buch euch allein gewidmet sein.

Selbstverständlich gilt mein tiefer Dank auch Ihnen. Nicht nur, weil Sie momentan dem emotionalen Höhepunkt dieses Buches beiwohnen, sondern weil Sie ja vielleicht auch dem Rest des Buches Ihre Aufmerksamkeit schenken werden. Mich würde es sehr freuen.

Berlin, März 2007

Denny Carl

Gestatten, Ruby

In diesem Kapitel:

- Es ist ein Rubin!
- Warum denn ausgerechnet Ruby?
- Programmieren kann so schön sein!
- Ruby und Rails abfahrbereit machen
- RadRails – der Ruby- und Rails-Hauptbahnhof
- RubyGems, die Bahnpost
- Zusammenfassung

Haben Sie nicht mal wieder Lust, sich hemmungslos zu verlieben? Mal wieder Schmetterlinge im Bauch zu haben und den ganzen Tag mit klopfendem Herzen nur noch an Ihre neue Liebe zu denken? Oder wie wär's mit einer heißen Affäre, die das Wort Leidenschaft völlig neu definiert? Ich hätte da jemanden für Sie. Gestatten, *Ruby*.

Ruby ist die Programmiersprache, auf der das Framework *Ruby on Rails* basiert. Und das bedeutet: Möchten Sie mit Ruby on Rails in Windeseile tolle Webanwendungen erstellen, bleibt Ihnen nichts anderes übrig, als sich zunächst einmal mit Ruby vertraut zu machen. Aber das wird Ihnen leicht fallen. Denn Ruby lernen und anwenden macht einfach Spaß und zeitigt schnelle Erfolge.

Dieses Kapitel stellt Ihnen Ruby und einige seiner vielen Vorteile und entwicklerfreundlichen Eigenschaften vor. Sie erfahren zudem, worin sich Ruby von anderen vergleichbaren Programmiersprachen unterscheidet, und weshalb Ruby in letzter Zeit immer mehr Programmiererherzen im Sturm erobert hat und auch Ihres bald erobern wird. Ein paar Codezeilen, die Ihnen einen ersten Eindruck vermitteln, wie schön Programmieren mit Ruby sein kann, könnten dafür sorgen, dass das vielleicht schon auf den nächsten Seiten der Fall ist.

Außerdem haben Sie am Ende dieses Kapitels eine komplette Ruby-on-Rails-Rundum-Sorglos-Entwicklungsumgebung auf Ihrem Rechner, inklusive eines komfortablen Editors. Oder etwas anders ausgedrückt: Wir bauen gleich zusammen Ihr kleines schnuckliges Liebesnest, in dem Sie mit Ruby im 2. Kapitel ganz ungestört ein wenig turteln und später im 4. Kapitel mit Ruby on Rails so richtig zur Sache kommen können.

Es ist ein Rubin!

1993 begann Yukihiro Matsumoto, genannt Matz, eine tollkühne Vision in die Tat umzusetzen. Völlig entnervt von der aus seiner Sicht gänzlich unzureichenden, weil unlogischen und unsauberer Quelltexterei mit den Programmiersprachen *Perl* und *Python*, die damals im fernen Nippon en vogue waren, nahm er Papier und Bleistift und machte das, was wohl jeder von uns in einer solchen Situation tun würde: Er begann, seine eigene Programmiersprache zu entwickeln.

Konnichiwa, Ruby desu!

Und er hat tatsächlich zwei Jahre voller kühner Ideen und Herzblut durchgehalten und seine ganze Freizeit, die bei den meisten berufstätigen Japanern mit recht knapp wohl treffend charakterisiert ist, dafür geopfert. 1995 schlug die Geburtsstunde von Ruby; eine erste Version wurde veröffentlicht und unter das skeptische bis uninteressierte Programmierervolk geworfen. Damals wie heute als Open Source und frei erhältlich. Übrigens: Ruby selbst ist komplett in C geschrieben. Ja, auch Programmiersprachen müssen mit Programmiersprachen erst einmal programmiert werden.

Es dauerte nicht lange und es gab tatsächlich die ersten echten Ruby-Liebhaber, wenngleich anfangs besonders Japaner, also Matz' Landsleute, den üppigen Verlockungen von Ruby verfallen sind. In Japan wurde Ruby nach und nach zu einem großen Erfolg und stürzte den dortigen Marktführer Python, einst selbst Motivation für die Schöpfung von Ruby, bald vom Sockel. So kann's gehen.

Die erste Ruby-Dokumentation war, wie man vermuten kann, so japanisch wie der Ruby-Erfinder selbst. Das erschwerte die Verbreitung jenseits der Sushi-Demarkationslinie sehr, obwohl sie im Internet verfügbar war. Denn, sind wir doch mal ehrlich, wie steht es denn um Ihre Japanischkenntnisse, mal abgesehen von den Namen all der Köstlichkeiten Ihres örtlichen Sushi-Wirts?

Kaum war jedoch das erste englischsprachige Buch zum Thema auf dem Markt, verknallten sich auch die ersten Programmierer westlich von Tokyo, Utsunomiya und Kitakyushu in Ruby. Einige von ihnen verknallten sich so sehr, dass sie Ruby völlig neu und speziell für den Einsatz als Webserver-Programmiersprache einkleideten. Dabei ist *Ruby on Rails* herausgekommen, was Ruby einen weiteren weltweiten Popularitätsschub verschaffte und mit Sicherheit der Grund ist, weshalb auch Sie mit Ruby anbandeln möchten. Auf jeden Fall ist das ein sehr guter Grund.

Glitzern und Funkeln

Vielleicht fragen Sie sich, wie es zu der Bezeichnung Ruby kam? Nach eigenen Angaben von Matz lieferte Perl die Inspiration. Schließlich heißt *Perl* – eigentlich *Pearl*, aber diese Programmiersprache gab es schon – übersetzt »Perle«.

Der Name für Matz' neue Programmiersprache sollte noch einen Tick wertvoller und edler sein. Dass dabei die Wahl nicht auf Diamant oder Topas fiel, beruht allein auf der Tatsache, dass der Geburtsstein eines Kollegen der Rubin war, wodurch der Name gefunden war. Im Nachhinein ist Matz aufgefallen, dass Rubin sowohl im Alphabet als auch in der Reihenfolge der Geburtssteine *nach* der Perle kommt. Wenn das kein Grund ist, Ruby als legitimen Nachfolger von Perl auszurufen. Es gibt aber noch eine ganze Menge anderer und viel beeindruckenderer Tatsachen, die dies unterstreichen und Ruby zu einem ganz besonderen Edelstein unter all den Programmiersprachen, die es auf der Welt gibt, machen.

Warum denn ausgerechnet Ruby?

Vielleicht haben Sie bei Ihrer Entscheidung, sich mit *Ruby on Rails* zu beschäftigen, mit der Frage gehadert, ob es sich denn wirklich lohnt, extra dafür eine neue Programmiersprache zu erlernen. Möglicherweise sind Sie bereits recht fit in PHP oder Python und hatten eigentlich nicht vor, sich nun noch eine weitere Sprache anzueignen. In der Tat: All das, was Sie mit Ruby und Ruby on Rails zaubern können, klappt auch mit anderen Programmiersprachen und Frameworks. Und doch gibt es einige echte Argumente, die *für* das Erlernen von Ruby sprechen und die vielleicht auch Ihre letzten Zweifel wegwischen werden.

Einfach, schnell und elegant programmieren

Programmieren mit Ruby macht Spaß. Als die Arbeit an Ruby 1993 begann, waren schon einige Jahrzehnte mit Programmiersprachen aller Art, und mit ihnen gute und groteske Ansätze ins Land gegangen. Die Gnade der späten Geburt verhalf Ruby dazu, auf gewisse Erfahrungen mit etablierten Programmiersprachen zurückgreifen zu können. Sprich: Die guten, bewährten Prinzipien der Oldies wurden übernommen, meist sogar verbessert, und all die Absonderlichkeiten, die die tägliche Programmierarbeit erschweren, gänzlich außen vor gelassen oder brauchbar gemacht. Das Ergebnis dieser Aschenputtel-Methode ist, dass Ruby praktisch die Essenz der Sprachen *Smalltalk*, *Perl*, *Lisp*, *Eiffel*, *Scheme*, *CLU* sowie einiger Wünsche für die perfekte Programmiersprache darstellt.

Mit Ruby können Sie nicht bessere Programme schreiben oder Programme, die mehr können als solche, die in anderen Programmiersprachen erstellt wurden. Ruby sorgt jedoch dafür, dass Sie schneller, einfacher, logischer und mit viel weniger Frust arbeiten können und an Ihr Ziel kommen. Selten hat die Welt eine Pro-

grammiersprache gesehen, die so durchdacht, sauber, homogen und frei von syntaktischen Widersprüchen ist. Oft können Sie Ruby-Code so aufschreiben, als würden Sie einem guten Freund einen Brief schreiben. Durch die einfache, gut leserliche Syntax können auch fremde Entwickler schnell hinter die Funktionalität von Ruby-Code kommen.

Ruby gehorcht dem *Principle Of Least Surprise*, dem *Prinzip der geringsten Überraschung*, was Sie als Entwickler in praktisch jeder Zeile Ruby-Code merken können. Ruby-Syntax ist somit (nahezu) frei von Widersprüchen, Fallen und Sackgassen.

Die hauseigenen Ruby-Funktionen sind einfach zu bedienen und dennoch bisweilen außerordentlich mächtig und flexibel. Mit *Iteratoren* und *Code-Blöcken*, Themen, die wir später noch ausführlich behandeln werden, können Sie binnen weniger Zeilen Dinge in Ruby tun, für die Sie in anderen Sprachen kilobyteweise Schwerarbeit leisten müssen. Wenn Sie mit Ruby programmieren, können Sie sich auf das Wesentliche konzentrieren, nämlich auf das, was ihr Programm letztlich leisten soll. Mit Rubys Syntax müssen Sie sich jedenfalls nicht quälen. Die schreibt sich fast von allein.

Für Sie als Entwickler gibt es nur ein paar Regeln, an die Sie sich halten müssen. Ruby richtet sich lieber nach Ihnen – und Sie müssen sich weniger nach Ruby richten. An vielen Stellen haben Sie die Möglichkeit, einen Sachverhalt syntaktisch unterschiedlich auszudrücken. Und Sie entscheiden sich einfach für die Variante, die Ihnen am besten gefällt.

Vieles passiert im Hintergrund und belastet Sie gar nicht. Sie brauchen Variablen, die Sie in Ihren Ruby-Programmen nutzen möchten, nicht wie in C++ vor dem ersten Gebrauch mit ihrem jeweiligen Datentyp beim System anzumelden, sondern Sie schreiben sie einfach auf und weisen ihnen einen Wert zu. Auch das Abmelden einer Variablen ist bei Ruby nicht notwendig. Dafür bringt Ruby einen *Garbage Collector* mit. Dieser Datenmüllmann sorgt dafür, dass Speicherbereiche, die Ihr Ruby-Programm nicht mehr braucht, automatisch wieder freigegeben und zur Verfügung gestellt werden. Bei vielen anderen Sprachen sind Sie als Entwickler dafür verantwortlich. C-Entwickler verbringen beispielsweise manchmal Tage mit der hoffnungslosen Suche nach speicherbedingten Fehlern in ihren Programmen. Diese Probleme werden in Ihren Ruby-Projekten nicht auftreten, was die Programmierung um so vieles angenehmer macht.

Trotz dieser vielen komfortablen Eigenschaften ist Ruby keine Kleinkindersprache, sondern echter Profistoff, und genauso mächtig wie andere Sprachen, die den Entwickler jedoch oft zum Haareraufen ermuntern. Sie werden im Verlauf dieses Kapitels noch einige Beispiele in Augenschein nehmen können, die das eben Gelesene recht eindrucksvoll illustrieren werden.

Objekte, wohin das Auge blickt

Die Tatsache, dass Ruby einem teuren Abendkleid in Sachen Eleganz in nichts nachsteht, liegt zum Teil an der starken Verwendung des *objektorientierten Programmierparadigmas*. Das heißt, so gut wie alles, was in Ruby kreucht und fleucht und Daten enthält, ist ein *Objekt*. Selbst eine einfache Zahl ist in Ruby ein Objekt. In fast allen anderen Programmiersprachen kommen bei Zahlen Basistypen zum Einsatz, wobei andere, komplexere Datenstrukturen dort wiederum in Objekten gespeichert werden. Dieses uneinheitliche Mal-so-mal-so-Prinzip bei herkömmlichen Sprachen war einer der Hauptgründe für Matz, Ruby zu kreieren. Eine Sprache ist einfach logischer, klarer und leichter zu verstehen, wenn alles *objektorientiert* ist, und nicht nur ein Teil.

Sie können in Ruby auch größere Abschnitte prozedural programmieren, wenn Ihnen das leichter fällt oder Sie bislang nur so gearbeitet haben. Dass Sie dabei trotzdem objektorientierten Code schreiben, wenngleich Sie davon nichts merken, sei dabei aber erwähnt. All das, was Sie bei dem augenscheinlich prozuderalen Ansatz fabrizieren, seien es Funktionen oder globale Variablen, ordnet Ruby einem allgegenwärtigen Basisobjekt zu, auf das Sie aber nicht explizit zugreifen müssen, möchten Sie den prozeduralen Teil Ihrer Anwendung nutzen. So macht es Ruby Umsteigern und weniger geübten OOPlern einfacher, ohne dabei den Grundsatz der kompletten Objektorientierung zu verletzen.

OOP on Rails

Sollten Sie bislang noch nie oder nur wenig mit der *objektorientierten Programmierung (OOP)* zu tun gehabt haben, möchte ich Ihnen einige Vorteile dieser modernen Programmierweise kurz darstellen. Im Laufe des Buchs werden wir übrigens noch öfter auf das Thema OOP zurückkommen, denn es ist ein sehr wichtiges bei Ruby.

Ein wichtiger Punkt der OOP ist, dass Sie der menschlichen Denke sehr nahe kommt. Nehmen wir als Beispiel mal eine Lokomotive, die wir als Objekt in Ruby abbilden möchten. Dieses Stahl gewordene Kraftpaket hat spezielle *Eigenschaften* und *Fähigkeiten*, die eng mit der Lok verbunden sind und sie charakterisieren. Anders ausgedrückt: Sie *hat* oder *ist* etwas und sie *kann* etwas. Beispielhaft zeigt das Tabelle 1.1.

Tabelle 1-1: Eine objektorientierte Lokomotive

Eigenschaften	Fähigkeiten
Farbe	beschleunigen
Momentane Geschwindigkeit in km/h	bremsen
Höchstgeschwindigkeit in km/h	pfeifen
Leistung in PS	Fahrtrichtung umschalten
Vorhandene Kohlemenge in t	Kesseldruck ermitteln
Achsenzahl	Dampf ablassen
Baujahr	Sand streuen



Die Loks der Deutschen Bahn AG haben übrigens noch die zusätzliche und gern genutzte Fähigkeit *verspäten* und die Eigenschaften *Verspätung in Minuten* beziehungsweise *Verspätung in Stunden*.

Das beispielhafte Objekt *Lok* beinhaltet die aufgeführten Eigenschaften und Fähigkeiten; sie sind direkt im Lok-Objekt implementiert. Wenn Sie ein derartiges Objekt programmieren, dann legen Sie beispielsweise unter *beschleunigen* fest, welche Vorgänge in der Lok gestartet werden müssen und wie sich das auf die Eigenschaften auswirkt, wenn der schnaufende Koloss ein bisschen schneller fahren soll. So sorgt *beschleunigen* beispielsweise dafür, dass die Eigenschaft *Momentane Geschwindigkeit in km/h* angepasst wird.

Damit erreichen Sie aber noch mehr: Am Ende haben Sie ein Objekt erstellt, das nach außen hin abgeschlossen ist und alles beinhaltet, was die Lok zum Fahren braucht. Nicht weniger, aber auch nicht mehr. Die komplizierten Vorgänge zum Fahren einer Lok haben Sie mit dem Erstellen eines Objekts mit wenigen Fähigkeiten einfach gemacht. Was dabei intern passiert, spielt nach außen keine Rolle mehr. Das nennt man übrigens *Kapseln*. Etwas Kompliziertes haben sie in eine Kapsel (das Objekt) gesteckt und es auf diese Weise einfach gemacht, für sich selbst und auch für Dritte, die Ihr Objekt einsetzen möchten. Man nennt übrigens die Fähigkeiten, über die so eine Kapsel bedient wird, *Schnittstellen*. Sie können sie sich auch als Bedienelemente der Lok vorstellen. Der Lokführer braucht nicht wirklich zu wissen, was sich in der Lok tut, er muss nur Hebel bedienen und Knöpfe drücken.

Nun fährt so eine Lok ja eher seltener allein durch die Weltgeschichte. Gerade für den Transport von Personen und Gütern hat es sich als praktisch erwiesen, noch ein paar Wagons anzukuppeln. So ein Waggon stellt ein weiteres Objekt dar, das seine eigenen Eigenschaften und Fähigkeiten besitzt. Und wenn mehrere Wagons der Lok hinterherfahren sollen, so erstellen Sie einfach weitere Waggon-Objekte. Diese müssen Sie natürlich nicht mehrmals neu implementieren, sondern Sie erzeugen einfach *Duplikate*, die von einer Vorlage gebildet werden. Im Fachjargon gibt es dafür das Wort *Klasse*. Sie fungiert wie eine Schablone, ein Stempel oder eine Bau-

anleitung. Nach dem Vorbild einer Klasse entstehen Objekte, man spricht dabei auch von der *Instanz dieser Klasse*. Rein theoretisch können Sie so viele Kopien der Klasse erstellen, wie Sie wollen.

In einer fiktiven Waggonfabrik gibt es auch Klassen und daraus resultierende Objekte: Der frisch von der Uni gekommene Ingenieur entwickelt den Bauplan eines neuen, feschen Passagierwaggons mit luxuriösem Interieur. Nach diesem Plan werden umgehend 2.000 Einheiten gebaut. Anschließend fällt dem Ingenieur auf, dass er vergessen hat, Türen und Fenster in seinem Plan zu berücksichtigen. 2.000 nach diesem Plan erstellte Wagen (da sie ja Instanzen dieses Plans sind) werden umgehend dem Recycling zugeführt.

Aber zurück zu Lok und Wagen als *Objekte* beziehungsweise *Klassen* einer Programmiersprache. Nehmen wir an, Sie haben die Klasse *Lok* und die Klasse *Wagen* bereits erstellt. Nun können Sie (oder dank den klar definierten Schnittstellen auch bislang Unbeteiligte) sich an diesen Klassen bedienen und daraus je nach Bedarf Objekte erzeugen. So bauen Sie Ihr Programm praktisch aus einzelnen Modulen, die jeweils einen bestimmten Teil der Gesamtfunktionalität Ihrer Software abdecken, zusammen. Das macht das Programmieren besonders komplexer Programme letztlich recht übersichtlich. Außerdem können Sie dank den Schnittstellen und der strikten Eingrenzung der implementierten Funktionalität Ihre Klassen auch *wiederverwenden*. Braucht mal wieder jemand eine Lok, ist die Klasse bereits fertig.

Und noch etwas geht mit Klassen: *Erben*. Und zwar ohne dass der Tod einer anderen Klasse beweint werden muss. Lok und Wagen haben ein paar Gemeinsamkeiten. Beide rollen zum Beispiel farbig durch die Welt, haben eine gewisse Anzahl an Achsen und bremsen ab und zu. Sie könnten also ein Basis-Objekt *Schienenflitzer* basteln, das die Eigenschaften Farbe und Achsenanzahl sowie die Fähigkeit bremsen besitzt. Die Klassen *Lok* und *Waggon* könnten dann auf *Schienenflitzer* basieren und dessen Funktionalität *erben*.

Objektorientierte Programmierung macht das Coden einfacher, modularer und strukturierter. Doch es gibt auch einen etwas anspruchsvollerlen Aspekt dabei. Bevor Sie sich an das Erstellen eines Programms mit OOP heranwagen, sollten Sie sorgfältig planen, wie Sie die Problematik auf Objekte aufteilen, welche Funktionalität jedes Objekt erhalten soll, wie die Objekte möglicherweise miteinander zusammenhängen und so weiter. Meist reicht da aber schon der Einsatz einer kleinen Menge Gehirnschmalz im Vorfeld. Bei den Beispielprojekten dieses Buchs werde ich konkret auf diesen Schritt eingehen.

Soviel zunächst zum Thema Objektorientierung. Im 2. Kapitel werden Sie dem Thema zusammen mit konkreter Ruby-Syntax das nächste Mal begegnen. Spätestens dann werden Sie sicher die Vorteile der OOP erkennen. Zum jetzigen Zeitpunkt sollten Sie sich nur merken, dass das objektorientierte Programmieren eine Grundsäule des Ruby-Konzepts ist, und dass fast alles in Ruby ein Objekt ist. Ein Objekt wird nach einer Schablone (Klasse) gebildet und kapselt die zu diesem

Objekt gehörende Funktionalität, die durch Schnittstellen bedient, wiederverwendet und an andere Klassen vererbt werden kann. OOP strukturiert und modularisiert den Quellcode, bedarf aber einer gewissen Planung.

Vielseitig und überall verwendbar

Mit Ruby können Sie wirklich eine Menge machen. Ob es kleine Skripten sind, die etwa beim Start Ihres Systems aktiv werden, *Computerspiele* mit schneller 3D-Grafik, *Content Management Systeme* großer Firmen oder große *datenbankgestützte Softwareprojekte* mit grafischer Benutzeroberfläche – Ruby macht immer mit. Das trifft sowohl auf die Entwicklung von Desktop-Software als auch auf die von Anwendungen auf Webservern zu. Bei der zuletzt genannten Variante haben Sie beispielsweise die Möglichkeit, ganz normalen Ruby-Code in eine HTML-Datei zu schreiben, so, wie Sie es vielleicht von PHP oder ASP kennen. Der Webserver interpretiert diesen Code, bevor er die HTML-Seite an den anfordernden Browser schickt. Natürlich sei in diesem Zusammenhang auch *Ruby on Rails* erwähnt, das bei diesem Prinzip ansetzt und dem Rubin auf einem Webserver einen ganz fantastischen Glanz verleiht, wie Sie später noch sehen werden.

Ruby ist es egal, ob Sie lieber unter Linux, Windows, DOS, Mac OS oder anderen Betriebssystemen entwickeln. Schließlich gibt es Ruby für viele Plattformen. Dabei können Sie Ruby-Programme, die Sie unter Linux entwickeln, auch ganz einfach nach Windows portieren. Das liegt hauptsächlich daran, dass Ruby eine *interpretierte Sprache* ist. Ruby-Code wird erst in dem Moment durch den Ruby-Interpreter in die interne Sprache Ihres Rechners übersetzt, wenn Sie das Ruby-Programm ausführen. Ein zeitaufwändiges Kompilieren, wie Sie es vielleicht von C/C++ kennen, gibt es nicht. Allerdings muss erwähnt werden, dass das Live-Interpretieren von Quelltext gegenüber dem Ausführen kompilierter Software in etwas langsameren Programmabläufen resultiert, was bei den heutigen Prozessorleistungen aber kaum ins Gewicht fällt.

Der Vorteil dieses Prinzips überwiegt diesen Nachteil aber um einiges: Mit Ruby können Sie praktisch plattformunabhängig entwickeln. Es sei denn, Sie benutzen eine Bibliothek, die beispielsweise Windows-spezifischen Code enthält, um die Innereien eines Microsoft-Systems anzusteuern, oder Code für Linux-Oberflächen. Abgesehen davon laufen Ihre Ruby-Programme mühelos auf Mac OS, DOS, Windows, Linux, BSD, Solaris, UNIX und OS/2.

Editoren, Dokumentationen, Community

Um Ruby nutzen zu können, benötigen Sie kein prall gefülltes Konto. Ruby ist frei erhältlich und nutzbar. Selbst seinen Quellcode können Sie sich ansehen und erweitern oder anderweitig verändern. Ob für private oder kommerzielle Zwecke – die GPL-Lizenz, unter der Ruby steht, lässt jede Verwendung von Ruby als Interpreter

zu. Heißt aber auch: Sollten Sie den Code des Ruby-Interpreters selbst verändern, müssen Sie diese Veränderungen der Ruby-Gemeinde unter Verwendung der GPL zur Verfügung stellen.

Mit der Zeit erhöhte sich auch die Anzahl von nützlichen Werkzeugen, mit denen das Erstellen von Ruby-Code noch einfacher wurde. Besonders bei den Quelltext-Editoren gibt es mittlerweile eine ganz ansehnliche Auswahl. Einen recht mächtigen Vertreter, *RadRails*, werden Sie gleich noch genauer unter die Lupe nehmen.

Die Anzahl von Bibliotheken, die den ohnehin großen Funktionsumfang von Ruby noch durch spezifische Fähigkeiten erweitern, wächst ebenso stetig. So finden Sie Bibliotheken, mit denen Sie komplizierte Berechnungen einfach durchführen können, E-Mails von einem POP3-Server abrufen oder ansehnliche grafische Benutzeroberflächen zaubern. Und nicht zu vergessen: Auch *Ruby on Rails* ist so eine Bibliothek. Für die Installation solcher Bibliotheken gibt es ein spezielles Tool namens *RubyGems*. Der Ruby-eigene Paketmanager holt die Bibliothek Ihrer Wahl direkt aus dem Internet und macht sie umgehend für Sie verfügbar. Einfacher geht's kaum – wie Sie gleich feststellen werden, wenn Sie Ihr Ruby-und-Rails-System einrichten.

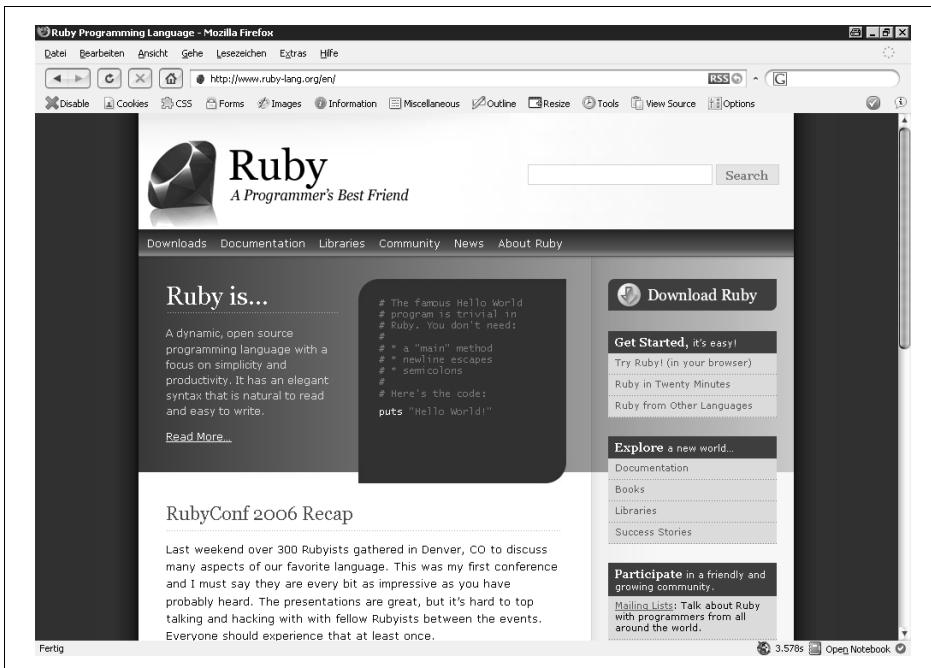


Abbildung 1-1: Die offizielle Ruby-Website

Die Dokumentation, die anfangs nur auf Japanisch vorlag, ist mittlerweile durch die ehrenamtliche Arbeit vieler Ruby-Verfallener in diversen Sprachen und äußerst ausführlich im Internet einsehbar. Eine Übersicht von lesenswerten Webseiten zum

Thema Ruby finden Sie im Anhang B. Aber hier und jetzt möchte ich Sie schon einmal auf die offizielle Ruby-Website im Internet verweisen. Sie ist unter <http://www.ruby-lang.org> zu finden und stets ein guter Anlaufpunkt für Informationen über aktuelle Ereignisse in der schnelllebigen Ruby-Welt oder Allgemeines rund um die Entwicklung mit Ruby. Unschlagbar ist auch die nette Ruby-Community, die täglich wächst und stets für jedes Ruby-Problem (sollte es denn mal wirklich eines geben) die richtige Anlaufstelle ist.

Auch der Buchhandel hält mittlerweile diverse Titel für Ruby-Interessierte bereit. Quasi die Bibel aller Ruby-Programmierer ist das als *Pickaxe* bekannt gewordene Buch *Programming Ruby* von Dave Thomas, Chad Fowler und Andy Hunt. Es enthält auch viele Dinge, auf die ich im Rahmen dieses Buchs nicht oder nur zu wenig eingehen kann. Wenn Sie also Ihren Ruby-Anwendungen den letzten Schliff geben möchten, sei Ihnen das über 800 Seiten starke Werk wärmstens empfohlen. Es hat schon vielen Ruby-Anfängern den Weg gewiesen. Außerdem hat Matz es mit einem persönlichen Vorwort geadelt.

Die jüngsten Entwicklungen in der Rubysphäre zeigen ganz klar, dass Ruby längst nicht mehr der Exot von einst ist, sondern sich immer mehr zur echten Alternative, insbesondere im Web-Bereich, entwickelt. Am besten, Sie werfen mal den ein oder anderen Blick auf einige Ruby-Beispiele, um eine Ahnung zu bekommen, warum das so ist.

Programmieren kann so schön sein!

Hereinspaziert in den Ruby-Showroom. Sie werden gleich ein paar Zeilen Ruby zu Gesicht bekommen, und ich werde Ihnen erklären, was genau diese Zeilen jeweils machen. Dabei ist es ganz normal, wenn Sie nicht alles davon verstehen. Schauen Sie mal auf den dicken Rest dieses Buchs, der muss ja schließlich auch noch irgendeinen Sinn haben. Aber vielleicht, ach was, ganz bestimmt springt ja bei Ihnen der Funke jetzt schon über und Sie bekommen eine Idee davon, wie einfach selbst etwas komplexere Probleme mit Ruby gelöst werden können. Sollten Sie schon Erfahrungen mit anderen Skriptsprachen haben, werden Sie umso beeindruckter sein. Vor allen Dingen dann, wenn Sie sich vor Augen führen, wie schwer die Umsetzung des gleichen Sachverhalts dort wäre.



Ich verrate Ihnen schon einmal, dass das #-Zeichen in Ruby den Beginn eines Kommentars darstellt. Kommentare werden gleich Erklärungen und Ergebnisse der Ruby-Codeschnipsel aufnehmen, so dass Sie schneller hinter den Sinn derselben kommen. Und noch etwas vorab: Die Befehle *puts* und *print* geben etwas auf dem Bildschirm aus.

Das erste Beispiel unserer Ruby-Schau zeigt Ihnen zwei Varianten, wie Sie zwei unterschiedlichen Variablen den gleichen Wert zuweisen können.

Beispiel 1-1: Zwei Variablen, ein Wert

```
# 1. Entweder separat  
a = 10  
b = 10  
# oder 2. verkettet:  
a = b = 10 # a=10 und b=10
```

Nur mal nebenbei: Sie wissen ja bereits, dass alles, was in Ruby Daten in sich trägt, ein Objekt ist. Die beiden Variablen *a* und *b* tragen Daten in sich, auch wenn es sich dabei jeweils nur um eine Zahl handelt. Aber das heißt auch: *a* und *b* sind *Objekte*. Im nächsten Beispiel soll *b* einen anderen Wert als *a* erhalten. Anschließend sollen *a* und *b* die Werte tauschen. In vielen Programmiersprachen muss nun eine Variable *c* her, um diese Funktionalität zu realisieren. Nicht so in Ruby!

Beispiel 1-2: Vertauschen durch Mehrfachzuweisung

```
a = 10  
b = 20  
a, b = b, a  
puts a # 20  
puts b # 10
```

Nehmen wir nun an, ein *Array* mit drei Werten kommt des Weges. Nun möchten wir drei unterschiedliche Variablen mit diesen Werten bestücken. Jede Variable soll einen Wert erhalten, der anschließend ausgegeben werden soll.

Beispiel 1-3: Dem Array-Inhalt Variablen zuweisen

```
erstens, zweitens, drittens = ["Affe", "Mensch", "Ruby-Programmierer"]  
puts erstens # Affe  
puts zweitens # Mensch  
puts drittens # Ruby-Programmierer
```

Ich kann an dieser Stelle ein weiteres Mal nicht widerstehen, auf die Objektorientierung von Ruby hinzuweisen. Das vorangegangene Beispiel enthält sieben Objekte: *erstens*, *zweitens*, *drittens*, "Affe", "Mensch", "Ruby-Programmierer" und – zugegeben, etwas schwer zu erkennen – auch den *Array*-Ausdruck, der durch die eckigen Klammern begrenzt ist. Glauben Sie nicht? Dann fragen wir doch einfach mal beim *Array* nach. Sie wissen bereits, ein Objekt wurde nach den Bauplänen einer Klasse erzeugt. Erkundigen wir uns also, nach welcher.

```
# 1. Variante  
puts ["Affe", "Mensch", "Ruby-Programmierer"].class # Array  
# 2. Variante  
evolution = ["Affe", "Mensch", "Ruby-Programmierer"]  
puts evolution.class # Array
```

Die Klasse *Array* stand also Pate bei der Erzeugung des Objekts, das Ruby ohne Aufsehen zu erregen mal eben selbst erstellt hat. Auch wenn wir das *Array* der Variablen *evolution* zuweisen, ändert sich nichts daran.

Um den Inhalt eines Arrays auszugeben, ohne vorher zusätzliche Variablen bemühen zu müssen, kennt Ruby beispielsweise *Iteratoren*. Ein mächtiges und doch einfache zu bedienendes Werkzeug, das Sie bald nicht mehr missen möchten. Der each-Iterator läuft das Array Schritt für Schritt durch. Für jedes Element wird der Teil zwischen den geschweiften Klammern ausgeführt und schritt mit dem jeweiligen Element bestückt.

Beispiel 1-4: Der each-Iterator

```
evolution.each { |schritt|
    puts schritt
}
# Affe Mensch Ruby-Programmierer
```

Weiter oben habe ich davon gesprochen, dass sich Ruby gern nach Ihren Wünschen richtet – und vielleicht gefällt Ihnen die eben angewandte Schreibweise ja nicht so besonders. Möglicherweise trifft die folgende eher Ihren Geschmack. An der Funktionalität ändert sich hier nichts.

```
evolution.each do |schritt|
    puts(schritt);
end
# Affe Mensch Ruby-Programmierer
```

Und noch eine Variante. Hierbei wird mit upto von 0 bis zu der Anzahl der Array-Elemente minus 1 gezählt. Allerdings büßt Ruby bei folgendem Beispiel ordentlich an Attraktivität ein. Daher mein Hinweis: zum Nachahmen nicht empfohlen – auch wenn es funktioniert. Hier verhält es sich wie im richtigen Leben: Rein theoretisch hat man Ihnen zwar die edle Sprache Goethes und Schillers beigebracht. Aber an der Currywurstausgabe lässt sich *Pommes Schranke* mit eher schlichterem Vokabular und frei von Versen doch viel einfacher bestellen. Trotzdem ist es Deutsch und man versteht Sie.

Beispiel 1-5: Zählen mit upto

```
0.upto(evolution.length - 1) { |index|
    puts evolution[index]
}
# Affe Mensch Ruby-Programmierer
```

Zählen ist eine große Stärke von Ruby, nicht nur, wenn es um die Anzahl von Array-Elementen geht. Wo andere Programmiersprachen einzig und allein for- und while-Schleifen anbieten, die Ruby natürlich auch kennt, gibt es hier weitere Möglichkeiten, ein Stück Quelltext mehrfach ausführen zu lassen. Den each-Iterator haben Sie eben auch schon in Aktion gesehen. Das folgende Beispiel preist Ruby in höchsten Tönen.

Beispiel 1-6: Dreimal hoch

```
print "Ruby lebe"  
3.times {  
    print " hoch"  
}  
puts "!"  
# Ruby lebe hoch hoch hoch!
```

Sie bemängeln die fehlenden Kommas zwischen den Jubelschreien? Kein Problem, sollen Sie haben. Ach, übrigens, Ihnen ist doch bestimmt aufgefallen, dass die 3 im vorhergehenden Beispiel ein Objekt ist, und dass sich .times genau auf dieses Objekt bezieht, oder?

Beispiel 1-7: Dreimal hoch, aber grammatisch richtig!

```
print "Ruby lebe"  
3.times { |index|  
    print " hoch"  
    print "," if index < 2  
}  
puts "!"  
# Ruby lebe hoch, hoch, hoch!
```

Ich möchte Ihr Augenmerk gern auf die Zeile lenken, in der mit print ein Komma gesetzt wird. Dieser Vorgang ist von einer Bedingung abhängig, die direkt *nach* der Anweisung folgt. Hier können Sie schreiben, wie Sie sprechen: *Nu' gib mal noch 'n Komma aus, aber nur, wenn der Index kleiner als 2 ist!*

Nun sollen aus einer Zeichenkette, die durch Kommas getrennte Zahlen enthält, alle geraden Zahlen herausgefiltert und im gleichen, kommahaltigen Format, aber absteigend sortiert ausgegeben werden. Die folgende einfache Variante geht dazu den Umweg über ein Array, das sich am einfachsten elementweise betrachten und filtern lässt.

Beispiel 1-8: Gerade Zahlen herauslösen und sortieren

```
numbers = "4, 2, 1, 3, 6, 5, 0"  
even_numbers = numbers.split(", ").delete_if { |number|  
    (number.to_i % 2).nonzero?  
}  
puts even_numbers.sort.reverse.join(", ")  
# 6, 4, 2, 0
```

Zum Abschluss dieser kleinen Ruby-Leistungsschau werfen wir noch einen Blick auf folgende Absonderlichkeit. Eine Zahl, in diesem Fall hat sich uns die 3 freundlicherweise nochmals zur Verfügung gestellt, kann man daraufhin überprüfen, ob sie eine Null ist. Rein mathematisch gesehen natürlich. Vier Möglichkeiten, dies zu prüfen, sollen dabei im Rampenlicht stehen.

Beispiel 1-9: 3 im Test

```
a = 3
puts a == 0                      # false
puts a.zero?                      # false
puts !a.nonzero?                  # false
puts a.eql?(0)                    # false
```

Treten wir nun die Gegenprobe an. Um dabei auf Nummer sicher zu gehen, prüfen wir gleich zweimal, ob denn die 0 auch 0 ist. Beim zweiten Durchlauf verschärfen wir sogar noch die Testsituation und prüfen 0.0.

Beispiel 1-10: 0 und 0.0 im Test

```
# 1. Testlauf
a = 3
puts a == 0                      # true
puts a.zero?                      # true
puts !a.nonzero?                  # true
puts a.eql?(0)                    # true
# 2. Testlauf
a = 0
puts a == 0                      # true
puts a.zero?                      # true
puts !a.nonzero?                  # true
puts a.eql?(0)                    # false
```

Ihren aufmerksamen Augen ist sicherlich nicht die letzte Auswertung entgangen. Sie bescheinigt offenbar, dass 0.0 nicht gleich (*equal*) 0 ist. Sollten Sie sich jetzt fragen, weshalb das so ist und was denn dieser Blödsinn soll, dann haben wir eine Situation erreicht, die im TV- und Film-Jargon gern als Cliffhanger betitelt wird. Im 2. Kapitel, in dem Sie nicht nur den Umgang mit Zahlen in Ruby ausführlich kennen lernen werden, geht es genau an dieser Stelle weiter. Außerdem stehen viele weitere Ruby-Aspekte im Rampenlicht, mit denen Sie bald Ihren eigenen, noch viel größeren Ruby>Showroom eröffnen können. Unser Rundgang hier ist aber beendet.

Ruby und Rails abfahrbereit machen

Ziel dieses Buchs ist es, Sie in Sachen Ruby und Ruby on Rails fit zu machen. Das verlieren wir auch nicht aus dem Auge, widmen uns aber kurz mal jemandem, der das sicher auch gern könnten würde. Denn was nützt es, wenn Sie in der Lage sind, ganze Office-Pakete in einer Zeile eleganten Ruby-Codes unterzubringen, wenn Ihr Rechner nicht versteht, was Sie von ihm wollen. Aber keine Sorge: Ihr Computer benötigt, um sich Ruby anzueignen, nur einen Bruchteil der Zeit, die Sie leider dafür aufwenden müssen.

InstantRails – Tüten-Rails für Windows

Ruby ist eine interpretierte Sprache. Das bedeutet, dass Sie beispielsweise Ruby-Code schreiben, in einer Datei mit der Endung `.rb` speichern und den Ruby-Interpreter beauftragen, Ihr Werk zu interpretieren. Und just in diesem Moment übersetzt der Ruby-Interpreter Ihren Ruby-Code in Maschinensprache. Für das Schreiben des Quelltextes reicht selbst der simpelste Editor. Es würde also genügen, wenn Sie sich den kostenlosen Ruby-Interpreter aus dem Internet laden und den Texteditor Ihres Vertrauens starten.

Damit wären Sie bestens gerüstet für Ihre ersten Gehversuche mit Ruby, die Sie im 2. Kapitel unternehmen werden. Aber für die Entwicklung mit Ruby on Rails reicht das bei weitem nicht aus.

Rails-Anwendungen können nur auf einem Webserver artgerecht gehalten werden. Möchten Sie also auf Ihrem Rechner mit Rails entwickeln, muss der zumindest so tun, als wäre er ein Webserver. Sie müssen also Webserver-Software installieren und dort Ruby und Ruby on Rails integrieren. Und damit nicht genug. Nur die wenigsten Webanwendungen kommen ohne Datenbank aus. Also: Datenbank und am besten noch ein Verwaltungsprogramm installieren. Und zum Schluss muss alles noch miteinander funktionieren – erkleckliche, ergebnisoffene Stunden mit diversen Konfigurationsdateien, Control Panels und tonnenweise Dokumentation liegen vor Ihnen, während deren Ihre Lust und Neugier auf Ruby und Rails jämmerlich verendet.

Wie machen Sie das, wenn Sie Hunger haben? So richtig Hunger. Sie sichten ja dann auch nicht erst stundenlang das Angebot der heimischen Rezeptsammlung, gehen dann einkaufen, besuchen alle Asia-Shops im Umkreis von 100 km auf der Suche nach der einen fehlenden exotischen Zutat, polieren das Tafelsilber und verbringen den Rest des Tages mit Serviettenfalten, oder? Nein, dann muss es schnell gehen. Und dann greifen doch auch Sie bestimmt ab und an zu den Spittenprodukten unserer Fertignahrungsindustrie, öffnen eine Bouillabaisse-Konserven oder rühren sich einen Rinderbraten aus der Tüte an. So etwas Ähnliches gibt es in Sachen Ruby und Rails auch – zumindest für Microsoft Windows: *InstantRails*.

Im Gegensatz zu den Convenience-Artikeln Ihres Vorratsschranks, bei denen die Zutaten schon mal zu Bauchschmerzen führen, kommt InstantRails nur mit feinsten Ingredientien, die perfekt aufeinander abgestimmt sind, gar wundervoll miteinander harmonieren, und Ihnen keinerlei Schmerzen bereiten.

Mit dabei ist natürlich der Ruby-Interpreter, der Paketmanager *RubyGems*, das komplette Rails-Framework, diverse weitere Bibliotheken, zwei kleine Editoren, der beliebte Webserver *Apache*, die nicht minder begehrte Datenbank *MySQL*, die Datenbankverwaltungssoftware *phpMyAdmin*, der auf Ruby spezialisierte Webserver *Mongrel*, zwei Beispieldatenbanken, eine Schaltzentrale und weiteres Zubehör. Und das Beste daran: Sie müssen nichts installieren, geschweige denn irgendetwas

konfigurieren. Damit folgt InstantRails dem grundlegenden Ruby-Prinzip: nicht lang überlegen, einfach machen und Zeit für wichtigere Sachen haben – zum Beispiel für das Erstellen Ihrer Ruby- oder Rails-Applikationen.

Um InstantRails einzusetzen (das sich zum Zeitpunkt des Entstehens dieses Buchs in Version 1.4 befindet), müssen Sie lediglich eine knapp 60 MByte große ZIP-Datei von der InstantRails-Projektseite <http://instantrails.rubyforge.org> herunterladen und entpacken.



Bitte achten Sie beim Entpacken darauf, dass das Zielverzeichnis keinerlei Leerzeichen enthält.

Belassen Sie beim Entpacken unbedingt die interne Verzeichnisstruktur, wodurch auch automatisch ein Verzeichnis *InstantRails* entsteht, in dem sich die ganze Pracht befindet. Wenn Sie also das Archiv nach C:\ entpacken, entsteht ein Verzeichnis C:*InstantRails*, worauf sich alles, was in diesem Buch noch folgt, stützt. Bitte beachten Sie das, wenn Sie einen abweichenden Verzeichnisnamen oder einen anderen Ort auswählen.

Starten Sie anschließend die Datei *InstantRails.exe* in C:*InstantRails*. Sie werden mit der Frage konfrontiert, ob die Konfigurationsdateien erneuert werden sollen, schließlich habe sich der Ort von InstantRails geändert. Beantworten Sie diese Frage mit OK und schließen Sie InstantRails wieder. Drücken Sie dazu die Taste F3, um die möglicherweise bereits automatisch gestarteten Anwendungen Apache und MySQL wieder zu stoppen, und schließen Sie dann das InstantRails-Fenster auf herkömmliche Weise.

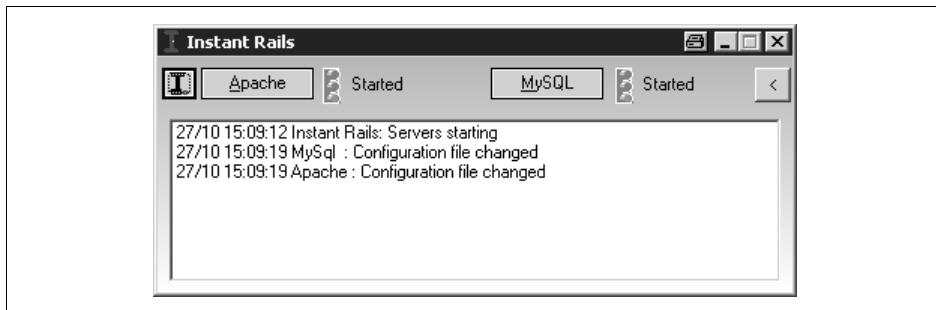


Abbildung 1-2: InstantRails, der erste Start

Im InstantRails-Verzeichnis befindet sich eine Datei namens *use_ruby.cmd*. Durch das Ausführen dieser Datei wird nun doch eine winzige Veränderung an Ihrem System vorgenommen, allerdings eine ganz harmlose. Es ist wichtig, dass Sie an jedem Ort Ihres Rechners den Ruby-Interpreter und seine Helfer ausführen können. Stel-

len Sie sich vor, Sie befinden sich in einem Projektverzeichnis und möchten eine Datei voller Ruby-Quelltext interpretieren, also ausführen lassen. Damit Sie nun nicht ins Ruby-Verzeichnis wechseln müssen, muss ein Eintrag in die Windows-Umgebungsvariable *Path* erfolgen. Und genau das übernimmt *use_ruby.cmd* für Sie. Somit trennt Sie nur noch dieser Doppelklick von einer kompletten und voll funktionsfähigen Ruby- und Rails-Installation auf Ihrem Windows-Rechner.



Sollten Sie jetzt oder im späteren Praxiseinsatz Probleme mit InstantRails' *Apache*- oder *MySQL-Server* haben, so prüfen Sie bitte, welche andere Server-Software auf Ihrem Rechner den Zugang verhindert. Das kann insbesondere ein *IIS (Internet Information Server)* sein, aber auch ein anderes Apache, ein Server-Komplettpaket wie XAMPP oder auch ein als Software implementierter Proxy-Server können ursächlich sein. Schalten Sie die Störenfriede am besten ab oder konfigurieren Sie diese entsprechend um. Beachten Sie dabei auch, dass viele Server unter Windows als Service laufen und daher im Hintergrund und eher unscheinbar agieren.

Nähere Informationen zu dieser Problematik gibt es auch auf der InstantRails-Website oder in den entsprechenden Foren von <http://www.rubyforge.org>.

Die meisten Schilderungen in diesem Buch gehen davon aus, dass Sie einen Windows-Rechner mit funktionstüchtigen InstantRails-Komponenten vor sich haben. Sollten Sie ein Betriebssystem nutzen, an dem Microsoft nicht mitarbeiten durfte, muss ich Ihnen zunächst eine traurige Nachricht mitteilen.

InstantRails für Mac OS

Als dieses Buch entstand, war eine InstantRails-Variante für *Linux*, *Mac OS* und andere Systeme in Planung – aber mehr auch nicht. Mac OS-User, sofern sie mindestens über Version 10.3 verfügen, können sich mit *Locomotive* behelfen, das einen ähnlichen Ansatz wie InstantRails verfolgt, aber teilweise aus anderen Komponenten besteht. Selbstverständlich sind Ruby und Rails dabei, doch statt *Apache* kommt *lighttpd* und statt *MySQL* ist *SQLite* mit dabei. Das macht sich aber kaum bemerkbar. Letztlich haben Sie mit Locomotive definitiv auch alles, was Sie für den schnellen, problemlosen Einstieg in die Ruby- und die Rails-Entwicklung benötigen. Über die Projektseite <http://locomotive.raaum.org> gelangen Sie zum Download einer *.dwg*-Datei, die alles nötige enthält.



Neuere Versionen des Apple-Betriebssystems Mac OS X beinhalten Ruby standardmäßig, allerdings nicht in der aktuellsten Version. In naher Zukunft ist sogar geplant, auch Ruby on Rails bei der Installation von Mac OS zu berücksichtigen. Nutzen Sie dennoch ein Angebot wie Locomotive, um wirklich alles Nötige für die Programmierung mit Rails in aktueller Form auf dem Rechner zu haben.

Sie sollten darüber hinaus in Betracht ziehen, an Ihre Lokomotive noch MySQL anzuhängen. Die Unterschiede in der Entwicklung von Rails-Anwendungen mit MySQL beziehungsweise SQLite sind zwar nur marginal, aber es gibt sie.

Freunde von Linux haben es bei der Installation von Ruby und besonders von Rails und den nötigen Komponenten am schwierigsten. Es gibt momentan keine vorgefertigte Komplett-und-sofort-Lösung wie *InstantRails* oder *Locomotive*. Handarbeit ist gefragt – aber das sind Linuxianer ja durchaus gewöhnt. Und meistens wollen sie es ja auch gar nicht anders.

Rails mit Linux

Nachfolgend möchte ich Ihnen ein paar Hinweise geben, wie Sie Ihr Linux um Ruby on Rails erweitern können – am Beispiel des Debian-Derivats *Ubuntu* in der Version 6.10. Sollten Sie *Kubuntu*, *Xubuntu* oder ein anderes Linux-System in Betrieb haben, das ebenfalls auf *Debian* basiert und über einen Desktop wie *Gnome* oder *KDE* verfügt, dann sollte die folgende Anleitung ebenfalls für Sie nützlich sein. Sie werden nur kleine Unterschiede feststellen.



Möchten Sie Ruby on Rails mit dem entwicklerfreundlichen Linux zunächst nur mal ausprobieren, so möchte ich Ihnen die *Rails-LiveCD* ans Herz legen. Dabei handelt es sich um ein auf *PCLinux* basierendes Linux-Live-System, welches nicht installiert werden muss. Nach dem Booten von RailsLiveCD haben Sie ein komplettes Rails-System vor sich. Auf der Projekt-Website <http://www.rails-livecd.org> gibt es mittlerweile sogar eine deutsche ISO-Datei, die Sie mit fast jedem gängigen Brennprogramm in eine bootfähige CD verwandeln können.

Informationen zur Schaffung eines kompletten Rails-Entwicklungsrechners auf Basis anderer Linux-Distributionen wie *openSUSE*, *Mandriva* oder *Fedora* finden Sie massenhaft im Netz. Einfach nach Rails und dem Namen Ihres Systems suchen und Sie werden umfangreich fündig. Auf der Website zum Buch (<http://www.praxis-wissen-ruby-on-rails.de>) finden Sie ebenfalls eine umfangreiche Liste.

Ruby installieren

Grundsätzlich sollten Sie zunächst dafür sorgen, dass Ihr System über alle aktuellen Updates verfügt. Öffnen Sie anschließend via *System → Administration → Software Quellen* ein Auswahldialog, in dem Sie Orte auswählen können aus denen der Paketmanager Daten holen soll. Sorgen Sie hier dafür, dass auch die Quellen *universe* und *multiverse* aktiviert sind. Schließlich sind dort wichtige Softwarepakete für Ihre Rails-Installation enthalten. Lassen Sie Ubuntu anschließend eine Aktualisierung der Quelleninformationen vornehmen, indem Sie der per Dialogfenster an Sie herangetragenen Bitte entsprechen.

Noch ein Hinweis zur nun folgenden Installation der einzelnen Komponenten. Beachten Sie, dass Sie viele Aktionen bei Ubuntu nur mit Superuser-Rechten ausführen dürfen, die Sie durch das Voranstellen des Befehls `sudo` erreichen.

Zunächst sollten Sie *Ruby* installieren. Öffnen Sie dazu ein Terminal. Mit dem Paketmanager *apt* können Sie sich nun alles aus dem Internet an Bord holen, was benötigt wird, um mit Ruby zu programmieren. Nicht nur bei diesem ersten Schritt werden eine ganze Menge an Daten auf Ihrem Rechner geschaufelt. Nehmen Sie sich also ein bisschen Zeit.

```
sudo apt-get install ruby ruby1.8-dev ri rdoc irb
```

Die Fragen, die Ihnen bei diesem Vorgang möglicherweise gestellt werden, können Sie grundsätzlich mit `J` beantworten. Anschließend können Sie mit der Eingabe von `ruby -v` prüfen, ob die Installation wirklich erfolgreich war. Das ist dann der Fall, wenn Sie sogleich die Versionsnummer der installierten Ruby-Version angezeigt bekommen.

RubyGems installieren

Mit der Installation von Rubys eigenem Paketmanager *RubyGems*, um den es gleich auch noch ausführlicher gehen wird, leisten Sie den ersten wichtigen Schritt auf dem Weg zu Ruby on Rails on Ubuntu.

Da RubyGems in den Ubuntu-Quellen nicht vorliegt, gestaltet sich die Installation etwas komplizierter. Starten Sie Ihren Webbrowser und besuchen Sie die Website http://rubyforge.org/frs/?group_id=126&release_id=9074. Dort finden Sie diverse RubyGems-Installationsdateien, wobei Sie Ihr Augenmerk auf `.tgz`-Archive richten sollten.

Finden Sie nun heraus, wie der URL der aktuellsten Version lautet. Zum Zeitpunkt, da dieses Buch entsteht, ist 0.9.1 die aktuellste Version, zu erreichen unter <http://rubyforge.org/frs/download.php/16452/rubygems-0.9.1.tgz>. Mit dem Befehl `wget` können Sie in Ihrem Terminal diese Datei auf Ihren Rechner laden.

```
wget http://rubyforge.org/frs/download.php/16452/rubygems-0.9.1.tgz
```

Das heruntergeladene Archiv können Sie nun entpacken. Nutzen Sie dafür am besten auch das Terminal; mit tar und der Angabe des Archives, hier `rubygems-0.9.1.tgz`, ist das schnell erledigt.

```
tar -xvzf rubygems-0.9.1.tgz
```

Dabei entsteht ein Verzeichnis, das ebenfalls abhängig von der Version benannt ist. Im vorliegenden Fall heißt es `rubygems-0.9.1`. Wechseln Sie in dieses und führen Sie das Setup aus, das übrigens eine waschechte Ruby-Datei ist und nur dank des eben installierten Ruby-Interpreters seine Dienste anbieten kann.

```
cd rubygems-0.9.1  
ruby setup.rb
```

Durch diesen Schritt wird RubyGems durch Ruby nutzbar, denn alle Dateien werden automatisch an den richtigen Ort kopiert. Den Erfolg dieser Aktion können Sie wieder mit der Abfrage der Versionsnummer, `gem -v`, überprüfen.

Nach dem Ausführen der Setup-Datei können Sie das heruntergeladene Archiv und das entpackte Verzeichnis wieder löschen.

MySQL installieren

Die Datenbank `MySQL`, die in den Beispielen dieses Buches zum Einsatz kommt, und eine Bibliothek, die Ruby-Programme und MySQL miteinander verbindet, installieren Sie wieder mit Hilfe von apt.

```
sudo apt-get install mysql-server mysql-client libmysql-ruby
```

Rails installieren

Nun können Sie das Rails-Framework installieren. RubyGems übernimmt das für Sie. Die folgende Zeile für Ihr Terminal fordert Rubys Paketmanager auf, alle Pakete, die Rails zum ordnungsgemäßen Betrieb braucht, zu installieren.

```
sudo gem install rails --include-dependencies
```

Nach der Installation, die möglicherweise mit einem Update der RubyGems-Daten beginnt und auch das Herunterladen von Dokumentationsdateien beinhaltet, können Sie den Erfolg ein weiteres Mal mit der Überprüfung der Version durch das Eingeben von `rails -v` testen.

Mongrel installieren

Während der Entwicklungs- und Testphase Ihrer Rails-Anwendungen laufen diese auf einem speziellen Server, der auf die Bedürfnisse eines Rails-Entwicklers zugeschnitten ist. In diesem Buch kommt dabei `Mongrel` zum Einsatz.

Mongrel kommt als Quelltext zu Ihnen. Da dieser nicht in Ruby – sondern in C – geschrieben ist und daher kompiliert werden muss, sollten Sie sich zunächst einen

entsprechenden Compiler zulegen. Diesen und für seinen Betrieb benötigte Software finden Sie im Ubuntu-Paket *build-essential*. Anschließend können Sie Mongrel via RubyGems laden und kompilieren.

```
sudo apt-get install build-essential  
sudo gem install mongrel --include-dependencies
```

RubyGems wird Sie mehrmals fragen, welche Version eines Gems Sie installieren möchten. Entscheiden Sie sich für die neueste Version, hinter der in Klammern *ruby* steht. Damit wählen Sie die auf Quelltext basierende Variante, die nach dem Herunterladen durch RubyGems kompiliert wird. Achten Sie während dieses Vorgangs auf eventuell auftretende Fehler, denn die sind einer funktionstüchtigen Mongrel-Installation sehr abträglich.

Mit Mongrel ist die letzte wichtige Komponente für den effektiven Einsatz von Ruby und Ruby on Rails auf Ubuntu installiert. Jetzt können Sie auch hier in die faszinierende Welt der Rails-Anwendungen einsteigen. Übrigens: Der Apache-Server, der bei InstantRails mitgeliefert wird, ist bei Ubuntu schon von Hause aus dabei. Er und der eben installierte MySQL-Server starten zudem automatisch beim Hochfahren Ihres Rechners. So können Sie stets sofort loslegen.

Egal, welches Betriebssystem Sie nutzen, es gibt einen Editor, der unter allen läuft und sich stetig wachsender Beliebtheit, besonders bei vielen Rails-Entwicklern erfreut.

RadRails – der Ruby- und Rails-Hauptbahnhof

InstantRails bringt zwei Editoren mit, *SciTE* und *FreeRIDE*. Sie finden die beiden Editoren in Windows unter *C:\InstantRails\ruby\scite* beziehungsweise *C:\InstantRails\ruby\freeride*. Grundsätzlich bringen die beiden jeweils einen Funktionsumfang mit, der völlig ausreicht, um Ruby-Dateien zu erstellen. Wenn es dann aber etwas komplexer wird, erst recht, wenn Ruby on Rails im Spiel ist, stoßen SciTE und FreeRIDE an ihre Grenzen. Daher legen ich Ihnen ans Herz, gleich von Anfang an mit einem echten Profi-Tool zu arbeiten: *RadRails*.

Die komplette Rails-Entwicklungsumgebung *RadRails* basiert auf der Programmierer-Allzweck-IDE *Eclipse*. Eclipse wiederum ist ein in Java geschriebenes Programm, wodurch sich schon die Plattformunabhängigkeit der Software andeutet. RadRails setzt Eclipse einfach ein paar zusätzliche Funktionalitäten auf, die das Entwickeln mit Ruby und Ruby on Rails noch ein Stück einfacher und komfortabler machen. So zum Beispiel

- *Syntax Highlighting* für Ruby- und Rails-Quelltexte
- *Code Folding*, um zusammenhängende Quelltextteile auszublenden und damit die Übersicht zu verbessern
- *Konsole*, die das Ergebnis eines Ruby-Programms ausgibt

- *Data Navigator*, der den Umgang mit Datenbanken enorm erleichtert, beispielsweise in Rails-Applikationen
- Einfaches *Generieren* von Rails-Komponenten
- Integration einer *Serverumgebung* wie Mongrel
- *Integrierter Browser* zum sofortigen Testen der Rails-Applikation
- *Code Templates* zum leichten Wiederverwenden von Ruby-Quelltexten

Sollte Eclipse bereits auf Ihrem Rechner laufen, reicht es, ein Plugin zu laden. Andernfalls gönnen Sie sich die Komplettinstallation und lassen Sie sich nicht von den über 44 MByte (Stand Version 0.7.2) großen Download abschrecken. Alle nötigen Informationen dazu und den Download-Link für Ihr Betriebssystem finden Sie unter <http://www.radrails.org>.



Bitte stellen Sie sicher, dass Ihr System bereits über Ruby und Ruby on Rails jeweils in aktuellen Versionen verfügt und ein aktuelles Java-Runtime-Environment, mindestens in Version 1.4.2, installiert ist. Andernfalls könnte es zu Problemen beim Ausführen von RadRails kommen.

Aktuelle Runtime-Dateien gibt es direkt bei Sun unter <http://www.java.com/de/download/manual.jsp>. Ubuntu- und andere Tux-Freunde können das Java Runtime Environment auch via sudo apt-get install sun-java5-jre erhalten.

Bei einer Neuinstallation laden Sie das ZIP-Archiv für Ihr Betriebssystem herunter und entpacken es in den Pfad Ihrer Wahl. Sie sehen, auch hier entfällt eine Installation. Dabei entsteht automatisch ein Verzeichnis namens *RadRails*. Sollten Sie sich unter Windows also für C:\ als Zielverzeichnis entscheiden, können Sie den Editor in C:\RadRails starten.

Wenn Sie genau das getan haben, fragt Sie RadRails zunächst nach Ihrem Work-space. Das ist der Ort, wo RadRails standardmäßig Ihre Ruby- und Ihre Rails-Projekte speichern wird. Sie sollten hier das Verzeichnis C:\InstantRails\rails_apps angeben, wenn Ihr Betriebssystem Windows heißt. In diesem Fall wird RadRails automatisch mit dem Ort, an dem sich Ruby, Rails und Mongrel aufhalten, versorgt, womit die Konfiguration der Ruby-IDE bereits abgeschlossen ist.

Bei anderen Betriebssystemen müssen Sie die Konfiguration manuell vornehmen. Nachdem Sie das Arbeitsverzeichnis Ihrer Wahl angegeben haben, klicken Sie auf *Window → Preferences... → Ruby → Installed Interpreters*. Klicken Sie anschließend auf *Add* und geben Sie als *Location* den Aufenthaltsort Ihres Ruby-Interpreters an. Unter Mac OS und Linux ist dies meist /usr/local/bin/ruby oder /usr/bin/ruby.

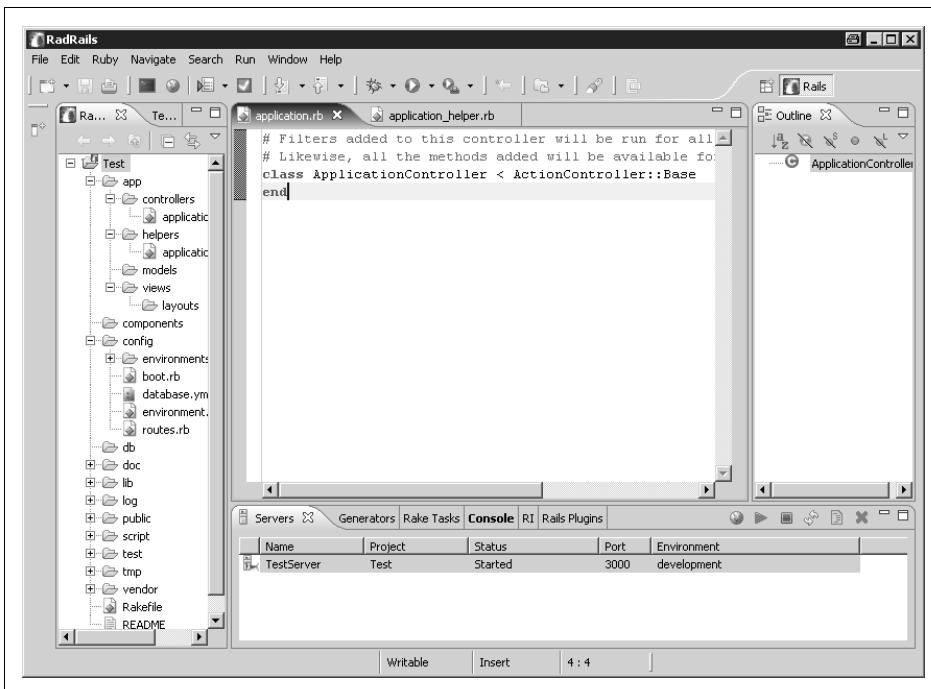


Abbildung 1-3: RadRails ist ein Allesköpper in Sachen Ruby- und Rails-Entwicklung

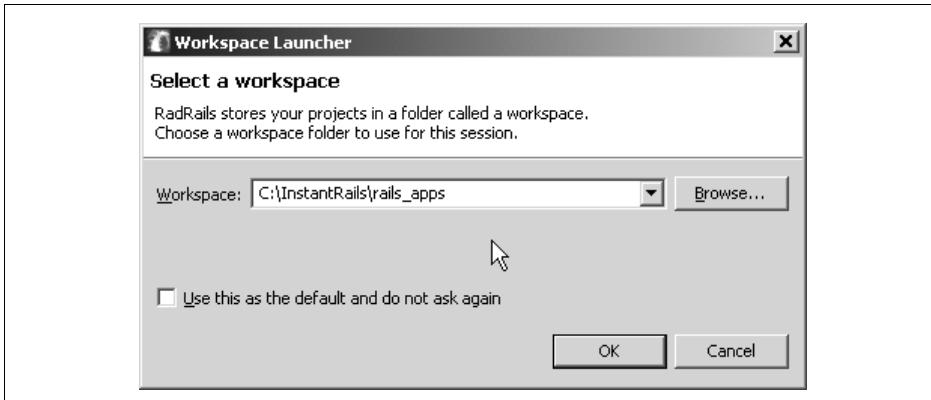


Abbildung 1-4: Wo sollen Ihre Rails-Anwendungen gespeichert werden?

Wechseln Sie dann auf der linken Seite des Preferences-Fensters auf *Rails → Configuration*. Dort fragt man Sie nach den Pfaden zu *Rails*, *Rake* und *Mongrel*. Hier sind */usr/bin/rails*, */usr/bin/rake* und */usr/bin/mongrel_rails* wahrscheinlich, aber auch die Varianten mit */usr/local/bin/** sollten Sie unter Betracht ziehen.

Sie können sich nun gern ein wenig in RadRails umsehen, um ein Gefühl für das Arbeiten mit diesem multifunktionalen Editor zu bekommen. Oder Sie schließen das Programm einfach wieder, denn Sie werden schon allein während des Studiums dieses Buchs noch einige angenehme Stunden mit RadRails verbringen und viele seiner Vorteile in Aktion erleben können. Außerdem gilt es nun noch, dem Ruby-Konglomerat auf Ihrem Rechner den letzten Schliff zu geben. Mit *RubyGems*.

RubyGems, die Bahnpost

Das kleine, aber äußerst nützliche Tool RubyGems beliefert Sie mit Paketen. Sollten Sie bei Linux oder Mac OS schon einmal mit dem Paketmanager *rpm* in *yast*, *apt-get*, *emerge*, *fink* oder wie sie alle heißen gearbeitet haben, kennen Sie den Komfort eines solchen *Paketmanagers* sicher schon. Er besorgt, installiert, aktualisiert und löscht Pakete. In einem Paket steckt entweder ein komplettes Programm oder aber eine Bibliothek. Der Paketmanager lädt Pakete, hier *Gems* genannt, aus dem Internet herunter und installiert sie automatisch an die richtige Stelle, inklusive Dokumentation, sollte es eine solche geben. Dabei berücksichtigt er eventuell vorhandene Abhängigkeiten, sogenannte *dependencies*. Das heißt, sollte ein Programm oder eine Bibliothek zum Laufen noch weitere Bibliotheken benötigen, die noch nicht bei Ihnen installiert sind, installiert RubyGem auf Wunsch auch diese.

Ich würde sagen, wir starten RubyGems einfach einmal. Da es sich bei RubyGems leider nicht um Software mit bunten Fenstern und Knöpfen, sondern um ein Kommandozeilenprogramm handelt, müssen Sie sich zunächst eine solche Kommandozeile organisieren. Klicken Sie dazu einfach auf *Start* und anschließend auf *Ausführen...*, woraufhin sich ein kleines Fenster öffnet, in welchem Sie *cmd* eingeben und *Enter* drücken. Ein standardmäßig schwarzes Fenster öffnet sich. Hier können Sie ganz entspannt mit RubyGems arbeiten. Die Bedienung von RubyGems ist denkbar einfach. Geben Sie einfach *gem* ein und was RubyGems für Sie tun soll, gefolgt von möglichen konkretisierenden Optionen.

Sie sollten erst einmal Ihr ganzes Ruby-Zeugs aktualisieren. Warum? Weil Sie möglicherweise nicht das Aktuellste auf der Platte haben. InstantRails wird nicht bei jeder kleinsten Veränderung an irgendeiner xbeliebigen Bibliothek neu erstellt und zum Download freigegeben. Mit RubyGems ist das Updaten ohnehin viel einfacher. Wir beginnen erst einmal damit, RubyGems selbst zu aktualisieren.



Achten Sie bei jedem Ausführen von RubyGems darauf, dass Sie mit dem Internet verbunden sind. Schließlich befinden sich dort die unerschöpflichen Schatzkammern voll von Gems aller Art.

Geben Sie dort, wo der Cursor so fröhlich blinkt, `gem update --system` ein. Damit fordern Sie RubyGems auf, ein Update durchzuführen, und zwar eines, das sich auf sich selbst bezieht. Dies können Sie mit der Option `--system` bestimmen.

Sobald Sie diesen Befehl mit *Enter* bestätigen, setzt sich RubyGems mit seiner Zentrale im Internet in Verbindung und versucht, ein Update herunterzuladen und zu installieren. Dies kann durchaus einige Momente dauern, haben Sie also etwas Geduld. Außerdem kann es passieren, dass Ihr Bildschirm von einer Flut an Meldungen heimgesucht wird. Die sind gar nicht so wichtig, wie sie aussehen. Wichtig ist, dass am Ende eine Erfolgsmeldung den Update-Vorgang beschließt.



Sie können jederzeit einsehen, welche Version von RubyGems Sie aktuell benutzen. Geben Sie dazu einfach `gem -v` ein.

Nun verpassen wir Ruby on Rails und dem Rest eine Frischzellenkur. Dazu starten wir den gleichen Vorgang wie eben, verzichten jedoch darauf, explizit RubyGems zu aktualisieren. Der Befehl `gem update` reicht also völlig aus.

```
C:\>gem update
Updating installed gems...
Attempting remote update of mongrel
Select which gem to install for your platform (i386-mswin32)
 1. mongrel 0.3.13.4 (ruby)
 2. mongrel 0.3.13.3 (mswin32)
 3. mongrel 0.3.13.0 (ruby)
 4. mongrel 0.3.13.2 (ruby)
 5. mongrel 0.3.13.2 (mswin32)
 6. mongrel 0.3.13.1 (ruby)
 7. mongrel 0.3.13.1 (mswin32)
 8. mongrel 0.3.13 (ruby)
 9. mongrel 0.3.13 (mswin32)
10. mongrel 0.3.12.4 (ruby)
```

Abbildung 1-5: Mit RubyGems können Sie Ihre Ruby-Bibliotheken up-to-date halten

Sollte man Sie beim nun startenden Aktualisierungsprozess bei einzelnen Komponenten auffordern: *Select which gem to install for your platform*, so suchen Sie sich am besten die neueste Version für Ihr System, beispielsweise *mswin32* für ein handelsübliches Windows XP, heraus. Varianten, die mit *ruby* gekennzeichnet sind, müssen nach dem Herunterladen noch kompiliert werden. Dazu ist ein C-Compiler nötig, der möglicherweise auf Ihrem System nicht zur Verfügung steht. Greifen Sie also lieber zu einer bereits kompilierten Version eines Gems. Die Plattformunabhängigkeit von Ruby wird dadurch grundsätzlich nicht beeinflusst.



Möchten Sie gezielt Gems aktualisieren, dann setzen Sie die Namen der Pakete einfach mit Leerzeichen getrennt hinter *gem update*. Möchten Sie das Rails-Framework und den Mongrel-Server aktualisieren, können Sie das mit *gem update rails mongrel* erledigen.

Sie sollten diesen Aktualisierungsvorgang regelmäßig durchführen. So sind Sie immer auf dem neuesten Stand. Beachten Sie dabei jedoch, dass eine neue Ruby-Version nicht per RubyGems installiert werden kann. Das geht nur mit dem Aktualisieren von InstantRails, wodurch das erneute Herunterladen und Entpacken eines ZIP-Archivs erforderlich wird.

Abschließend möchte ich Ihnen noch ein wenig den Mund wässrig machen. Geben Sie doch mal `gem list --local` ein. Sogleich zeigt Ihnen RubyGems, welche Pakete momentan bei Ihnen vorhanden sind. Ich verspreche Ihnen: Einige davon werden wir im Laufe des Buchs noch einsetzen.

Zusammenfassung

Die Vorbereitungen sind abgeschlossen. Und? Schon ein bisschen Herzklagen, wenn Sie an Ruby denken? Möglich wäre es, schließlich wissen Sie nun, dass Ruby eine interpretierte Sprache ist, die im Gegensatz zur Konkurrenz sehr sauberes, schnelles, widerspruchloses und dennoch einfaches Programmieren ermöglicht und die Basis von Ruby on Rails ist. Mit Hilfe von Lösungen wie InstantRails und Locomotive haben Sie Ihr System in Windeseile für die Entwicklung mit Ruby und Ruby on Rails vorbereitet. Dabei haben Sie ganz nebenbei auch einen Webserver und eine Datenbank erhalten, die Sie bei der Entwicklung mit Ruby on Rails brauchen werden. Der Editor RadRails wird Ihre Arbeit durch seine vielen Features noch weiter erleichtern, was dem Paketmanager RubyGems durch die Bereitstellung einer einfach zu bedienenden Funktionalität für die Installation und das Update von Ruby-Bibliotheken und -Programmen gelingt.

Nun wird es wirklich Zeit, all die Dinge im praktischen Einsatz zu erleben. Fangen wir mit den Einzelteilen von Ruby an. Nächste Station: Kapitel 2.

Programmieren mit Ruby

In diesem Kapitel:

- Mit Ruby ist zu rechnen
- Zeichen und Wunder: Strings
- Variablen und Konstanten
- Von von bis bis – Wertebereiche
- Elementares über Arrays
- Weichen stellen
- Die Könige unter den Schleifen: Iteratoren
- Methoden
- Selbst gemachte Klassen

Ihr Rechner spricht jetzt *Ruby*. Er hat im vorangegangenen Kapitel alles bekommen, was er zum Interpretieren und Ausführen der tollsten, größten, gigantischsten, innovativsten Ruby-Programme aller Zeiten braucht. Und wer soll die schreiben? Sie natürlich! Ihr nun möglicher Einwand, dass Sie doch gar kein Ruby können, wird nur noch ein paar Seiten glaubwürdig sein.

Dieses Kapitel versorgt Sie mit wichtigen Grundlagen der Ruby-Programmierung und schafft damit eine hervorragende Ausgangssituation für Ihre späteren *Ruby-on-Rails*-Anwendungen, mit denen wir uns ab dem vierten Kapitel beschäftigen werden und für die Sie ein solides Grundwissen der Programmiersprache Ruby zwingend benötigen. Sie lösen in diesem Kapitel die Fahrkarte für Ihre späteren Hochgeschwindigkeitsreisen *on Rails*. Allerdings wird das viel einfacher sein, als einem Standardticketautomaten ein Billet in den Nachbarort zu entlocken. Und viel mehr Spaß wird es auch machen.

Stück für Stück arbeiten wir uns dazu gemeinsam durch die Syntax, die Besonderheiten und die verblüffenden Eigenheiten von Ruby. Viele kleine anschauliche Beispiele sollen Ihnen dabei das Lernen erleichtern. Dabei werden Sie immer wieder staunen, wie elegant Ruby einige programmiertechnischen Probleme löst und wie dies die Entwicklung Ihrer Programme vereinfacht und beschleunigt.

Neben den Sprachgrundlagen von Ruby werden Sie auch einige Ruby-eigene Werkzeuge und die bereits installierte Entwicklungsumgebung RadRails näher kennen lernen. Auch auf einige Grundbegriffe der Programmierung, die auch in anderen Programmiersprachen eine wichtige Rolle spielen, werde ich eingehen.

Sollten Sie Ruby bereits beherrschen, so nutzen Sie dieses Kapitel einfach, um ein paar Dinge zu wiederholen und aufzufrischen. Ansonsten kann ich Sie nur ermuntern, sich wagemutig in das aufregende Abenteuer des Erlernens einer neuen und außergewöhnlichen Programmiersprache zu stürzen.

Übrigens: Das folgende Kapitel steckt voller Codeschnipsel und kompletter Programme. Das meiste davon können Sie selbst ausprobieren, wozu ich Sie wirklich ermuntern möchte. Denn nur so lernen Sie schnell und mit viel mehr Spaß. Beachten Sie dabei, dass Sie in den Quelltexten dieses Buchs oftmals Kommentare finden, die Ihnen illustrieren sollen, mit welchem Ergebnis an dieser oder jener Stelle zu rechnen ist. Selbstverständlich brauchen Sie die nicht abzutippen. Also: Alles, was mit # beginnt, können Sie beim Ausprobieren getrost ignorieren.

Mit Ruby ist zu rechnen

Fangen wir mit einem ganz einfachen Beispiel an. Ob Sie's glauben oder nicht, aber das ist echter Ruby-Code: `1 + 2`. Mit diesem *Ausdruck* stellen Sie Ruby eine Rechenaufgabe, eine zugegebenermaßen recht einfache. Sollten Sie selbst übrigens Probleme mit dem Berechnen dieses Ausdrucks haben, dann empfehle ich Ihnen, das Buch, das Sie gerade in den Händen halten, noch etwas zurückzustellen und vorerst viel grundlegendere Literatur zu konsumieren. Andernfalls möchten Sie nun sicher gern wissen, ob Ruby auf das gleiche Ergebnis kommt wie Sie. Fragen wir Ruby mal.

Was ist ein Ausdruck?

Ausdrücke werden Ihnen bei Ruby, aber auch in anderen Programmiersprachen, ständig über den Weg laufen. In der englischsprachigen Literatur heißen sie übrigens *expressions*. Ein Ausdruck ist ein syntaktisches Gebilde, das der Ruby-Interpreter auswerten kann, woraufhin er ein Ergebnis ausgibt. Eine Rechenaufgabe ist typisch für einen Ausdruck. Aber selbst eine einzelne Zahl ist ein Ausdruck. Es wird zwar nichts berechnet, aber Ruby kann die Zahl trotzdem auswerten und ausgeben. Nachfolgend sehen Sie eine kleine Liste mit Ausdrücken und, durch das #-Zeichen getrennt, deren Auswertung durch den Ruby-Interpreter. Dabei sehen Sie auch, dass ein Ausdruck nicht unbedingt mathematisch sein muss.

```
9 # => 9
31 + 2 # => 33
"Ruby on Rails" # => "Ruby on Rails"
"Ruby " + "on" + " Rails" # => "Ruby on Rails"
5 == 5 # => true
-459.asb # => 459
a = 1 # => 1
a += 5 # => 6
```

Um Ruby diese einfache Aufgabe zu stellen, bemühen wir *Interactive Ruby*, das oft auch als *irb* abgekürzt wird. Dabei handelt es sich um einen ganz speziellen Ruby-Interpreter, mit dessen Hilfe Sie interaktiv mit Ihrer neuen Lieblingsprogrammiersprache kommunizieren können. Sie geben Ihren Wunsch ein, drücken *Enter* und Ruby antwortet – wenn Sie die Frage richtig gestellt haben, so dass Ruby Sie verstehen kann.

Interactive Ruby wird am besten über die *Kommandozeile* Ihres Betriebssystems ausgeführt. Dazu öffnen Sie zunächst eine solche. In Windows, wo das früher einmal *MS-DOS-Eingabeaufforderung* hieß, gelingt das recht schnell, wenn Sie auf *Start → Ausführen...* klicken und dann *cmd* in das frisch auf den Bildschirm gezauberte Feld eingeben. Daraufhin öffnet sich ein standardmäßig schwarz hinterlegtes Fenster mit weißer Schrift. Außerdem sehen Sie, in welchem Verzeichnis Sie sich gerade befinden. Rechts daneben befindet sich der sogenannte *Prompt*, der durch nervöses Blinken auf sich aufmerksam macht. Geben Sie dort einfach *irb* ein. Durch diese drei Buchstaben starten Sie *Interactive Ruby*. Dabei ist es völlig gleich, in welchem Verzeichnis Sie sich gerade aufhalten. Durch den Eintrag des Ruby-Verzeichnisses in die *PATH*-UmgebungsvARIABLE via *use_ruby.cmd* aus InstantRails haben Sie schließlich überall Zugriff auf Ruby.



Möchten Sie *Interactive Ruby* verlassen, geben Sie einfach *exit* in eine neue Zeile ein und drücken Sie die *Enter*-Taste.

Auf Betriebssystemen, die nicht aus dem Hause Microsoft stammen, gibt es diverse andere Bezeichnungen für den eben erwähnten schwarzen Kasten. Sollten Sie bei Ihrer Linux- oder MacOS-Installation auf *Terminal*, *Shell*, *Console* oder Ähnliches stoßen, starten Sie eben das. Auch hier sollte Ihnen durch die Eingabe von *irb* die Kontaktaufnahme mit Ruby gelingen.

The screenshot shows a Windows command prompt window titled 'C:\WINNT\system32\cmd.exe - irb'. The title bar also displays 'Microsoft Windows 2000 [Version 5.00.2195]'. The window content shows the text: '<C> Copyright 1985-2000 Microsoft Corp.' followed by the prompt 'C:\Dokumente und Einstellungen\Administrator>irb' and 'irb(main):001:0>'. The window has a standard Windows border and title bar.

Abbildung 2-1: *Interactive Ruby* wartet auf Gesprächsstoff

Lassen Sie nicht durch die kryptischen Zeichen zu Beginn der ersten *Interactive-Ruby*-Zeile stören. Sie werden gleich merken, dass eine Angabe dort die Zeilennummer Ihrer *irb*-Sitzung repräsentiert. Geben Sie nun einfach den Ausdruck $1 + 2$ ein. *Interactive Ruby* wird diesen Ausdruck auswerten – was in diesem Fall auch eine Berechnung beinhaltet –, sobald Sie die *Enter*-Taste drücken. In der nächsten Zeile erhalten Sie sogleich die Antwort auf Ihre Frage.



Gewöhnen Sie sich schon einmal daran, dass vor und hinter einem Operator – in diesem Fall handelt es sich um das Pluszeichen, den Additionsoperator – jeweils ein Leerzeichen steht. Das ist kein Muss, Ruby würde Sie auch so verstehen. Aber es gehört zu den ungeschriebenen Gesetzen unter Ruby-Programmierern: Die Lesbarkeit des Quelltextes, besonders wenn dieser etwas unfangreicher als im vorangegangenen Beispiel ausfällt, soll dadurch erhöht werden. Mehr zu derartigen Konventionen beim Schreiben von Ruby-Code finden Sie fein säuberlich aufgelistet im Anhang dieses Buchs und im weiteren Verlauf dieses Kapitels.

Vermutlich wird es Sie relativ unbeeindruckt lassen, dass da jetzt eine Drei als Ergebnis auf dem Monitor zu sehen ist. Aber immerhin haben Sie somit Ihre erste Zeile Ruby-Code erfolgreich ausgeführt. Meine herzliche Gratulation, Welch bewegender Augenblick!

Plus, minus, mal ...

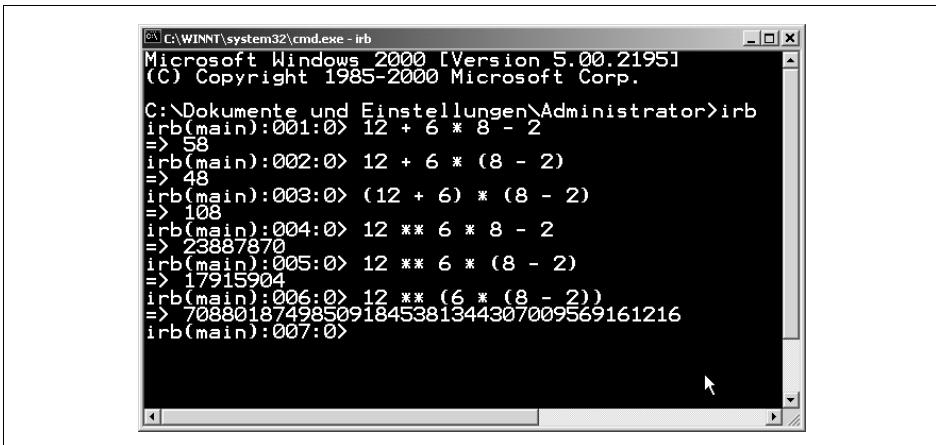
Bevor Sie sich aber nun »Ruby-Programmierer« aufs Visitenkärtchen schreiben lassen, sollten Sie zumindest noch ein paar weitere Berechnungen durchführen. Und damit es nicht ganz so langweilig wird: Wie wäre es mit ein paar weiteren Rechenarten? Denn selbstverständlich sind die vier Grundrechenarten und weitere mathematische Operatoren in Ruby vertreten.

Tabelle 2-1: Wichtige mathematische Operatoren in Ruby

Rechenart	Operator
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Rest einer Division (Modulo)	%
Potenz (x^y)	**

Probieren Sie ruhig ein paar Berechnungen mit diesen Operatoren aus, gern auch in Kombination. Selbstverständlich gilt auch bei Ruby Punkt- vor Strichrechnung, wobei Sie durch die Verwendung von Klammern Einfluss auf die Reihenfolge nehmen können. Beachten Sie, dass Sie Berechnungen, die ausschließlich den Exponenten oder dessen Basis betreffen, ebenfalls klammern müssen.

Bei meinen Experimenten, die in Abbildung 2.2 für die Nachwelt festgehalten sind, komme ich in der letzten Berechnung auf eine recht hohe Zahl. Gern können Sie nun versuchen, diese Zahl in Worte zu fassen. Viel wichtiger ist aber, festzuhalten, dass Ruby prinzipiell mit jeder noch so großen Zahl zurechtkommt. Lediglich Ihr



```
C:\WINNT\system32\cmd.exe - irb
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\Dokumente und Einstellungen\Administrator>irb
irb(main):001:0> 12 + 6 * 8 - 2
=> 58
irb(main):002:0> 12 + 6 * (8 - 2)
=> 48
irb(main):003:0> (12 + 6) * (8 - 2)
=> 108
irb(main):004:0> 12 ** 6 * 8 - 2
=> 23887870
irb(main):005:0> 12 ** 6 * (8 - 2)
=> 17915904
irb(main):006:0> 12 ** (6 * (8 - 2))
=> 708801874985091845381344307009569161216
irb(main):007:0>
```

Abbildung 2-2: Mathematische Experimente mir Ruby

System kann diesem Größenwahn eine Grenze setzen. Im praktischen Einsatz werden Sie wahrscheinlich nie an diese Begrenzung stoßen. Das soll aber nicht heißen, dass Sie sich jetzt zurückhalten müssen. Gönnen Sie sich doch die Freude einiger Berechnungen mit besonders hohen Zahlen. Sehen Sie's einfach als Vorabkalkulation für die zu erwartenden Einnahmen aus Ihren zukünftigen Ruby-on-Rails-Projekten.

Ruby wendet intern einen Trick an, um selbst mit den größten Zahlen zurechtzukommen. Dieser Trick ist allerdings recht aufwändig. Damit dieser Aufwand nicht unnötigerweise auch bei Zahlen im eher kleineren Bereich betrieben werden muss, unterscheidet Ruby einfach zwei Zahlenbereiche.

Für Klein und Groß: Fixnum und Bignum

Genau genommen sind Fixnum und Bignum keine speziellen Zahlenbereiche, sondern Ruby-Klassen mit diesem Namen, die Bildungsvorschriften für Objekte enthalten, welche mit kleinen beziehungsweise großen Zahlen klarkommen. Die Klasse *Fixnum* konzentriert sich auf ganze Zahlen zwischen -2^{30} und $2^{30} - 1$. Um Zahlen, die davor oder danach kommen, kümmert sich *Bignum*. Wohlgemerkt: Wir sprechen von *ganzen* Zahlen; um kommahlältige kümmern wir uns gleich.



Sollte es sich bei Ihrem Rechner um ein hochgezüchtetes Powerpaket mit einem 64-Bit-Herz handeln, reicht der Fixnum-Bereich sogar von -2^{62} bis $2^{62} - 1$. Bitte beachten Sie das bei den folgenden Beispielen.

Um Ihr Vertrauen als Leser in mich als Autor auf eine nie dagewesene Stufe zu bringen, biete ich Ihnen jetzt die Möglichkeit, das eben so locker Dahingeschriebene einer strengen Prüfung zu unterziehen. Geben Sie einfach folgende zwei Zeilen

nacheinander in Ihre Interactive-Ruby-Shell ein. Vergessen Sie nicht die Klammern, denn wir möchten ja das Ergebnis der Berechnungen prüfen – und zwar bezüglich ihrer Klassenzugehörigkeit.

Beispiel 2-1: Fixnum oder Bignum?

```
(2 ** 30 - 1).class  
(-2 ** 30).class  
(2 ** 30).class  
(-2 ** 30 - 1).class
```

Auf diese Weise können Sie auch andere Zahlen überprüfen. Sie könnten beispielsweise die Potenzen gegen ihren ausgerechneten Wert ersetzen und so statt $2^{**} 30$ auch 1073741824 schreiben. Um Ihnen die fehlerträchtige Eingabe solch langer Zahlen zu erleichtern, gestattet Ihnen Ruby die Verwendung von *Tausendertrennzeichen*, wobei es sich nicht um den gewohnten Punkt, sondern um einen *Unterstrich* handelt.

Beispiel 2-2: Große Zahlen einfach schreiben

```
5_134_509_233_198.class  
1_989.class
```

Aber wie groß oder klein Ihre Zahlen auch sind, ob sie in Potenzschreibweise oder mit Tausendertrennzeichen auftreten – eigentlich kann es Ihnen relativ egal sein, ob sie der Klasse Bignum oder Fixnum angehören. Ruby kümmert sich selbst um die Zuordnung und gegebenenfalls um eine Konvertierung von Fix- zu Big- und Big- zu Fixnum, je nach Situation. Sie können also ganz beruhigt mit Ihren Zahlen arbeiten und brauchen sich darum keine Gedanken zu machen. Sie wissen ja bereits: Ruby macht die Drecksarbeit und Sie sind Chef.

Ach, jetzt hätte ich fast noch eine Variante vergessen, die Ihnen das Schreiben nicht nur von großen, sondern auch ganz kleinen Zahlen erleichtern kann: Zehnerpotenzen, die auch andernorts gern in der *e-Schreibweise* notiert werden. Zum Beispiel: $50000 = 5 \cdot 10^4 = 5e4$. Eine Fünf mit vier Nullen. Werfen Sie Interactive Ruby einfach einmal ein paar Zahlen in dieser Schreibweise zu; sie werden umgehend zu normalen Zahlen ausgewertet.

Ist Ihnen aufgefallen, dass Ruby $5e4$ zu einer Zahl mit einer Nachkommastelle auswertet? 50000.0 lautet Rubys Antwort. Nun mag die Null hinter dem Komma zunächst überflüssig erscheinen, doch weist sie hier auf etwas Wichtiges hin: Sie ist Rubys Indikator dafür, dass das Ergebnis der Auswertung eine (*Gleit-)*Kommazahl ist.

Ganz klar, Zehnerpotenzen werden natürlich nicht nur für große, sondern auch für ganz exakte Zahlen verwendet, die erst ab der dreißigsten Stelle hinter dem Komma wirklich interessant werden. Deshalb hält sich Ruby von vornherein sämtliche Komplikationen vom Hals und formt aus jeder in der e-Schreibweise notierten Zahl

```
C:\WINNT\system32\cmd.exe - irb
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\Dokumente_und_Einstellungen\Administrator>irb
irb(main):001:0> 50_000
=> 50000
irb(main):002:0> 5 * 10 ** 4
=> 50000
irb(main):003:0> 5e4
=> 50000.0
irb(main):004:0>
```

Abbildung 2-3: Eine Zahl, drei Schreibweisen

eine Gleitkommazahl. Und nicht nur bei Ruby, sondern auch in einigen anderen Programmiersprachen wird eine solche Zahl *Float* genannt. Wie für die ganzen Zahlen (Fixnum und Bignum) gibt es auch hierfür eine spezielle Klasse.

Floats – um es ganz genau zu nehmen

Gleitkommazahlen werden in Ruby immer mit mindestens einer Nachkommastelle angegeben, auch wenn es sich dabei nur um die Null handelt. So erkennt Ruby, dass Sie unbedingt mit einer Zahl der Klasse *Float* arbeiten möchten, komme, was da wolle.



Das englischsprachige Äquivalent des Begriffs Gleitkommazahl lautet *floating point number*. Das weist schon darauf hin, dass nicht überall auf der Welt ein Komma genutzt wird, um den ganzzahligen Teil einer Zahl vom gebrochenen zu trennen. Im weiten Universum der Programmierung ist der Punkt Standard. Notieren Sie Ihre Gleitkommazahlen in Ruby also immer als Gleitpunktzahlen.

Werfen Sie einen Blick auf folgendes Beispiel. Dort können Sie sehen, dass Ruby anhand der Form der eingegebenen Zahlen die passende Klasse wählt.

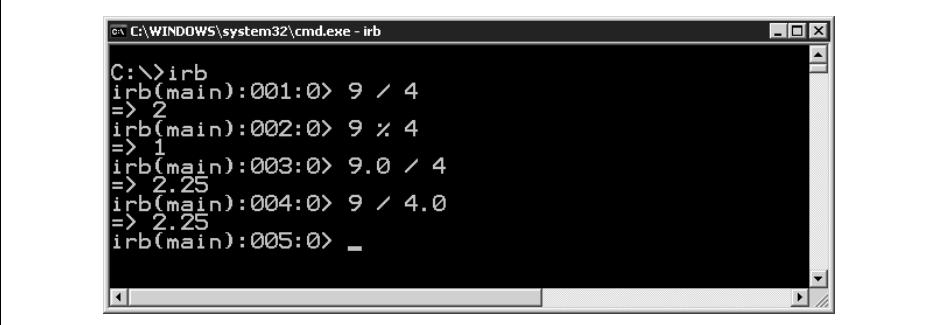
```
C:\WINNT\system32\cmd.exe - irb
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\Dokumente_und_Einstellungen\Administrator>irb
irb(main):001:0> 78.class
=> Fixnum
irb(main):002:0> 78.0.class
=> Float
irb(main):003:0> _
```

Abbildung 2-4: Komma Null ist doch was wert

So richtig wichtig wird die Unterscheidung zwischen einer Ganzzahl und einer Gleitkommazahl beispielsweise dann, wenn es ums Dividieren geht. So eine ganze Zahl lässt sich nun einmal nicht immer restlos in andere ganze Zahlen zerteilen. Das erklärt auch die Existenz des %-Operators in den Klassen Fixnum und Bignum.

Eine ganze Zahl geteilt durch eine ganze Zahl ergibt eine ganze Zahl. Teilen Sie hingegen eine Float-Zahl oder eine ganze Zahl durch eine Float-Zahl, so ist auch das Ergebnis ein kommahaltiger Wert. Probleme mit Resten gibt's dabei also nicht.

A screenshot of a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe - irb'. The window contains the following text:

```
C:>irb
irb(main):001:0> 9 / 4
=> 2
irb(main):002:0> 9 % 4
=> 1
irb(main):003:0> 9.0 / 4
=> 2.25
irb(main):004:0> 9 / 4.0
=> 2.25
irb(main):005:0> _
```

The window has standard Windows-style scroll bars on the right and bottom.

Abbildung 2-5: Dividieren mit ganzen und Gleitkommazahlen

Grundsätzlich gilt: Enthält Ihre Rechnung auch nur eine Kommazahl, so wird auch das Ergebnis ein Wert der Klasse Float sein. Dennoch ist größte Wachsamkeit Ihrerseits dabei erforderlich. Bei kombinierten Berechnungen, in denen beispielsweise eine aus ganzen Zahlen bestehende Division den Vorrang vor einer Multiplikation mit einer Kommazahl hat, kommt zwar ein Wert mit Komma als Ergebnis heraus. Die erste Division wurde dennoch im Bereich der ganzen Zahlen ausgeführt. Probieren Sie doch einmal Beispiel 2.3 #BQV 2.3# in Ihrem Interactive Ruby aus, und Sie werden den Stolperstein erkennen:

Beispiel 2-3: Gefährlich: Kombinierte Berechnungen

```
4 / 3 * 2.5
4.0 / 3 * 2.5
```

Ihre Rechenaufgabe arbeitet Ruby von links nach rechts ab. Ruby teilt also erst $4 / 3$ beziehungsweise $4.0 / 3$ und multipliziert erst dann das Zwischenergebnis mit 2.5. Das Problem ist nur, dass $4 / 3$ im Fixnum-Bereich berechnet wird und somit 1 statt 1.33333 im Float-Bereich ergibt. Dementsprechend unterschiedlich ist die Ausgangssituation für's anschließende Malnehmen.

Neben der Division und der Multiplikation können Sie auch alle anderen mathematischen Operatoren, die für die ganzen Zahlen in Ruby implementiert und in Tabelle 2.1 aufgelistet sind, für das Rechnen mit Gleitkommazahlen nutzen. Selbst den Rest einer Division mit mindestens einer Kommazahl rechnet Ihnen Ruby aus.

Zahlen konvertieren

Es gibt noch eine andere Möglichkeit, wie Sie Ruby sagen können, dass Sie gern mit einer Kommazahl rechnen möchten. Schreiben Sie einfach eine ganze Zahl auf und weisen Sie Ruby dann an, diesen Wert in ein Objekt der Klasse Float zu konvertieren. Die Methode `to_f` übernimmt diese Aufgabe. Das `f` steht dabei, wie Sie sich vielleicht schon denken können, für `Float`.



Es ist durchaus üblich, Methoden einer Klasse durch die Schreibweise `Klasse#Methode` darzustellen. Das findet besonders in der Ruby-Referenz Anwendung. `Fixnum#to_f` ist eben kürzer als »Die Methode `float` der Fixnum-Klasse.«

Und es geht auch andersrum: Mit `to_i` (wobei `i` für das englische Wort *Integer* und damit für eine ganze Zahl steht) können Sie eine Kommazahl entkommafizieren. Beachten Sie hierbei, dass sich `to_i` dabei überhaupt nicht für irgendwelchen Rundungskokolores interessiert. Alles nach dem Komma (oder besser: dem Punkt) einer Zahl wird rigoros entsorgt.

Hängen Sie einfach den Namen der Methode, getrennt durch einen Punkt, an die Zahl oder den Ausdruck an, den Sie umwandeln möchten. So wie hier:

```
C:\>irb
irb(main):001:0> 45.to_f
=> 45.0
irb(main):002:0> 23 * 6.to_f
=> 138.0
irb(main):003:0> 18.93.to_i
=> 18
irb(main):004:0> 1.2.to_i.to_f
=> 1.0
irb(main):005:0> _
```

Abbildung 2-6: `to_f` und `to_i`

Bei der Umwandlung einer Gleitkommazahl in eine ganze Zahl besteht natürlich auch die Möglichkeit, den Nachkommabereich mathematisch korrekt zu berücksichtigen. Das Runden einer Zahl ist aber nur eine der weiteren Möglichkeiten, die Ihnen Ruby bei der Verarbeitung von Zahlenwerten anbietet.

Noch mehr Mathe

Die folgenden Methoden können Sie auf eine Zahl oder das Ergebnis einer Berechnung anwenden. Auch hier gilt: Punkt dahinter, dann den Methodennamen – fertig.

Tabelle 2-2: Wichtige mathematische Operatoren in Ruby

Methode	Anwendung	Beispiel mit Auswertung
abs	Gibt den absoluten (stets positiven) Betrag einer Zahl zurück. Das Ergebnis ist von der gleichen Zahlenklasse.	-1.7.abs # => 1.7
ceil	Rundet eine Gleitkommazahl unabhängig von ihrem Nachkommabereich auf und gibt eine ganze Zahl zurück.	1.7.ceil # => 2
floor	Rundet eine Gleitkommazahl unabhängig von ihrem Nachkommabereich ab und gibt eine ganze Zahl zurück.	1.7.floor # => 1
round	Rundet eine Gleitkommazahl abhängig von ihrem Nachkommabereich auf oder ab und gibt eine ganze Zahl zurück.	1.7.round # => 2 23.1.round # => 23
succ	Liefert den Nachfolger einer Bignum- oder Fixnum-Zahl	5.succ # => 6

Das ist natürlich noch nicht alles, was Ruby mathemäßig draufhat. Einige weitere Methoden werden Sie im Laufe des Buchs noch kennen lernen.

Zufallszahlen

Um mit Ruby Zufallszahlen zu erzeugen, gibt es die Methode `rand`. Sie erzeugt bei jedem Aufruf genau eine Zufallszahl, wobei Sie die obere Grenze selbst festlegen und der Methode übergeben können. Beachten Sie, dass dieser Wert hier der Methode folgt, umgeben von Klammern.

Der Aufruf von `rand(200)` gibt zum Beispiel eine ganze Zahl wieder, die einen Wert von mindestens 0, höchstens aber 199 hat. Oder anders ausgedrückt: `rand(200)` liefert Ihnen eine von 200 möglichen Zufallszahlen, beginnend mit 0. Negative Höchstgrenzen werden von Ruby übrigens automatisch in positive umgewandelt.

Rufen Sie `rand` ohne Höchstwert auf, so erhalten Sie eine Gleitkommazahl zurück, für die gilt: $0 \leq \text{Ergebnis} < 1$. Das Ergebnis kann in diesem Fall also 0, aber nie- mals 1 sein.

Beim Ausfüllen eines 6-aus-49-Lottoscheins kann Sie Ruby so unterstützen:

```
rand(49) + 1
```

Falls beim nächsten Spieleabend Ihre Würfelhand einen Ermüdungsbruch erleidet, lassen Sie Ruby würfeln:

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

Sie betreiben ein aufstrebendes Unternehmen, das telefonisch Lottospielgemeinschaftsanteile verkauft? Ruby besorgt Ihnen ganz schnell die Telefonnummern Ihrer zukünftigen Kundenschaft.

```
rand(9000000) + 1000000
```

Damit verlassen wir ganz kurz die Welt der Zahlen. Sonst bekommen Sie noch den Eindruck, dass Ruby nur ein eher schlecht zu bedienender Taschenrechner ist, der keinen 60-MByte-Download rechtfertigt. Und außerdem gibt es im Leben ja nicht nur Zahlen – wie langweilig und schlank wäre sonst dieses Buch.

Zuvor checken wir kurz noch das Wichtigste, was Sie bislang über Rubys Umgang mit Zahlen wissen sollten: Es gibt ganze und Gleitkommazahlen. Die ganzen Zahlen gehören der Klasse *Fixnum* an, es sei denn, sie überschreiten eine gewisse Grenze. Dann schlägt die Stunde der Klasse *Bignum*. Sie als Ruby-Programmierer merken davon aber gar nichts, Ruby übernimmt die Verwaltung und eine eventuelle Umwandlung. Gleitkommazahlen müssen als solche kenntlich gemacht werden, damit Ruby weiß, dass sie zur Klasse *Float* gehören. Dabei nutzen Sie beim Schreiben einen Punkt anstatt des Kommas. Die vier Grundrechenarten sowie Modulo-Berechnung und Potenzierung sind mit allen Zahlen möglich. Gleitkommazahlen können mit *ceil*, *floor* oder *round* je nach Wunsch gerundet und zu ganzen Zahlen umgewandelt werden. Alle Zahlen können mit *Tausendertrennzeichen* oder als Zehnerpotenz in *e-Schreibweise* notiert werden.

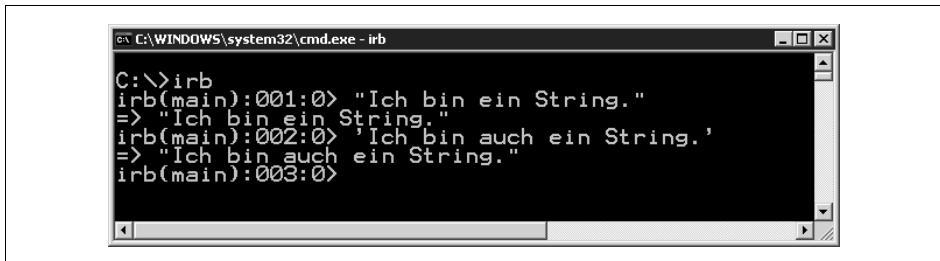
Zeichen und Wunder: Strings

Lassen Sie uns nun einen Schritt weiter gehen und uns den *Zeichenketten* zuwenden. Die sind in einem Programm schließlich mindestens genauso wichtig wie Zahlen. Sie werden gleich einige wichtige Dinge über Ruby-Strings erfahren und diverse Techniken kennenlernen, um mit ihnen umzugehen.

String-Literale

Wie für Zahlen stellt Ruby auch für Texte eine spezielle Klasse zur Verfügung. In diesem Falle heißt sie *String*. Und ähnlich wie bei den Zahlen gibt es auch bei buchstabenlastigen Werten eindeutige Kennzeichen, die einen Text als solchen für Ruby erkennbar machen und Ruby dazu veranlassen, die Klasse String zu verwenden. Das ist genau dann der Fall, wenn Sie Ihren Text in Anführungszeichen kleiden, was aus dem Text ein *String-Literal* macht, wobei ein String-Literal auch ein Ausdruck ist. Dabei sind sowohl einfache als auch doppelte Anführungszeichen möglich. Allerdings muss die Sorte Anführungszeichen, für die Sie sich entscheiden, sowohl am Anfang als auch am Ende Ihres Literals stehen.

Eine weitere Möglichkeit, einen String zu erzeugen, bietet die Methode `to_s`, die auch als `to_str` auf ganzzahlige oder Float-Werte angewendet werden kann und sie zu Objekten der Klasse String macht. Danach können Sie nur noch mit Methoden der Klasse String auf die Zahlen einwirken. Das macht sich übrigens auch dann bemerkbar, wenn Sie Strings addieren.

A screenshot of a Windows command prompt window titled "C:\>irb". The window shows an interactive Ruby session (irb) with the following code and output:

```
C:\>irb
irb(main):001:0> "Ich bin ein String."
=> "Ich bin ein String."
irb(main):002:0> 'Ich bin auch ein String.'
=> "Ich bin auch ein String."
irb(main):003:0>
```

Abbildung 2-7: Einfach oder doppelt – Strings brauchen Anführungszeichen

Text plus Text

Sie können mit Strings nämlich auch rechnen, wenngleich dieses Wort vielleicht ein wenig übertrieben ist, da praktisch nur zwei mathematische Operatoren und Klammern möglich sind. Mit dem + -Zeichen können Sie mehrere Strings verketten, was auch gern *konkatenieren* genannt wird. Für Freunde dieses Profi-Begriffs steht alternativ zum Additionszeichen noch die Methode concat zur Verfügung. Und noch eine Möglichkeit gibt es: <<. Mit diesen doppelten *Kleiner-als-Zeichen* können Sie ebenfalls zwei oder mehr Strings aneinander heften.

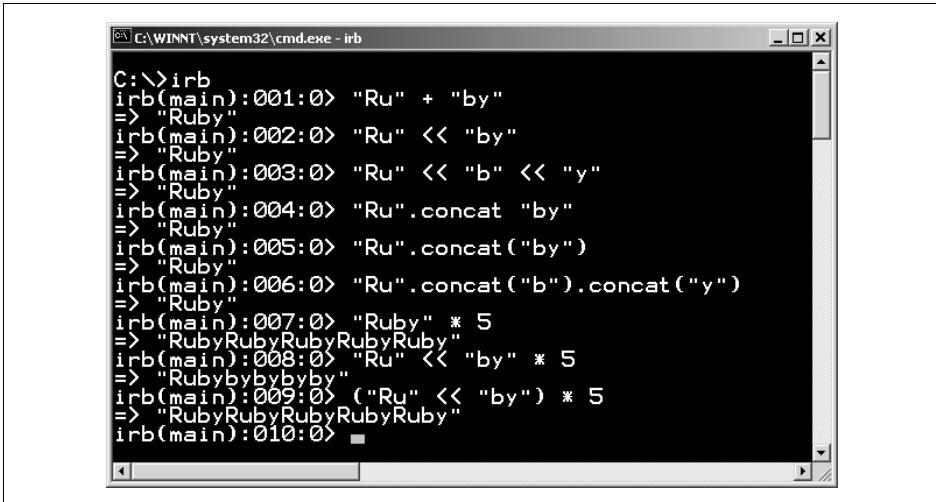


Das Pluszeichen erzeugt aus dem ersten und dem zweiten beteiligten String-Objekt ein drittes, völlig neues String-Objekt, das die beiden Quellinhalte in sich vereint, anders als bei << und concat, wo der Inhalt des zweiten String-Objekts an den des ersten String-Objekts angehängt wird. Es entsteht also kein neues Objekt, wodurch Ressourcen und Zeit gespart werden. Dennoch gibt es Situationen, in denen die Verwendung des Additionsoperators und das Erzeugen eines völlig neuen Objekts sinnvoller sind als die Manipulation des ersten String-Objekts.

Außerdem führt auch das Multiplikationssternchen ein Doppel Leben. Sie können es verwenden, um einen Text mehrfach auszugeben. Allerdings müssen Sie Malzeichen und Anzahl *nach* dem zu vervielfältigenden String-Objekt notieren, damit Ruby weiß, dass Sie die Kopiefunktion der String-Klasse verwenden und nicht die Multiplikationsfähigkeiten von Fixnum oder Bignum ausloten möchten.

Wie bei mathematischen Berechnungen können Sie auch die Priorität mehrerer Methoden am String-Objekt gewichten. Dazu stehen Ihnen Klammern zur Verfügung, mit denen Sie eindeutig festlegen können, was zuerst passieren soll.

Interessant wird es, wenn Sie als String verkleidete Zahlen addieren möchten. Selbstverständlich greift auch dann das Konkatenationsprinzip. Soll heißen: Aus "2" + "2" wird nicht etwa 4 sondern "22".

A screenshot of an old Windows command window titled "C:\WINNT\system32\cmd.exe - irb". The window contains a series of Ruby commands and their results. The commands involve concatenation and repetition of strings. The results show how Ruby handles string operations.

```
C:>irb
irb(main):001:0> "Ru" + "by"
=> "Ruby"
irb(main):002:0> "Ru" << "by"
=> "Ruby"
irb(main):003:0> "Ru" << "b" << "y"
=> "Ruby"
irb(main):004:0> "Ru".concat "by"
=> "Ruby"
irb(main):005:0> "Ru".concat("by")
=> "Ruby"
irb(main):006:0> "Ru".concat("b").concat("y")
=> "Ruby"
irb(main):007:0> "Ruby" * 5
=> "RubyRubyRubyRubyRuby"
irb(main):008:0> "Ru" << "by" * 5
=> "Rubybybybyby"
irb(main):009:0> ("Ru" << "by") * 5
=> "RubyRubyRubyRubyRuby"
irb(main):010:0>
```

Abbildung 2-8: +, <<, concat, *, 0

Einfach oder doppelt?

Es gibt Unterschiede in der Verwendung von doppelten und einfachen Anführungszeichen. Während Ruby bei dem Anführungszeichenpärchen schaut, ob nicht innerhalb des String-Literals noch ein Ausdruck ausgewertet oder etwas anderes beachtet werden muss, werden in einfache Anführungszeichen gekleidete Zeichenketten ungesenen von Ruby übernommen.

Ja, Sie haben richtig gelesen: Ein String-Ausdruck kann einen oder mehrere weitere Ausdrücke enthalten, die von Ruby vor dem Anzeigen des Strings verarbeitet werden. Mag verwirrend klingen, ist aber ein großartiges Werkzeug. Sie können innerhalb eines String-Literals beispielsweise eine kleine Rechenaufgabe platziieren, die Ruby on-the-fly durchkalkuliert und an deren Stelle es das Ergebnis in die Zeichenkette setzt – vorausgesetzt, Sie haben sich für die doppelten Anführungszeichen entschieden.

Ein Ausdruck, der innerhalb eines Strings platziert werden soll, muss von zwei geschweiften Klammern umgeben sein und mit dem # -Zeichen beginnen.



Sollten Sie Probleme haben, geschweifte Klammern, Backslashes oder andere Zeichen bei irb einzugeben, dann starten sie irb mit dem Schalter `--noreadline`. Beachten Sie hierbei bitte die doppelten Bindestriche am Anfang des Schalters. Mit dem Eingeben von `irb --noreadline` schalten Sie die Erweiterung Readline aus, die Funktionen für das leichtere Arbeiten mit Kommandozeilen enthält, aber nicht zwingend für den Betrieb von Interactive Ruby nötig ist, und in diesem Fall sogar behindert.

Probieren Sie Ausdrücke in Strings selbst aus, füttern Sie Interactive Ruby mit folgenden Zeilen und sehen Sie selbst, welche Unterschiede die Verwendung von einfachen und doppelten Anführungszeichen zur Folge hat.

Beispiel 2-4: Ausdrücke in Strings mit doppelten und einfachen Anführungszeichen

```
"3 und 2 sind #{3 + 2}."  
'3 und 2 sind #{3 + 2}'.'
```

Doch nicht nur Ausdrücke machen den Unterschied zwischen einfach und doppelt. Vielleicht kennen Sie bereits so genannte *Escape-Sequenzen*. Sie setzen sich zusammen aus einem Backslash gefolgt von einem Buchstaben und stehen jeweils für eine bestimmte Aktion. Die bekannteste Escape-Sequenz ist wohl `\n`. Es sorgt dafür, dass an der gleichen Stelle, an der sich `\n` befindet, ein Zeilenumbruch im Text erzwungen wird. Auch in diesem Fall ignorieren einfache Anführungszeichen den Umbruch und jede andere Escape-Sequenz. Leider weigert sich `irb`, erzwungene Zeilenumbrüche als solche anzuzeigen. Dennoch ist, wenn Sie folgendes Beispiel ausprobieren, ein klarer Unterschied zu sehen.

Beispiel 2-5: Escape-Sequenzen in Strings

```
"Ein umgebrochener\nText"  
'Ein umgebrochener\nText'
```

Warum kennt Ruby eigentlich zwei Arten von String-Begrenzungen? Ganz einfach. Wenn Sie beim Schreiben eines String-Literals merken, dass da nicht zum Auswerten oder Escapen enthalten ist, nutzen Sie einfache Anführungszeichen. Denn dann muss sich ja Ruby auch nicht die Mühe machen und den String akribisch untersuchen. Das spart Zeit und Ressourcen.

Ruby bietet Ihnen Alternativen zur Verwendung von doppelten und einfachen Anführungszeichen als String-Begrenzer. Die aus Beispiel 4 könnten Sie auch so notieren:

Beispiel 2-6: Escape-Sequenzen in Strings

```
%Q{3 und 2 sind #{3 + 2}.}  
%q{3 und 2 sind #{3 + 2}.}
```

Diese Schreibweise sieht auf den ersten Blick möglicherweise etwas seltsam für Sie aus, allerdings bringt sie einen entscheidenden Vorteil mit. Immer wieder gibt es Probleme, wenn beispielsweise in einer Zeichenkette, die von doppelten Anführungszeichen umgeben ist, eine in ebensolche gekleidete wörtliche Rede enthalten ist. Das gleiche Problem tritt auf, wenn ein Apostroph in einem Text steckt, den einfache Anführungszeichen zieren. Dann kommt selbst Ruby durcheinander, denn der Grundsatz lautet schließlich: Ein String geht von einem Anführungszeichen zum nächsten. Und wenn da ein Zeichen, das zumindest so aussieht, dazwischen kommt, bricht Verwirrung aus.

Sie können also `%Q{...}` als Ersatz für doppelte sowie `%q{}` als Ersatz für einfache Anführungszeichen verwenden. Wenn Sie wissen, dass Anführungszeichen im Englischen *quotes* genannt werden, merken Sie sich bestimmt auch recht schnell, dass der Buchstabe Q hier das Entscheidende ist.

Sollten Sie bereits Erfahrungen mit anderen Programmiersprachen haben, in denen das Problem von vermeintlichen Anführungszeichen im Text natürlich auch vor kommt, werden Sie jetzt vielleicht insistieren, dass man missverständliche Zeichen doch maskieren kann. Guter Einwand – selbstverständlich geht das auch in Ruby. Sie können ein Zeichen mit einem vorangestellten Backslash maskieren und damit anzeigen, dass das nächste Zeichen eben kein abschließendes Anführungszeichen ist.

Und dann gibt es natürlich noch eine weitere Variante, diesem Problem aus dem Weg zu gehen, wenngleich diese nicht gerade die eleganteste Lösung ist: Notieren Sie ein String-Literal, das einen Apostroph enthält, einfach mit doppelten Anführungszeichen am Anfang und am Ende. Die einfachen können dann zum Einsatz kommen, wenn der Text Gänsefüßchen enthält und weder etwas ausgewertet noch Rücksicht auf Escape-Sequenzen genommen werden muss. Um grammatisch Amok zu laufen und Ruby den Text *Rosi's Bierbar* aufzuzwingen, gibt es somit folgende Möglichkeiten:

Beispiel 2-7: Viermal Rosi's Bierbar

```
%Q{Rosi's Bierbar}  
%q{Rosi's Bierbar}  
"Rosi's Bierbar"  
'Rosil\'s Bierbar'
```

Wie Sie bereits wissen, bildet Ruby Strings automatisch nach den Vorgaben der gleichnamigen Klasse als Objekt ab. Außerdem wissen Sie schon, dass Objekte mehr können, als nur irgendwelche Daten zu repräsentieren. Sie bringen beispielsweise Methoden und Operatoren mit, die auf die Daten direkt angewendet werden. So ist das auch in der String-Klasse.

Wortspielereien mit Methode(n)

Nachfolgend sollen Sie einen kleinen Einblick erhalten, welche Möglichkeiten Sie mit der Benutzung der String-Klasse haben. Sollten Sie sich fragen, ob die ein oder andere Methode überhaupt einen praktischen Nutzen hat, so seien Sie versichert: Ja. Und irgendwann in ferner oder naher Zukunft werden Sie sich bei einem konkreten Problem an die zunächst unterschätzte Funktionalität erinnern und sich über ihre Existenz freuen!

Zählen mit Buchstaben

Völlig klar: Betrachtet man ein Alphabet, so hat jeder Buchstabe bis auf das Z einen Nachfolger. Da liegt doch die Vermutung nahe, dass Rubys String-Klasse mit einer entsprechenden Methode daherkommt. Und so ist es auch. Sie heißt wie bei den Zahlen `succ`. Allerdings kann `succ` hier viel mehr, als nur den Nachfolger eines Buchstabens zu ermitteln. Selbst eine Kombination aus Zahlen und Buchstaben bekommt `succ` genachfolgert. Nach A kommt B, nach a kommt b, nach Z kommt AA und nach A9 kommt B0 – was sonst?

Beispiel 2-8: succ – auch Buchstaben haben Nachfolger

```
'AZ9'.succ # => "BA0"  
'Jazz'.succ # => "Jbaa"  
'Hanf'.succ # => "Hang"
```

Zahlen aus Buchstaben

Sie haben beim ersten Rendezvous mit der Float-Klasse Konvertierungsmethoden kennen gelernt. Sie haben aus einem ganzzahligen Wert einen kommahaltigen geformt und umgekehrt. So etwas gibt es auch in der String-Klasse. Mit Hilfe der Methoden `to_i` und `to_f` können Sie Zahlen aus einem String filtern und Sie als Fixnum/Bignum- beziehungsweise Float-Objekte weiterverwenden. Beachten Sie bitte, dass bei der Konvertierung keine Rundung erfolgt und dass die zu konvertierende Zahl den String-Inhalt eröffnen muss, eventuell folgende Buchstaben und Zeichen oder später folgende Zahlen werden ignoriert. Sollte die Konvertierung misslingen oder der String keine Zahl beinhalten, geben beide Methoden 0 zurück. Die Verwendung des Tausendertrennzeichens im String ist möglich.

Beispiel 2-9: Strings werden Zahlen: to_i und to_f

```
"5245".to_f # => 5245.0  
"11 Freunde sollt ihr sein.".to_i # => 11  
"Der 7. Sinn".to_f # => 0  
"2_500_000 Autos".to_i # => 2500000
```

Strings im Interview

Rubys String-Klasse bringt eine Vielzahl von Methoden mit, die Ihnen diverse Informationen über eine Zeichenkette verraten – man muss so einen String-Objekt nur die richtigen Fragen stellen.

Um zu erfahren, ob ein bestimmtes Zeichen innerhalb eines Strings vertreten ist, empfiehlt sich die Methode `include?`. Das Fragezeichen gehört dabei zum Methodennamen und kennzeichnet, übrigens nicht nur in diesem Fall, dass eine Frage gestellt wurde, die mit Ja oder Nein beantwortet werden kann. Oder um genau zu sein: mit `true` oder `false`. Ach, nur nebenbei: `true` ist ein Objekt der Klasse `TrueClass`, `false` eines der Klasse `FalseClass`. Nicht, dass Sie denken, bei Ruby wäre irgendetwas klassenlos.

Sie können `String#include?` auch zur Feststellung des Vorhandenseins einer zusammenhängenden Zeichenkette verwenden. Hierbei gilt: Ruby achtet penibel auf Groß- und Kleinschreibung. Und nicht nur hier, wie Sie noch sehen werden.

Beispiel 2-10: Include? Oder: Drin oder nicht drin?

```
"Ruby on Rails".include?("R") # => true  
"Ruby on Rails".include?("r") # => false  
"Ruby on Rails".include?("Python") # => false  
"Ruby on Rails".include?("Ruby") # => true
```

Durch den Aufruf von `length` oder auch `size` gelangen Sie an die Menge der Zeichen, die ein String-Literal insgesamt enthält, Leerzeichen und andere Nicht-Buchstaben inklusive. Ein ähnlicher Befehl der String-Klasse ist `count`. Allerdings verlangt `count` nach einem Parameter, in dem Sie eine Sammlung von Zeichen angeben müssen – mindestens ein Zeichen sollte es schon sein. Daraufhin gibt Ihnen `count` die Anzahl von Vorkommen der im Parameter notierten Zeichen wieder.

Beachten Sie hierbei, dass `count` den Parameterwert nicht als komplettes Wort betrachtet, sondern jedes Zeichen einzeln. Darüber hinaus unterscheidet `count` nach Groß-/Kleinschreibung. Möchten Sie also wissen, wie oft beispielsweise der Buchstabe `Z` in einem String vorkommt, garantieren Sie mit dem Parameter `Zz`, dass sowohl kleine als auch große Lettern dieser Sorte gezählt werden.

Sie können die Auswahl mit dem Zeichen `^` ins Gegenteil drehen und so die Anzahl an Zeichen erfragen, die dem Parameter *nicht* entsprechen. Setzen Sie einen Bindestrich zwischen zwei Zeichen, um einen ganzen, auf dem Alphabet basierenden Bereich von Buchstaben zählen zu lassen. Die beiden begrenzenden Buchstaben werden dabei einbezogen. Auch diese Auswahl können Sie mit `^` ins Gegenteil verkehren. Möchten Sie, dass mehrere Regeln gelten sollen, dann trennen Sie sie einfach mit einem Komma und notieren sie hintereinander.

Beispiel 2-11: length, size und count

```
'Ruby on Rails'.length # => 13  
'Ruby on Rails'.size # => 13  
'Ruby on Rails'.count('Rr') # => 2  
'Ruby on Rails'.count('Ruby') # => 2  
'Ruby on Rails'.count('^Rr') # => 11  
'Ruby on Rails'.count('s-y') # => 3  
'Ruby on Rails'.count('s-y', '^u') # => 2
```

Mit `count` können Sie also erfahren, wie oft ein oder mehrere Buchstaben sich in einem String befinden. Möchten Sie jedoch wissen, an welcher Stelle sich ein Zeichen genau befindet, muss eine andere Methode des String-Objekts ran.

Mit `index` können Sie ermitteln, an welcher Stelle ein Zeichen oder eine Zeichenkette zuerst auftritt. Dabei geht `index` die Zeichenkette von links nach rechts durch. Wichtig hierbei: Das erste Zeichen eines Strings hat den Index `0`, und es wird nach

Groß- und Kleinschreibung unterschieden. Bei der Suche nach einer Zeichenkette wird die Startposition des ersten Auftretens zurückgegeben. Im Gegensatz zu `count` wird bei `index` wirklich nach einer zusammenhängenden Buchstabenkette gesucht, und nicht nach mehreren einzelnen Buchstaben einer Sammlung. Übrigens: Sie können durch die Festlegung eines Offsets, das als zweiter Parameter angegeben wird, festlegen, ab welcher Position `index` suchen soll. Standardmäßig ist dieser Wert 0, verweist also auf ganz links.

Konnte `index` seinen Suchauftrag nicht erfüllen, erfahren Sie dies durch den Rückgabewert `nil`, was mit »nix gefunden« sehr frei übersetzt werden kann. Wie Sie sich vielleicht schon denken können, ist natürlich auch `nil` wie so vieles in Ruby ein Objekt und wurde nach den Vorgaben der Klasse `NilClass` gebildet. Verwechseln Sie `nil` aber nicht mit der 0, denn diese Zahl steht bekanntermaßen für die erste Stelle.

Möchten Sie von rechts suchen, also das letzte Vorkommen eines Zeichens oder einer Zeichenkette ermitteln, benutzen Sie die Methode `rindex`, die sich ansonsten wie `index` verhält.

Beispiel 2-12: Positionen von Buchstaben ermitteln

```
'Ruby on Rails'.index('R') # => 0  
'Ruby on Rails'.rindex('R') # => 8  
'Ruby on Rails'.index('on') # => 5  
'Ruby on Rails'.index('R', 5) # => 8  
'Ruby on Rails'.index('r') # => nil
```

Mit einem Paar eckiger Klammern können Sie auch den zu `index` und `rindex` umgekehrten Weg gehen und ein oder mehrere Zeichen über deren Position ansprechen. Dabei haben Sie mehrere Möglichkeiten, diesen Operator zu nutzen. Wenn Sie eine *ganze Zahl* zwischen die Klammern packen, erhalten Sie den *ASCII-Code* des Zeichens, das sich an der entsprechenden Position im String befindet. Bei zwei *durch Komma getrennten Ganzzahlen* erhalten Sie einen Teil des Strings zurück, wobei die erste Zahl für die *Startposition* und die zweite Zahl für die *Menge* der auszugebenden Zeichen steht. Trennen Sie die beiden Zahlen durch *zwei Punkte*, so stellt der zweite Parameter die Position dar, an der der Teilstring enden soll.



Zwei Punkte zwischen zwei Zahlen umschreibt eine besondere Notation in Ruby, die *Bereich* oder *Range* genannt wird. Sie erfahren im Laufe dieses Kapitels noch mehr zu diesem Thema.

Geben Sie eine Startposition als negativen Wert an, beginnt der `[]`-Operator von rechts zu zählen. In diesem Fall ist das rechte Zeichen über -1, also nicht -0, zu erreichen. Liegt eine Startposition, also der erste beziehungsweise einzige Parameter, außerhalb der möglichen Werte, wird `nil` zurückgegeben.

Beispiel 2-13: Der []-Operator

```
'Ruby on Rails'[0] # => 82
'Ruby on Rails'[-13] # => 82
'Ruby on Rails'[0..4] # => "Ruby"
'Ruby on Rails'[-4..4] # => "Ruby"
'Ruby on Rails'[8..13] # => "Rails"
'Ruby on Rails'[-5...-1] # => "Rails"
'Ruby on Rails'[15] # => nil
```

Sollten Ihnen die eckigen Klammern optisch nicht so zusagen, können Sie alternativ auch eine Methode benutzen, die die gleichen Ergebnisse liefert. Sie heißt slice.

Beispiel 2-14: slice statt []

```
'Ruby on Rails'.slice(-4,4) # => "Ruby"
'Ruby on Rails'.slice(8..13) # => "Rails"
```

Nun wissen Sie, wie Sie Strings allerlei Infos entlocken können. Was ist aber, wenn Ihnen einige der in Erfahrung gebrachten Fakten nicht gefallen? Ein klarer Fall für die manipulativen Methoden der String-Klasse.

Zeichen wechsle Dich!

Sie können allerlei anstellen mit so einem String-Objekt. Zum Beispiel Buchstabentransformationen von groß zu klein und klein zu groß. Sie gelingen mit capitalize, swapcase, upcase und downcase. Während upcase alle Kleinbuchstaben eines Strings zu großen macht, verfolgt downcase den entgegengesetzten Weg. Die Methode capitalize verwandelt den ersten Buchstaben eines Strings in einen großen und alle anderen werden zu Kleinbuchstaben. Mit swapcase machen Sie kleine Buchstaben eines Strings zu großen und umgekehrt.

Beispiel 2-15: Methoden für groß und klein

```
"Ruby on Rails".upcase # "RUBY ON RAILS"
"Ruby on Rails".downcase # "ruby on rails"
"Ruby on Rails".capitalize # "Ruby on rails"
"Ruby on Rails".swapcase # "rUBY ON rAILS"
```



Leider übergehen die Methoden capitalize, swapcase, downcase und upcase standardmäßig so seltsame Buchstaben wie Umlaute oder β. Dies trifft auf viele weitere Methoden ebenfalls zu.

Möchten Sie Teile einer Zeichenkette gegen andere Zeichen austauschen, ist die Methode tr der richtige Partner für Sie. Zwei Parameter sind zum ordnungsgemäßen Betrieb nötig: Der erste gibt an, welche Buchstaben Sie bearbeiten möchten, der zweite umschreibt die Ersetzungen. Bei beiden Parametern gilt das Format, dass Sie schon bei count kennengelernt haben.

Methode an Methode

Im Programmiereralltag ist es manchmal eher hinder- denn förderlich, wenn Methoden wie index und rindex case-sensitive, also so kleinlich sind, Groß- und Kleinschreibung von Buchstaben penibel zu unterscheiden. Um mit ihnen dennoch case-insensitiv zu arbeiten und somit nach Buchstaben unabhängig ihrer Größe zu suchen, können Sie einen kleinen Trick anwenden. Wandeln Sie doch den zu durchsuchenden String komplett in Groß- oder Kleinbuchstaben um. Wenn alle Buchstaben eine einheitliche Größe haben, ist eine Fallunterscheidung schlicht nicht möglich.

Da in Ruby natürlich auch die Rückgabewerte von Methoden vollwertige Objekte sind, können Sie auf dieses Ergebnis alle in der dazugehörigen Klasse festgelegten Methoden anwenden. Dadurch können Sie eine richtige Methodenkette aufbauen, die es ermöglicht, in einer Zeile eine umfangreiche Funktionalität zu implementieren.

```
'Ruby on Rails'.index('0') # => nil  
'Ruby on Rails'.upcase.index('0') # => 5  
'Ruby on Rails'.downcase.rindex('r') # => 8  
'Ruby on Rails'.downcase.rindex('r').succ # => 9  
'Ruby on Rails'.downcase.rindex('R') # => nil
```

Um das Prinzip zu verdeutlichen: 'Ruby on Rails' ist ein String-Objekt. Es besitzt die Methode downcase, die als Ergebnis ein weiteres String-Objekt ausgibt und somit über die Methoden index und rindex verfügt. Diese beiden Methoden geben, vorausgesetzt das gewünschte Zeichen oder die gewünschte Zeichenkette wurde gefunden, ein Fixnum-Objekt zurück, auf das Sie wiederum Methoden der Klasse Fixnum anwenden könnten. Wie gesagt, könnten. Denn es ist fraglich, ob Sie wirklich die Quadratwurzel aus der Position des ersten Bs einer Zeichenkette oder dergleichen sinnvoll verwenden können.

```
'Ruby on Rails'.class # => String  
'Ruby on Rails'.downcase.class # => String  
'Ruby on Rails'.downcase.index('r').class # => Fixnum  
'Ruby on Rails'.downcase.index('R').class # => NilClass
```

Beispiel 2-16: Zeichenaustausch mit tr

```
'Ruby on Rails'.tr('a-z', '_') # => "R_____R____"  
'Ruby on Rails'.tr('^R ', '_') # => "R_____R____"  
'Ruby on Rails'.tr('uby', 'ats') # => "Rats on Rails"
```

Die delete-Methode funktioniert so ähnlich, allerdings wird hier destruktiver agiert. Wie der Name schon verrät, löscht delete Zeichen eines Strings. Die Regeln, die bestimmen, welche Zeichen es erwischen soll, sind wie bei tr und count. Mit delete können Sie also in einem Rutsch alle Vorkommen eines Zeichens oder mehrerer Zeichen aus einem String entfernen oder bestimmte Buchstaben von der Ausradierung verschonen.

Beispiel 2-17: Buchstaben entsorgen

```
'Ruby on Rails'.delete('R') # => "uby on ails"  
'Ruby on Rails'.delete('aeiou') # => "Rby n Rls"  
'Ruby on Rails'.delete('^aeiou') # => "uoai"  
'Ruby on Rails'.delete('a-s', '^bn') # => "Ruby n R"
```

Anfangs habe ich Ihnen versprochen, dass String-Objekte durchaus auch kuriose Funktionalitäten mit sich bringen. Hier sind zwei: reverse und squeeze. Frei nach dem Matthäusevangelium, »die Letzten werden die Ersten sein«, tauscht String#reverse die Reihenfolge der Buchstaben einer Zeichenkette komplett um. Die Methode squeeze lässt alle mehrfach hintereinander auftretenden Zeichen der gleichen Sorte zu einem verschmelzen. Dabei können Sie die betreffenden Zeichen mit einem Parameter, dessen Form Sie von count und delete kennen, eingrenzen.

Beispiel 2-18: Sonderbare String-Methoden

```
'Lager'.reverse # => "regal"  
'Rentner'.reverse # => "rentneR"  
'Lager'.reverse.capitalize # => "Regal"  
'Haussee'.squeeze # => "Hause"  
'Himmbeergellee'.squeeze('^e') # => "Himbeergelee"
```

Wie gesagt, Sie werden die beiden garantiert mal brauchen. Oftmals stellt ein String gar keinen Text dar, sondern beherbergt eine selbst kreierte Art von Datenstruktur, zum Beispiel für ein Spiel. Und da kann es eben doch ganz gut sein, zu wissen, wie man einen String umdreht.

Vor ein paar Absätzen haben Sie den Operator [] kennen gelernt. Mit dem konnten Sie konkrete Positionen eines Strings ansteuern und erhielten Teile der Zeichenkette oder den ASCII-Code eines Zeichens zurück. Es gibt diesen Operator auch noch in einer anderen Form, mit der Sie das ausgewählte String-Stück gegen eine andere Zeichenkette ersetzen können. Implementiert ist er als []=, Sie können ihn allerdings, um nicht die ungeschriebenen Gesetze über das Notieren von Ruby-Code zu brechen oder sich einen Sonderfall merken zu müssen, als [] = benutzen. Mit dem zusätzlichen Leerzeichen kommt Ruby problemlos klar.

Probieren Sie doch mal, folgende Zeile in *Interactive Ruby* unterzubringen. Sie sollte zur Folge haben, dass aus *Python Ruby* wird. Warum? Weil *Ruby* an die nullte Stelle des String gesetzt werden und die daran anschließenden sechs Zeichen ersetzen soll. Eigentlich.

Beispiel 2-19: Aus Python wird Ruby

```
'Python ist toll.'[0,6] = 'Ruby'
```

Sie sollten festgestellt haben, dass dieser Ausdruck nicht so ausgewertet wird, wie Sie sich das vielleicht vorgestellt haben. Auch mit einer alternativen Nutzung dieses Zuweisungsoperators ändert sich das nicht.

Beispiel 2-20 Aus Python wird Ruby – vielleicht jetzt?

```
'Python ist toll.'[0..5] = 'Ruby'
```

Bei beiden Varianten spuckt `irb` nur *Ruby*, also den Wert, den Sie zuweisen möchten, aus. Das ist auch völlig richtig so. Sie erinnern sich bestimmt: `irb` wertet Ausdrücke aus. Ein Ausdruck mit einer Zuweisung wird stets mit dem zugewiesenen Teil als Ergebnis ausgewertet. Um an die Änderungen, die der Satz *Python ist toll.* völlig zu Recht erhalten hat, zu gelangen, muss eine andere Technik her, nämlich Variablen.

Variablen und Konstanten

Es gibt sie in jeder Programmiersprache und ohne sie wäre der Alltag eines Softwareentwicklers praktisch nicht meisterbar. Und, seien Sie mal ehrlich, so richtig sinnvoll waren unsere Spiele mit den String-Literalen bislang auch nicht. Oftmals waren es String-Objekte mit gleichem Inhalt, auf die Sie nacheinander unterschiedliche Methoden angewendet haben. Das String-Literal mussten Sie dabei immer wieder neu notieren. Hätten Sie es einer *Variable* übergeben, die sich den Wortlaut gemerkt hätte, wäre Ihnen viel Tipparbeit erspart geblieben.

Variablen verweisen auf Objekte. Über den Variablenbezeichner können Sie immer wieder auf eine Variable und damit auf das Objekt, welches der Variable zugeordnet ist, zugreifen. Weisen Sie einer Variable beispielsweise ein String-Objekt zu, so können Sie alle Methoden des String-Objekts auch mit der Variable nutzen. Der Variablenname beginnt mit einem kleinen Buchstaben und darf Zahlen und Unterstriche enthalten. Anders als bei konkurrierenden Programmiersprachen erzeugen Sie eine Variable automatisch mit einer Zuweisung. Sie müssen sie nicht explizit vorher deklarieren. Wenn sie nicht mehr benötigt wird, kümmert sich *Ruby* um die Beseitigung der Variable und des Objekts, auf das sie verwiesen hat.

Probieren Sie doch einmal, die eben gescheiterte Korrektur des Satzes »*Python ist toll.*«, durch die Verwendung von Variablen zum Erfolg zu bringen. Geben Sie folgende Zeilen nacheinander in Ihr *Interactive Ruby* ein. Dabei werden Sie merken, dass auch eine Variablenzuweisung von `irb` ausgewertet und die Zuweisung als Ergebnis der Auswertung ausgegeben wird. Wenn Sie nur den Variablenbezeichner in einer Zeile angeben, dann wird die Variable ausgewertet und Sie gelangen an ihren Inhalt.

Beispiel 2-21: Und Ruby ist doch toll!

```
text = 'Python ist toll.'  
text[0..5] = 'Ruby'  
text # => "Ruby ist toll."
```

Und das ist passiert: Durch die Angabe des String-Literals *Python ist toll.* erzeugte *Ruby* automatisch ein String-Objekt, das nun via `text` dauerhaft angesprochen werden kann.

Um Ihnen zu beweisen, dass Variablen nicht nur Text, sondern auch Zahlen und jede andere Art von Objekten referenzieren können, treiben wir Beispiel 2-21 einfach noch etwas auf die Spitze. Ersetzen Sie die Zahlen der Begrenzung einfach durch Variablen, die zwei Fixnum-Objekte beinhalten.

Beispiel 2-22: Variablen als Begrenzung

```
text = 'Python ist toll.'  
a = 0  
b = 5  
text[a..b] = 'Ruby'  
text # => "Ruby ist toll."
```

Anschließlich können Sie natürlich noch weiter mit der Variable `text` arbeiten. Nehmen wir an, wir möchten die Botschaft von Rubys Tollheit durch die ausschließliche Nutzung von Großbuchstaben unterstreichen.

Beispiel 2-23: Großbuchstaben für text

```
text.upcase  
text # => "Ruby ist toll."
```

Was ist da los? Die Nutzung der `upcase`-Methode hat in der Auswertung einen großbuchstabigen Satz zur Folge. Nur der Inhalt von `text` hat sich dennoch nicht verändert. Ein Ruby-Fehler? Nein, volle Absicht. Mit der Nutzung von `upcase` haben Sie das String-Objekt angewiesen, ein neues Objekt auf Basis des bestehenden auszugeben. Somit müssen Sie `text.upcase` einer Variablen zuweisen. Dabei kann es sich durchaus wieder um `text` handeln.

Beispiel 2-24: Großbuchstaben für text (2)

```
text = text.upcase  
text # => "RUBY IST TOLL."
```

Diese Vorgehensweise ist allerdings etwas problematisch. Die Methode `upcase` erzeugt ein neues Objekt, das zwar gleich heißt, aber ein völlig anderes ist. Das können Sie erkennen, wenn Sie die `ID`, über die jedes Ruby-Objekt zu internen Zwecken verfügt, mittels `object_id` abfragen.

Beispiel 2-25: Unterschiedliche IDs = unterschiedliche Objekte

```
text = 'Ruby ist toll.'  
text.object_id # => 21026780  
text = text.upcase  
text.object_id # => 20991890
```

Auch wenn Sie andere Zahlen erhalten werden, können Sie doch erkennen, dass hier *zwei* String-Objekte erzeugt wurden. Das ist zwar prinzipiell kein Problem und in manchen Situationen durchaus erwünscht, doch es gibt für unseren Fall einen besseren Weg, der kein neues Objekt erzeugt.

Möchten Sie, dass genau das Objekt, durch das Sie die Methode aufgerufen haben, verändert wird, so fügen Sie dem Methodennamen noch ein *Ausrufungszeichen* hinzu. Viele Methoden, die Sie bis jetzt kennen gelernt haben, gibt es auch in dieser Variante – darunter auch `upcase`. Also, auf ein Neues:

Beispiel 2-26: Großbuchstaben für text (3)

```
text = 'Ruby ist toll.'  
text.object_id # => 21339810  
text.upcase!  
text # => "RUBY IST TOLL."  
text.object_id # => 21339810
```

Es gibt noch mehr, was erst mit Variablen möglich wird. Beispielsweise erleichtern sie Ihnen das Durchführen von Berechnungen.

Beispiel 2-27: Kürzere Gleichungen

```
j = 3  
j += 4 # 7, statt j = j + 4  
j -= 2 # 5, statt j = j - 2  
j *= 10 # 50, statt j = j * 10  
j /= 5 # j 10, statt j = j / 5
```

Noch etwas geht mit Variablen. Sie können mit einer Zeile gleich zwei Variablen Werte zuweisen. Außerdem gestaltet sich die Über-Kreuz-Zuweisung von Variablen in Ruby viel einfacher. Muss in anderen Programmiersprachen eine temporäre Variable eingesetzt werden, nutzen Sie hier einfach die Mehrfachzuweisung.

Beispiel 2-28: Mehrfachzuweisungen

```
m, n = 10, 40  
m # 10  
n # 40  
m, n = n, m  
m # 40  
n # 10
```

Variablen heißen vor allen Dingen deshalb Variablen, weil sie variabel sind. Es gibt aber auch Variablen, die einen konstanten Inhalt haben und deshalb nicht Variablen sondern *Konstanten* heißen.

Werte, die einer Konstante zugeordnet werden, ändern sich im Programmablauf nicht – auch wenn Ruby mögliche Versuche nur mit einer Warnung quittiert. Konstanten eignen sich daher zur Festlegung von feststehenden mathematischen Werten oder Konfigurationsparametern.

Konstanten gelten global. Das heißt, sie können ihre Werte an jedem Ort eines Ruby-Programms auslesen. Notieren Sie Konstantenbezeichner nach den Namensregeln von Variablen, benutzen Sie jedoch nur Großbuchstaben, beispielsweise `PATH = '/usr/bin/'`.

Variablen und Konstanten sind sehr wichtig für die Programmierung in Ruby und vielen anderen Programmiersprachen. Aber es gibt ein weiteres Konzept in Ruby, das dem der Variablen und Konstanten zwar ähnelt, aber doch ganz anders ist.

Symbole

Fast jede Desktop-Software bringt *Symbole*, auch Icons genannt, mit. Ob unter Windows, KDE, Gnome oder MacOS: Symbole stehen für etwas, das durch die Benutzung von kleinen Grafiken optisch illustriert wird. Das eine steht für den Start eines Texteditors, das andere lässt sie in den örtlichen Papierkorb blicken. Symbole in Ruby stehen auch für etwas, meist für einen ganz konkreten Text oder Bezeichner. Anders als Variablen verweisen Symbole nicht auf ein Objekt, so wie es weiter oben text getan hat, sondern sind selbst Objekte, Instanzen der Klasse *Symbol*. Sobald Sie ein Symbol erzeugt haben, bleibt es so lange bestehen, bis das Ruby-Programm beendet wird. Es kann auch nicht mehr verändert werden.

Ein Symbol-Literal erkennt man am Doppelpunkt, der dem Symbolbezeichner vorangeht. Der Bezeichner selbst muss den Regeln gehorchen, die Sie schon bei den Variablen kennen gelernt haben. Sollte Ihnen das nicht passen, können Sie mit String-Literalen arbeiten, wobei dort sogar ein Ausdruck enthalten sein darf, der bei der Verwendung von doppelten Anführungszeichen ausgewertet wird. Der führende Doppelpunkt bleibt jedoch verbindlich.

Beispiel 2-29: Symbole erzeugen

```
:symbol  
:'Symbol als String'  
sym = "String"  
:"Symbol als #{sym}"
```

Und jetzt das Spannende: Sie haben mit diesen Zeilen nicht drei, sondern nur zwei Symbole erzeugt, nämlich :symbol und :"Symbol als String". Glauben Sie nicht? Na dann testen wir das mal. Da Symbole Objekte sind, verfügen Sie über eine Objekt-ID. Fragen wir die einmal ab und setzen wir als Gegenbeispiel das Verhalten von String-Objekten dagegen.

Beispiel 2-30: IDs von zwei Symbolen

```
:symbol.object_id # => 157217  
:'Symbol als String'.object_id # => 158211  
sym = "String"  
:"Symbol als #{sym}".object_id # => 158211  
'Symbol als String'.object_id # => 20981287  
"Symbol als #{sym}".object_id # => 20961922
```

Den Text, den ein Symbol wortwörtlich symbolisiert, können Sie über die Methode `id2name` oder `to_s` abfragen.

Beispiel 2-31: Textentsprechung eines Symbols

```
:symbol.to_s # => "symbol"  
:symbol.id2name # => "symbol"
```

Symbole spielen für Sie zum jetzigen Zeitpunkt noch keine Rolle. Und wenn Sie das Konzept der Symbole noch nicht ganz verstanden haben, ist das kein Beinbruch. Ich habe offen gesagt auch eine Weile gebraucht, diesen Ansatz zu verstehen. Den Wert von Symbolen werden Sie im Laufe dieses Buchs noch kennen lernen. Bis jetzt sollten Sie sich nur merken, dass Symbole in Ruby einen Text oder einen Namen repräsentieren und die gesamte Laufzeit eines Programms über existieren und dass jeder Symbolbezeichner nur einmal vorkommt.

Sie kennen nun ganze und gebrochene Zahlen, Strings, Variablen, Symbole und diverse Methoden und Operatoren. Ich finde, es wird Zeit, dass Sie Ihr erstes großes – nun ja, etwas größeres Ruby-Programm schreiben. Bevor Sie damit anfangen, sollten Sie *Interactive Ruby* verlassen. Geben Sie in einer neuen Zeile *exit* ein und schließen Sie das Kommandozeilenfenster.

Für das Schreiben von Programmen ist *Interactive Ruby* nicht zu gebrauchen. Außerdem steht Ihnen schließlich eine beachtliche Entwicklungsumgebung zur Verfügung. Und die werden Sie gleich näher kennen lernen. Lassen Sie *Interactive Ruby* dennoch nicht in Vergessenheit geraten. Um kleine Teile eines Programms vor der eigentlichen Programmierung zu testen oder das Verhalten von bestimmten Methoden in gewissen Situationen auszuprobieren, ist *irb* ein sehr wertvolles Werkzeug.

Ruby-Programm 1: Der Buchstabenzähler

Ihr erstes richtiges Programm, das Sie in Ruby schreiben, soll ein simpler Auskunftsgeber über die Beschaffenheit eines Strings sein. Er soll, basierend auf einer Benutzereingabe, ausgeben, wie viele Zeichen, Vokale und Konsonanten diese hat. Dabei kommen erstmals RadRails und vier neue Methoden zum Einsatz, die sich um Ein- und Ausgabe von Daten verdient machen.

Ein neues RadRails-Projekt

Starten Sie zunächst *RadRails* und klicken Sie auf *File → New...*, wodurch sich ein Assistent öffnet, der gern wissen möchte, was Sie vorhaben. Besonders interessant sind für uns dabei natürlich die Punkte *Rails* und *Ruby*. Unter *Ruby* finden Sie wiederum die Optionen *Ruby Class* und *Ruby Project*. Da wir ein vollwertiges Programm schreiben möchten, entscheiden Sie sich für *Ruby Project*. Damit legen Sie ein neues Projekt an, das wie eine Kiste ist, in die Sie all die zum Programm gehörigen Dateien reinpacken können.

Die Arbeit in Projekten ist für so kleine Programme wie in diesem Fall möglicherweise etwas protzig. Schließlich wird unser Projekt nur eine Datei enthalten. Dennoch: Gewöhnen Sie sich ruhig die Projektarbeit an. Wenn Sie später mir Rails entwickeln, werden Sie dankbar sein, dass Ihnen RadRails die Arbeit in Projekten ermöglicht. Klicken Sie anschließend auf *Next*.

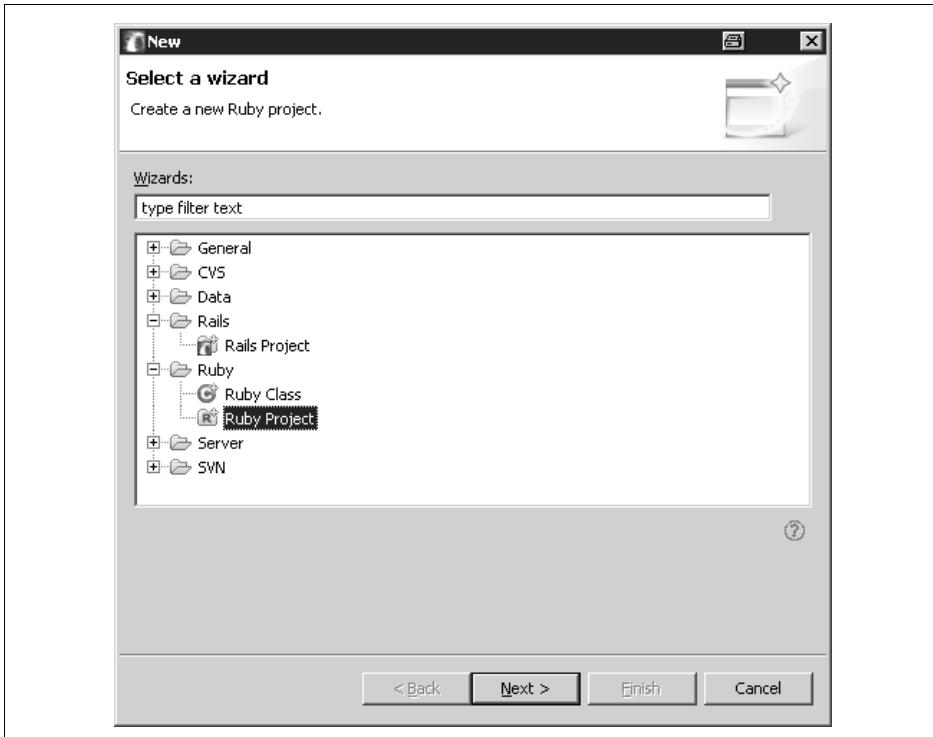


Abbildung 2-9: Ein neues Ruby-Projekt erstellen

Auf der nächsten Seite können Sie Projektnamen und Speicherpfad für Ihre Projektdateien festlegen. Geben Sie dem Projekt den Namen *Prog1*. Wählen Sie außerdem einen anderen Ort für die Speicherung Ihres Projekts aus. Standardmäßig ist hier der Pfad für Rails-Applikationen eingestellt, den wir beim ersten RadRails-Start festgelegt hatten. Da wir momentan nur an reinen Ruby-Programmen arbeiten, empfiehlt sich dieser Pfad nicht. Ich habe mich für *C:\Ruby_apps\Prog1* entschieden. Anschließend erstellen Sie mit einem Klick auf *Finish* das leere Projekt.

Im linken Bereich der Editor-Oberfläche finden Sie unter *Ruby Ressources* Ihr neu angelegtes Projekt. Noch ist es ein leerer Etwa ohne Dateien. Klicken Sie deshalb mit der rechten Maustaste auf den Projektnamen und wählen Sie dann *New*. An dieser Stelle könnten Sie Ihr Projekt auch durch neue Unterverzeichnisse organisieren oder eine neue Ruby-Klasse initiieren. Das benötigen wir aber derzeit noch nicht.

Mit einem Klick auf *File* können Sie eine neue Datei ins Projekt einbauen. Entscheiden Sie sich für *main.rb* als Dateinamen. Die Dateiendung *.rb* ist obligatorisch für Ruby-Dateien. Ruby-Programme können aus diversen *.rb*-Dateien bestehen. Eine davon sollte stets die Schaltzentrale Ihres Programms sein. In diesem Fall ist es selbstredend die Datei *main.rb*, die sich sogleich im Editor öffnet.

Sie werden zunächst Ruby-Programme schreiben, die in einer Konsole laufen und textorientiert sind. Auf Knöpfchen und Eingabefelder müssen Sie vorerst noch verzichten. Aber ich finde, gerade diese archaische Art der Programmiererei hat einen gewissen Reiz.

Die Ein- und Ausgabe von Text und sonstigen Informationen läuft also in einem kleinen Fenster ab. Dieses Fenster ist Bestandteil von RadRails, möglicherweise müssen Sie es noch einblenden. Klicken Sie dazu auf *Window* → *Show View*. Hier können Sie diverse Bereiche einblenden, die Ihnen die Arbeit mit RadRails und Ruby erleichtern. Zunächst interessiert uns aber nur *Console*.

Es geht los

Stellen wir ein paar Vorüberlegungen an: Das Programm soll einen Text entgegennehmen und die Anzahl der Vokale und Konsonanten zählen. Das Ergebnis soll anwenderfreundlich in einem Satz ausgegeben werden. Zunächst wird der zu analysierende Text in der Variable *text* gespeichert. Den Kern des Programms, das Zählen bestimmter Zeichen, wird eine Ihnen bereits bekannte Methode übernehmen: *count*. Über Parameter legen wir fest, auf welche Art von Buchstaben *count* jeweils achten soll. Damit *count* weiß, was Vokale sind, hinterlegen wir eine entsprechende Sammlung von Buchstaben als Variable. Nach der Zählung soll ein Satz ausgegeben werden, der als String mit auszuwertenden Ausdrücken realisiert wird. Auf der Basis dieser Überlegungen notieren wir folgende erste Zeilen in *main.rb*.

Beispiel 2-32: Version 1

```
text = "Ruby on Rails"
vowels = 'aeiou'
v_count = text.count(vowels)
c_count = text.count("^#{vowels}")
%Q{In "#{@text}" sind #{v_count} Vokale und #{c_count} Konsonanten enthalten.}
```

Die Variable *vowels* enthält ein String-Objekt mit allen Vokalen. Auf dieser Basis ermittelt *count* die Anzahl von Vokalen und die Anzahl von Nicht-Vokalen, letztere gekennzeichnet durch *^*. Die Ergebnisse werden den Variablen *v_count* beziehungsweise *c_count* übergeben, welche in der letzten Zeile als Ausdrücke in einem String mit zwei Anführungszeichen, symbolisiert durch *%Q{}}, platziert werden.*



Grundsätzlich sollten Sie nur eine Anweisung pro Zeile notieren. Möchten Sie mit dieser Konvention beim Verfassen von Ruby-Code brechen, dann beenden Sie einfach jede Anweisung mit einem Semikolon. Aber bedenken Sie: Das kann schnell unübersichtlich werden.

Beim Eingeben dieser Zeilen werden Sie sicher schon bemerken, dass RadRails Sie genau beobachtet. Aber nur aus reiner Nettigkeit. Zeilen, die während der letzten Speicherung verändert wurden, werden an der linken Seite mit einem kleinen grauen Balken markiert. Findet RadRails innerhalb einer Zeile eine Ungereimtheit, werden Sie auf diesen Missstand umgehend durch einen rot gefüllten Kreis mit weißem Kreuz hingewiesen.

Sobald Sie den Quelltext gespeichert haben, können Sie den ersten Testlauf starten. Klicken Sie den Tab *main.rb* und anschließend *Run → Run As → Ruby Application* an. Das Programm läuft ab. Sorgen Sie nun noch dafür, dass Sie die *Console* einsehen können, denn dort sehen Sie das Ergebnis des Programms.

Seien Sie nicht enttäuscht, wenn es dort aber noch nichts zu lesen gibt. Es gibt einen Grund dafür: In der letzten Zeile unseres bisherigen Quelltextes haben wir einen String notiert, dessen Auswertung in Interactive Ruby noch klaglos ausgegeben wurde. Hier nicht. Da wir aber nun keine Ausdrücke mehr auswerten, sondern so richtig programmieren, bestimmen wir selbst, was wann wo ausgegeben werden soll. Das heißt, dass dezidierte Abweisungen zum Ausgeben von Daten im Programm enthalten sein müssen. Wäre ja auch nicht auszudenken, wenn uns Ruby-Programme ständig mit allerlei Ausdrucksauswertungen belästigen würden.

Es gibt zwei Befehle in Ruby, die etwas auf den Bildschirm schreiben können: `puts` und `print`. Während `puts` nach der Ausgabe der gewünschten Daten einen Zeilenumbruch vollführt, verzichtet `print` darauf. Für unser kleines Programm entscheiden wir uns für `puts`. Die Änderungen im Quelltext sind fett gedruckt.

Beispiel 2-33: Version 2

```
text = "Ruby on Rails"
vowels = 'aeiou'
v_count = text.count(vowels)
c_count = text.count("^#{vowels}")
puts "%Q{In \"#{text}\" sind #{v_count} Vokale und #{c_count} Konsonanten enthalten.}
```

Vergessen Sie nicht, *main.rb* abzuspeichern. Dann können Sie einen erneuten Start Ihres Ruby-Programms wagen. Und siehe da: Die Console spuckt eine Meldung aus. Allerdings stimmt die nicht ganz – im Ausdruck »Ruby on Rails« sind keinesfalls 9 Konsonanten enthalten.

Der Grund für diese Falschmeldung liegt klar auf der Hand: Wenn `count` nach Zeichen suchen soll, die nicht Vokale sind, dann werden natürlich auch Leerzeichen mitgezählt. Daher müssen wir sicherstellen, dass bei den Konsonanten die zu

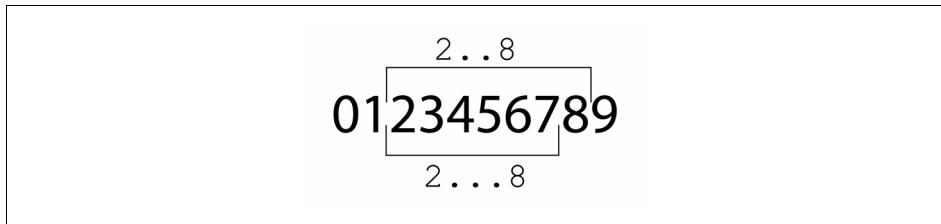


Abbildung 2-10: Ein Bug im Programm?

berücksichtigenden Zeichen auf das Alphabet beschränkt werden. Das erreichen wir mit einer zweiten Regel für die konsonantenzählende Methode `count`, die die zu berücksichtigenden Zeichen auf die Buchstaben des Alphabets einschränkt.

Hätten wir Vokale in Form von Großbuchstaben in unserem String-Objekt `text`, würden diese momentan auch als Konsonanten gezählt. Fälschlicherweise. Daher müssen wir das Zählen case-insensitive gestalten, indem alle Zeichen in `text` zum Zählen in Kleinbuchstaben verwandelt werden – die Vokale in `vowels` stehen auch in dieser Form zur Verfügung, das passt.

Beispiel 2-34: Version 3

```
text = "Ruby on Rails"
vowels = 'aeiou'
v_count = text.downcase.count(vowels)
c_count = text.downcase.count("^#{vowels}", 'a-z')
puts %Q{In "#{text}" sind #{v_count} Vokale und #{c_count} Konsonanten enthalten.}
```

Jetzt stimmen die Angaben. Gönnen wir uns abschließend ein bisschen Luxus, indem wir das Programm so umbauen, dass der zu untersuchende Text nicht statisch im Programm steht, sondern durch den Benutzer eingegeben werden kann. Um dies zu realisieren, steht Ihnen die Methode `gets` zur Verfügung. Das `s` bei `gets` steht, genau wie bei `puts`, für *String*. Das heißt: Das, was `gets` aufnimmt, liegt anschließend als String-Objekt vor. Somit kann man sagen, dass `gets` wie ein String-Objekt behandeln können. Dabei ist zu beachten, dass `gets` so lange die eingegebenen Zeichen in sich aufsaugt, bis der Benutzer die *Enter*-Taste drückt. Damit der Benutzer überhaupt weiß, dass er etwas eingeben muss, setzen wir noch eine entsprechende Aufforderung vor der eigentlichen Eingabe in das Programm.

Beispiel 2-35: Version 4

```
print "Bitte geben Sie ein Wort ein: "
text = gets
vowels = 'aeiou'
v_count = text.downcase.count(vowels)
c_count = text.downcase.count("^#{vowels}", 'a-z')
puts %Q{In "#{text}" sind #{v_count} Vokale und #{c_count} Konsonanten enthalten.}
```

Die Aufforderung, ein Wort einzugeben, wurde hier mit `print` realisiert. Wie Sie bereits wissen, entsteht bei der Ausgabe mit `print` kein abschließender Zeilenumbruch. Das bedeutet: Sämtliche Eingaben via `gets` erfolgen exakt hinter dem Doppelpunkt.

Speichern Sie Ihre Änderungen – aber führen Sie das Programm jetzt nicht aus. Es gibt zwei Gründe, weshalb Sie das nicht machen sollten: 1. Die RadRails-Console hat ein paar grundsätzliche Schwierigkeiten mit Eingaben, was Ihnen 2. die Gelegenheit eröffnet, zu sehen, wie Ruby-Programme normalerweise, also ohne die Hilfe von RadRails, gestartet werden.

Öffnen Sie dazu noch einmal die Kommandozeile Ihres Betriebssystems – also zum Beispiel `cmd.exe` – und wechseln Sie in das Verzeichnis, in dem sich Ihre `main.rb` befindet. Bei mir ist dies, wie oben bereits erwähnt, `C:\Ruby_apps\Prog1`. Geben Sie dann einfach `ruby main.rb` ein – und schon geht's los. Wir teilen hier dem Ruby-Interpreter (`ruby.exe`) lediglich mit, dass dieser die quelltexthaltige Datei `main.rb` interpretieren und ausführen soll. Und das macht er dann auch umgehend.

Ein Problem gibt es noch: Der Satz, der uns über die Inhaltsstoffe des eingegebenen Wortes informieren soll, beinhaltet einen Zeilenumbruch. Die Ursache dafür liegt bei `gets`. Das Drücken der `Enter`-Taste, das die Eingabe beendet, wird von `gets` ebenfalls aufgezeichnet und befindet sich im String-Objekt. Das muss entfernt werden, schließlich bedeutet dies einen ungewollten Zeilenumbruch. Für solche Aufgaben eignet sich die Methode `chomp`, die jedem String-Objekt beiliegt. Durch die Verwendung von `chomp` schneiden Sie das Enter-Zeichen am Ende eines Strings, sofern vorhanden, ab. Diese Methode wird oft in Verbindung mit `gets` verwendet. Fügen wir dem Resultat von `gets` noch `chomp` hinzu, dann ist unser Programm erst einmal fertig.

Beispiel 2-36: Version 5

```
print "Bitte geben Sie ein Wort ein: "
text = gets.chomp
vowels = 'aeiou'
v_count = text.downcase.count(vowels)
c_count = text.downcase.count("^{#{vowels}}", 'a-z')
puts %Q{In "#{text}" sind #{v_count} Vokale und #{c_count} Konsonanten enthalten.}
```

Kritisch wird es nur, wenn lediglich ein Vokal oder Konsonant im eingegebenen Wort enthalten ist, da der Satz durch die Verwendung des Plurals auf mindestens zwei jeder Sorte zugeschnitten ist. Das klingt dann etwas holprig. Um dies zu vermeiden, müsste ein Satz, der abhängig von den Werten in `c_count` und `v_count` gestaltet ist, ausgegeben werden. Auch dafür gibt es Mittel und Wege. Fallunterscheidungen oder Programmweichen – um bei der Schiene zu bleiben – sind elementare Bestandteile und somit auch in Ruby enthalten.

Noch bevor Sie das nächste Beispielprogramm in Ruby schreiben, werden Sie die entsprechenden Konstrukte in Ruby kennen lernen. Zuvor jedoch soll es um Themen gehen, die bei der Programmierung von mindestens gleich großer Bedeutung sind.

Von von bis bis – Wertebereiche

Es ist noch gar nicht so viele Seiten her, da hatten Sie bereits Kontakt mit *Wertebereichen*. Die Methode `slice` beziehungsweise die Operatoren `[]` und `[]=` der String-Klasse können einen ganz bestimmten Abschnitt eines Strings wiedergeben oder verändern. Den Beginn und das Ende dieses Abschnitts können Sie durch die Angabe eines Bereichs, also mit einem Objekt der Klasse `Range`, eingrenzen. Sie erinnern sich?

Beispiel 2-37: Abschnitte eines Strings festlegen

```
'Bereiche sind eine Bereichserung'[0..7] # => "Bereiche"  
'Bereiche sind eine Bereichserung'[19..26] # => "Bereiche"
```

Schauen wir uns nun die Range-Klasse etwas genauer an. Wie Sie sehen, werden Range-Objekte wie schon Strings mittels eines bestimmten Literals, einer bestimmten Art der Notierung, erzeugt. Sie geben einfach einen auch durchaus negativen Start- und einen ebensolchen Endwert an und setzen *zwei Punkte* dazwischen. Der so definierte Bereich enthält alle Zahlen, die zwischen dem Start- und Endwert liegen, wobei der Startwert kleiner als der Endwert sein muss. Bei der Variante mit zwei Punkten sind beide begrenzenden Werte in der Range enthalten.

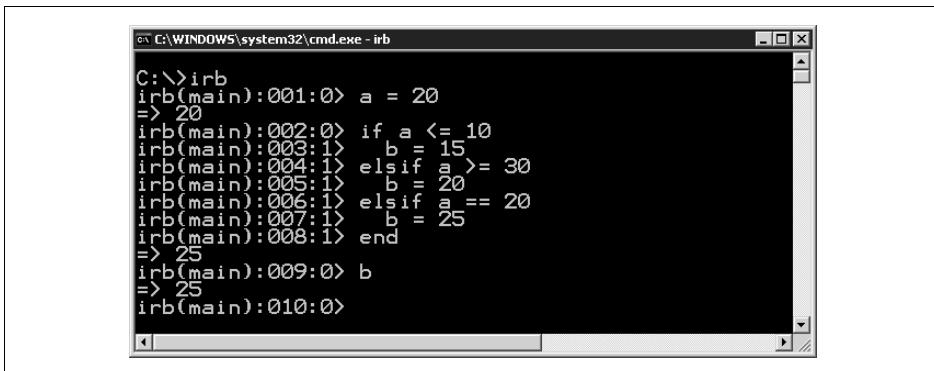
A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe - irb'. The window contains an Interactive Ruby session (irb). The user has defined a variable 'a' with the value 20. They then define a range 'b' from 15 to 25. An if-elsif-else block is used to set 'b' based on the value of 'a'. If 'a' is less than or equal to 10, 'b' is 15. If 'a' is greater than or equal to 30, 'b' is 20. Otherwise, 'b' is 25. The range 'b' is then printed, showing it includes both 25 and 25, which is highlighted in red in the screenshot.

Abbildung 2-11: Von 2 bis 8

Wie Sie diesem bedeutungsschwangeren Satz schon entnehmen können, gibt es auch noch eine andere Variante. Und zwar eine mit *drei Punkten* zwischen den beiden Zahlen. Hierbei ist der Endwert *exklusiv* und gehört somit nicht zu dem Zahlenspektrum, den das entsprechende Range-Objekt umschreibt. Schauen Sie selbst – vielleicht, indem Sie noch einmal Interactive Ruby starten?

Beispiel 2-38: ... und ...

```
'Bereiche sind eine Bereicherung' [19..26] # => "Bereiche"  
'Bereiche sind eine Bereicherung' [19...26] # => "Bereich"
```

Sie müssen Ranges übrigens nicht ausschließlich als Mittel zur Definition eines spezifischen Zahlenbereichs verstehen. Sie können ein Range-Objekt auch dafür nutzen, um ein Intervall, zwei inhaltlich zusammengehörende Zahlen oder ähnliche Informationen zu speichern, bei denen es völlig egal ist, ob und welche Zahlen zwischen den beiden Begrenzungen liegen. In solchen Fällen können Sie auch gebrochene Zahlen als Start- oder Endwert nutzen.

Beispiel 2-39: Ranges erzeugen

```
a = 1..6  
b = 1...6  
c = 4.225...90.61
```

Range-ieren

Die Klasse Range enthält einige ganz brauchbare Methoden, aber ihre eher überschaubare Anzahl ist mit der beispielsweise der String-Klasse überhaupt nicht vergleichbar. Besonders interessant für das eben geschilderte Einsatzszenario von Ranges als Intervalgrenzen sind dabei die Methoden *first* und *begin*, sowie *last* und *end*. Damit erhalten Sie die erste beziehungsweise letzte Zahl einer Range. Dabei ist unerheblich, ob der Bereich mit zwei oder drei Punkten gebildet wurde. Das Ergebnis von *last* oder *end* stellt immer den angegebenen Endwert dar.

Beispiel 2-40: Anfang und Ende

```
a.first # => 1  
a.last # => 6  
b.first # => 1  
b.end # => 6  
a.begin # => 1  
c.last # => 90.61  
c.end + a.first # => 91.61  
c.end - c.first # => 86.385
```

Bleiben wir doch gleich bei unseren drei Variablen a, b und c. So einem Buchstaben, der für ein Range-Objekt steht, sieht man ja nicht unbedingt an, ob der Endwert nun innerhalb des Bereichs liegt oder nicht. Aber wir können ihn ja mal fragen. Das Zauberwort heißt *exclude_end?* – es gibt true zurück, wenn der Bereich sozusagen dreipunklig ist; dann ist der Endwert nicht enthalten.

Beispiel 2-41: Exklusive Endwert?

```
a.exclude_end? # => false  
b.exclude_end? # => true
```

Die Methode `include?`, der Sie schon in der Klasse `String` begegnet sind, gibt es auch für Ranges. Mit ihrer Hilfe können Sie abfragen, ob sich ein bestimmter Wert innerhalb des Wertebereichs befindet. Bei dem Ergebnis `true` ist dies der Fall. Übrigens: `include?` berücksichtigt, ob ein Endwert zum Bereich gehört oder nicht.

Beispiel 2-42: Drin oder nicht drin?

```
a.include?(6) # => true  
b.include?(6) # => false  
c.include?(90.61) # => false  
c.include?(90.6099999) # => true
```

Alternativ zu `include?` können Sie bei Range-Objekten auch den Operator `==` nutzen. Ganz genau, *drei* Gleichheitszeichen! Sehr wichtig!

Von anderen Klassen kennen Sie bereits Methoden zum Konvertieren von Objekten. So etwas gibt es natürlich auch in der Range-Klasse, wenngleich eingeschränkt, was in der Natur der Sache liegt: Einen Bereich aus mehreren Zahlen allgemein gültig in eine zu verwandeln, fällt schwer. Mit `to_s` können Sie das Range- immerhin in ein String-Literal verwandeln. Und `to_a` spuckt einen Array aus, der alle Werte des Bereichs als Einzelelemente enthält, abhängig von ... oder ... natürlich.

Beispiel 2-43: Konvertierte Bereiche

```
"Die Range #{a.to_s} beinhaltet Werte von #{a.first} bis #{a.last}." # => "Die Range  
1..6 beinhaltet Werte von 1 bis 6."  
a.to_a # => [1, 2, 3, 4, 5, 6]  
b.to_a # => [1, 2, 3, 4, 5]
```

Sie können auch gern versuchen, aus `c` ein Array zu generieren. Das wird Ihnen aber nicht gelingen, denn `to_a` braucht dringend ganze Zahlen als Ausgangsbasis, um in Einerschritten Array-Elemente zu erzeugen. Für gebrochene Zahlen gibt es einfach keine Regeln, um sinnvoll aus ihnen ein Array zu bilden. Denken Sie beispielsweise daran, dass auch die Methode `succ` nur für ganze Zahlen der Klassen `Bignum` und `Fixnum` definiert ist. Aus diesem Grund gibt es übrigens auch keine Methode, die Ihnen die Anzahl der in einer Range enthaltenen Werte zurückgibt. Die Methoden `size` und `length`, die in früheren Ruby-Versionen noch zu Range gehörten, fielen Matz' Logikverständnis und seinem Wunsch, eine glasklare Sprache zu entwickeln, zum Opfer.



Wie Sie dennoch die Anzahl von Werten einer Range mit ganzzahligen Start- und Endwerten ermitteln können, erfahren Sie ein paar Absätze weiter.

Sollten Sie sich die ganze Zeit fragen, »Moment mal, was ist eigentlich ein Array?«, dann erhalten Sie gleich alle nachfolgenden Informationen. Mit `to_a` haben Sie aber zumindest schon eine Variante kennen gelernt, wie Sie ein Array auf Basis von konkreten Daten bilden können.

Elementares über Arrays

Genau genommen haben Sie eben sogar schon ein typisches Charakteristikum von *Arrays* kennen gelernt: die eckigen Klammern. Sie sind das, was Anführungszeichen für String- und der Punkt für Float-Objekte sind. Und daher lassen sich Arrays nicht nur über die Methode `to_a` der Range-Klasse bilden, sondern ganz einfach über *Array-Literale*.

Beispiel 2-44: Arrays erzeugen

```
numbers = [1, 2, 3, 4]
chars = ['A', 'B', 'C']
strings = ["#{2 + 5} Birnen", "300g Butter", "2kg Zimt"]
arrays = [numbers, chars, strings, 5, :mixed]
empty_arr = [] # leeres Array, nicht mit dem String-Operator verwechseln!
```

Sie sehen, so ein Array ist flexibel wie ein Gummibärchen: Alles Mögliche können Sie in ein Array hineinquetschen. Und so müssen Sie ein Array auch betrachten – insbesondere bei Ruby.

Um es noch plastischer zu machen: Sehen Sie ein Array einfach als Koffer. In einem solchen Gepäckstück können Sie alles, was Ihnen wichtig ist, verstauen, es transportieren und dann wieder auspacken. Es soll ja Menschen geben, die packen ihren Koffer am Zielort nicht aus, sondern greifen sich einfach nur das heraus, was sie jeweils benötigen. Auch das geht mit Ruby. Übrigens: Koffer in einigen anderen Programmiersprachen können oft nur eine Sorte von Kleidung enthalten. Dort müssen Sie sogar vorab dezidierte Angaben machen, dass beispielsweise nur Socken hinein dürfen.

Bei Ruby ist das anders: Da können Sie sogar Koffer mit Koffern in einem Koffer verstauen. Oder eben Arrays in Arrays. So enthält die Variable `arrays` aus dem obigen Beispiel die Arrays `numbers`, `chars` und `strings`. Darüber hinaus stecken aber auch noch eine Fixnum-Zahl und ein Symbol dort drin, also Objekte, die keine Arrays sind.

Ach ja, fast hätte ich es vergessen zu erwähnen – aber Sie können es sich ja eh schon denken: Arrays können nicht nur diverse verschiedene Objekte aufnehmen, sie sind natürlich auch selbst welche. So wie das eben in Ruby ist. Die Klasse, nach deren Vorbild sie gebildet werden, heißt *Array*.

Das Array-Einmaleins

Bevor Sie aber die in der Array-Klasse deklarierten Methoden kennen lernen, möchte ich Ihnen ein paar grundlegende Techniken beim Umgang mit Arrays anhand einiger Beispiele zeigen. Dabei werden Sie gleich zwei weitere Wege kennen lernen, Arrays zu erstellen.

Über die Klassenmethode `new` der Array-Klasse können Sie ein Array-Objekt erzeugen. Möchten Sie es zugleich mit Werten füllen, stehen Ihnen zwei optionale Parameter zur Verfügung. Der erste beinhaltet die *Menge an Elementen*, die das Array zunächst haben soll, und der zweite, welchen Wert jedes dieser Elemente *standardmäßig* erhalten soll.

Außerdem können Sie die Elemente eines Arrays aus einem String speisen. Alle dort enthaltenen Leerzeichen gelten als Abgrenzung zwischen einzelnen Elementen. Die Leerzeichen werden nicht in das Array aufgenommen. Damit aus dem String ein Array wird, schließen Sie ihn in `%w{}` ein und behandeln Sie ihn so, als würden Sie ihm mit `%q{}` Anführungszeichen verpassen wollen.

Etwas flexibler als `%w()` ist die Methode `split`, die jedem String-Objekt beigelegt wird. Auch sie wandelt einen String in ein Array um, wobei Sie hier explizit festlegen können, wie das Zeichen, das im String die einzelnen Elemente abtrennt, lautet.

Beispiel 2-45: Noch mehr Arrays erzeugen

```
numbers2 = Array.new(3, 4) # => [4, 4, 4]
chars2 = %w{C D E} # => ["C", "D", "E"]
strings2 = "Currywurst, Pommes, Cola".split(", ") # => ["Currywurst", "Pommes", "Cola"]
```

Lassen Sie uns mit den eben erstellten Arrays experimentieren. Sie werden feststellen, dass Ruby einige sehr interessante Funktionalitäten für den Umgang mit Arrays bereithält, und zwar Funktionalitäten, die in anderen Programmiersprachen weder enthalten sind noch mal eben auf die Schnelle nachprogrammiert werden können. Ruby macht es Ihnen sehr einfach.

Um die Differenz zweier Arrays zu bilden, brauchen Sie nur das eine vom anderen abzuziehen – als ob es Zahlen wären. Hierbei steht Ihnen das Minuszeichen als Operator zur Verfügung. Als Ergebnis erhalten Sie ein neues Array-Objekt, das nur die Elemente des ersten Arrays enthält, welche im zweiten nicht enthalten sind.

Wenn Sie zwei Arrays zusammenkoppeln möchten, dann steht Ihnen das Pluszeichen hilfreich zur Seite. Ähnlich der Konkatenation mehrerer Strings erhalten Sie als Ergebnis ein neues Objekt, das zunächst alle Elemente des ersten Arrays und anschließend die des zweiten beinhaltet. Dabei ist es völlig unerheblich, ob dadurch doppelte Elemente entstehen.

Möchten Sie das ausschließen, benutzen Sie das `|`-Zeichen (Pipe). Es fügt ebenfalls mehrere Arrays in der Reihenfolge ihrer Nennung zusammen, entfernt jedoch doppelte Einträge. Mit dem Sternchen und einem Faktor können Sie ein und dasselbe Array vervielfältigen.

Eine Schnittmenge von mindestens zwei Arrays erhalten Sie durch die Verwendung des kaufmännischen Und-Zeichens (`&`). Das Ergebnis enthält nur Elemente, die in allen beteiligten Arrays enthalten sind.

Beispiel 2-46: Mengenlehre à la Ruby-Arrays

```
chars - chars2 # => ["A", "B"]
chars2 - chars # => ["D", "E"]
numbers + numbers2 # => [4, 4, 4, 1, 2, 3, 4]
numbers | numbers2 # => [1, 2, 3, 4]
chars2 | chars # => ["C", "D", "E", "A", "B"]
chars * 3 # => ["A", "B", "C", "A", "B", "C", "A", "B", "C"]
chars & chars2 # => ["C"]
numbers2 & numbers # => [4]
numbers & chars # => []
```

Mit diesen Verknüpfungen ist die Leistungsfähigkeit von Ruby-Arrays natürlich noch längst nicht erschöpft. Werfen wir einen Blick auf die Methoden, die Arrays mitbringen.

Array-Methoden

Viele Methoden der Array-Klasse kennen Sie bereits von den Strings. So auch diese beiden: Um zu erfahren, wie viele Elemente ein Array enthält, können Sie die Methoden `length` oder `size` befragen. Sie zählen für Sie durch. Array-Elemente, die selbst ein Array sind, werden dabei als ein Element gezählt.

Beispiel 2-47: Anzahl von Elementen eines Arrays

```
arrays.length # => 5
empty_arr.size # => 0
```



Vielleicht ging Ihnen gerade ein Licht auf, was das Zählen der Werte eines Bereichs angeht: Ranges, die ganzzahlige Start- und Endwerte haben, können mit `to_a` in Arrays umgewandelt werden, wobei jeder Wert der Range ein Array-Element wird. Und mit `length` können Sie die Anzahl von Array-Elementen zählen lassen. Packen Sie beides zusammen, erhalten Sie eine Lösung für unser offenes Problem: Behandeln Sie eine Range zum Zählen einfach als ein Array: `range_size = range.to_a.length`.

Jedes Array-Element besitzt einen Index. Das ist eine fortlaufende Nummerierung, die bei 0 beginnt. Somit trägt das erste Array-Element stets den Index `0`, das zweite `1` und so weiter. Bei einer Zeichenkette können Sie via `[]` auf einzelne oder mehrere Buchstaben zugreifen. Bei Arrays greifen Sie so auf Elemente zu. Platzieren Sie den gewünschten Index einfach zwischen den eckigen Klammern. Sie erhalten das entsprechende Objekt umgehend zurück.

Um mehrere Elemente zu erhalten, können Sie mit Ranges arbeiten. Oder Sie geben, und auch das ist beim String ähnlich, einen Startindex und die Menge an Elementen an, die Sie interessieren. Beachten Sie, dass die Anforderung mehrerer Elemente zwangsläufig zu einem Ergebnis führt, das ebenfalls ein Array ist. Außer-

dem gilt auch hier: Alternativ können Sie die Funktion `slice` benutzen. Möchten Sie gezielt auf das erste oder letzte Element eines Arrays zugreifen, können Sie die Methoden `first` und `last` benutzen. Und: Mit `[]`= können Sie auf den Inhalt des Arrays gezielt einwirken und einzelne oder mehrere Elemente verändern oder gegen andere Objekte eintauschen.

Beispiel 2-48: [], slice und []=

```
numbers[0] # => 1
numbers[1..3] # => [2, 3, 4]
numbers[2, 1] # => [3]
numbers.first # => 1
numbers.slice(1, 2) # => [2, 3]
numbers.slice(1, 2).last # => 3
numbers2[1..2] = 2 # => numbers2 = [4, 3]
```

Auch bei den Arrays gibt es den umgekehrten Weg. Um zu erfahren, an welcher Stelle sich ein bestimmtes Element innerhalb eines Arrays befindet, können Sie `index` befragen. Als Parameter geben Sie einfach das gesuchte Objekt an. Als Ergebnis erhalten Sie die Position des ersten Vorkommens dieses Objekts. Die Methode sucht dabei von links nach rechts. Mit `rindex` können Sie von rechts nach links suchen lassen. Denken Sie daran, dass das erste Element eines Arrays den Index 0 trägt.

Um nicht nur das erste Auftreten eines Objekts innerhalb eines Arrays zu erfragen, sondern über alle Vorkommen Bescheid zu wissen, können Sie die Methode `values_at` nutzen. Sie sucht nach dem entsprechenden Objekt und gibt, auch wenn das Objekt nur einmal vorkommt, ein Array mit Werten zurück. Dieses Array enthält alle Indizes von links nach rechts jeweils als einzelnes Element.

Beispiel 2-49: Erste und letzte Vorkommen von Elementen in Arrays

```
[20, 10, 20, 10, "10", 30].index(10) # => 1
[20, 10, 20, 10, "10", 30].rindex(10) # => 3
[20, 10, 20, 10, "10", 30].rindex("10") # => 4
```

Um zu erfahren, ob ein bestimmtes Element überhaupt in einem Array enthalten ist, können Sie die Methode `include?` befragen. Wenn Sie so nett sind und ihr das fragliche Objekt als Parameter übergeben, schaut sie sofort nach. Im Erfolgsfall meldet sie sich mit `true` zurück.

Es gibt weitere Methoden der Array-Klasse, die Sie bereits von Strings kennen. Zum Beispiel die zum Zusammenführen von Arrays. Mit `concat` können Sie ein Array an ein anderes anhängen. Hier sei ein weiteres Mal auf den Unterschied zwischen `concat` und dem Pluszeichen hingewiesen: Während `+` zwei Arrays aneinander heftet, diese aber unverändert lässt und ein völlig neues Array-Objekt ausgibt, wird bei `concat` das erste Array, durch das der Methodenaufruf erfolgt, durch das zweite ergänzt und somit verändert. Auf dabei entstandene doppelte Elemente achtet `concat` nicht.

Mit `<<` und `push` bietet Ihnen die Array-Klasse noch zwei weitere Werkzeuge, um bestehenden Arrays etwas anzuhängen. Aber Achtung: Beide verhalten sich etwas anders als `concat`. Während Sie mit `concat` den Inhalt eines ganzen Arrays hinzufügen, geht das mit `<<` und `push` nur elementweise. Verwenden Sie dennoch ein Array als Parameter, so erhält das Zielarray eben ein neues Element, welches selbst ein Array ist.



Es gibt noch einen weiteren Weg, ein einzelnes Element an ein Array zu hängen: Sagen wir, ein bestehendes Array, nennen wir es `x`, ist von Index 0 bis 5 mit Elementen ausgestattet. Dann können Sie mit `x[6] = "Neues Element"` ganz einfach ergänzen.

Um mehrere Elemente auf einmal mittels `push` oder `<<` an ein Array zu hängen, trennen Sie die einzelnen Objekte durch ein Komma

Beispiel 2-50: `concat`, `push` und `<<`

```
numbers2.concat([6, 5]) # => [4, 3, 6, 5]
numbers2.concat(numbers) # => [4, 3, 6, 5, 1, 2, 3, 4]
numbers << 5 # => [1, 2, 3, 4, 5]
numbers.push(12, 9) # => [1, 2, 3, 4, 5, 12, 9]
strings2[3] = "Ketchup" # => strings2 = ["Currywurst", "Pommes", "Cola", "Ketchup"]
```

Die Methode `push` bietet Ihnen übrigens auch einen Rückgabewert, konkret das komplette, frisch ergänzte Array, an. Was Sie mit `push` am Ende eines Arrays anfügen, können Sie mit `pop` auch wieder entfernen. Allerdings entfernt `pop` pro Aufruf nur ein Element. Die Methode stellt Ihnen aber netterweise das entfernte Element zur Verfügung. Sollte der Rückgabewert `nil` sein, so konnte `pop` nichts entfernen. Mit `push` und `pop` können Sie so ein Datenmodell realisieren, das in der Informatik *Stack* oder *Stapel* heißt.

Dank den Methoden `shift` und `unshift` können Sie so eine Stapelei auch am Anfang eines Arrays durchführen: Statt `pop` benutzen Sie dann einfach `shift`, statt `push` `unshift`. Denken Sie daran, dass Sie mit `push`, `pop`, `shift` und `unshift` direkte Änderungen am betroffenen Array vornehmen.

Beispiel 2-51: Methoden für Stapel

```
numbers.pop # => 9
numbers # => [1, 2, 3, 4, 5, 12]
numbers.shift # => 1
numbers # => [2, 3, 4, 5, 12]
numbers.unshift(1) # => [1, 2, 3, 4, 5, 12]
```

Die nächste Methode, die ich Ihnen präsentieren möchte, hört auf den Namen `uniq`. Sie prüft ein Array auf doppelte Einträge und gibt eines aus, das keine mehr enthält. Momentan hat das Beispiel-Array `numbers2`, das uns schon die ganze Zeit begleitet, einige doppelt auftretende Fixnum-Objekte. Mit `uniq` können Sie das ändern. Oder noch besser: Sie benutzen `uniq!` und erinnern sich gleichsam daran, was das Ausru-

fungszeichen am Ende eines Methodennamens bewirkt: Ohne ! wird auf Basis des bestehenden Arrays ein neues – inklusive Änderungen – erzeugt und zurückgegeben. Mit ! werden die Änderungen direkt in dem Objekt vorgenommen, das die Methode aufgerufen hat.

Beispiel 2-52: uniq macht Array-Elemente einzigartig

```
numbers2.uniq # => [4, 3, 6, 5, 1, 2]
numbers2 # => [4, 3, 6, 5, 1, 2, 3, 4]
numbers2.uniq! # => [4, 3, 6, 5, 1, 2]
numbers2 # => [4, 3, 6, 5, 1, 2]
```

Mit reverse können Sie die Reihenfolge der Elemente eines Arrays umdrehen. Auch von reverse gibt es eine Variante mit Ausrufungszeichen. Gleiches gilt für sort. Diese Methode sortiert die Elemente eines Arrays. Beachten Sie hierbei, dass diese Methode wirklich ihr bestes versucht, aber bei Arrays mit unterschiedlichsten Objekten gelingt es mitunter nicht. Am besten, Sie wenden diese Methode nur auf homogene Arrays an.

Beispiel 2-53: Reihenfolge von Array-Elementen ändern

```
numbers2.reverse! # => [2, 1, 5, 6, 3, 4]
numbers2.sort! # => [1, 2, 3, 4, 5, 6]
```

Und wenn Ihnen trotz Umdrehen und Sortieren alles zu bunt wird, dann löschen Sie doch den Inhalt eines Arrays. Entweder komplett, das macht die parameterlose Methode clear, oder aber elementweise, wofür delete und delete_at ganz gut geeignet sind.

Mit delete können Sie ein oder mehrere Elemente löschen, indem Sie dieses oder diese als Parameter übergeben. Bei delete_at funktioniert das über die Angabe eines konkreten Index. Somit ist auch klar, dass Sie mit delete_at nur ein Zeichen auf einmal ersatzlos streichen können. Mit der als Frage gestellten Methode empty? können Sie ermitteln, ob ein Array leer ist. Als Antwort erhalten Sie true oder false. Übrigens: Selbst wenn ein Array leer ist, existiert es. Sie können also so oft clearen, wie Sie wollen, das Array-Objekt existiert weiterhin.

Beispiel 2-54: Array-Elemente löschen

```
strings.delete("7 Birnen") # => "7 Birnen"
strings # => ["300g Butter", "2kg Zimt"]
strings.delete_at(0) # => "300g Butter"
strings # => ["2kg Zimt"]
strings.clear # => []
strings.empty? # => true
```

Statt delete_at können Sie auch slice! verwenden. So können Sie nicht nur Teile eines Arrays ausgeben, wie es die ausrufungszeichenlose Variante slice bereits vermag, sondern diese Teile auch noch aus dem Array ausschneiden.

Arraykadabra!

Werfen wir nun, fast schon traditionellerweise, einen Blick auf Konvertierungsmethoden. Allerdings bietet sich dabei, wie schon bei Ranges, ein eher tristes Bild. Was soll aus einem Haufen aneinander gereihter Objekte auch Sinnvolles entstehen? Aber zu einem String bekommen Sie ein Array recht leicht umgebogen. Die Methode `to_s` reiht von links nach rechts alle Elemente eines Arrays aneinander und gibt dies als String-Objekt aus. Einen ähnlichen Weg geht die Methode `join`. Hier können Sie allerdings noch einen String angeben, der zwischen die einzelnen Elemente gesetzt wird. Belassen Sie `join` parameterfrei, haben Sie die Funktionalität von `to_s`.

Beispiel 2-55: Aus Arrays werden Strings

```
chars2.push("F").reverse.unshift("G").to_s # => "GFEDC"  
["Friede", "Freude", "Eierkuchen"].join(", ") # => "Friede, Freude, Eierkuchen"
```

Zu `join` muss noch gesagt werden, dass die Methode `split`, die Sie bereits weiter oben kennen gelernt haben, genau das Gegenteil leistet. Man merke sich: `Split` splittet und `join`, nun ja, `joint`.

Soviel zum Thema Array. Vorerst zumindest, denn wir werden im weiteren Verlauf des Buchs noch umfangreich auf Arrays zu sprechen kommen. Sollte sich in Ihrem Kopf auf den letzten Seiten ein kleines Gedanken-Array gebildet hat, dessen Elemente aus den grundlegenden Wegen zur Erzeugung eines Arrays, den Varianten der Verknüpfung mehrerer Arrays, den Möglichkeiten für den Zugriff auf einzelne Elemente und aus ein paar Methoden bestehen, wäre das schon ganz hervorragend. Möglicherweise regt sich in Ihnen aber auch Protest, weil der hinter Ihnen liegende Überblick über Arrays in Ruby die assoziativen Vertreter dieser Art völlig ausklammerte. Das hatte aber einen Grund. Nämlich diesen:

Hash mich, ich bin ein assoziatives Array

So ein *Hash* hat einige Gemeinsamkeiten mit einem Array. Die Grundidee ist dieselbe, lediglich die Art und Weise, Elemente anzusprechen, ist anders. Können Sie ein Array-Element über dessen Index ansprechen, so steht Ihnen bei einem Hash-Element so etwas wie sein Name zur Verfügung. Genau genommen ist dieser Name ein Schlüssel. Ein Hash besteht also aus lauter *Schlüssel/Wert-Paaren*.

Bevor wir uns nun detailliert mit dieser Möglichkeit, komplexe Daten zu speichern, auseinander setzen, möchte ich Ihnen verraten, dass Hashes eine große Rolle bei *Ruby on Rails* spielen werden. Sie finden praktisch bei jedem Aufruf einer Rails-Funktionalität Verwendung, meist zum Festlegen von zahlreichen Parametern in einem Rutsch. Aber dazu gibt es später noch genug zu sagen.

Jetzt soll zunächst die Frage geklärt werden, wie ein Hash gebildet wird und welche syntaktischen Besonderheiten ihn auszeichnen. Wie bei den Arrays bilden Klammern den Mantel eines Hashs – allerdings sind es diesmal keine eckigen. Geschweifte müssen es schon sein. Im Hash-Innenen können Sie die Elemente notieren, die Sie jeweils mit Komma trennen.

Ein Hash-Element besteht aus einem *Schlüssel* und dem dazugehörigen *Wert*. Jeder Schlüssel darf nur einmal verwendet werden und sollte ein Objekt der Klassen *Symbol*, *String* oder *Fixnum* sein. Aber selbst `true`, `false`, `nil`, *Ranges*, *Arrays*, *FLOATs* oder andere, für Hash-Schlüssel eher ungewöhnliche Klassen können Sie verwenden. Ja, auch Hashes – aber ob man das jemals braucht? Grundsätzlich gilt jedenfalls: Als Schlüssel geht alles, was ein Objekt ist. Und: Alle Schlüssel eines Hashes können unterschiedlichen Klassen angehören. Das Gleiche gilt für die dazugehörigen Werte.

Schlüssel und Werte müssen Sie nur noch durch einen Pfeil trennen, der aus einem Gleichheitszeichen und einem Größer-als-Zeichen besteht (`=>`). Und fertig ist ein *Hash-Literal*. Dass Ruby daraus automatisch ein Objekt bastelt, das den Bildungsvorschriften der Klasse Hash genügt, muss ich wohl nicht mehr erwähnen.

Beispiel 2-56: Hash-Literale

```
countries = {"GER"=>"Deutschland", "SUI"=>"Schweiz", "SWE"=>"Schweden"}  
modes = {:+on=>2, :standby=>1, :off=>0}  
hashes = {"countries"=>countries, "modes"=>modes}
```

Es gibt eine weitere interessante Art und Weise, einen Hash zum Leben zu erwecken. Mit der Klassenmethode `new` können Sie ein leeres Hash-Objekt erzeugen. Allerdings, so ganz leer ist es nicht. Wenn Sie nämlich einen Parameter an `new` heften, können Sie einen Standardwert definieren, der immer dann ausgegeben wird, wenn Sie auf einen Schlüssel zugreifen wollen, den der betreffende Hash gar nicht kennt.

Das folgende Beispiel soll Ihnen diese Variante in Aktion und gleichsam eine Anwendung von Hashes zeigen, die recht typisch ist: Eine größere Menge von einzelnen Variablen, die inhaltlich zusammengehören, kann in einer gesammelt werden.

Beispiel 2-57: Hash mit Default-Wert erzeugen

```
config = Hash.new('Einstellung unbekannt')  
config[:host] = 'http://www.ruby.org'  
config[:port] = 80  
config # => {:+host=>"http://www.ruby.org", :port=>80}  
config[:host] # => "http://www.ruby.org"  
config[:login] # => "Einstellung unbekannt"
```

Der Schlüssel `:login` existiert in diesem Beispiel nicht – also wird der Standardwert ausgegeben. Den können Sie übrigens auch noch nachträglich vergeben oder aber immer wieder ändern.

Ein Grundkurs in Hash-isch

Die Methoden `default` und `default=` ermöglichen Ihnen das Auslesen und Verändern des Standardwerts. Übrigens auch dann, wenn Sie den Hash über ein Literal erzeugt haben. Wenn Sie den Standardwert nicht setzen, ist dieser automatisch `nil`. Standardmäßig.

Beispiel 2-58: Hash-Standardwerte mit default und default=

```
config.default # => "Einstellung unbekannt"
config.default = '[Fehler] '.upcase + config.default
config.default # => "[FEHLER] Einstellung unbekannt"
modes.default = -1
countries.default = 'Absurdistan'
hashes["1455_aEB"] # => nil
```

Ist Ihnen eigentlich aufgefallen, dass Sie in den vorangegangenen Beispielen auch gesehen haben, wie Sie auf einzelne Elemente zugreifen können? Möglichweise nicht, denn `[]` und `[]=` sollten Ihnen ja spätestens seit dem Kennenlernen von Bruder Array recht vertraut sein.

Der Unterschied in der Verwendung liegt in der Natur der Sache: Hashes funktionieren über Schlüssel statt über Indizes. Also müssen Sie ein Hash-Element auch beim Namen nennen – beim Lesen und beim Schreiben eines Werts. Genau wie beim Array können Sie mit `[]=` auf bestehende Elemente verändernd einwirken aber auch neue Elemente erzeugen – so wie beim Befüllen der Variable `config` weiter oben gesehen.



Nicht verwechseln: Arrays erhalten durch [...] ihre Elemente, welche mit [...] abgefragt werden können. Hashes werden durch {...} charakterisiert, und dennoch erfolgt der elementweise Zugriff via [...].

Ist so ein Hash erst einmal befüllt, gibt er auf Wunsch einige Informationen preis: Um zu erfahren, wie viele Elemente ein Hash enthält, kommen wieder die Methoden `length` oder `size` zum Einsatz. Sollte dabei als Ergebnis die Zahl `0` herauskommen, hätten Sie das auch hier über `empty?` herausfinden können. Die Methode gibt in diesem Fall `true` zurück. Auch dann, wenn Sie vorher die Methode `clear` aufgerufen habe. Sie befreit einen Hash von all seinen Elementen.

Beispiel 2-59: Leer oder nicht leer?

```
hashes.length # => 2
hashes.empty? # => false
hashes.clear # => {}
hashes.empty? # => true
```

Das Vorhandensein eines Schlüssels können Sie mit `has_key?`, das eines bestimmten Werts mit `has_value?` erfragen. Geben Sie jeweils den Schlüssel beziehungsweise

den gesuchten Wert als Parameter an. Sie erhalten true oder false als Antwort. Möchten Sie nun noch wissen, über welchen Schlüssel Sie einen bestimmten Wert erreichen können, so gibt Ihnen die Methode index Auskunft. Allerdings nur, wenn Sie den Wert exakt als Parameter angeben.



Werte können im Gegensatz zu Schlüsseln, die zwingend eindeutig sein müssen, mehrfach innerhalb eines Arrays auftreten. Die Methode index liefert Ihnen in diesem Fall immer das erste Vorkommen innerhalb eines Hashs zurück.

Auf jeden Fall sollten Sie wieder beachten, dass Ruby wie so oft einen Unterschied zwischen Groß- und Kleinschreibung macht.

Beispiel 2-60: Existieren Schlüssel und Werte?

```
config.has_key?("port") # => false
config.has_key?(:port) # => true
config.has_value?('http://www.php.net') # => false
config.has_value?('http://www.ruby.org') # => true
config.has_value?('http://www.RUBY.org') # => false
```

Wie Sie einen Hash um ein einzelnes Element erweitern können, haben Sie bereits gesehen. Möchte man aber mehrere Elemente auf einmal anfügen, kommt man damit nicht weit. Dafür bietet sich aber die Methode merge an. Es gibt sie auch als merge!, wobei ein weiteres Mal das Ausrufungszeichen den Unterschied macht und Änderungen direkt an dem Hash-Objekt vornimmt, durch das merge! aufgerufen wurde. Das Hinzuzufügende wird als Parameter übergeben.

Beispiel 2-61: Hashes zusammenführen

```
config2 = { :user=>'kampfkroete', :pass=>'7EB79C2'}
config3 = config.merge(config2) # => { :user=>"kampfkroete", :host=>"http://www.ruby.org", :port=>80, :pass=>"7EB79C2"}
config # => { :user=>"kampfkroete", :host=>"http://www.ruby.org", :port=>80, :pass=>"7EB79C2"}
config.merge!(config2) # => { :user=>"kampfkroete", :host=>"http://www.ruby.org", :port=>80, :pass=>"7EB79C2"}
```

Zwei wichtige Dinge gibt es zum Verkuppeln von Hashes noch zu sagen: 1. Die Elemente erscheinen nach dem Mergen mitunter bunt durcheinandergewirbelt. Die Reihenfolge der Hash-Bestandteile hält sich nicht an die Reihenfolge der Zusammenführung. 2. Sollten beide Hashes gleiche Schlüssel haben, so werden diese und die dazugehörigen Werte überschrieben. Schlüssel müssen eindeutig sein.

Um einzelne Elemente eines Hashs zu entfernen, haben Sie mehrere Möglichkeiten: Mit delete können Sie gezielt ein Element über dessen Schlüssel ansprechen und eliminieren. Mit shift löschen Sie stets das erste Element eines Hashs. Während delete dabei den Wert des gelöschten Schlüssel/Wert-Paars oder auch den Stan-

dardwert bei einem nicht existierenden Schlüssel zurückgibt, rückt shift gleich mit dem ganzen Paar heraus oder ebenfalls mit dem Standardwert, sollte der Hash bereits leer sein.

Beispiel 2-62: Hash-Elemente löschen

```
config.delete(:user) # => "kampfkroete"
config.delete(:pass) # => "7EB79C2"
hashes.shift # => nil
```

Nun drängen sich wieder die Konvertierungsmethoden in den Vordergrund. Namentlich to_a, mit der Sie einen Hash in ein Array verwandeln können, und to_s, die das Gleiche mit einem String als Ergebnis vermag. Allerdings sind die beiden etwas eigenwillig: Denn to_a macht aus jedem Schlüssel/Wert-Paar ein zweielementiges Array, in dem der ehemalige Schlüssel an erster, der ehemalige Wert an zweiter Stelle steht. Am Ende werden alle Paar-Arrays vereint in einem gemeinsamen Array ausgegeben. Mit to_s erhalten Sie einen String, in dem abwechselnd Schlüssel und Wert ohne Trennung aneinander gesetzt sind. Das ist nicht sonderlich attraktiv.

Vielleicht gefallen Ihnen ja die Methoden keys und values besser. Hier erhalten Sie jeweils ein Array zurück, das nur alle Schlüssel beziehungsweise alle Werte eines Hashs enthält.

Beispiel 2-63: Hashes konvertieren

```
countries.to_a # => [["SWE", "Schweden"], ["SUI", "Schweiz"], ["GER", "Deutschland"]]
countries.to_s # => "SWE Schweden SUI Schweiz GER Deutschland"
countries.keys # => ["SWE", "SUI", "GER"]
countries.values # => ["Schweden", "Schweiz", "Deutschland"]
```

Damit hätten Sie auch alles Wichtige zu Hashes kennen gelernt, womit wir dieses Thema abschließen können. Und nicht nur das: Mit diesen Zeilen schließen wir auch das Beschnuppern der elementarsten Ruby-Klassen und ihrer eingebauten Methoden ab. Sie haben schon jetzt eine große Menge an Ruby-Funktionalität intus: Ganze und gebrochene Zahlen, Strings, Variablen, Symbole, Bereiche, sowie komplexe Datenstrukturen, die aus Objekten der eben erwähnten Klassen bestehen. Damit lässt sich schon viel anstellen. Und doch ist das nur der Schotter unter den Schienen, über die Ihre Rails-Anwendungen später flitzen werden.

Nun beschäftigen wir uns mit Techniken, die zum Basisrepertoire einer jeden Programmiersprache gehören und allgemein als *Kontrollstrukturen* rubriziert werden. Aber Sie haben sicher nichts dagegen, wenn ich nachfolgend lieber von *Weichen* spreche. Dieser Begriff beschreibt sehr exakt, um was es gleich gehen wird. Und er passt doch auch viel besser zu diesem Buch.

Weichen stellen

Man stelle sich nur mal vor, die grandiose Erfindung des schienengebundenen Verkehrs müsste ohne Weichen auskommen. Das ganze Land bestünde quasi nur aus Schienen. Der Bahnhof einer Großstadt würde über bestimmt 100 Gleise verfügen, denn für fast jede Stadt, die ein Zug ansteuern soll, müsste ein separates Gleis existieren – Abzweigungen von gemeinsam genutzten Strecken gäbe es ja nicht. Kurzum: Ohne Weichen wäre die Eisenbahn wohl kaum zu dem wichtigen Verkehrsmittel geworden, das sie heute ist. Allerdings müssen die Weichen stets richtig gestellt sein, damit etwa der exquisite Hochgeschwindigkeitszug nicht plötzlich und unerwartet im längst stillgelegten Bahnhof von Hinterdensiebenbergen parken muss.

Und genau so ist es auch beim Programmieren von Software. Gäbe es keine Weichen, dann müssten unzählige eigenständige Programmvarianten gecodet werden, die auf ein eventuell eintretendes Ereignis vorbereitet sein müssten – oder anders ausgedrückt: Es gäbe dann wahrscheinlich keine Software, oder ausschließlich unbrauchbare. Doch so wie ein Zug sein Ziel durch richtige Weichenstellungen erreicht, können auch Abläufe innerhalb eines Programms abhängig von Bedingungen gezielt gesteuert werden. Und Sie erfahren jetzt, wie das in Ruby geht.

Einfache Weichen

Weichen werden in der Programmierung über Bedingungen gesteuert. Ein Sachverhalt wird auf die Erfüllung einer Bedingung hin überprüft. Dabei lautet die Frage: Ist eine Bedingung wahr oder falsch? Die Beantwortung dieser Frage legt fest, ob und wenn ja wo und womit es im Programmablauf weitergeht. Viele Programmiersprachen benutzen dafür eine if-Konstruktion. So auch Ruby. Damit können Sie ein Wenn-dann ausdrücken. Es gilt folgendes Schema:

```
if Bedingung
  Aktionen, die ausgeführt werden, wenn die Bedingung erfüllt wird.
end
```

Möchten Sie, dass die Nichterfüllung der Bedingung ebenfalls behandelt werden soll, können Sie dies mit else tun.

```
if Bedingung
  Aktionen, die ausgeführt werden, wenn die Bedingung erfüllt wird.
else
  Aktionen, die ausgeführt werden, wenn die Bedingung nicht erfüllt wird.
end
```

Sie können auch mehr als zwei Fälle behandeln. Mit elsif können Sie eine weitere Bedingung überprüfen, sollte die vorangegangene Ihrer if-Konstruktion nicht erfüllt worden sein. Sie können mehr als einen elsif-Block implementieren.

```

if Bedingung
  Aktionen, die ausgeführt werden, wenn die Bedingung erfüllt wird.
elsif Bedingung2
  Aktionen, die ausgeführt werden, wenn die Bedingung nicht, aber Bedingung2
  erfüllt wird.
elsif Bedingung3
  Aktionen, die ausgeführt werden, wenn die Bedingung2 nicht, aber Bedingung3
  erfüllt wird.
else
  Aktionen, die ausgeführt werden, wenn keine der Bedingungen erfüllt wird.
end

```

Denken Sie immer daran, dass else- oder elsif-Blöcke nicht zwingend nötig sind. Außerdem darf nach einem else-Block kein elsif mehr kommen. Danach ist Schluss mit dem Vergleichen. Das erklärt sich aber auch schon aus dem logischen Zusammenhang, der sich Ihnen schnell erschließt, wenn Sie if-Konstruktionen wortwörtlich lesen.

Auch für den eher pessimistisch eingestellten Menschen hält Ruby etwas bereit. Nämlich eine Konstruktion, die dem Wenn-nicht-dann-Prinzip folgt und mit unless gebildet wird. Hier können Sie auch else und elsif nutzen. Ein elsunless, oder wie immer es auch sonst heißen könnte, existiert nicht.

```

unless Bedingung
  Aktionen, die ausgeführt werden, wenn die Bedingung nicht erfüllt wird.
else
  Aktionen, die ausgeführt werden, wenn die Bedingung erfüllt wird.
end

```

Beachten Sie, dass das großzügige Verwenden neuer Zeilen keinen ästhetischen Hintergrund hat, sondern den Konventionen von Ruby geschuldet ist. Also denken Sie daran: if, elsif und unless und die zu prüfende Bedingung haben das gleiche Recht auf eine neue Zeile wie else und end.

Bedingungen formulieren

Ein paar einfache Grundstrukturen zum Programmieren von Weichen kennen Sie nun schon. Lediglich die Formulierung von Bedingungen steht der ungebremsten Freude an Rubys Kontrollstrukturen noch im Weg.

Ganz einfach formuliert kommt es dabei nur darauf an, ob eine Bedingung, die sie auch als Frage sehen können, true oder false ist. So eine Antwort kann aber von ganz unterschiedlichen Fragen kommen. Nachfolgend erhalten Sie einen Überblick über Fragen, die Sie mit Hilfe von Methoden Ihnen bereits bekannter Klassen stellen können.

Beispiel 2-64: Rückgabewerte von Methoden als Bedingungen

```
(12 * 4 - 48).zero? # => true
'Ruby'.include?('t'.succ) # => true
['A', 'C', nil].empty? # => false
{}.empty? # => true
{1=>"first", 2=>"second"}.has_key?(2) # => true
```

Natürlich stehen einem nicht immer Bedingungen, die true oder false zurückgeben, zur Seite, wenn der weitere Programmablauf beschieden werden soll. Viel häufiger wird die Sache eher mathematisch gelöst, nämlich mit Vergleichen wie *gleich*, *größer als* und *kleiner als*. Das funktioniert besonders gut bei Zahlen. Beachten Sie bitte den Unterschied zwischen `a = 12` und `a == 12`. Mit dem doppelten Gleichheitszeichen wird verglichen, mit dem einfachen zugewiesen.

Beispiel 2-65: Vergleiche mit Zahlen

```
a = 12; b = 25.28; c = 14
a == 13 # => false
a == 12 # => true
b == a # => false
(a * b) > c # => true
a == (b.ceil - c) # => true
```

Sie können auch ganz einfach gleich zwei Fälle in Ihren Bedingungen berücksichtigen. Dazu stehen Ihnen die Vergleiche *kleiner gleich* (`<=`) und *größer gleich* (`>=`) zur Verfügung. Das Gleichheitszeichen steht stets rechts.

Mit Buchstaben, also String-Objekten, kann man ähnlich verfahren – auch wenn man sich das etwas schwerer vorstellen kann. Doch Sie wissen bereits, dass in der String-Klasse die Methode `succ` formuliert ist, mit der Sie den Nachfolger eines Strings bestimmen können. Wenn Sie das im Hinterkopf behalten, fällt Ihnen das Vorstellen von String-Vergleichen leichter. Grundsätzlich gilt auch, dass alle Großbuchstaben im computerinternen Alphabet vor den Kleinbuchstaben kommen. String-Vergleiche, die Zahlen enthalten, werden nicht nach ihrem nummerischen Wert verglichen, sondern wie Buchstaben behandelt und Zeichen für Zeichen verglichen.

Beispiel 2-66: Vergleiche mit Strings

```
d = 'einem'; e = d.succ
d == "einem" # => true
d == e # => false
d < e # => true
d > e # => false
d[0,4] == e[0,4] # => true
d = 'einem'; e = d.succ # => 'einen'
"10" < "2" # => true, aber...
10 < 2 # => false
```

Interessant und sinnvoll werden solche Vergleiche von Strings, wenn es darum geht, eine ganze Reihe von Zeichenketten zu sortieren. Mit diesem Problem werden wir uns später noch beschäftigen.

Auch Ranges, Arrays und Hashes können Sie vergleichen, wenn Sie auch nur Gleichheit oder Ungleichheit feststellen können. Ranges sind dann gleich, wenn sie den exakt gleichen Wertebereich abdecken und ihre Literale die gleiche Anzahl an Punkten aufweisen.



Sollte in der Formulierung einer Bedingung ein Range-Literal auftreten, so setzen Sie es unbedingt in Klammern, damit Ruby erkennt, dass die Zahlen und die Punkte zusammengehören und nicht getrennt betrachtet werden dürfen.

Zwei Arrays sind dann gleich, wenn Sie die gleiche Anzahl an Elementen enthalten und wenn jedes der Elemente des ersten Arrays mit dem entsprechenden Element (gleicher Index!) des zweiten Arrays übereinstimmt – so als ob beispielsweise jeweils zwei eigenständige String- oder Fixnum-Objekte miteinander verglichen werden. Bei einem Hash muss diese Gleichheit natürlich noch auf die Schlüssel ausgeweitet sein.

Beispiel 2-67: Vergleiche mit Ranges, Arrays und Hashes

```
(1..12) == (1..12) # => true
(1..12) == (1...13) # => false
["Ruby", "Python", "PHP"] == ["Ruby", "Python", "PHP"] # => true
["Ruby", "Python", "PHP"] == ["ruby", "Python", "PHP"] # => true
["Ruby", "Python", "PHP"] == ["Python", "PHP", "Ruby"] # => false
{:w=>'West'} == {:w=>'West'} # => true
{:w=>'West'} == {:W=>'West'} # => false
```

Sie können mehrere Bedingungen natürlich auch miteinander verknüpfen und zusammenfassen. Dafür stehen Ihnen die *booleschen Operatoren* and und or zur Verfügung. Bei der Verknüpfung mit and müssen alle Einzelbedingungen true ergeben, damit der gesamte Ausdruck von Ruby mit true ausgewertet wird. Bei or reicht es, wenn eine Bedingung true ist. Alternativ können Sie für die Und-Verknüpfung auch &&, sowie für die Oder-Verknüpfung || schreiben.



Sie sollten die Einzelbedingungen in separaten Klammern unterbringen. Das erhöht die Lesbarkeit und Verständlichkeit des Codes enorm, da die logischen Zusammenhänge deutlicher hervortreten.

Sie können Teile der Bedingung auch *negieren*. Setzen Sie ein Ausrufezeichen vor einer Variable, die den Wert true enthält, so wird sie als false ausgewertet. Das funktioniert auch mit einem ganzen Ausdruck, wobei Sie diesen dann mit Klammern umgeben sollten. Mit dem Ausrufezeichen können Sie eine Untersuchung auf

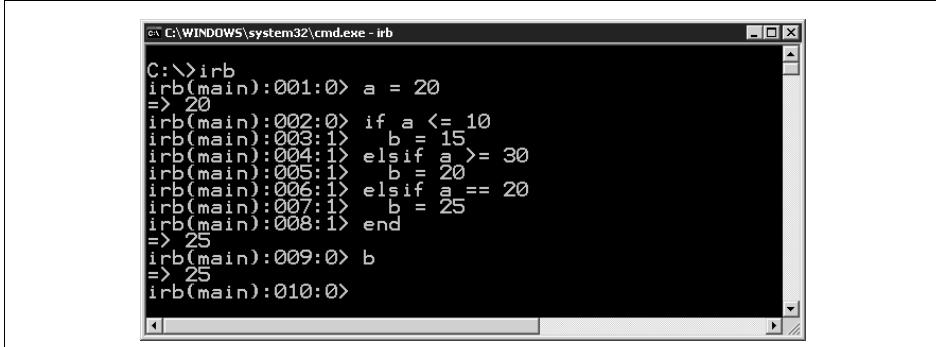
Gleichheit ganz einfach zu einer auf Ungleichheit umgestalten. Aber aufgepasst: Bei einer doppelten oder dreifachen Negation, möglicherweise noch in Kombination mit unless, kann der Programmiererkopf leicht überhitzen.

Beispiel 2-68: Boolesche Verknüpfung und Negation

```
g = true; h = !g; i = g, j = !i
g != h # => true
(g == h) || i # => true
!(g == h) or i # => true
(g == h) || !i # => false
g && !h && !j # => true
(g && !h && !j) and i # => true
(g != h) and !(i == j) # => true
```

Werfen wir nun einen Blick auf die beispielhafte Verwendung einer echten if-elsif-else-Konstruktion mit einer echten Bedingung. Wenn Sie das Lesen dieses Buchs vorbildlich durch reges Ausprobieren mit *Interactive Ruby* begleiten, möchte ich Ihnen folgenden Hinweis zum Eingeben der folgenden Zeilen geben: Eine Weiche besteht aus mehreren Zeilen, die Sie alle nacheinander eingeben und jeweils mit der *Enter*-Taste abschließen sollten. Erst nach dem letzten *end* wertet irb Ihre Eingaben aus und Sie sehen das Ergebnis.

Im folgenden Fall wird die if-Konstruktion als 25 ausgewertet. Das liegt daran, dass in dem Block, der die Bedingung erfüllt, ebendiese Zahl einer Variablen zugeordnet wird.

A screenshot of a Windows command prompt window titled "C:\>irb". The window shows an Interactive Ruby session. The user has entered the following code:

```
C:\>irb
irb(main):001:0> a = 20
=> 20
irb(main):002:0> if a <= 10
irb(main):003:1>   b = 15
irb(main):004:1> elsif a >= 30
irb(main):005:1>   b = 20
irb(main):006:1> elsif a == 20
irb(main):007:1>   b = 25
irb(main):008:1> end
=> 25
irb(main):009:0> b
=> 25
irb(main):010:0>
```

The output shows that the variable 'b' is assigned the value 25, indicating that the condition 'a == 20' was matched.

Abbildung 2-12: Mehrere Bedingungen werden überprüft

Ein praxisnäheres Beispiel zur Verwendung von Weichen kommt gleich. Bis dahin möchte ich Ihnen noch weitere Möglichkeiten der Weichenstellung präsentieren.

Modifikatoren

Das hat auch nicht jede Sprache: *Modifikatoren*. Das klingt vielleicht ziemlich kompliziert, ist es aber gar nicht. Modifikatoren sind einfach eine spezielle Art von Weichen. Sie kommen Elementen unserer täglichen Sprache recht nahe.

Vielleicht sind Sie in Besitz von Nachwuchs. Dann haben Sie diesen doch sicher auch schon mit folgenden Sätzen versorgt: »Du kannst heute länger aufbleiben. Aber nur, wenn du dein Zimmer aufräumst.« Oder »Klar kannst du dir ein Piercing durch die Schädeldecke jagen. Es sei denn, es kostet mehr als zehn Euro.«

Die Bedingung folgt also der Aktion, die damit verknüpft ist und somit nur bedingt ausgeführt wird. Das geht mit `if` und `unless`. Alternativverzweigungen können Sie hier allerdings nicht verwenden.

Beispiel 2-69: Modifikatoren entscheiden nachgelagert

```
puts "1 ist kleiner als 2" if 1 < 2  
puts "5 ist nicht kleiner als 3" unless 5 < 3  
puts "16 ist durch 5 teilbar" if 16 % 5 == 0
```

In den ersten beiden Fällen wird die Zeichenkette ausgegeben, da die nachgestellte Bedingung jeweils erfüllt wird. Im dritten Fall passiert hingegen nichts.

Modifikatoren sind ein elegantes Mittel, einfache Weichen schnell zu implementieren. Wenn ein Programmablauf je nach Bedingung in diverse Gleise verzweigen muss, kommt man damit nicht weit. Da sind `if`-Konstruktionen mit beliebig vielen `elsif`-Blöcken schon besser. Doch solchen Gegebenheiten können Sie in Ruby noch mit anderen Mitteln begegnen, nämlich mit `case` und `when`.

Mehrfachverzweigungen mit `case`

Nehmen wir an, Sie programmieren ein Menü, mit dem der Benutzer Ihr Programm steuern kann. Der Benutzer teilt dem Programm seine Entscheidung mit, indem er eine der Zahlen eingibt, die vor jedem Menüpunkt stehen. Jedem Menüpunkt ist eine andere Methode zugeordnet, die nur dann ausgeführt wird, wenn der Benutzer die entsprechende Zahl des Menüpunkts eingegeben hat. Und genau an dieser Stelle lohnt sich die Verwendung einer `case-when`-Verzweigung. Blicken wir zunächst auf das Grundschema:

```
case Wert  
  when Wert1: [Aktion, die ausgeführt wird, wenn Wert gleich Wert1]  
  when Wert2: [Aktion, die ausgeführt wird, wenn Wert gleich Wert2]  
  ...  
  when Wertx: [Aktion, die ausgeführt wird, wenn Wert gleich Wertx]  
  else [Aktion, die ausgeführt wird, wenn kein when-Zweig in Frage kam]  
end
```

Mit einem `else`-Zweig können Sie festlegen, was passieren soll, wenn keiner der zuvor durchlaufenen `when`-Zweige passte. So können Sie auch das Standardverhalten einer `case-when`-Verzweigung festlegen. Allerdings sind `else`-Zweige nur optional. Wenn Sie aber genutzt werden, dann nur einmal pro `case`.

Ein `when`-Zweig kommt mit Zahlen, Strings und Ranges klar. So können Sie beispielsweise auf Werte reagieren, die in einem bestimmten Bereich liegen. Sie kön-

nen in den when-Zweigen auch Ranges und Strings oder Zahlen kombinieren. Es ist auch möglich, mehrere when-Zweige in einem einzigen miteinander zu verknüpfen, wenn sie die gleichen Aktionen beinhalten. Dazu trennen Sie einfach die möglichen Werte mit Kommas.

Beispiel 2-70: Wert trennen

```
case 14
when 0: puts 'Zahl ist 0.'
when 1..10: puts 'Zahl liegt zwischen 1 und 10.'
when 11..20: puts 'Zahl liegt zwischen 11 und 20.'
when 21..26, 28...30, 30: puts 'Zahl liegt zwischen 21 und 26 oder 28 und 30'
when '27'.to_i: puts 'Zahl ist 27.'
else puts 'Zahl ist kleiner als 0 oder größer als 30.'
end
```

Beachten Sie, dass nachfolgende when-Zweige nicht mehr durchlaufen werden, sobald ein passender gefunden wurde. Das ist wichtig, sollten mehrere when-Zweige ein und denselben Fall abdecken. Dabei gilt, dass when-Zweige von oben nach unten in der Reihenfolge der Implementierung durchlaufen werden. Im folgenden Beispiel wird etwa der zweite when-Zweig gar nicht beachtet, obwohl er zutreffend wäre.

Beispiel 2-71: Mehrere zutreffende Zweige

```
a = "C"
case a
when 'C': b = 10
when 'A'..'Z': b = 20
end
b # => 10
```

Mit diesem Beispiel schließen wir das Thema Weichen ab. Nachdem Sie nun if, unless, else, elsif, case und when sowie Modifikatoren kennen gelernt haben, wird es Zeit, das Gelesene der letzten Seiten in einem neuen Programm zur praktischen Anwendung zu bringen. Das hilft Ihnen nicht nur bei Ihrer Entwicklung als Ruby-Programmierer. Richtig eingesetzt kann es sogar dafür sorgen, dass Sie zukünftig so richtig viel Zeit haben, mit Ruby und Ruby on Rails zu arbeiten.

Ruby-Programm 2: Der Ehe-Checker

Rubert ist selbstständiger Programmierer. Ruby-Programmierer natürlich. Jahrelang widmete er seine ganze Leidenschaft einzig dem Entwickeln ausgeklügelter Anwendungen. Das konnte er auch, anderweitige Verwendung fanden seine Emotionen ohnehin nicht. Es gab zwar ab und an einen kleinen Flirt am Telefon, allerdings waren die anrufenden Damen mehr an der Beantwortung ihrer Supportanfragen interessiert als an der Beschreibung ihrer Haar- und Augenfarbe.

Seit vier Tagen ist das anders. Seine Gedanken kreisen seither nur noch um *sie*. (Trotzdem kann er weiterhin mit Ruby schnell und effizient Software entwickeln.

Der Quelltext schreibt sich ja quasi von selbst, ist einfach, besteht aus wenigen Regeln, ist logisch und objektorientiert.) Ein Ruby-Programm, das die Menge der in seinem Blut schwimmenden Glückshormone errechnen sollte, müsste dies mit Big-
num-Objekten tun. Sein interner Speicher hat nur noch Platz für ein einziges String-
Objekt, das ihren Namen beinhaltet: *Rubina*.

Nein, hübsch ist sie nicht. Reich auch nicht. Aber sie hat zwei große, oh pardon, ein großes Herz. Und das ist Rubert wichtig. Entgegen seiner Hauptbeschäftigung verfährt er im wahren Leben nämlich eher subjektorientiert. Die inneren Werte zählen. Weiteren Luxus kann er sich bei der Partnerwahl eh nicht leisten.

Rubert und Rubina haben sich auf einer der vielen pilzgleich emporschießenden Verkuppelseiten im Internet kennen gelernt. Die basierte natürlich auf *Ruby on Rails* und entstand innerhalb von nur zwei Tagen. Schon allein das machte Rubert tierisch an. Rubina war dort, weil sie bei allen anderen Seiten dieser Art erfolglos ihr Glück suchte. Man traf sich und verstand sich: ==.

Die nächsten vier Tage verbrachten beide auf `wolke = 6.succ`. Lediglich der fleischlichen Konkatenation gaben sie sich noch nicht hin. Gestern fragte Rubina Rubert, ob er sie ehelichen wolle. Klar, schließlich gilt auch in der Liebe und gerade nach vier Tagen: Was man hat, das hat man. Sie sagte außerdem unter Zuhilfenahme eines Ruby-besetzten Schmuckstücks, dieser :ring solle das Symbol ihrer Liebe sein und #*{Ausdruck}* ihrer engen Verbundenheit. Allerdings war Rubert skeptisch, ob eine viertägige Beziehung Basis für eine Ehe sein kann. Kennen Sie sich denn überhaupt? Und wie kann man das denn auf die Schnelle rausfinden? Er nahm sich vor, ein Ruby-Programm zu schreiben, das genau diese und Rubinas Frage beantworten soll.

Und so soll es funktionieren: Rubert hackt all seine Hobbys in das Programm. Dann soll Rubina ihr Wissen über Ruberts Hobbys beweisen, indem Sie diese möglichst komplett eingibt. Anschließend analysiert das Programm die Übereinstimmungen und teilt Rubina mit, wie die Antwort auf ihr eheanbahnendes Ersuchen lautet.

Wir blicken Rubert einfach mal über die Schulter. Und ich schlage vor, Sie entwickeln mit. Möglicherweise können Sie ein solches Programm ja auch ganz gut im privaten Bereich gebrauchen. Außerdem können Sie besonders Arrays und Kontrollstrukturen im Live-Einsatz erleben. Legen Sie in *RadRails* ein neues Ruby-Projekt an. Nennen Sie es diesmal *Prog2* und wählen Sie ein Verzeichnis, das es aufnimmt. Zum Beispiel *C:\Ruby_apps\Prog2*. Erzeugen Sie dann eine neue Datei, die mit *main.rb* einen guten Namen erhalten würde. Und schon kann es losgehen.

Daten einlesen

Der Anfang ist einfach. Mittels eines Literals erstellt Rubert ein String-Objekt, das seine Hobbys beinhaltet, welche durch Kommas voneinander getrennt werden. Anschließend spuckt das Programm mittels `print` ein paar Hinweise für Rubina aus. Dort steht beispielsweise, dass Rubina Ruberts vermeintliche Hobbys ebenfalls

durch Kommas trennen soll. Rubinas Eingaben werden durch gets und somit als String eingelesen. Beachten Sie hierbei, dass das abschließende Berühren der *Enter*-Taste als Zeichenumbroch im String steckt und mit der String-Methode `chomp` entfernt werden muss.

Im weiteren Verlauf gilt es, die beiden Strings miteinander zu vergleichen und Übereinstimmungen zu finden. Das geht am besten, wenn jedes einzelne Hobby als eigenständig betrachtet wird. Wenn Sie sich daran erinnern, was Sie auf den vergangenen Seiten gelesen haben, werden Sie mir sicher zustimmen, dass das rein organisatorisch betrachtet am besten mit Arrays geht. Mit dem Umwandeln der Strings in Arrays machen Sie sich zudem die zahlreichen Möglichkeiten von Ruby zu Nutze, die das Verknüpfen zweier Arrays betreffen. Ich möchte nur an *Schnittmenge* und *Differenz* erinnern.

Da diese Funktionalitäten *case-sensitive* sind, werden die Buchstaben der Ausgangs-Strings komplett verkleinert, bevor sie in ein Array verwandelt werden. Das junge Glück soll ja schließlich nicht an Ruberts oder Rubinas Schwierigkeiten mit der Groß-/Kleinschreibung scheitern.

Die Umwandlung von String in Array erfolgt schließlich durch die String-Methode `split`, die das Trennzeichen Komma als Parameter erhält. Werfen wir einen Blick auf den ersten Teil des Programms.

Beispiel 2-72: Aus Hobbys werden Array-Elemente

```
rubert = 'Angeln,Ruby,Musik,Lesen,Lachen'.downcase.split(',')
print "Liebste Rubina! Ich habe #{rubert.size} Hobbys hinterlegt. Welche sind es? \
Gib Deine Vermutungen durch Kommas getrennt jetzt ein: "
rubina = gets.chomp.downcase.split(',').uniq
```

Bei Rubinas Array wird übrigens abschließend die Methode `uniq` ausgeführt. Damit werden doppelt eingegebene Hobbys, die in der Aufregung schon einmal vorkommen können, grundsätzlich ausgeschlossen.

Die Informationsfütterung ist damit abgeschlossen. Ruberts und Rubinas Angaben liegen nun in leicht verdaulichen Arrays vor. Alles, was jetzt kommt, umfasst die Analyse der Eingaben und die Auswertung und Ausgabe der Ergebnisse. Also, jetzt wird's ernst, Rubina.

Analyse der Eingabe

Die Auswertung soll auf zwei Zahlen basieren: Der Anzahl der Übereinstimmungen und der daraus resultierenden Prozentzahl der richtig eingegebenen Hobbys. Beide Zahlen passen inhaltlich gut zusammen, also organisieren wir sie in einem *Hash* mit den Schlüsseln `:match` und `:percent`. Der Hash wird über `Hash.new` erzeugt und der Variablen `rubert_rubina` zugeordnet.

Der Wert für `:match` ergibt sich durch das Ermitteln der Schnittmenge beider Arrays. Dazu verknüpfen wir sie mit `&`. Heraus kommt ein Array, welches alle Elemente enthält, die in beiden Arrays vorkommen. Uns interessiert die Menge der dort enthaltenen Elemente. Diese Zahl halten wir in `rubina_rubert[:match]` fest.

Die Prozentzahl, welche in `rubert_rubina[:percent]` Platz nimmt, wird anhand der Anzahl der richtig genannten und der Anzahl der von Rubert vorgegebenen Hobbys berechnet.

Beispiel 2-73: Die Angaben werden analysiert

```
rubert_rubina = Hash.new;
rubert_rubina[:match] = (rubert & rubina).size
rubert_rubina[:percent] = 100 * rubert_rubina[:match] / rubert.size
```

Die Werte sind ermittelt, jetzt gibt es kein zurück mehr. Nun sollen die Zahlen, zunächst unausgewertet, auf den Bildschirm ausgegeben werden. Und zwar in einem Satz.

```
print "Du hast #{rubert_rubina[:match]} Hobbys gewusst. Das sind #{rubert_rubina[:percent]} Prozent."
```

Nichts Aufregendes. Ein String enthält zwei Ausdrücke, die dank der doppelten Anführungszeichen von Ruby netterweise ausgewertet werden. Interessant wird es aber, wenn die gute Rubina nur ein Hobby weiß – was eine durchaus realistische Zahl ist. In diesem Fall hieße die Formulierung »*Du hast 1 Hobbys gewusst*«. Das ist noch optimierungsbedürftig. Es müsste eine Art Schalter eingebaut werden, der das `s` von *Hobbys* nur dann setzt, wenn Rubina mehr oder weniger als einen Treffer erzielt. Kein Problem mit einem Modifikator, den wir in einen Ausdruck innerhalb des Strings setzen.

```
print "Du hast #{rubert_rubina[:match]} Hobby#{'s' unless rubert_rubina[:match] == 1} gewusst. Das sind #{rubert_rubina[:percent]} Prozent."
```

Das ist doch recht elegant und einfach gelöst, oder?! Abschließend soll Rubina noch die Antwort auf ihre ursprüngliche Frage erhalten, die das Erstellen dieses kleinen Progrämmchens erforderlich machte.

Die Beurteilung der Ergebnisse

Dabei soll eine `case-when`-Verzweigung helfen. Je nach erreichter Prozentzahl soll entweder ein eher verneinernder, ein skeptischer oder ein zustimmender, leicht euphorischer Text ausgegeben werden. Das heißt, es wird drei `when`-Zweige oder zwei `when`-Zweige und ein `else` innerhalb des `case`-Blocks geben. Jeder Zweig kümmert sich um einen anderen Bereich möglicher Prozentzahlen.

Beispiel 2-74: Eine Beurteilung entscheidet

```
case rubert_rubina[:percent]
when 0..40: print "Ich programmiere dann mal eine neue Singleseite.
Mit Ruby on Rails."
when 41..80: print "Vielleicht sollten wir doch noch warten."
else print "Ja, ich will."
end
```

Damit ist das Programm fertig und darf Schicksal spielen. Blicken wir abschließend auf den gesamten Programmcode. Möchten Sie das Programm ausführen, so vergessen Sie nicht, die Änderungen an *main.rb* zu speichern. Erwecken Sie dann Ihre Kommandozeile wieder zum Leben, wechseln Sie in das Projektverzeichnis und geben Sie ruby *main.rb* ein.

```
# Ausgangsdaten und Eingabe
rubert = 'Angeln,Ruby,Musik,Lesen,Lachen'.downcase.split(',')
print "Liebste Rubina! Ich habe #{rubert.size} Hobbys hinterlegt. Welche sind es?\nGib Deine Vermutungen durch Kommas getrennt jetzt ein:"
rubina = gets.chomp.downcase.split(',').uniq

# Analyse
rubert_rubina = Hash.new;
rubert_rubina[:match] = (rubert & rubina).size
rubert_rubina[:percent] = 100 * rubert_rubina[:match] / rubert.size

#Ausgabe und Beurteilung
print "Du hast #{rubert_rubina[:match]} Hobby#{'s' unless rubert_rubina[:match] == 1} gewusst. Das sind #{rubert_rubina[:percent]} Prozent."
case rubert_rubina[:percent]
when 0..40: print "Ich programmiere eine neue Singleseite. Mit Ruby on Rails."
when 41..80: print "Vielleicht sollten wir doch noch warten."
else print "Ja, ich will."
end
```

Fertig ist ein Programm, das Arrays, einen Hash, einen Modifikator und ein case-when-Konstrukt benutzt. Was nun mit Rubert und Rubina geschehen ist? Nun, Rubina hat beim nächsten Vollmond Ihren Computer im örtlichen Dorfteich versenkt und besucht seitdem einen Töpfkurs. Und Rubert arbeitet an einer Ruby-Klasse namens Dreamgirl. Aber er kommt, trotz Ruby, eher schleppend voran. Besonders die Methode *create* bereitet ihm Schwierigkeiten.

Bevor Sie sich an ein solches Projekt wagen, sollten Sie vielleicht noch weitere Bestandteile der Ruby-Syntax kennen lernen.

Und noch 'ne Runde

Sie gehören genauso zum dringend benötigten Rüstzeug eines Programmierers wie if und else: *Schleifen*. Mit ihnen können Sie ein Quelltextstück mehrmals durchlaufen. Wie oft das geschehen soll, können Sie durch Angabe einer bestimmten

Anzahl festlegen. Man könnte fast sagen, dass Sie eine solche Schleife schon kennen gelernt haben, nämlich bei Ihrem Rendezvous mit der String-Klasse. Da konnten Sie einen String durch das Malzeichen und die Angabe einer die Häufigkeit darstellenden Ganzzahl mehrfach ausgeben. Sie erinnern sich?

```
'Nein! ' * 5 # => "Nein! Nein! Nein! Nein! Nein!"
```

So ähnlich funktioniert das auch mit Schleifen, nur geht ihre Macht weit über die einer mehrfachen Darstellung eines Strings hinaus. Neben der konkreten Angabe einer Häufigkeit gibt es zudem noch einen anderen Ansatz. Sie können auch angeben, dass Code so lang wiederholt wird, bis eine bestimmte Bedingung erfüllt ist oder nicht mehr erfüllt wird. Auf den folgenden Seiten werden Sie die diversen Möglichkeiten kennen lernen, die Ruby allen Schleifennutzungswilligen zur Verfügung stellt.

Schleifen durch Methoden

Eine recht einfache, Ruby-typische Art und Weise, Schleifen zu realisieren, sind Methoden, die Bestandteil der Fixnum-Klasse sind. Sie nutzen die Zahl, welche ein Fixnum-Objekt beinhaltet, als Ausgangsbasis für eine Schleife. Mit diesen Methoden können Sie eine exakte Anzahl an Schleifendurchläufen festlegen.

Die einfachste von ihnen ist `times`. Mit dieser Methode können Sie fast wortwörtlich beispielsweise *drei Mal* ausdrücken. Den Code, der entsprechend oft ausgeführt werden soll und mit *Schleifenkörper* oder *-rumpf* umschrieben wird, notieren Sie dahinter und zwischen geschweiften Klammern oder zwischen `do` und `end`. Er kann mehrere Zeilen umfassen. Es hat sich etabliert, den Code innerhalb des Schleifenkörpers um zwei Leerzeichen nach rechts einzurücken. Ist einfach übersichtlicher. Das menschliche Auge erfasst das Ende des Schleifenrumpfs viel schneller.

Beispiel 2-75: Dreimal, bitte!

```
a = 5
3.times {
  a = a + 6
}
a # => 23
3.times do
  a = a - 6
end
a # => 5
```

Die Methode `times` führt den Code jeweils dreimal aus. Intern wird dabei übrigens von 0 bis 2 gezählt. Glauben Sie nicht? Hier der Beweis: Alle Schleifenmethoden liefern Ihnen auf Wunsch den internen Zähler mit. Sie können ihn abfragen, indem Sie als Erstes innerhalb des Schleifenkörpers eine Variable bestimmen, die ihn aufnehmen soll. Setzen Sie diese Variable, welche übrigens nur innerhalb des Schleifenkörpers existiert, zwischen zwei `|`-Zeichen.

Beispiel 2-76: Schleife mit Zählervariable

```
b = ""  
5.times { |z|  
  b << z.to_s  
}  
b # => "01234"
```

Sollte Ihnen gar nicht passen, dass Ruby beim Zählen von 0 beginnt, dann können Sie mit den Methoden `upto` und `downto` nicht nur Einfluss auf den Startwert nehmen, sondern auch den Endwert der Schleife bestimmen, und zwar in zwei Richtungen: `upto` zählt *aufwärts* und `downto` zählt – bitte spielen Sie jetzt einen Trommelwirbel ein – *abwärts*. Dementsprechend müssen auch Start- und Endwerte gewählt werden. Während `upto` erst mit einem Startwert funktioniert, der kleiner als der Endwert ist, gibt sich `downto` erst mit dem umgekehrten Fall zufrieden. Es gilt das Schema `<Startwert>.upto(<Endwert>)` beziehungsweise `<Startwert>.downto(<Endwert>)`. Bei beiden Methoden haben Sie wie bei `times` auf Wunsch Zugriff auf den internen Zähler.

Beispiel 2-77: Auf und ab

```
c = Array.new  
10.upto(13) { |cnt|  
  c.push(cnt)  
}  
c # => [10, 11, 12, 13]  
150.downto(140) { |cnt|  
  c.push(cnt - c[0])  
  c.shift  
}  
c # => [9, 136, 134, 132]
```

Eine weitere Methode, `step`, verschafft Ihnen noch mehr Kontrolle über den Schleifenablauf. Während `times`, `upto` und `downto` jeweils in Einerschritten zählen, richtet sich `step` diesbezüglich ganz nach Ihren Wünschen, solange diese als zweiter Parameter angegeben werden, ganzzahlig und ungleich 0 sind. Schließlich ist eine Schrittweite von 0 der Politik vorbehalten. Der erste Parameter gibt auch bei `step` den Endwert an. Bedenken Sie, dass Sie eine negative Schrittweite angeben müssen, wenn Sie von einem Startwert zu einem niedriger liegenden Endwert zählen lassen möchten. Der interne Zähler wird Ihnen auf gewohnte Art und Weise zur Verfügung gestellt.

Beispiel 2-78: Doppelschritte

```
d = Array.new(5,1)  
d # => [1, 1, 1, 1, 1]  
0.step(d.size - 1, 2) { |x|  
  d[x] = x  
}  
d # => [0, 1, 2, 1, 4]
```

Die Methoden `times`, `upto`, `downto` und `step` bieten Ihnen eigentlich schon alles, was Sie zum Bauen von Schleifen, die *x-mal* durchlaufen werden sollen, benötigen. Doch es gibt in Ruby auch allgemeinere Ansätze, was Schleifen betrifft. Diese drängen nun darauf, sich Ihnen vorzustellen. Der erste Vertreter dieser Art ist wohl der über viele Programmiersprachen hinweg bekannteste.

Der Klassiker: `for`-Schleifen

Die gute alte `for`-Schleife – es gibt sie selbstverständlich auch in Ruby. Für Umsteiger ist diese Tatsache natürlich besonders erfreulich. Mit ihr können Sie ebenfalls von einem Startwert zu einem Endwert zählen und einen Schleifenrumpf entsprechend oft ausführen lassen. Die Angabe der beiden Werte erfolgt hierbei mit einer *Range*. Beachten Sie, dass bei einer `for`-Schleife die Zählervariable Pflicht ist und direkt im Schleifenkopf notiert wird. Hierbei hat sich übrigens der Variablenbezeichner `i` als Standard etabliert, was hauptsächlich daran liegt, dass diese Variable fachsprachlich als *Iterator-Variable* bezeichnet wird.

Zum besseren Verständnis enthält das folgende Codebeispiel noch eine `upto`-Schleife, mit der Sie bereits etwas anfangen können. Beide Schleifen leisten aber das Gleiche.

Beispiel 2-79: upto und for im Gleichklang

```
e = 0
1.upto(10) do |i|
  e += i
end
e # => 55
e = 0
for i in 1..10
  e += i
end
e # => 55
```

Was passiert nun bei einer solchen `for`-Schleife intern? Zunächst müssen Sie die angegebene Range nicht unbedingt nur als Mittel zum Angeben eines Start- und eines Endwerts betrachten, sondern vielmehr als etwas, in dem 1, 2, 3, 4, 5, 6, 7, 8, 9 und 10 als Einzelteile enthalten sind. Die `for`-Schleife schnappt sich nach und nach bei jedem Durchlauf ein Einzelteil und übergibt es der Variablen `i`, die innerhalb des Schleifenrumpfs abgefragt werden kann. Das erfolgt so lange, bis alle Einzelteile der Range dieses Schicksal über sich ergehen gelassen haben.

Statt »Einzelteile« könnte man auch *Element* sagen, was gleichsam den Verdacht nahe legt, dass das auch mit Arrays funktioniert, oder? Und genau so ist es. Geben Sie statt einer Range ein Array an, so enthält `i` bei jedem Durchlauf ein Array-Element.

Beispiel 2-80: for-Schleife mit Array

```
f = ""
for i in ["Rails", "on", "Ruby"]
  f = "#{i} #{f}"
end
f.strip # => "Ruby on Rails"
```

Diese Vorgehensweise funktioniert auch mit Hashes. Allerdings kann, wie Sie bereits wissen, die Reihenfolge der einzelnen Elemente eines Hashs von der Erstellungsreihenfolge abweichen. Außerdem sieht eine for-Schleife sowohl *Schlüssel* als auch *Werte* als eigenständige Elemente an und übergibt sie somit nicht als zusammenhängendes Paar an die Iterator-Variable. So richtig nützlich ist das also nicht.

Schleifen mit Abbruchbedingung

Konzentrieren Sie sich also lieber auf eine weitere Möglichkeit in Ruby, Code mehrfach abzuarbeiten. Konkret blicken wir auf eine Schleife, die, einmal gestartet, endlos läuft – es sei denn, Sie formulieren eine Abbruchbedingung.

Sie werden jetzt quasi die Urschleife in Ruby kennen lernen. Deshalb heißt diese Schleife auch *Schleife*. Oder wie es im Englischen heißt: *loop*. Die Benutzung ist ganz einfach. Notieren Sie einfach das Schlüsselwort *loop* und fügen Sie den Schleifenkörper an, den Sie wie bei *times* mit geschweiften Klammern oder *do* und *begin* eingrenzen müssen. Platzieren Sie innerhalb des Schleifenkörpers unbedingt eine Abbruchbedingung, die, wenn sie erfüllt ist, die Methode *break* aufruft.



Sollten Sie keine Abbruchbedingung oder eine durch das Programm nicht erfüllbare Bedingung implementieren, so läuft und läuft und läuft die Schleife unendlich weiter. Von außen können Sie auf dieses Malheur mitunter nur mit einem beherzten Druck auf den Power-Knopf Ihres Rechners reagieren.

Da *loop* keine Zählschleife ist, gibt es auch keinen internen Zähler, den Sie abfragen können. Sollten Sie dennoch einen solchen brauchen, müssen Sie ihn selbst implementieren – so wie im folgenden Beispiel. Hier wird eine Zahl so lang mit Zufallszahlen addiert, bis das Ergebnis durch 6 teilbar ist. Der Zähler (hier *counter*) gibt anschließend Auskunft über die Anzahl der dafür benötigten Durchläufe.

Beispiel 2-81: Endlosschleife mit Abbruchbedingung

```
z = 0; counter = 0
loop {
  counter += 1
  z += rand(100)
  break if z % 6 == 0
}
counter # => 12
```

Die Abbruchbedingung wird hier, wie Ihr inzwischen messerscharfer Ruby-Blick sicherlich schon herausgefunden hat, über einen *Modifikator* realisiert. Wenn die Division von z durch 6 keinen Rest hinterlässt, wird `break` ausgeführt und die Schleife umgehend abgebrochen. Alles, was nach `break` innerhalb des Schleifenkörpers kommt, wird in diesem Fall nicht mehr beachtet.



Sie können auch an mehreren Stellen innerhalb des Schleifenkörpers `breaken` und somit sehr flexibel mehrere Abbruchsituationen realisieren.

Es gibt zwei weitere Schleifen in Ruby, die ebenfalls mit einer solchen Abbruchbedingung arbeiten, `while` und `until`. Allerdings ist dazu die Verwendung von `break` nicht erforderlich, da die Abbruchbedingung direkt im Schleifenkopf notiert wird.

Der Unterschied zwischen `while` und `until` liegt in der Interpretation der Bedingung. Während das Lebensmotto von `while` »Ich werde erst dann ruhen, wenn ich die Bedingung nicht mehr erfüllen kann.« lautet, verfährt `until` nach der Losung »Ich ackere, bis die Bedingung erfüllt ist.« Lassen Sie uns doch einmal das obige `loop`-Beispiel mit `while` und `until` formulieren.

Beispiel 2-82: until und while

```
z = 0; counter = 0
until z % 6 == 0 do
  counter += 1
  z += rand(100)
end # Ende der until-Schleife
counter # => 0
z = 0; counter = 0
while z % 6 != 0 do
  counter += 1
  z += rand(100)
end # Ende der while-Schleife
counter # => 0
```

Sie meinen, an den beiden Beispielen stimmt etwas nicht? Prinzipiell ist alles OK, `while` und `until` arbeiten vorschriftsmäßig. Die Angelegenheit ist nur deshalb problematisch, weil die Bedingung, die über die Beendigung der Schleife entscheidet, schon vor dem ersten Durchlauf überprüft wird. Und da zu diesem Zeitpunkt die Variable z den Wert 0 hat und damit durch 6 teilbar ist, kommt es erst gar nicht zum Durchlaufen der Schleife, weder bei `while` noch bei `until`.

Es gibt zwei Lösungen für dieses Problem. Wir könnten z schon vor dem Abarbeiten der Schleife einen Zufallswert zuweisen, so wie es sonst erst im ersten Schleifendurchlauf geschehen würde. Allerdings müssten wir dann auch die Variable `counter` mit 1 initialisieren, damit diese Änderung statistisch auch berücksichtigt wird.

Oder, und das ist zumindest in diesem Fall die bessere Lösung: Wir verlegen die Überprüfung der Bedingung hinter den Schleifenkörper. Schließlich können Sie in Ruby `while` und `until` nicht nur, wie im obigen Beispiel geschehen, als *vorprüfende* oder *kopfgesteuerte* Schleife realisieren, sondern auch als *nachprüfende* oder *fußgesteuerte*. Der Unterschied: Der Schleifenkörper wird mindestens einmal durchlaufen. Erst dann wird über die Bedingung entschieden. Kleiden Sie hier den Schleifenkörper in ein *begin-end*-Pärchen. Wir stellen um:

Beispiel 2-83: until und while – fußgesteuert

```
z = 0; counter = 0
begin
  counter += 1
  z += rand(100)
end until z % 6 == 0
counter # => 6
z = 0; counter = 0
begin
  counter += 1
  z += rand(100)
end while z % 6 != 0
counter # => 4
```

Sie haben hiermit alle grundlegenden Techniken zur Verwirklichung von Schleifen in Ruby kennen gelernt. Aber ich möchte Sie noch mit ein paar Hinweisen versorgen, über deren Kenntnis Sie möglicherweise während Ihrer Arbeit mit Ruby recht glücklich sein könnten.

Zählschleifen mit Abbruchbedingungen

Selbstverständlich können Sie `break` nicht nur bei `loop` einsetzen. Alle Schleifen kommen mit `break` klar. So können Sie Zählschleifen vor dem Erreichen des Endwerts abbrechen.

Beispiel 2-84: for-Schleife mit Abbruchbedingung

```
m = 0
for i in 1..30
  m += i
  break if m > 20
end
m # => 21
```

Damit lassen sich beispielsweise Schleifen realisieren, in deren Körper eine Bedingung erfüllt werden muss, ohne dass eine bestimmte Anzahl an Versuchen dabei überschritten wird.

Der Nächste, bitte!

Sie können mit next einen kompletten oder partiellen Durchlauf des Schleifenkörpers verhindern. Platzieren Sie next zum Beispiel in Verbindung mit einer Bedingung in den Schleifenkörper, so wird der nachfolgende Teil des Schleifenkörpers nur dann berücksichtigt, wenn diese Bedingung nicht erfüllt ist. Andernfalls geht die Schleife zum nächsten Element über. Das folgende Beispiel zählt die Anzahl der Zahlen zwischen 5 und 20, die nicht durch 5 teilbar sind.

Beispiel 2-85: upto wird wählerisch

```
n = 0
5.upto(20) do |o|
    next if o % 5 == 0
    n += 1
end
n # => 12
```

Sie können next wie break (und auch mit break zusammen) an mehreren Stellen innerhalb des Schleifenkörpers platzieren. Zur Verdeutlichung der Funktionsweise von next sei noch einmal explizit auf den Unterschied zwischen break und next hingewiesen: Während break die gesamte Abarbeitung der Schleife stoppt, beendet next lediglich den aktuellen Durchlauf. Die Schleife wird aber – vorausgesetzt, es ist noch mindestens ein Durchlauf vorgesehen – mit dem nächsten Element fortgesetzt.

Doppelschleifen

Sie können auch mehrere Schleifen ineinander verschachteln. Dabei ist es unerheblich, welche Typen Sie verwenden. Eine while-Schleife hat kein Problem damit, eine for-Schleife zu umgeben, in deren Schleifenkörper eine loop-Schleife ihre Runden dreht. Lassen Sie uns einen Blick auf zwei Beispiele werfen. Im ersten erleben Sie eine for-for-Kombi, in der zweiten kooperieren eine fußgesteuerte until- und eine darin eingeschlossene upto-Schleife miteinander. Beide Beispiele erzeugen den gleichen Ergebnis-String.

Beispiel 2-86: Schleifen um Schleifen

```
q = ""
for i in 'a'..'c'
    for j in 1..2
        q.concat(i + j.to_s)
    end
end
q # => "a1a2b1b2c1c2"
q = ""; r= 'a'
begin
    1.upto(2) { |s|
```

Beispiel 2-86: Schleifen um Schleifen (Fortsetzung)

```
    q << r << s.to_s  
}  
end until r.succ! > 'c'  
q # => "a1a2b1b2c1c2"
```

Das Grundprinzip hier ist: Bei jedem Durchlauf der äußeren Schleife wird die innere von vorn gestartet. Erst wenn die innere Schleife komplett durchlaufen ist, kann sich die äußere Schleife ihrem nächsten Durchlauf widmen. Wichtig bei mehreren ineinander verschachtelten Schleifen ist das richtige Schließen der Schleifenkörper. Sie können beispielsweise nicht den Schleifenkörper der äußeren Schleife dichten machen, während der inneren noch offen ist. Außerdem sollten Sie unbedingt unterschiedliche Iteratoren-Variablen benutzen.

Die Könige unter den Schleifen: Iteratoren

Sie werden nun eine Funktionalität kennen lernen, die zu den besonderen Schätzen von Ruby gehört: Iteratoren. Nicht wenige Programmiersprachen beneiden Ruby darum und versuchen, etwas ähnlich Gartetes wenigstens ansatzweise zu simulieren. Die Ergebnisse regen nicht selten zum Schmunzeln an. Die Eleganz rubyscher Iteratoren erreichen sie jedenfalls nicht. Denn Rubys Iteratoren sind absolut *cool* – und glauben Sie mir, dieses Wort benutze ich wirklich ausgesprochen selten.

Damit Sie wissen, wovon ich überhaupt rede, verrate ich Ihnen etwas Erstaunliches: Sie haben bereits *vier* Iteratoren kennen gelernt. Es waren *numerische Iteratoren*: times, upto, downto und step. Na, erstaunt, was Sie schon alles beherrschen?

Sicher erinnern Sie sich noch an die Kennzeichen der vier Genannten: Sie waren grundsätzlich als Methoden eines Objekts implementiert, wobei die Zahl, die so ein Objekt beinhaltet, stets Startwert für die Schleife war. Ausgehend von diesem Wert und gegebenenfalls vorhandenen Parametern war es Ruby möglich, Schleifendurchläufe für eine konkrete Menge an Objekten durchzuführen. Bei times war es beispielsweise das Objekt selbst, welches durch seinen Wert die Anzahl der Durchläufe festlegte. Und damit wurde gleichsam eine Menge an ganzzahligen Objekten bestimmt, die, eins nach dem anderen, dem Schleifenkörper übergeben wurden. In dem stand, was mit jedem Objekt geschehen sollte. Der Schleifenkörper ist bei Iteratoren übrigens ein *Block*, der wiederum ein besonderes Sprachmittel von Ruby ist.

Blöcke spielen bei Ruby auch noch an anderen Stellen eine wichtige Rolle, die uns vorerst nicht interessieren sollen. Für jetzt bleibt festzuhalten, dass Blöcke ausnahmsweise keine Objekte sind und dass sie stets nur in Verbindung mit einer Methode genutzt werden können.



Damit Ruby weiß, dass ein Block zu einer Methode gehört, muss die öffnende geschweifte Klammer oder das Schlüsselwort `do` in der gleichen Zeile des Methodenaufrufs notiert werden.

Ihr wichtigstes Merkmal kennen Sie aber schon: Sie werden durch `{` und `}` oder, je nach Geschmack, durch `do` und `begin` umgeben. Ein Block erhält einen Parameter, der durch die Iterator-Methode mit dem Element, das gerade an der Reihe ist, gefüllt wird. Das Ding wird meist *Blockvariable* genannt und als erstes im Block zwischen zwei `|`-Zeichen geklemmt. Aber das kennen Sie auch schon, wenngleich bislang eher als *Zählervariable*.

Wenn Sie also verstanden haben, wie `times`, `upto`, `downto` und `step` funktionieren, dann haben Sie auch das faszinierende Thema Iteratoren schon so gut wie begriffen. Sie müssen nur noch eine Vorstellung davon bekommen, was passiert, wenn statt einer Menge von Zahlen beispielsweise eine Menge von Array-Elementen die Basis bildet.

each, der Ur-Iterator

Arrays und Hashes sind die Hauptnutznießer von Iteratoren in Ruby; auch auf Ranges sind sie anwendbar. Der Iterator schlechthin ist dabei `each`. Alle anderen Iteratoren von Arrays, Hashes und Ranges basieren auf der klassenspezifischen Implementation von `each`. Umso wichtiger ist also das Verstehen dieser Methode.

Nehmen wir an, Sie möchten über die drei Fixnum-Objekte 0, 1 und 2 *iterieren*. Blicken wir noch einmal kurz auf den `times`-Iterator und wie er die Menge der drei Zahlen behandelt.

Beispiel 2-87: times und die Fixnum-Objekte 0, 1 und 2

```
3.times { |z|
  print(z)
} # 012
```

Alle Zahlen von 0 bis 2? Das ist ein optimales Einsatzgebiet von Ranges. Der `each`-Iterator der Range-Klasse sieht jede Zahl, die die Range beinhaltet, als ganzzahliges Objekt, wodurch sich die Menge an Objekten bildet, über die `each` iteriert. Angefangen beim Startwert der Range wird jedes Objekt dabei in Form der Variable `z` durch den Block geschickt. Denken Sie aber daran, dass das nur funktioniert, wenn Start- und Endwert Ihrer Range ganzzahlig sind.

Beispiel 2-88: each und die Fixnum-Objekte 0, 1 und 2 als Range

```
(0..2).each { |z|
  print(z)
} # 012
```

Die drei Fixnum-Objekte können Sie auch in einem Array als eine Menge von Objekten ausdrücken. Die Iterator-Methode `each` durchläuft dann jedes Array-Element von links nach rechts.

Beispiel 2-89: each und die Fixnum-Objekte 0, 1 und 2 als Array

```
[0, 1, 2].each { |z|
  print(z)
} # 012
```

Für den Gebrauch von `each` in Verbindung mit einem Hash-Objekt sollten Sie wissen, dass `each` dem Block jeweils ein komplettes Schlüssel/Wert-Paar übergibt. Die Übergabe dieses Paares erfolgt dabei in *zwei* getrennten Variablen, wobei zuerst der Schlüssel und dann der dazugehörige Wert in die zwei blockinternen Variablen geschrieben werden. Das obige Beispiel als Hash-Variante sähe somit beispielsweise so aus:

Beispiel 2-90: each und die Fixnum-Objekte 0, 1 und 2 als Hash

```
{:a=>0, :b=>1, :c=>2}.each { |key, value|
  print(value)
}
```

Die Hash-Klasse hält aber auch zwei Varianten bereit, mit denen Sie entweder nur den Schlüssel oder nur den Wert pro Durchlauf an den Block übergeben. Sie heißen `each_key` und `each_value`.

Beispiel 2-91: each_value und die Fixnum-Objekte 0, 1 und 2 als Hash

```
{:a=>0, :b=>1, :c=>2}.each_value { |z|
  print(z)
}
```

Wie Sie sehen, ist die Methode `each` noch ziemlich *schleifig*. Will sagen: Hier erkennt man sehr leicht, dass `each` auch ganz einfach durch `for` oder andere Schleifen ersetzt werden könnte. Sie ist eben recht allgemein gehalten. Mit `each` können Sie sowohl ein komplexes Array nach ihren Wünschen formatiert ausgeben als auch grundsätzliche Dinge elementweise prüfen oder jedes Element verändern. Bei den Iteratoren-Methoden, die ich Ihnen jetzt vorstelle, ist es nicht ganz so offensichtlich, dass da eigentlich eine Schleife im Hintergrund arbeitet. Sie werden auch merken, dass Sie während des Benutzens anderer Iterator-Methoden gar nicht mehr an das Schleifenmodell, sondern vielmehr an auf einzelne Elemente bezogenen Code denken werden.

Typische Iteratoren für Arrays

Zunächst möchte ich Ihnen drei heißbegehrte Methoden vorstellen, die Sie mit Arrays nutzen können. Sie werden bald sehen: Mit `collect`, `find_all` und `reject` haben Sie ganz mächtige und gleichzeitig äußerst elegante Werkzeuge für den Umgang mit Arrays zur Hand.

collect

Die Methode collect ist eine der beliebtesten Iterator-Methoden für Arrays, möglicherweise sogar die beliebteste. Sie sammelt Daten des Ausgangsobjekts, verändert sie gegebenenfalls und gibt sie dann als neues Array aus. Versehen Sie die Methode mit einem Ausrufungszeichen, wird das Ausgangs-Array direkt bearbeitet. Das Prinzip kennen Sie ja bereits.

Was mit jedem Element passieren soll, geben Sie innerhalb des Blocks an. Hierbei gilt es zwei Dinge zu beachten: 1. Nehmen Sie keine Änderungen am Parameter vor – weisen Sie ihm also nichts zu und verändern Sie ihn nicht mit ausführungszeitlichen behafteten Methoden. Änderungen am Parameter sind direkte Änderungen am jeweiligen Objekt und damit direkte Änderungen am Array. Somit würden Sie die Elemente des Ausgangsarrays auch mit collect und nicht nur mit collect! direkt verändern. 2. Denken Sie daran, dass das Ergebnis des ganzen Blocks die Auswertung des letzten im Block vorkommenden Ausdrucks ist. Dieses Ergebnis bildet wiederum den Wert für ein Element im Ergebnisarray.

Das folgende Beispiel wandelt alle großbuchstabigen Elemente eines Arrays in kleinbuchstabige um – und umgekehrt. Das Ergebnis wird in einem neuen Array ausgegeben.

Beispiel 2-92: collect erzeugt in Array b das elementweise veränderte Array a

```
a = ["a", "B", "C", "d", "E"]
b = a.collect do |chr|
  chr.swapcase
end
b # => ["A", "b", "c", "D", "e"]
```

Der Ausdruck `chr.swapcase` ist hier das Entscheidende. Hier findet die Anpassung statt. Was dabei ausgewertet wird, kommt als neues Element ins Array b. Dieses Element trägt den gleichen Index wie das jeweilige Ausgangselement des Arrays a. Um Zweit-Array b zu sparen und die Änderungen direkt in a zu speichern, genügen folgende Zeilen.

Beispiel 2-93: Array a wird elementweise bearbeitet

```
a.collect! do |chr|
  chr.swapcase
end
a # =>["A", "b", "c", "D", "e"]
```

Statt `collect` und `collect!` können Sie auch die Methoden `map` und `map!` nutzen. Sie sind einfach nur Synonyme.

Vorsicht bei heterogenen Arrays!

Ich möchte Sie an dieser Stelle auf einen kleinen Stolperstein hinweisen. Er liegt Ihnen meist dann im Weg, wenn Sie über Arrays mit Objekten unterschiedlicher Klassen iterieren und innerhalb eines Blocks klassenspezifische Methoden auf die Blockvariable anwenden möchten. So wie beispielsweise swapcase. Diese Methode ist nur für ein String-Objekt implementiert. Was aber, wenn chr mal ein Array-Element repräsentiert, für das diese Methode nicht existiert? Es tritt ein Fehler auf. Der kann vermieden werden, wenn Sie vor Benutzung dieser Methode ihre Existenz in der jeweiligen Klasse überprüfen. Das geht ganz einfach mit der Methode respond_to?, der Sie als Parameter den fragwürdigen Methodenbezeichner übergeben. Gibt respond_to? true zurück, dann können Sie die gewünschte Methode anwenden. Bei false sollte dringend Abstand davon genommen werden.

Nehmen wir an, wir ergänzen das Array a unseres Beispiels um einen ganz-zahligen Wert. Fixnum-Objekte kennen keinen swapcase-Befehl, also soll dieses Element unbearbeitet dem neuen Array übergeben werden. Um das *unbearbeitet übergeben* realisieren zu können, müssen wir sicherstellen, dass die nackte Blockvariable, in diesem Fall chr, als Letztes ausgewertet wird. Nur so kommt der Wert schließlich in das Ergebnisarray. Diese Auswertung soll natürlich nur dann stattfinden, wenn swapcase nicht machbar ist.

```
a = ["a", "B", "C", "d", "E"]
a.push(1)
b = a.collect do |chr|
  if chr.respond_to?("swapcase")
    chr.swapcase
  else
    chr
  end
end
b # => ["A", "b", "c", "D", "e", 1]
```

Übrigens: Würden wir auf die Auswertung des Ausdrucks chr verzichten, erhielte Array b statt 1 ein Element nil.

Alternativ zum Check des Vorhandenseins einer Methode in einer Klasse können Sie auch die Klasse selbst überprüfen. Dazu brauchen Sie lediglich die Klassenzugehörigkeit der Blockvariable via class-Methode abzufragen. Somit ergibt sich eine zweite Variante:

```
a = ["a", "B", "C", "d", "E"]
a.push(1)
b = a.collect do |chr|
  if chr.class == String
    chr.swapcase
```



```

else
  chr
end
end
b # => ["A", "b", "c", "D", "e", 1]

```

Grundsätzlich sollten Sie diese Art der Absicherung innerhalb jedes Iterator-Blocks, aber auch innerhalb jedes anderen Schleifenkörpers realisieren, sobald die Möglichkeit besteht, dass dort mit Elementen unterschiedlicher Klassenzugehörigkeit umgegangen werden muss.

find_all

Sinn und Zweck von `find_all` sind schnell erklärt. Fast reicht schon die deutsche Übersetzung dafür: *Finde alle!* Und zwar alle, die bestimmte Kriterien erfüllen. Selbige werden innerhalb des Blocks formuliert, der `find_all` folgt. Wie es sich für eine Iterator-Methode gehört, geht `find_all` das Array elementweise durch. Für jedes Element werden die festgelegten Kriterien überprüft – sprich: Der Block wird mit dem Element als Blockvariable ausgeführt. Wird der Block als `true` ausgewertet, kommt das Element in das Ergebnis-Array. Wenn nicht, dann nicht. Die Variante `find_all!` ist ebenfalls vorhanden.

Das folgende Beispiel findet *alle* Elementen aus Array `c`, die die Buchstabenkombination *aus* enthalten und packt sie – und nur sie – ins Array `d`.

Beispiel 2-94: find_all Wörter mit aus

```

c = ["Pflaume", "Bauschaum", "Auster"]
d = c.find_all { |item|
  item.downcase.index("aus") != nil
}
d # => ["Bauschaum", "Auster"]

```

Wie gesagt, ich denke, `find_all` beschreibt ganz gut, was `find_all` leistet. Sollten Sie anderer Meinung sein, erfreue ich Sie vielleicht mit der Botschaft, dass Sie alternativ auch `select` beziehungsweise `select!` nutzen können. Die beiden Methoden finden auch alles, nennen den Vorgang aber »Selektieren«.

reject

Es gibt auch den zu `find_all` umgekehrten Weg. Mit `reject` können Sie einzelne Elemente eines Arrays zurückweisen. Und wo die Abweisungskriterien hinkommen, ahnen Sie bestimmt längst. Der Block muss zu `false` ausgewertet werden, damit ein Element Bestandteil des Ergebnisses wird. Alle Elemente, bei denen das nicht der Fall ist, deren Auswertung also `true` ergibt, werden gnadenlos *rejected*. Was mag also im Ergebnis-Array stehen, würden wir im vorhergehenden Beispiel `find_all` gegen

reject austauschen? Ganz klar: Alle Wörter mit *aus* würden diesmal *nicht* berücksichtigt werden. Einzig die Pflaume würde den Weg nach Array d schaffen.

Beispiel 2-95: reject Wörter mit aus

```
c = ["Pflaume", "Bauschaum", "Auster"]
d = c.reject { |item|
  item.downcase.index("aus") != nil
}
d # => ["Pflaume"]
```

Ich finde, das ist eine gute Gelegenheit, um darauf hinzuweisen, dass bei Iteratoren stets ein Array als Ergebnis ausgespuckt wird – selbst wenn dieses nur ein einziges Objekt enthält. Lediglich einige Iteratoren geben `nil` zurück, wenn das Ergebnis theoretisch ein gänzlich leeres Array wäre.

Sortieren nach eigenen Regeln

Die `sort`-Methode der Array-Klasse haben Sie bereits kennen gelernt. Je nach Beschaffenheit des Arrays werden die dort enthaltenen Objekte mit Hilfe der `sort`-Methode beispielsweise alphabetisch oder nummerisch sortiert. Aber nicht alles in dieser Welt möchte und kann so sortiert werden. Und deshalb gibt es eine Möglichkeit, wie Sie selbst Sortierrichtlinien formulieren können. Wenn Sie sich bewusst sind, von welchem Thema dieser Kasten umzingelt wird, ahnen Sie möglicherweise schon, wie das gehen soll.

Es sollte Sie also nicht erstaunen, wenn ich Ihnen mitteile, dass das mit einer Iteratoren-Methode bewerkstelligt werden kann. Überraschungspotenzial birgt aber möglicherweise noch der Hinweis, dass `sort` selbst diese Aufgabe übernehmen kann. Lassen Sie uns ein bisschen mit dem `sort`-Iterator experimentieren. Basis unser lehrreichen Spielchen soll ein Array sein, das Schweizer Städte und Städtchen sammelt.

```
cities = ['Luzern', 'Olten', 'Basel', 'Horw', 'Zug', 'Illnau-
Effretikon', 'Zofingen']
```

Wenden wir die `sort`-Methode auf bekannte Art und Weise an, wird das Array alphabetisch sortiert. A-Wörter kommen vor B-Wörter, H-Wörter vor V-Wörter – Sie kennen das und ahnen schon, dass Zug eher einen der hinteren Plätze belegen wird. Rein alphabetisch betrachtet.

```
cities.sort
=> ["Basel", "Horw", "Illnau-
Effretikon", "Luzern", "Olten", "Zofingen", "Zug"]
```

Nehmen wir nun an, wir möchten eine Variante implementieren, die umgekehrt sortiert. Auch dafür ist `<=>` zu gebrauchen. Sie müssen nur die Seiten vertauschen:



```

cities.sort { |city1, city2|
  city2 <=> city1
} # => ["Zug", "Zofingen", "Olten", "Luzern", "Illnau-
Effretikon", "Horw", "Basel"]

```

Oder wie wär's mit dieser Variante, bei der das Vorzeichen des Ergebnisses der Analyse verändert wird?

```

cities.sort { |city1, city2|
  (city1 <=> city2) * -1
} # => ["Zug", "Zofingen", "Olten", "Luzern", "Illnau-
Effretikon", "Horw", "Basel"]

```

Das Umkehren des Standardverhaltens von `sort` kommt unserem Ziel des Sortierens nach eigenen Regeln zwar nahe, aber das hätte man auch mit der Anwendung der `reverse`-Methode auf das Ergebnis-Array erreichen können.

Also, wir wär's, wenn wir die Elemente des `cities`-Arrays nach Länge sortieren würden? Das geht ganz einfach, wenn man sich vor Augen hält, dass die Differenz zwischen den Längen zweier Wörter mal positiv, mal negativ und auch mal 0 sein kann. Das passt doch prima zu dem, was `sort` als Iterator von seinem Block erwartet.

```

'Basel'.length - 'Olten'.length # => 0
'Zug'.length - 'Horw'.length # => -1
'Basel'.length - 'Zug'.length # => 2

```

Damit ist klar, wie der Block unserer nach Länge sortierenden `sort`-Methode aussen muss:

```

cities.sort { |city1, city2|
  city1.length - city2.length
} # => ["Zug", "Horw", "Olten", "Basel", "Luzern", "Zofingen", "Illnau-
Effretikon"]

```

Nun fügen wir noch ein zweites Kriterium beim Sortieren hinzu: Sollten zwei Elemente die gleiche Anzahl an Buchstaben haben – die Differenz also 0 sein – soll noch alphabetisch sortiert werden. Dazu können wir das Ergebnis der Differenz beispielsweise in einem Fixnum-Objekt, hier `diff`, zwischenspeichern.

```

cities.sort { |city1, city2|
  diff = city1.length - city2.length
  if diff == 0
    city1 <=> city2
  else
    diff
  end
} # => ["Zug", "Horw", "Basel", "Olten", "Luzern", "Zofingen", "Illnau-
Effretikon"]

```

Besteht kein Unterschied in der Länge, wird der Block anhand des `<=>`-Operators ausgewertet. Andernfalls kommt wie bereits bekannt die Differenz zur Auswertung, was hier durch das schlichte Notieren von `diff` erfolgt.

Typische Hash-Iteratoren

Grundsätzlich gilt, dass fast alle Iteratoren von Arrays auch bei Hashes funktionieren. Es gibt aber einige Besonderheiten bei der Anwendung. Eine haben Sie bereits in Zusammenhang mit each kennen gelernt. Ein Schlüssel/Wert-Paar braucht in einem Block zwei Blockvariablen. Die erste birgt den Schlüssel, die zweite den Wert in sich. Als Ergebnis geben aber auch Hash-Iteratoren ein Array aus. Und das hat ein Format, welches Sie bereits von der Hash-Methode `to_a` kennen: Ein Schlüssel/Wert-Paar wird zu einem zweielementigen Array, dessen erstes Element den Schlüssel, das zweite den Wert aufnimmt. Jedes dieser Arrays kommt wiederum als Element in das Ergebnis-Array.

Diese Tatsache ist auch der Grund, weshalb es keine Hash-Iteratoren gibt, deren Methodenbezeichner mit einem Ausrufungszeichen endet. Somit sollten Sie bei Hashes von der Verwendung von `collect!`, `find_all!` oder `reject!` absehen. Außerdem sollten Sie stets bedenken, dass die Reihenfolge der Hash-Elemente selten so ist, wie erstellt oder anderweitig gedacht.

Das folgende Beispiel soll Ihnen zeigen, wie Sie an Schlüssel und Wert eines Pärchens innerhalb eines Iterator-Blocks gelangen, und welches Format Sie als Ergebnis erwarten dürfen.

Beispiel 2-96: find_all mit einem Hash

```
e = { :on=>2, :standby=>1, :off=>0 }
f = e.find_all { |state, state_nr|
  state_nr != 0
}
f # => [[:on, 2], [:standby, 1]]
```

Selbstverständlich sind Sie nicht daran gebunden, Ihre Kriterien für die Aufnahme des Schlüssel/Wert-Paares in das Ergebnis-Array nur mit dem Wert, hier `state_nr`, zu formulieren. Sie können natürlich auch dafür den Schlüssel heranziehen.

Beispiel 2-97: find_all mit einem Hash (2)

```
e = { :on=>2, :standby=>1, :off=>0 }
f = e.find_all { |state, state_nr|
  state != :off
}
f # => [[:on, 2], [:standby, 1]]
```

Haben Sie Iteratoren verstanden? Dann sind Sie einen großen Schritt vorangekommen auf Ihrem Weg zum Rails-Entwickler. Ein noch viel größerer Schritt wartet nun auf Sie.

Methoden

Alle Codeschnipsel dieses Kapitels und auch die Beispielanwendungen hatten eins gemeinsam: Sie funktionierten recht linear. Von oben nach unten. Teilweise wurden Quelltextteile mehrfach aufgeschrieben, weil eine Funktionalität an mehreren Stellen innerhalb des Programms benötigt wurde. Diese Art und Weise zu programmieren ist natürlich hochgradig ineffizient und zudem völlig untypisch für gute Ruby-Programme. Die nächsten Seiten sagen diesem Missstand den Kampf an. Ihre Waffe dabei: *Methoden*.

Sie sind innerhalb dieses Kapitels bereits Dutzenden von Methoden begegnet und haben dabei sicherlich ihr Grundprinzip erkannt. Sie hatten alle einen meist aussagekräftigen Namen, über den man sie aufrufen konnte. Und so manch eine lehzte zudem nach Zusatzinformationen, die zum korrekten Ausführen der Methode nötig waren und als Parameter an die Methode übergeben wurden. Außerdem konnten alle Methoden etwas. Klar, warum hätte man Sie sonst benutzt?! Die meisten Methoden gaben einen Wert zurück, mal die durch die Methode manipulierten Parameter, mal true, mal nil, mal das Ergebnis einer Zählung.

Also: Methoden begeistern mit speziellen Fähigkeiten, geben was zurück, tragen bedeutungsschwangere Namen und benötigen für ihre Funktionstüchtigkeit unter Umständen Parameter. Und noch eine Eigenschaft besitzen Methoden: Sie können selbst welche schreiben.

Eigene Methoden schreiben

Das Grundmuster einer *Methodendeklaration* ist ganz simpel. Beginnen wir mit dem *Methodenkopf*. Der, und nur der, steht in der ersten Zeile einer Methodendeklaration. Er besteht aus dem Schlüsselwort def, gefolgt vom Methodennamen und den möglichen Parametern. Bei der Wahl eines Namens können Sie sich an den Namensrichtlinien für Variablen orientieren: Mit einem Kleinbuchstaben beginnen und kleine Buchstaben, Zahlen oder Unterstriche folgen lassen. Als letztes Zeichen des Methodennamens sind auch noch ?, ! und = erlaubt. Wie Sie bereits wissen, deuten diese Zeichen auf ein besonderes Können der Methode hin. Mit empty? oder slice! haben Sie solche Methoden bereits kennen gelernt. Es gibt aber auch einige Sonderfälle, die gar keine Buchstaben oder Zahlen im Methodenbezeichner benutzen. Wie sonst könnten Sie sonst beispielsweise <=>, + oder / nutzen? Auch diese Operatoren sind als Methoden implementiert.

Soll Ihre Methode mit Parametern versorgt werden, folgen diese ohne Leerzeichen dem Methodennamen in Klammern. Jeder Parameter wird dabei durch eine Variable repräsentiert, die Sie innerhalb des gleich folgenden Methodenkörpers nutzen können.

Der *Methodenkörper*, der frühestens in der zweiten Zeile einer Methodendeklaration beginnen darf, nimmt Code auf, der nur die Aufgabe hat, den Anspruch, den ein Benutzer an die Methode hat, auch zu erfüllen. Wie erwähnt, kann er die Parametervariablen benutzen, die übrigens nur innerhalb des Methodenkörpers gültig sind.

Eine weitere wichtige Aufgabe des Methodenkörpers besteht darin, festzulegen, was die Methode zurückgeben soll – also welcher Wert an den Programmteil übergeben werden soll, der die Methode aufgerufen hat. Dies erfolgt durch das Schlüsselwort `return`, dem ein Leerzeichen und dann der Wert folgen. Beachten Sie, dass `return` die Verarbeitung der Methode abbricht. Alles, was nach `return` und dem Rückgabewert folgt, findet also keinerlei Beachtung. Der Methodenkörper wird mit `end` in einer separaten Zeile abgeschlossen. Notieren Sie Ihre komplette Methodendefinition am besten immer vor dem Code, der die Methode benutzt.



Achten Sie bei dem Entwurf einer Methode darauf, dass sie wirklich nur einen spezifischen Teil der Gesamtaufgaben Ihres Programms löst. Teilen Sie Großaufgaben lieber in mehrere Methoden auf.

Schauen wir uns das einfach praktisch an und entwerfen wir eine einfache Methode, die eine Byte-Angabe in KByte umwandelt. Der Methode soll es dabei völlig egal sein, ob ein ganzzahliger oder ein gebrochener Zahlenwert übergeben wird.

Beispiel 2-98: `byte_to_kbyte` rechnet Byte in KByte um

```
def byte_to_kbyte(value)
  return value.to_f / 1024
end
```

Die Methode steht, probieren wir sie aus. Bisher haben Sie Methoden meist nur als solche eines bestimmten Objekts aufgerufen, zum Beispiel `"Ruby".reverse`. Lediglich bei `puts` und `gets` war das nicht nötig. Bei `byte_to_kbyte` ist das genauso. Wir haben die Methode nicht als Teil einer Klasse implementiert, also kann es auch kein Objekt geben, das für den Aufruf herangezogen werden kann. Probieren wir einfach drauf los.

```
byte_to_kbyte(1000) # => 0.9765625
byte_to_kbyte(128_000) # => 125.0
byte_to_kbyte(18900.0) # => 18.45703125
byte_to_kbyte("1024 Bytes") # => 1.0
```

Ja, auch Strings mit Einheit werden akzeptiert. Darum kümmert sich die Implementation von `to_f` innerhalb der String-Klasse, sofern der numerische Wert den String eröffnet. Aber das nur als kleine erinnerungsfördernde Nebenbemerkung.

Es ist möglich, innerhalb des Methodenkörpers auf `return` zu verzichten und dennoch einen Rückgabewert anzubieten. Dann gibt die Methode den zuletzt innerhalb des Methodenkörpers ausgewerteten Ausdruck zurück. Bei `byte_to_kbyte` ist das die Berechnung. Ein Methodenkörper ohne `return` wird garantiert komplett durchlaufen. Hier also die Alternative. Übrigens: Wenn Sie eine Methode ein zweites Mal verfassen, wird die alte Methode gleichen Namens überschrieben.

Beispiel 2-99: byte_to_kbyte ohne return

```
def byte_to_kbyte(value)
  value.to_f / 1024
end
```

Jetzt werden wir unsere kleine Methode ein bisschen frisieren und aufmotzen. Fangen wir damit an, dass sie bei einem Aufruf gleich mehrere Werte annehmen und auf einmal umrechnen soll. Die Anzahl der Werte soll dabei flexibel festlegbar sein. Es stellen sich drei Fragen in diesem Zusammenhang. 1. Wie sollen mehrere Parameter übergeben werden? 2. Wie sollen auf dieser Basis gleich mehrere Werte berechnet werden? Und 3. Wie sollen mehrere Werte auf einmal zurückgegeben werden?

Die erste Frage ist einfach zu beantworten: Wir fügen einfach weitere Parameter hinzu. Damit `byte_to_kbyte` mit zweien zurechtkommt, müsste der Methodenkopf so aussehen:

```
def byte_to_kbyte(value1, value2)
```

Das funktioniert auch, allerdings nur mit exakt zwei Parametern. Soll die Methode weiterhin nur einen Wert berechnen, müsste ein zweiter zwangsläufig mit angegeben werden, sonst hagelt es Fehlermeldungen. Es sei denn, man verfasst den Kopf so:

```
def byte_to_kbyte(value1, value2 = 0)
```

Bei dieser Variante wird `value2` automatisch der Wert 0 zugewiesen, sollte kein zweiter Parameter beim Aufruf von `byte_to_kbyte` angegeben worden sein. Sie können so einem Parameter einen Standardwert zuweisen. Wichtig hierbei: Nach einem mit einem Standardwert versehenen Parameter darf kein Parameter ohne kommen. Warum das so ist, erschließt sich leicht aus diesem Beispiel:

```
def testmeth(a, b = 3, c, d = 12, e, f)
  ...
end
...
testmeth(12, 18, 29, 30)
```

Ruby würde sich hier fragen müssen, ob denn die 30 beispielsweise der Wert für `d`, `e` oder `f` sein soll. Das muss vermieden werden, denn Verwirrung beim Interpreter ist das Letzte, was man als Programmierer gebrauchen kann. Daher die eben erwähnte Regel.

Aber wie dem auch sei, dieser Ansatz ist in einem anderen Kontext sicher brauchbar, und es ist gut, dass Sie ihn kennen, doch für unser Anliegen taugt er nichts. Besser ist die Variante, bei dem einen Parameter `value` zu bleiben und einfach ein *Sternchen* davor zu setzen. Dadurch erkennt Ruby, dass dieser Parameter bereit ist, mehrere Werte aufzunehmen. Aus `value` wird also `*value`.

Im Methodenkörper müssen Sie aber weiterhin auf die sternchenlose Variante des Parameternamens zugreifen. Außerdem müssen Sie den Parameter als Array behandeln. Jedes Element des Arrays steht dabei für einen der unbestimmt vielen an die Methode übergebenen Werte.

Die erste Frage ist damit geklärt. Frage Nummer zwei ist auch schon fast gelöst, denn Sie wissen nun, dass die Werte, die umgerechnet werden sollen, als Array vorliegen. Die Umrechnung kann also nur elementweise erfolgen. Und wenn in diesem Moment das Wort *Iteratoren* durch Ihren Kopf schwirrt, sind Sie schon ein echter Ruby-Profi. Aber auch beim Wort *Schleife* gebührt Ihnen Lob. Welcher Iterator soll es aber sein?

Da jedes Element bearbeitet werden soll, empfiehlt sich `collect` beziehungsweise `map`. Aber auch mit `each` kämen Sie ans Ziel. Der Iterator-Block, der bekanntlich nur einen Wert `v` und nicht das ganze Array `value` berücksichtigen muss, entspricht logischerweise der Berechnung, die wir vorher für einen Parameter genutzt haben.

```
value.collect { |v|  
  v.to_f / 1024  
}
```

Bekanntlich spuckt `collect` ein neues Array aus, das die geänderten Elemente des Ursprungsarrays enthält. Ein Array also. Eigentlich ist das die perfekte Antwort auf die dritte Frage. Ein Array, in dem alle Ergebnisse der Berechnungen fein säuberlich und unmissverständlich in einzelnen Päckchen verstaut sind, und das mit allerlei nützlichen Methoden beliefert wird, ist wahrlich ideal als Rückgabewert. Und noch ein Argument dafür: Eine Methode kann eh nur ein Objekt als Antwort ausliefern. Indem wir genau ein Array benutzen, umgehen wir diese Beschränkung. Die Methode `byte_to_kbyte` für beliebig viele Werte ist damit fertig.

Beispiel 2-100: `byte_to_kbyte` für viele Werte

```
def byte_to_kbyte(*value)  
  value.collect { |v|  
    v.to_f / 1024  
  }  
end
```

Um `byte_to_kbyte` nun mit mehreren Werten zu bestücken, trennen Sie diese beim Methodenaufruf ganz normal durch Kommas.

Beispiel 2-101: byte_to_kbyte mit mehreren Werten testen

```
byte_to_kbyte(1024, 2048, 4096, 8192) # => [1.0, 2.0, 4.0, 8.0]
byte_to_kbyte(10240, 40960) # => [10.0, 40.0]
byte_to_kbyte(102400) # => [100.0]
byte_to_kbyte # => []
```

Wie Sie sehen, gibt die Methode auch bei nur einem Wert ein Array als Ergebnis aus. Das liegt natürlich daran, dass wir einen Iterator benutzt haben, der ebenfalls unabhängig vom Datenmaterial ein Array erzeugt.

Noch ein Wort zu Verwendung von Parametern mit Sternchen-Präfix. Auch hier ist Vorsicht beim Festlegen der Parameterliste in einem Methodenkopf geboten: Sie dürfen nur einmal innerhalb und ganz am Ende einer Parameterliste erscheinen. Da beim Aufrufen der Methode die Bestandteile dieses Parameters wie ganz normale, einwertige Parameter durch Kommas getrennt werden, ist für Ruby unter Umständen nicht ersichtlich, welcher Parameter mit welchem Wert bestückt werden soll und welcher Wert zum Sternchenparameter gehört.

Richtig wertvoll werden Methoden erst dann, wenn sie, was in Ruby üblich sein sollte, in Klassen Verwendung finden. Ruby-Klassen bestehen übrigens nahezu ausschließlich aus Methoden. Und es ist gar nicht schwer, eigene Klassen zu schreiben.

Selbst gemachte Klassen

Wir stoßen damit in den Kern von Ruby vor. Dass Ruby objektorientiert ist, muss ich wohl nicht schon wieder erwähnen. Dennoch sei daran erinnert, dass Klassen das Verhalten von Objekten beschreiben. Und da in Ruby so gut wie alles ein Objekt ist, und auch Ihre eigenen Entwicklungen mit Ruby von diesem Grundsatz tunlichst nicht abweichen sollen, erklärt sich die Wichtigkeit von Klassen praktisch von ganz allein.

Und noch mehr spricht für Klassen und den objektorientierten Ansatz: Code, der einmal in einer Klasse notiert ist, kann als Schablone für viele, viele Objekte genutzt werden. Durch die Vereinbarung von Schnittstellen einer Klasse kann Code in zukünftigen Projekten bequem wiederverwendet werden. Quelltext wird zudem besser beherrschbar, wenn Teilaufgaben von einzelnen Klassen übernommen werden.

Klassen sind, nüchtern betrachtet, eine zusammenhängende Sammlung von Methoden, die alle in einem Kontext stehen. Die Fixnum-Klasse enthält beispielsweise hauptsächlich Methoden, die ausschließlich für die Verwendung ganzer Zahlen relevant sind. Oder erinnern Sie sich an das erste Kapitel und die Lok-Klasse? Sie enthielt die für den Betrieb eines Dampfrosses nötigen Methoden. Funktionen zum Kaffeekochen oder Preisausschreibenlösen waren dort nicht vorgesehen.

Empfänger und Methoden

Bevor es konkret wird, möchte ich Sie noch mit einem wichtigen Fachbegriff in der objektorientierten Programmierung vertraut machen. Es handelt sich um den *Empfänger* oder auch den *Receiver*. Damit wird das Objekt bezeichnet, durch das eine Methode aufgerufen wurde. In den folgenden Beispielen sind "Text", a und 6...12 Empfänger.

Beispiel 2-102: Beispielhafte Empfänger

```
"Text".reverse!  
a.class  
(6...12).first
```

Object, self und self

In Kürze werden Sie Ihre erste eigene Klasse schreiben. Aber zunächst möchte ich Ihnen eine überraschende Mitteilung machen. Sie haben bereits ohne Ihr Wissen eine Methode einer bestehenden Klasse geschrieben. Die hieß `byte_to_kbyte` und erweiterte die Klasse `Object`. Glauben Sie nicht? Überzeugen Sie sich selbst!



Der folgende Test funktioniert natürlich nur, wenn Ihre aktive Interactive-Ruby-Sitzung die Methode `byte_to_kbyte` kennt. Sollten Sie zwischenzeitlich Interactive Ruby verlassen haben, so blättern Sie am besten ein paar Seiten zurück und geben Sie die Methode noch einmal ein.

Geben Sie in Interactive Ruby `Object.methods` ein. Mit `methods` erhalten Sie ein Array aller in einer Klasse oder einem Objekt enthaltenen Methoden. Sie werden sehen, dass irgendwo in der großen Menge an Methoden der Klasse `Object` auch Ihre Methode `byte_to_kbyte` steckt. Wenn Sie die Suche scheuen, fragen Sie gezielt mit der bereits bekannten Methode `Object.respond_to?('byte_to_kbyte')`. Ruby wird Ihnen mit `true` antworten.

Die Klasse `Object` ist übrigens die Mutter aller Klassen. Sie ist in der Klassenhierarchie von Ruby ganz oben. Alle anderen Klassen basieren auf `Object` und erweitern die Basisklasse um die je nach Klasse nötigen Funktionalitäten.

Und dann muss ich Ihnen wohl etwas beichten – ich war da nämlich etwas unpräzise bei der Besprechung der Methode `byte_to_kbyte`. Sie kann sehr wohl als Methode eines Objekts aufgerufen werden. Es gibt in Ruby nämlich ein allgegenwärtiges Objekt namens `self`. Das Objekt `self` repräsentiert stets die Umgebung, in der sich der `self`-benutzende Code gerade befindet.

Nehmen wir ein frisch gestartetes Interactive Ruby an. Sobald der Prompt nervös blinkt und Ihre Eingaben erwartet, existiert dieses `self`-Objekt als Instanz der Klasse `Object`. Deklarieren Sie dann eine Methode, so gehört diese automatisch diesem `self`-

Objekt an. Nutzen Sie eine Variable, so gehört auch die dem `self`-Objekt an. Und gleichzeitig erweitert sie die Klasse `Object`, da `self` eine Instanz der Klasse `Object` ist. Ruby nennt diese Umgebung selbst `main`. Sie wird oft auch als Top-Level bezeichnet.

Außerhalb von `main`, also beispielsweise in einer Klassendeklaration, steht `self` für ein (zukünftiges) Objekt dieser Klasse, auf das Methoden der Klasse dank `self` zugreifen können. Das schauen wir uns gleich noch viel genauer an. Vorerst testen Sie doch einmal:

```
self.class # => Object
self.byte_to_kbyte(10240) # => 10
self.byte_to_kbyte(46080.0) # => 45
```

Sie haben längst gesehen, dass Sie sich das `self` in der `main`-Umgebung getrost sparen können. Ruby weiß mit diesem Quelltextgeiz in der Top-Level-Umgebung umzugehen und versucht zunächst alle Anweisungen als auf `self` bezogen zu interpretieren.

Bestehende Klassen verändern

Vielleicht gefällt Ihnen ja die Idee, an bestehendem Ruby-Rüstzeug ergänzend tätig zu werden, so wie Sie es eben, wenngleich unwissend, mit `byte_to_kbyte` und der Klasse `Object` getan haben? Dann machen wir doch damit einfach mal weiter. Lassen Sie uns das ehrenwerte Ziel verfolgen, die Methode `byte_to_kbyte` zu optimieren. Bislang müssen Sie die Methode aufrufen und ihr in einem Parameter den umzurechnenden Wert mitteilen. Das ist absolut Ruby-untypisch und eher etwas für all die Programmiersprachen von gestern. Viel stimmiger wäre eine Variante, bei der der umzurechnende Wert Empfänger ist, als dessen Methode `byte_to_kbyte` aufgerufen wird. Also etwa so: `10240.byte_to_kbyte`.

Um das zu realisieren, müssen wir die Klasse `Fixnum` um die Methode `byte_to_kbyte` ergänzen. Nichts leichter als das, zumindest in Ruby. Lediglich zwei kleine Stückchen Vorwissen benötigen Sie dazu.

Offene Klassen

Rubys Klassen sind *offen*. Das heißt, sie können jederzeit an nahezu jedem Ort erweitert oder verändert werden. Einfach so. In anderen Programmiersprachen geht das meist nicht. Da werden Klassen einmal an einer dafür vorgesehenen Stelle definiert und treten dann in einen steingleichen Zustand über.

Die Möglichkeit der einfachen, dynamischen Veränderbarkeit von Klassen ist besonders brauchbar, wenn Sie bestehende Klassen manipulieren möchten. Es genügt, wenn Sie eine neue Klassendefinition programmieren, die nur das enthält, was neu oder anders sein soll. Eventuell bestehende gleichnamige Methoden beispielsweise werden dabei überschrieben. Die jeweils letzte Version, die der Ruby-Interpreter durchlaufen hat, gilt.

Die Wirkungen Ihrer Änderungen an einer Klasse sind dann sogar auf Objekte anwendbar, die bereits vor Ihrem Eingreifen bestanden und mit einer älteren Variante der Klasse erzeugt wurden.

Sie sehen, Ruby versucht Sie nicht mit irgendwelchen Vorschriften zu gängeln, sondern erhöht den Programmierkomfort enorm. Dennoch sollten Sie natürlich darauf achten, Klassenmanipulationen nicht wahllos im Quelltext zu platzieren. Ein bisschen Ordnung sollten Sie in Ihrem eigenen Interesse halten.

Also: Sie können eine Klasse verändern und ergänzen, indem Sie eine neue Klassendefinition verfassen. Aber wie wird so eine Klasse eigentlich definiert?

Klassendefinitionen

Die Grundstruktur einer Klassendefinition ist denkbar einfach und hat eine große Ähnlichkeit mit der von Methoden. Auch hier gibt es einen *Kopf*, der den Namen der Klasse aufnimmt. Klassennamen müssen zwingend mit einem Großbuchstaben beginnen, dem kleine Buchstaben, Zahlen und Unterstriche folgen können. Der Kopf der Klassendefinition beansprucht eine eigene Zeile.

Es schließt der *Klassenkörper* an, der hauptsächlich Methoden beherbergt. Hier legen Sie die Funktionalität der Klasse fest. In der letzten Zeile einer Klassendefinition folgt das abschließende Schlüsselwort `end`. So könnte also eine Klasse aussehen:

```
class Klasse
  # Klassenkörper
  def methode1(parameter1, parameter2)
    # Methodenkörper methode1
  end # Ende methode1
  def methode2(parameter1)
    # Methodenkörper methode2
  end # Ende methode2
  #
end # Ende Klassendefinition
```

Mit diesem Wissen ausgestattet, können Sie sich jetzt gezielt der Klasse Fixnum widmen, welche um `byte_to_kbyte` erweitert werden soll.

Achtung, Fixnum! Jetzt komm' ich!

Die Frage lautet nun: Wie muss die Klassendefinition aussehen, die die Klasse Fixnum ergänzt? Der Klassenname ist klar: `Fixnum`. Die Methode `byte_to_kbyte` wird der einzige Inhalt des Klassenkörpers sein. Allerdings wird die Methode keinen Parameter haben. Der Wert, der umgerechnet werden soll, wird schließlich durch das Objekt selbst geliefert. Und genau an dieser Stelle kommt `self` wieder ins Spiel. Allerdings in einem anderen Kontext: Diesmal ist der Lebensraum von `self` nicht `main`, sondern die Klasse `Fixnum`. Konkret zeigt `self` auf den Empfänger der Methode, also auf ein `Fixnum`-Objekt.

Beispiel 2-103: Fixnum um byte_to_kbyte erweitert

```
class Fixnum
  def byte_to_kbyte
    self.to_f / 1024
  end
end
```

Das war es schon. Probieren Sie die erweiterte Fixnum-Klasse doch mal aus. Sie werden begeistert sein.

Beispiel 2-104: Fixnum#byte_to_kbyte in Aktion

```
1024.byte_to_kilobyte # => 1.0
10240.byte_to_kilobyte.round # => 10
102400.byte_to_kilobyte.to_s # => "100.0"
```

Das war einfach, aber noch nicht so ganz perfekt. Denn schließlich sollen ja beispielsweise auch Float- oder Bignum-Zahlen umgerechnet werden können. Da unsere Anpassung bislang aber ausschließlich der Fixnum-Klasse gilt, sind gebrochene und sehr große Zahlen außen vor.

Besser: Numeric erweitern

Alles kein Problem. Man könnte der Float- und der Bignum-Klasse auch noch je eine byte_to_kbyte-Methode anhängen. Ein bisschen umständlich, zumal die Implementierung sich von byte_to_kbyte für Fixnum-, Bignum- und Float-Zahlen nicht unterscheiden würde.

Lassen Sie uns einen anderen Weg gehen. Und der soll über die Klasse *Numeric* führen. Von Numeric *erben* sowohl Fixnum, Bignum als auch Float einige Methoden; die Klasse Numeric ist ein *Vorfahre* der genannten Klassen. Das bedeutet: Jede Methode, die Sie Numeric hinzufügen, steht automatisch den drei Genannten zur Verfügung.



Welche Vorfahren eine Klasse besitzt und von welchen Klassen sie erbtt, erfahren Sie, wenn Sie den fraglichen Klassennamen über die Methode ancestors fragen, zum Beispiel Bignum.ancestors.

Beispiel 2-105: Fixnum, Bignum und Float werden um byte_to_kbyte erweitert

```
class Numeric
  def byte_to_kbyte
    self.to_f / 1024
  end
end
```

Und nun können Sie sich verschiedenste Zahlen schnappen – byte_to_kbyte wird stets zu Ihrer Verfügung stehen und die Umrechnung vornehmen.



Wichtig: Fixnum hat noch immer seine eigene `byte_to_kbyte`-Methode! Zwar erbt Fixnum die Methode `byte_to_kbyte` von Numeric, doch misst Ruby der klassenspezifischen Methode von Fixnum höhere Bedeutung bei. Der Nachwuchs weiß eben auch in der irrealen Welt der Programmierung immer alles besser.

Nun aber genug von Spielereien mit fremden Klassen. Die Zeit ist reif für etwas Eigenes. Die Grundstruktur einer Klasse kennen Sie nun schon. Für die Erstellung eigener Klassen fehlen Ihnen nur noch ein paar Dinge, um die es jetzt gehen soll.

Reisen mit der 1. Klasse

Anhand eines Beispielszenarios möchte ich Ihnen zeigen, wie Sie eigene Klassen erstellen können. Ein virtueller Zug soll uns an dieses wichtige Ziel bringen. Dafür greifen wir das Beispiel des ersten Kapitels noch einmal auf, dem Sie dort im Zusammenhang mit der Vorstellung des objektorientierten Programmieransatzes begegnet sind.

Sie werden gleich etwas programmieren, das es Ihnen ermöglicht, Zugführer zu spielen. Sie sitzen am Ende des folgenden Beispiels quasi im Führerstand einer Lok, die allerdings mit der eben genannten nicht viel zu tun hat. Es soll ein richtig moderner Flitzer werden, mit hoher Geschwindigkeit, digitalem Zugzielanzeiger und Bordcomputer. Alles Dinge, die das dampfende Ungeheuer, von dem eben die Rede war, allem Anschein nach nicht zu bieten hat.

Da ein Zug meist aus Lok und mehreren Waggons besteht, lohnt sich die Verwendung zweier Klassen, die Lok und Waggon separat betrachten. Das ist auch deshalb sinnvoll, weil eben mehrere Waggons mit der Lok zusammen durch die Gegend rasen sollen. Und schließlich sind Klassen dazu da, Vorbild für die Erzeugung gleicher Objekte zu sein.

Ein Blick in den Fahrplan

Bevor Sie auch nur ein Zeichen Quelltext schreiben, der zu einer Klassendefinition gehört, sollten Sie sich stets Gedanken darüber machen, wie denn diese Klasse aussehen soll. Das gilt für alle Programmiersprachen mit OOP. Nur wenn Sie gut vorüberlegen, sind Sie mit der objektorientierten Programmierung auch danach überlegen.

Stellen Sie sich einfach ein paar Fragen: Welche Art von Objekten möchten Sie in Ihrem Programm nutzen und wie müssen sie beschaffen sein? Welche Funktionalitäten müssen in die Klasse rein, welche Methoden ermöglichen die Bedienung eines Objekts, wie sehen also die Schnittstellen aus? Und: Muss möglicherweise die Interaktion mit Instanzen anderer Klassen berücksichtigt werden?

Daher lautet mein Rat für die Planungsphase objektorientierter Programmierung: Vergessen Sie einfach mal für eine Weile, dass es Computer gibt und erfreuen Sie sich an den genialen Erfindungen *Papier* und *Bleistift*. Sie werden preiswert im Fachhandel zum Kauf angeboten.

Schreiben Sie auf, was ein Zug, also eine Lok, ein einzelner Waggon können muss und was sie auszeichnet. Das muss kein Essay werden, eine einfache Skizze oder eine Liste von Fähigkeiten und Eigenschaften reichen völlig. Die objektorientierte Programmierung fußt in ihrer Grundkonzeption bekanntlich auf der menschlichen Denke. Also nutzen Sie diesen Vorteil. Selbst wenn Sie noch keine rechte Vorstellung von OOP haben, ihr Kopf funktioniert doch, oder?

Die Lok. Unsere Lok soll Folgendes können: Sie soll beschleunigen und bremsen, die Türen der angekuppelten Waggons öffnen und schließen und Waggons an- und abkuppeln können. Außerdem soll sie uns ihre momentane Geschwindigkeit anzeigen, beim Beschleunigen eine Höchstgeschwindigkeit nicht überschreiten und keine neuen Waggons an den Haken nehmen, wenn ihre maximale Zugkraft damit überschritten würde. Und über den Status der Türen, also ob offen oder geschlossen, soll sie auch noch Auskunft geben. Schließlich reagieren einige Fahrgäste bisweilen etwas mürrisch, wenn sich der Zug in Bewegung setzt, während sie gerade ihren berstenden Rollkoffer in die Bahn wuchten.

Um das gänzlich auszuschließen, soll der Zug auch erst dann beschleunigen können, wenn alle Türen geschlossen sind. Außerdem sollen sie auch nicht aufgehen, wenn der Zug gerade durch die Gegend rollt. Auch das könnte zu Irritationen bei den Fahrgästen führen. Ach ja, ein schicker Zugzielanzeiger muss auch noch sein. Der sollte bereit sein, auch während der Fahrt ein neues Ziel anzuzeigen und das aktuell eingestellte auf Wunsch auf dem Bordcomputer auszugeben. Denn so ein Lokführer kann ja mal vergessen, wo er hinmuss. Außerdem soll der Bordcomputer auch über die Anzahl der Sitze des Zugverbandes Auskunft geben.

Lassen Sie uns aus diesem Lok-Wunschzettel Fähigkeiten und Eigenschaften ableiten, die wir in der Lok-Klasse umsetzen werden. Dabei müssen wir auch überlegen, welche Eigenschaften möglicherweise nur klassenintern benutzt und welche nach außen zugänglich gemacht werden und als Schnittstelle dienen sollen.

Aus Fähigkeiten werden später Methoden der Klasse. Und somit werden wir Methoden zum Bremsen, Beschleunigen, Öffnen beziehungsweise Schließen der Türen und zum An- und Abkuppeln der Waggons implementieren.

Die hier noch als Eigenschaften titulierten Werte werden später teilweise so genannte *Attributes* der Klasse sein. Das sind Variablen, die innerhalb eines Objekts, das auf dieser Klasse basiert, Werte speichern können. Jedes Objekt hat seine eigenen, die nur für dieses eine Objekt zuständig und gültig sind. Die Eigenschaften der Lok sind beispielsweise Geschwindigkeit, Höchstgeschwindigkeit (beides in km/h),

Anzahl der Sitzplätze des ganzen Zuges, das Gewicht der Waggons (in Tonnen), der Status der Türen des Zuges und das Ziel der Fahrt. Außerdem muss eine Möglichkeit gefunden werden, die angekoppelten Wagen zu speichern.

Ein Lok-Objekt soll von diesen Eigenschaften aber nur die momentane Geschwindigkeit, die Sitzplatzanzahl, den Türenstatus und das Fahrtziel auf Verlangen preisgeben. Die anderen Werte interessieren nur intern und bleiben den Methoden der gekapselten Lok vorbehalten. Sie gehören nicht zur Schnittstelle. Die Höchstgeschwindigkeit ist beispielsweise nur für die Beschleunigen-Methode relevant.

Auf die Eigenschaft Zugziel soll auch schreibend zugegriffen werden. Die Sitzplatzanzahl, der Türstatus und das Gewicht der Anhänger werden automatisch auf Basis der angekoppelten Waggons ermittelt. Eine Manipulation durch Schreibzugriff auf diese Daten ist daher ausgeschlossen.

Die Waggons. Bleiben wir gleich bei den Waggons und den Erfordernissen der Waggon-Klasse. Die leiten sich bereits teilweise aus dem Informationshunger des Lok-Bordcomputers ab. Ein Waggon-Objekt muss die Sitzplatzanzahl, das Gewicht (in Tonnen) und den Zustand der Türen speichern und via Lesezugriff ausgeben. Schließlich sind diese Infos für die Lok interessant. Während ein Lok-Objekt Informationen beispielsweise über alle Türen eines Zuges erhält, muss sich ein Waggon-Objekt nur um die Türen eines bestimmten Waggons kümmern. Eine Möglichkeit zum Öffnen oder Schließen der Türen eines Waggons muss natürlich auch implementiert werden. Eine entsprechende Funktionalität wird später durch die Lok ausgelöst.

Was beim Erzeugen eines Objekts passiert

Sie werden mir sicher zustimmen, dass die Funktionalität der Waggons wesentlich überschaubarer ist als die der Lok. Doch nicht nur deshalb lohnt es sich, die Implementierung unseres Zuges mit der Waggon-Klasse zu starten. Viele Funktionen des Lok-Bordcomputers basieren auf Informationen, die ein Waggon ausgibt. Ohne die Waggon-Klasse konkret zu kennen, gestaltet sich die Implementierung der Lok aber etwas schwieriger, wenngleich sie machbar wäre.

Die vor ein paar Seiten eingeführte Grundstruktur einer Klasse gilt natürlich auch für Ihre Eigenkreationen. Sie beginnen also mit einem Klassennamen und definieren nach und nach die Methoden, die für die Funktionalität Ihrer Klasse benötigt werden, innerhalb des Klassenkörpers, der mit einem *end* abschließt. Denken Sie dabei an die Konventionen für Klassennamen.

```
class Waggon  
end
```

Damit hätten Sie bereits eine vollständige wenn auch eher armselige Klasse implementiert. Probieren Sie trotzdem aus, ein Objekt der Klasse Waggon zu erstellen. Da die Erstellung von Objekten durch Literale den eingebauten Ruby-Klassen vorbehalten ist, behelfen wir uns mit der Methode new, die grundsätzlich stets zum Erzeugen von Objekten genutzt werden kann. Rufen Sie new parameterlos mit dem Klassennamen als Empfänger auf.

```
testwaggon = Waggon.new
```

Und schon steht testwaggon auf den Schienen. An dieser Stelle folgen ein paar Hintergrundinfos für Wissbegierige: Klassen sind selbst Objekte, namentlich Instanzen der Klasse Class. Die Klasse Class wiederum enthält die Methode new, welche eine neue Instanz des Objekts und somit der Klasse erzeugt. In Class sind bereits diverse Methoden enthalten, die an Waggon weitergegeben wurden. Einige von denen kennen Sie bereits, allerdings haben Sie sie bislang nur auf bestehende Klassen angewandt. Aber vielleicht vermittelt es Ihnen ein Gefühl von persönlicher Größe, wenn Sie sehen, dass object_id und class nun bereit sind, Auskünfte über Ihre selbst erstellte Klasse zu erteilen?

```
testwaggon.class # => Waggon  
testwaggon.object_id # => 22543740
```

Sie haben beispielsweise bei der Erzeugung eines Array-Objekts gesehen, dass es klassenspezifische Varianten von new gibt, bei deren Aufruf Sie noch Parameter angeben können. Diese Parameter dienen stets dazu, das neue Objekt mit Startwerten zu belegen, damit es direkt nach der Erzeugung nicht ganz so nackt ist.

Das könnten wir mit unseren Waggons auch machen. Schließlich gibt es Attribute, die gleich beim Erzeugen eines Waggons mit Werten gefüllt werden sollten, weil Sie den Waggon charakterisieren: Sitzplatzanzahl, Gewicht und Türstatus. Allerdings denke ich, dass man ruhig davon ausgehen kann, dass ein fabrikneuer Waggon mit geschlossenen Türen ausgeliefert wird. Bleiben noch zwei Attribute, die für die Parametrisierung der new-Methode in Frage kommen würden.

Eine klassenspezifische Variante der new-Klasse erreichen Sie durch die Verwendung einer initialize-Methode mit den entsprechenden Parametern. Sie wird automatisch dann aufgerufen, wenn mit new ein Objekt der Klasse erzeugt wird. Die Methode initialize wird auch für weitere Aktionen, die das Erzeugen eines Objekts betreffen, gern genutzt. Beispielsweise können Sie in ihr alle Attribute einer Klasse erzeugen, auch ohne ihnen einen Wert zuzuweisen. Das ist kein Muss, denn ein Attribut entsteht auch später automatisch dann, wenn es erstmals einen Wert erhält. Dennoch gehört es zum guten Stil, die Initialisierung der Attribute an einem zentralen Ort gleich zu Beginn der Lebenszeit eines Objekts vorzunehmen. Damit werden auch Fehler vermieden, die entstehen, wenn ein Objekt auf ein Attribut zugreift, dass noch nicht durch eine Wertzuweisung erzeugt wurde.

```

class Waggon
  def initialize(sitze, gewicht)
    @sitze = sitze
    @gewicht = gewicht
    @tueren_offen = false
  end
end

```

Wie Sie sich vielleicht denken können, sind die Variablen mit einem @-Zeichen im Bezeichner Attribute. Sie erhalten durch `initialize` einen Wert; `@sitze` und `@gewicht` sogar einen von außen. Das Attribut `@tueren_offen` hat mit `false` einen Standardwert erhalten, der auf geschlossene Türen hinweisen soll. Sind die Türen offen, enthält `@tueren_offen` wenig überraschend `true`.

Vielleicht sei an dieser Stelle auf die Gültigkeitsbereiche von Methodenparametern und Attributen hingewiesen. Daraus erklären sich Zuweisungen wie `@sitze = sitze`. Attribute einer Klasse gelten *klassenweit* und können von allen Methoden der Klasse standardmäßig gelesen und geschrieben werden. Parameter einer Methode sind nur innerhalb der Methode sichtbar und gültig. Damit sie klassenweit genutzt werden können, müssen sie Attributen übergeben werden.

Attribute heißen auch *Instanzvariablen*, weil sie nur für die Instanz einer Klasse gelten, in der sie erzeugt und ihren Wert erhalten haben. Die Instanzvariable in Objekt A kann einen Wert beinhalten, der unabhängig von der gleichen Instanzvariable des Objekts B ist. Doch es gibt auch *Klassenattribute*. Sie beginnen mit zwei @-Zeichen. Erhält das beispielhafte Klassenattribut `@@count` also in Objekt A einen Wert, so erhält ihn auch Objekt B – und umgekehrt.

Neben Klassenattributen gibt es auch *Klassenmethoden*. Sie haben schon einige Male eine benutzt, zum Beispiel bei `Array.new`. Klassenmethoden sind die Methoden der Klasse, die auch ohne Instanzbildung genutzt werden können. Die Klasse, die diese Methode enthält, fungiert hierbei als Empfänger. Mit `new` kennen Sie eine typische Klassenmethode. Schließlich erzeugt erst `new` ein Objekt nach Vorbild einer Klasse.

Um eine Klassenmethode selbst zu implementieren, setzen Sie dem Methodenbezeichner noch ein `self.` voran. Die Schreiben eigener Klassenmethoden empfiehlt sich beispielsweise dann, wenn Sie mit dem Erzeugen eines Objekts auch gleich eine oder mehrere Aktionen durchführen möchten. Doch zurück zu unserem Waggon.

Erzeugen Sie doch jetzt einfach mal einen neuen `testwaggon`, diesmal mit Angaben zu Sitzanzahl und Gewicht. Ein Aufruf von `new` ohne Parameter ist nun nicht mehr statthaft und wird mit einer Fehlermeldung bestraft. Der folgende Ausdruck erzeugt in `testwaggon` einen Wagen mit 80 Sitzen und 50 Tonnen Gesamtgewicht, inklusive 80 Normpassagieren, die jeweils einen 75 kg schweren Körper und zusätzlich 15 kg in Koffern und Taschen zum Gesamtgewicht beitragen.

```
testwaggon = Waggon.new(80, 50)
```

Nutzen Sie Interactive Ruby, dann werden Sie beim Betätigen der *Enter*-Taste mit einer Information belohnt, die alle Attribute des Objekts *testwaggon* enthält. Sie können diese Daten stets auch dann erhalten, wenn Sie die Methode *inspect* auf einen beliebigen Empfänger anwenden.

Getter, Setter, Akzessoren

Natürlich gelangen Sie mit *inspect* nicht wirklich auf sinnvolle Weise an die Daten, die innerhalb des Objekts stecken. Die Methode existiert auch nur für Sie als Entwickler. Doch wie soll man sonst von außen an den Inhalt der Attribute gelangen?

Attribute sind von außen nicht sichtbar. Um das zu ändern, müssen Sie für jedes Attribut, welches diese Regel durchbrechen soll, eine *Getter*-Methode für den Lese- und gegebenenfalls eine *Setter*-Methode für den Schreibzugriff implementieren. *Getter*- und *Setter*-Methoden werden fachsprachlich gern *Akzessoren* genannt.

In der Waggon-Klasse benötigen wir für jedes Attribut eine *Getter*-Methode, da die Lok alle drei Werte benötigt und sonst nicht an die Werte gelangen würde. Der Name einer *Getter*-Methode muss nicht zwangsläufig mit dem des (um das @-Zeichen gekürzten) Attributs übereinstimmen, dessen Inhalt die Methode ausgibt. Dennoch ist es sinnvoll, den Bezeichner zu übernehmen. So wird eine *Getter*-Methode auch wirklich schnell als solche erkannt.

Denken Sie daran, dass folgender Codeschnipsel die bestehende Waggon-Klasse ergänzt und sie nicht ersetzt. Die *initialize*-Methode ist also weiterhin Bestandteil der Klasse.

```
class Waggon
  def sitze
    @sitze
  end

  def gewicht
    @gewicht
  end

  def tueren_offen
    @tueren_offen
  end
end
```

Möchten Sie diese Änderungen testen, brauchen Sie keinen neuen *testwaggon* zu erzeugen. Der bereits bestehende enthält automatisch die drei neuen *Getter*-Methoden:

```
testwaggon.sitze # => 80
testwaggon.gewicht # => 50
testwaggon.tueren_offen # => false
```

Jetzt müssen wir noch im Interesse der Fahrgäste dafür sorgen, dass die Türen auch geöffnet werden können. In unserer kleinen Simulation eines Zuges soll es genügen, wenn wir dazu einfach eine Setter-Methode coden, die `@tueren_offen` einen anderen Wert zuweist.

Eine Setter-Methode zeichnet sich dadurch aus, dass der Methodenname mit einem Gleichheitszeichen endet, ohne Leerzeichen dazwischen, und einen Parameter zur Wertübergabe benötigt. Bei der Wahl des Methodennamens sollten Sie sich ein weiteres Mal an dem zu verändernden Attribut orientieren.

```
class Waggon
  def tueren_offen= (wert)
    @tueren_offen = wert
  end
end
```

Bei der Verwendung von Zuweisungsmethoden wie `tueren_offen=` sollten Sie ein Leerzeichen zwischen Methodenname und Gleichheitszeichen setzen. Das entspricht den Schreibkonventionen für Ruby-Code, und Ruby kommt problemlos damit klar, wenngleich der ursprüngliche Methodenname eigentlich damit zerstört wird. Aber sehen Sie selbst:

```
testwaggon.tueren_offen # => false
testwaggon.tueren_offen = true
testwaggon.tueren_offen # => true
```

Damit ist unsere Waggon-Klasse fertig. Vielleicht möchten Sie sie noch einmal in ihrer kompletten Schönheit bewundern? Bitte sehr!

Beispiel 2-106: Waggonbau mit Schablone: Die Klasse Waggon

```
class Waggon
  def initialize(sitze, gewicht)
    @sitze = sitze
    @gewicht = gewicht
    @tueren_offen = false
  end

  def sitze
    @sitze
  end

  def gewicht
    @gewicht
  end

  def tueren_offen
    @tueren_offen
  end

  def tueren_offen= (wert)
```

Beispiel 2-106: Waggonbau mit Schablone: Die Klasse Waggon (Fortsetzung)

```
    @tueren_offen = wert  
end  
end
```

Schon ganz nett. Aber Ruby wäre nicht Ruby, wenn es da nicht noch eine kleine Vereinfachung gäbe.

attr_aktivere Getter, Setter, Akzessoren

Im Code unserer Waggon-Klasse wird der größte Teil damit gefüllt, Attribute außerhalb der Objektgrenzen lesbar und in einem Fall sogar beschreibbar zu machen. In den vier dafür nötigen Methoden passiert gar nichts außer der Rückgabe des Werts eines Attributs oder der Zuweisung eines Werts an ein Attribut.

Für solche einfachen, häufig vorkommenden Fälle hält Ruby kleine Helper bereit. Arbeiten Sie mit diesen Helfern, bleiben Ihnen Getter- und Setter-Methoden, wie sie in der Klasse Waggon benutzt werden, zukünftig erspart. Zumindest müssen Sie die nicht mehr selbst schreiben, sondern nur noch eine entsprechende Anweisung erteilen. Der Aufwand schmilzt enorm. Vergleichen Sie doch einmal. Hier rollt die kurze Variante der Klasse Waggon heran:

Beispiel 2-107: Waggon-Klasse verkürzt

```
class Waggon  
  attr_reader('sitze', 'gewicht')  
  attr_accessor('tueren_offen')  
  def initialize(sitze, gewicht)  
    @sitze = sitze  
    @gewicht = gewicht  
    @tueren_offen = false  
  end  
end
```

Das Prinzip dieser Helfer ist ganz einfach: Sie beginnen alle mit dem Kürzel attr, was für *Attribut* steht. Anschließend, getrennt durch einen Unterstrich, folgen die drei Varianten reader, writer und accessor.

Mit attr_reader und den von @-Zeichen befreiten Attributbezeichnern, die als String übergeben werden, weisen Sie Ruby an, dass bestimmte Attribute von außerhalb eines Objekts lesbar sein sollen. Notieren Sie maximal zehn Attributbezeichner durch Kommas getrennt als Parameter der Methode. Ruby kümmert sich dann selbstständig um die entsprechenden Getter-Methoden.



Es ist auch möglich und sehr beliebt, die Attributbezeichner als Symbol zu übergeben statt als String. Auch hier entfällt das vorangestellte @-Zeichen.

Für Attribute, die nach außen nur mit Schreibzugriff versehen werden sollen, können Sie nach gleichen Regeln auf attr_writer zurückgreifen. Attribute, die von außen sowohl gelesen als auch beschrieben werden sollen, können in attr_accessor untergebracht werden.



Bitte beachten Sie, dass Sie attr_reader, attr_writer und attr_accessor natürlich nur dann sinnvoll nutzen können, wenn Sie auf das Implementieren reiner Akzessoren verzichten möchten. Sollte eine Setter-Methode z. B. noch einen Kontrollmechanismus beinhalten, kommen Sie um das Schreiben der kompletten Methode natürlich nicht herum.

Alle Akzessoren-Hilfsmethoden sollten Sie außerhalb einer Methode der Klasse notieren. Idealerweise recht weit oben.

Waggontext

Wissen Sie, was unserer Klasse noch fehlt? Eine to_s-Methode. Fast jede Standardklasse in Ruby hat diese Methode, warum also nicht auch Waggon?

Eine Implementation von to_s sollte jeder Klasse gegönnt werden, und sei es nur aus Gründen, die in einem komfortableren Debugging liegen. Nehmen wir doch einmal an, Sie testen Ihre Anwendung gerade und stellen fest, dass in den Tiefen Ihres Codes ein dicker Fehler steckt. Um ganz schnell hinter die internen Vorgänge Ihres Programms zu kommen, reicht manchmal schon das einfache Analysieren eines Objektinhalts mit `puts`.

```
puts(testwaggon) # <Waggon:0x2afdb4>
```

Ruby zeigt sich an dieser Stelle etwas hilflos und gibt die relativ unbedeutende Auskunft über Klasse und ID des Objekts `testwaggon` aus. Wenn Sie aber Ihrer Klasse eine to_s-Methode beifügen, können Sie bestimmen, was in dieser oder anderen Situationen, in denen das Objekt zu einem String konvertiert werden muss, ausgegeben werden soll.

```
class Waggon
  def to_s
    "Personenwagen mit #{@sitze} Sitzen. Gewicht: #{@gewicht} Tonnen."
  end
end
```

Zugegeben, eine bahnbrechende to_s-Methode ist das natürlich nicht. Aber sie gibt immerhin Auskunft über die Beschaffenheit dessen, was ein Objekt der Klasse Waggon repräsentiert.

```
testwaggon.to_s # => "Personenwagen mit 80 Sitzen. Gewicht: 50 Tonnen."
```

Dank unserer Klasse Waggon können wir nun Waggons ohne Ende erzeugen. Aber so richtig Sinn macht das natürlich nicht. Eine Lok muss her, die die vielen Wägelchen über die Gleise zieht.

class Lok

Auch für die Lok-Klasse ist eine initialize-Methode sehr sinnvoll. Die Werte für die maximale Last, die die Lok ziehen kann, und ihre Höchstgeschwindigkeit werden mit der Erzeugung übergeben.

Damit stehen schon zwei Attribute fest, die die Klasse Lok haben soll: @hgeschw und @hgewicht. Und was braucht sie noch? Die aktuelle Geschwindigkeit beispielsweise. Die sollte beim Herstellungsprozess wünschenswerterweise 0 sein – also initialisieren wir das Attribut @geschw entsprechend.

Wir benötigen noch einen Datenspeicher für eine unbestimmte Menge an Objekten der Klasse Waggon. Ein hervorragendes Einsatzgebiet für ein Array. Also erzeugen wir in der initialize-Methode ein entsprechendes, leeres Array-Objekt und weisen es dem Attribut @waggons zu. Bleibt noch das Attribut, das den Text des Zugzielanzeigers beinhalten soll, @ziel soll es heißen. Klar, ein String-Objekt muss her, vorerst leer.

In den Vorüberlegungen zum Lok-Objekt war von Eigenschaften des Zuges die Rede, die das Gesamtgewicht der Waggons, ihre Türen und die gesamte Sitzplatzanzahl betreffen. Ich schlage vor, diese Eigenschaften als Methode zu implementieren. Das hat den Vorteil, dass die entsprechenden Infos erst in dem Moment, in dem eine dieser Eigenschaften abgefragt wird, gesammelt werden können. So sind Sie stets bestens informiert über ihren Zug. Also legen wir direkte Akzessoren für @geschw (lesend) und @ziel (lesend und schreibend) an.

Beispiel 2-108: Die Basis der Klasse Lok

```
class Lok
  def initialize(hgewicht, hgeschw)
    @hgewicht = hgewicht
    @hgeschw = hgeschw
    @geschw = 0
    @waggons = Array.new
    @ziel = ''
  end

  attr_reader('geschw')
  attr_accessor('ziel')
end
```

Damit wäre alles dafür getan, dass eine Lok mit ihren charakteristischen Daten erzeugt werden kann. Noch besser: Das Wichtigste an der Lok funktioniert schon jetzt – der Zugzielanzeiger.

```
lok = Lok.new(800, 160)
lok.ziel = 'Hamburg'
lok.ziel # => "Hamburg"
```

Die Methoden für das Beschleunigen und Bremsen sind schnell geschrieben. Das liegt ehrlich gesagt daran, dass diese Funktionen doch stark vereinfacht implementiert werden sollen. Aber am Ende des Buches sollen Sie ja auch keine ICEs bauen, sondern Ruby auf die Schienen stellen können. Daher nehmen wir einfach an, dass die Lok bei jedem Aufruf von beschleunigen um 10 km/h schneller wird, bei bremsen um 25 km/h langsamer. Die beiden Methoden müssen zudem darauf achten, dass die Geschwindigkeit nie unter 0 sinkt oder über die Höchstgeschwindigkeit steigt. Außerdem sollen sie die neue Geschwindigkeit zurückgeben.

Beispiel 2-109: Lok#beschleunigen und Lok#bremsen

```
class Lok
  def beschleunigen
    @geschw += 10
    @geschw = @geschw if @geschw > @hgeschw
    @geschw # Rückgabewert
  end

  def bremsen
    @geschw -= 25
    @geschw = 0 if @geschw < 0
    @geschw # Rückgabewert
  end
end
```

Dann lassen Sie uns doch mal den Motor anwerfen und etwas beschleunigen, um gleich danach zu bremsen.

```
lok.geschw # => 0
lok.beschleunigen # => 10
lok.beschleunigen # => 20
lok.geschw # => 20
lok.bremsen # => 0
lok.geschw # => 0
```

Wie Sie sicherlich bemerkt haben und unter Anmahnungen von grundlegenden Sicherheitsmaßnahmen zurecht anprangern, berücksichtigt beschleunigen noch nicht, ob auch alle Wagentüren geschlossen sind. Darauf kommen wir noch zurück, sobald Rollmaterial anhängig ist. Und um die Waggonen kümmern wir uns jetzt mit den Methoden ankuppeln und abkuppeln.

Die Methode ankuppeln soll zunächst prüfen, ob im Parameter überhaupt ein Waggon-Objekt übergeben wurde. Dazu wird die Klasse des Parameter-Objekts geprüft. Anschließend wird mittels Array#index gecheckt, ob denn dieser Waggon nicht schon längst am Haken hängt.

Eine weitere Bedingung muss erfüllt sein: Müsste die Lok über ihre Schmerzgrenze gehen, wenn der Wagen angekuppelt würde? Sollte keine solche Gefahr bestehen, hängt der Waggon am Zug. In diesem Fall wird `Waggon#to_s` mit einem entsprechenden Hinweis zurückgegeben. Sollte das Ankuppeln nicht erfolgreich gewesen sein, wird eine informative, dies begründende Meldung zurückgegeben.

Beispiel 2-110: Lok#ankuppeln prüft und kuppelt

```
class Lok
  def ankuppeln(waggon)
    # Ist waggon ein Objekt der Waggon-Klasse?
    return 'Waggon hat keine Verkehrserlaubnis.' unless waggon.class == Waggon

    # Waggon bereits angekuppelt?
    return 'Waggon ist bereits angekuppelt.' if @waggons.include?(waggon)

    # Wird Höchstgewicht überschritten?
    if (waggon.gewicht + aktuelle_zuglast) < @hgewicht
      @waggons.push(waggon)
      return "Angekuppelt: #{waggon.to_s}"
    else
      return 'Waggon zu schwer.'
    end
  end
end
```

Wie Sie sehen, nutzt die Methode `ankuppeln` die Eigenart von `return` für sich aus, den Fortgang der Methode zu beenden. Das ist zugegebenermaßen nicht unbedingt die sauberste, aber die einfachste Variante, den Methodenkörper zu gestalten. Alternativ könnten Sie hier auch ohne `return`, aber mit mehrfach verschachtelten `if-` oder `unless`-Konstrukten arbeiten.

Sichtbarkeitsbereiche von Methoden einer Klasse

Sicherlich ist Ihnen aufgefallen, dass mit `aktuelle_zuglast` in `Lok#ankuppeln` auf etwas zurückgegriffen wird, was bislang in der Klassendefinition noch nicht aufgetaucht ist. Es handelt sich bei `aktuelle_zuglast` um eine kleine Hilfsmethode, die die aktuelle Last am Lokaugen berechnet.

Beispiel 2-111: Wie viele Tonnen hängen schon an der Lok?

```
class Lok
  def aktuelle_zuglast
    zl = 0
    @waggons.each { |w|
      zl += w.gewicht
    }
    return zl
  end
end
```

Die Funktionalität ist recht simpel: Alle angekoppelten Waggons werden mit each nach ihrem jeweiligen Gewicht befragt. Innerhalb des Blocks repräsentiert `w` einen Waggon des Anhangs. Bei jedem Blockdurchlauf wird die Hilfsvariable `z1` um das jeweilige Gewicht eines Waggons erhöht und fungiert am Ende der Methode als Rückgabewert.

Nun handelt es sich bei `aktuelle_zuglast` um eine Methode, die eigentlich nur für das Innenleben der Lok-Klasse wichtig ist und nur von ihr benutzt werden sollte. Das bedeutet, wir müssen den Zugriff von außen auf diese Methode unterbinden. Dies geschieht in Ruby, wie auch in vielen anderen Sprachen, über festgelegte *Sichtbarkeitsbereiche*. In Ruby heißen selbige wie in anderen Sprachen, haben aber eine leicht andere Bedeutung.

Grundsätzlich sind alle Methoden in Ruby-Klassen öffentlich, also von außen zugänglich. Unsere bislang in der Lok-Klasse implementierten Methoden sind beispielsweise alle `public` – bis auf `initialize`. Die Methode zum Initialisieren eines Objekts ist grundsätzlich nicht öffentlich.

Sie erinnern sich? Bei den Attributen ist das genau andersherum: Alle Attribute sind nicht öffentlich und müssen beispielsweise erst durch `attr_accessor` lesbar und beschreibbar gemacht werden.

Um eine Methode `public` zu machen, müssen Sie nichts weiter tun. Bei den Sichtbarkeiten `protected` und `private` verhält es sich erwartungsgemäß anders. Mit `protected` und `private` können Sie Methoden nach außen hin unsichtbar, also unbenutzbar machen. Innerhalb der Klasse und aller Klassen, die auf dieser Klasse basieren, ist eine als `protected` oder `private` gekennzeichnete Methode natürlich sehr wohl nutzbar.

Ist noch die Frage offen: Was unterscheidet denn `private` und `protected` voneinander? Die Antwort: `private` deklarierte Methoden können nur mit `self` als Empfänger benutzt werden. Wie Sie wissen, steht `self` innerhalb einer Klassendefinition für das Objekt, welches aus dieser Klasse gebildet werden kann. Methoden, die als `protected` gekennzeichnet sind, können auch auf externe Objekte angewendet werden. Dies kann beispielsweise dann der Fall sein, wenn die Klasse eine Methode beinhaltet, die als Parameter ein Objekt der gleichen Klasse erhält. Das kann durchaus passieren, zum Beispiel dann, wenn zwei Objekte miteinander verglichen werden sollen.

Um Methoden einem Sichtbarkeitsbereich zuzuordnen, haben Sie zwei Möglichkeiten. Die erste basiert auf der Einrichtung von entsprechenden *Abschnitten* innerhalb einer Klassendefinition. Methoden der Klasse setzen Sie dann an die entsprechende Stelle. Es müssen dabei nicht alle drei Sichtbarkeitsbereiche berücksichtigt werden.

```
class Beispielklasse
  public
    def public1
      ...
    end
```

```

def public2
  ...
end

protected
def protected1
  ...
end

private
def private1
  ...
end
def private2
  ...
end
end

```

Grundsätzlich ist zu sagen: Ein Sichtbarkeitsbereich gilt, bis ihn ein weiterer ablöst oder das Ende der Klassendefinition erreicht ist. Sie sollten Ihre Klassendefinition mit Sichtbarkeitsbereichen ähnlich strukturieren, wie Sie sie es im Schema sehen. Eine Leerzeile vor einem neuen Sichtbarkeitsbereich erleichtert das Verstehen Ihrer in Ruby-Code gegossenen Ideen. Außerdem sollten Sie jedem Sichtbarkeitsbereich nur einen Abschnitt widmen.



Rein theoretisch ist es möglich, etwa mehrere protected-Zonen mit Methoden zu füllen – doch das wird irgendwann für überflüssige Irritationen sorgen.

Ich bin Ihnen noch die angekündigte zweite Variante, mit der Sie Methoden vor äußerem Zugriff schützen, schuldig. Sie greift erst ganz am Ende einer Klassendefinition, nachdem alle Methoden implementiert sind. Hierbei können Sie public, protected und private auch optisch als das nutzen, was sie eigentlich sind: Methoden. Rufen Sie die Sichtbarkeitsbegrenzer als solche auf und listen Sie als durch Kommas getrennte Parameter die Methoden auf, deren Sichtbarkeit Sie bestimmen möchten. Geben Sie dabei die gewünschten Methodenbezeichner als String oder Symbol an. Das adaptierte Grundschema von eben:

```

class Beispielklasse
  def public1
    ...
  end

  def public2
    ...
  end

  def protected1
    ...
  end

```

```

def private1
  ...
end

def private2
  ...
end

protected(:protected1)
private('private1', 'private2')
end

```

Für welche Variante Sie sich entscheiden, bleibt natürlich Ihnen überlassen. Die weitaus gebräuchlichere und meines Erachtens auch viel übersichtlichere stellt die erste Möglichkeit dar.

Möchten Sie also aktuelle_zuglast nach außen hin unsichtbar, nach innen aber nutzbar machen, haben Sie die Wahl zwischen protected und private. Da ein Lok-Objekt nie ein weiteres Lok-Objekt verarbeiten muss, ist private die korrekte Wahl.

Beispiel 2-112: Lok#aktuelle_zuglast zieht sich ins Private zurück

```

class Lok
  private
  def aktuelle_zuglast
    zl = 0
    @waggons.each { |w|
      zl += w.gewicht
    }
    return zl
  end
end

```

Nun wird es Zeit für ein paar Tests. Gleich zwei Dinge sollen dabei auf den Prüfstand. Denn nicht nur das Ankuppeln eines Waggon(-Objekt)s soll ausprobiert werden, sondern auch, ob aktuelle_zuglast wirklich nicht als nach außen sichtbare Methode eines Lok-Objekts genutzt werden kann.

```

waggon1 = Waggon.new(30, 15)
waggon2 = Waggon.new(50, 23)
lok.ankuppeln(waggon1) # => "Angekuppelt: Personenwagen mit 30 Sitzen. Gewicht:
  15 Tonnen."
lok.ankuppeln(waggon2) # => "Angekuppelt: Personenwagen mit 50 Sitzen. Gewicht:
  23 Tonnen."
lok.ankuppeln(waggon1) # => "Waggon ist bereits angekuppelt"
lok.aktuelle_zuglast # NoMethodError

```

Was anzukuppeln geht, muss auch wieder abgekuppelt werden können. Das übernimmt die Methode abkuppeln. Sie entfernt, so vorhanden, den letzten Waggon am Zug und gibt ihn zurück. Dafür eignet sich die Methode pop der Array-Klasse hervorragend, so dass Lok#abkuppeln schnell geschrieben ist.

Beispiel 2-113: Lok#abkuppeln entledigt sich des letzten Waggons

```
class Lok
  def abkuppeln
    if @waggons.size > 0
      "Abgekoppelt: #{@waggons.pop.to_s}"
    else
      'Kein Waggon angekuppelt.'
    end
  end
end
```

Ähnlich der ankuppeln-Methode wird hier ein String-Objekt zurückgegeben, das Informationen zum Vorgang enthält. Sinnvoll wäre auch die Rückgabe des abgekoppelten Waggon-Objekts, um es später wieder ankuppeln zu können – das Stichwort lautet »Rangieren«. Array#pop schließlich liefert das entfernte Element. Aber für Ihre ersten Gehversuche mit Klassen sollen uns aussagekräftige Meldungen reichen, die das Nachvollziehen des Innenlebens für Sie leichter machen.

```
lok.abkuppeln # => "Abgekoppelt: Personenwagen mit 50 Sitzen. Gewicht: 23 Tonnen."
lok.ankuppeln(waggon2) # => "Angekuppelt: Personenwagen mit 50 Sitzen. Gewicht:
23 Tonnen."
```

Vorsicht an den Türen!

Nun soll der Lokführer die Macht über all die Türen der hinter ihm dahinrollenden Waggons erhalten. Die Implementation der Methode, die den Türenstatus aller Waggons prüft, soll eine für den Lokführer lesbare Textbotschaft ausgeben. Aber wie Sie sich so richtig erinnern, soll die Methode beschleunigen nur dann ansprechen, wenn auch wirklich alle Türen geschlossen sind. Da Ruby-Methoden anders ticken als Lokführer, soll beschleunigen keine Text- sondern eine verwertbarere Botschaft in Ruby-Art erhalten.

Zum Beispiel in Form von Symbolen: :geschlossen, wenn alle Türen dies von sich behaupten können oder :offen, wenn alle oder einige dies sind. Die Art der Rückgabe (Text für den Menschen, Symbole für Lok#beschleunigen) und die Vorgehensweise hängen dabei von dem optionalen Parameter intern ab, der nur für den Aufruf der Methode von innerhalb des Objekts bedeutsam ist. Für den Aufruf von außen erhält er durch seine Nichtbeachtung den Standardwert false.

Beispiel 2-114: Zustand der Türen

```
class Lok
  def tuerenstatus(intern = false)
    case intern
    when false
      # Text erzeugen
      return "Keine Waggon angekuppelt." if @waggons.size == 0
      status = ""
      @waggons.each { |w|
        if w.tueren_offen
```

Beispiel 2-114: Zustand der Türen (Fortsetzung)

```
status.concat("Tueren offen: #{w}\n")
else
  status.concat("Tueren geschlossen: #{w}\n")
end
}
return status
when true
# :offen oder :geschlossen?
@waggons.each { |w|
  return :offen if w.tueren_offen
}
return :geschlossen
end
end
end
```

Wenn `intern false` ist, wird ein String-Objekt in der Variable `status` erzeugt. Anschließend wird `status` pro Blockdurchlauf mit der `waggonspezifischen` Info zum Thema Türen ergänzt und letztendlich als Rückgabewert genutzt. Innerhalb des Blocks kommt wieder `Waggon#to_s` zum Einsatz, wenngleich unbemerkt. Ruby verwendet die Methode automatisch zum Auswerten des im String enthaltenen Ausdrucks, der einfach nur aus dem Waggon-Objekt `w` besteht.

Sollte die Methode `intern` aufgerufen worden sein, wird ebenfalls Waggon für Waggon analysiert. Allerdings endet die Suche mit dem Rückgabewert `:offen`, sobald auch nur ein Waggon mit offener Tür gefunden wurde. Das reicht schließlich als Kriterium, um nicht abzufahren. Sollte dies nicht der Fall sein, wird `:geschlossen` zurückgegeben.

Nun können wir auch die Methode `beschleunigen` ergänzen. Allerdings nur um eine Zeile, die einen informativen Hinweistext für den Lokführer bereithält. Beachten Sie hierbei, dass die Methode `tuerenstatus` von `beschleunigen` mit dem Parameter `true` aufgerufen wird.

Beispiel 2-115: Erst Türen schliessen, dann abfahren!

```
class Lok
def beschleunigen
  return 'Erst Tueren schliessen, dann abfahren!' if tuerenstatus(true) == :offen
  @geschw += 10
  @geschw = @hgeschw if @geschw > @hgeschw
  @geschw
end
end
```

Für das Öffnen und Schließen der Türen schreiben wir noch fix eine Methode, die die Türen bedient und ihre Befehle über einen Parameter erhält, der als Werte Symbole erwartet, `:oeffnen` und `:schliessen`. Hier bauen wir gleich einen Sicherheitscheck ein, der bei der Aktion `:oeffnen` und einer für den Ausstieg hinderlichen

Geschwindigkeit jenseits von 0 km/h die Türen geschlossen lässt. Diese beiden Bedingungen werden dabei mit der logischen Verknüpfung and gemeinsam berücksichtigt.

Beispiel 2-116: Erst den Zug anhalten, dann Türen öffnen!

```
class Lok
  def tueren(aktion)
    return 'Erst den Zug anhalten, dann Tueren oeffnen!' if aktion == :oeffnen and
      @geschw > 0
      @waggons.each { |w|
        w.tueren_offen = aktion == :oeffnen
      }
      tuerenstatus
    end
  end
```

Um den Rückgabewert der Methode kümmert sich tuerenstatus. Diesmal verzichten wir auf den Parameter true, denn tuerenstatus soll ja etwas für die Außenwelt ausgeben. Bei einer kleinen Probefahrt sollen die neuen Methoden getestet werden.

```
lok.tuerenstatus
# => "Tueren geschlossen: Personenwagen mit 30 Sitzen. Gewicht: 15 Tonnen.
Tueren geschlossen: Personenwagen mit 50 Sitzen. Gewicht: 23 Tonnen."
lok.beschleunigen # => 10
lok.beschleunigen # => 20
lok.tueren(:oeffnen) # => "Erst den Zug anhalten, dann Tueren oeffnen!"
lok.bremsen # => 0
lok.tueren(:oeffnen)
# => "Tueren offen: Personenwagen mit 30 Sitzen. Gewicht: 15 Tonnen.
Tueren offen: Personenwagen mit 50 Sitzen. Gewicht: 23 Tonnen."
lok.beschleunigen # => "Erst Tueren schliessen, dann abfahren!"
```

Zwei Methoden fehlen uns noch bei unserer Lok. Da sie allerdings nicht sonderlich kompliziert sind, nichts Neues enthalten und sich von selbst erschließen, verfahre ich wie die Deutsche Bahn AG bei ihren Auskünften bei Zugausfällen, Verspätungen oder spontanen Streckenänderungen: Ich halte mich sehr, sehr kurz. Die Methode waggonzahl zählt alle Anhänger, to_s gibt die Lok in Schriftform wieder.

Beispiel 2-117: Lok#waggonzahl und Lok#to_s

```
class Lok
  def waggonzahl
    @waggons.size
  end
  def to_s
    "Lok mit #{@hgewicht} Tonnen Zugkraft. Waggons: #{@waggonzahl}.
    Maximale Geschwindigkeit: #{@hgeschw} km/h."
  end
end
```

Damit ist unsere Lok fertig. Sie können sie nun vor diverse Waggons spannen und mit ihr ein paar erholsame Ausflüge nach all den Anstrengungen machen. Aber kommen Sie bald wieder, denn es folgen noch einige wichtige Informationen zu Klassen und Objekten, die Sie sich nicht entgehen lassen sollten.

Objekte duplizieren

Wie Sie wissen, enthält `Lok#ankuppeln` einen Prüfmechanismus, der checkt, ob der anzuhängende Waggon nicht schon längst am Haken hängt.

```
lok.ankuppeln(waggon1) # => "Waggon ist bereits angekuppelt"
```

Lassen Sie uns ein wenig hinter die Kulissen blicken und erforschen, wie das funktioniert, und etwas experimentieren. Zunächst soll die Frage beantwortet werden, ob dieser Mechanismus vielleicht ausgehebelt werden kann, wenn wir dem Objekt einen anderen Namen geben.

```
waggon3 = waggon1  
lok.ankuppeln(waggon3) # => "Waggon ist bereits angekuppelt"
```

Das funktioniert nicht, weil es sich bei `waggon3` um exakt dasselbe Objekt handelt, das schon bei `waggon1` hinterlegt ist. Variablen sind eben nur Stellvertreter des Objekts, aber nichts Eigenständiges. Das können Sie auch erkennen, wenn Sie die ID der Objekte hinter `waggon1` und `waggon3` abfragen. Sie wissen bereits, dass jedes Objekt eine solche eindeutige Kennung besitzt, über die es ebenso eindeutig von Ruby identifiziert werden kann.

```
waggon1.object_id # => 22616320  
waggon3.object_id # => 22616320  
waggon1.object_id == waggon3.object_id # => true
```

Diese Gleichheit der Objekte wirkt sich natürlich nicht nur auf die Objekt-ID aus. Veränderungen, die Sie an `waggon1` vornehmen, werden sich auch auf `waggon3` auswirken.

```
waggon3.tueren_offen # => true  
waggon1.tueren_offen = false # => false  
waggon3.tueren_offen # => false  
waggon3.tueren_offen = true # => true  
waggon1.tueren_offen # => true
```

Es gibt allerdings eine Möglichkeit, wie Sie die *Kopie* eines Objekts nutzen können. Bei diesem Vorgang entsteht ein zweites Objekt, das dieselben Attributwerte aufweist wie das erste zum Zeitpunkt des Duplizierens. So können Sie beispielsweise einen Waggon, den Sie mit zwei Werten initialisiert haben, duplizieren. Es entsteht ein Waggon mit genau denselben Werten, allerdings als völlig selbstständiges Objekt. Die Methode `initialize` wird bei der Kopie allerdings nicht ausgeführt. Sie können ein Objekt, welcher Klasse auch immer, mit der Methode `dup` vervielfachen.

```

waggon3 = waggon1.dup
waggon3.sitze # => 30
waggon3.tueren_offen # => true
waggon3.tueren_offen = false # => false
waggon1.tueren_offen # => true
lok.ankuppeln(waggon3) # => "Angekuppelt: Personenwagen mit 30 Sitzen. Gewicht:
15 Tonnen."
waggon1.object_id # => 22616320
waggon3.object_id # => 22521170

```

Nun wissen Sie, wie Sie ein ganz spezielles Objekt herkömmlich erzeugen, individualisieren und dann mittels dup diverse Kopien davon erstellen können, die die Eigenschaften des Ursprungsobjekts besitzen. Das geht aber noch anders, womit wir zu einem wichtigen Thema im Bereich objektorientierter Programmierung kommen: *Vererbung*.

Erben ohne sterben

Sie haben im Verlauf dieses Buches bereits ab und zu etwas zum Thema *Vererben* gelesen. Hier und da war von Klassen die Rede, auf denen andere basieren. Und von der Mutter aller Klassen, die da Object hieß. Aber was genau bedeutet das?

Wie im richtigen Leben auch, erben in Ruby und anderen Programmiersprachen Kinder von Eltern. Eine *Elternklasse* gibt Ihre Funktionalität an eine *Kindklasse* weiter. Die Kindklasse basiert somit auf der Elternklasse und erbt alles von ihren Vorfahren. Somit steckt in der Kindklasse auch das, was die Elternklasse schon von ihrer Elternklasse geerbt hat – es sei denn, dieses Erbe hat die Elternklasse verändert. Die direkte Elternklasse, der direkte Vorfahre einer Klasse, wird in Ruby *Superklasse* genannt.

Schauen wir uns die Klasse Fixnum an. Mit der Methode ancestors sehen Sie alle Vorfahren der Klasse Fixnum. Mit der Methode superclass erfahren Sie, welche der Klassen der direkte Vorfahre war, von welcher Klasse also direkt geerbt wurde.

```

Fixnum.ancestors # => [Fixnum, Integer, Precision, Numeric, Comparable, Object, Kernel]
Fixnum.superclass # => Integer
Integer.superclass # => Numeric
Numeric.superclass # => Object
Object.superclass # => nil

```

Fixnum hat also von Integer geerbt, Integer von Numeric und Numeric von Object. Hierbei können Sie auch gut sehen, warum Object der Urahn aller Klassen ist. Object bildet die oberste Ebene der Ruby-Klassenhierarchie.

Anhand des Fixnum-Stammbaums können Sie prima erkennen, was denn hinter dem Prinzip Vererbung steckt. Object enthält allgemeine Funktionalitäten, die alle Ruby-Objekte gebrauchen können. Numeric fügt dem Methoden hinzu, die einzig für Zahlen interessant sind. Integer macht das auch, allerdings nur für ganzzahlige

Werte. Und Fixnum enthält schließlich alles Nötige für die eher etwas kleineren ganzzahligen Werte. Jede Klasse auf dem Weg von Object zu Fixnum ist spezialisierter als ihr Vorfahre.

Und damit kommen wir wieder zurück auf `waggon3`. Dieses Objekt ist eine Kopie eines speziellen Waggon-Objekts, nämlich von `waggon1`, welches wir mit konkreten Werten erzeugt haben.

Wenn Sie nun Ihre Lok hauptsächlich mit Waggons bestücken möchten, die 30 Sitze haben und 15 Tonnen wiegen, dann lohnt sich die Implementierung eines Nachfahren von Waggon. Lassen Sie uns die Kindklasse Spezialwaggon nennen.

Um Ruby klar zu machen, von welcher Klasse die neue erben soll, notieren Sie den Namen der Elternklasse getrennt durch ein Kleiner-als-Zeichen rechts von dem der Kindklasse. In der Klassendefinition können Sie eine `initialize`-Methode schreiben, die die der Superklasse überschreibt.

Da wir mit dem Spezialwaggon einen ganz speziellen Waggon erzeugen möchten, sparen wir uns die Übergabe von Parametern beim Erzeugen einer Instanz von Spezialwaggon. In `Spezialwaggon#initialize` rufen wir allerdings `Waggon#initialize` auf – und zwar mit unseren speziellen Werten. Dies erfolgt über die Methode `super`. Damit starten Sie eine Methode der Superklasse, die den gleichen Namen trägt wie die der Kindklasse, in der `super` aufgerufen wurde. Dementsprechend müssen Sie `super` mit den dazugehörigen Parametern der gleichnamigen Methode der Superklasse bestücken. Für unseren 30-Sitze-15-Tonnen-Waggon sieht das so aus:

Beispiel 2-118: Spezialwaggon erbt von Waggon

```
class Spezialwaggon < Waggon
  def initialize
    super(30, 15)
  end
end
```

So einfach ist Erben. Abgesehen von der Erzeugung hat sich nichts gegenüber einem Waggon-Objekt geändert, lediglich die Klassenzugehörigkeit.

```
waggon4 = Spezialwaggon.new
waggon4.to_s # => "Personenwagen mit 30 Sitzen. Gewicht: 15 Tonnen."
lok.ankuppeln(waggon4) # => "Waggon hat keine Verkehrserlaubnis"
```

Da tut sich ein Problem auf: Da `waggon4` der Klasse Spezialwaggon angehört, die Klasse Lok aber auf Instanz der Klasse Waggon prüft, wird das Ankuppeln verweigert. Das geht natürlich nicht, schließlich ist Spezialwaggon baugleich mit Waggon.

Lassen Sie uns daher die Methode `ankuppeln` der Lok-Klasse in einer Zeile ändern. Beim Überprüfen der Klassenzugehörigkeit reicht nicht mehr nur das Abfragen der eigenen Klasse, sondern auch die Vorfahren müssen betrachtet werden. Wenn bei denen die Klasse Waggon verzeichnet ist, soll die Verkehrserlaubnis erteilt werden.

Mit der Methode `is_a?` können Sie überprüfen, ob ein Objekt einer bestimmten Klasse angehört, wobei auch alle beerbten Klassen berücksichtigt werden.

Beispiel 2-119: Ist dein Vorfahre ein Waggon?

```
class Lok
  def ankuppeln(waggon)
    # Ist waggon ein Objekt oder Nachfahre der Waggon-Klasse?
    return 'Waggon hat keine Verkehrserlaubnis.' unless waggon.class.ancestors.
      include?(Waggon)

    # Waggon bereits angekuppelt?
    return 'Waggon ist bereits angekuppelt.' if waggon.is_a?(Waggon)

    # Wird Höchstgewicht überschritten?
    if (waggon.gewicht + aktuelle_zuglast) < @hgewicht
      @waggons.push(waggon)
      return "Angekuppelt: #{waggon.to_s}"
    else
      return 'Waggon zu schwer.'
    end
  end
end
```

Nun sollte es möglich sein, an unserer Lok auch einen Spezialwaggon zu hängen.

```
lok.ankuppeln(waggon4) # => "Angekuppelt: Personenwagen mit 30 Sitzen. Gewicht:
  15 Tonnen."
lok.waggonzahl # => 4
```

Lassen Sie uns jetzt noch die Schablone eines anderen Spezialwaggons bauen. Ein Speisewaggon soll den Reisekomfort erhöhen. In der gleichnamigen Klasse soll lediglich die `to_s`-Methode angepasst werden. Indem Sie eine Methode der Superklasse in der Nachfolgerklasse überschreiben, ist dieser Teil des Erbes vergessen.

Beispiel 2-120: Zu Tisch!

```
class Speisewaggon < Waggon
  def to_s
    "Speisewagen mit #{@sitze} Sitzen. Gewicht: #{@gewicht} Tonnen."
  end
end
```

Ein Speisewaggon-Objekt muss wie ein Waggon-Objekt erzeugt werden. Es hat sich schließlich im Gegensatz zu Spezialwaggon nichts an der `initialize`-Methode geändert.

```
waggon5 = Speisewaggon.new(20, 16)
lok.ankuppeln(waggon5) # => "Angekuppelt: Speisewagen mit 20 Sitzen. Gewicht:
  16 Tonnen."
```

Sie sehen also, auch beim Erben ist Ruby recht flexibel. Aber das war noch nicht alles zum Thema »Weitergeben von Funktionalitäten«. Mit Speisewaggon und *Spezialwaggon* haben Sie zwei Klassen implementiert, die von einer Klasse erbten, *Waggon*.

Nicht ganz zufällig spricht man dabei von *Einfachvererbung*. Nun gibt es bei einigen Programmiersprachen auch das Konzept der Mehrfachvererbung. In diesem Fall kann eine Klasse sich an dem Können gleich mehrerer Vorfahren laben. Und jetzt die traurige Nachricht: Mehrfachvererbung kann Ruby nicht. Der Grund: Ruby hat was viel, viel Besseres.

Module, Mixins, Namensräume

Erinnern Sie sich an unsere Frage an die Fixnum-Klasse bezüglich ihrer Vorfahren? Mit `ancestors` und `superclass` sind wir den Fixnum-Stammbaum entlanggeklettert. Vielleicht ist Ihnen aufgefallen, dass wir dabei gar nicht allen Klassen begegnet sind, die die Methode `ancestors` gelistet hat?

```
Fixnum.ancestors # =>
  [Fixnum, Integer, Precision, Numeric, Comparable, Object, Kernel]
Fixnum.superclass # => Integer
Integer.superclass # => Numeric
Numeric.superclass # => Object
```

Mit `Object` haben wir die Wurzel erreicht. Was ist also mit den Klassen `Precision`, `Comparable` und `Kernel`? Die Antwort: Sie tauchen nicht als Vorfahren auf, weil diese Klassen gar keine Klassen sind – sondern *Module*. Sie sind Rubys Antwort auf Mehrfachvererbung und können als so genannte *Mixins* Ruby-Klassen beigemengt werden.

```
Fixnum.class # => Class
Object.class # => Class
Comparable.class # => Module
Precision.class # => Module
Kernel.class # => Module
```

Wir Sie hier sehen, sind auch Module Objekte, Instanzen der Klasse `Module`. Aber das überrascht natürlich keinen mehr. Module haben einige Gemeinsamkeiten mit Klassen. So können auch Module als eine Ansammlung zusammengehöriger Methoden angesehen werden. Im Gegensatz zu Klassen kann man mit ihnen aber keine Objekte erzeugen. Sie sind nicht instanziierbar.

Module werden vielmehr dafür genutzt, Funktionalitäten, die mehrere Klassen ganz gut gebrauchen können, einmal zu implementieren und sie als Mixin in diese Klassen einzupflanzen. Schauen Sie beispielsweise in das Familienbuch der `String`-Klasse, die grundsätzlich nicht so viel mit Zahlen am Hut hat. Dennoch werden Sie feststellen, dass `Fixnum` und `String` (und nicht nur die beiden) ein Stück ihres Könbens aus ein und demselben Modul entnommen haben.

```
String.ancestors # => [String, Enumerable, Comparable, Object, Kernel]
String.ancestors & Fixnum.ancestors # => [Comparable, Object, Kernel]
(String.ancestors & Fixnum.ancestors).find_all { |m|
  m.class == Module
} # [Comparable, Kernel]
```

Das Kernel-Modul ist bei beiden vertreten, weil es schon in die Klasse `Object`, auf der beide basieren, gemixt wurde. In Kernel stecken unheimlich viele grundsätzliche Methoden, die jede Klasse gebrauchen kann. Darüber hinaus enthält Kernel auch einige Methoden, die ohne Receiver aufgerufen werden, so zum Beispiel `gets`, `puts` und `rand`.

`Comparable` enthält allerlei Methoden zum Vergleichen. Und vergleichen, das geht sowohl mit Ganzzahlen und Strings. In `Comparable` enthalten sind `<`, `<=`, `==`, `>=`, `>` und `between?`. Diese Methoden basieren alle auf einer einzigen: `<=>`. Die kennen Sie bereits, wenn Sie einen ganzen Schwung Seiten zuvor dem Sortieren Schweizer Städte nach Wortlänge beigewohnt haben. Jede Klasse, die `<=>` implementiert und `Comparable` integriert, kann auf die eben genannten Methoden zurückgreifen.

Probieren wir das mal mit unserer `Waggon`-Klasse aus und fügen wir ihr die Methode `<=>` und das Modul `Comparable` hinzu. Basis für `<=>` soll das Verhältnis zwischen Sitzplatzanzahl und Gewicht eines Waggons sein. Je wirtschaftlicher das ist, also je kleiner das eben angesprochene Verhältnis, desto höher ist der Wert eines Waggons. Der Quotient der beiden mitunter ganzzahligen Werte für Sitze und Gewicht soll als Float-Wert betrachtet werden. Die `Float`-Klasse besitzt bereits die Methode `<=>`, so dass wir uns Arbeit sparen können. Und diese Tatsache lässt völlig richtig erahnen, dass auch die Klasse `Float` vom Modul `Comparable` zehrt.

Beispiel 2-121: Wagons werden vergleichbar

```
class Waggon
  def <=>(w)
    (@sitze / @gewicht.to_f) <=> (w.sitze / w.gewicht.to_f)
  end
end
```

Wenn Sie nun bereits versuchen, Vergleiche wie `waggon1 < waggon2` oder `waggon2.between?(waggon1, waggon4)` anzustellen, wird Ihnen das nicht gelingen, schließlich müssen Sie noch das Modul `Comparable` in die Klasse `Waggon` integrieren.

Dies erfolgt mit der Methode `include`, die als Parameter das benötigte Modul erhält. Achten Sie hier besonders auf die Groß- und Kleinschreibung. Notieren Sie `include` möglichst vor allen Methoden innerhalb einer Klassendefinition.

Beispiel 2-122: Wagons werden noch vergleichbarer

```
class Waggon
  include(Comparable)
end
```

Durch diese eine Zeile hat unsere `Waggon`-Klasse ganz leicht gleich fünf neue Methoden erhalten, die Sie sonst von Hand hätten schreiben müssen. Und nicht nur `Waggon`, auch `Spezialwaggon` und `Speisewaggon` profitieren davon, da beide Klassen Erben von `Waggon` sind.

```
Waggon.ancestors # => [Waggon, Object, Comparable, Kernel]
Spezialwaggon.ancestors # => [Waggon, Object, Comparable, Kernel]
Speisewaggon.ancestors # => [Waggon, Object, Comparable, Kernel]
```

Das bedeutet, dass Sie nun getrost `waggon4` (Spezialwaggon) mit `waggon1` (Waggon) oder `waggon5` (Speisewaggon) nach Wirtschaftlichkeit untersuchen können.

```
waggon1 >= waggon2 # => false
waggon2 < waggon3 # => false
waggon5 < waggon1 # => true
waggon1.between?(waggon5, waggon2) # => true
```

Alle Vergleichsoperatoren, die Sie hauptsächlich von Zahlen kennen, können Sie nun also auch auf Objekte der Klassen Waggon, Spezialwaggon und Speisewaggon anwenden.



Durch die Implementation der Methode `<=>` haben Sie eine weitere Funktionalität erhalten! Sie können nun mit der blockfreien Variante von `Array#sort` ein Array von Waggon- und Waggon-Nachfahren-Objekten sortieren lassen. Die `sort`-Methode arbeitet ebenfalls auf Basis von `<=>`.

Natürlich sind Sie nicht darauf beschränkt, ausschließlich bereits integrierte Module in Ihren Klassen zu nutzen. Wenn Sie meinen, Code schreiben zu wollen, der in mehreren Klassen Verwendung finden kann, dann können Sie natürlich auch ein eigenes Modul schreiben.

Die Struktur eines Moduls ähnelt dabei der einer Klasse sehr. Notieren Sie im Modulkopf den Namen nach den gleichen Konventionen wie bei Klassen. Im Rumpf Ihrer Moduldefinition können Sie Methoden notieren, mit `self` eine Instanz der Klasse referenzieren, der das Modul eingepflanzt wurde, und Sie können die in einem Modul definierten Attribute in der Klasse benutzen – wenn Sie das möchten. Im folgenden Modul kapseln wir das Licht für unsere Waggon-Objekte.

Beispiel 2-123: Waggonlicht als Modul

```
module Waggonlicht
  attr_reader(:licht)

  def lichtschalter
    @licht = !@licht
  end
end
```

Benutzt eine Klasse das Modul `Waggonlicht` als Mixin, dann verfügt sie automatisch über ein Attribut `@licht`, das über einen Akzessor von außen als `licht` lesbar ist, und eine Methode `lichtschalter`, die den booleschen Wert von `@licht` ins Gegenteil verkehrt.

Es gibt nur ein Problem: Ein Modul hat keine `initialize`-Methode, wie Sie es von Klassen kennen. Das ist auch nicht nötig, da Module ohnehin nicht instanziert werden können. Dennoch braucht `@licht` einen Startwert. Da aber eine Klasse die Attribute eines eingebundenen Moduls wie die eigenen behandeln kann, erweitern wir einfach `Waggon#initialize`. Und bei der Gelegenheit weisen wir an, das Modul `Waggonlicht` zu integrieren. In diesem Zusammenhang sei erwähnt, dass Sie theoretisch Unmengen von Modulen in einer Klassendefinition unterbringen können. Mit `include` bleiben auch diese Massen übersichtlich.

Beispiel 2-124: Waggons bekommen Licht

```
class Waggon
  include(Waggonlicht)

  def initialize(sitze, gewicht)
    @sitze = sitze
    @gewicht = gewicht
    @tueren_offen = false
    @licht = false
  end
end
```

Auf gleiche Weise könnten Sie nun auch Licht ins Dunkel der Lok bringen. Denn erst mit der Wiederverwendung von Modulen werden diese so richtig wertvoll. Testen wir aber erst einmal nur die Lichtenlage unserer Waggons.

```
waggon6 = Waggon.new(40,27)
waggon6.licht # => false
waggon6.lichtschalter # => true
waggon6.licht # => true
```

Module können nicht nur wie hier als Mixins und exzellenter Mehrfachvererbungsersatz dienen. Sie schaffen auch Ordnung in besonders großen Codemengen – wenn Sie Module zur Schaffung von *Namensräumen* nutzen.

Namensräume durch Module

Mit *Namensräumen* können Sie beispielsweise einen Satz von semantisch zusammenhängenden Klassen bündeln und sie so von einem weiteren Bündel trennen, welches eine ganz andere Funktionalität in sich vereint. So ist es sogar möglich, Klassen mit gleichen Klassenbezeichnern zu definieren, ohne Angst zu haben, dass es zu Konflikten kommt. Sie werden dann über den jeweiligen Namensraum ange- sprochen.

Doch nicht nur Klassen, sondern auch Methoden, Variablen und Konstanten können Sie in einen Namensraum sperren. Wenn es sein muss, können Sie sogar andere Module dort unterbringen. Im folgenden Fall hat es eine Sammlung von thematisch zueinander passenden Konstanten erwischt.

Beispiel 2-125: Familie Schmitt, modularisiert

```
module Schmitt
  VATER = 'Klaus'
  MUTTER = 'Gerlinde'
  KIND1 = 'Ludger-Oliver'
  KIND2 = 'Zoe'
end
```

Ein Zugriff auf die Konstante VATER ist nun in herkömmlicher Art nicht mehr möglich. Setzen Sie den Modulbezeichner davor und fügen Sie ihm zwei Doppelpunkte gefolgt von der gewünschten Konstante hinzu.

```
VATER # NameError: uninitialized constant VATER
Schmitt::VATER # => "Klaus"
Schmitt::MUTTER # => "Gerlinde"
 "#{Schmitt::KIND1} und #{Schmitt::KIND2} sind Geschwister" # => "Ludger-
Oliver und Zoe sind Geschwister"
```

Lassen Sie uns noch eine Familie erzeugen und ihr zugleich eine realistische Note verleihen, damit Sie sehen können, wie Namensräume miteinander können.

Beispiel 2-126: Familie Schneider, kurz vor der Scheidung der Eltern

```
module Schneider
  VATER = 'Otto'
  WAHRER_VATER = Schmitt::VATER
  MUTTER = 'Martha'
  KIND1 = 'Lotte'
end
```

Auch hier können Sie auf die im Modul definierten Konstanten nur über die Nennung des Namensraums zugreifen.

```
Schneider::VATER # => "Otto"
Schneider::MUTTER # => "Martha"
Schneider::VATER == Schneider::WAHRER_VATER # => false
 "#{Schmitt::KIND1} und #{Schneider::KIND1} sind auch Geschwister. Aber pssst!" # =>
  "Ludger-Oliver und Lotte sind auch Geschwister. Aber pssst!"
```

Wir hätten beispielsweise auch unseren kleinen hochmodernen Zug der vergangenen Seiten mit einem Namensraum versehen können. Dann bestünde Zug aus den Klassen Zug::Lok und Zug::Waggon. Eine durchaus sachgerechte Lösung.

Achten Sie aber darauf, dass jede Referenz auf etwas, das sich innerhalb des Moduls befindet, auch über die Namensraumnotation angesprochen werden muss. Das ist besonders wichtig, wenn eine Klasse explizit von einer anderen erben soll, wie hier etwa Spezialwaggon. Wenn Sie das Modul Waggonlicht in das Modul Zug stecken möchten, müssen Sie ebenfalls darauf achten, include mit Zug::Waggonlicht in der Waggon-Klasse zu bestücken.

```
module Zug
  class Waggon
    ...
  end
```

```

class Spezialwaggon < Zug::Waggon
  ...
end
class Speisewagon < Zug::Waggon
  ...
end
class Lok
  ...
end
end

```

Wenn Sie nun einen Zug zusammenstellen möchten, ist die Angabe des Namensraums beim Erzeugen mit der Klassenmethode *new* dringend erforderlich. Interessant sind auch die Angaben, die Ihnen die *ancestors*-Methode bei einem solchen Objekt liefert.

```

lok2 = Zug::Lok.new(650, 200)
waggon7 = Zug::Waggon.new(50,30)
waggon8 = Zug::Spezialwaggon
lok2.ankuppeln(waggon7)
lok2.ankuppeln(waggon8)

```

Damit verlassen wir die aufregende Welt des Zugbaus und widmen uns einem Menschen, der Züge nur sitzend in der 1. Klasse benutzt. Im folgenden Beispielprogramm werden alle Techniken der vergangenen Seiten in einem praxisnahen Beispiel angewendet, wiederholt, vertieft und ergänzt. Dabei entsteht ein komplettes Programm mit Benutzerschnittstelle und der Speicherung von Daten in einer Datei.

Ruby-Programm 3: Der Arbeitsplatzoptimierer

Rubrecht Heuschreck ist reich. Sehr reich. Sein Geheimnis: skrupellose Geschäfte mit zwielichtigen Private-Equity-Fonds. Unternehmen anonym kaufen, Gewinn optimieren, Gewinn entnehmen, Unternehmen nahezu insolvent aber völlig über Wert und steuerbedarfsgünstigt wieder abstoßen – alles innerhalb weniger Wochen. Und das immer und immer wieder. Seine Gewinne aus den Fonds waren dabei in den letzten Monaten so exorbitant gestiegen, dass Rubrecht langsam Probleme bekam, neue Fonds zur Reinvestition zu finden. Und so nahm sich Rubrecht vor, doch mal selbst und ganz allein eine gut laufende Firma gewinnbringend in den Ruin zu treiben.

Nur wenige Tage später griff Rubrecht zum Telefon, um die ersten unternehmerischen Anweisungen an die Mitarbeiter seiner neu erworbenen Gummimuffen-Manufaktur durchzugeben: 1. Alle Mitarbeiter der Marketingabteilung zur ehrenamtlichen Arbeit überreden oder dem Arbeitsmarkt zur Verfügung stellen. 2. Hausmeister und Azubis umgefragt vor die Tür setzen, aber erst, nachdem sie geholfen haben, 3. die Damen der Buchhaltung auf ihren gut eingesessenen Bürostühlen zur örtlichen Agentur für Arbeit zu rollen. Um die Abwicklung, pardon, Optimierung der restlichen Belegschaft wird sich Rubrecht selbst kümmern.

Damit das leicht gelingt, tätigt der neue Firmeninhaber die erste und einzige Investition für den neuen Betrieb. Er kauft sich ein Ruby-Buch, weil er mal gehört hat, dass man mit Ruby ganz effizient programmieren kann. Und Effizienz – das ist was für Rubrecht. Schon kurze Zeit später gab Rubrecht die Daten seiner Mitarbeiter in sein selbst geschriebenes Ruby-Programm ein.

Was Rubrechts Software kann

Mit seiner Software kann er Mitarbeiterdaten eingeben, Mitarbeiter entlassen, eine Übersicht aller Mitarbeiter ansehen und ihren Lohn kürzen. Auf eine Funktion zum Erhöhen der Löhne hat er aus Gründen der Effizienz verzichtet. Aus dem gleichen Grund hat seine Software allerdings eine Funktion zum Speichern und Laden der Mitarbeiterdaten – sonst müsste Rubrecht bei jedem Programmstart die Daten neu eingeben.

Der ein oder andere Blick auf sein effizientes Programm lohnt auf alle Fälle, da Sie so die Chance haben, besonders Schleifen, Iteratoren und Klassen im Praxiseinsatz zu erleben. Aber es hält auch noch einige Neuigkeiten für Sie bereit. So erfahren Sie beispielsweise, wie einfach es ist, Objekte egal welcher Beschaffenheit in einer Datei zu speichern und aus selbiger auch wieder zu laden. Außerdem begegnen Sie Exceptions, einer sehr komfortablen Art, auf Fehler oder Probleme im Programmablauf adäquat zu reagieren.

Grundlegendes

Der Quelltext besteht diesmal aus mehreren eigenständigen Dateien. Sie laufen in der Schaltzentrale zusammen, die ebenfalls eine separate Datei belegt und *main.rb* heißt. Drei weitere Dateien werden Klassendefinitionen aufnehmen – pro Klasse eine Datei. Das ist übersichtlich, erleichtert die Pflege und ist optimal für eine weitere Verwendung der einzelnen Klassen außerhalb des Projekts.

Die Entscheidung, welche Klassen benötigt werden und was sie können sollen, basiert auf der Analyse des Problems, welches das Programm lösen soll. Im Mittelpunkt stehen die Arbeiter der Firma. Man kann Sie als *Menge von Objekten* formulieren, wodurch sich schon andeutet, dass ein Array das ideale Mittel ist, um sie programmseitig zu organisieren. Die Klasse, *Workers* soll sie heißen, soll aber nicht nur die Daten aller Arbeiter aufnehmen, sondern auch Methoden zur Verfügung stellen, die das Speichern, Laden, Einstellen, Entlassen, Anzeigen und das Senken der Löhne übernehmen.

Ein einzelnes Arbeiter-Objekt dieses Arrays soll mehrere Eigenschaften haben, die den Namen, den Vornamen und das Gehalt eines Arbeiters speichern sollen. Daher bietet es sich an, eine Klasse zu schreiben, die Getter und Setter für diese drei Attribute eines Arbeiters bereitstellt.



Die Benutzung zweier eng zusammengehöriger Klassen, von denen die eine als Bezeichner die Mehrzahl des Namens der anderen trägt, soll Sie schon einmal einstimmen auf eine ganz wichtige Konvention in Ruby on Rails.

Eine dritte Klasse ist lediglich die Erweiterung einer bereits bestehenden. Da die Senkung der Löhne prozentual erfolgen wird, ergänzt sie die Klasse Numeric, von der alle zahlenorientierte Ruby-Klassen erben, um eine Methode, die eine entsprechende Berechnung vornimmt.

In der Schaltzentrale wird das Interface des Programms untergebracht, über das das Programm gesteuert werden kann. Die zur Verfügung stehenden Aktionen können über die Auswahl eines Menüpunkts gestartet werden. Die Wahl sorgt anschließend für das Ausführen der entsprechenden, in Workers implementierten Methoden.

Damit Sie selbst nachvollziehen können, wie das Programm arbeitet, starten Sie jetzt RadRails und legen Sie ein neues Ruby-Projekt an. Nennen Sie es *Prog3* und wählen Sie Ihren bevorzugten Speicherort, beispielsweise C:\Ruby_apps\Prog3.

Klasse Worker

Um einem Projekt eine Datei hinzuzufügen, die eine Klasse enthalten soll, wählen Sie nach dem Rechtsklick auf den Projektbezeichner nicht *New → File*, sondern *New → Ruby Class*. Im folgenden Fenster können Sie angeben, welchem Projekt die Klasse zugeordnet werden (*Container*), von welcher Klasse die neue Klasse erben (*Superclass*) und wie Sie heißen soll (*Class name*).



Klicken Sie im Dialog New Ruby Class auf Browser... in der Zeile Superclass, um den Vorfahren aus einer Liste von bereits existierenden Klassen zu wählen. Hier sehen Sie allerdings nur Klassen, die innerhalb des Projekts definiert wurden.

Achten Sie darauf, dass Sie den Klassennamen nach den bekannten Konventionen wählen: Der Klassenname beginnt mit einem großen Buchstaben, gefolgt von kleinen, Zahlen und Unterstrichen. Mehrere Wörter können mit Unterstrich getrennt (Klassen_erzeuger) oder in Camel Case (KlassenErzeuger) notiert werden. RadRails erzeugt bei einem Klick auf OK automatisch eine Datei, die den Klassennamen als Dateinamen verwendet, allerdings komplett kleingeschrieben und mit Unterstrichtrennung zwischen zwei Wörtern, gefolgt von der Endung .rb – eben so, wie Ruby-Dateien heißen sollten.

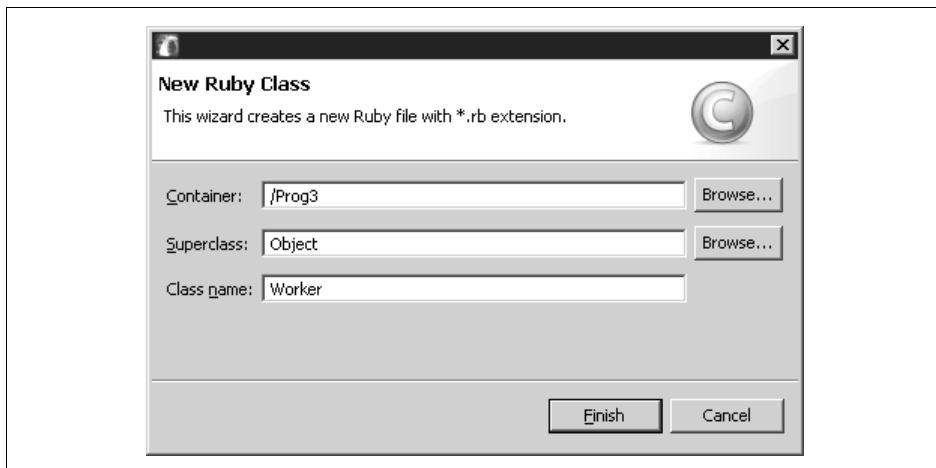


Abbildung 2-13: Mit einem Assistenten erstellen Sie in RadRails eine neue Klassendatei

Legen Sie hier fest, dass Sie eine Klasse *Worker* erzeugen möchten, die keinen besonderen Vorfahren hat und somit von *Object* erben soll. Die entstandene Datei *worker.rb* erhält von RadRails bei ihrer Erzeugung ein Klassen-Grundgerüst, bestehend aus Kopf mit Klassennamen und leerem Rumpf.

Die Klasse *Worker* soll lediglich Vorname, Name und Gehalt eines Arbeiters aufnehmen. Dementsprechend kurz ist die Niederschrift der Klasse.

Beispiel 2-127: Die komplette Klasse Worker

```
class Worker

  attr_accessor('forename', 'name', 'wage')

  def initialize(forename = '', name = '', wage = 0)
    @forename = forename
    @name = name
    @wage = wage
  end

  def <=>(other)
    result = @name <=> other.name
    result = @forename <=> other.forename if result == 0
    return result
  end
end
```

Wie Sie sehen, werden in *Worker* drei Attribute definiert, die von außen les- und schreibbar sind. Dies erfolgt über entsprechende Getter- und Setter-Methoden, welche automatisch von *attr_accessor* erzeugt werden und daher nicht selbst geschrie-

ben werden müssen. Die Methode `attr_accessor` kann hier deshalb angewendet werden, weil weder beim Setzen noch beim Auslesen der Werte von `forename`, `name` und `wage` weitere Programmfunktionalität wie Prüfungen oder Berechnungen implementiert werden muss.

Die `initialize`-Methode ist dank Parametern mit vordefinierten Standardwerten so konzipiert, dass Sie dem Anwender der Klasse freistellt, ein Worker-Objekt gleich von Beginn an mit den benötigten Werten zu versorgen oder nicht. Dank des Schreibzugriffs der Attribute können sie auch noch später mit Inhalt ausgestattet werden.

Das Highlight der Worker-Klasse ist sicher die `<=>`-Methode. In ihr ist klar definiert, nach welchen Kriterien mehrere Worker-Objekte zum Beispiel in einem Array sortiert werden sollen. Konkret steht dort drin, dass alle Arbeiter zunächst nach ihren Namen sortiert werden. Sollten dabei einmal zwei gleiche Namen zum Vergleich angetreten sein, dann ist in diesem Fall der Vorname das entscheidende Kriterium. Da String-Objekte – und `forename` und `name` sind als solche vorgesehen – werkseitig bereits mit `<=>` ausgestattet worden sind, wird darauf natürlich für die Teilvergleiche auch zurückgegriffen.



Wie Sie wissen, erlaubt die Implementation von `<=>` auch den Einsatz des Moduls `Comparable`. So könnten Sie `<`, `<=`, `>=`, `>` und `between?` auch für Worker-Objekte nutzen.

Mehr muss die Klasse `Worker` nicht können. Die restliche Funktionalität bezüglich der Verwaltung der Arbeiter, welche besonders die Pflege des aus mehreren Worker-Objekten bestehenden Datenmaterials betrifft, findet Platz in der `Workers`-Klasse, welche schon allein deshalb viel umfangreicher ausfällt. Sehen Sie `Worker` nur als Klasse für Objekte, die Arbeiterdaten speichert.

Proletarier, vereinigt euch!

Die Datei, die die Klasse `Workers` enthalten soll, können Sie wiederum über einen Rechtsklick auf `Prog3` erzeugen. Auch `Workers` soll von `Object` erben.



Natürlich könnten Sie auch eine Klasse entwickeln, die ein Nachfahre von `Array` ist und der nur noch die nötigen speziellen `Workers`-Methoden hinzugefügt werden. Damit würden Sie aber völlig die Kontrolle über Methoden verlieren, die jedes `Array`-Objekt mitbringt, die aber nicht in die Architektur der `Workers`-Klasse passen. Benutzer der `Workers`-Klasse würden viel zu viel Macht über die in `Workers` gespeicherten Worker erhalten.

Das beim Erzeugen der Datei `workers.rb` entstandene Grundgerüst kann gleich mit der obligatorischen `initialize`-Methode befüllt werden. In ihr soll lediglich das Attribut `@workers` als ein leeres Array erzeugt werden. Hier werden die einzelnen Worker-Objekte später einmal gespeichert.

Beispiel 2-128: Beginn der Klasse Workers

```
class Workers

  def initialize
    @workers = Array.new
  end
```

Die Methode zum Einstellen eines neuen Arbeiters, `hire`, die nun die `Workers`-Klasse ergänzen soll, ist recht kurz. Sie erhält als Parameter ein mit allen nötigen Informationen versehenes Worker-Objekt und fügt dieses per `Array#push` am Ende von `@workers` an. Anschließend wird die Liste auf Grund des Neueintrags sortiert. Wenn ich Sie erinnern darf: Das geht nur deshalb, weil in `Worker#<=>` steht, nach welchen Kriterien das Sortieren erfolgen soll. Der folgende Codeschnipsel ergänzt übrigens den vorhergehenden.

Beispiel 2-129: Arbeiter einstellen mit hire

```
def hire(worker)
  @workers.push(worker)
  @workers.sort!
end
```

Bitte beachten Sie das Ausrufungszeichen am Ende der `sort`-Methode, ohne das keinerlei Veränderungen am Array `@workers` vorgenommen würden.

Viel wichtiger als das Einstellen ist Rubrecht natürlich das Entlassen eines Arbeiters. Ruby-seitig übernimmt das `Workers#fire`. Die Methode erhält den Index desjenigen Worker-Objekts im sortierten Array `@workers`, das gelöscht werden soll. Später wird diese Methode durch das Menü der Zentrale aufgerufen. Somit kommt auch von ihr der Parameter.

Beispiel 2-130: Arbeiter entlassen mit fire

```
def fire(index)
  @workers.delete_at(index)
end
```

Die Methode `Array#delete_at` verlangt nach dem Index des zu löschen Objekts des Arrays, welches als Empfänger fungiert. Wichtig hierbei: `Array#delete_at`, und damit auch `Workers#fire`, gibt das Element zurück, welches gelöscht wurde – oder aber `nil`, wenn der angegebene Index nicht existierte. Mit diesem Rückgabewert kann die Zentrale eine Meldung für den Benutzer generieren, die entweder das Löschen eines Arbeiters bestätigt oder vom Scheitern des Vorhabens berichtet.

Mit der Methode `cut_wages`, einer weiteren Ruby gewordenen Leidenschaft Rubrechts, können die Löhne aller Arbeiter auf einen Schlag um einen gewissen Prozentsatz gesenkt werden. Dazu erhält die Methode den gewünschten Prozentwert als Parameter. Mit dem Iterator `Array#each` wird die Lohnkürzung dann allen Worker-Objekten in `@workers` mitgeteilt.

Beispiel 2-131: cut_wages: Allgemeine Lohnkürzung leicht gemacht

```
def cut_wages(value)
  @workers.each{ |worker|
    worker.wage -= value.percents_of(worker.wage)
  }
end
```

Sie sehen in `cut_wages` den Aufruf einer Methode, die es noch gar nicht gibt, von der Sie aber zumindest in den einführenden Erläuterungen zu Rubrechts Software schon gelesen haben. Die Methode `percents_of` berechnet, wie viel x Prozent von einem Wert y sind. Mit `percents_of` können Sie Ruby-Code so schreiben, wie Sie normalerweise sprechen: *40 Prozent von 160*, zum Beispiel. Das Ergebnis der Berechnung wird mittels `-=` von dem momentanen Verdienst eines jeden Arbeiters abgezogen. Um die konkrete Implementierung von `percents_of` soll es einige Absätze weiter gehen.

Mit `Workers#workers_list` existiert eine Methode, die eine Auflistung aller Arbeiter mit Namen, Vornamen und Gehalt zurückgibt.

Beispiel 2-132: Auskunft über die Belegschaft

```
def workers_list
  return "Keine Arbeiter vorhanden." unless @workers.size > 0
  list = ""
  @workers.each_with_index{ |worker, index|
    list.concat("#{index + 1} | #{worker.name}, #{worker.forename}, #{worker.
wage} Euro \n")
  }
  return list
end
```

Zunächst untersucht die Methode, ob im Array `@workers` überhaupt Arbeiter gespeichert sind. Sollte dies nicht der Fall sein, wird die Methode mit einer entsprechenden Meldung sofort abgebrochen. Im anderen Fall iteriert `each_with_index` über alle Elemente von `@workers`. Der Iterator `each_with_index` funktioniert wie `each`, liefert bei jedem Blockdurchlauf aber auch den Indexwert des jeweiligen Array-Elements innerhalb des Arrays. Die Übergabe an den Block erfolgt über eine zweite Blockvariable, hier `index` genannt.

Der Block wird elementweise immer wieder durchlaufen, wobei jedes Mal die als leerer String initialisierte lokale Variable `list` mit einer neuen Zeile ergänzt wird. Jede Zeile besteht aus einer fortlaufenden Nummer, dem Nachnamen, dem Vorna-

men und dem Verdienst. Die Nummerierung, die sich aus den Indexwerten der Worker-Objekte ergibt, ist besonders dann wichtig, wenn ein Arbeiter bestimmt werden soll, der fortan keine Berechtigung für diese Bezeichnung mehr haben soll. Wie Sie wissen, beginnen Arrays beim Zählen bei 0. Der Mensch ist das aber nicht gewohnt. Somit wird der Indexwert bei der Anzeige der Arbeiterliste stets um 1 erhöht, so dass diese Liste bei 1 beginnt. Diese Veränderung muss bei der Verwertung der Benutzereingabe unbedingt berücksichtigt werden – aber dazu kommen wir gleich noch.

Beachten Sie, dass jede Zeile, die `list` angefügt wird, mit einem erzwungenen Zeilenumbruch `\n` endet. So ist sichergestellt, dass die Auflistung bei der Ausgabe auf dem Bildschirm wirklich eine Liste ist und kein bunt aneinander geheftetes String-Durcheinander.

Marshalling, Datei-Handling, Exceptions

Wir werden uns nun mit der Methode `Workers#save_to_file` beschäftigen. In dieser einen Methode steckt eine ganze Menge Neues für Sie drin. Sie erfahren, wie Sie Objekte speichern, Dateien öffnen und dabei auftretende Fehler sicher abfangen können.

Vielleicht wissen Sie von den Schwierigkeiten, die einem Programmierer begegnen, wenn dieser versucht, komplexe Datenstrukturen zu speichern und zu laden. Das Attribut `@worker` ist recht komplex: Selbst ein Array-Objekt, speichert es diverse und eine in der Anzahl unbestimmte Menge an Worker-Objekten, die selbst durch je drei Attribute charakterisiert werden. Während Sie in den meisten anderen Programmiersprachen eigene Dateiformate oder komplizierte XML-Konstrukte erzeugen müssen, geht es in Ruby ganz einfach, diese komplizierte Datenstruktur zu speichern.

Selbst die komplexesten Strukturen können Sie mit dem so genannten *Marshalling* in eine Datei speichern. Dabei wird der komplette Objektinhalt *serialisiert*, also nach einem bestimmten Muster hintereinander in die Datei geschrieben. Ruby stellt Ihnen Marshalling über eine Bibliothek zur Verfügung, die automatisch von Ruby eingebunden und als Modul zur Verfügung gestellt wird.

Das Modul `Marshal` besitzt lediglich zwei Methoden: `dump` und `load`. Mit `Marshal.dump` wandeln Sie ein Objekt in das Marshal-Format um, mit `Marshal.load` können Sie den umgekehrten Weg beschreiten.

Die `dump`-Methode erwartet als ersten Parameter das Objekt, das *gemarshalt* werden soll. Der zweite Parameter ist optional und kann eine Referenz auf eine Datei, genauer eine Instanz der Ruby-Klasse `IO` oder einer ihrer Nachfahren, aufnehmen. Wenn Sie den zweiten Parameter weglassen, liefert `dump` einen String zurück, den Sie dann natürlich auch in einer Datei oder beispielsweise in einer Datenbank speichern können.

Beispiel 2-133: Speichern von @workers in eine Datei

```
def save_to_file(filename)
  f = File.open(filename, 'w')
  Marshal.dump(@workers, f)
  f.close
end
```

Die Methode `Workers#save_to_file` erhält als einzigen Parameter den Dateinamen, unter dem das serialisierte Array `@workers` abgespeichert werden soll. Diese Datei wird mit Hilfe eines Objekts der `File`-Klasse geöffnet, welches via `File#new` erzeugt und in `f` gespeichert wird.

Die Erzeugung des `File`-Objekts `f` erfolgt mit zwei Parametern. Der erste gibt die Datei inklusive Pfad an, die geöffnet werden soll. Der zweite Parameter, hier '`w`', bestimmt, *wie* die Datei geöffnet werden soll. Schließlich gibt es eine ganze Menge an unterschiedlichen Möglichkeiten, eine Datei zu öffnen. Folgende Werte können als String-Objekt angewendet werden, wobei `File#new` automatisch '`r`' als zweiten Parameter nutzt, sollten Sie keinen solchen angeben.

`'r'`

Lesezugriff, beginnend am Dateianfang

`'r+'`

wie '`r`', jedoch zusätzlich mit Schreibzugriff

`'w'`

Schreibzugriff; sollte die betreffende Datei bereits existieren, wird sie geleert, andernfalls erzeugt.

`'w+'`

wie '`w`', jedoch zusätzlich mit Lesezugriff

`'a'`

wie '`w`', jedoch wird eine bereits existierende Datei hierbei nicht geleert. Neue Inhalte werden ans Dateiende angehängt.

`'a+'`

wie '`a`', jedoch zusätzlich mit Lesezugriff

Mit `File.new` erzeugen Sie also ein Objekt, das den Zugriff auf eine Datei repräsentiert. Da die Klasse `File` Nachfahre der Klasse `I0` ist, können Sie ein solches Objekt bestens für den zweiten Parameter von `Marshal#dump` nutzen.

Nachdem alles Notwendige in die Datei geschrieben wurde, kann die offene Datei mit der Methode `File#close` geschlossen werden. Das Beenden eines Dateizugriffs sollten Sie nie vergessen.

Was noch fehlt, ist eine Kontrolle, die über das Öffnen der Datei und das anschließende Schreiben wacht. Es gibt mannigfaltige Möglichkeiten, weshalb ein Zugriff, zumal ein schreibender, auf eine Datei misslingen kann: Die Festplatte ist randvoll, ein anderes Programm greift auf die Datei zu, die Datei ist schreibgeschützt und so weiter.

Sie sehen schon, gerade der Dateizugriff ist eine ganz schön fehleranfällige Angelegenheit, zumal Probleme an den verschiedensten Stellen auftreten können. Mal kann ein Fehler beim Öffnen und mal erst beim Beschreiben einer Datei auftreten.

Daher nutzt Ruby – wie übrigens viele andere Sprachen auch – *Exceptions* für eine derartig umfangreiche Anzahl an möglichen Fehlerquellen. Tritt also irgendwo beim Zugriff auf eine Datei ein Fehler auf, löst Ruby eine Exception (engl. für Ausnahme) aus. Exceptions sind – tadaa! – natürlich Objekte in Ruby. Dabei gibt es diverse Klassen, die die Grundlage für ein Exception-Objekt bilden können. Für jeden Bereich, in dem Fehler auftreten können, gibt es eine spezifische, der Fehlerursache angepasste Klasse, wobei alle von der obersten Klasse *Exception* erben.

Sie müssen sich aber nicht unbedingt mit den einzelnen Klassen auseinander setzen. Es genügt völlig zu wissen, dass es in Ruby Exceptions gibt. Und mindestens genau so wichtig ist es, mit ihnen umzugehen. Die Tatsache, dass eine Exception, beispielsweise verursacht durch fehlenden Festplattenplatz des Nutzers, den gesamten Programmablauf stoppen kann, ist natürlich alles andere als befriedigend. Der bessere Weg wäre doch, eine Fehlermeldung anzuzeigen.

Das geht. Das Zauberwort heißt hier Exception-Behandlung. Damit ist gemeint, dass eine Exception durch explizit für sie geschriebenen Code *abgefangen* wird. Dabei wird das Programm nicht abgebrochen, und Sie können auf den Fehler reagieren. Das ist genau das Richtige für alle Dateioperationen und damit auch für `Workers#save_to_file`.

Zu erwähnen ist in diesem Zusammenhang, dass die Zentrale in *main.rb* erwartet, von `save_to_file` einen Rückgabewert zu erhalten, mit dessen Hilfe Rubrecht erfährt, ob das Speichern erfolgreich war oder nicht. Und das geht effektiv nur mit Exception-Behandlung.

Das Grundschema einer Exceptionbehandlung ist in Ruby recht logisch aufgebaut und bietet genügend Flexibilität für jeden Verwendungszweck.

```
begin
  # Code, in dem Exceptions, die behandelt werden sollen, vorkommen können
rescue
  # Code, der im Fehlerfall ausgeführt werden soll. Optionaler Teil.
else
  # Code, der ausgeführt werden soll, wenn alles fehlerfrei lief.
  # Optionaler Teil und nur, wenn ein rescue-Abschnitt existiert
ensure
  # Code, der auf jeden Fall ausgeführt werden soll. Optionaler Teil.
end
```

Bitte achten Sie bei der Implementierung Ihrer Exceptionbehandlung darauf, dass Sie diese Reihenfolge auch dann beachten, wenn einzelne Blöcke ausgelassen werden. Und vergessen Sie nicht, dass lediglich der Quelltext nach `begin` bei der Exceptionsbehandlung berücksichtigt werden kann.



Tritt in einer Methode eine Exception auf, die nicht behandelt wird, schaut Ruby, ob sich der Aufruf dieser Methode in einer Exceptionbehandlung befindet. Ist dies der Fall, wird so die Exception abgefangen.

Die Methode `save_to_file` sieht mit einer Exceptionbehandlung wie folgt aus:

Beispiel 2-134: `Workers#save_to_file` ist auf alle Eventualitäten vorbereitet

```
def save_to_file(filename)
begin
  f = File.new(filename, 'w')
  Marshal.dump(@workers, f)
rescue
  false
else
  true
ensure
  f.close unless f.nil?
end
end
```

Wie Sie sehen, wird im `ensure`-Abschnitt die Methode `close` nur dann ausgeführt, wenn es sich bei `f` auch wirklich um `File`-Objekt handelt. Es kann nämlich sein, dass `f` den Wert `nil` besitzt. Das ist dann der Fall, wenn beispielsweise eine Datei mit Schreibzugriff geöffnet werden soll, die aber einen Schreibschutz besitzt. Dann erzeugt `File#new` gar kein `File`-Objekt. Die beiden Objekte `false` und `true` sind die möglichen Rückgabewerte der Methode `save_to_file`.

Die Methode `load_from_file` kommt ebenfalls mit Exceptionbehandlung. Auch in ihr wird eine via `filename` definierte Datei geöffnet, allerdings nur mit Lesezugriff. Mit `Marshal.load` wird die Datei gelesen und wieder in ein Objekt gewandelt. Das `File`-Objekt `f`, welches den Dateizugriff regelt, wird dabei als Parameter übergeben.

Sollte das Lesen und Umwandeln des Dateiinhalts gelingen, zeigt `@workers` darauf, und die Methode gibt `true` zurück. Andernfalls bleibt `@workers` unverändert, und `false` wird zurückgegeben. Das Schließen der Datei wird durch den `ensure`-Teil der Exceptionbehandlung sichergestellt.

Beispiel 2-135: `load_from_file` bringt gespeicherte Daten wieder zurück

```
def load_from_file(filename)
begin
  f = File.open(filename, 'r')
  workers = Marshal.load(f)
rescue
  false
else
  @workers = workers
end
```

Beispiel 2-135: load_from_file bringt gespeicherte Daten wieder zurück (Fortsetzung)

```
    true
  ensure
    f.close
  end
end # Ende der Klassendefinition Workers
```

Damit ist die Klasse `Workers` einsatzbereit. Nun ja, fast. Denn es fehlt noch die Implementierung der Methode `percents_of`, ohne die ein `Workers`-Objekt nicht fehlerfrei arbeiten kann.

Numeric#percents_of

Der Klasse `Numeric` wird die Ehre zuteil, um eine selbst geschrieben Methode ergänzt zu werden. Das hat damit zu tun, dass sie ein Vorfahre aller Klassen ist, die mit Zahlen zu tun haben. Somit ist alles, was Sie `Numeric` hinzufügen, automatisch auch in `Fixnum`, `Bignum` und `Float` verfügbar.

Erzeugen Sie eine neue Ruby-Klasse auf die bekannte Art und Weise. Belassen Sie *Superclass* bei `Object` und wählen Sie `Numeric` als *Class name*. Durch die Implementierung einer vollwertigen Klassendefinition können Sie eine existierende Klasse ergänzen. Sie wissen ja, Klassen in Ruby sind offen. Offen für jede – auch nachträgliche – Veränderung.

Beispiel 2-136: Numeric bekommt eine neue Methode

```
class Numeric
  def percents_of(value)
    (self.to_f * value / 100)
  end
end
```

Vielleicht fragen Sie sich, warum ein Stück Quelltext der Klasse `Numeric` eine Methode, nämlich `to_f`, benutzen kann, obwohl die doch erst bei ihren Nachfolgern implementiert ist. Das ist schnell erklärt: Sie nutzen nicht `Numeric` direkt bei der Erzeugung von Zahlen-Objekten, sondern einen der Nachfolger, `Fixnum` oder `Float`. Da ist `to_f` sehr wohl bekannt. Und das reicht völlig aus.

Nun muss noch `workers.rb` mit `numeric.rb` bekannt gemacht werden. Denn nur, weil sich beide Dateien in ein und demselben Projekt befinden, kennen sie sich noch lange nicht. Fügen Sie zu Beginn von `workers.rb` und auf jeden Fall *außerhalb* der Klassendefinition die Zeile

```
require('numeric.rb')
```

ein. Mit `require` können Sie andere Ruby-Dateien einbinden. Übergeben Sie den Dateinamen via String-Objekt als Parameter. Nun fehlt eigentlich nur noch das Herz der Anwendung.

Die Schaltzentrale

Hier wird die Benutzerschnittstelle definiert und der Benutzer gebeten, einen Menüpunkt auszuwählen. Rubrechts *main.rb* fängt so an:

Beispiel 2-137: main.rb, Teil 1

```
require 'worker.rb'
require 'workers.rb'

# Konstanten
DATAFILE = 'workers.dat'

# Hauptschleife
$workers = Workers.new
loop {
    show_menu
    break if gets.to_i == 7
    navigate($_.to_i)
}
```

Wie Sie sehen, integriert *main.rb* zunächst einmal die beiden Klassendateien *worker.rb* und *workers.rb*. So kann *main.rb* auf die beiden Klassen zugreifen. Danach wird eine Konstante DATAFILE definiert, die den Dateinamen enthält, unter dem die Daten abgespeichert werden sollen. Mit \$workers wird zudem eine globale Variable erzeugt, die eine Instanz der Workers-Klasse enthält und alle anfallenden Daten aufnehmen wird. Eine globale Variable ist es deshalb, weil auch eine der zwei Methoden in *main.rb*, auf die wir gleich noch genauer schauen werden, an \$workers rankommen soll.

Das Menü selbst wird durch die Methode `show_menu` aufgerufen und angezeigt. Anschließend erwartet das Programm die Eingabe einer Zahl zwischen 1 und 7, mit der jeweils eine bestimmte Aktivität verknüpft ist. Weil das Menü nach der Ausführung einer gewünschten Aktion immer und immer wieder angezeigt werden soll, um neue Anweisungen des Benutzers zu erhalten, ist es im Schleifenkörper einer unendlichen `loop`-Schleife untergebracht. Es gibt jedoch eine Abbruchbedingung: Die Schleife und damit auch das komplette Programm wird beendet, sobald der Benutzer 7 eingegeben hat.

Sollte die Wahl auf eine andere Zahl gefallen sein, wird die Methode `navigate` mit der eingegebenen Zahl aufgerufen. Schauen Sie genau hin, und Sie werden als Parameter von `navigate` eine merkwürdige Zeichenfolge erkennen: `$_`. Diese standardmäßig stets vorhandene globale Variable enthält die jüngste Eingabe via `gets`. So brauchen Sie das Ergebnis von `gets` nicht in einer extra erzeugten lokalen Variable zu speichern.

show_menu und navigate

Nun fehlt noch die konkrete Implementierung der beiden innerhalb des Loops genannten Methoden `show_menu` und `navigate`. Während `show_menu` stets die zur Wahl stehenden Menüpunkte anzeigt, kümmert sich `navigate` darum, dass `$workers` mit einer mit der Eingabe korrespondierenden Methode ausgeführt wird.



Notieren Sie die beiden Methoden unbedingt zwischen der Konstanten und der Erzeugung des `$workers`-Objekts. Die `loop`-Schleife kann nur auf Methoden zurückgreifen, die ihr bekannt sind, die also vorher schon von Ruby durchlaufen worden sind.

In `show_menu` ist nichts Spektakuläres enthalten. Dort werden lediglich die Möglichkeiten via `puts` und `print` (falls eine in der gleichen Zeile anzuseigende Eingabe folgen soll) ausgegeben. Das parameterlose `puts` am Anfang der Methode sorgt dafür, dass dem Menü eine Leerzeile vorangeht, die es optisch vom Rest der Bildschirmanzeige trennt.

Beispiel 2-138: Menü anzeigen

```
def show_menu
  # Menü anzeigen
  puts
  puts('Navigation')
  puts('1 - Daten laden')
  puts('2 - Daten speichern')
  puts('3 - Arbeiter anzeigen')
  puts('4 - Arbeiter einstellen')
  puts('5 - Arbeiter feuern')
  puts('6 - Lohn senken')
  puts('7 - Programm beenden')
  print('Ihre Wahl: ')
end
```

Nach dem Anzeigen des Menüs und dem Eingeben einer einem Menüpunkt entsprechenden Zahl (siehe `loop`-Schleife) wird `navigate` ausgeführt. Ein `case-when`-Konstrukt sorgt dafür, dass die korrekte Methode ausgeführt wird. Bei einigen Menüpunkten ist es erforderlich, zusätzliche Angaben vom Benutzer abzufragen. So löst beispielsweise die 4 die Abfrage der Daten eines neuen Arbeiters aus. Daraus formt `navigate` ein `Worker`-Objekt und übergibt es `Workers#hire`.

Wenn ein Arbeiter gefeuert werden soll und die Taste 5 eine benutzerseitige Berührung erfuhr, wird eine Zahl abgefragt, die den Index des zu feuernden Arbeiters repräsentiert. Diese Zahl wird bei der Übergabe an `Workers#fire` um 1 verringert, schließlich beginnt der Index eines Arrays bei 0, die Anzeige der Liste jedoch bei 1. Die Methode `fire` gibt `nil` oder ein `Worker`-Objekt zurück, das Informationen über den Entlassenen zur Verfügung stellt.

Noch ein Hinweis zu der Verzweigung, die dem Drücken der Taste 6 geschuldet ist. Hier wird der eingegebene Prozentwert zunächst mit `to_f` in ein Float-Objekt gewandelt. Anschließend wird durch `abs` sichergestellt, dass Rubrecht nicht etwa eine Lohnerhöhung durch die Eingabe einer negativen Prozentzahl anordnet. Das wäre schließlich sein Ruin.

Beispiel 2-139: Programmverzweigungen in navigate

```
def navigate(choice)
  puts
  case choice
  when 1
    if $workers.load_from_file(DATAFILE) == false
      puts('Daten konnten nicht geladen werden.')
    else
      puts('Daten erfolgreich geladen.')
    end
  when 2
    if $workers.save_to_file(DATAFILE) == false
      puts('Daten konnten nicht gespeichert werden.')
    else
      puts('Daten erfolgreich gespeichert.')
    end
  when 3
    puts('Arbeiter anzeigen')
    puts($workers.workers_list)
  when 4
    puts('Arbeiter einstellen')
    worker = Worker.new
    print('Name: ')
    worker.name = gets.chomp
    print('Vorname: ')
    worker.forename = gets.chomp
    print('Lohn: ')
    worker.wage = gets.to_i.abs
    $workers.hire(worker)
  when 5
    puts('Arbeiter feuern')
    puts($workers.workers_list)
    print('Welcher Arbeiter soll gefeuert werden? ')
    worker = $workers.fire(gets.to_i - 1)
    if worker == nil
      puts('Arbeiter konnte nicht gefeuert werden.')
    else
      puts("#{worker.forename} #{worker.name} wurde erfolgreich gefeuert.")
    end
  when 6
    puts('Lohn senken')
    print('Um wie viel Prozent soll der Lohn sinken? ')
    $workers.cut_wages(gets.to_f.abs)
  end
end
```

Rubrechts Programm können Sie nun ausprobieren und kurzzeitig so selbst zum Turbokapitalisten mutieren. Stellen Sie doch einmal ein paar Arbeiter ein und speichern Sie Ihre Eingaben. Riskieren Sie einmal einen Blick in die Datei *workers.dat*, die bekanntlich ihren Inhalt durch Marshalling erhielt. So erhalten Sie einen Eindruck davon, was gemeint ist, wenn es heißt, dass Marshalling Objektdaten serialisiert. Ach ja, und danach das Feuern der Arbeiter nicht vergessen!

Rubrecht Heuschreck hat es sich übrigens doch anders überlegt und sein Unternehmen existiert wider Erwarten noch. Allerdings wurden alle Arbeiter der Gummimuffen-Manufaktur kurzerhand zu Ruby-Programmierern umgeschult.

Zusammenfassung

Mit dieser Arbeiterverwaltungssoftware schließt der Ruby-Grundkurs dieses Buches. Sie haben in diesem Kapitel gelernt, wie Ruby funktioniert und was das Besondere an dieser Sprache ist. Selbst Programmiersprachenkenner sollten das ein oder andere erstaunte Gesicht gemacht haben. Angefangen von den essenziellen Ruby-Klassen wie Fixnum, Float, String und Ranges bis hin zu Array und Hash. Selbst dem Klassenkampf haben Sie sich erfolgreich gestellt, denn Sie wissen nun, wie Sie eigene Klassen erzeugen können. Sie haben von den Formalien bei der Namensgebung, von Variablen, Konstanten und Symbolen erfahren, und Sie wissen auch, was Module, Mixins und Namensräume sind. Sie können zudem Exceptions abfangen und Dateien öffnen und speichern.

Ich finde, das ist schon eine ganze Menge. Jedenfalls mehr als genug, um damit tolle Webanwendungen mit Ruby on Rails zu schreiben. Und nun verspreche ich Ihnen, dass es für den Rest des Buches nur noch um Rails gehen wird – so zum Beispiel im nächsten Kapitel, welches Ihnen die wichtige Frage beantwortet, was denn Rails eigentlich genau ist.

Einsteigen, bitte

In diesem Kapitel:

- Ruby on Rails ist ...
- Ruby und Rails auf Fach-schienesisch
- Ein Blick ins Kochbuch
- Meine Verehrung, eure Exzellenz
- Zusammenfassung

Auf den zurückliegenden Seiten haben Sie sich mit den Charakteristiken und sprachlichen Grundlagen von Ruby vertraut machen können. Sie sind also nun in der Lage, eigene Ruby-Programme zu schreiben. Dazu zählen nicht nur Anwendungen, die auf Ihrem lokalen Rechner laufen. Nein, Ruby hat gerade durch seine Fähigkeiten auf einem Webserver einen gewaltigen Popularitätsschub erhalten. Und das liegt wiederum hauptsächlich an *Ruby on Rails*.

Da ich annehme, dass Sie dieses Buch nicht wegen seines dramatischen Plots oder seiner philosophischen Weisheiten das Leben betreffend lesen, sondern an der Entwicklung von modernen Webanwendungen interessiert sind, haben Sie vielleicht schon eine Ahnung, was denn Ruby on Rails eigentlich ist.

Die nächsten Seiten sollen aus Ihrer Ahnung Gewissheit machen und Sie gleichzeitig mit allerlei wichtigen Hintergrundinformationen versorgen. Sie werden einige Begrifflichkeiten aus der Rails-Welt kennen lernen und anschließend Ihre erste kleine Webapplikation mit Ruby on Rails schreiben.

Schon die Vorstellung der einzelnen Bestandteile und grundlegenden Konzepte des Rails-Frameworks wird Sie sicher begeistern. Ruby on Rails ist nicht wegen seines schönen Namens so beliebt, sondern weil es eine Vielzahl cleverer Ideen und komfortabler Werkzeuge in sich vereint, die vor allem dem Entwickler zu ausufernder Freude während des Entwicklungsprozesses gereichen.

Viele der folgenden Eigenschaften von Ruby on Rails werden Sie übrigens noch in diesem Buch ausführlich im praktischen Einsatz kennen lernen. Daher grämen Sie sich nicht, wenn Sie nachfolgend nicht alles hundertprozentig verstehen.

Ruby on Rails ist ...

... zunächst einmal großartig. In zweiter Linie ist es ein mächtiges und dennoch einfach zu verwendendes Framework für die Entwicklung von dynamischen Websites und Anwendungen im Internetbrowser auf Basis von Ruby. Was ist ein Framework? Da hilft, wie so oft, die direkte Übersetzung weiter: *Framework* = Rahmen = Gerüst = ein Gebilde, das die Grundstruktur vorgibt und vom Programmierer erwartet, selbige mit Code zu füllen, der auf den Vorgaben aufbaut.

In dem Moment, in dem Sie beginnen, eine Rails-Anwendung zu schreiben, funktioniert sie bereits. Sie bringt zwar, Framework sei Dank, alles mit, um eine Anwendung im Browser zum Laufen zu bringen, dennoch kann sie noch nichts. Aber ihr das beizubringen ist ja schließlich auch Ihr Job, den Rails umfangreich und mit allerlei Hilfsmitteln sehr zuvorkommend unterstützt.

Mit Rails können Sie alle Arten von Webapplikationen oder Internetseiten erstellen. Technisch setzt lediglich der Browser eine Grenze. Besonders, wenn Datenbanken im Spiel sind, entfälltet Ruby on Rails seine ganze Stärke und ermöglicht einfachsten Code selbst für die komplexesten Datenbankabfragen.

Rails ist außerdem hervorragend für die Entwicklung von State-of-the-Art-Applikationen im Web geeignet, die auf *Ajax*, *Tagging*, *Web Services*, externen *APIs* und all dem anderen modernen technischen Kokolores des Web 2.0 basieren. Denn Rails hat alles dafür Notwendige an Bord.

Und wenn das nicht reichen sollte, kann Ruby on Rails durch *Plugins* und *Bibliotheken* erweitert werden, die es haufenweise im Internet gibt. Einige Adressen von öffentlich zugänglichen Sammelstellen können Sie im Anhang dieses Buches finden.

Dennoch bleibt festzuhalten: Mit Ruby und Rails können Sie letztlich nichts machen, was mit der Konkurrenz, sei es eine andere Programmiersprache oder ein anderes Framework, nicht auch machen könnten. Das klingt zwar ernüchternd, ist es aber gar nicht. Denn die Art und Weise, wie Sie dieses Ziel erreichen, ist einzigartig. Schnell, unkompliziert, direkt. Der Weg ist das Ziel.

Wer hat's erfunden?

Ruby on Rails wurde von *David Heinemeier Hansson* entwickelt und 2004 zum ersten Mal der skeptischen Öffentlichkeit präsentiert. Damals wie heute war es Basis für die webbasierte Projektmanagementsoftware *Basecamp* (<http://www.basecamphq.com>), einen Vorreiter in Sachen Web 2.0.

Und ob Sie es glauben oder nicht: Der eben genannte Rails-Erzeuger, übrigens Kind des nach Ansicht des Autors dieses Buches glanzvollen Jahrgangs 1979, entstammt dem im Web bislang dezent im Hintergrund agierenden Land Dänemark. Betrachtet man den rasanten Aufstieg von Ruby on Rails im Internet von der belächelten

Spielerei mit einer unbekannten Sprache zum revolutionären Kult-Framework, so muss David Heinemeier Hansson zukünftig wohl in einem Atemzug mit Dänemarks großen Köpfen wie Niels Bohr, Hans-Christian Andersen, Carl Nielsen und Ebbe Sand genannt werden.



David Heinemeier Hansson ist übrigens sehr engagiert in der Verbreitung und Weiterentwicklung von Ruby on Rails. Überzeugen Sie sich selbst und besuchen Sie doch einmal sein Blog *Loud Thinking*: <http://www.loudthinking.com>. Dort können Sie auch seinen 2005 gewonnenen und von Google und O'Reilly gestifteten Award *Best Hacker Of The Year 2005* bewundern, den er für Ruby on Rails erhalten hat.

Dieses Buch behandelt also die wohl tollste japanisch-dänische Koproduktion aller Zeiten. Doch Rails wäre auch dann ein Meilenstein in der Entwicklung von Webanwendungen, wenn Ruby und Rails aus anderen Ländern stammen würden.

Maßgeblicher Grund für die rasche Verbreitung von Rails war sicher auch die Quellenoffenheit des Frameworks. Es war von Beginn an Open Source und steht unter der *MIT-Lizenz*, einem der freisten Lizenzmodelle, die es gibt. Mit der von Apple angekündigten Integration von Ruby on Rails in das Betriebssystem MacOS 10.5 wird der endgültige Durchbruch vollzogen werden. Denn ein solcher Akt ist Heilsprechung und Ritterschlag in Einem.

Warum eigentlich Rails?

Vielleicht ist Ihnen ja auch schon einmal die Freude zuteil geworden, auf dem Bahnhof eines kleinen, romantisch-verträumten Dörfchens zu stehen und der frohen Hoffnung zu sein, dass der letzte Woche noch dampfbetriebene Bummelzug in den nächsten drei bis vier Stunden eintrifft und sie in die nächste Stadt mitnimmt. Und vielleicht war es eine Strecke, die nicht nur regionale, sondern auch superschnelle Fernzüge nutzen. Sollte also ein *ICE*, *TGV*, *ICN* oder *Cisalpino* mit einem Affenzahn und ordentlich Krach an Ihnen vorbeigedonnert sein, so haben Sie doch bestimmt gedacht: »Mensch, was für eine Wahnsinngeschwindigkeit!« (Falls Sie das noch nicht erlebt haben, sei Ihnen gesagt, dass einem eine solche Vorbeifahrt gehörig Respekt einflößt.)

Logisch, Hochgeschwindigkeitszüge rollen durchs Land, weil ihre Passagiere so schnell wie möglich ans Ziel kommen wollen. Die Züge sind in der Lage, die erforderliche hohe Geschwindigkeit zu erzielen, weil sie auf vorgefertigten Wegen, den *Schienen*, fahren. Das hat auch zur Folge, dass man beispielsweise im Ersatzteilkatalog eines ICES ergebnislos nach einem Lenkrad sucht. Die Schiene ist der Weg.

Während Busfahrer ab und an einen verzweifelten Blick in einen Stadtplan werfen und slalomgleich durch die Innenstadt manövriieren müssen und sich letztlich doch

verfahren, könnten *Lokführer* während der Ausübung ihres Jobs eigentlich entspannt die Augen schließen. Sehen Sie *Busfahrer* einfach als Synonym für all die Programmierer, die ohne Rails entwickeln und ab und an Umwege fahren. Und Sie können sich ruhig schon einmal Lokführer nennen.

Somit halten wir fest: Rails steht für ein schnelles Erreichen des Ziels durch die Nutzung von festen, vordefinierten Wegen in kürzester Zeit. Mit Rails werden Ihre Ideen in Windeseile zur realen Webapplikation.

Doch nicht nur die vorgefertigen Wege sind es, die einen Hochgeschwindigkeitszug so stark beschleunigen können. Auch die Tatsache, dass viele Streckenkilometer extra für ICE, TGV und Co. gebaut worden sind und exklusiv von ihnen genutzt werden dürfen, sorgt für Tempo.

Und so ist es auch bei Rails: Das Framework wurde einzig und allein für den Einsatz im Web konzipiert und programmiert. Es ist für die Bedürfnisse eines Webentwicklers optimiert. Sie können übrigens ganz beruhigt sein: Anders als bei zu den geschotterten Schienen musste für den Bau der virtuellen bislang kein fröhlich quakendes Feuchtbiotop oder tirilierendes Vogelschutzgebiet platt gemacht werden.

Noch etwas ist der Grund für das hohe Tempo der Schienenflitzer: Sie halten einfach nicht an jedem romantischen Dorfbahnhof, sondern nur in größeren Städten. Denn das dauernde Anhalten und Anfahren würde schließlich viel zu viel Zeit kosten. So ist das auch bei Ruby on Rails: Sie brauchen nicht dauernd anzuhalten bei der Entwicklung Ihrer Anwendungen und zu überlegen, wie Sie dieses oder jenes machen. Durch die konsistente Verwendung von Ruby als Sprache für fast alles, was Sie mit Rails entwickeln, können Sie Ihre Webapplikation in einem Atemzug durchprogrammieren. Und da Ruby eine sehr durchdachte und einfach zu verwendende, intuitive Sprache ist, die Ihnen genügend Freiräume lässt und viel Arbeit abnimmt, werden Sie bei Ruby on Rails schlicht nicht gezwungen, auf die Entwicklungsbremse zu treten.

Ruby und Rails auf Fachschiensesisch

Wie darf man sich nun einen solchen vorgefertigten, exklusiven und optimierten Weg ohne Zwischenstopps konkret vorstellen? Um diese Frage zu beantworten, lassen Sie uns einen genaueren Blick auf die wichtigsten Produktivitätsbeschleuniger von Rails werfen. Sie werden dabei merken, dass sie alle eine mehr oder weniger engagierte Mitarbeit des Programmierers erfordern.

Konvention über Konfiguration

Wenn Sie schon einmal Webapplikationen mit anderen Programmiersprachen entwickelt haben, so kennen Sie bestimmt den Aufwand, den man allein schon mit der Bestückung diverser Konfigurationsdateien hat. Besonders aufwändig ist das bei-

spielsweise bei Datenbankzugriffen. Mit Ruby on Rails können Sie diesen Aufwand enorm reduzieren. Allerdings nur, wenn Sie bereit sind, sich an die von Rails angebotenen Konventionen zu halten.



Natürlich gibt es auch bei der Entwicklung einer Rails-Applikation einen minimalen Konfigurationsbedarf. Die nötigen Dateien werden Ihnen aber von Rails von Beginn an zur Verfügung gestellt und mit Standardwerten befüllt. In ihnen legen Sie beispielsweise die Parameter für den Zugriff auf Ihre Datenbank fest.

Es gibt wirklich nur wenige Regeln, die man auch als Rails-Neuling recht schnell verinnerlicht und spätestens ab der dritten Applikation schon völlig unbewusst nutzt. Auf das lästige Anlegen von überflüssigen Konfigurationsdateien werden Sie also bald sehr gern verzichten können.

Doch nicht nur einige Konfigurationsparameter sind Ruby on Rails zum Opfer gefallen. Auch während des Entwicklungsprozesses selbst ermöglichen Rails' Konventionen, dass Sie viel weniger Code schreiben müssen. Rails kennt die Bedürfnisse (und Abneigungen) von Webentwicklern und bietet bei Beachtung der Regeln viel Funktionalität mit wenig Code.

Die rasche Entwicklung von Anwendungen auf Basis einiger simpler Regeln nennt man fachsprachlich auch *agile Softwareentwicklung*.

Das Model-View-Controller-Pattern

Das MVC-Pattern (Model View Controller) ist keine Erfindung von Rails. Und doch wäre Rails wohl nur halb so toll ohne dieses Programmierschema. Es wurde bereits 1979 (wie gesagt, ein großartiges Jahr) von Trygve Mikkjel Heyerdahl Reenskaug entwickelt. Nein, kein Däne – ein Norweger.

Und ein sehr interessanter noch dazu. In seiner langen Laufbahn hat er an vielem mitgewirkt, was noch heute die Softwareentwicklung bestimmt. Auch an der Entwicklung der Theorien zur objektorientierten Programmierung, dem Herz von Ruby, war er beteiligt. Sie können einige seiner Aufsätze auf seiner persönlichen Website <http://heim.ifi.uio.no/~trygver/> nachlesen. Norwegisch müssen Sie dafür nicht beherrschen, das meiste ist auf Englisch hinterlegt.

Das MVC-Entwurfsmuster besagt, dass die Entwicklung einer Software mit grafischer Benutzeroberfläche in drei voneinander relativ unabhängigen, getrennten Schichten stattfinden soll.

Das *Model* stellt das Herz einer Anwendung dar. Es soll sich hauptsächlich um die Beschaffung, Bereitstellung, Verarbeitung und Speicherung von Daten kümmern. Oft werden diese Fähigkeiten auch als *Geschäftslogik* bezeichnet. Das Model stellt also Funktionalitäten zur Verfügung, die ausschließlich den eigentlichen Zweck einer Anwendung abbilden. Die optische Darstellung gehört nicht zum Model.

Diese erfolgt in der *Präsentationslogik*, die in einem *View* untergebracht wird. Hier steht alles, was mit der Darstellung der dem Model entstammenden Daten zu tun hat. Ein View stellt zudem Elemente zur Bedienung der Software zur Verfügung. Diese können Aktionen auslösen, die das Model auffordern, neue Daten zu liefern oder im View eingegebene Daten zu speichern. In Ruby on Rails wird die View-Schicht durch *Templates* realisiert, die auf *HTML*, *JavaScript* oder *XML* basieren und Ruby-Code enthalten.

Die Koordination zwischen Model und View übernimmt der *Controller*. Er enthält die *Programmsteuerungslogik* einer Software. Der Controller-Teil fordert die Daten des Models an und leitet sie, gegebenenfalls adäquat aufbereitet, an einen View weiter, der sie darstellt. Der Controller empfängt zudem Daten beispielsweise aus einem Formular des Views und leitet sie kompetent an den Model-Teil weiter. Ein Controller hat zwar keinen blassen Schimmer, was genau in Model und View vorgeht, weiß aber sehr wohl, wie er beide korrekt anzusprechen hat. Model und View hingegen wissen gar nicht, dass es den jeweils anderen gibt. Auch von der Funktionsweise des Controllers haben sie keine Ahnung. Aber das ist auch gar nicht nötig und sollte sogar tunlichst vermieden werden.

Durch diese strikte Trennung der Programmteile entsteht Quelltext, der sehr leicht gepflegt und erweitert werden kann. Zudem zwingt er einen Programmierer besonders in komplexen Projekten zu einer gewissen Ordnung. Das erleichtert die Weitergabe des Quellcodes an bei der Entwicklung Unbeteiligte.

Ein kleines Beispiel. Ein Kunde loggt sich in den Kundenbereich eines Onlineshops ein. Er möchte seine dort hinterlegte Adresse als logische Konsequenz des gestrigen Transports dutzender Umzugskartons bearbeiten. Er gibt den URL des Shops ein und klickt auf den Bearbeiten-Button seiner Mein-Konto-Seite. Von diesem Knöpfchendrücken bekommt der Controller Wind. Er kontaktiert das Model und weist es an, die aktuellen Adressdaten herauszugeben. Das Model blättert in der Datenbank und gibt ein Objekt zurück, dass alle nötigen Daten enthält. Der Controller nimmt das Objekt entgegen und bestückt einen View mit den Adressdaten, in diesem Fall eine HTML-Datei mit entsprechendem Formular, die eigens dafür geschaffen wurde, eine Adresse zu ändern. Dann gibt der Controller den View an den Webbrowser zurück, der sie darstellt. Nun gibt der Benutzer seine neue Adresse ein. Nehmen wir an, er vergisst dabei die Postleitzahl. Klickt er auf den Senden-Button, werden die neuen Angaben an den Controller überspielt, der sie umgehend an die Speicher-Funktion des Models weitergibt. Das Model speichert die neuen Daten aber nicht, sondern gibt eine Fehlermeldung an den Controller zurück. Dieser schnappt sich einen View, der diese Fehlermeldung anzeigt. Und so weiter, und so weiter.

Die strikte Implementierung des Model-View-Controller-Patterns in Rails sorgt insbesondere bei Internetagenturen für heitere Stimmung. Immer wieder hört man von nie dagewesener Harmonie zwischen Webprogrammierern und -designern des

Hause. Hatte man sich vor Ruby on Rails so manches Mal in der Verachtung des jeweils anderen überboten, so kann der Webdesigner nun ganz entspannt die Views in herkömmlicher Art und Weise mit HTML erstellen und mit ganz wenig schnell erlerntem Ruby-Code Stellen definieren, an denen später die gewünschten Daten erscheinen sollen. Sie werden in Kürze sehen, wie das funktioniert.

Übrigens: In der Model-Schicht kann es mehrere Modelle geben, beispielsweise, wenn mehrere Datenbanktabellen berücksichtigt werden sollen. Gleiches gilt auch für Controller und Views. Allerdings muss nicht jedes Model einen Controller haben und nicht jeder Controller ein Model.

Das DRY-Prinzip

Ausgeschrieben bedeutet die Abkürzung *DRY Don't Repeat Yourself* – wiederhole dich nicht. Das besagt, dass jede Information nur einmal in einem Softwareprojekt vorkommen soll. Redundante Informationen sollen dadurch vermieden werden. Sie sind ein immer wieder auftauchendes Ärgernis bei der Softwareentwicklung, egal mit welcher Sprache.

Wenn eine Information beispielsweise in zwei Variablen vorkommt, der Quelltext sie aber nur in einer der beiden Variablen verändert, sind Programmfehler unweigerlich die Folge.

Ruby und Ruby on Rails bieten allerlei Techniken an, damit Sie als Entwickler das DRY-Prinzip sehr leicht berücksichtigen können. Das fängt bei Symbolen und globalen Konstanten an und endet darin, dass Sie Rails nicht explizit mitteilen müssen, über welche Spalten eine Datenbanktabelle verfügt. Da die Spaltenbezeichner ohnehin in der Datenbank enthalten sind, brauchen Sie sie nicht noch einmal anzugeben.

Die Tatsache, dass Rails ein *Full-Stack-Framework* ist, bildet die Grundlage für das konsequente Einhalten des DRY-Prinzips. Full Stack bedeutet, dass jede Information, die in einem Teil des Programms (Model, View oder Controller) erzeugt wird, in den anderen verfügbar ist.

Wunderwerk objektrelationales Mapping

Ruby on Rails kommt mit eingebautem *objektrelationalem Mapping* (ORM) zu Ihnen. Auch diese Technologie ist keine Rails-Erfundung, steigert aber die Effizienz beim Programmieren datenbankgestützter Webanwendungen enorm.

Objektrelationales Mapping bezeichnet ein Prinzip, wie relationale Datenbanken zu Objekten geformt werden können. Wie Sie sicher wissen, sind viele Datenbanksysteme, wie zum Beispiel MySQL, in Tabellen angeordnet. Jeder Datensatz ist dabei eine Tabellenzeile. Eine Tabelle ist zudem in Spalten aufgeteilt, wobei jede einzelne eine bestimmte Information eines Datensatzes aufnehmen kann.

Möchte nun ein Programm, das in einer objektorientierten Sprache geschrieben ist, auf diese nicht objektorientierte Datenbanktabelle zugreifen, ergibt sich ein Problem. Wie um alles in der Welt bekommt man die Struktur und die Daten der Tabelle in ein Objekt mit Methoden und Eigenschaften?

Viele Programmierer machen sich in einem solchen Augenblick daran, ganze Programmteile zu schreiben, die nur auf diese konkrete Tabelle spezialisiert sind und nur sie in ein oder mehrere Objekte verwandeln können. Das ist aber hochgradig ineffizient und muss für jede Tabelle aufs Neue durchgeführt werden. Änderungen in der Struktur der Tabelle bedeuten somit auch Änderungen im Quelltext, der sie für das objektorientierte Programmieren zugänglich macht.

ORM geht einen allgemeineren Weg und stellt Funktionalitäten zur Verfügung, die ein oder mehrere Objekte aus jeder relationalen Datenbank, egal welcher Beschaffenheit, formen. Das ORM von Ruby on Rails geht da noch einen Schritt weiter und bietet Ihnen ein fertiges Modell an, das das DRY- und das Konvention-über-Konfiguration-Prinzip beherzigt und alles beinhaltet, was Sie für den objektorientierten Zugang zur Datenbank benötigen.

Dazu stellt Ihnen Rails eine Klasse zur Verfügung, die so heißt wie die abzubildende Tabelle – allerdings in der Einzahl. Aus einer Tabelle namens dogs zaubert Ihnen Ruby on Rails eine Klasse namens Dog. Jede Instanz dieser Klasse, die Sie erzeugen, bildet eine bestimmte Zeile der Datenbanktabelle ab. Über Akzessoren, die genauso heißen wie die einzelnen Spalten der Tabelle, können Sie so einen Datensatz der Tabelle zielgerichtet auslesen oder verändern.

Meta-Programmierung

Diese Stärke von Ruby on Rails haben Sie auch schon im 2. Kapitel kennen gelernt. Erinnern Sie sich noch an die Methoden attr_reader, attr_writer und attr_accessor? Mit diesen Methoden konnten Sie sich das Schreiben diverser *Setter*- und *Getter*-Methoden sparen. Das hat Ruby für Sie übernommen. Oder anders ausgedrückt: Sie haben Code geschrieben, der Ruby anwies, Code zu schreiben – wenn auch nur virtuell. Das ist *Meta-Programmierung*. Und davon bietet Ruby on Rails reichlich.

Übrigens: Sollte Sie das an Makros erinnern, dann liegen Sie goldrichtig. Über ein solches Makro können Sie beispielsweise mit wenigen Zeichen Assoziationen zwischen mehreren Datenbanktabellen herstellen.

Was ohne Meta-Programmierung recht komplex werden könnte, geht mit in Ruby ganz einfach: Nehmen wir an, Sie entwickeln eine Filmdatenbank, die die Tabellen *Filme*, *Schauspieler* und *Verleihe* beinhaltet. Mit Rails' Meta-Programmierung können Sie nun einfach schreiben: Ein Film hat mehrere Schauspieler. Ein Schauspieler spielt in mehreren Filmen mit. Und: Jeder Film hat genau einen Verleih.

Mit diesen *Assoziationen* kann eine Rails-Anwendung nun ganz leicht alle Filme eines Schauspielers oder alle Filme eines Verleihs, in denen bestimmte Schauspieler mitspielen, herausfinden – und zwar ohne dass Sie sich einen Kopf über den Quelltext machen müssen, der das bewerkstellt.

Active Record

Die eben beschriebene Funktionalität ist Bestandteil von *Active Record*. Auch Rails' ORM ist hier untergebracht. Active Record ist eines von mehreren *Sub-Frameworks* in Ruby on Rails. Ein Sub-Framework ist ein eigenständiges Framework, das gut und gerne auch allein und unabhängig von Ruby on Rails genutzt werden kann.

In Active Record ist auch definiert, wie Ruby on Rails intern auf spezielle Datenbanksysteme zugreifen kann. Mit Rails schreiben Sie grundsätzlich Applikationen, die unabhängig von der verwendeten Datenbank arbeiten. Active Record sorgt durch systemspezifische Implementierungen dafür, dass Ihre Anwendung mit *MySQL*, *PostgreSQL*, *SQLite*, *DB2*, *Firebird*, *Oracle*, *SQL Server*, *Sybase* und weiteren Systemen läuft.

Ebenfalls in Active Record enthalten sind Validatoren. Auch hier kommt Meta-Programmierung zum Einsatz. So können Sie mit nur einer Zeile festlegen, dass Rails vor dem Speichern von Daten in der Datenbank kontrollieren soll, ob auch dieses oder jenes Feld durch den Benutzer ausgefüllt wurde, ob dieses oder jenes Feld mindestens x Zeichen lang ist und vieles mehr.

Action View

Mit *Action View* bringt Rails ein weiteres Sub-Framework mit; dieses wird allerdings in der Präsentationslogik eingesetzt. Es ermöglicht Ihnen, mit der serverseitigen Sprache Ruby clientseitigen Code zu schreiben. Dieser wird durch Action View vor der Auslieferung an den Browser in JavaScript umgewandelt.

So können Sie zum Beispiel grafische Effekte oder Ajax-gestützte Formulare realisieren. JavaScript-Kenntnisse sind zwar nie verkehrt – aber in diesem Fall nicht unbedingt nötig. Ihr Kopf braucht also nicht zwischen zwei Programmiersprachen hin- und herzuwechseln, was meistens eh schief geht. Stattdessen können Sie mit Ruby ungebremst Ihrem Ziel entgegensteuern und dennoch clientseitig programmieren.

Dabei werden Sie in Action View von weiteren Methoden unterstützt, mit deren Hilfe Sie Webformulare generieren oder Inhalte mit wenigen Zeilen Code paginieren können. Kurz: In Action View steckt alles, was Sie zum Erstellen der View-Schicht Ihrer Anwendung benötigen.

Action Controller

Noch ein Sub-Framework. In *Action Controller* steckt all das drin, was Sie befähigt, eine anständige Controller-Schicht für Ihre Rails-Anwendungen zu programmieren.

Ein wichtiger Teil von Action Controller arbeitet quasi unbemerkt hinter den Kulissen. Dort werden als URL formulierte Anfragen, fachsprachlich *Requests* genannt, die an den mit Ruby on Rails bestückten Webserver herangetragen werden, interpretiert und ausgewertet. Dabei löst Action Controller die URL auf und filtert alle wertvollen Informationen, die für den weiteren Ablauf der Rails-Applikation wichtig sind, heraus. Dieser Teil von Action Controller wird übrigens *Router* oder *Dispatcher* genannt.

Als Ergebnis der Auswertung wird eine *Action* ausgeführt. Actions sind nichts anderes als öffentliche Methoden einer Klasse, die von einer in Action Controller enthaltenen Klasse erbt. Es ist Ihre Aufgabe, diese Actions zu formulieren.

Indem Sie Kindklassen mit den Actions schreiben, entwickeln Sie die komplette Controller-Schicht Ihrer Rails-Anwendung. Alle in einer Controller-Klasse benutzten Instanzvariablen stehen Ihnen auch im View zur Verfügung.

Weitere Sub-Frameworks

Active Support bringt Methoden ins Spiel, mit denen Ihr Leben als Rails-Entwickler noch einfacher wird, die grundsätzlich aber auch ohne Rails ganz gut zu gebrauchen sind. So können Sie mit Active Support beispielsweise Zahlen in Geldbeträge mit zwei Nachkommastellen umwandeln oder problemlos Minuten, Stunden und Sekunden addieren.

Mit *Action Mailer* können Sie in Windeseile eine E-Mail in Ihrer Rails-Anwendung generieren und verschicken. *Action Web Service* ermöglicht Ihnen, eine zeitgemäße API zum Benutzen Ihrer Anwendung oder ihrer Daten in *Mashups* zu erstellen.

ERb – EmbeddedRuby

Keine Sorge, *EmbeddedRuby*, auch *eRuby* oder *ERb* genannt, ist kein eigenständiger Ruby-Dialekt, den Sie speziell lernen müssen.

Mit EmbeddedRuby ist grundsätzlich der Quelltext gemeint, den Sie innerhalb einer Ruby-fremden Umgebung einbetten. Sie werden EmbeddedRuby oft begegnen, wenn Sie Ihre hauptsächlich in HTML geschriebenen Templates erstellen. An die Stellen, an denen Informationen des Controllers eingefügt werden sollen, setzen Sie einfach ein bisschen Ruby-Code, der dies übernimmt. EmbeddedRuby ist essenziell für die Erstellung von HTML-Views.

Wenn Sie schon einmal mit *PHP* oder *ASP* entwickelt haben, kennen Sie das bereits: Der Code, der vor dem Anzeigen des Templates ausgeführt werden soll, wird zwischen speziellen Begrenzern notiert. Fügen Sie *PHP*-Code beispielsweise zwischen <?php und ?> ein, so ist es bei *Ruby on Rails* <% und %>.

Scaffolding

Wir kommen nun zu etwas, was Sie spätestens im nächsten Kapitel, also im praktischen Einsatz, verblüffen wird, und worum *Rails* am meisten beneidet wird. Es ist eine geniale Idee, die die logische Konsequenz der *Rails*-Maxime ist, rasend schnell zu einer modernen Webapplikation zu kommen: *Scaffolding*.

Sollten Sie ein schon oft (leid-)geprüfter Webentwickler sein, dann wissen Sie sicher, dass sich bei vielen Projekten das Grundmuster ähnelt. Dies trifft insbesondere dann zu, wenn eine Webanwendung Datensätze bearbeiten muss – sei es im Front- oder Backend. Immer wieder ist es doch dasselbe System, das programmiert werden muss: Ein Datensatz muss *neu angelegt*, *bearbeitet* und *gelöscht* werden können. Und eine *Gesamtübersicht* aller Datensätze ist meist ebenfalls notwendig.

Eine solche Funktionalität nennt man übrigens *CRUD*. Das steht für *Create*, *Read*, *Update* and *Delete*. Und da viele Webanwendungen in irgendeiner Form auf dieser Basis beruhen, ist es doch nur logisch, wenn einem so produktiven Framework wie *Ruby on Rails* eine Technologie beiliegt, die diesen immer wiederkehrenden Aufwand minimiert.

Genau das leistet das in *Rails* eingebaute Scaffolding. Damit erzeugen Sie eine *CRUD*-Applikation mit einer Zeile. Scaffolding erzeugt für Sie alle nötigen Actions auf der Controller-Ebene sowie alle Views mit Formularen und übersichtlichen Tabellen. Das Ergebnis ist eine voll funktionsfähige Anwendung.

Es gibt übrigens zwei Arten von Scaffolding. Bei der ersten ist *Rails*' fantastische Metaprogrammierung am Werk. Sie wird meist genutzt, um Anwendungsprototypen zu erstellen, bei denen es nur auf Grundfunktionen ankommt, die Scaffolding liefern kann.

Bei der zweiten schreibt Ihnen *Rails* den nötigen Quelltext direkt in Ihre Dateien. Im Gegensatz zur Metaprogrammierung können Sie den Code also richtig sehen und auch bearbeiten. Der Quelltext, der dabei entsteht, ist eine hervorragende Ausgangsbasis für Ihren eigenen Code.

Generatoren

Die eben beschriebene zweite Variante des Scaffoldings verdanken Sie der Existenz eines entsprechenden *Generators*.

Generatoren erzeugen selbstständig Ruby-Code, auf den Sie durch Hinzufügen eigenen Codes aufbauen können. Neue Modelle und neue Controller beispielsweise

legen Sie mit Generatoren an. In beiden Fällen sorgen Generatoren dafür, dass alle notwendigen Dateien erzeugt werden, und zwar da, wo das Rails-Framework sie auch laut Konvention (und nicht Konfiguration) erwartet. Darüber hinaus befüllen Controller- und Model-Generatoren die neuen Quelldateien automatisch mit einem leeren Klassenkonstrukt, so dass Sie sich überhaupt nicht um Namen und Abstammung Ihrer Controller-Klasse kümmern müssen, sondern sofort mit dem Coden der wichtigen Dinge loslegen können.

Die Rails-Console

Rails bringt auch eine *Console* mit. Sie funktioniert so ähnlich wie *Interactive Ruby*, ist aber darauf spezialisiert, interaktiv mit einer bestimmten Rails-Anwendung zusammenzuarbeiten. Mit der Rails-Console können Sie direkt in das Geschehen einer Anwendung eingreifen und beispielsweise manuell Actions eines Controllers auslösen oder neue Datensätze anlegen, etwa zum Erzeugen eines Grunddatenbestands.

Environments

Rails-Anwendungen können in drei unterschiedlichen Umgebungen laufen, die *Environments* genannt werden. Jede Umgebung bringt spezielle Funktionen mit, die maßgeblich den Zweck einer Umgebung berücksichtigen.

Zunächst werden Rails-Anwendungen in der *Development*-Umgebung entwickelt. Hier werden durch Rails auch die kleinsten Fehler geloggt und diverse Debug-Funktionalitäten angeboten. Das verlangsamt natürlich die Ausführungsgeschwindigkeit einer Rails-Anwendung. Doch die kann getrost vernachlässigt werden, wenn auf der anderen Seite wertvolle Informationen über Fehler der Anwendung und Mittel zu deren Beseitigung zur Verfügung gestellt werden. In der zweiten Umgebung, *Test*, können Sie Tests entwickeln, die Ihre Rails-Anwendung durch das Nachstellen von möglichen Einsatzszenarien auf Fehler überprüfen, etwa, ob eine Berechnung auch das zu erwartende Ergebnis ausgibt.

Die dritte Umgebung, *Production*, ist schließlich für den produktiven Einsatz gedacht. Hier fehlen Entwicklungs- und Testwerkzeuge. Aber die werden ja auch nicht mehr benötigt. Sobald Ihre Anwendung fertig ist und veröffentlicht werden kann, lassen Sie sie einfach in dieser Umgebung laufen.

Ajax und Web 2.0

Rails-Anwendungen sind durch die Integration der JavaScript-Bibliotheken *Prototype* und *Script.aculo.us* bestens gerüstet für das Erstellen von Anwendungen für Web 2.0. Mit Leichtigkeit können Sie so Ajax-basierte Oberflächen erstellen, die extrem interaktiv, reaktionsschnell und intuitiv sind und durch die Verwendung

von außergewöhnlichen Steuerelementen und optischen Effekten auffallen. Das Besondere: Auch wenn beide Bibliotheken JavaScript-basiert sind, so können Sie diese in Rails mit Ruby-Code steuern. Sie wissen ja... Hochgeschwindigkeitszug ohne Anhalten.

Ajax und Rails, das ist von Anfang an eine gute Verbindung gewesen. Rails war das erste Framework, das Ajax und somit die Fähigkeit, Daten vom Webserver nachträglich zu holen und in eine bestehende Webseite zu setzen oder anderweitig clientseitig zu verarbeiten, integrierte. Von Beginn an wurde dies durch die enge, nahtlose Verbindung mit Prototype realisiert.



Die beiden Bibliotheken Prototype (<http://www.prototypejs.org>) und Script.aculo.us (<http://script.aculo.us>) gibt es übrigens auch als Stand-alone-Varianten. So können Sie die beiden Bibliotheken beispielsweise auch in PHP-basierten Projekten nutzen. Allerdings müssen Sie dabei leider auf Rails' komfortable und einfache Art der Nutzung von Prototype und Script.aculo.us verzichten. Das bedeutet auch, dass Sie dabei im Gegensatz zu Rails mit mindestens zwei Programmiersprachen arbeiten müssen: PHP für die Server- und JavaScript für die Clientseite.

Vielleicht haben Sie schon einmal Ajax eingesetzt und sich mit *XMLHttpRequest*-Objekten, *Callbacks* und *Status-Codes* herumgeschlagen. Das gibt's bei Rails natürlich auch – allerdings erst, wenn Sie das wollen. Bis dahin schreiben Sie einfach Ruby-Code so, als ob es nichts Besonderes wäre.

Mit *Script.aculo.us* können Sie die Oberfläche Ihrer Rails-Anwendung um benutzerfreundliche Steuerelemente ergänzen, beispielsweise Slider, sortierbare Listen, Auto-Vervollständigen- und In-Place-Eingabefelder und Drag-and-drop-Elemente. Wie einfach das geht, werden Sie in diesem Buch noch erleben.

Ferner können Sie aus einem großen Schatz an visuellen Effekten schöpfen. Die werden beispielsweise dann wichtig, wenn Sie vorhaben, die Benutzerfreundlichkeit Ihrer Ajax-haltigen Rails-Anwendung zu optimieren. Grafische Effekte können einen Veränderung auf der Benutzeroberfläche signalisieren, die der Benutzer auf Grund Ajax' unauffälliger Arbeit im Hintergrund sonst nicht mitbekommen würde.

Wir halten fest: Webanwendungen, die im wahrsten Sinne des Wortes Anwendungen sind, können Sie mit Rails durch die gute Integration leistungsfähiger Ajax-Bibliotheken enorm schnell erstellen. Eben noch Idee, jetzt schon Applikation. Das hat aber auch seine Nachteile, betrachtet man all die pilzgleich empor sprießenden Anwendungen, die als Ewig-Beta-Versionen die Leitungen des Web 2.0 verstopfen und deren Businessplan auf der Hoffnung fußt, dass Google mal einen dicken Scheck schickt. Aber das ist ein ganz anderes Thema.

Und noch vieles mehr ...

Es gibt noch viele weitere große und kleine Dinge, die in einer Vorstellung des Rails-Frameworks aufgezählt werden könnten. Aber uns soll das an dieser Stelle vorerst genügen.

Blicken wir nun lieber auf Ruby on Rails im praktischen Einsatz. Dazu muss eine bereits fertige Anwendung als Anschauungsmaterial herhalten. Mit InstantRails haben Sie zwei davon bekommen: Ein Weblog und ein Kochbuch.

Ein Blick ins Kochbuch

Das Kochbuch soll helfen, Ihren Hunger nach einer realen Rails-Anwendung zu stillen. Um es zum Laufen zu bringen, starten Sie zunächst InstantRails. Sollte es sich bei Ihnen noch immer in C:\InstantRails befinden, so geben Sie bei *Start → Ausführen* C:\InstantRails\InstantRails.exe ein.



Sollten Sie kein InstantRails benutzen, weil Sie beispielsweise unter Linux entwickeln, können Sie sich die kleine Applikation auch aus dem Internet herunterladen. Auf O'Reillys Open Source-Webplattform *ONLamp* finden Sie nicht nur die Anwendung als ZIP-Datei, sondern auch gleich noch eine ausführliche Erläuterung, wie Sie diese Anwendung selbst nachbauen können. Die Adresse: <http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html>.

Achten Sie nach dem Start von InstantRails darauf, dass die beiden Ampeln innerhalb des Panels irgendwann auf grün schalten. Erst dann sind Apache und MySQL einsatzbereit. Und erst dann können Sie mit Rails auf die Datenbank zugreifen.

Klicken Sie dann auf den Button mit dem InstantRails-Symbol, der sich direkt neben dem Apache-Button befindet. Über *Configure → Rails Applications → Manage Rails Applications...* gelangen Sie in ein Fenster, das Ihnen auf der linken Seite alle bislang zur Verfügung stehenden Rails-Anwendungen auflistet.

Versehen Sie hier den Eintrag *cookbook* mit einem Häkchen. Klicken Sie dann auf *Start with Mongrel*.

Daraufhin sollte sich ein Fenster öffnen, das Ihnen Meldungen von Mongrel zeigt. Hier können Sie den Start des kleinen Rails-Servers genau verfolgen. Warten Sie, bis er meldet, *available* zu sein. Zugleich verrät er Ihnen eine Adresse, unter der er das ist, beispielsweise 0.0.0.0:3001.



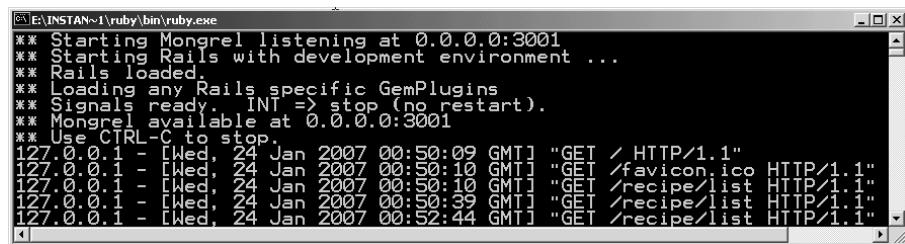
Abbildung 3-1: Hier werden später auch Ihre Anwendungen aufgelistet sein!

Mongrel

Mongrel ist ein *HTTP Application Server*, der grundsätzlich nicht viel kann. Man könnte auch sagen, er ist schlank. Er nimmt HTTP-Anfragen eines Clients entgegen, verarbeitet sie, leitet sie weiter an die dazugehörige Anwendung und schickt deren Ausgaben an den Client zurück. Er kann genau das, was Ruby-Entwickler benötigen, deren Anwendungen im Web laufen sollen. Und das gut, einigermaßen schnell und absturzsicher. Attribute, die die Alternativen der Zeit vor Mongrel nicht von sich behaupten konnten.

WEBrick, ein vom Grundsatz her ähnliches Projekt, das schon lange mit Rails ausgeliefert wurde und noch immer wird, interessierte sich nicht für so etwas Banales wie Zeit während der Bearbeitung von Requests. Kamen dann noch mehrere Requests zur gleichen Zeit, war der Zeitvorteil der Rails-Entwicklung fast schon dahin. Andere Ansätze, etwa die Nutzung der FastCGI-Schnittstelle eines herkömmlichen Webservers waren zwar schneller, aber sie erwiesen sich oft als Glücksspiel und waren zudem schwer zu konfigurieren. Der hauptsächlich in Ruby geschriebene Application Server interpretiert alle Anfragen auf HTTP-Ebene und wandelt sie nicht erst wie FastCGI in eine serverseitig vorhandene Sprache um.

Mongrel ist praktisch mit dem Installieren sofort einsatzbereit. Einfach mit Ruby-Gems anfordern, fertig. Der schlanke Application Server ist ideal zum Entwickeln von Rails-Entwicklungen auf dem lokalen Rechner. Aber auch im produktiven Einsatz kann er genutzt werden und dank seiner Fähigkeit, in Clustern zu arbeiten, wird er auch immer interessanter für große Websites.



```
E:\INSTAN~1\ruby\bin\ruby.exe
** Starting Mongrel listening at 0.0.0.0:3001
** Starting Rails with development environment ...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready. INT => stop (no restart).
** Mongrel available at 0.0.0.0:3001
** Use CTRL-C to stop.
127.0.0.1 - [Wed, 24 Jan 2007 00:50:09 GMT] "GET / HTTP/1.1"
127.0.0.1 - [Wed, 24 Jan 2007 00:50:10 GMT] "GET /favicon.ico HTTP/1.1"
127.0.0.1 - [Wed, 24 Jan 2007 00:50:10 GMT] "GET /recipe/list HTTP/1.1"
127.0.0.1 - [Wed, 24 Jan 2007 00:50:39 GMT] "GET /recipe/list HTTP/1.1"
127.0.0.1 - [Wed, 24 Jan 2007 00:52:44 GMT] "GET /recipe/list HTTP/1.1"
```

Abbildung 3-2: Mongrel startet und erhält die ersten Requests

Beachten Sie, dass *0.0.0.0* als IP-Adresse natürlich völlig unbrauchbar ist. Mongrel möchte Ihnen damit nur sagen, dass die Cookbook-Applikation unter jeder lokalen IP-Adresse erreichbar ist, solange der Port, in diesem Fall *3001*, stimmt. Ersetzen Sie am besten *0.0.0.0* durch *127.0.0.1* oder auch, als zufällig gewähltes Beispiel, *127.111.22.3* und geben Sie die komplette Adresse inklusive Port in die Adresszeile des Browsers Ihrer Wahl ein.

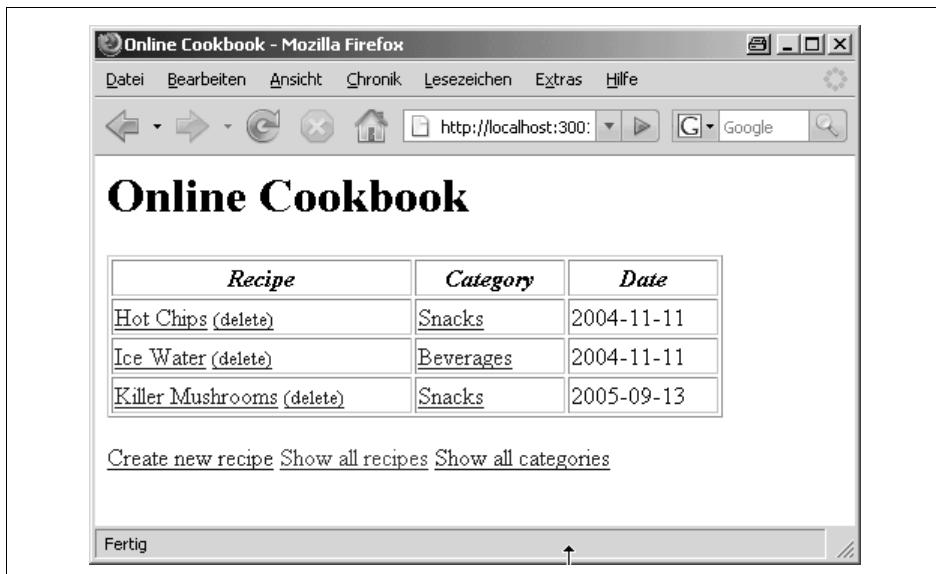


Abbildung 3-3: Schon einmal Killerpilze zubereitet?

Sogleich erhalten Sie eine Übersicht über alle enthaltenen Rezepte. Nun sollten Sie etwa 5 bis 10 Sekunden lang den seltsamen Geschmack des Autors der Cookbook-Applikation gehörig missbilligen. Klicken Sie anschließend auf *Ice Water* – das ist mit Abstand noch das Leckerste und Bekömmlichste.

Der Dispatcher bei der Arbeit

Werfen Sie nun einen Blick auf die Adresszeile Ihres Browsers. Da stehen interessante Sachen drin. Hinter `http://127.0.0.1:3001/recipe/show/2` steckt aber nur bedingt eine Webadresse, obwohl es wie eine aussieht. Alles, was nach `http://127.0.0.1:3001` kommt, sind vielmehr Anweisungen an eine Rails-Anwendung. Konkret soll hier die Action `show` des Controllers `recipe` den Datensatz mit der ID 2 anzeigen.

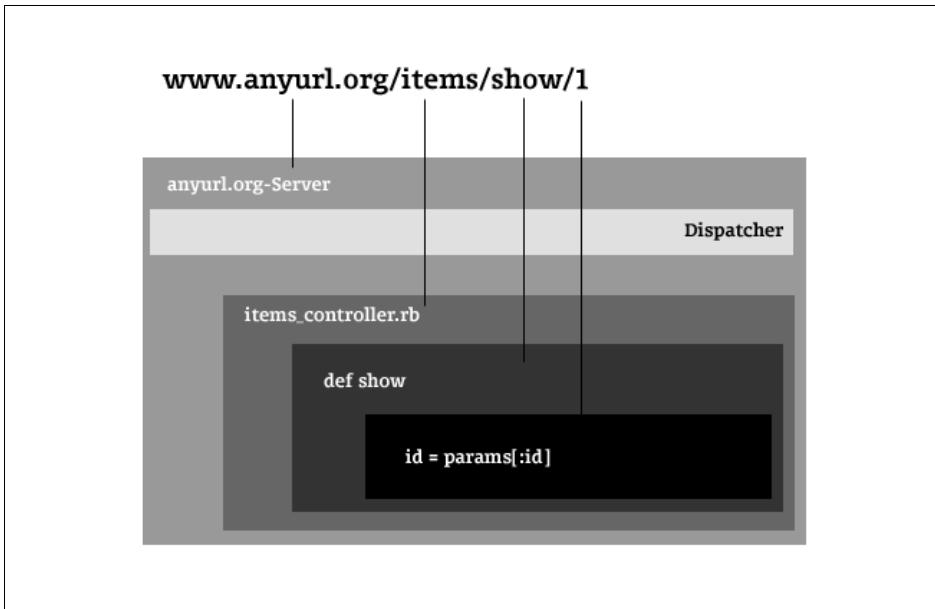


Abbildung 3-4: Der Dispatcher oder Router löst eine Anfrage nach einem Schema auf

Die Form `http://<Basis-URL>/<Controller>/<Action des Controllers>/<ID des zu verarbeitenden Datensatzes>` ist essenziell. Nur so kann der Router oder Dispatcher, von dem weiter oben die Rede war, eine Anfrage an die Rails-Anwendung verstehen und den Programmablauf entsprechend steuern.

Die Angabe einer Datensatz-ID ist dabei natürlich optional. Schließlich benötigt nicht jede Action eines Controllers diese Information. Wenn Sie beispielsweise auf `Show all recipes` klicken, erkennen Sie, dass die Action `list` selbstredend keine ID benötigt, zeigt sie doch alle Datensätze an.



Natürlich können Sie einer Action auch mehr Informationen als nur eine Datensatz-ID übermitteln. Die entsprechenden Parameter können durch ein Fragezeichen getrennt als Bezeichner/Wert-Paare an den Request angehängt werden. Zum Beispiel `http://127.0.0.1/recipe/show/2?pic=false`. Diese Form, Query-String genannt, sollte Ihnen aber als Webtägler bereits geläufig sein.

Klicken Sie ruhig durch die Anwendung. Dabei werden Sie möglicherweise feststellen, dass dies eine typische CRUD-Applikation ist.

Fast Food dank Scaffolding

Der Großteil der Cookbook-Anwendung wurde durch Scaffolding erzeugt. Das ist nur folgerichtig, schließlich soll man mit Cookbook Rezepte *erstellen*, *anzeigen*, *bearbeiten* und *löschen* können.

Der Autor der Anwendung hat zwar ein paar strukturelle und optische Änderungen an dem durch Scaffolding erzeugten Code vorgenommen. Grundsätzlich sehen Sie aber noch vieles, was unangetastet blieb.

Sicher werden Sie feststellen, dass das Kochbuch optisch nun wahrlich kein Leckerbissen ist. Aber genau das ist typisch für Scaffolding: Es ist eine Möglichkeit, in Sekunden ein CRUD-System auf die Beine zu stellen. Und so ist es möglich, eine Anwendung, die eben noch Idee war, zumindest funktionell in die Tat umzusetzen.

Stellen Sie sich vor, Sie treten mit kolibrieskem Puls in das Zimmer eines potenziellen Auftraggebers, der Sie unter dem Vorwand in sein Mahagoni-Büro gelockt hat, einen gigantischen Auftrag für ein neues Webportal vergeben zu wollen. Klar, den wollen Sie haben. Aber die Konkurrenz auch. Sie unterhalten sich über den Funktionsumfang, eine Frage hier, eine Antwort da – und am Ende holen Sie Ihr Notebook heraus, hauchen ein »Kleinen Moment, bitte!« in die stickige Luft und präsentieren nach ein, zwei Minuten die erste funktionierende Kernfunktionalität, welche eben noch Gesprächsthema war. Die ist nicht schick, aber beeindruckend.

Sollte Ihr potenzieller Auftraggeber überhaupt noch in der Lage sein, etwas zu sagen, er würde Ihnen sicher mitteilen, dass Sie den Auftrag haben. Das ist Ruby on Rails. Produktivität und Effizienz ist alles.

Die Verzeichnisstruktur

Verlassen wir das gedachte Büro und blicken wir lieber auf die Verzeichnisstruktur der Anwendung. Oder besser: Auf die Verzeichnisstruktur jeder Anwendung, denn so sehen Verzeichnisse und Dateien jeder Rails-Applikation aus. Hier macht sich bemerkbar, dass Rails ein Framework ist und dass der Grundsatz »Konvention über Konfiguration« natürlich auch für Verzeichnisse und Dateien gilt. Völlig klar: Für Dateien, die in jeder Rails-Anwendung an der gleichen Stelle stehen, müssen keine Konfigurationsdateien angelegt werden, in denen ihr Aufenthaltsort steht.

Die Cookbook-Anwendung befindet sich in einem Unterverzeichnis von Instant-Rails, *ruby_apps/cookbook*.

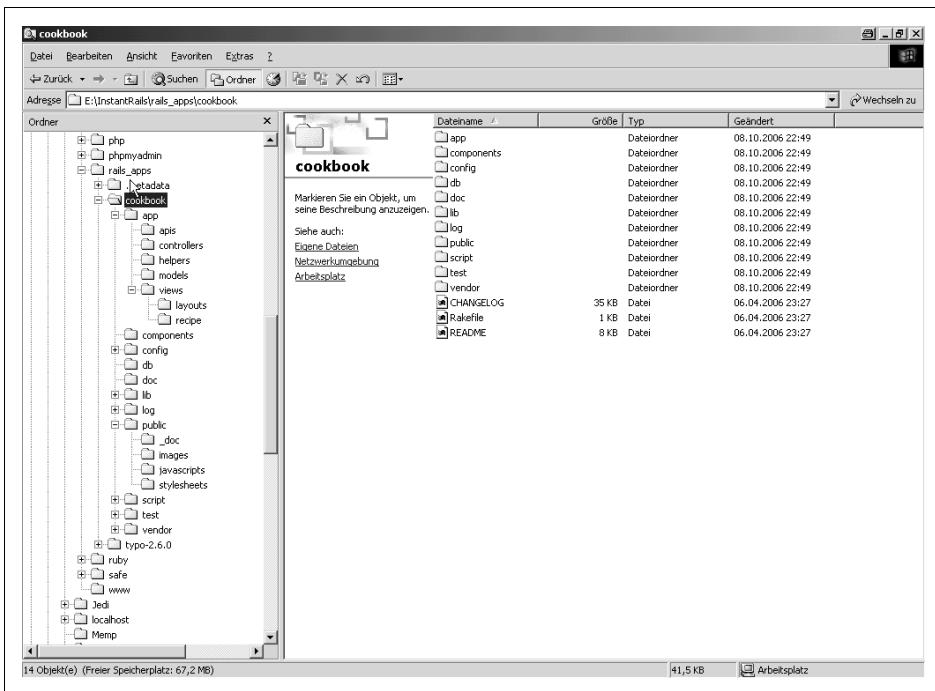


Abbildung 3-5: Verzeichnisstruktur der Cookbook-Anwendung

models, views, controllers

Werfen Sie zunächst einen Blick in den *apps*-Ordner. Hier finden Sie weitere Verzeichnisse, bei denen Sie sicher das Model-View-Controller-Pattern wiedererkennen. Bei jeder Rails-Anwendung müssen Model-Dateien im Verzeichnis *models*, alle Controller-Dateien in *controllers* und alle Views in *views* gespeichert werden. Wie Sie sehen, handelt es sich dabei um ganz normale *.rb*-Dateien.

Das *views*-Verzeichnis enthält zwei weitere Unterordner: *layouts* und *recipe*. Das *layouts*-Verzeichnis enthält meist HTML-Dateien, die als Basis-Template dienen. Das ist praktisch, wenn bei allen oder vielen Unterseiten eines Rails-Projekts der Inhalt zwar dynamisch, die äußere Hülle aber gleich ist. Dann lohnt es sich beispielsweise, eine Datei zu schreiben, die die grundlegende HTML-Struktur inklusive *Head* und *Body* enthält und über einen mit ERb realisierten Platzhalter den Ort für die dynamischen Inhalte festlegt.

Das *recipe*-Verzeichnis beinhaltet Dateien der View-Schicht, welche ausschließlich für den *recipe*-Controller gedacht sind. Scaffolding legt diesen Unterordner automatisch an und generiert zudem alle View-Dateien, welche hier Platz finden. Werfen Sie einen Blick hinein, und Sie werden erkennen, dass sich dort drei Dateien befinden: *new.rhtml*, *edit.rthml* und *list.rhtml*.

Diese Dateien tragen nicht durch Zufall die Namen von Actions des recipe-Controllers. Wenn Sie keine anderslautenden Angaben innerhalb des Controllers machen, sucht Rails automatisch nach einer Datei, die so heißt wie die Action, um sie zu rendern und auszugeben.

Rendern heißt in diesem Fall, dass der Controller den gesamten EmbeddedRuby-Code des Views auswertet. Dadurch gelangen Werte, die im Model erzeugt wurden, via Controller in den View.

Die Dateiendung, hier *.rhtml*, weist Rails übrigens explizit darauf hin, dass es sich bei diesem View um eine HTML-Datei handelt, die Ruby-Code enthält. Rails kann übrigens nicht nur *.rhtml*, sondern auch Ruby-haltige JavaScript- (*.rjs*) und XML-Dateien (*.rxml*) als View rendern. Rails erkennt anhand der Endung, welcher Inhalt in der Datei steckt. Konvention über Konfiguration eben.

Ein bisschen Konfiguration muss sein

Der Konventionengrundsatz gilt natürlich, auch wenn Ihre Augen als Nächstes womöglich ungläubig den *config*-Ordner erblicken. Ja, ganz ohne Konfigurationsdateien geht es nun einmal nicht. Beispielsweise muss eine Verbindung zum Datenbankserver hergestellt werden können. Und das geht eben nur mir konkreten Zugangsdaten, die irgendwo hinterlegt werden müssen – hier in *database.yml*.

In *environment.rb* können Sie bestimmen, in welcher Umgebung Ihre Anwendung laufen soll. Außerdem ist dort Platz für Ihre eventuellen Konfigurationsbedürfnisse. Und die dürfen Sie natürlich auch weiterhin haben. Schließlich muss man auch mal einen Pfad zentral festlegen können, unter dem beispielsweise Dateien gespeichert werden sollen.

Rails' ver-log-endes Angebot

Im *log*-Verzeichnis finden Sie möglicherweise zum jetzigen Zeitpunkt noch keine verwertbaren Daten. Wenn Sie jedoch den Mongrel-Server stoppen, erscheint hier eine Textdatei namens *development.txt*. Hier steht alles drin, was während der Ausführung der Cookbook-Anwendung passiert ist.

Stoppen Sie den Mongrel-Server, indem Sie in der Windows-Eingabeaufforderung, die Ihnen schon wertvolle Informationen zum Start von Mongrel ausgegeben hat, einfach STRG+C drücken. Die Cookbook-Anwendung ist nun nicht mehr erreichbar, doch wurde die angekündigte Log-Datei geschrieben. Schauen Sie ruhig mal in *development.txt* hinein. Das geübte Auge erkennt sofort, dass diverse Anfrage an die MySQL-Datenbank via *Data Definition Language (DDL)* gestellt worden sind. Zum Beispiel: `SELECT * FROM recipes WHERE (recipes.id = '2')`. Die beruhigende Nachricht an dieser Stelle: So etwas brauchen Sie als Rails-Entwickler niemals (mehr) zu schreiben. Es sei denn, Sie bestehen darauf. Grundsätzlich formuliert Rails so etwas aber selbst.

In der Log-Datei können Sie auch erkennen, wie der Dispatcher gearbeitet hat. Aus <http://127.0.0.1:3001/recipes/show/2> formte dieser beispielsweise den Hash `{"action"=>"show", "id"=>"2", "controller"=>"recipe"}`. So ist's recht.

Sie können in der Log-Datei sogar erkennen, welches Layout genutzt wurde und wie lang das Bearbeiten der Anfrage gedauert hat. Bedenken Sie: Diese Informationen bekommen Sie ganz automatisch und nebenbei. Rails ist echt nett. Bei Projekten ohne Rails hätte man sich so einen Log-Mechanismus erst selbst bauen oder umständlich von Drittanbietern besorgen und anpassen müssen.

Das public-Verzeichnis

Ein sehr wichtiges Verzeichnis ist *public*. Und dieser Ordner macht seinem Namen alle Ehre: Alles was hier drinsteckt, kann vom Web aus erreicht werden. Alle anderen Verzeichnisse bleiben Rails vorbehalten und können nicht via Browser aufgerufen werden. Das *public*-Verzeichnis ist praktisch das Wurzelverzeichnis Ihrer auf Rails basierenden Website.

Daher eignet sich *public* prima für alles, was wie JavaScript-Dateien, Bilder und Stylesheets nun einmal zwangsläufig öffentlich sein muss, und für alles, was nicht von Rails bearbeitet werden muss, beispielsweise eine *robots.txt* oder statische HTML-Dateien wie etwa Fehlerseiten.

Rails kopiert bei der Generierung der Verzeichnisstruktur automatisch alle von Rails benötigten JavaScript- und Bilddateien in das *public*-Verzeichnis. Daher werden Sie bei einem Blick in das *javascripts*-Verzeichnis *prototype.js* und alle *Script.aculo.us*-Dateien vorfinden – auch wenn die Anwendung diese Funktionalität gar nicht benötigt. Aber das kann ja Rails am Anfang gar nicht wissen.

Wie dieser *Anfang* eigentlich aussieht, schauen wir uns nun an. Denn es ist an der Zeit, dass Sie Ihre erste Rails-Anwendung selbst schreiben. Einer alten, ziemlich einfallslosen Tradition folgend, die schon unsere Urahnen gepflegt hätten, wäre der Computer nicht erst tausende Jahre später erfunden worden, müsste jetzt eigentlich eine *Hello-Welt-Applikation* folgen. Aber da auch Rails nicht viel von Tradition hält, sondern neue, eigene Maßstäbe setzt, machen wir mal etwas leicht anderes.

Meine Verehrung, eure Exzellenz

Was auch immer Ihre erste Rails-Anwendung von sich geben wird, eines steht fest: Sie werden nun eine ganze Reihe an Dingen, von denen Sie eben nur gelesen haben, bereits bei dieser einfachen Applikation praktisch erleben. Und Sie werden merken, wie einfach all das ist, was möglicherweise auf den vergangenen Seiten etwas kompliziert klang. Eine Anwendung, die ohne Datenbank auskommt, soll uns dabei zunächst reichen. Im nächsten Kapitel erstellen Sie zwei Anwendungen, die auf eine Datenbankanbindung allerdings nicht verzichten können.

Lassen Sie uns ein paar Vorbereitungen treffen. Stellen Sie sicher, dass der Mongrel-Server der Cookbook-Applikation nicht mehr aktiv ist. Öffnen Sie anschließend RadRails.

An die Tasten!

Die Schritte, die Sie jetzt durchführen werden, sind obligatorisch für jede Rails-Anwendung. Stellen Sie zunächst sicher, dass RadRails für die Erstellung einer Rails-Anwendung eingestellt ist. Dies ist dann der Fall, wenn der Rails-Button, welcher sich oben rechts im Programmfenster befindet, aktiviert ist. Alternativ können Sie dies über *Windows → Open Perspective...* bewerkstelligen.

Klicken Sie auf *File → New...* und entscheiden Sie sich jetzt für *Rails Project*. Sie finden diesen Eintrag im Rails-Ordner der Baumansicht.

Wählen Sie einen Projektnamen. Der sollte am besten den Regeln gehorchen, die auch für Variablen in Ruby gelten. Unser Beispiel soll hello heißen. Bevor Sie auf *Finish* klicken, sollten Sie noch dafür sorgen, dass *Generate Rails application skeleton* angeklickt ist. Dadurch erhalten Sie automatisch all die Dateien und Verzeichnisse, die wir eben noch bei der Cookbook-Anwendung bewundert haben. Es gibt eigentlich kein Grund, warum diese Option nicht angeklickt werden sollte.

Auf den *WEBrick-Server*, der Ihnen standardmäßig noch angeboten wird, können Sie allerdings sehr wohl verzichten. WEBrick ist ein Ruby-Server, so wie auch Mongrel einer ist – allerdings ist Mongrel der bessere. Daher: *Create a WEBrick server deaktivieren*, *Create a Mongrel server* aktivieren.

Nach einem Klick auf *Finish* beginnt die Magie. In der RadRails-Console können Sie nun verfolgen, welche Dateien für Sie erstellt werden.



Das Generieren der Verzeichnisstruktur und das Bereitstellen diverser Dateien sind keine Funktionen von RadRails. RadRails führt für Sie entsprechende Aktionen des Rails-Frameworks nur aus. Wie Sie das Rails-Grundgerüst auch ohne RadRails erzeugen und alle anderen Generatoren und sonstigen Tools von Rails bedienen können, erfahren Sie im Anhang. Grundsätzlich kann RadRails aber alles, was Sie brauchen.

Im linken Teil von RadRails können Sie nun die hello-Applikation mit all ihren Dateien und Verzeichnissen sehen. Die Anwendung funktioniert schon. Probieren Sie es aus! Sie könnten dazu unser Projekt wieder über das InstantRails-Panel starten, so, wie Sie es bereits bei der Cookbook-Anwendung getan haben. Aber auch aus RadRails heraus geht das.

Neben *Console* finden Sie einen Tab namens *Server*. Hier wird die Mongrel-Server-Instanz angezeigt, die Sie beim Erstellen des Projekts angefordert haben. Klicken Sie

den Eintrag an und drücken Sie den *Start*-Button, der Ihnen als grüner Pfeil zur Verfügung steht. Nach kurzer Zeit wechselt der Status auf *Started*. Öffnen Sie dann ein Browserfenster und geben Sie als Adresse `http://127.0.0.1:3000` ein.

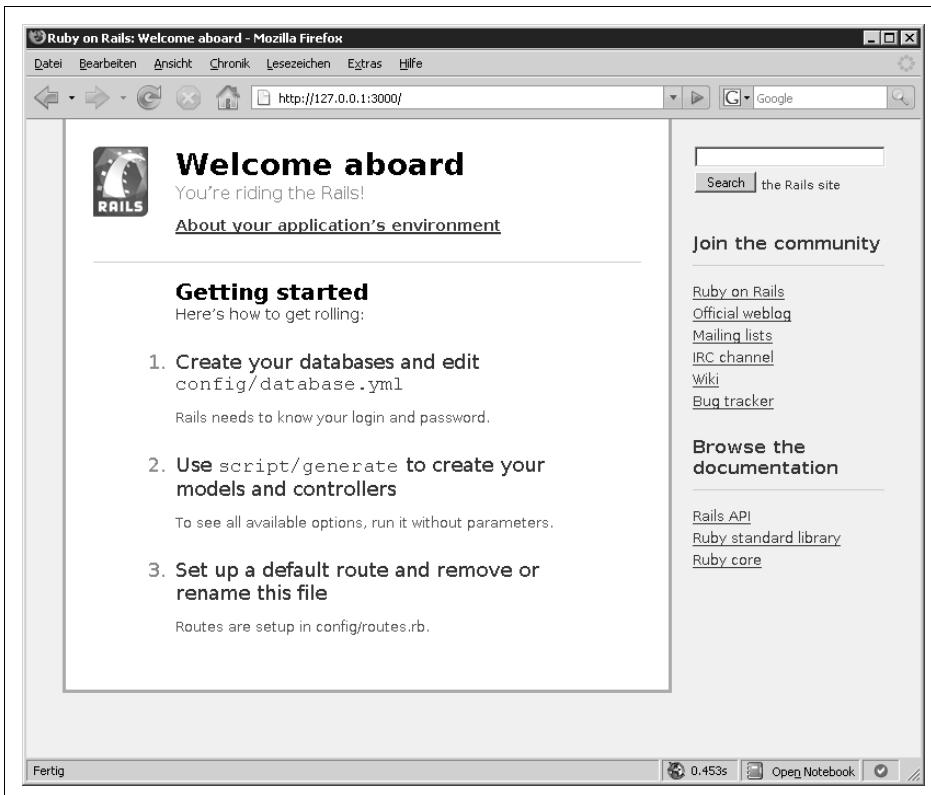


Abbildung 3-6: Ihre erste Rails-Anwendung läuft!

Nun muss das Grundgerüst der Anwendung mit Funktionalität gefüllt werden. An dieser Stelle sollten Sie stets überlegen, was die Anwendung eigentlich leisten soll, und wie Sie das in Model, View und Controller aufteilen können.

Unsere hello-Anwendung soll eine nette Begrüßung im Browserfenster anzeigen. Die Erzeugung dieser Phrase ist eindeutig ein Fall für die Model-Schicht, da dort stets Informationen erzeugt werden. Die Ausgabe der freundlichen Worte erfolgt natürlich in der View-Schicht. Die Aufgabe, die im Model erzeugte Begrüßung im View zur Ansicht zu bringen, erledigt die Controller-Schicht.

Beginnen Sie am besten zunächst mit der Model-Schicht. Sie können ein Model erzeugen, wenn Sie in RadRails von der *Server*- zur *Generators*-Ansicht wechseln. Ein paar Tabs weiter.

Wählen Sie hier aus der linken Combobox *model* und geben Sie in der rechten den Namen des Models ein. Nennen Sie das Model *greeting*. Nutzen Sie bei der Vergabe von Model-Namen am besten stets die Einzahlform und englische Begriffe. Dies gehört zu den Rails-Konventionen. Welchen tieferen Sinn das hat, erfahren Sie in Kürze. Darüber hinaus gelten die Vorschriften für Variablenbezeichner. Klicken Sie nun auf *Go*. Alle nötigen Dateien für Ihr *greeting*-Model werden nun generiert.



Auch wenn die Bezeichnung *Generator* das nicht vermuten lässt, können Sie hier bestehende Models und Controller auch wieder entfernen. Wählen Sie dazu in der Generators-Ansicht die Option *Destroy*.

Nachdem das Model generiert wurde, finden Sie eine neue Datei in Ihrem Projektbaum. Unter *hello/app/models* finden Sie nun die Datei *greeting.rb*. Öffnen Sie diese. Sie werden einen leeren Klassenkörper finden. Hier platzieren wir nun eine Methode, die eine Begrüßung generiert und als Rückgabewert ausgibt. Denken Sie hier daran, dass Sie eine ganz normale Ruby-Klasse vor sich haben. Fügen Sie folgende Methode in den Klassenkörper ein.

Beispiel 3-1: Greeting#create gibt eine zufällige Begrüßung aus

```
def create
  srand
  part1 = ['Hallo', 'Meine Verehrung', 'Tach'].sort_by{rand}[0]
  part2 = ['Liebster', 'Eure Exzellenz', 'gottgleiches Wesen'].sort_by{rand}[0]
  part1 + ', ' + part2 + '!'
end
```

In dieser Methode werden zwei zufällig ausgewählte Array-Elemente zu einer Begrüßungsformel zusammengesetzt. Auch hier gilt: Der Rückgabewert der Methode ist der zuletzt ausgewertete Ausdruck.

Damit wäre unser Model schon fertig. Eine kleine Änderung müssen wir allerdings noch vornehmen. Rails hat Ihnen die Basis eines klassischen Rails-Models generiert. Selbiges erfordert zwingend eine Datenbankanbindung. Da wir die in diesem Projekt nicht benötigen, entfernen wir einfach die entsprechende Funktionalität aus unserem Model. Dazu lassen wir unsere Model-Klasse *Greeting* nicht von *ActiveRecord::Base* erben, sondern von *Object*. Und da das Erben von *Object* nicht expliziert notiert werden muss, sondern von Ruby standardmäßig angenommen wird, reicht *class Greeting* als Klassenkopf völlig aus.

Lassen Sie uns nun den Controller erzeugen. Denn ohne ihn ist selbst das schönste Model wertlos, da es nichts zum Anzeigen gibt. Wählen Sie in der Generator-Ansicht diesmal in der linken Combobox den Wert *controller*. Vergeben Sie als Bezeichner *greetings*. Ja genau, die Mehrzahl des Modelbezeichners. Das ist hier zwar noch nicht unbedingt nötig, wird in späteren Projekten aber wichtig. Daher halten wir uns schon jetzt an diese Konvention.

In *app/controllers* liegt jetzt *greetings_controller.rb*. Öffnen Sie die Datei und fügen Sie die Methode show ein, die in diesem Fall eine Action des Controllers greetings ist.

Beispiel 3-2: Dem Controller wird eine Action hinzugefügt

```
def show  
  @greeting = Greeting.new  
end
```

In dieser Methode wird mit new eine Instanz unserer Model-Klasse erzeugt und einer Instanzvariablen des Controllers übergeben. Auf @greeting können Sie innerhalb des Views zugreifen. Sie können der Variablen übrigens auch einen anderen Namen geben. Dennoch empfiehlt es sich aus Gründen der Übersichtlichkeit, einen Model-referenzierenden Bezeichner zu wählen.

Sie können schon jetzt die show-Action des Controllers in Ihrem Browser aufrufen. Wie Sie wissen, erfolgt dies durch einen Browser-Request an den Rails-Server in der Form /<controller>/<action>.



Vergessen Sie nie, Ihre bearbeiteten Projektdateien zu speichern, bevor Sie sich Ihr Werk im Browser ansehen.

Wenn Sie <http://127.0.0.1:3000/greetings/show> aufrufen, werden Sie allerdings eher unhöflich begrüßt. Mit einer Fehlermeldung. Kein Grund zur Traurigkeit – im Gegenteil: 1. Sie wissen nun, wie sich Rails bemerkbar macht, wenn ein Fehler beim Ausführen Ihrer Rails-Anwendung auftritt. Beachten Sie die große Menge an Informationen, die Ihnen Rails hierbei zur Verfügung stellt. 2. Sie erfahren so, dass noch kein View für die Action show existiert. Lassen Sie uns das jetzt nachholen.

Es soll ein kleiner HTML-Schnipsel sein, wobei wir hier einfach mal auf alle die nötigen Formalien einer HTML-Datei verzichten. Die Begrüßungsbotschaft soll als Überschrift h1 angezeigt werden. Das Template enthält zudem einen statischen Text, der in einem Absatz-Element p sitzt.

Navigieren Sie in Ihrem Projektbaum zu *app/views/greetings*. Hier befinden sich alle View-Dateien, die der greetings-Controller benötigt. Beziehungsweise, hier werden Sie sich befinden. Denn Sie müssen die View-Datei noch anlegen. Da sie explizit für die Action show gedacht ist, erzeugen Sie mit einem Rechtsklick und *New → File* eine Datei namens *show.rhtml*. Platzieren Sie dort folgenden Code:

Beispiel 3-3: show.rhtml zeigt die Begrüßung und einen Text an

```
<h1><% @greeting.create %></h1>  
<p>Gratulation zur ersten Rails-Anwendung!</p>
```

Sie erkennen sicher den EmbeddedRuby-Teil. Hier wird durch `@greetings` die Instanz unserer Model-Klasse, die wir im Controller erzeugt haben, referenziert und die Methode `create` aufgerufen.

Aktualisieren Sie nun den Inhalt Ihres Browsers. Sie werden sehen, dass die Fehlermeldung verschwindet und der eben erstellte View erscheint. Allerdings fehlt das Wichtigste – die Begrüßungsphrase.

Das liegt daran, dass es zwei Arten gibt, EmbeddedRuby-Code zu notieren. In der Form `<% #Code %>` wird der Code lediglich ausgeführt. Möchten Sie jedoch, dass das Ergebnis der Code-Auswertung an dieser Stelle auch ausgegeben wird, notieren Sie EmbeddedRuby-Code als `<%= #Code %>`. Beachten Sie das Gleichheitszeichen beim öffnenden ERb-Tag. Korrigieren Sie unseren View nun.

Beispiel 3-4: Das Ergebnis von `@greetings.create` wird ausgegeben

```
<h1><%= @greeting.create %></h1>
<p>Gratulation zur ersten Rails-Anwendung!</p>
```

Unsere erste Rails-Anwendung ist fertig. Aktualisieren Sie den Browserinhalt erneut, und Sie werden mit einer freundlichen Begrüßung für Ihre Mühen belohnt. Wenn Sie die Quelltextansicht Ihres Browsers aktivieren, erkennen Sie sofort das Template `show.rhtml` wieder. Nur der ERb-Teil wurde ersetzt.

Zugegeben, diese Anwendung im MVC-Schema zu erstellen ist etwas übertrieben. Wichtig war hierbei nur, dass Sie die grundlegende Vorgehensweise bei der Erstellung einer Rails-Applikation kennen lernen.

Zusammenfassung

Sie sind jetzt mittendrin in der Faszination Ruby on Rails. Sie kennen nun diverse Fachbegriffe und all die Konzepte, die Rails so großartig machen. Einiges davon haben Sie sogar schon live erlebt. Doch das war nur der Anfang. Im nächsten Kapitel geht es so richtig los. Sie werden eine komplette Photoblog-Software mit Rails erstellen, datenbankgestützt natürlich.

Dafür brauchen wir natürlich noch ein paar Fotos. Am besten, Sie schnappen sich Ihre Digitalkamera, gehen einfach mal raus, knipsen, was das Zeug hält, und lassen das Gelesene dabei ein bisschen sacken. Versuchen Sie sich zu vergegenwärtigen, was das Model-View-Controller-Pattern besagt, welche Vorteile das Prinzip »Konvention über Konfiguration« besagt und was Meta-Programmierung ist. Wenn Sie diese Begriffe verinnerlicht haben, sind Sie auf dem besten Weg, zu einem begehrten Rails-Entwickler zu werden.

Ein Photoblog mit Rails

In diesem Kapitel:

- Photoblog? Ähm...
- Die Datei environment.rb
- Beitrags-Model
- Datenbank mit Migrationshintergrund
- PostsController, übernehmen Sie!
- Rails' Kurzzeitgedächtnis: flash
- Models erweitern
- Bessere Views durch Partials
- Navigationsmenü
- LoginGenerator: Zutritt nur bedingt gestattet

Wenn Sie diese Zeilen lesen, sind Sie bereits stolzer Besitzer eines profunden Basiswissens in Sachen *Ruby* und *Rails*. Das wäre zumindest sehr wünschenswert, denn in diesem Kapitel wird beides dringend benötigt. Ihre erste eigene Rails-Applikation drängt schließlich darauf, entwickelt zu werden.

In diesem Kapitel erleben Sie den Geburtsvorgang einer kompletten Anwendung. Es wird eine Photoblog-Software sein, in der viele Funktionen des Rails-Frameworks zum Einsatz kommen. Sie wird auf den Namen *Picsblog* hören. Schritt für Schritt werden Sie dabei erleben, wie Sie *Partials*, *Form-* und *Tag-Helper*, *Migrations*, *Generatoren*, *Assoziationen* und viele andere Zauberwörter einsetzen können. Schon bald werden Sie merken, wie Ihr möglicherweise noch großer Respekt vor diesen kompliziert klingenden Begriffen schnell schwinden wird. Das gelingt natürlich am besten, wenn Sie sich dieses Buch neben Ihren Rechner legen, diesen mit Strom versorgen und ohne Scheu einfach mitmachen. Ich möchte Sie jedenfalls unbedingt dazu ermutigen, auch wenn Sie vielleicht nicht vorhaben, ein bebildertes Tagebuch ins Netz zu stellen.

Denn erstens kann sich das ganz schnell ändern, zweitens erlernen Sie Rails wie so vieles am besten im Praxiseinsatz und drittens soll es auch um Themen gehen, die nicht nur für *Picsblog* wichtig und für die Entwicklung von Webanwendungen mit Ruby on Rails ganz nützlich sein können. Ich möchte Ihnen auch ein paar Tipps geben, wie Sie bei der Entwicklung grundsätzlich vorgehen können oder wie Sie die Benutzeroberfläche mit HTML strukturieren und mit CSS optisch verfeinern.

Wenn Sie sehen möchten, was am Ende dieses Kapitels steht, so fordern Sie Ihren Browser auf, sich mal fix auf die Website zum Buch zu bewegen. Unter <http://www.praxiswissen-ruby-on-rails.de> finden Sie eine Live-Demo von Picsblog.

Photoblog? Ähm...

Betrachtet man den nicht enden wollenden Wortschwall, den so mancher Blogger täglich in sein Online-Tagebuch gießt, so erinnert man sich manchmal wehmütig an die traditionsreiche deutsche Floskel »Ein Bild sagt mehr als tausend Worte«. Und in der Tat, Online-Tagebücher müssen nicht unbedingt nur textlich verfasst sein.

In einem Photoblog liegt der Informationsgehalt, so denn vorhanden, überwiegend in Pixeln vor und eben nicht in Buchstaben. Photoblogger drücken ihre Stimmung oder ein schönes Ereignis durch den Upload eines passenden Fotos aus. Manch einer täglich, manch einer monatlich. Mit Fotos, die mal tonnenschwerem, japanischem Profi-Equipment, mal einem Prepaid-Handy der örtlichen Kaffeerösterfiliale entstammen.

Widmen wir uns der Frage: »Was muss ein Photoblog können?« Lassen Sie uns diese Frage durch »Was soll unsere Photoblog-Software können?« erweitern:

- Upload von Bilddateien
- Chronologisches Anordnen von Fotos
- Anzeigen einer Übersicht aller Fotos mit Thumbnails und Paginierung
- Anzeigen einer Detailseite, die das Foto, seinen Titel und eine nähere Beschreibung enthält
- Anzeigen der fünf neuesten Fotos in der Sidebar
- Kommentarfunktion bei jedem Foto
- Speicherung von Bilddaten und Kommentaren in einer Datenbank
- Administration von Bildern und Kommentaren direkt in der Website
- Administrationsberechtigung durch Eingabe von Benutzername und Passwort

Das sind also die Attraktionen unserer Reise on Rails auf den kommenden Seiten. Bevor wir nun richtig in die Entwicklung unserer Photoblog-Software einsteigen, möchte ich Ihnen noch folgenden Hinweis auf den Weg geben: Jeder Entwickler von Anwendungen hat seinen eigenen Weg, um von der Idee und den Vorabplänen zu einer vollwertigen Anwendung zu kommen. Der hier gezeigte soll für Sie ein Angebot sein, wie man an ein solches Projekt mit Rails herangehen könnte. Vielleicht ist er ja die Basis für Ihren eigenen? Rails selbst schreibt Ihnen jedenfalls keinen konkreten Weg vor, sondern unterstützt Sie lediglich nach Kräften dabei, Ihr Ziel zu erreichen.

Viele Rails-Entwickler arbeiten sich sukzessive an ihr Ziel heran. Sicher gibt es eine Planungsphase, aber sie ist aufgrund der Struktur des Rails-Frameworks bei weitem nicht so wichtig wie bei anderen Programmierumgebungen. Und so möchte ich Ihnen nachfolgend zeigen, wie Sie eine Anwendung Schritt für Schritt bauen und dabei immer wieder die Baustellen wechseln. Die Erweiterung eines Controllers kann eine Änderung am Model nach sich ziehen, was wiederum einen neuen View erforderlich macht.

Ich verspreche Ihnen: Auch wenn es jetzt danach klingt, werden Sie dennoch nicht den Überblick verlieren. Im Gegenteil: Es wird Ihnen sicher viel mehr Freude machen, so zu entwickeln, als vorab ihre Applikation penibelst zu planen und dann erst Model, dann Controller und schließlich View in einem Rutsch zu schreiben.

Der Bilderrahmen

Zu Beginn lassen wir Rails mal Rails sein und machen uns bewusst, dass wir hier für das grafische Internet entwickeln. Und das bedeutet, dass eine wie auch immer geartete Oberfläche gebraucht wird.

Sie hatten im dritten Kapitel bereits einen kurzen Flirt mit *Layouts*. Diese kurze Bekanntschaft bescherte Ihnen das Wissen, dass ein Layout eine Art Basis-View oder ein Grundgerüst ist, in das eine Rails-Anwendung ihre Inhalte platziert. Mehrere Actions diverser Controller können dabei ein und dasselbe Layout nutzen. Layouts unterstützen also das DRY-Prinzip, welches in Rails verankert ist und besagt, dass sich nichts wiederholen sollte, was einmal an einer Stelle in Code gegossen wurde.

Sehen Sie Rails-Layouts zum Beispiel als eine Bühne. Eine solche im Theater wird ja auch nicht bei jedem neuen Akt eines Stücks gesprengt und neu gebaut – wengleich dies die Eintrittspreise erklären würde. Nein, es wird höchstens das Bühnenbild geändert. Und den Rahmen um das schmalzige Portrait Ihres oder Ihrer Liebsten kann man ja beispielsweise auch für ein schönes Katzenfoto nutzen, wenn er oder sie sich gefühlsmäßig grundlegend neu orientiert hat und keiner optischen Huldigung mehr bedarf.

Lassen Sie uns also als erstes eine Bühne bauen, auf der unser aufregendes Rails-Spektakel aufgeführt werden soll. Dabei soll es wirklich nur um die Grundstruktur des Photoblogs gehen. Die optischen Leckerbissen kommen erst ganz zum Schluss, wenn der Prototyp der Software steht.

Das Layout der Seite können Sie auf herkömmliche Art und Weise zimmern. Da es nur aus *HTML* und *CSS* besteht, reicht ein einfacher Texteditor völlig. Ich habe mich für ein zweispaltiges Layout entschieden, das auf Tabellen als Strukturgeber verzichtet und stattdessen mit floatenden *div*-Elementen arbeitet. So erreichen Sie eine rundherum moderne Anwendung, vor und hinter den Kulissen.

Speichern Sie folgenden HTML-Quelltext als *standard.html* zunächst an einem Ort Ihrer Wahl. Später werden Sie sehen, wie Sie daraus RHTML, also Ruby-haltiges Hypertext Markup machen, was den Anforderungen von Picsblog Rechnung trägt.

Beispiel 4-1: Die Picsblog-Bühne

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Picsblog</title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
    <link rel="stylesheet" type="text/css" href="layout.css">
  </head>
  <body>
    <div id="page">
      <div id="header">
        Header
      </div>
      <div id="main">
        <div id="content">
          Content
        </div>
        <div id="sidebar">
          Sidebar
        </div>
      </div>
      <div id="footer">
        Footer
      </div>
    </div>
  </body>
</html>
```

Wie Sie sehen, werden alle Bereiche der Seite von einem div mit der ID page umgeben. Innerhalb dieses Containers kommt zunächst der Seitenkopf header. Hier erscheint später die Navigation. Darunter folgt ein weiterer Container namens main, gefolgt von einem Footer.

Der main-Bereich beherbergt ein content- und ein sidebar-div. Die beiden sollen im Browser nebeneinander erscheinen und zwei Spalten ergeben. Das wiederum erreichen Sie durch die Fähigkeit von HTML-Block-Elementen, einander umfließen zu können. Dass sie das tun sollen, erzielen Sie mit der CSS-Eigenschaft float.

Diese und weitere Festlegungen, welche insbesondere die Breite einzelner Bereiche betreffen, sind Bestandteil der Datei *layout.css*, welche bereits im head-Bereich von *standard.html* verlinkt ist. Die Datei, welche zunächst im gleichen Verzeichnis wie *standard.html* gespeichert wird, hat folgenden Inhalt:

Beispiel 4-2: layout.css

```
#page {  
    width: 800px;  
    margin: 0 auto;  
}  
  
#content {  
    width: 640px;  
    float: left;  
}  
  
#sidebar {  
    width: 160px  
    float: right;  
}  
  
#footer {  
    clear: both;  
}
```

Die gesamte Bühne (page) wird somit 800 Pixel breit und soll mittig im Browser platziert werden. Der 800 Pixel breite Platz wird zu 640 beziehungsweise 160 Pixel unter content und sidebar aufgeteilt. Beachten Sie auch die Eigenschaft clear im Selektor #footer. Durch den Wert both wird sichergestellt, dass der Footer nicht auch noch in die muntere Umfließerei von content und sidebar hineingezogen wird. Das Ergebnis können Sie sich nun in Ihrem Browser anzeigen lassen.



Abbildung 4-1: Das Picsblog-Grundgerüst, zur besseren Veranschaulichung mit großer Schrift und Rahmen

Als nächstes werden wir unser eigenliches Rails-Projekt erstellen und die eben erstellten Dateien in die vom Framework generierte Verzeichnisstruktur einbinden. Dabei werden Sie auch Ihre erste Zeile *ERB* schreiben.

Der Herr sprach, es werde Picsblog

Starten Sie Ihr *InstantRails*-Panel, falls Sie unter Windows arbeiten, und die Entwicklungsumgebung *RadRails*. Sobald beides läuft, können Sie in RadRails ein neues Projekt namens *picsblog* anlegen. Das Erzeugen eines neuen Projekts kennen

Sie bereits aus dem dritten Kapitel. Achten Sie hierbei wieder darauf, dass das *Rails-Skeleton* erstellt und eine *Mongrel*-Instanz erzeugt wird. Das Erzeugen der einzelnen Dateien durch Rails können Sie in der *Console-View* verfolgen.

Bevor Sie loslegen, sollten Sie sicherstellen, dass die Rails-Perspektive des Editors aktiv ist, so dass Sie alle nötigen Ansichten für die effektive Entwicklung einer Rails-Anwendung im Auge haben. Sie können das sehen und ändern, wenn Sie Ihre Blitze in die rechte obere Ecke des Programmfensters schweifen lassen oder auf *Window → Open Perspective* klicken.

Lassen Sie uns zunächst testen, ob das Erzeugen der Rails-Anwendung erfolgreich war und ob der Mongrel-Server ordnungsgemäß werkelt. Wie Sie wissen, erzeugt Rails leere, aber komplett lauffähige Applikationen. Klicken Sie also auf den Server-View, wählen Sie den *picsblogServer* aus und drücken Sie wieder auf den grünen Start-Pfeil.



Sollten Sie mal eine leere Liste von Servern vorfinden, beispielsweise nach dem erneuten Öffnen von RadRails und Projekt, so können Sie sich ganz einfach wieder eine Server-Instanz für Ihre Anwendung verschaffen. Klicken Sie dazu im Rails-Navigator mit der rechten Maustaste auf den Projektbezeichner, der Wurzel des Verzeichnisbaums, und wählen Sie *New... → Mongrel Server*. Das daraufhin erscheinende Dialogfenster können Sie unbearbeitet mit *Finish* bestätigen.

Anschließend sollte Ihnen Ihr Webbrowser unter <http://127.0.0.1:3000> die bereits bekannte Willkommensbotschaft anzeigen.

Hallo Datenbank, bitte melden!

Doch nicht nur ein nette Grußbotschaft ist auf der Seite enthalten. Auch eine Liste der ersten Schritte weist Ihnen den Weg. Und das steht als Erstes: "*Create your databases and edit config/database.yml!*" Genau das machen wir jetzt.

Gemeint ist damit, dass Sie vor Ihren ersten Codezeilen die Datenbanken erzeugen müssen, welche Ihre Anwendung nutzen soll. Wie diese Datenbanken heißen und wie Rails Zugang zu diesen bekommt, schreiben Sie dann in die Datei *database.yml*. Diese Datei, die Ruby on Rails Ihnen netterweise schon angelegt und mit Standardwerten gefüllt hat, ist eine der wenigen Ihrer Anwendung, die Konfigurationsdaten enthält.

Wie Sie wissen, kennt eine Rails-Anwendung drei Environments, in denen sie laufen kann. Jede einzelne Umgebung greift dabei auf eigenes Datenmaterial zurück. Daher benötigen Sie grundsätzlich auch drei Datenbanken: eine für den Entwicklungszeitraum, eine für die Testphase und eine, die während des produktiven Einsatzes benutzt wird. Um nicht den Überblick zu verlieren, empfiehlt es sich, den

Projektnamen in die Datenbankbezeichner zu integrieren, gefolgt von `_development`, `_test` und `_production`. Sie können Ihre Datenbanken aber auch ganz anders benennen; Sie sind da an nichts gebunden.

Um die bislang nur genannten Datenbanken zu erstellen, haben Sie mehrere Möglichkeiten: Als InstantRails-Nutzer können Sie das mitgelieferte Tool *phpMyAdmin* nutzen. Damit können Sie diesen Vorgang mit einer schicken, komfortablen Oberfläche erledigen. Sie erreichen *phpMyAdmin*, wenn Sie im InstantRails-Panel auf den I-Button und dann *Configure → Database (with PhpMyAdmin)* klicken. Daraufhin sollte sich ein Browser-Fenster öffnen. Dort verlangt ein Eingabefeld nach Daten, über dem *Neue Datenbank* anlegen steht. Hier geben Sie zunächst `picsblog_development` ein und bestätigen mit *Enter* oder dem *Anlegen*-Button. Anschließend klicken Sie auf das kleine Häuschen auf der linken Seite. Danach können Sie auch `picsblog_test` und `picsblog_production` erzeugen.

Die zweite Möglichkeit ist das manuelle Erzeugen der Datenbanken über die Kommandozeile. Öffnen Sie eine solche. Sollte sich Ihre MySQL-Datenbank noch im Werkszustand befinden, so existiert ein Datenbankbenutzer namens `root`, dem kein Passwort zugeordnet ist. In diesem Falle geben Sie `mysql -u root` ein und Sie gelangen in den *MySQL Monitor*.



Sollte `root` oder ein anderer MySQL-User bei Ihnen über ein Passwort verfügen, so können Sie den MySQL-Monitor mit `mysql -u <user> -p <password>` aufrufen.

Mit drei SQL-Statements können Sie die Datenbanken erstellen. Achten Sie dabei darauf, jede Zeile mit einem Semikolon enden zu lassen. Schließen Sie den MySQL-Monitor anschließend mit `exit`.

Beispiel 4-3: Drei Datenbanken werden im MySQL Monitor erzeugt

```
CREATE DATABASE picsblog_development;
CREATE DATABASE picsblog_test;
CREATE DATABASE picsblog_production;
```

Nun können Sie in RadRails die Datei `database.yml` öffnen. Die Dateiendung `.yml` kommt natürlich nicht von ungefähr, sondern sie weist darauf hin, dass der Dateinhalt in YAML verfasst ist. YAML ist mal wieder eine dieser grotesken Abkürzungen der IT-Welt, die sich mit sich selbst erklären, denn die vier Buchstaben stehen doch tatsächlich für *YAML Ain't Markup Language*. YAML ist einfach nur eine Sprache, die konzipiert wurde, um Daten für Mensch und Maschine gleichermaßen effizient lesbar zu gestalten. YAML wird daher sehr gern für Konfigurationsdateien verwendet, besonders von Skriptsprachen.

Um `database.yml` richtig zu verwenden, müssen Sie übrigens kein YAML beherrschen. Die eben angesprochene Lesefreundlichkeit von YAML-Code und die Tatsa-

che, dass Rails die Datei für Sie schon einmal mit Standardwerten belegt hat, macht das Bearbeiten auch so zum Kinderspiel. Blicken wir einmal hinein.

Beispiel 4-4: database.yml im Ursprungszustand

```
development:
  adapter: mysql
  database: picsblog_development
  username: root
  password:
  host: localhost

test:
  adapter: mysql
  database: picsblog_test
  username: root
  password:
  host: localhost

production:
  adapter: mysql
  database: picsblog_production
  username: root
  password:
  host: localhost
```

Wie Sie sehen, hat Ruby on Rails hier schon ganze Arbeit für Sie geleistet und für jede der drei möglichen Umgebungen einen Eintrag mit den entsprechenden Daten erzeugt. Prinzipiell brauchen Sie an *database.yml* nichts ändern, schon gar nicht, wenn Sie InstantRails einsetzen. Haben Sie allerdings beim Anlegen der Datenbanken andere Bezeichner als die hier aufgeführten gewählt, so passen Sie diese jetzt an. Sollten Sie einen anderen Datenbankbenutzer als *root* wünschen, so sollten Sie ebenfalls Anpassungen vornehmen.



Mac-OS-Nutzer aufgepasst: *Locomotive* arbeitet grundsätzlich mit *SQLite*. Möchten Sie diese Art der Datenbank nutzen, geben Sie als `adapter: sqlite3` und zusätzlich die Eigenschaft `dbfile: db/development.sqlite3` beziehungsweise `db/test.sqlite3` und `db/production.sqlite3` an.

Einige Rails-Gurus stimmt es missmutig, dass die *database.yml* ganz gehörig gegen das DRY-Prinzip verstößt. Und in der Tat finden Sie in der Datei Passagen, die sich unnötigerweise wiederholen. Daher können Sie obige *database.yml* auch so schreiben:

Beispiel 4-5: database.yml geDRYt

```
login: &login
  username: root
  password:
```

Beispiel 4-5: database.yml (Fortsetzung)

```
adapter: mysql
host: localhost

development:
  database: picsblog_development
  <<: *login

test:
  database: picsblog_test
  <<: *login

production:
  database: picsblog_production
  <<: *login
```

Stets gleiche Daten, wie *Benutzername*, *Passwort*, *Host* und vor allen Dingen *Adapter* stehen nun nur einmal in der Datei und müssen im Falle eines Falles auch nur einmal geändert werden. Mittels &*login* – das & ist ein Erfordernis von YAML – werden diese Daten den einzelnen Umgebungen übergeben.



Nach Änderungen an *database.yml* muss Mongrel neu gestartet werden. Nur so werden Ihre dort gemachten Angaben auch sofort berücksichtigt. Für solche Fälle steht in der Server-View von RadRails ein Restart-Button zur Verfügung.

Neben *database.yml* gibt es noch eine Datei, die interessant für Sie ist, wenn es doch ein, zwei Sachen zu konfigurieren gibt.

Die Datei environment.rb

Sie befindet sich ebenfalls im *config*-Verzeichnis und trägt den Namen *environment.rb*. Diese Datei ist hauptsächlich aus folgenden Gründen wichtig:

1. Sie können hier angeben, mit welcher Rails-Version Ihre Anwendung arbeiten soll. Das ist besonders dann wichtig, wenn Sie via *RubyGems* ein Update von Ruby on Rails vorgenommen haben. Nur wenn Sie Ihre Anwendung explizit auffordern, die neue Version zu nutzen, tut sie das auch. Ändern Sie dazu die Variable *RAILS_GEM_VERSION*. Hier ist standardmäßig die Rails-Version verzeichnet, mit der Sie die Anwendung erzeugt haben. Die aktuelle Versionsnummer Ihres Ruby on Rails erfahren Sie bekannterweise durch die Eingabe von *rails -v* in der Kommandozeile.
2. Über die Zuweisung von *development*, *test* oder *production* an *ENV['RAILS_ENV']*, die Sie im oberen Bereich finden und die standardmäßig durch ein Kommentarzeichen deaktiviert ist, können Sie festlegen, in welcher Umgebung Ihre Rails-Anwendung laufen soll. Allerdings sollten Sie hier nur Änderungen vornehmen,

wenn Sie diese Einstellung nicht über den Server vornehmen können. Da Sie zum jetzigen Zeitpunkt volle Kontrolle über Mongrel haben und über dessen Einstellungen festlegen können, welche die aktuelle Umgebung sein soll, braucht Sie dieser Punkt zunächst nicht zu interessieren.

3. Wichtig wird es noch einmal am Ende von *environment.rb*. Hier bietet Ihnen Ruby on Rails die Möglichkeit, eigene Konfigurationsparameter zu hinterlegen. Vielleicht ist die Tatsache, dass Rails aus Prinzip etwas gegen Konfiguration hat, eine Erklärung für die schlechte Position. Aber manchmal geht es einfach nicht anders. Außerdem spricht das DRY-Prinzip eindeutig für die Verwendung von eigenen Einstellungen an zentraler Stelle, die andernfalls mehrfach im Code auftauchen würden.

Lassen Sie uns die Möglichkeit eigener Konfigurationsparameter nutzen. Picsblog soll Dateien uploaden können. Und die müssen irgendwo hin. Nun eignet sich *irgendwo* relativ schlecht, um es als Ortbeschreibung an Ruby weiterzureichen, wenn es um den Speicherort der Dateien geht. Daher definieren wir zwei Konstanten und ergänzen das Ende von *environment.rb* um folgende zwei Zeilen:

```
IMAGE_DIR = '/uploads'  
THUMBNAIL_DIR = IMAGE_DIR + '/thumbnails'
```

Durch die Konstanten mit den frei gewählten Namen `IMAGE_DIR` und `THUMBNAIL_DIR` legen Sie den Ort fest, an dem die Fotos und deren Thumbnails hingehören sollen. In der Anwendung werden wir noch dafür sorgen, dass diese Angaben relativ zum `public`-Verzeichnis zu verstehen sind. Schließlich sollen Fotos und Thumbnails der Öffentlichkeit zur Verfügung gestellt werden und nur so können Sie in einem Browser angezeigt werden. In der Anwendung werden wir dafür sorgen, dass diese Verzeichnisse bei Bedarf angelegt werden.



Was für *database.yml* gilt, ist auch hier wichtig. Wann immer Sie während des Entwicklungsprozesses Ihrer Rails-Anwendung Änderungen vornehmen, müssen Sie Mongrel neu starten. Nur so wird die veränderte *environment.rb* auch sofort genutzt.

Das Grundlegende ist nun abgeschlossen. Sie haben ein Basis-Layout für Ihre Anwendung angelegt und alles Nötige konfiguriert. Somit können Sie sich jetzt an das Programmieren der Applikation machen.

Beitrags-Model

Wie Sie wissen, liegt die Programmlogik einer MVC-Anwendung in der Model-Schicht. Das heißt, hier platzieren Sie den ganzen Quelltext, der die eigentliche Bestimmung eines Programms widerspiegelt.

Das erste Model von Picsblog wird Post heißen. Hier implementieren Sie alles, was Ihre Rails-Anwendung braucht, um *Beiträge (Posts)* zu speichern, zu manipulieren und zu löschen. Da sich alle Daten in einer Datenbank befinden sollen, ist es auch Aufgabe des Models, einen entsprechenden Schreib- und Lesezugriff auf die Datenbanktabelle zu organisieren, in der die Posts gespeichert werden. Ferner obliegt es dem Model, Daten an den Controller zurück zu geben. Aber Rails wäre nicht Rails, würde das, was sich so kompliziert anhört, nicht kinderleicht sein.

Vielleicht fällt Ihnen noch mehr ein, wenn Sie an die Entwicklung Ihre bisherigen Web-Anwendungen mit Datenbankanbindung denken? Vielleicht ja das umständliche und immer wiederkehrende Programmieren von Methoden, die überhaupt erst einmal eine Verbindung zur Datenbank herstellen. Auch das übernimmt *Active Record* für Sie.

Generieren Sie das Model Post. Klicken Sie dazu auf den *Generator*-Tab und wählen Sie in der linken Listbox `model` aus. Geben Sie im rechten Eingabefeld `Post` ein und klicken Sie auf den OK-Button oder drücken Sie `Enter`. In der Konsole sollten Sie nun verfolgen können, welche Dateien bei diesem Vorgang erstellt werden.

Navigieren Sie sich dann durch den Verzeichnisbaum Ihres Projekts und öffnen Sie `app/models/post.rb`. Hier sehen Sie eine leere Klasse namens `Post`, die von `ActiveRecord::Base` erbt. In dieser Superklasse steckt eine ganze Menge drin. So zum Beispiel alles, was Ihre Anwendung für den Kontakt mit der Datenbank braucht: Methoden zum Anlegen, Speichern, Lesen und Löschen eines Datensatzes – all das und noch viel mehr schenkt Ihnen Rails nur durch diese Erbschaft. Sie selbst brauchen grundsätzlich keine Methoden zu schreiben, wie etwa `save_data` oder `get_data`. Das hat schon jemand für Sie übernommen. Erben macht das Leben eben angenehmer.

Das Model `post` soll seine Daten in einer Datenbank speichern. In MySQL und anderen Systemen erfolgt das in Datenbanktabellen. Die gibt es allerdings noch gar nicht. Bislang haben wir gänzlich nackte und leere Datenbanken. Daher sollten wir nun eine solche Tabelle erstellen, die die Beitragsdaten aufnimmt.

Datenbank mit Migrationshintergrund

Es gibt, wie so oft in Rails, mehrere Möglichkeiten, eine solche Datenbanktabelle zu erstellen. Und ich könnte Ihnen jetzt auch den Luxus bieten, Ihnen alle Möglichkeiten vorzustellen. Aber ich denke gar nicht daran, denn ich möchte, dass Sie Ihr ganzes Ruby-on-Rails-Entwicklerleben nur diese eine Variante kennen und benutzen. Und diese Variante heißt *Migration*, ist recht handlich, sehr einfach und vor allen Dingen: Migrations lassen die Datenbanktabellen mit Ihrer Anwendung wachsen und bei Bedarf auch schrumpfen. Es mag sein, dass Ihnen Migrations zunächst unverständlich vorkommen. Aber es wird nicht lange dauern, und Sie werden sie lieben.

Migrations sind Teil von Active Record. Klar, da ist schließlich das ganze Datenbankzeugs drin. Sie werden genutzt, um festzulegen, welche Änderungen an der Datenbank erfolgen sollen, wenn beispielsweise eine Rails-Anwendung erweitert wird. Migrations definieren Transformationen von Datenbanktabellen.

Nehmen wir an, Sie erweitern Ihre Rails-basierte Website um einen Mitgliederbereich. Sie brauchen also neben der Datenbanktabelle, die die Inhalte Ihrer Site speichert, auch noch eine, in der die Daten Ihrer Mitglieder gespeichert werden. Sie schreiben also eine Migration, in der drin steht, dass Ihre Datenbank um die Tabelle users erweitert wird und dass diese Tabelle die Felder name und password enthält. Nehmen wir weiter an, Sie kommen ein paar Tage später auf die Idee, dass Ihre Benutzer auch ihr Geschlecht angeben sollen. Mit einer Migration weisen Sie an, dass die Datenbanktabelle users um das Feld gender erweitert wird.

Nun kann es sein, dass sich keine Menschenseele für Ihren Mitgliederbereich interessiert. Soll ja vorkommen. Ergo, Sie schaffen ihn wieder ab. Rückbau ist also ange sagt. Und genau aus diesem Grund enthält jede Migration auch Anweisungen für diesen Fall. Hier würde sich anbieten anzuweisen, einfach die komplette Tabelle users zu löschen.

Sie können mit Migrations also verschiedene Versionen oder verschiedene Zustände Ihrer Datenbanktabellen definieren und sind in der Lage, zu jeder Version oder zu jedem Zustand zu gelangen, egal ob vorwärts oder rückwärts.

Migrations bieten aber weitere Vorteile: Arbeiten Sie im Team an einer Rails-Applikation, so reicht eine Migration, damit alle Teammitglieder auf Ihrem Rechner die aktuelle Datenbankstruktur einspielen können, die von einem Teammitglied festgelegt wurde. Die entsprechende Datei muss dafür nur weitergeleitet werden. Migrations sind auch dann sehr komfortabel, wenn Sie beispielsweise an zwei Rechnern arbeiten. Sollte Sie also, und ich spreche da auch aus Erfahrung, abends noch die Lust oder der terminliche Zwang packen und Sie möchten noch daheim an Ihrer auf einem USB-Stick gespeicherten Rails-Anwendung pfeilen, dann ist eine Migration ein wahrer Freund. In Windeseile haben Sie daheim die gleiche Datenbankstruktur wie im Büro.

Genug gelobt, werfen wir einen Blick in eine Migration. Ruby on Rails war so freundlich und hat beim Erzeugen des Models Post automatisch eine Migration erzeugt. Die ist natürlich noch leer und wartet auf Anweisungen. Doch sie hat schon eine vielsagende Bezeichnung und heißt *001_create_posts.rb*, trägt also die Bezeichnung der Tabelle, die zum Model gehört, im Namen. Sie finden sie im Verzeichnis *db/migrate*.



Migrations werden zwar automatisch mit jedem Model erzeugt, doch sie können auch unabhängig von einem Model generiert werden. Wählen Sie dazu im Generator-View von RadRails *migration* aus und vergeben Sie einen treffenden Namen. Dieser kann frei gewählt werden, sollte aber wie eine Ruby-Variable oder in CamelCase gebildet werden. Zum Beispiel: `create_users` oder `add_gender_column`.

Die Nummer zu Beginn des Dateinamens einer Migration gibt Auskunft über ihren Rang. 001 steht für die erste Version der Datenbanktabelleninformationen. Die nächste Migration beginne selbstredend mit 002. Diese Zahlen nutzt Ruby on Rails auch intern für die Speicherung der gerade aktuellen Migration. Außerdem können Sie über diese Versionierung Rails mitteilen, welche Migration die aktuelle sein soll. Rails leitet daraufhin alle erforderlichen Schritte für den Weg dorthin ein.

Wie Sie bereits wissen, können Sie in beide Richtungen migrieren. Das gelingt durch die beiden Methoden `self.up` und `self.down`, die in jeder Migration enthalten sind und die Sie nun auf Ihrem Bildschirm sehen sollten. Sie sind Klassenmethoden der automatisch erzeugten Klasse `CreatePosts`.

Beispiel 4-6: Leere Migration: 001_create_posts.rb

```
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
    end
  end

  def self.down
    drop_table :posts
  end
end
```

Zum jetzigen Zeitpunkt haben Sie noch keine Migration eingespielt. Die virtuelle Migration `000_irgendwas` ist also gerade aktuell. Würden Sie nun Rails anweisen, Migration `001_create_posts` einzupflegen, würde Rails `CreatePosts#up` ausführen. Möchten Sie von `001` zu `000` zurückkehren, dann käme `CreatePosts#down` zur Ausführung. Einen Vorschlag, was in diesem Falle zu tun wäre, hat Ihnen Rails schon unterbreitet: `drop_table :posts`. Die Tabelle `posts` würde also in diesem Fall ersatzlos aus der Datenbank entfernt werden. Das können Sie so übernehmen. Widmen wir uns dem Tabellen-Upgrade.

Dabei soll eine Datenbanktabelle entstehen, die `posts` heißen soll. Auch das hat Ihnen Rails schon vorgegeben. Das Ergebnis dieser Aktion ist eine Tabelle, die allerdings keinerlei Felder oder Spalten enthält. Sie erhalten jedoch Zugriff auf diese Tabelle über die Blockvariable `t` in dem Block, den Sie `create_table` noch anhängen können.

Ein jeder Beitrag in Picsblog soll zunächst aus den Feldern *Titel*, *Beschreibung*, *Datum*, *Dateiname des Fotos* und Dateiname des *Thumbnails* bestehen. Über die Methode `column` können Sie der Tabelle `t` diese Spalten hinzufügen.

Beispiel 4-7: Tabelle posts wird in self.up mit fünf Spalten erzeugt

```
class CreatePosts < ActiveRecord::Migration
  def self.up
    create_table :posts do |t|
      t.column(:title, :string)
      t.column(:description, :text)
      t.column(:date, :datetime)
      t.column(:image, :string)
      t.column(:thumbnail, :string)
    end
  end

  def self.down
    drop_table :posts
  end
end
```

Die Methode `column` benötigt mindestens zwei Parameter: den Namen der zu erzeugenden Spalte und den Typ derselben. Somit legen Sie fest, welche Art von Daten eine Spalte zu erwarten hat. Die `posts`-Tabelle von Picsblog brauchte Spalten mit kurzen Texten (`:string`), langen Texten (`:text`) und Zeit- und Datumsangaben (`:datetime`). Sie können aber auch Spalten für ganzzahlige Werte (`:integer`), gebrochene Zahlen (`:decimal`), Datum (`:date`), Zeit (`:time`) und Wahr/Falsch (`:boolean`) definieren.

Rails legt übrigens beim Erstellen einer neuen Tabelle mit einer Migration automatisch ein Feld mit der Spaltebezeichnung `id` an. Dadurch erhält jeder Datensatz eine Zahl, mit der er eindeutig identifiziert und angesprochen werden kann. Im weiteren Verlauf dieses Kapitels werden Sie sehen, wie wichtig diese Spalte ist.

Nun können Sie die *Transformation* vornehmen lassen und eine Tabelle `posts` mit fünf Spalten anlegen. Dies erledigt für Sie ein Rails-eigenes Tool namens Rake. Sie finden einen entsprechenden Karteikartenreiter neben *Server-* und *Console*-Tab. Klicken Sie *Rake Tasks* zum ersten Mal an, wird RadRails möglicherweise versuchen herauszufinden, mit welchen Befehlen Sie Rake füttern können. Warten Sie also gegebenenfalls ein paar Sekunden ab. Daraufhin sollte die linke Liste mit allerlei Anweisungen gefüllt sein. Falls nicht, können Sie hier auch manuell Rake-Befehle eingeben.

Sorgen Sie dafür, dass die linke Combobox `db:migrate` anzeigt. Damit weisen Sie an, alle vorhandenen Migrations bis zur höchsten Versionsnummer durchzuarbeiten. Möchten Sie diesen Vorgang bis zu einer gewissen Version einschränken, so geben Sie in dem rechten oberen Eingabefeld des Rake-Tasks-Views `VERSION=4` ein,

um beispielsweise Migrations nur bis einschließlich 004 zu berücksichtigen. In unserem Fall könnten Sie hier `VERSION=1` eintragen, aber nötig ist es nicht. Durch die Eingabe einer Versionsnummer können Sie Migrations auch wieder rückgängig machen. Mit `VERSION=0` würden Sie zum jetzigen Zustand zurückkehren.

Was Migrations können

Sie haben gerade erfahren, dass Sie mit einer Migration eine Datenbanktabelle mit diversen Feldern erzeugen und selbige auch wieder löschen können. Migrations können aber noch viel mehr. ActiveRecord::Migration wartet insgesamt mit folgenden Funktionen auf:

- Hinzufügen einer Tabelle
- Löschen einer Tabelle
- Umbenennen einer Tabelle
- Hinzufügen von Tabellenspalten
- Entfernen von Tabellenspalten
- Umbenennen von Tabellenspalten
- Hinzufügen eines Index'
- Hinzufügen eines Primärschlüssels
- Hinzufügen eines Fremdschlüssels (zur Verknüpfung mit anderen Tabellen)
- Anlegen, Bearbeiten und Löschen von Datensätzen

Wie sich einige dieser Funktionalitäten in Ruby-Code niederschlagen, erfahren Sie noch im Laufe dieses Buches.

Mit einem Klick auf Go wird die Transformation durchgeführt; sie kann in der *Console* beobachtet werden. Wenn alles gut lief, sollten Sie ähnliche Zeilen dort lesen können:

```
== CreatePosts: migrating =====
-- create_table(:posts)
 -> 0.3600s
== CreatePosts: migrated (0.3600s) =====
```

In der Console erhalten Sie auch Hinweise auf Fehler oder Probleme, wenn das Erstellen der Tabelle *posts* nicht erfolgreich war.



Standardmäßig migriert Rake Ihre Datenbankwünsche innerhalb des Environments development. Wie Sie Transformationen an Datenbanktabellen für den produktiven Einsatz vornehmen können, erfahren Sie im 6. Kapitel dieses Buches.

Den Erfolg des Migrierens können Sie sich auch ansehen. Entweder, Sie benutzen dafür wieder *phpMyAdmin* oder Sie schalten in die RadRails-Perspektive namens *Data*. Dort präsentiert sich Ihnen eine kleine Oberfläche, mit der Sie die Datenbanktabellen des Projekts verwalten können.

Durch das Nutzen der Migration *001_create_posts.rb* haben Sie also eine Tabelle in *picsblog_development* erstellt. Und weil die posts heißt, weiß Ruby on Rails, dass diese Tabelle zum Model Post gehört. Das ist ein typisches Beispiel für Rails' Prinzip *Konvention über Konfiguration*. Was außerhalb von Rails explizit niedergeschrieben werden muss, wird hier über die Vereinbarung gelöst, dass die zum Model gehörende Datenbanktabelle den Namen des Models in der Mehrzahl trägt.

In Rails werkelt extra ein Algorithmus, der die Pluralisierung übernimmt. Nun gut, so schwierig ist das ja auch nicht. Meistens muss eigentlich nur ein S an das Einzahlwort angehangen werden. Aber auch exotische Einzahl-Mehrzahl-Konvertierungen hat Rails drauf. So ist Rails beispielsweise auf den Fall vorbereitet, dass ein Model mal *person* heißt. Die Datenbanktabelle heißt dann eben *people*. Selbst wenn ein Programmierer mal vorhat, ein *mouse*-Model zu programmieren – was immer er damit auch vorhat – Rails weiß, dass die Mehrzahl *mice* heißt.

Nochmal in Kurzform: Sie haben in einer Migration definiert, dass eine neue Datenbanktabelle erstellt werden soll. Durch eine Namenskonvention ist diese Tabelle mit dem Model Post verbunden. Das Model Post enthält die komplette Funktionalität zum Bearbeiten und Lesen der Datenbanktabelle – ohne, dass Sie selbst dafür viel tun mussten.

Zum Leistungsumfang des Models Post soll das Speichern von Bilddateien gehören, die über die Benutzeroberfläche von Picsblog hochgeladen werden. Da dies natürlich keine Standardfunktionalität von Active Record ist, steht hiermit fest, dass *post.rb* nicht nackt bleibt. Aber darum kümmern wir uns gleich.

PostsController, übernehmen Sie!

Vergegenwärtigen Sie sich noch einmal das MVC-Modell. In der Model-Schicht steckt das eigentliche Programm, das durch die Verwendung von Views sichtbar wird. Die Controller-Schicht ist der Mittler zwischen Model und View. Er reagiert auf Benutzeraktivitäten, fordert Daten vom Model an und reicht Daten des Benutzers an ein Model zur Speicherung oder Bearbeitung weiter. Erst der Controller löst Aktivitäten im Model aus, beispielsweise weil er via *HTTP-Request* einen entsprechenden Autrag durch den Benutzer erhalten hat.

Lassen Sie uns also einen Controller erstellen, der das eben generierte Model erst richtig zum Leben erweckt und es sinnvoll nutzbar macht. Klicken Sie dazu wieder auf den *Generator*-Tab und lassen Sie sich einen controller (linkes Feld) namens Posts (rechtes Feld) generieren. Auch hier ist die Mehrzahl des Model-Namens, auf

den sich der Controller bezieht, eine gute Wahl - allerdings besteht keinerlei Zwang zu dieser Pluralform. Bedenken Sie aber stets, dass der Controller-Name auch Bestandteil eines URLs sein wird.

Genau wie das Model liegt der Controller an einer bestimmten Stelle in einer eigenen Datei vor, die ein leeres Klassenkonstrukt mit dem Namen des Controllers enthält. Rails fügte dem Controller-Bezeichner noch ein Controller hinzu, so dass die eben generierte Klasse PostsController und die Datei *posts_controller.rb* heißt. Schauen Sie selbst: *app/controllers/posts_controller.rb*.

Controller enthalten hauptsächlich Methoden, die als Action nach außen in Erscheinung treten. Wenn Sie überlegen, welche Actions denn nötig sind, um mit Beiträgen des Blogs vernünftig umgehen zu können, dann kommen Sie sicher auch auf diese: Beiträge müssen neu angelegt, bearbeitet, gelöscht und angezeigt werden können.

Sollte Ihnen beim Durchdenken der nötigen Actions das Wort *Scaffolding* durchs Hirn geschliddert sein, so lassen Sie mich an dieser Stelle meine Bewunderung zum Ausdruck bringen. In der Tat schreit diese Gruppe von Aktionen förmlich nach dem Einsatz von Scaffolding.

Aber Sie wissen ja, wie das so läuft. Schon Ihre Grundschullehrer haben Ihnen sicher oft einfache Methoden zunächst vorenthalten, um sie mit dem richtig harten Stoff so lange zu quälen, bis sie kapiert haben, auf was Sie zukünftig verzichten können. Und so ist es hier auch. Aber es lohnt sich zu wissen, wie die Dinge auch ohne Scaffolding laufen.

Und... Action!

Die erste Action des Controllers soll new sein. Wie der Name schon verrät, wird diese Action dann aufgerufen, wenn ein neuer Datensatz und somit ein neuer Blog-Beitrag angelegt werden soll. In diesem Falle wird eine Instanz unserer Model-Klasse Post erzeugt und an das Attribut @post der Controller-Klasse übergeben. Bei der Wahl des Attributbezeichners haben Sie natürlich freie Wahl. Aber @post bietet sich einfach an.

Beispiel 4-8: Die erste Action des Posts-Controllers

```
class PostsController < ApplicationController
  def new
    @post = Post.new
  end
end
```

Ganz wichtig: Zwar war eben die Rede von der Neuanlage eines Datensatzes. Die Methode Post#new leistet dies jedoch nicht, legt aber den Grundstein dafür. Mit ihrem Aufruf finden keinerlei Veränderungen an der Datenbank statt. Hier wird

lediglich in leeres Objekt der Model-Klasse Post erzeugt. Sie können sich @post auch als virtuelle und leere Zeile der Datenbanktabelle posts vorstellen, inklusive aller Datenbankfelder als *Akzessoren*. Sie könnten das Objekt jetzt mit Werten füllen oder anderweitig verändern. Erst wenn Sie @post explizit dazu auffordern, den Datensatz zu speichern, wird er auch physisch in die Datenbank geschrieben. Aber dazu kommen wir gleich.



Erwähnenswert ist noch, dass Sie das Post-Model nicht mit require einbinden müssen. Ruby on Rails übernimmt das für Sie.

Wie Sie bereits wissen, werden Actions eines Controllers über einen URL ausgeführt. Unsere eben implementierte Action heißt new, der dazugehörige Controller Posts - es ergibt sich also folgende Adresse: <http://127.0.0.1:3000/posts/new>. Rufen Sie die ruhig mal auf und beobachten Sie, was passiert. Vergessen Sie aber vorher nicht, *posts_controller.rb* zu speichern.

Ruby on Rails beschwert sich, dass ein Template schmerzlich vermisst wird. Zumindest befindet es sich nicht an der Stelle, wo es per Konvention hingehören würde, nämlich *app/views/posts/new.rhtml*. Und Rails hat natürlich recht. Die Anwendung verfügt mittlerweile zwar über Model und ansatzweise Controller, doch das V in MVC fehlt noch.

Ansichtssache

Im nächsten Schritt sollten Sie das eingangs erstellte Grundlayout in Ihre Anwendung einbauen und anschließend die erste View, die für die optische Darstellung der Action new des Posts-Controllers verantwortlich ist.

Kopieren Sie die Datei *standard.html* in das Unterverzeichnis *app/views/layouts*. Da sie gleich Ruby-Code erhalten wird, ändern Sie die Dateiendung bitte in *.rhtml*. Die Datei *standard.rhtml* gehört nicht ins *public*-Verzeichnis, da sie lediglich als Basis zum Rendern der Benutzeroberfläche dient und nie direkt an einen Webbrowser geschickt wird, von selbigem also auch nicht erreicht werden muss. Anders verhält es sich mit der CSS-Datei *layout.css*. Diese kopieren Sie bitte in das Verzeichnis *public/stylesheets*. Die Dateiendung bleibt dabei unverändert.

Öffnen Sie nun *standard.rhtml*, um ihr etwas Ruby einzupflanzen. Die erste Änderung betrifft das Einbinden der CSS-Datei, damit diese ordnungsgemäß mit dem durch Rails erzeugten Output verknüpft wird. Mit *stylesheet_link_tag* treffen Sie zum ersten Mal auf eine Methode des Sub-Frameworks *Action View*. Es handelt sich hierbei um eine der vielen Tag-Helper-Funktionen, von denen Sie noch zahlreiche kennen lernen werden. Sie werden benutzt, um HTML oder JavaScript mit Ruby auszudrücken.

Beispiel 4-9: stylesheet_link_tag bindet layout.css ein

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>Picsblog</title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
    <%= stylesheet_link_tag('layout') %>
  </head>
  <body>
    ...
  </body>
</html>
```

Geben Sie als Parameter den Dateinamen des Stylesheets an. Wenn die betreffende Datei mit .css endet, können Sie, wie hier, die Extension einfach weglassen. Möchten Sie mit einem Mal mehrere Stylesheets einbinden, dann trennen Sie diese durch Komma.

Später, wenn das Layout durch Rails verarbeitet wurde, wird statt der Helper-Funktion echtes HTML an den Browser gesendet. `stylesheet_link_tag('layout')` ergibt dabei `<link href="/stylesheets/layout.css" media="screen" rel="Stylesheet" type="text/css" />`. Damit dies auch wirklich passiert, sollten Sie unbedingt darauf achten, dass der öffnende ERb-Tag ein Gleichheitszeichen enthält, sonst erfolgt zwar die Umwandlung, jedoch erscheint deren Ergebnis nicht. Nur `<%=` wertet einen Ausdruck aus und setzte das Ergebnis in den HTML-Code ein.

Im nächsten Schritt legen Sie fest, wohin die Daten sollen, die die Action des Controllers erzeugt. Bislang steht an dieser Stelle in `standard.rhtml` nur Content, ändern Sie das in `<%= yield %>`. Mit dieser Methode wird ausgegeben, was Actions wollen.

Beispiel 4-10: Dynamische Inhalte mit yield

```
...
<div id="main">
  <div id="content">
    <%= yield %>
  </div>
  <div id="sidebar">
  ...
</div>
```

Statt `<%= yield %>` finden Sie in Views ab und an `<%= @content_for_layout %>`. Beide Varianten bewirken die gleiche Funktionalität, nur ist `@content_for_layout` nicht mehr aktuell. *Deprecated*, wie es fachsprachlich heißt.

In `standard.rhtml` sind damit noch nicht die letzten ERb-Stellen eingesetzt. Wie Sie sehen, muss beispielsweise der Platzhalter der Sidebar ersetzt werden. Darum werden wir uns später aber noch kümmern.

Nun soll erst einmal der Controller Posts Wind davon bekommen, dass es da ein neues Layout für ihn gibt, welches er für das Rendern der Daten nutzen soll, die an

den Browser geschickt werden. Fügen Sie dazu einfach die Methode `layout` mit einem entsprechenden Parameter zu Beginn der Controller-Klasse und vor der Action `new` ein.

```
layout('standard')
```

Damit haben Sie allerdings nur das Layout für den Posts-Controller festgelegt. Verwendet Ihre Anwendung mehrere Controller, deren Action etwas auf den Bildschirm zaubern sollen, müssen Sie diese Zeile in jedem dieser Controller unterbringen.



Nennen Sie Ihr Layout `application.rhtml`, dann brauchen Sie keine expliziten Angaben zu einem Layout innerhalb eines Controllers machen. Standardmäßig wird dann stets `application.rhtml` herangezogen, in jedem Controller. Es sei denn, Sie geben dort explizit ein Layout an. Wichtig ist, dass sich `application.rhtml` in `app/views/layouts` befindet.

Damit ist unser Grundgerüst schon um vieles mehr Rails-konform. Machen wir uns nun an das Befüllen des Skeletts.

Die erste View

Erzeugen Sie in `app/views/posts` mit einem Rechtsklick und der Auswahl von `New → File` eine View namens `new.rhtml`. Wie Sie sich bestimmt denken können, schreit Rails nicht ohne Grund nach einer View, die genau so heißt wie die Action, die via URL aufgerufen wurde. Das Suchen einer View, die genau so heißt wie eine dazugehörige Action, ist Standardverhalten. Und so sieht `new.rhtml` aus:

Beispiel 4-11: `new.rhtml`

```
<h2>Neues Foto anlegen</h2>
<%= form_tag({:action => :create}, :multipart => true) %>
<h3>Bilddaten</h3>
<p>Titel<br/>
<%= text_field(:post, :title) %></p>
<p>Beschreibung<br/>
<%= text_field(:post, :description) %></p>
<p>Datum und Uhrzeit<br/>
<%= datetime_select(:post, :date, :order => [:day, :month, :year, :hour]) %><p>
<h3>Datei-Upload</h3>
<p>Bilddatei:<br/>
<%= file_field(:post, :image_file) %></p>
<p>Thumbnail:<br />
<%= file_field(:post, :thumbnail_file) %></p>
<%= submit_tag('Speichern') %></p>
</form>
```

Völlig klar: Diese View trägt die Dateiendung *.rhtml* vollkommen zu recht. Neben einigen echten HTML-Tags wie `<h2>`, `<p>` und `</form>` stecken dank ERb wieder einige Tag-Helper im Quelltext. Genau genommen sind es hier Form-Helper, da sie alle für das leichte Erstellen eines HTML-Formulars zuständig sind.

Um zu verstehen, was diese kleinen Dienstleister machen, lohnt sich ein weiteres Mal ein Blick in den finalen Code, den Rails generiert und der in Ihrem Browser ankommt. Laden Sie wieder `http://127.0.0.1:3000/posts/new` in Ihrem Browser. Diesmal sollten Sie ein Webformular sehen, das in das Layout *standard.rhtml* eingebaut wurde.

Mit `form_tag` generieren Sie ein `form`-Element und statthen es mit Informationen über den Ziel-URL aus. Für HTML-Experten: Das ist der Wert des `action`-Attributs eines `form`-Elements. Das Ziel wird hier allein durch die Action definiert, die die Formulareingaben bearbeiten soll. In diesem Fall handelt es sich um `create`. Diese Action werden wir gleich noch implementieren.

Die Angaben zum Ziel des Formulars werden bei `form_tag` stets als erstes angegeben, und zwar in Form eines Hashes, auch wenn dieser hier nur einen Schlüssel, `action`, aufnimmt. Denn es kann sein, dass das Ziel eines Formulars eine Action eines anderen Controllers ist. In diesem Fall müssten Sie den Hash noch durch den Schlüssel `controller` und einem entsprechenden Wert ergänzen. Da in unserem Fall kein Controller-Wechsel stattfindet, ersparen wir uns diese Angabe. Rein theoretisch könnten Sie aber auch

```
<%= form_tag({:action => :create, :controller => :posts}, :multipart => true) %>
```

schreiben. Mit `:multipart => true` sorgen Sie dafür, dass Dateifelder, die innerhalb des Formulars auftreten, auch verarbeitet werden können. Standardmäßig erfolgt die Datenübertragung mit der POST-Methode, wodurch `form_tag` das `method`-Attribut des resultierenden `form`-Tags auf eben diesen Wert setzt. So sieht das gerenderte Ergebnis in HTML aus:

```
<form action="/posts/create" enctype="multipart/form-data" method="post">
```

Mit dem Tag-Helper `text_field` erzeugen Sie ein simples Eingabefeld. HTML-Auskenner wissen, dass damit ein `input`-Element gemeint ist, dessen `type`-Attribut den Wert `text` besitzt. Grundsätzlich braucht `text_field` zwei Parameter: das Objekt, welches den momentanen Inhalt des Eingabefeldes enthält und den Akzessor, durch den dieser gelesen werden kann. Da das Attribut `@post` des Controllers `Posts`, das auch in der View zugänglich ist, ein Abbild des Models `Post` ist, besitzt es Akzessoren, die so heißen, wie die Spalten der Tabelle `posts`. Ja, hier geht ordentlich die Post ab.

Die beiden Parameter sind aber nicht nur wichtig für das Lesen von Werten. Auch die Übergabe des Werts aus dem Formular an den Controller erfolgt in einem Objekt mit entsprechenden Akzessoren. Das macht das Weiterverarbeiten einfacher.

cher und systematisiert die grundsätzliche Kommunikation zwischen Model, View und Controller ganz wunderbar. Sie werden das gleich in der Action create erleben.

Ein weiterer Verwendungszweck der beiden Parameter besteht in der automatischen Bildung des Wertes für das id-Attribut eines Eingabefeldes. Das erkennen Sie bei einem Blick in den Code, der durch `text_field` generiert wird. Hier am Beispiel von `title`:

```
<input id="post_title" name="post[title]" size="30" type="text" />
```

Über das id-Attribut eines HTML-Elements können Sie es via CSS stylen und / oder via JavaScript und DOM dynamischer machen.

Ein weiterer nützlicher Helfer ist `datetime_select`. Er kommt unserem Ansinnen sehr entgegen, bei jedem Beitrag Datum und Uhrzeit des Bildes in der Tabellenspalte `date` zu speichern. Das Eingeben des Zeitpunkts ist eine recht komplexe Angelegenheit, wenn man bedenkt, dass das Datum aus drei, die Zeit aus mindestens zwei Angaben besteht. Mit `datetime_select` gelingt das mit wenigen Zeichen.

Auch hier ist die Angabe des Objekts und des Akzessors obligatorisch. Im vorliegenden Fall schloss sich der Befehl an, die einzelnen Elemente in einer bestimmten Reihenfolge darzustellen. Das deutsche Datums- und Zeitformat ist nun einmal nicht Standard in Ruby on Rails.

```
<%= datetime_select(:post, :date, :order => [:day, :month, :year, :hour]) %>
```

Neben den Daten des Bildes soll hier auch die Bilddatei selbst und eine Vorschauansicht, ein Thumbnail auf den Server geladen werden. Dazu bietet HTML Eingabefelder an, die speziell für dieses Vorhaben optimiert sind. Auch hierfür wird Ihnen ein Form-Helper angeboten, `file_field`. Die Besonderheit hier ist die Benutzung der Attribute `:image_file` und `:thumbnail_file`. Die kennt das Model Post und die Datenbanktabelle posts gar nicht. Was das soll, erfahren Sie gleich, wenn wir uns mit der Action create beschäftigen, die Ziel dieses Formulars ist.

Den Abschluss des Formulars bietet ein *Submit*-Button, der, sobald der angeklickt wurde, das Formular versendet. In HTML handelt es sich um ein `input`-Element des Typs `submit`, hier ist es ein weiterer Form-Helper, dem Sie als Parameter die Beschriftung der Schaltfläche übergeben können,

Ganz wichtig ist der schließende `form`-Tag. Er bildet das Gegenstück zum Befehl `form_tag`, mit dem das Formular geöffnet wurde. Ohne ihn funktioniert das ganze Formular unter Umständen im Webbrowser nicht.

Beiträge speichern

In der Action `create`, die Sie dem Controller Posts nun hinzufügen sollten, wird das Datenmaterial des Formulars entgegengenommen und an das Model Post zur Speicherung in der Datenbank übergeben. Allerdings nur, wenn die Eingaben vollständig sind und das Hochladen der Bilddateien erfolgreich war.

Neues Foto anlegen

Bilddaten

Titel

Beschreibung

Datum und Uhrzeit
 14 January 2007 — 14 : 37

Datei-Upload

Bilddatei:
 Durchsuchen...

Thumbnail:
 Durchsuchen...

Abbildung 4-2: Neues Foto anlegen

Beispiel 4-12: Action create

```
def create
  @post = Post.new(params[:post])
  if !@post.valid?
    Flash.now[:notice] = "Bitte füllen Sie alle Felder aus und überprüfen Sie Ihre Angaben."
    render(:action => :new)
  elsif !@post.save_files
    Flash.now[:notice] = "Es trat ein Fehler beim Hochladen der Dateien auf."
    render(:action => :new)
  else
    @post.save
    flash[:notice] = "Dateien wurden hochgeladen und die Daten gespeichert."
    redirect_to(:action => :list)
  end
end
```

Zunächst einmal wird wieder eine Instanz des Post-Models erzeugt und in @post gespeichert. Diesmal wird @post allerdings mit den Formulareingaben initialisiert.

Die sind Bestandteil von `params`, einem Hash, der mit allerlei Daten des HTTP-Requests, der die Action auslöste, gefüllt ist. Unter anderem stecken auch alle Eingaben des Formulars dort drin, konkret im Schlüssel `post`.

Das ist natürlich kein Zufall, sondern röhrt daher, dass wir den Form-Helpers wie `text_field` oder `datetime_select` angewiesen haben, ein Objekt namens `post` zu benutzen.

Im nächsten Schritt wird überprüft, ob die Formulareingaben, die nun in `@post` stecken, auch gültig sind. Dies erfolgt mit der Methode `valid?`, welche in Active Record definiert ist. Die Regeln, welche bestimmen, ob die Angaben in Ordnung sind oder nicht, werden im Model festgelegt. Dazu gleich mehr.

Sollten die Angaben den Anforderungen unserer Anwendung nicht genügen, erhält `flash.now[:notice]` eine entsprechende Notiz und das Formular, das in `new.rhtml` steckt, wird wieder angezeigt. Das passiert dank der Methode `render`. Durch den Schlüssel `action` und dessen Wert `:new` machen wir deutlich, dass hier nicht die zur Action gehörende Standard-View genutzt werden soll, sondern das der Action `new`.

Da `@post` diesmal bereits mit Werten, den bisherigen Eingaben, bestückt ist, erscheinen die Eingabefelder des Formulars in `new.rhtml` aber nicht mehr leer. Der Benutzer der Anwendung braucht in diesem Fall also nicht alles noch einmal einzugeben, sondern muss nur noch falsche oder unzureichende Angaben korrigieren.

Das Formular wird auch dann angezeigt, wenn die Angaben ausreichend sind, aber die Dateien, Bild und Thumbnail, aus welchen Gründen auch immer, nicht auf den Server geladen werden konnten. Dieser Fall tritt ein, wenn die Methode `save_files` des Post-Models nicht erfolgreich war. Diese Methode fügen wir übrigens gleich noch dem Model hinzu.

Sollten die Eingaben den Anforderungen der Anwendung genügen und war das Hochladen der Dateien kein Problem, können die Daten in der Datenbank gespeichert werden. Dies erfolgt durch die Methode `save`, die unser Model von seinem Vorfahren geerbt hat. Nach dem Speichern soll die Anwendung zur list-Action weiterleiten.

In diesem Zusammenhang möchte ich Sie auf die Unterschiede zwischen `render` und `redirect_to` hinweisen und versuchen die Frage zu klären, wann Sie welche Methode nutzen sollten.

Mit `render` weisen Sie Rails lediglich an, eine bestimmte, an eine Action gekoppelte View zu rendern, eine Action gleichen Namens wird dabei aber nicht ausgeführt. Das bedeutet auch, dass die im Template benötigten Daten durch die aufrufende Action bereitgestellt werden müssen.

Anders verhält es sich bei `redirect_to`. Hier wird die benannte Action ausgeführt, als ob Sie diese durch einen entsprechenden URL aufrufen würden. Die Methode `redirect_to` macht das übrigens ganz ähnlich und forciert eine komplett neue Anfrage an den Server. Was aber hat es mit dieser ominösen Variable `flash` auf sich?

Rails' Kurzzeitgedächtnis: flash

Bei `flash` handelt es sich um einen speziellen Hash, mit dem es möglich ist, Objekte zwischen Actions auszutauschen. Alles, was Sie an `flash` übergeben, existiert auch nach dem nächsten HTTP-Request – im Gegensatz zu allen anderen Variablen, die Sie zum Beispiel innerhalb einer Action benutzen und die grundsätzlich verloren gehen, sobald eine neue Action aufgerufen wird.

Werfen Sie doch noch mal einen Blick in unsere Action `create`. Hier wird am Ende die frohe Botschaft über die Speicherung von Daten und Dateien an `flash[:notice]` übergeben. Anschließend wird die Anwendung zu einer anderen Action, hier `list`, umgeleitet. Dabei erfolgt ein neuer HTTP-Request. Das tolle an `flash` ist nun, dass trotz des neuen Requests die Botschaft weiterhin abrufbar ist. Das in `flash[:notice]` gespeicherte String-Objekt kann nun von `list` ausgelesen oder direkt in der View ausgegeben werden.



Es gibt `flash` auch in der Variante `flash.now`. Alle Daten, die Sie `flash.now` zuordnen, werden *nicht* bis nach dem nächsten Request aufbewahrt, sondern verfallen zuvor. Mit `flash.now` sollten Sie arbeiten, wenn Sie eine Anwendungsbotschaft anzeigen möchten, ohne dass zuvor ein neuer HTTP-Request erfolgt. Andernfalls kann es dazu kommen, dass `flash` eine Botschaft enthält, die gar nicht mehr aktuell oder relevant ist, wegen ihrer Lebensdauer aber immer noch existiert und angezeigt wird.

In `Post#create` wird `flash.now` in Zusammenhang mit `render` verwendet. Wie Sie wissen, wird bei `render` einfach eine bestimmte View zur Anzeige gebracht. Ein neuer Request erfolgt hierbei nicht, im Gegensatz zu `redirect_to`.

Unabhängig davon, ob Sie nun `flash` oder `flash.now` wählen, steht der Inhalt in der View stets in `flash` zur Verfügung.

Vielleicht erinnert Sie das Verhalten von `flash` an das von *Sessions*, die im Web einen ganz ähnlichen Dienst leisten. Der Unterschied zwischen dem `flash`-Objekt und Sessions liegt jedoch in der Lebensdauer der enthaltenen Daten. Während Session-Daten beispielsweise bis zum Schließen des Browser vorgehalten werden können, halten `flash`-Inhalte nur bis nach dem nächsten Request. Sobald die damit verbunden Action ausgeführt wurde, sind sie nicht mehr nutzbar.

`Flash[:notice]` wird übrigens sehr gern benutzt, um Hinweise an den Benutzer zu speichern, die bei der nächsten Gelegenheit angezeigt werden. Die Übergabe der Texte ist bereits geregelt, nur gibt es keine Stelle, wo sie der Benutzer zu Gesicht bekommt. Eine gute Stelle zum Platzieren der Hinweise ist das Layout, *standard.rhtml*. Hier können Sie eine feste Stelle für derartige Informationen definieren.

Beispiel 4-13: Meldung machen im Layout!

```
...
<body>
<div id="page">
  <div id="header">
    <% if flash[:notice] %>
      <div id="notice"><p><%= flash[:notice] %></p></div>
    <% end %>
    Header
  </div>
  <div id="main">
    <div id="content">
      <%= yield %>
    ...
  
```

Diese Erweiterung des Layouts besagt, dass, sobald eine Meldung in `flash[:notice]` vorliegt, soll im Header-Bereich ein `div`-Element integriert werden, das den Text anzeigt. Das `div` hat hier schon eine `id`-Zuweisung erfahren, die wir uns später bei der finalen optischen Gestaltung noch zu Nutze machen werden.

Models erweitern

Bevor die Action `create` in Betrieb gehen kann, müssen Sie das Model `Post` noch erweitern. Zum Beispiel muss noch die Frage geklärt werden, was mit den Eigenschaften `image_file` und `thumbnail_file` geschehen soll. Diese existieren schließlich nicht in der Datenbank `posts` und somit auch nicht als *Akkessoren* der Klasse `Post`. Und überhaupt: Was soll das Ganze eigentlich?

Der Grund für diesen Weg liegt in der Natur der Eingabefelder für Dateien. Sie liefern nicht einfach einen Dateinamen, sondern vielmehr ein Bündel von Dateinamen, Pfaden und Dateinhalt. In der Datenbanktabelle sollen allerdings nur Dateinamen von Foto und Thumbnail erscheinen. Das Auslesen des Dateinamens sollen zwei Methoden erledigen, die unser `Post`-Model um die Eigenschaften `image_file` und `thumbnail_file` erweitern. Hier genügen Setter-Methoden, die das Dateiobjekt jeweils entgegennehmen, den Dateinamen auslesen und an das dank der Datenbanktabelle bereits existierende Attribut `image` beziehungsweise `thumbnail` übergeben. Fügen Sie Ihrem Model in `post.rb` also folgende zwei Methoden hinzu.

Beispiel 4-14: Post#image_file= und Post#thumbnail_file=

```
def image_file=(fileobj)
  if fileobj.size > 0
    @image_file = fileobj
    self.image = fileobj.original_filename
  end
end

def thumbnail_file=(fileobj)
  if fileobj.size > 0
```

Beispiel 4-14: Post#image_file= und Post#thumbnail_file= (Fortsetzung)

```
    @thumbnail_file = fileobj
    self.thumbnail = fileobj.original_filename
  end
end
```

Sie erinnern sich? In der Action PostsController#create wird ganz am Anfang eine Instanz des Models erzeugt und mit den Formularwerten initialisiert. Genau an dieser Stelle kommen diese beiden Methoden zum Einsatz, wenn auch nur intern. Lassen Sie die beiden weg, beschwert sich Rails, denn das Model kann nichts mit den Attributen `image_file` und `thumbnail_file` anfangen, da entsprechende Akzessoren fehlen. Jetzt aber steht ganz klar fest: Die beiden Dateiobjekte werden in `@image_file` beziehungsweise `@thumbnail_file` gesichert. Außerdem werden die reinen Dateinamen aus der Eigenschaft `original_filename` entnommen und den entsprechenden Attributen des Models zugeordnet. Dieser Vorgang soll nur stattfinden, wenn die Größe der hochgeladenen Datei 0 übersteigt. Sollte dies nicht der Fall sein, so bleiben `Post#image` und / oder `Post#thumbnail` leer.

Nun gibt es noch ein paar Problemchen: Es ist möglich, dass Browser in `original_filename` nicht nur den Dateinamen, sondern den kompletten lokalen Pfad übergeben. Und noch etwas: Es kann ja sein, dass der Picsblog-Besitzer Dateien hoch lädt, dessen Namen schon auf dem Server existieren. So manch einer nennt seine floralen Motive schlicht stets nur *blume.jpg*. Um ihn dann nicht jedes Mal mit lästigen »Gibt's schon!«-Hinweisen zu belästigen, stellen wir dem Dateinamen einfach noch etwas voran, was ihn einzigartig macht. Da jede Sekunde – *diese* hier, und *die*, und *die* auch – einzigartig ist, arbeiten wir an dieser Stelle einfach auf Basis sekundengenauer Zeit.

Um diesen beiden möglichen Stolpersteine aus dem Weg zu räumen, lohnt es sich, eine eigene Methode zu schreiben. Das ist auch deshalb eine gute Idee, weil diese Funktionalität sowohl in `image_file_field=` als auch in `thumbnail_file_field=` gebraucht wird.

Die Methode `unique_and_proper_filename` soll nur ein kleiner Helfer sein. Daher empfiehlt es sich, sie als private zu deklarieren. Ordnen Sie also die Methode am besten am Ende des Klassenkörpers ein und setzen Sie ein private darüber.

Beispiel 4-15: unique_and_proper_filename findet einen korrekten, einzigartigen Dateinamen

```
private
def unique_and_proper_filename(filename)
  Time.now.to_i.to_s + '_' + File.basename(filename)
end
```

Mit `Time.now` erhalten Sie die aktuelle Zeit inklusive Datum. Mit `to_i` wandeln Sie diesen Zeitwert in das UNIX-Format um. Diese Art der Zeitdarstellung kennen Sie bestimmt. Die UNIX-Zeit gibt an, wie viele Sekunden seit dem Beginn des UNIX-

Zeitalters am 1. Januar 1970 bereits vergangen sind. Das heißt, UNIX-Zeit besteht nur aus einer Zahl. Das macht sie attraktiv für den Beginn eines Dateinamens. Ein Bild namens *rubrecht_beim_grillen.jpg* wird dank unserer Hilfsmethode und wenn es beispielsweise am 15. Januar 2007 um 13:30 Uhr auf den Server geladen wurde, zu *1168864200_rubrecht_beim_grillen.jpg*.

Die Klassenmethode `File#basename` sorgt dafür, dass ein eventueller Pfad in einem Dateinamen abgeschnitten wird.

Nachdem `unique_and_proper_filename` implementiert ist, können Sie die Methode in den beiden vorhin geschriebenen Setter-Methoden nutzen.

Beispiel 4-16: unique_and_proper_filename im Einsatz

```
def image_file= (fileobj)
  if fileobj.size > 0
    @image_file = fileobj
    self.image = unique_and_proper_filename(fileobj.original_filename)
  end
end

def thumbnail_file= (fileobj)
  if fileobj.size > 0
    @thumbnail_file = fileobj
    self.thumbnail = unique_and_proper_filename(fileobj.original_filename)
  end
end
```

In der Controller-Action `create` wird die Frage gestellt, ob die Formularangaben auch `valid?` sind. Sie sind es dann, wenn alle Felder ausgefüllt worden sind. Das zu überprüfen können Sie gern Rails' eingebauten *Validierungsmechanismen* überlassen.

Eingaben validieren – einfach und schnell

Diese Mechanismen greifen immer schon, bevor etwas in die Datenbank geschrieben wird, so dass fehlerhafte oder unzureichende Eingaben erst gar nicht mit einer Speicherung gewürdigt werden. Dank *Metaprogrammierung* können Sie Validierungsrichtlinien in Rails ganz einfach definieren. Picsblog benutzt dabei die einfachste und gebräuchlichste Methode, `validates_presence_of`.

Ein Beitrag darf dann aufgenommen werden, wenn der resultierende Datensatz *Titel*, *Beschreibung*, *Datum*, *Dateiname des Fotos* und *Dateiname des Thumbnails* enthält. Sonst nicht. Wenn Sie sich erinnern, was Metaprogrammierung bedeutet, werden Sie über die Kürze des folgenden Quelltextes, der das eben beschriebene leistet, nicht verwundert sein. Notieren Sie folgende Zeile am besten gleich zu Beginn des Klassenkörpers Ihres Post-Models.

```
validates_presence_of(:title, :description, :date, :image, :thumbnail)
```

Fehlt eine der Angaben beim Speichern eines Datensatzes, so gibt die Methode `valid?` des Post-Models `false` zurück. Die Methode nutzen Sie in `PostsController#create`.



Die Methode `save`, die einen Datensatz speichert, würde ihren Dienst verweigern, wenn die Anforderungen nicht erfüllt sind und die Validierung fehl schlägt. In den meisten Fällen können Sie sich die explizite Abfrage über `valid?` also ersparen und direkt mit `save` arbeiten.

Neben `validates_presence_of` hält Ruby on Rails in Active Record weitere Methoden bereit, mit denen Sie weitere Regeln für erlaubte Daten festlegen können. Mehr Informationen dazu finden Sie im Anhang .

Dateien speichern

Sind alle erforderlichen Daten vorhanden, veranlasst `PostsController#create` die Model-Instanz `@post`, die beiden Bilddateien zu speichern. Dazu wird `Post#save_files` aufgerufen, dessen Rückgabewert bestimmt, ob das Speichern erfolgreich war.

Beispiel 4-17: Dateien speichern mit save_files

```
def save_files

  # Bilddatei speichern
  if !save_uploaded_file(@image_file, IMAGE_DIR, self.image)
    return false
  end

  #Thumbnail speichern
  if !save_uploaded_file(@thumbnail_file, THUMBNAIL_DIR, self.thumbnail)
    return false
  end

  return true
end
```

Wie Sie sehen, kommen hier die beiden Objekte wieder zum Einsatz, die einst an `image_file=` und `thumbnail_file=` übergeben worden sind und nun durch `@image_file` und `@thumbnail_file` angesprochen werden können. Die beiden Objekte werden benötigt, weil in ihnen die Daten stecken, aus denen die Methode `save_uploaded_file` Dateien formen soll. Dateinamen inklusive Speicherort (relativ zum `public`-Verzeichnis) werden der Methode übergeben.

Auch `save_uploaded_file` ist wie `save_files` ein Eigengewächs. Da sie auch nur eine Hilfsmethode und außerhalb des Models zudem ohne Belang ist, deklarieren Sie `save_uploaded_file` einfach als `private`. Notieren Sie also folgende Zeilen nach `unique_and_proper_filename` in Ihrer Post-Klasse.

Beispiel 4-18: Eine hoch geladene Datei wird gespeichert

```
def save_uploaded_file(fileobj, filepath, filename)

  # Kompletter Pfad
  complete_path = RAILS_ROOT + '/public/' + filepath

  # Falls notwendig, Verzeichnis erstellen
  FileUtils.mkdir_p(complete_path) unless File.exists?(complete_path)

  # Datei speichern
  begin
    f = File.open(complete_path + '/' + filename, 'wb')
    f.write(fileobj.read)
  rescue
    return false
  ensure
    f.close unless f.nil?
  end
end
```

Diese Methode speichert die in `fileobj` übergebenen Daten in die in `filename` angegebene Datei an einem Ort, der durch `filepath` bestimmt wird. Das Speichern einer Datei kennen Sie ja bereits aus Kapitel 3.

Zu Beginn der Methode wird der komplette Pfad aus allerlei Einzelteilen zusammengesetzt. Hier wird der absolute Pfad unserer Rails-Applikation, der immer automatisch in der Konstante `RAILS_ROOT` steht, um das Unterverzeichnis *public* und das Bild- beziehungsweise Thumbnail-Verzeichnis ergänzt. So entsteht der komplette Pfad zu der Datei, die gespeichert werden soll.

Anhand dieses Pfades und mit Hilfe von `File#exists?` überprüft die Methode dann, ob es diesen Pfad schon gibt. Falls nicht, wird er mit `FileUtils#mkdir_p` erstellt.



Vielleicht ist Ihnen aufgefallen, dass der Dateimodus, der in `File#open` gesetzt wird, ein *b* enthält. Dies ist eine reine Vorsichtsmaßnahme für mögliche Probleme mit Windows-Rechnern. Aber auch alle anderen Betriebssysteme kommen damit klar, wenn die Datei im Binärmodus geöffnet wird.

Allerdings möchte ich Sie noch auf zwei Dinge aufmerksam machen. Mit `RAILS_ROOT`, einer Konstante, die jede Rails-Applikation automatisch zur Verfügung steht, können Sie den lokalen Pfad zu Ihrer Anwendung erfragen. Darauf aufbauend wird der absolute Pfad zu der neuen Datei zusammengesetzt. Weiterhin interessant ist der Modus, in welchem die neue Datei auf dem Server geöffnet wird. Mit `wb` sorgen Sie dafür, dass Uploads auf Windows-Rechner auch garantiert funktionieren. Dabei wird die Datei im Binärmodus geöffnet. Andere Betriebssysteme stören sich nicht weiter daran.

Wenn das Speichern der Dateien auf dem Server erfolgreich war, fordert die Action `create` die Model-Instanz `@post` mit dessen Methode `save` zum Speichern der Bilddaten in der Datenbank auf. Sie brauchen wirklich nur diese eine Methode aufrufen, die in jedem Rails-Model Standard ist, um zu speichern. SQL-Akrobatik, wie sie bei herkömmlicher Webentwicklung anstünde, ist an dieser Stelle nicht nötig.

Wenn Sie nun <http://127.0.0.1:3000/posts/new> aufrufen, lassen Sie doch einmal alle Felder leer und klicken Sie auf Speichern. Seien Sie ehrlich, selten war eine Fehlermeldung so erfreulich. Aber auch mit kompletten Daten und zwei Dateien funktioniert Picsblog schon. Laden Sie doch mal ein Bild mit Thumbnail hoch.

Das Layout ist darauf ausgelegt, dass Ihre Fotos maximal 610 Pixel breit sein sollten. Bei der Wahl der Thumbnailgröße sind Sie etwas freier, wobei es sich empfiehlt, für alle Thumbnails die gleiche Größe zu wählen.



Es gibt natürlich auch eine serverseitige Lösung, um nach dem Upload eines Fotos automatisch ein Thumbnail zu erstellen. Aber da ich Sie lieber mit den wirklich wichtigen Rails-eigenen Funktionalitäten vertraut machen, als Sie mit Algorithmen zum Bildverkleinern langweilen möchte, beschränken wir uns auf das Hochladen bereits fertiger Thumbnails.

Sollten Sie dennoch Interesse an der automatischen Variante haben, empfehle ich Ihnen den Besuch der Website <http://rmagick.ruby-forge.org>. Windows-Benutzern sei aber schon hier gesagt, dass die Installation nicht ganz einfach ist.

Lassen Sie sich bei erfolgreichem Abschluss des Vorhabens nicht durch Rails' Hinweis stören, dass die Action `list` noch nicht existiert. Die schieben wir sofort nach. Zuvor werfen Sie doch kurz mal einen Blick in die Data-Perspektive von RadRails. Dort können Sie sehen, dass Ihre Daten ordnungsgemäß eingetragen wurden.



Sie werden in Ihrer Datenbank `picsblog_development` überraschenderweise auch auf eine Tabelle namens `schema_info` treffen. Die hat Rails angelegt, als Sie die erste Migration des Projekts einpflegten. Hier wird festgehalten, mit welcher Migration Ihre Anwendung gerade arbeitet. Das ist wichtig für das Up- und Downgrade Ihrer Datenbankstruktur.

Nun aber zur Action `list`. Schließlich sollen Sie sich ja an den hochgeladenen Fotos auch in Bild- und nicht nur in Textform erfreuen.

Alle Fotos im Überblick

Die Action `list` soll dem Picsblog-User eine Übersicht über alle vorhandenen Bilder geben. Auf Controller-Seite ist der entsprechende Quellcode schnell geschrieben.

Beispiel 4-19: PostsController#list

```
def list
  @posts = Post.find(:all, :order => 'date DESC')
end
```

Das war's schon. Das Herz (und einziges Organ) von PostsController#list ist die Methode `find`, die Active Record entstammt und ein ziemlich mächtiges Werkzeug ist, das Sie oft benutzen werden. Find sammelt Datensätze und kann auf drei verschiedene Arten genutzt werden.

1. `:first` – find gibt den ersten Datensatz einer Datenbanktabelle zurück, wobei auf Wunsch Bedingungen angegeben werden können.
2. ID oder IDs – find gibt einen oder mehrere Datensätze zurück, die der oder den gesuchten Datensatz-ID(s) entspricht. Die gewünschten IDs können als einzelne Zahl, als mehrere durch Komma getrennte Zahlen oder als Array übergeben werden.
3. `:all` – find gibt alle Datensätze einer Datenbanktabelle zurück, wobei über die Definition von Bedingungen Einschränkungen gemacht werden können.

Wenn Sie sich für eine Variante von `find` entschieden haben, besteht die Möglichkeit, zusätzliche Optionen als Hash-Elemente zu definieren. In der `list`-Action wird beispielsweise durch den Schlüssel `order` eine Sortierreihenfolge festgelegt. Der Wert ist ein SQL-Fragment, hier `date DESC`, wodurch die Picsblog-Beiträge in der Übersicht, an der wir gerade basteln, chronologisch dargestellt werden. Und zwar absteigend, was durch `DESC` deutlich gemacht wird.

Beispielanwendungen für find

Da `find` einen hohen Stellenwert in der Rails-Programmierung besitzt, lohnt sich ein Blick auf einige Einsatzbeispiele für diese findige Methode.

- `Post.find(1)` # Gibt den Datensatz mit der ID 1 als Instanz von `Post` zurück.
- `Post.find(10,25,67)` # Gibt die Datensätze mit den IDs 10, 25 und 67 als ein Array aus `Post`-Objekten zurück.
- `Post.find(:first)` # Gibt den Datensatz als `Post`-Objekt zurück, der physisch als erstes in der Datenbanktabelle steht.
- `Post.find(:first, :order => "date DESC", :offset => 5)` # Gibt ein `Post`-Objekt wieder. Es handelt sich um das 6., wenn alle Datensätze chronologisch sortiert werden.
- `Post.find(:all, :order => "date DESC", :offset => 5, :limit => 10)` # Gibt ein Array von maximal 10 `Post`-Objekten zurück, die alle chronologisch geordnet sind. Das erste Objekt ist dabei das 6. der Sortierung.

Das Array von Datensätzen, welches in @posts steckt, können Sie nun in der View nutzen, um die Thumbnails aller Beiträge anzeigen zu lassen. Da nichts Anderweitiges in PostsController#list vereinbart wurde, sucht Rails nach *list.rhtml* in *app/views/posts*.

Bessere Views durch Partials

Der RHTML-Code des list-Views erscheint recht kurz, aber er birgt auch eine Spezialität von Rails in sich.

Beispiel 4-20: *list.rhtml*

```
<h2>&Uuml;bersicht</h2>
<%= 'Keine Bilder enthalten.' if @posts.empty? %>
<%= render(:partial => 'thumbnail', :collection => @posts) %>
```

Hier kommt ein so genanntes *Partial* zum Einsatz. Partials sind spezielle Templates, die dann eingesetzt werden, wenn ein Stück (R)HTML innerhalb eines Views ständig wiederholt werden muss oder in verschiedenen Views zum Einsatz kommt. Partials unterstützen das DRY-Prinzip von Ruby on Rails aufs Vortrefflichste.

Warum kommt aber hier ein Partial zum Einsatz? Die Antwort liegt in der Natur einer Übersicht. Sie soll später einmal eine große Menge an Thumbnails anzeigen. Jedes Thumbnail wird dabei mit Hilfe von ein paar HTML-Tags angezeigt, die sich für jedes Vorschaubildchen ständig wiederholen. Mit einem Partial quelltexten Sie diesen Teil nur einmal. Rails sorgt dann dafür, dass ein Partial so oft angezeigt und der enthaltene ERb-Code so oft ausgeführt wird, wie es erforderlich ist. In unserem Fall ergibt sich die Häufigkeit aus der Anzahl der Elemente des Arrays @posts. Für jedes Element von @posts wird das Partial gerendert, wobei die Daten des betroffenen Elements jeweils innerhalb des Partials zur Verfügung stehen. Dadurch können Sie beispielsweise innerhalb des Partials den URL des Thumbnails bilden, so dass es angezeigt werden kann.

Durch das Schlüssel-Wert-Paar `:partial => 'thumbnail'` wird ein Template als Partial genutzt, welches im *views*-Ordner des aufrufenden Controllers, hier also in *apps/views/posts*, als *_thumbnail.rhtml* vorliegen muss. Der Unterstrich zu Beginn des Dateinamens ist dabei ganz wichtig. Über den Schlüssel `:collection` erhält das Partial seine Datenbasis, hier das Arrays aus Post-Instanzen, @posts.

Bei jedem Durchlauf wird das jeweilige Element von @posts durch die Variable `thumbnail` referenziert. Etwas ähnliches kennen Sie bereits von Rubys Iteratoren, von `each` zum Beispiel. Da zeigte die Blockvariable auf das aktuelle Element. Bei Partials ist die Wahl des Bezeichners im Gegensatz zur Blockvariable von Iteratoren nicht frei, sondern gehorcht einer Konvention: Der Variablenbezeichner muss mit dem Namen des Partials, hier `template`, übereinstimmen.

Beispiel 4-21: _thumbnail.rhtml – ein Partial für viele Thumbnails

```
<div>
  <%= link_to(image_tag(THUMBNAIL_DIR + '/' + thumbnail.thumbnail, :title => thumbnail.title), {:action => 'show', :id => thumbnail.id}) %>
  <p>
    <%= h(thumbnail.title) %><br />
    <%= h(thumbnail.description) %><br />
    <%= link_to('Bearbeiten', {:action => :edit, :id => thumbnail.id}) %>
    <%= link_to('Löschen', {:action => :delete, :id => thumbnail.id}, :confirm =>
'Möchten Sie das Bild wirklich löschen?') %>
  </p>
</div>
```

So ein Partial ist eine tolle Sache, die Sie sicher bald schätzen werden. Sie müssen aber keine Partials verwenden. Sie können selbstverständlich den Inhalt des Partials auch in *list.rhtml* unterbringen und dort mit einer Schleife arbeiten, die über die Elemente des Arrays @posts iteriert. Folgende Zeilen beinhalten genau das.

Beispiel 4-22: list.rhtml, diesmal ohne Partial

```
<h2>&Uuml;bersicht</h2>
<%= 'Keine Bilder enthalten.' if @posts.empty? %>
<%= render(:partial => 'thumbnail', :collection => @posts) %>
<% for thumbnail in @posts %>
  <div>
    <%= link_to(image_tag(THUMBNAIL_DIR + '/' + thumbnail.thumbnail, :title =>
thumbnail.title), {:action => 'show', :id => thumbnail.id}) %>
    <p>
      <%= h(thumbnail.title) %><br />
      <%= h(thumbnail.description) %><br />
      <%= link_to('Bearbeiten', {:action => :edit, :id => thumbnail.id}) %>
      <%= link_to('Löschen', {:action => :delete, :id => thumbnail.id}, :confirm =>
'Möchten Sie das Bild wirklich löschen?') %>
    </p>
  </div>
<% end %>
```

Für welche Variante Sie sich entscheiden, bleibt ganz Ihnen überlassen. Aber mal unter uns: Ich würde mit einem Partial arbeiten. Das ist übersichtlicher.

Blicken wir aber nun in das Innere des Partials (oder der for-Schleife): Auf dem ersten Blick fallen Ihnen sicher ein paar *Tag-Helper* ins Auge. Zur Darstellung des Thumbnails kommt *image_tag* zum Einsatz. Damit erzeugen Sie ein *img*-Element in HTML nach eigenen Wünschen. Der erste Parameter von *image_tag* entspricht dabei dem Wert des *src*-Attributs von **. Hier geben Sie also den URL des Thumbnails an. Bedenken Sie dabei, dass diese Angabe relativ zum *public*-Verzeichnis sein muss, da dies bekanntlich die Wurzel Ihrer Rails-Website ist.

Hier erhält das *img*-Element noch einen Wert für das *title*-Attribut. Ist das gesetzt, erscheint ein Tooltip beim Überfahren des Bildes mit der Maus.

Das Bild umgibt ein Link-Tag, HTMLern als `<a>` bekannt. Damit erreichen Sie, dass das Bild verlinkt und somit anklickbar wird. Als ersten Parameter erwartet `link_to` das, was verlinkt werden soll. Hier ist es ein Bild, es kann aber auch ein simpler Text sein.

Wie Sie sehen, ist der Link des Thumbnails auf die Action `show` des Posts-Controllers gerichtet, die das Originalbild zeigen soll. Dem aufrufenden URL wird zu diesem Zwecke noch der Parameter `id` angehängt. `PostsController#show` muss ja schließlich wissen, welcher Beitrag en detail angezeigt werden soll. Dadurch ergibt sich folgender URL: `http://127.0.0.1:3000/posts/show/1` – vorausgesetzt, es handelt sich um den Datensatz mit der ID 1.

Weiter unten im Partial finden Sie noch zwei weitere Links, die beitragsspezifische Aktionen ausführen und daher ebenfalls mit der ID des betroffenen Beitrags versorgt werden müssen. Die Actions `edit` und `delete` schreiben Sie in Kürze.

Beim Löschen-Link muss unbedingt hervorgehoben werden, dass dieser nur ausgeführt wird, wenn der Löschwillige sein destruktives Ansinnen durch Bestätigung eines Dialogfensters verfestigt. JavaScript-Programmierer kennen vielleicht den Befehl `confirm` und seine Auswirkungen. Genau der kommt hier zum Einsatz. Allerdings kann diese JavaScript-Funktion hier dank `link_to` viel komfortabler genutzt werden. Keine `onclick`-Behandlung, kein Auswerten der Button-Wahl – das erledigt alles der `link_to`-Helper. Wird die Nachfrage verneint, erfolgt natürlich auch kein Aufrufen der Action `delete`, das ist klar.

Ein kleiner Hinweis soll noch die fast zu übersehende Methode `h` würdigen. Sie sorgt dafür, dass Titel und Beschriftung des Partials korrekt dargestellt werden. HTML ist schließlich manchmal etwas penibel, was Sonderzeichen angeht. Mit `h` ist das kein Problem, da problematische Zeichen zu unproblematischen gewandelt werden. Die Methode ist auch ganz brauchbar, wenn es um die Vermeidung von böswilligen Attacken mit eingeschleusten Script-Tags geht.

Rufen Sie nun `http://127.0.0.1:3000/posts/list` auf. Sie sollten nun das Thumbnail, Titel und Beschreibungstext Ihres vorher hoch geladenen Fotos sehen. Am besten, Sie laden gleich noch ein zweites, drittes, viertes hoch. Eine Übersicht macht dann einfach mehr Sinn – und wir brauchen gleich mindestens zwei Beiträge.

Zu Beginn des Artikels habe ich Ihnen versprochen, dass diese Übersicht eine *Pagierung* erhält. Ab einer gewissen Anzahl an Fotos sollen also Links zu Seiten mit weiteren Fotos erscheinen. Mit Ruby on Rails ist dieses Ansinnen kein Problem.

Fotos seitenweise

Auch für diesen Zweck gibt es bereits vorgefertigte Lösungen in Ruby on Rails, sowohl für die Controller- als auch für die View-Ebene. Um die entsprechende

Übersicht



Sonnenuntergang
Ende eines Sommertages über den Dächern Berlins
[Bearbeiten](#) [Löschen](#)



Herbst
Laub am Fluss in bunten Farben
[Bearbeiten](#) [Löschen](#)

Abbildung 4-3: Es geht voran: Picsblog kann schon eine Übersicht anzeigen.

Datenbasis für die Paginierung im Allgemeinen und den `PaginationHelper`, der im View zum Einsatz kommt, im Besonderen zu erzeugen, passen Sie `PostsController#list` folgendermaßen an:

```
def list
  @post_pages, @posts = paginate(:posts, :order => 'date DESC', :per_page => 15)
end
```

Statt eine Klassenmethode des Models zu nutzen, das alle Datensätze liefert und in `@posts` speichert, nutzen Sie nun die Methode `paginate` und geben an, auf welche Datenbanktabelle sich die Paginierung beziehen soll. Das Ergebnis ist eine Instanz der Klasse `Paginator`, die dem Sub-Framework Action Controller entstammt, und gleich in der View verwendet wird. In `@posts` befinden sich nun nur noch Instanzen des Post-Models, welche für die aktuelle Seite der gleich paginierten Übersicht benötigt werden. Die maximale Anzahl von Thumbnails pro Übersichtsseite können Sie durch `per_page` definieren.

Auf View-Seite müssen Sie nur noch Links zu den Einzelseiten der Übersicht einfügen. Der bereits genannte `PaginationHelper` bietet Ihnen dazu als einfachste Variante die Methode `pagination_links` an. Als Parameter übergeben Sie der Methode die im Controller erzeugte Paginator-Instanz `@post_pages`. So brauchen Sie `list.rhtml` nur um eine einzige Zeile erweitern.

```
<h2>&Uuml;bersicht</h2>
<%= 'Keine Bilder enthalten.' if @posts.empty? %>
<%= render(:partial => 'thumbnail', :collection => @posts) %>
<%= pagination_links(@post_pages) %>
```

Sie können zum Ausprobieren den Wert von `per_page` innerhalb der `paginate`-Methode von `PostsController#list` kurzzeitig auf 1 setzen. Dann können Sie das Ergebnis Ihrer Anstrengungen schon jetzt sehen und müssen nicht erst 15 Beiträge anlegen.

Egal ob paginiert oder nicht, jedes Thumbnail linkt zu drei Actions, die wir noch implementieren müssen: `show` (Anzeige des Originalbilds bei Klick auf das Thumbnail), `edit` (Bearbeiten der Bilddaten) und `delete` (Löschen eines Fotos aus Datenbank und Dateisystem).

Beiträge bearbeiten

Lassen Sie uns mit `edit` weitermachen. Ihr Wissen über Partials ist gerade so herrlich frisch, dass es sich lohnt, es gleich wieder anzuwenden. Bevor es soweit ist und Sie sich dem View der Action zuwenden können, sollten Sie erst die Action schreiben, auf deren Daten die View basiert.

Denken Sie beim Betrachten des folgenden Quelltextchens an die Art und Weise, wie die Action aus der Übersicht heraus aufgerufen wird. Denn ihr wird dabei die ID des zu editierenden Datensatzes übertragen. Konkret heißt es im thumbnail-Partial `<%= link_to('Bearbeiten', {:action => :edit, :id => thumbnail.id}) %>`. Damit `edit` den richtigen Datensatz aus der Datenbank fischen kann, muss sie sich also des Parameters `id` ermächtigen.

Beispiel 4-23: PostsController#edit

```
def edit
  @post = Post.find(params[:id])
end
```

Die Methode `find` haben Sie schon wenige Seiten zuvor kennen gelernt. In der Action `edit` wird sie genutzt, um einen bestimmten Datensatz zu finden. Dessen `id` wurde durch den Rails-Dispatcher aus dem URL gefischt und liegt nun in `params[:id]` vor. Mit `@post` im Gepäck geht's zur View, die in `app/posts/edit.rhtml` vorliegen sollte.

Denken Sie daran: in `@post` stecken alle Daten des Beitrags. Wenn Sie beispielsweise `text_field` mit dem Namen des Objekts versorgen, befüllt es ein `input`-Element mit dem Wert, der in der Datenbank in einem bestimmten Feld, das Sie mit dem zweiten Parameter bestimmten, enthalten ist. So zeigt das Formular also die Daten an, die Sie ändern können.

Beispiel 4-24: edit.rhtml

```
<h2>Foto bearbeiten</h2>
<%= image_tag(THUMBNAIL_DIR + '/' + @post.thumbnail) %>
<%= form_tag({:action => 'update', :id => @post.id}, :multipart => true) %>
<h3>Bilddaten</h3>
<p>Titel<br/>
<%= text_field(:post, :title) %></p>
<p>Beschreibung<br/>
<%= text_field(:post, :description) %></p>
<p>Datum und Uhrzeit<br/>
<%= datetime_select(:post, :date, :order => [:day, :month, :year, :hour]) %><p>
<p><%= submit_tag('&Uuml;bernehmen') %></p>
</form>
```

Grundsätzlich steckt hier nichts Neues drin. Zu Beginn zeigt die View das Thumbnail des Bildes an, dessen Daten hier bearbeitet werden können. Das HTML-Formular hat die Action update des Posts-Controllers zum Ziel, wobei noch die ID des betroffenen Datensatzes mitgeschickt wird.

Foto bearbeiten



Bilddaten

Titel

Beschreibung

Datum und Uhrzeit

[Zurück zur Übersicht](#)

Abbildung 4-4: Foto bearbeiten

Sollten Ihnen Ähnlichkeiten zu new.rhtml aufgefallen sein, so sind Sie schon auf der richtigen Spur. Denn in der Tat gibt es Teile des Codes, die in beiden Views identisch sind. Das betrifft die Eingabefelder für den Titel und die Beschreibung, sowie

das Element zur Datums- und Zeitwahl. Um das DRY-Prinzip von Rails zu würdigen, sollte hier ein Partial her. Rendert `list.rhtml` ein Partial mehrfach innerhalb der View, so kommt hier ein Partial zum Einsatz, das Sie nur einmal, aber in mehreren Views nutzen.

Erstellen Sie eine neue Datei in `app/views/posts` und nennen Sie diese `_details_form.rhtml`. Vergessen Sie den Unterstrich nicht.

Beispiel 4-25: details_form.rhtml

```
<h3>Bilddaten</h3>
<p>Titel<br/>
<%= text_field(:post, :title) %></p>
<p>Beschreibung<br/>
<%= text_field(:post, :description) %></p>
<p>Datum und Uhrzeit<br/>
<%= datetime_select(:post, :date, :order => [:day, :month, :year, :hour]) %><p>
```

Nun können Sie den Quelltext-Teil, den Sie gerade ausgelagert haben, aus `new.rhtml` und `edit.rhtml` herausnehmen und stattdessen jeweils die Zeile

```
<%= render(:partial => 'details_form') %>
einsetzen. Danach sieht new.rhtml wie folgt aus:
```

Beispiel 4-26: new.rhtml – mit Partial

```
<h2>Neues Foto anlegen</h2>
<%= form_tag({:action => :create}, :multipart => true) %>
<h3>Datei-Upload</h3>
<p>Bilddatei:<br/>
<%= file_field(:post, :image_file) %></p>
<p>Thumbnail:<br />
<%= file_field(:post, :thumbnail_file) %></p>
<%= render(:partial => 'details_form') %>
<%= submit_tag('Speichern') %></p>
</form>
```

Die Datei `edit.rhtml` hat ebenfalls an Umfang abgenommen und präsentiert sich nun so:

Beispiel 4-27: edit.rhtml – mit Partial

```
<h2>Foto bearbeiten</h2>
<%= image_tag(THUMBNAIL_DIR + '/' + @post.thumbnail) %>
<%= form_tag({:action => 'update', :id => @post.id}, :multipart => true) %>
<%= render(:partial => 'details_form') %>
<p><%= submit_tag('&Uuml;bernehmen') %></p>
</form>
```

Stellen Sie sich vor, Sie möchten eines Tages noch ein weiteres Eingabefeld hinzufügen. Das geht ganz schnell mit dem hier verwendeten Partial. Ihre Änderungen müssen Sie nur einmal vornehmen, damit sie in zwei Views zu sehen sind.

Ach ja, Sie können jetzt ein Bild bearbeiten. Klicken Sie doch einmal den Bearbeiten-Link eines Fotos auf der Übersichtsseite an. Zugegeben, ein Problem gibt es dabei noch. Ihre Änderungen werden nämlich noch nicht gespeichert.

Geänderte Beiträge speichern

Was die Action `create` für `new` ist, soll `update` für `edit` sein. Soll heißen: `PostsController#update` soll die Daten des Formulars, welches Sie eben implementiert haben, entgegen nehmen und dem Model zur Speicherung übergeben.

Beispiel 4-28: PostsController#update

```
def update
  @post = Post.find(params[:id])
  if @post.update_attributes(params[:post])
    flash[:notice] = "Ihre Änderungen wurden gespeichert."
    redirect_to(:action => :list)
  else
    Flash.now[:notice] = "Es trat ein Fehler auf.
Bitte füllen Sie alle Felder aus und überprüfen Sie Ihre Angaben."
    render(:action => :edit)
  end
end
```

Um den Code der `update`-Action zu verstehen, sollten Sie sich zunächst bewusst werden, was der HTTP-Request, der durch das Absenden des Bearbeiten-Formulars erzeugt wurde, alles liefert. In der `form_tag`-Methode des Formulars haben Sie die ID des Datensatzes notiert. Innerhalb der Action finden Sie die Angabe ein weiteres Mal in `params[:id]` wieder. Damit lässt sich der Datensatz, an dem die Änderungen vorgenommen werden sollen, wieder bestens aus der Datenbank holen.

Des weiteren haben wir im Formular ein Objekt namens `post` benutzt, um die Eingabefelder mit den zu bearbeitenden Daten zu befüllen. Im gleichen Objekt stecken jetzt natürlich auch die Werte nach der Bearbeitung. Mit `params[:post]` kommen Sie im Controller an die neuen Daten.

Zunächst organisiert `find` den zu verändernden Datensatz und speichert ihn in `@post`. In diese `Post`-Instanz werden nun die bestehenden Daten durch die neuen Werte aus dem Bearbeiten-Formular (*Titel*, *Beschreibung* und *Datum*) überschrieben. Ihr `Post`-Model kennt zu diesem Zweck, erneut durch vorteilhaftes Erben begründet, die Methode `update_attributes`.

Auch bei `update_attributes` greift übrigens die Validierung, deren Regeln Sie einst im `Post`-Model definiert haben. Sie wird bei allen Aktivitäten an der Datenbank aktiv, die mit Speichern zu tun haben. So kann es also sein, dass `update_attributes` nicht erfolgreich ausgeführt werden kann. In diesem Fall wird eine Fehlermeldung erzeugt und das Formular erneut angezeigt. Falls alles in Ordnung war, zeigt sich wieder die Übersicht der Fotos. Dort können Sie ein Foto auch löschen.

Löschen von Beiträgen

Die Action `delete` macht sich eine Methode des Sub Frameworks Active Record zu nutze, die einen Datensatz anhand seiner ID löscht und eine Model-Instanz zurück gibt, die den gelöschten Datensatz enthält. Sie heißt `destroy`. Und die ID bekommt die Action wieder aus dem URL, dafür haben Sie schon im Partial `_thumbnail.rhtml` gesorgt.

Beispiel 4-29: PostsController#delete

```
def delete
  @post = Post.destroy(params[:id])
  flash[:notice] = "Das Bild <strong>#{@post.title}</strong> wurde gelöscht."
  redirect_to(:action => :list)
end
```

In `@post` steckt der gelöschte Datensatz. Es ist also eine Leichtigkeit, eine Hinweismeldung mit dem Titel des gelöschten Fotos zu erzeugen. Die erscheint, wenn die Übersichtsseite im Browser angezeigt wird. Zu dieser leitet `redirect_to` hin.

Nun soll aber nicht nur der Datensatz entfernt werden. Auch die beiden Bilddateien müssen natürlich von der Platte geputzt werden. Und das ist eindeutig Angelegenheit des Models. Dieses hält auch eine ganz interessante Funktionalität für solche Zwecke bereit: *Callbacks*.

Callbacks im Model

Wann immer etwas innerhalb des Models in Zusammenhang mit Daten passiert, wird eine Callback-Methode ausgeführt – natürlich nur, wenn Sie eine entsprechende definiert haben.

Auf diese Weise können Sie in einem Model Methoden schreiben, die beispielsweise dann ausgeführt werden, *bevor* ein Datensatz gespeichert wird, *nachdem* ein Datensatz erzeugt wurde und eben auch *bevor* ein Datensatz gelöscht wird.

Eine Methode namens `before_destroy` wird dazu vor dem Entfernen eines Datensatzes ausgeführt. Wenn Sie eine solche Methode in den Quelltext Ihres Models aufnehmen und im Methodenkörper Dateien gelöscht werden, dann existieren die schon gar nicht mehr, wenn der Datensatz ausgelöscht wird.

Beispiel 4-30: Post#before_destroy

```
def before_destroy
  imagefile = RAILS_ROOT + '/public/' + IMAGE_DIR + '/' + self.image
  thumbnailfile = RAILS_ROOT + '/public/' + THUMBNAIL_DIR + '/' + self.thumbnail
  File.delete(imagefile) if File.exists?(imagefile)
  File.delete(thumbnailfile) if File.exists?(thumbnailfile)
end
```

Die Klassenmethode `delete` der Klasse `File` kümmert sich um das Entsorgen der nun überflüssigen Dateien. Über die Attribute `image` und `thumbnail` des Models werden, ein letztes Mal, die Dateinamen ermittelt. Mit einem Modifikator, der die Existenz der zu löschen Dateien zur Bedingung macht, man weiß ja nie, wird sichergestellt, dass `File#delete` fehlerfrei arbeiten kann.

Die Grundfunktionalität von Picsblog ist hiermit fast fertig. Es fehlt noch die Detailansicht, die das Foto eines Beitrags in voller Schönheit präsentiert. Auf so einem winzigen Thumbnail wirkt ein Foto irgendwie nicht.

Beiträge anzeigen

Nun soll die Action `show` entstehen. Auf Controller-Seite gilt es ein weiteres Mal, den richtigen, einer übergebenen ID entsprechenden Datensatz aus der Datenbanktabelle zu holen.

Beispiel 4-31: PostsController#show

```
def show
  @post = Post.find(params[:id])

  # Vorheriges / nächstes Foto
  @posts = Post.find(:all, :order => 'date DESC')
  @next_post = @posts[@posts.index(@post) + 1] unless @post == @posts.last
  @prev_post = @posts[@posts.index(@post) - 1] unless @post == @posts.first
end
```

Doch `PostsController#show` soll noch mehr können. Über Links, die zum vorherigen und nachfolgenden Foto führen, können Sie den Benutzern von Picsblog das Blättern durch die Fotoausstellung erleichtern. Die Reihenfolge ist dabei dieselbe, die auch in der Übersicht gilt.

In `@posts` stecken alle Beiträge des Blogs als Array von Model-Instanzen, in `@post` ist nur der enthalten, der auch angezeigt werden soll. Mit `Array#index` wird die Position von `@post` innerhalb von `@posts` ermittelt und diese um eins erhöht beziehungsweise vermindert, um benachbarte Fotos zu bestimmen. Das soll allerdings nur dann erfolgen, wenn `@post` nicht schon der letzte beziehungswise erste Beitrag des Blogs ist, sonst würde sich der Picsblog-Nutzer noch ins Datennirvana navigieren.

Legen Sie nun die View `show.rhtml` an. Auch sie muss in `app/views/posts`. Ihr Quelltext besteht hauptsächlich aus dem Anzeigen und Auslesen einzelner Akzessoren von `@post`. Außerdem werden die beiden Links zu den umgebenden Fotos mit `link_to` erstellt.

Beispiel 4-32: show.rhtml

```
<h2><%= h(@post.title) %></h2>
<%= image_tag(IMAGE_DIR + '/' + @post.image) %>
<p><%= h(@post.description) %></p>
```

Beispiel 4-32: show.rhtml (Fortsetzung)

```
<p>
  <%= link_to("< #{@prev_post.title}>", :id => @prev_post.id) unless @prev_post.nil? %>
  <%= link_to("#{@next_post.title}>", :id => @next_post.id) unless @next_post.nil? %>
</p>
```

Wie Sie sehen, werden die Links zum vorherigen oder nachfolgenden Foto nur dann angezeigt, wenn ein solches auch wirklich existiert. Als Link-Text wird der jeweilige Bildtitel herangezogen und durch einen Größer-als- oder Kleiner-als-Pfeil ergänzt..

Es ist Sitte bei Photoblogs, dass die Startseite das aktuellste Foto zeigt. Darum soll Picsblog das auch machen.

Die Startseite

Zur Realisierung einer Startseite sollten Sie eine eigene Action innerhalb eines Controllers erstellen. Nennen Sie diese Action index. Innerhalb der Methode muss hier das neueste Bild ermittelt und in @post festgehalten werden. Mit der ID des Datensatzes können Sie dann die Action show aufrufen.

Beispiel 4-33: PostsController#index

```
def index
  @post = Post.find(:first, :order => 'date DESC')
  redirect_to(:action => :show, :id => @post.id);
end
```

Probieren Sie doch mal die index-Action aus. Dazu können Sie <http://127.0.0.1:3000/posts/index> in die Adresszeile Ihres Browsers eingeben. Es reicht aber auch, wenn Sie nur <http://127.0.0.1:3000/posts> nutzen. Wenn ein URL keine Action-Angabe enthält, greift der Rails-Router standardmäßig automatisch auf index zu.

PostsController#index zur echten Startseite machen

Was halten Sie davon, dass Sie Ihre Applikation mit ihrer Startseite aufrufen, wenn Sie nur die IP-Adresse und den Port eingeben? Das geht. Für diesen Zweck können Sie das Standardverhalten des Rails-Routers beeinflussen. Öffnen Sie dazu die Datei *config/routes.rb*. Hier können Sie viele Dinge machen, die Sie vielleicht schon von einer *.htaccess* kennen, also Regeln definieren, wie ein URL aufgelöst werden soll. Je weiter oben eine Regel ist, desto wichtiger ist sie. Notieren Sie daher am besten gleich in der zweiten Zeile, aber auf jeden Fall innerhalb des Klassenkörpers folgenden Ausdruck:

```
map.connect('', :controller => 'posts')
```

Damit weisen Sie den Rails-Router an, zum Posts-Controller zu wechseln, wenn der URL keinerlei Angaben über IP-Adresse und Port oder später über die Domain hinaus enthält. Speichern Sie die Datei. Bevor Sie nun wagemutig `http://127.0.0.1:3000` in Ihrem Browser aufrufen, müssen Sie unbedingt noch die Datei `public/index.html` entfernen, die Rails bei der Generierung der Anwendung dort hin kopierte. Doch dann sollte es funktionieren.

Eine Startseite ist ganz nett, doch bislang hakt die Benutzerfreundlichkeit etwas aufgrund des Fehlens einer Navigation, die zumindest grundlegende Links für die Bedienung der Website enthält. Also, lassen Sie uns doch einfach eine anlegen.

Navigationsmenü

Die Navigation zum einfacheren Bedienen von Picsblog soll im Header der Website erscheinen und zwar auf jeder Seite. Also wäre es doch eine gute Idee, die Navigation direkt in `standard.rhtml` zu schreiben. Das ginge natürlich. Praktikabler ist es, wenn Sie das Menü extern ausgliedern. So lässt es sich leichter pflegen und stört das Grundlayout nicht.

Da die Navigation ein allgemeines Element ist und keinem bestimmten Controller zugeordnet werden sollte, legen Sie am besten in `app/views/layouts` eine neue Datei namens `navigation.rhtml` an. Ihr Inhalt könnte so aussehen:

Beispiel 4-34: `navigation.rhtml`

```
<ul id="navigation">
  <li><%= link_to("Startseite", :controller => :posts) %>
  <li><%= link_to("&Uuml;bersicht", :controller => :posts, :action => :list) %></li>
  <li><%= link_to("Neues Foto", :controller => :posts, :action => :new) %></li>
</ul>
```

Semantisch korrekt ist eine Navigation in HTML ausgezeichnet, wenn sie auf einer Liste aus Links basiert. Genau das erzielen Sie mit einem `ul`-Element, das mehrere `li`-Elemente enthält, welche wiederum aus einem Link bestehen. Um diese Navigation in das Layout `standard.rhtml` zu integrieren, ersetzen Sie das platzhalterische Wort `Header` einfach durch folgende Zeile:

```
<%= render('layouts/navigation') %>
```

Achten Sie hierbei darauf, dass Sie den Pfad relativ zu `apps/views/` angeben. Nur dann kann Rails das Template finden, auch wenn es keinem Controller zugeordnet wurde.

Damit sind die Grundfunktionalitäten von Picsblog implementiert. Sie können neue Bilder inklusive Informationen zu den Inhalten auf den Server laden, die Daten bearbeiten, eine Übersicht einsehen, Fotos löschen und in voller Schönheit betrachten. Nicht zu vergessen: Eine Startseite und eine luxuriöse Navigation ist ebenfalls enthalten. Auf den kommenden Seiten soll es nun darum gehen, der jetzigen Version ein paar Schmankerl hinzuzufügen.

- [Startseite](#)
- [Übersicht](#)
- [Neues Foto](#)

Sonnenuntergang

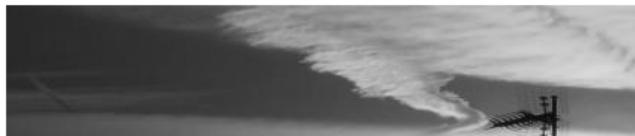


Abbildung 4-5: Picsblog. Jetzt auch mit Navigationsleiste

LoginGenerator: Zutritt nur bedingt gestattet

Sie haben sich bestimmt schon Gedanken um die grenzenlose Offenheit der Software gemacht. Zu recht. Denn zum jetzigen Zeitpunkt kann *jeder* Fotos einpflegen, löschen und Informationen ändern. Bliebe das so, würden diese Seiten sicher bald voll von Bildern sein, die sonst im Spam-Filter des heimischen E-Mail-Clients hängen bleiben.

Wie wäre es, wenn einzelne Funktionen der Software nur Benutzern zugänglich wären, die sich zu diesem Zwecke authentifizieren müssen? Gar kein Problem mit Rails.

Um ein einfaches Login-System zu implementieren, das nur zur Absicherung der für Sie wichtigen Funktionen dienen soll, können Sie auf eine vorgefertigte Lösung zurückgreifen. Sie heißt *LoginGenerator* und kann bequem und schnell via *Ruby-Gems* bezogen werden. Sie kennen bereits andere Generatoren, mit denen Sie Models und Controllers generiert haben. Gleich haben Sie auch einen Generator, mit dem Sie in Windeseile Zugangsbeschränkungen implementieren können.

LoginGenerator installieren

Öffnen Sie zunächst Ihre Kommandozeile oder Ihr Terminal und installieren Sie die Software durch die Eingabe folgender Zeile.

```
gem install login_generator
```

Das war's schon. Jetzt können Sie mit LoginGenerator in Ihrer Anwendung arbeiten. Klicken Sie dazu auf den *Generators*-Tab. Die linke Combobox sollten Sie jetzt nur als Eingabefeld nutzen, da Sie hier den LoginGenerator nicht finden werden. Er gehört eben nicht zur Grundausstattung. Geben Sie also links `login` ein, rechts `Account`.



Sie können auch einen Namen als Account für das zu generierende Login-System vergeben. Allerdings ist das mit Änderungen im generierten Code verbunden, also sollten Sie sich das gut überlegen.

In der *Console* können Sie, nachdem Sie das Generieren gestartet haben, beobachtet, welche Dateien dabei erzeugt werden.

Benutzerdatendatenbanktabelle

Wechseln Sie zurück zum *Generators-View*. Im nächsten Schritt erzeugen Sie hier eine neue Migration. Schließlich braucht das Login-System eine Datenbanktabelle, in der die Benutzerdaten gespeichert werden. Dabei soll eine Tabelle entstehen, die auf den Namen `users` hört. Wählen Sie aus der linken Liste `migration` und geben Sie in das rechte Eingabefeld `create_users` ein. Wieder tut sich was in der *Console*. Im Verzeichnis `db/migrate` müsste sich nun eine Datei namens `002_create_users.rb` befinden. Öffnen Sie diese.

Das Erzeugen einer neuen Datenbanktabelle ist Ihnen bereits von `001_create_posts.rb` bekannt. Es stellt sich also nur die Frage, welche Spalten die Tabelle `users` benötigt. Welche Benutzerdaten sind also für Sie relevant? Im einfachsten Fall reicht ein Benutzername und ein Passwort. Die Migration sähe dann so aus:

Beispiel 4-35: `002_create_users.rb`

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column(:login, :string)
      t.column(:password, :string)
    end
  end

  def self.down
    drop_table(:users)
  end
end
```

Bevor Sie nun munter drauf los migraten, nutzen Sie die Chance, schon innerhalb der Migration einen ersten Benutzer anzulegen. Denn wie bereits erwähnt, sind Migrations nicht auf das Anlegen von Datenbanktabellen allein beschränkt.

Dazu eine Quizfrage: Wenn die Datenbanktabelle `users` heißt, wie mag dann wohl das dazugehörige Model heißen? Stichwort: *Pluralbildung*. Die richtige Antwort ist natürlich `User`. Dieses Model wurde beim Benutzen von `LoginGenerator` automatisch erzeugt. Überzeugen Sie sich selbst. In `app/models` finden Sie jetzt eine Datei namens `user.rb`.



Öffnen Sie die Datei `app/models/user.rb` ruhig einmal. Im unteren Bereich finden Sie schöne Beispiele zum Thema Validatoren in Rails. Hier sehen Sie beispielsweise, dass der Login-Name zwischen 3 und 40 Zeichen lang sein darf und dass der Login-Name einzigartig sein muss.

Zurück zur Migration. Sie können das Model User schon innerhalb der Migration nutzen. So können Sie einen ersten Datensatz anlegen. Hierbei müssen Sie beachten, dass das hier verwendete Login-System beim Erzeugen eines neuen Datensatzes ein Attribut namens `password_confirmation` benötigt. Im Model User wird das Anlegen eines neuen Benutzers versagt, wenn `password` nicht mit `password_confirmation` übereinstimmt. Ergänzen Sie `002_create_users.rb`. Optimalerweise mit Ihren eigenen Daten.

Beispiel 4-36: #002_create_users.rb inklusive erstem Datensatz

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.column(:login, :string)
      t.column(:password, :string)
    end

    User.create(:login => 'denny',
               :password => 'pass2000',
               :password_confirmation => 'pass2000')
  end

  def self.down
    drop_table(:users)
  end
end
```

Nun können Sie diese Migration im Rake-Tab mit `db:migrate` einpflegen. Anschließend können Sie in die Data-Perspective wechseln und sich vom Erfolg der Aktion überzeugen. Sie sollten eine neue Datenbanktabelle vorfinden, die bereits einen Datensatz enthält. Dort sollte der in der Migration definierte Benutzernamen und ein codiertes Passwort ersichtlich sein.

Und wenn Sie sich schon einmal in der Data-Perspektive befinden: Schauen Sie doch mal in die Tabelle `schema_info`. Hier werden Sie sehen, dass der Wert, welcher die aktuelle Migrations-Version enthält, nun nachvollziehbar auf 2 steht.

Einbinden des Login-Systems in Picsblog

Damit Picsblog das Login-System nutzen kann, müssen Sie nun noch einige Einträge in app/controllers/application.rb vornehmen. Öffnen Sie diese Datei und nehmen Sie folgende Ergänzungen vor:

Beispiel 4-37: application.rb

```
# Filters added to this controller apply to all controllers in the application.
# Likewise, all the methods added will be available for all controllers.

require_dependency 'login_system'

class ApplicationController < ActionController::Base
  # Login-System für alle Controller einbinden
  include LoginSystem
  model :user

  # Pick a unique cookie name to distinguish our session data from others'
  session :session_key => '_picsblog_session_id'
end
```

Durch diese Ergänzungen in *application.rb* steht Ihnen der Login-Mechanismus nun in allen Controllern Ihrer Anwendung zur Verfügung. Zugegeben, Picsblog verfügt momentan zwar nur über einen Posts, der davon Gebrauch machen wird. Aber wenn Sie die Erweiterungslust erst einmal gepackt hat, kann sich das ganz schnell mal ändern. Gerade mit Ruby on Rails.

Öffnen Sie nun Ihren Posts-Controller. Hier können Sie bestimmen, welche Actions des Controllers überhaupt eine Zugangsbeschränkung erhalten sollen. Bei Picsblog stehen lediglich die Actions show, list und index auch nicht registrierten Benutzern zur Verfügung. Somit können Sie anweisen, dass alle Actions außer den genannten für den normalen Besucher tabu sind.

Dieses Verhalten wird in diesem Fall mit einem *Filter* implementiert. Filter stellen eine Möglichkeit in Controllern dar, Quelltext *vor* oder *nach* einer Action auszuführen. Das Überprüfen der Zugangsberichtung zu Actions eines Controllers ist eine typische Verwendungsart dieser Technik.

Beispiel 4-38: index, show und list können ohne Login genutzt werden

```
class PostsController < ApplicationController
  before_filter(:login_required,
    :except => [:index, :show, :list]
  )

  layout('standard')

  def new
    @post = Post.new
  end
  ...

```

Und das war's schon. Probieren Sie doch einmal, ein neues Foto zu erstellen. Ohne Login wird Ihnen das nicht mehr gelingen.



Sie können auch statt einer Liste von Actions, die kein Login erfordern auch eine Liste von Actions angeben, die ein Login benötigen. Wechseln Sie dazu :except gegen :only und tauschen Sie die Actions in der Liste aus.

Das hier verwendete Login-System bietet Ihnen einige Actions an, mit denen sich neue Benutzer anmelden oder Benutzer gelöscht werden können. Schauen Sie sich doch einmal *app/controllers/account_controller.rb* an. Alle dort verzeichneten Actions können Sie über *http://127.0.0.1:3000/account/<action>* aufrufen.

Wenn Sie einige der Actions nicht benötigen, dann leiten Sie doch jede Anfrage etwa an *signup* oder *delete* um, zum Beispiel zur Startseite von Picsblog. Dafür müssen Sie Änderungen in *app/controller/account_controller.rb* vornehmen, hier am Beispiel von *AccountController#signup*.

AccountController#signup? Kein Bedarf.

```
...
def signup
  redirect_to(:controller => "posts")
  case @request.method
  when :post
    ...
  ...
```

Nun gilt es noch, die Benutzeroberfläche dem neuen Login-System anzupassen. Denn es gibt schließlich einige Links, die Actions aufrufen, welche nur registrierten Benutzern zur Verfügung stehen. Besucher, die keinen Account haben, brauchen diese Links erst gar nicht zu Gesicht bekommen. Die Frage an der Stelle lautet, wie man heraus finden kann, ob ein Benutzer eingeloggt ist oder nicht.

Die Antwort: *Sessions*. In der Session-Variablen *user* steht der Name eines eingeloggten Benutzers. Die Variable ist leer, wenn ein Benutzer nicht angemeldet ist.

Eine Session kann Daten speichern, die Webseiten-Reloads oder -Wechsel überdauern. Alle Session-Variablen einer Website können Sie über die stets präsente Variable *@session* abrufen. Hierbei handelt es sich um ein Hash, wodurch der Name einer Session-Variablen zum Schlüssel wird.

Ist *@session['user']* nicht *nil*, dann sollen die Links zu *Bearbeiten*, *Neues Foto* und *Löschen* sichtbar sein. Außerdem können Sie die Navigation um zwei Punkte, nämlich *Login* und *Logout*, erweitern, wobei stehts nur einer der beiden Links sichtbar sein soll.

Beispiel 4-39: navigation.rhtml

```
<ul>
  <li><%= link_to("Startseite", :controller => :posts) %>
  <li><%= link_to("&Uuml;bersicht", :controller => :posts, :action => :list) %></li>
<% if @session['user'].nil? %>
  <li><%= link_to("Login", :controller => :account, :action => :login) %></li>
<% else %>
  <li><%= link_to("Neues Foto", :controller => :posts, :action => :new) %></li>
  <li><%= link_to("Logout", :controller => :account, :action => :logout) %></li>
<% end %>
</ul>
```

Auch im Partial *_thumbnail.rhtml* gibt es was auszublenden, wenn ein nicht angemeldeter Benutzer des Weges kommt.

Beispiel 4-40: _thumbnail.rhtml

```
<div>
  <%= link_to(image_tag(THUMBNAIL_DIR + '/' + thumbnail.thumbnail, :title => thumbnail.title), {:action => 'show', :id => thumbnail.id}) %>
  <p>
    <%= h(thumbnail.title) %><br />
    <%= h(thumbnail.description) %><br />
    <% if !@session['user'].nil? %>
      <%= link_to('Bearbeiten', {:action => :edit, :id => thumbnail.id}) %>
      <%= link_to('L&ouml;schen', {:action => :delete, :id => thumbnail.id}, :confirm => 'Möchten Sie das Bild wirklich löschen?') %>
    <% end %>
  </p>
</div>
```

Damit sind alle wichtigen Funktionen geschützt und Links zu ihnen nur dann sichtbar, wenn es sich lohnt, auf sie zu klicken. Damit ist es uns auch gelungen, eine Anwendung zu schreiben, die ohne Admin-Bereich auskommt, sondern praktisch am Ort des Geschehens gepflegt werden kann.

Kommentare erwünscht

Wenn Sie Ihren Besuchern schon verbieten, Bilder zu löschen oder Bilttitel zu ändern, dann gestatten Sie Ihnen doch wenigstens, ein paar nette, kommentierende Worte zu hinterlassen. Zu diesem Zweck soll nun eine Kommentarfunktion entstehen, die es ermöglicht, zu jedem Foto eine persönliche Anmerkung zu hinterlassen.

Zu diesem Zweck ist es erforderlich, ein neues Model zu erstellen, das sich nur um Kommentare kümmert. Generieren Sie ein Model namens `Comment` und öffnen Sie anschließend Migration `003_create_comments.rb`, die automatisch angelegt wurde, um die Datenbanktabelle zu spezifizieren.

Beispiel 4-41: 003_create_comments.rb

```
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
      t.column(:name, :string)
      t.column(:text, :text)
      t.column(:email, :string)
      t.column(:url, :string)
      t.column(:created_at, :datetime)
      t.column(:post_id, :integer)
    end
  end

  def self.down
    drop_table :comments
  end
end
```

Zur Definition der Tabelle `comments` gibt es einige interessante Dinge zu sagen. Neben den für die Speicherung eines Kommentars obligatorischen Felder `name`, `text`, `email` und `url`, wobei hier die Adresse einer eventuell vorhandenen Website des Kommentators Unterschlupf finden kann, werden auch die Spalten `created_at` und `post_id` angelegt. In den beiden letztgenannten schlummert etwas Magie.

Besitzt eine Datenbanktabelle, die mit Ruby on Rails benutzt wird, ein Feld namens `created_at`, so brauchen Sie sich nicht um das Befüllen dieses Feldes mit einem Datums- und Zeitwert kümmern. Das macht das Framework für Sie. Wenn ein Datensatz erzeugt wird, wird in `created_at` stets der Zeitpunkt niedergeschrieben, zudem das geschah.



Etwas Ähnliches wie mit `created_at` passiert mit einem Feld namens `updated_at`. Hier hält Ruby on Rails Datum und Uhrzeit der letzten Änderung fest.

Noch ein ordentliches Stück Faszination mehr bietet das Feld `post_id`. Hier wird die ID des Datensatzes der Tabelle `posts` festgehalten, zudem der Kommentar gehört. Die Spalte `post_id` ist also der Schlüssel zur Verknüpfung der Kommentar- und Beitragsdatenbanktabelle.

Assoziationen

Im vorliegenden Fall gehört *ein* bestimmter Kommentar zu exakt *einem* bestimmten Beitrag. Auf der anderen Seite kann ein Beitrag mehrere Kommentare aufweisen. Mit einer solchen Überlegung, die die Verbindung von beiden Seiten betrachtet, können Sie eruieren, in welcher Relation Kommentare und Beiträge zueinander stehen sollen.

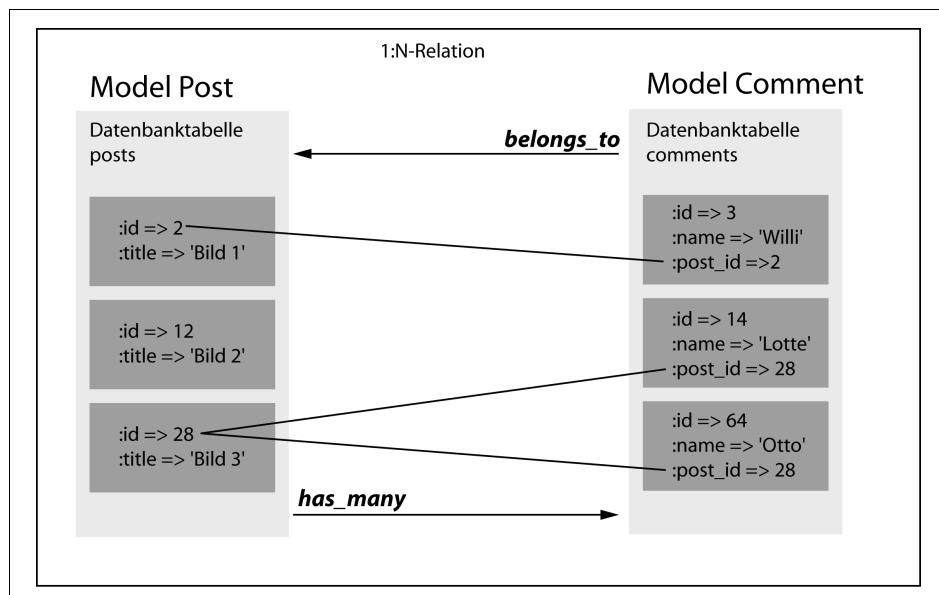


Abbildung 4-6: Assoziationen in Picsblog

Welche Konsequenzen sollten Sie nun aus diesen Gedanken ziehen? Vielleicht fällt Ihnen zunächst auf, dass es sich hier um eine 1:N-Relation ist. Früher hätten Sie sich möglicherweise nun ein dickes Buch zum Thema SQL gegriffen, mit dessen Hilfe Sie zeilenweise SQL-Statements gebastelt hätten, die zwei Datenbanktabellen mittels JOIN irgendwie miteinander verbinden. Meist hat das eine Weile gedauert, war dennoch selten von sofortigem Erfolg gekrönt und war nicht sonderlich komfortabel.

Bei der Entwicklung mit Ruby on Rails können Sie die Problematik in wenigen Zeilen erledigen und dabei die eben erwähnten Grundgedanken fast wortwörtlich aufschreiben. In Englisch natürlich. Rails nimmt Ihnen damit wirklich eine ganze Menge lästiger Arbeit ab, so dass Sie sich nicht wegen falscher SQL-Statements ärgern müssen, sondern flott bei der Entwicklung Ihrer Anwendung vorankommen. Assoziationen gehören zu den vielen Gründen, weshalb Ruby on Rails als äußerst produktiv angesehen wird.

Active Record bringt grundsätzlich vier Assoziationen mit: `belongs_to`, `has_one`, `has_many` und `has_many_and_belongs_to_many`. Zwei von ihnen möchte ich Ihnen nachfolgend vorstellen. Natürlich handelt es sich um die zwei, die Ihnen bei der Implementierung der Kommentarfunktion helfen und die die oben bereits festgestellte Relation realisieren können.

Ein Kommentar gehört also zu einem Beitrag? Dann schreiben Sie das doch einfach in Ihren Quelltext! Öffnen Sie dazu `app/models/comment.rb`.

```
class Comment < ActiveRecord::Base
  belongs_to(:post)
end
```

Und nun noch zur anderen Seite. Dem Post-Model müssen Sie mitteilen, dass ein Beitrag viele Kommentare besitzt. Fix ins Englische übersetzt, *app/models/post.rb* geöffnet und niedergeschrieben:

```
class Post < ActiveRecord::Base
  has_many(:comments)
  validates_presence_of(:title, :description, :date, :image, :thumbnail)
  ...

```

Beide Assoziationen brauchen als Parameter den Namen des jeweils anderen Models. Da die Wortkombination *has many* rein grammatisch betrachtet verlangt, dass ihr ein Wort in der Mehrzahl folgt, tun Sie ihr auch den Gefallen und schreiben Sie `:comments` statt `:comment`.

Damit ist die Verknüpfung beider Tabellen schon erledigt. Und mehr noch ist passiert: Die beiden Models Post und Comment wurden mit einer Reihe an Methoden ausgestattet, die das Benutzen dieser Verknüpfung sehr angenehm gestaltet. Beispielsweise können Sie einen bestimmten Beitrag aus der Datenbanktabelle holen und bekommen auf Wunsch alle dazugehörigen Kommentare gleich mitgeliefert, ohne, dass Sie das Comment-Model explizit ansprechen müssen. Sie werden das gleich noch live erleben.

Ganz wichtig: Diese `belongs_to` / `has_many` Verknüpfung funktioniert nur deshalb, weil die Tabelle `comments` über eine Spalte namens `post_id` verfügt. Eine solche Spalte dient zum Speichern des Fremdschlüssels. In diesem Falle ist es die ID des mit dem Kommentar verknüpften Beitrags. Ohne eine solche Spalte funktioniert die Kombination `belongs_to` und `has_many` nicht. Achten Sie darauf, dass der Fremdschlüssel mit der Bezeichnung des verbundenen Models beginnt und mit `_id` endet.



Wenn Sie später einmal unsicher sein sollten, in welcher der beiden beteiligten Datenbanktabellen die Spalte für den Fremdschlüssel enthalten sein muss, merken Sie sich einfach folgenden Grundsatz:
»Wenn ein Model `belongs_to` enthält, braucht seine Tabelle das Fremdschlüsselfeld.« (Sollten Sie dieses Buch im Lyrik-Regal Ihrer Buchhandlung gefunden haben, wissen Sie jetzt, warum.)

Lassen Sie uns nun die reiche Ernte dieser kleinen Vorarbeit einfahren. In Picsblog soll es möglich sein, jedes Bild zu kommentieren. Daher ist die `show`-Action des Posts-Controllers der geeignete Ort, um Kommentare und ein Formular zum Abgeben eines Kommentars zum jeweiligen Bild zu platzieren.



Sie können an dieser Stelle auf einen separaten Controller für das Model Comment verzichten. Das passt, nicht zuletzt durch die Verknüpfung der beiden Tabellen, bestens in den Controller Posts hinein.

Rufen Sie `app/controllers/posts_controller.rb` auf und begeben Sie sich zur Action `show`. Ergänzen Sie selbige um zwei Zeilen.

Beispiel 4-42: show.rhtml wird auf die Kommentarfunktion vorbereitet

```
def show
  @post = Post.find(params[:id])

  # Vorheriges / nächstes Foto
  @posts = Post.find(:all, :order => 'date DESC')
  @next_post = @posts[@posts.index(@post) + 1] unless @post == @posts.last
  @prev_post = @posts[@posts.index(@post) - 1] unless @post == @posts.first

  # Neuer Kommentar
  @comment = Comment.new
end
```

Das kennen Sie schon von der Action `new` des Controllers: Hier wird einfach eine datenleere Instanz des `Comment`-Models erzeugt. Es könnte ja schließlich bei jedem Aufruf von `posts/show` sein, dass ein Nutzer seinen Senf zum angezeigten Bild abgeben möchte.

Das Objekt `@comment` können Sie nun in den Feldern eines Formulars benutzen, mit dem ein Kommentar abgegeben werden kann. Öffnen Sie dazu `app/views/posts/show.rhtml` und fügen Sie folgende Zeilen einfach am Dateiende an:

Beispiel 4-43: show.rhtml wird ergänzt

```
<%= form_tag(:action => :create_comment, :id => @post.id) %>
<h2>Neuer Kommentar</h2>
<p>Name:<br />
<%= text_field(:comment, :name) %></p>
<p>E-Mail:<br />
<%= text_field(:comment, :email) %></p>
<p>Website (falls vorhanden):<br />
<%= text_field(:comment, :url) %></p>
<p>Kommentar:<br />
<%= text_area(:comment, :text, :rows => 5) %></p>
<p><%= submit_tag('Kommentieren') %></p>
</form>
```

Hier wird ein Formular erzeugt, das seine Eingaben an die Action `create_comment` des `Posts`-Controllers schickt. Die ID des Beitrags wird dabei ebenfalls übermittelt. Nur mit dieser ID kann `create_comment` den angelieferten Kommentar dem passenden Beitrag zuordnen. Das ist genau wie im Formular von `edit.rhtml`.

Innerhalb des Formulars kommt ein neuer Form-Helper zum Einsatz: Mit `text_area` können Sie ein wesentlich größeres, weil mehrzeiliges Eingabefeld für Text erzeugen. Die Anzahl der Zeilen können Sie mit dem Schlüssel `rows` übergeben. In HTML wird daraus ein `textarea`-Element.

Sonnenuntergang



Ende eines Sommertages über den Dächern Berlins

[Herbst >](#)

Neuer Kommentar

Name:

E-Mail:

Website (falls vorhanden):

Kommentar:

Abbildung 4-7: Das Kommentar-Fomular ergänzt show.rhtml

Nun können Sie die formularverarbeitende Action `create_comment` schreiben.

Kommentare speichern

Sie erhält die Formulareingaben via `params[:comment]` und die ID des zum Kommentar gehörenden Beitrags, wie gewohnt, mit `params[:id]`.

Beispiel 4-44: PostsController#create_comment

```
def create_comment
  @comment = Comment.new(params[:comment])
  @post = Post.find(params[:id])
  if @comment.valid?
    @post.comments << @comment
    flash[:notice] = 'Vielen Dank f&uuml;r Ihren Kommentar.'
    redirect_to(:action => :show, :id => @post.id)
  else
    Flash.now[:notice] = 'Bitte f&uuml;llen Sie alle ben&ouml;tigten Felder aus.'
    render(:action => :show, :id => @post.id)
  end
end
```

Zunächst wird auf den Formulareingaben basierend eine neue Instanz des `Comment`-Models erzeugt und in `@comment` abgelegt. In `@post` steckt eine Instanz des `Post`-Models, welche den gerade angezeigten Datensatz beinhaltet.

Im nächsten Schritt wird geprüft, ob denn die Angaben, die im Kommentarformular gemacht wurden, auch ausreichen. Die Methode `valid?` kennen Sie bereits. Die Definition, was `valid` ist und was nicht, können Sie gleich dem Model `Comment` hinzufügen.

Ist der abgegebene Kommentar einwandfrei, so wird er gespeichert. Und hier treffen Sie auf den Zauber von Rails' Assoziationen. Denn in `@post.comments` stecken alle Kommentare, die zu diesem Beitrag gehören. Bei der Abgabe des ersten Kommentars ist dieses Array natürlich leer. Später werden sich hier eine Reihe von `Comment`-Instanzen befinden.

Um diese Kommentarsammlung um eine neue Meinungsäußerung zu erweitern, benutzen Sie einfach die Methode `<<`, welche Sie ebenfalls aufgrund der Verwendung von Assoziationen erhalten haben und die Sie in gleicher Form schon aus der Array-Klasse kennen. Wichtig hierbei: `<<` nimmt nicht einfach nur einen neuen Kommentar entgegen, sondern speichert ihn sofort in der Datenbank ab. Durch die Verwendung der Assoziationen wird der Kommentar natürlich in der Tabelle `comments` gespeichert und automatisch mit dem richtigen Fremdschlüssel versehen. Und das passiert alles im Hintergrund und belästigt Sie nicht weiter.

Sollten die Angaben im Kommentar nicht den Ansprüchen des Picsblog-Admins genügen, dann bekommt der Kommentator eine neue Chance, basierend auf seinen bisher gemachten Eingaben, die noch in `@comment` enthalten sind.

Damit `@comments.valid?` auch sinnvoll verwendet werden kann, sollten Sie noch Validierungsrichtlinien definieren. In `app/models/comment.rb` ist genügend Platz.

Beispiel 4-45: comment.rb

```
class Comment < ActiveRecord::Base
  belongs_to(:post)
  validates_presence_of(:name, :email, :text)
end
```

Da die Angabe der Website freiwillig ist, wird sie natürlich nicht als Validierungskriterium aufgeführt. Aber Name, E-Mail-Adresse und natürlich der Kommentar-Text müssen schon sein.

Bevor Sie nun Ihren ersten Probekommentar zu einem Foto abgeben, sollten Sie `create_comment` noch in die Ausnahmeliste des Login-Systems aufnehmen. Es sei denn, Sie möchten, dass nur registrierte Benutzer Kommentare schreiben dürfen.

```
class PostsController < ApplicationController
  before_filter(:login_required,
    :except => [:index, :show, :list, :create_comment])
  ...
  ...
```

Damit steht dem Kommentieren nichts mehr im Weg. Besucher von Picsblog können nun ihr künstlerisches Verständnis unter Beweis stellen und mit tiefscrüfenden Analysen glänzen.



Sicher wissen Sie, dass neben E-Mails auch Webformulare von zwielichtigen Personen mit Hilfe entsprechender Software heftig mit Spam versorgt werden können. Um den Inhalt zu gebieten, haben sich *Captchas* etabliert. Dabei handelt es sich um kleine Grafiken, die Buchstaben und Zahlen enthalten und praktisch nur vom menschlichen, nicht aber vom softwaregestützten Auge eindeutig entziffern können. So zeugt die korrekte Eingabe dieser Zeichen davon, dass es sich beim Kommentierenden um einen Menschen mit Hirn und Verstand handelt.

Um Captchas in Ihren Anwendungen zu nutzen, können Sie beispielsweise auf *Turing* zurückgreifen, einer Rails-freundlichen Ruby-Implementation. Sie ist via RubyGems verfügbar: `gem install turing`

Damit die gesamte Besucherschaft von Picsblog auch weiß, was andere Fotoliebhaber von einem bestimmten Bild halten, müssen die Kommentare noch angezeigt werden.

Kommentare anzeigen

Dabei sollen mehrere Kommentare untereinander angezeigt werden. Es ist also mal wieder Zeit für ein Partial. Nennen Sie es `_comment.rhtml` und speichern Sie es in `app/views/posts` ab.

Beispiel 4-46: `_comment.rhtml`

```
<p><strong>
<% if comment.url.nil? || comment.url.empty? %>
  <%= h(comment.name) %>
<% else %>
  <%= link_to(h(comment.name), h(comment.url)) %>
<% end %>
</strong> schrieb am <%= comment.created_at %>: </p>
<p><%= comment.text %></p>
<%= link_to('Löschen', :action => 'delete_comment', :id => comment.
id) unless @session['user'].nil? %>
<hr />
```

Sie erinnern sich? Das Partial ist unter `_comment.rhtml` abgespeichert, also steht ein einzelner Kommentar in der Variable `comment` innerhalb des Partials zur Verfügung. Sicher ist Ihnen auch der Löschen-Link aufgefallen, der nur dann eingeblendet wird, wenn ein Löschberechtigter eingeloggt ist. Es soll ja doch mal vorkommen, dass einige Kommentare nicht ganz öffentlichkeitsstauglich sind. Und die kann man so leicht löschen – natürlich erst, wenn die Action `delete_comment` geschrieben ist. Zum Abschluss des Kommentars wird mit `<hr />` noch eine Trennlinie angezeigt.

Damit `_comment.rhtml` auch genutzt wird, müssen Sie die show-View noch geringfügig erweitern. Die show-Action kann so bleiben, denn die Kommentare stehen Ihnen bereits in `@post.comments` zur Verfügung.

`show.rhtml` zeigt bisherige Kommentare an

```
...
<p>
  <%= link_to("< #{@prev_post.title}>", :id => @prev_post.id) unless @prev_post.nil? %>
  <%= link_to(" #{@next_post.title} >", :id => @next_post.id) unless @next_post.nil? %>
</p>

<h2>Kommentare</h2>
<%= '<p>Bisher wurde noch kein Kommentar abgegeben.</p>' if @post.comments.
count == 0 %>
<%= render(:partial => 'comment', :collection => @post.comments) %>

<%= form_tag(:action => :create_comment, :id => @post.id) %>
  <h2>Neuer Kommentar</h2>
  ...
  ...
```

Sollte noch kein Kommentar zum jeweiligen Foto vorliegen, wird eine entsprechende Botschaft angezeigt. Das Partial `comment` wird mit dem Array in `@post.comments` gefüttert. Diese Zuweisung erfolgt über den Schlüssel `collection` der `render`-Methode.

Es fehlt noch die Action `delete_comment`. Wie Sie dem Partial entnehmen können, erhält die Action Kenntnis von der ID des Kommentars, so dass folgende Ergänzung für den Posts-Controller entsteht.

Beispiel 4-47: PostsController#delete_comment

```
def delete_comment
  @comment = Comment.destroy(params[:id])
  flash[:notice] = 'Kommentar wurde entfernt.'
  redirect_to(:action => :show, :id => @comment.post_id)
end
```

Wenn Ihnen bei näherer Betrachtung die Darstellung des Datums und der Uhrzeit missfällt, und davon kann man aufgrund des unansehnlichen Standardformats wohl ausgehen, dann schreiben Sie sich doch mal eben einen kleinen Helfer, der das ändert.

Ein eigener Helper

Sie haben schon einige *Helper* benutzt: `form_tag`, `text_field`, `link_to` – um nur einige zu nennen. Mit Helpern bezeichnet man grundsätzlich kleine Hilfsmethoden, die in der View-Ebene zum Einsatz kommen. Ihre Rails-Anwendung ist darauf vorbereitet, dass Sie auch selbst mal Helper schreiben möchten.

Öffnen Sie dazu `app/helpers/application_helper.rb`. Alle Methoden, die Sie hier innerhalb des Modules `ApplicationHelper` definieren, stehen Ihnen in allen Views Ihrer Anwendung zur Verfügung, unabhängig von deren Controller-Zugehörigkeit. So auch `german_datetime`. Sie erhält den unformatierten Wert aus der Datenbank im Parameter `date_time` und gibt einen ansehnlichen zurück.

Beispiel 4-48: Helper german_datetime

```
module ApplicationHelper
  def german_datetime(date_time)
    date_time.strftime("%d.%m.%Y, %H:%M Uhr")
  end
end
```

Mit der Methode `strftime` können Sie eine im Empfänger enthaltene Zeitangabe nach eigenen Wünschen formatieren und erhalten ein String-Objekt. Jeder mit einem Prozentzeichen beginnende Buchstabe in der Formatvorgabe von `strftime` symbolisiert einen Teil des Empfängers und seine Position und Formatierung im Rückgabewert.

- %d – Tag des Monats mit führender Null
- %m – Monatsnummer mit führender Null
- %Y – vierstellige Jahreszahl
- %H – Stunde mit führender Null
- %M – Minute mit führender Null

Neben den Format-Platzhaltern können Sie auch Punkte, Doppelpunkte oder Worte innerhalb des Strings notieren, die im Ergebnis enthalten sein sollen.

Den Helper können Sie nun ganz einfach im Partial `_comment.rhtml` nutzen, ohne irgendwas mit `include` oder `require` einbinden zu müssen.

Beispiel 4-49: german_datetime in _comment.rhtml

```
<p><strong>
<% if comment.url.nil? || comment.url.empty? %>
  <%= h(comment.name) %>
<% else %>
  <%= link_to(h(comment.name), h(comment.url)) %>
<% end %>
</strong> schrieb am <%= german_datetime(comment.created_at) %>: </p>
<p><%= comment.text %></p>
<%= link_to('Löschen', :action => 'delete_comment', :id => comment.
id) unless @session['user'].nil? %>
<hr />
```

Die Kommentarfunktionalität von Picsblog ist damit fast fertig. Allerdings ist es ratsam, der `has_many`-Assoziation des Post-Models noch zwei Einstellungen zu verpassen.

has_many verfeinern

Bislang ist die Reihenfolge der Kommentare von der physischen Reihenfolge der entsprechenden Datensätze in der Datenbanktabelle abhängig. Damit die Elemente in `@post.comments` chronologisch geordnet ausgegeben werden, ergänzen Sie `has_many` in `post.rb`:

```
has_many(:comments, :order => 'created_at DESC')
```

Und noch etwas können Sie an dieser Stelle bestimmen. Bislang ist es so, dass Kommentare in der Datenbanktabelle verbleiben, obwohl der dazugehörige Beitrag gelöscht wurde und längst nicht mehr existiert. Das können Sie durch den Schlüssel `:dependent` ändern.

```
has_many(:comments, :order => 'created_at DESC', :dependent => :destroy)
```

Damit bewirken Sie, dass beim Löschen (`Post#destroy`) eines Datensatzes in `posts` auch alle damit in Verbindung stehenden Kommentare gelöscht werden.

Mit diesen Feineinstellungen sollen unsere Arbeiten an der Kommentarfunktion von Picsblog beendet sein. Schließlich wartet noch eine Funktionalität der eingangs aufgeführten Liste auf Umsetzung.

In der Sidebar, die bislang nur durch einen gleichlautenden Schriftzug auffällt, sollen die fünf neuesten Bilder des Blogs aufgeführt werden, jeweils mit Titel und Thumbnail. Die Herausforderung hier: Die Thumbnails-Liste soll in jeder Seite sichtbar sein und dennoch dynamisch generiert werden. Das würde bedeuten, dass neben der eigentlichen Action wie list, show oder new stets eine zweite ausgeführt und angezeigt werden müsste. Geht nicht? Geht sehr wohl. Und wer wollte ernsthaft daran zweifeln?

Ganz frisch: die neuesten Fotos

Für unsere Liste der neuesten Fotos legen Sie zunächst einmal eine funkelnagelneue Action in PostsController an. Nennen Sie sie list_newest. Mit Hilfe von find soll sie die fünf neuesten Beiträge heraussuchen und in @newest_posts ablegen.

Beispiel 4-50: PostsController#list_newest

```
def list_newest
  @newest_posts = Post.find(:all, :limit => 5, :order => 'date DESC')
end
```

Mit dem Schlüssel limit übergeben Sie die Anzahl der Post-Instanzen, die Sie wünschen. Diese Zahl können Sie übrigens als eine *maximale* Angabe sehen. Sollten gar nicht fünf Fotos in der Datenbank existieren, ist das gar kein Problem. Das Prädikat newest erhalten die fünf Beiträge, weil eine Sortierung gewählt wurde, die die neuesten Beiträge an den Anfang stellt.

Nun können Sie eine View für die Action list_newest erstellen. Innerhalb der View könnten Sie ein weiteres Mal auf ein Partial zurückgreifen. Aber da diesmal wirklich nur ein verlinktes Thumbnail angezeigt werden soll, erscheint ein Partial vielleicht doch etwas fehl am Platze. Wie wär's als Alternative mit dem each-Iterator? Lassen Sie ihn in app/views/posts/list_newest.rhtml über @newest_posts iterieren.

Beispiel 4-51: list_newest.rhtml

```
<h3>Neue Fotos</h3>
<p>
<% @newest_posts.each do |post| %>
  <%= link_to(image_tag(THUMBNAIL_DIR + '/' + post.thumbnail, :title => post.title),
  {:action => 'show', :id => post.id}) %><br />
<% end %>
</p>
```

Bevor Sie `list_newest` erstmalig in Betrieb nehmen, sollten Sie noch daran denken, diese Action in die Ausnahmeliste des Login-Systems aufzunehmen. Sie soll schließlich unabhängig vom Login-Status eines Besuchers funktionieren.

```
class PostsController < ApplicationController
  before_filter(:login_required,
    :except => [:index, :show, :list, :create_comment, :list_newest])
)
...
...
```

Schauen Sie sich nun das Ergebnis von `PostsController#list_newest` an, indem Sie `http://127.0.0.1:3000/posts/list_newest` aufrufen. Es gibt in Rails eine Möglichkeit, genau das, was Sie nun sehen, in das Layout einzubauen. Wie das geht, erfahren Sie gleich. Vorab gibt es ein Problem zu beseitigen.

Die Action `list_newest` wird, selbstverständlich, in das Layout `standard.rhtml` eingebettet. Das ist bei normalen Actions auch völlig in Ordnung. Bei dieser aber nicht. Ihre gerenderte View soll schließlich in der Sidebar erscheinen. Beim jetzigen Stand würde das bedeuten, dass dort das ganze Layout auch erscheinen würde. Das Layout würde das Layout würde das Layout würde das Layout einbinden. Das muss verhindert werden. Und zwar in `posts_controller.rb`:

```
layout('standard', :except => :list_newest)
```

So sollte nun die Layoutzuweisung in Ihrem Post-Controller aussehen. Hier wird dem Schlüssel `except` der Name einer Action übergeben, für die das vorher angegebene Layout nicht gelten soll.

So weit, so gut. Jetzt muss diese Action nur noch permanent im Layout angezeigt werden. Das geht, wenn Sie Rails anweisen, in der Sidebar eine *Component* zu rendern.

Components einsetzen

In `standard.rhtml` hockt noch immer der Platzhalter *Sidebar*. Entfernen Sie ihn und setzen Sie an seiner Stelle folgende Anweisung hinein:

```
<%= render_component(:controller => 'posts', :action => 'list_newest') %>
```

Durch diese Zeile führt Ruby on Rails die angegebene Action im angegebenen Controller aus und setzt das gerenderte Ergebnis an genau dieser Stelle ein.



Bitte beachten Sie, dass `render_component` hier mit Werten, die als Symbol übergeben werden, nicht klar kommt. Nutzen Sie stattdessen String-Werte.

Vielleicht erinnern Sie sich noch an die Integration der Navigation? Dabei wurde einfach nur ein RHTML-Template in das Layout gepflanzt, ohne, dass vorher eine spezielle Action ausgeführt wurde.

Mit Components ist es im Gegensatz dazu möglich, eine Action auszuführen und das gerenderte Ergebnis zu verwenden. So gelingt es Ihnen auch, nicht nur eine Stelle (Stichwort: `yield`) im Layout mit dynamisch erzeugten Inhalten zu füllen, sondern mehrere. Ein modularer Aufbau einer Website ist somit sehr einfach möglich.

Unser Picsblog soll schöner werden

Zum Abschluss dieses Kapitels und zum Abschluss der Entwicklung Ihrer ersten Rails-Applikation sollten Sie der Oberfläche von Picsblog noch eine Schönheitskur verpassen. Denn bislang sehen Sie nur reinstes, feinstes HTML. Und schließlich sieht ein Foto in einem schönen Rahmen noch viel besser aus.

Damit schließt sich auch der Kreis in diesem Kapitel. Sie haben angefangen mit der Fertigung eines Grundlayouts mit Hilfe von CSS. Jetzt, da der erste Prototyp von Picsblog fertig ist, lohnt es sich, an der Optik zu feilen.

Dazu legen Sie einfach eine neue CSS-Datei in `public/stylesheets` an und nennen sie `styles.css`. Im Gegensatz zu `layout.css` kommen hier nur Formatierungen hinein, die der Optik dienen.

Anschließend sollten Sie das Layout, `standard.rhtml`, davon in Kenntnis setzen, dass es neue Darstellungsvorschriften gibt.

Beispiel 4-52: Einbinden von styles.css in standard.rhtml

```
...
<html>
  <head>
    <title>Picsblog</title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
    <%= stylesheet_link_tag('layout') %>
    <%= stylesheet_link_tag('styles') %>
  </head>
  <body>
  ...

```

Schon ist `styles.css` eingebunden. Allein, die Datei ist noch leer. Beginnen wir mit grundlegenden Dingen, die die Oberfläche betreffen. Mein persönlicher Geschmack wünscht sich ein eher dunkles Erscheinungsbild, damit das Foto richtig zur Geltung kommen kann. Als Schriftfarbe soll eine helle, aber ebenfalls dezente Farbe zum Einsatz kommen. Als grundlegende Schriftart für die komplette Seite ist *Georgia* mein Favorit. Vielleicht ja auch Ihrer?

```
body {  
    background: #11110D;  
    color: #E4EE85;  
    font-family: Georgia, Times, serif;  
}
```

Die Links der Website sollen zunächst in Weiß erscheinen. Sobald sich ein Mauszeiger über ihnen befindet, sollen sie zwecks Benutzer-Feedback zur Standardtextfarbe wechseln. Außerdem weisen Thumbnails, die verlinkt sind, einen Rahmen auf. Der muss auch noch eliminiert werden.

```
a {  
    color: #FFF;  
}  
  
a:hover {  
    color: #E4EE85;  
}  
  
img {  
    border: 0px;  
}
```

Die Navigation, deren ul-Element durch den ID-Selektor #navigation angesprochen werden kann, soll zunächst ihre Bullets und die standardmäßigen Abstände und Einrückungen verlieren. Außerdem soll ein horizontaler Strich am unteren Rand der Navigation für Abgrenzung sorgen.

```
#navigation {  
    list-style: none;  
    margin: 0;  
    padding: 0;  
    height: 25px;  
    border-bottom: 1px solid #663D0D;  
}
```

Die einzelnen Navigationselemente sollen nicht vertikal, sondern horizontal angeordnet werden. Mit float:left, angewendet auf die li-Elemente der Navigation, ist das schnell erledigt. Die Menüpunkte sollen in einem dezenten Braun erscheinen. Zwischen ihnen soll ein 5 Pixel großer horizontaler Abstand für Struktur sorgen. Die Schriftart wechselt zu *Verdana* und wird etwas verkleinert. Der Grund: Der Text der Navigations-Tabs soll ausschließlich mit Großbuchstaben angezeigt werden.

```
#navigation li {  
    float:left;  
    background: #663D0D;  
    padding: 3px 8px;  
    margin-right: 5px;  
    font-family: Verdana, Arial, sans-serif;
```

```
font-size: 80%;  
font-weight: bold;  
text-transform: uppercase;  
}
```

Da die Navigationselemente aus Links bestehen, können Sie auch die noch gesondert behandeln. In den folgenden Zeilen wird ihnen die obligatorische Unterstreichung geklaut.

```
#navigation li a {  
    text-decoration: none;  
}
```

Das Foto im Großformat soll noch einen weißen Rahmen erhalten. 6 Pixel dick. Damit Sie das erwähnte img-Element gezielt ansprechen können, verpassen Sie ihm noch eine eindeutige ID. Wie wär's mit `big_image`? Ergänzen Sie die Angabe in `app/views/show.rhtml`.

```
<h2><%= h(@post.title) %></h2>  
<%= image_tag(IMAGE_DIR + '/' + @post.image, :id => 'big_image') %>  
<p><%= h(@post.description) %></p>  
<p>
```

Nun können Sie die Formatierung für `#big_image` in `styles.css` vornehmen.

```
#big_image {  
    border: 6px solid #FFF;  
}
```

Abschließend soll der Bereich, der eventuelle Hinweismeldungen aus `flash[:notice]` anzeigt, eine gesonderte Formatierung erhalten. Damit's auch auffällt, wenn dort eine Botschaft erscheint.

```
#notice {  
    color: #11110D;  
    background: #E4EE85;  
    padding: 0px 8px;  
}
```

Damit wäre die grundlegenden kosmetischen Maßnahmen für Picsblog abgeschlossen. Natürlich können Sie noch weitermachen oder ihr ganz eigenes Design entwerfen.

Ach ja, der Footer

Abgesehen davon gibt es noch einen Platzhalter zu besetzen, den Footer. Was Sie dort hinein platzieren, bleibt Ihnen überlassen. Aber vielleicht ist ja ein Hinweis auf das Herz von Picsblog ein interessanter. Mein Footer sieht so aus:

Beispiel 4-53: Footer in standard.rhtml

```
...
<div id="footer">
  Picsblog basiert auf <%= link_to('Ruby on Rails', 'http://www.rubyonrails.org') %>.
</div>
</div>
</body>
</html>
```

Ich finde, es soll ruhig jeder wissen, dass Sie so clever sind und mit Ruby on Rails entwickeln. Erst recht, wenn Sie `link_to` direkt in `standard.rhtml` benutzt haben – denn natürlich geht das auch. Wenn Sie ERb-Code ins Layout setzen möchten, für den es sich einfach nicht lohnt ein separates Template anzulegen, dann tun Sie das einfach.

Damit der textbasierte Footer auch ansprechend zur Geltung kommt, soll der Footer auch in `styles.css` bedacht werden.

```
#footer {
  text-align: center;
  padding: 10px 0;
  font-style: italic;
  border-top: 1px solid #663D0D;
}
```

Nun soll es aber wirklich genug sein. Womit ich Sie aber nicht bremsen möchte, Picsblog noch um einige Funktionen zu erweitern. Trauen Sie sich ruhig. Immerhin haben Sie auf den vergangenen Seite eine ganze Menge über das Programmieren mit Ruby on Rails erfahren.

Auf der Website zum Buch, <http://www.praxiswissen-ruby-on-rails.de> finden Sie alle Dateien dieses Projekts. Außerdem können Sie sich dort eine erweiterte Picsblog-Version herunterladen, die eine Menge zusätzlicher Funktionen enthält, wie zum Beispiel RSS-Feed und E-Mail-Benachrichtigung bei neuen Kommentaren.

Zusammenfassung

Viele Begriffe der Rails-Programmierung, denen Sie beispielsweise im 3. Kapitel begegnet sind, sollten Ihnen nun schon recht vertraut erscheinen. Zum Beispiel wissen Sie, was Migrations sind, wie Sie mit ihnen arbeiten können und welche Vorteile Sie als Entwickler von Migrations haben. Partials, Layouts, Components – das alles sind keine Unbekannten mehr für Sie. Assoziationen, Validierungsmethoden, Filter, Callbacks, Generatoren, Form-Helper, Tag-Helper, flash, Pagination, Routing und vieles mehr haben Sie in diesem Kapitel kennengelernt. Nicht schlecht, oder?

Sonnenuntergang



Neue Fotos



Ende eines Sommertages über den Dächern Berlins

[Herbst >](#)

Kommentare

Test schrieb am 18.02.2007, 08:56 Uhr:

Test

Neuer Kommentar

Name:

E-Mail:

Website (falls vorhanden):

Kommentar:

[Kommentieren](#)

Picsblog basiert auf [Ruby on Rails](#).

Abbildung 4-8: Picsblog im fertigen Zustand

Aber das war noch längst nicht alles. Im folgenden Kapitel entwickeln Sie Ihr nächstes Projekt. Dabei werden Sie vieles von dem, was Sie bei der Entwicklung von Picsblog gelernt haben, noch einmal in einem anderen Kontext anwenden können. Doch es wird auch noch viel Neues geben, zum Beispiel Ajax-basierte Steuerelemente, grafische Effekte, JavaScript-Templates und Kommunikation via REST – um nur einige Schwerpunkte zu nennen, auf die Sie sich schon freuen können.

TVsendr – mit dem Eigenen sieht man besser

In diesem Kapitel:

- Erst anmelden, dann einschalten!
- TVsendr erzeugen
- Model Broadcast
- Antenne ausrichten
- Wunderwerkzeug Rails-Console
- Informationen zur Sendung
- Das Grundlayout
- Auf dem Weg zum Programmchef

Im vorangegangenen Kapitel 4 haben Sie gesehen, wie Sie mit Ruby on Rails eine Photoblog-Software zaubern können. Dabei haben Sie wichtige Techniken kennengelernt, die Sie gut gebrauchen können, wenn Sie zukünftig mit Rails Ihre eigenen Anwendungen entwickeln. Sie wissen, wie Sie mit Daten umgehen müssen, wie sich die Kommunikation zwischen den einzelnen Ebenen *Model*, *View* und *Controller* gestaltet und sind in den Genuss feinster Rails-Technologien wie *Migrations*, *Metaprogrammierung* und *Validierung* gekommen. Alles sehr schön, aber noch längst nicht alles, was Ruby on Rails zu bieten hat.

So wie einst aus starren Fotos bewegliche Bilder wurden, deren Aneinanderreihung zweckmäßigerweise Film genannt wird, so haben sich auch Websites weiterentwickelt. Auch sie waren einst starre Gebilde, die nur durch komplette Reloads dazu bewegt werden konnten, überhaupt einmal andere Inhalte anzuzeigen. Seit einiger Zeit ist das anders. Websites können heute so flink und dynamisch sein wie Tour-de-France-Teilnehmer. Im Gegensatz zu den eben erwähnten Berufspedaltretern gibt es im Web kein Geheimnis um das Wundermittel, das diese Leistungssteigerung gegenüber früheren Zeiten ermöglicht. *Ajax* heißt es .

Ajax und Ruby on Rails. Das ist perfekte Harmonie. Ruby on Rails war das erste Framework für Web-Applikationen, das mit einer umfangreichen Unterstützung für *Asynchronous JavaScript and XML* aufwartete. Von Beginn an war Ajax nicht irgendeine Hype-bedingte Dreingabe, sondern elementar für das ganze Framework. Und wenn man sich die Ajax-Implementation in Rails genauer ansieht, was wir gleich tun werden, kann man selbige fast schon als liebevoll bezeichnen.

Aber diesen zärtlichen Umgang mit Ajax können Sie gleich selbst erfahren. Denn in diesem Kapitel soll es um eine Anwendung gehen, die einen Großteil ihrer Funktionalität durch Ajax erhält. Schon allein das qualifiziert sie zu einer modernen Anwendung, die im *Web 2.0* zu Hause ist. Noch eindeutiger wird diese Zuordnung, wenn ich Ihnen sage, dass die folgende Anwendung sich eines externen Webservices bedient. Auch das ist ziemlich *Web-2.0-ig*.

In diesem Kapitel werden Sie, Ihre engagierte aktive Teilnahme vorausgesetzt, mit Hilfe der Programmierer-Schnittstelle von *AOL Video Search* Ihren eigenen Fernsehsender aus dem Boden, besser aus dem Browser stampfen. Sie werden in diesem Beispiel erleben, wie Sie mit externen Datenquellen via REST kommunizieren können und die gewonnenen Daten für Ihre ganz eigenen Zwecke modellieren und einsetzen können.

Die Anwendung, welche übrigens den trendigen Titel *TVsendr* tragen wird, durchforstet das dank *YouTube*, *MySpace* und all den anderen prall gefüllte Internet nach Videos, die Ihren Wünschen entsprechen, und ermöglicht Ihnen anschließend, aus den Unmengen an Material, Ihr eigenes TV-Programm zusammen zu stellen. Natürlich mit so viel Sendern, wie Sie wollen: *myMusic* bietet Ihnen coole Musikvideos, *HomerOnDemand* sendet die besten Folgen einer gewissen gelben Fernsehfamilie und *TVGaga* zeigt, dass die PISA-Studie maßlos untertrreibt. Dieser Fernsehgenuß ist natürlich GEZ- und werbefrei.

Darüber hinaus werden Sie sehen, wie Sie dank *Scaffolding* binnen weniger Augenblicke die Basis Ihrer Anwendung erzeugen können.

Ein Ajax-basiertes Video-Mashup, dessen Namen auf *r* und nicht auf *er* endet, mit REST, Ruby on Rails und dessen Scaffolding – also moderner geht's nun wirklich nicht mehr. Nun gut, der Buzzwörter sind genug gewechselt, lassen Sie uns Taten folgen.

Erst anmelden, dann einschalten!

So was aber auch. Jetzt habe ich Ihnen gerade den Mund mit der Versprechung wässrig gemacht, dass unser *TVsendr* frei von GEZ-Gebühren ist, und doch muss ich Sie nun auffordern, sich zunächst einmal anzumelden. Allerdings bei AOL und völlig kostenfrei. Und es gibt noch weitere Vorbereitungen zu treffen, bevor Sie richtig loslegen können.

AOL Video Search API Account

Wie bei vielen anderen ähnlichen Angeboten im Web ist eine solche Anmeldung erforderlich, damit Sie einen Schlüssel erhalten. Mit diesem Schlüssel, dem *Developer Key*, sind Sie berechtigt, den jeweiligen Dienst zu nutzen. Zugleich werden Ihre Anfragen an den Dienst, bei denen Sie den Key stets mitsenden müssen,

protokolliert und gezählt. Bei AOL dürfen Sie beispielsweise standardmäßig nur 10.000 nach Videos lechzende Suchanfragen pro Tag starten. Das reicht aber locker für ungetrübten hausgemachten TV-Genuss.

The screenshot shows the AOL Developer Network homepage. At the top, there's a navigation bar with links for Home, APIs, Community, News & Events, and Resources. The main content area features a banner for the "2007 TOPCODER OPEN Sponsored by AOL". Below the banner, there's a "DEVELOPERQUICKSTART" section with links for Explore, Sign Up & Go, Expand Skills, and Get Tips & Tricks. A text box says: "Whether you're into building something with social networks, video, music, maps, instant messaging and web presence, or more, just follow one overall road map." Below this, there are sections for NEWS & EVENTS (SXSW Interactive MAR 9-13, 2007; ETech MAR 26-29, 2007; Web 2.0 Expo APR 15-17, 2007; World Wide Web MAY 8-12, 2007), FEATURED BLOGS (Starting Your Own Online Community for Convenience and Profit, FEB 28 2006), and FORUMS. On the right side, there's a sidebar titled "Featured APIs" with links for AIM (with a screenshot of its API documentation), AOL Video, userplane, Boxely, AIM Pages, AIM Partners, AIM Phoneline, and AOL Pictures.

Abbildung 5-1: AOL Developer Network: Allerlei Nützliches für den Webentwickler von heute

Wie viele andere Große der Branche, besonders *Yahoo!* und *Google* sind hier zu nennen, bemüht sich AOL in letzter Zeit sehr, der Entwicklergemeinde Tools und Schnittstellen an die Hand zu geben, mit denn sich allerlei geniale Webanwendungen entwickeln lassen. Einen Eindruck davon bekommen Sie, wenn Sie Ihren Browser veranlassen, <http://dev.aol.com> zur Anzeige zu bringen. Schauen Sie sich ruhig um, schreiben Sie Ihre Ideen für zukünftige Applikationen auf, und klicken Sie dann auf *APIs → AOL Video*.

Auf der folgenden Webseite erhalten Sie einen knappen Überblick über *AOL Video Search*. Im linken Menü finden Sie im zweiten Abschnitt den Punkt *My API Account*, welchen Sie mit einem gezielten Mausklick würdigen sollten. Sollten Sie bereits einen *AOL-Screennamen* besitzen, so loggen Sie sich jetzt mit diesem ein. Wenn Sie den *AOL Instant Messenger* benutzen, genügen auch dessen Zugangsdaten. Sollten Sie zu den Menschen gehören, die sich bislang erfolgreich vor AOL gedrückt haben – es gab Zeiten, da war das nicht ganz so leicht, wie heute – sind Sie jetzt fällig. Kostet ja nix.

Anschließend sollten Sie die Nutzungsbedingungen abnicken und eine Webadresse bekannt geben, unter der die Anwendung, welche AOL Video Search nutzt, zu erreichen ist (oder irgendwann eventuell einmal zu erreichen sein wird). Und schon sind Sie Schlüsselbesitzer. Sie finden ihn auf der Bestätigungsseite als *appid* verklei-

det. Darunter sehen Sie die Nutzungsstatistik, welche noch jungfräulich sein sollte. Hier können Sie jedoch später die Anzahl Ihrer täglichen Anfragen an AOL Video Search ablesen. Am besten, Sie notieren sich den Schlüssel oder speichern ihn in einer Datei. Wir werden ihn gleich benötigen.



Shared Secret kann übrigens erst einmal ein Geheimnis bleiben. Der dort hinterlegte Wert wird erst interessant, wenn Sie Anwendungen mit AOL Video Search entwickeln, die weit über das Absenden von Suchanfragen hinaus gehen.

Wenn Sie nun noch auf das Knöpfchen *Test My API Account* klicken, erhalten Sie schon einen Vorgesmack auf das, was Ihre Anwendung gleich verarbeiten muss. Das ist reinstes XML und gleichsam das Format, in dem AOL seine Daten an Ihren selbst gezimmerten Fernseher ausliefert. Sieht aber viel schlimmer aus, als es mit Ruby ist.

XML? Kein Problem!

Um aber ganz easy mit XML umgehen zu können, werden wir uns einer kleinen, aber feinen Bibliothek bemächtigen, die besonders die (spätestens nach diesem Kapitel) ehemaligen Perl-Entwickler unter Ihnen schon kennen könnten. *XML Simple* heißt das gute Stück. Es kann, sollte es noch nicht den Weg gefunden haben, mit RubyGems ganz einfach auf Ihren Rechner gelockt werden. Also flugs die Kommandozeile geöffnet und RubyGems gestartet, und zwar wie folgt:

```
gem install xml-simple --include-dependencies
```

Wenn Ihnen RubyGems mitteilt, dass *xml-simple successfully installed* wurde, ist die Sache schon erledigt und Sie können sich bereits der Erzeugung der eigentlich Anwendung widmen.

TVsendr erzeugen

Wie das geht, wissen Sie inzwischen schon ganz gut: InstantRails, falls vorhanden, starten, RadRails öffnen, neues Rails-Projekt anlegen, dieses *tvsendlr* nennen, Rails-Skeleton und Mongrel-Server erzeugen, fertig.

Als nächstes sollten Sie sich wieder um Ihre Datenbank kümmern. Denn natürlich speichert auch TVsendr seine wertvollen, durch Sie generierten Daten in Datenbanktabellen. Öffnen Sie daher *config/database.yml* und nehmen Sie gegebenenfalls Änderungen vor.

Wenn Sie die Vorgaben in *database.yml* zumindest in puncto database übernommen haben, müssen Sie nun die Datenbanken *tvsendlr_development*, *tvsendlr_test*

und tvsendl_production erzeugen. Wenn Sie den Weg über den MySQL-Monitor gehen möchten, dann öffnen Sie ihn in Ihrer Kommandzeile.

```
mysql -u root
```

Wie Sie sich bestimmt erinnern, erfolgt das Anlegen der Datenbanken mit dem CREATE-Statement. Achten Sie wieder auf das Semikolon am Ende jeder Zeile.

```
CREATE DATABASE tvsendl_development;
CREATE DATABASE tvsendl_test;
CREATE DATABASE tvsendl_production;
```

Sobald das erledigt ist, können Sie den MySQL-Monitor mit exit wieder verlassen und den Mongrel-Server starten, den RadRails für Sie angelegt hat. In Ihrem Browser müsste die Anwendung nun unter <http://127.0.0.1:3000> zu erreichen sein.

Ihnen geht das inzwischen bestimmt schon recht flott von der Hand, oder? Na bestens. Und ganz schnell soll es auch weitergehen.

Ich habe Ihnen im 4. Kapitel vom pädagogischen Zwang berichtet, den Lernenden zunächst mit allerlei anspruchsvollem Stoff zu maltretieren, um ihn anschließend zu zeigen, wie viel leichter es auch gehen kann. Das war der Grund, warum Sie alle Actions und Views von *Picsblog* mühsam selbst geschrieben haben. Ich will nicht sagen, dass das nun vorbei ist, aber es wird doch um vieles leichter. Dank *Scaffolding*.

Scaffolding

Wie Sie bereits wissen, ermöglicht Ihnen Scaffolding das Erzeugen von Actions und Views, die für eine häufig verwendete Funktionalität benötigt werden. Es handelt sich dabei um die Teile einer Anwendung, mit denen Sie Datensätze neu anlegen, bearbeiten, anzeigen und löschen können. Auch das Kürzel *CRUD* verbrauchte in diesem Buch bereits Druckerschwärze. Es beschreibt genau diese Funktionalitäten und steht für *Create, Read, Update* und *Delete*.

Scaffolding können Sie in Ihrer Rails-Anwendung in zwei Varianten nutzen. Einerseits via Metaprogrammierung, in dem Sie der Methode scaffold den Namen des Models übergeben, für das die CRUD-Funktionalitäten zur Verfügung gestellt werden sollen. Diese Variante sollten Sie allerdings keine weitere Aufmerksamkeit schenken. Schließlich haben Sie keinerlei Möglichkeit, den durch Metaprogrammierung nur virtuell erzeugten Action- und View-Code zu bearbeiten.

Die andere Variante werden Sie nun live kennen lernen. Sie basiert auf dem *Scaffold-Generator*. Wie diese Bezeichnung schon erahnen lässt, generiert sie echten Quelltext, den Sie natürlich nach Belieben bearbeiten können.

Aber wie sieht dieser Quelltext wohl aus? Sicher nicht viel anders als der, den Sie bei *Picsblog* noch selbst geschrieben haben. Auch Ruby-Generatoren kochen schließlich nur mit, ähm, Ruby.

Erinnern Sie sich beispielsweise an die View `show.rhtml`. Da mussten Sie explizit auf die Attribute `title` und `description` eines Posts zugreifen, um Titel und Beschreibung zu erhalten. Damit der Scaffold-Generator ebenfalls solchen Code erzeugen kann, muss er natürlich wissen, auf welche Datenbankfelder die View basieren soll.

Lassen Sie uns also zunächst eine Datenbanktabelle anlegen. Sie soll `stations` heißen und alle Informationen zu Ihren eigenen Fernsehsendern speichern. Im vorliegenden Fall reicht uns die Bezeichnung und eine kurze Beschreibung eines Sender.



Mit Ihrem Wissen aus dem 4. Kapitel sollte es Ihnen auch leicht gelingen, die Datenbanktabelle für Sender um ein Feld zu erweitern, dass ein hoch geladenes und in einer Bilddatei vorliegendes Logo speichert.

Datenbasis für Scaffolding

Um eine Datenbanktabelle namens `stations` anzulegen, wechseln Sie in den `Generators`-View von RadRails. Befehlen Sie hier die Erzeugung der Migration `create_stations`. Nach der erfolgreichen Geburt steht Ihnen die Datei unter `db/migrate/001_create_stations.rb` zur Verfügung. Ergänzen Sie hier die nötigen Informationen zur Datenbanktabelle `stations`.

```
Beispiel 5-1 : 001_create_stations.rb
class CreateStations < ActiveRecord::Migration
  def self.up
    create_table(:stations) { |t|
      t.column(:name, :string)
      t.column(:description, :text)
    }
  end

  def self.down
    drop_table(:stations)
  end
end
```

Wie Sie sehen, sollen beide Felder Zeichenketten speichern, wobei `description` auf Romangröße vorbereitet wäre. Die Information über die Beschaffenheit einer Datenbanktabellenspalte nutzt Scaffolding, um Formularelemente zu erzeugen. Wie Sie gleich sehen werden, wird das Eingabefeld für `name` durch den Helper `text_field` erzeugt, für `description` wird jedoch der Helper `text_area` genutzt. Da passt eben mehr rein. Bevor Sie sich selbst davon überzeugen können, müssen Sie die Migrations natürlich noch ins System einspeisen. Und das ist immer noch ein *Rake-Task*, der unter dem gleichnamigen Tab zu finden ist.

Wechseln Sie nach dem erfolgreichen Migrieren wieder in den *Generators*-View von RadRails und wählen Sie in der linken Combobox den Generator scaffold aus, denn außer einer Datenbanktabelle namens stations haben Sie schließlich noch nichts weiter erzeugt.

Im rechten Eingabefeld geben Sie an, wie das Model heißt, auf dessen Basis Actions und Views erzeugt werden sollen, gefolgt von dem gewünschten Namen des zu generierenden Controllers.

Die Datenbanktabelle heißt stations, was Station als Model-Bezeichner erfordert. Und den Controller können Sie Stations nennen. Somit ergibt sich für das rechte Eingabefeld: Station Stations. Klicken Sie auf OK oder drücken Sie *Enter* – und sogleich wird's magisch. Den Höhepunkt der Zaubershow erleben Sie aber spätestens dann, wenn Sie in Ihren Browser <http://127.0.0.1:3000/stations> eingeben.

The screenshot shows a web form titled "New station". It has two input fields: "Name" (with an empty input box) and "Description" (with a large empty text area). At the bottom are two buttons: "Create" (highlighted in grey) and "Back".

Abbildung 5-2: Die durch Scaffolding generierte Eingabemaske für einen neuen Sender

Klicken Sie sich ruhig einmal durch Ihre Anwendung. Sie werden sehen, dass wirklich alles Nötige für das Anlegen, Bearbeiten, Anzeigen und Löschen eines Datensatzes vorhanden ist. Zwar ist das, was Ihr Browser dabei anzeigt, sicher nicht das

Werk eines Profi-Webdesigners. Aber da der Code physisch generiert wurde, haben Sie alle Möglichkeiten, das zu ändern. Im Laufe des Projekts werden Sie erfahren, wie Sie dabei verfahren können.

Blicken Sie in den Quellcode, den Sie an den Ihnen bekannten Stellen vorfinden werden, und Sie werden relativ einfach nachverfolgen können, was hier intern abläuft. Sie werden zudem feststellen, dass der durch Scaffolding generierte Code gar nicht so anders ist als der, den Sie für *Picsblog* geschrieben haben.

Sollten Sie bei Ihrer Begutachtung des Quelltextes auch einen Blick in *app/views/stations/_form.rhtml* werfen, so sorgen Sie doch gleich einmal dafür, dass die Textarea, die dank dieses Partials in *new.rhtml* und *edit.rhtml* zum Einsatz kommt, ein bisschen weniger großzügig mit dem ihr zur Verfügung stehenden Platz umgeht. Fünf Zeilen sollten hier reichen.

Beispiel 5-2 : _form.rhtml

```
<%= error_messages_for 'station' %>

<!--[form:station]-->
<p><label for="station_name">Name</label><br/>
<%= text_field 'station', 'name' %></p>

<p><label for="station_description">Description</label><br/>
<%= text_area 'station', 'description', :rows => 5 %></p>
<!--[eoform:station]-->
```

Die erste Zeile des Partials *_form.rhtml* beinhaltet einen Helper, genauer einen *ActiveRecordHelper*, der das ausgibt, was seine Bezeichnung verspricht. *Error_messages_for* zeigt Fehlermeldungen an, die im Zusammenhang mit der hier verwendeten Model-Instanz *station* auftreten können, beispielsweise bei einer fehlgeschlagenen Validierung.

Validierungsregeln für Station

Um das einmal in Aktion zu erleben, sollten Sie nun dem Model *Station* Validierungsregeln verpassen. Ich denke, man kann vom Benutzer verlangen, dass er beide Felder ausfüllt. Zusätzlich wäre es von Vorteil, wenn jeder Sendername nur einmal vergeben werden würde. Im richtigen Fernsehen ist das ja auch so. Kein Sender heißt wie ein anderer, was sie natürlich nicht davon abhält, dass alle den gleichen Quark senden.

Wie dem auch sei. Öffnen Sie *app/models/station.rb* und fügen Sie die Validatoren *validates_presence_of* und *validates_uniqueness_of* mit den entsprechenden Parametern hinzu.

Beispiel 5-3 : Valierungsregeln für station.rb

```
class Station < ActiveRecord::Base
  validates_presence_of(:name, :description)
  validates_uniqueness_of(:name)
end
```

Wenn Sie nun versuchen, zum Beispiel einen Datensatz ohne Beschreibung anzulegen, werden Sie mit Sicherheit daran gehindert und entsprechend informiert. Der Hinweis geht dabei exakt auf das Problem ein und weist Sie auf das Fehlen einer Beschreibung hin.

The screenshot shows a web form titled "New station". A prominent error message at the top states "1 error prohibited this station from being saved". Below this, a list of validation errors is displayed: "There were problems with the following fields: Description can't be blank". The form itself has two input fields: "Name" containing "RailsTV" and "Description", which is currently empty. At the bottom of the form are two buttons: "Create" (highlighted in grey) and "Back".

Abbildung 5-3: Da fehlt doch was!

Natürlich reichen Sender allein nicht aus. Damit sie Sinn ergeben, braucht es Sendungen. Und die zur Verfügung zu stellen, soll Aufgabe eines weiteren Models sein, das den vielversprechenden Namen Broadcast tragen soll.

Model Broadcast

Erzeugen Sie das Model Broadcast über den model-Generator, den Sie unter dem *Generators*-Tab von RadRails finden. Öffnen Sie anschließend die automatisch angelegte Migration *002_create_broadcasts.rb*.

Hier legen Sie fest, welche Charakteristika so eine Sendung aufweisen soll. Grundsätzlich soll es uns reichen, wenn die Datenbanktabelle broadcasts den Titel einer Sendung in einem String-Feld speichert. Darüber hinaus müssen aber noch weitere Spalten erzeugt werden.

Damit jede Sendung einem Sender zugeordnet werden kann, braucht broadcasts noch ein Fremdschlüsselfeld. Da hier die Sender-ID gespeichert werden soll, ergibt sich der Bezeichner dieser Spalte: `station_id`.

Darüber hinaus muss Ihre Anwendung in der Lage sein, die Videodaten einer Sendung bei AOL Video Search abzurufen. Damit das reibungslos funktioniert, bietet die AOL API eine eigene ID an, die jede Sendung bei AOL Video Search eindeutig identifiziert. Diese Video-IDs soll das Feld `video_id` speichern. Dazu sollten Sie ein String-Feld nutzen; wer weiß, ob die Video-IDs, die heute noch aus Zahlen bestehen, nicht einmal buchstabenhaltig werden.

Und noch ein Feld benötigen Sie. Beim Zusammenstellen des Programms werden Sie über Drag and Drop in der Lage sein, die gewählten Sendungen in einer bestimmten Reihenfolge, die Ihren Wünschen entspricht, zu sortieren. Diese Reihenfolge muss natürlich auch irgendwie, irgendwo festgehalten werden. Daher spendieren Sie broadcasts am besten noch ein Feld namens `position`.

Beispiel 5-4 : 002_create_broadcasts.rb

```
class CreateBroadcasts < ActiveRecord::Migration
  def self.up
    create_table :broadcasts do |t|
      t.column(:title, :string)
      t.column(:title, :string)
      t.column(:video_id, :string)
      t.column(:station_id, :integer)
      t.column(:position, :integer)
    end
  end

  def self.down
    drop_table :broadcasts
  end
end
```

Nun können Sie *Rake* beauftragen, sich doch einmal um die eben erstellte Migration zu kümmern. Wählen Sie dazu im View *Rake Tasks* links `db:migrate` aus und bestätigen Sie Ihre Wahl mit *OK*.

Assoziationen

Anschließend können Sie Assoziationen einfügen, welche nötig sind, damit Station und Broadcast harmonisch miteinander verschmelzen – oder zumindest so tun, als ob. Wie bei Picsblog besteht auch hier eine 1:N-Relation: Ein Sender hat mehrere Sendungen; eine Sendung gehört zu einem Sender.



Natürlich kann es vorkommen, dass eine Sendung in mehreren Sendern auf dem Programm steht, was in TVsendlr allerdings trotz 1:N-Relation kein Problem ist. Dann wird eben für eine Sendung mehrere Datensätze mit unterschiedlichen Werten bei station_id angelegt. Möchten Sie diese kleine Unschönheit vermeiden, implementieren Sie eine M:N-Relation: Ein Sender hat mehrere Sendungen, eine Sendung kann zu mehreren Sendern gehören. Selbstverständlich gibt es auch dafür eine Assoziation in Rails, die den kurzen Namen has_and_belongs_to_many trägt und in Rails-Kreisen oft mit habtm abgekürzt wird.

Im Station-Model fügen Sie die has_many-Assoziation hinzu. Wie Sie wissen, können Sie durch die Option order festlegen, in welcher Reihenfolge die assoziierten Datensätze sortiert werden sollen. Im vorliegenden Fall sollen natürlich die Werte in der Spalte position maßgeblich für die Sortierung sein. Zudem soll die Sortierreihenfolge aufsteigend sein.

Außerdem sollten Sie noch verfügen, dass alle Sendungen eines Senders aus der Datenbank gelöscht werden, wenn der Sender dicht macht und somit ebenfalls aus der Datenbank entfernt wird. So wie die Kommentare eines Picsblog-Posts.

Beispiel 5-5 : station.rb

```
class Station < ActiveRecord::Base

  validates_presence_of(:name, :description)
  validates_uniqueness_of(:name)

  has_many(
    :broadcasts,
    :order => 'position ASC',
    :dependent => :destroy
  )
end
```

In broadcast.rb notieren Sie die andere Seite der Relation zwischen den Models. Auch das ist für Sie kein Neuland.

Beispiel 5-6 : broadcast.rb

```
class Broadcast < ActiveRecord::Base
  belongs_to(:station)
end
```

Auf den durch Scaffolding generierten Code haben die Assoziationen keine Auswirkungen, das liegt allein in Ihrer Hand.

Antenne ausrichten

Lassen Sie *broadcast.rb* gleich offen. Hier implementieren wir nun die gesamte Funktionalität, welche Ihre Anwendung benötigt, um mit *AOL Video Search* zu kommunizieren. Ich finde, das Broadcast-Model ist ein guter Platz dafür, da es sich schließlich per definitionem um alles, was Sendungen betrifft, kümmern soll.

Das hat Methoden zur Konsequenz, die Anfragen an AOL initiieren und die darauf hin zurückgegebenen Daten verarbeiten können. Wie diese Daten aussehen, haben Sie bei der Erlangung Ihres Entwicklerschlüssels auf der AOL-Webseite gesehen.

Damit Ihre Anwendung mit externen Webressourcen kommunizieren kann, benötigen Sie die zusätzliche Funktionalität aus den Ruby-eigenen Bibliotheken *Net::HTTP* und *URI*, welche Sie bekanntermaßen über *require* einbinden können.

Beispiel 5-7 : Net::HTTP und URI in broadcast.rb einbinden

```
require('net/http')
require('uri')

class Broadcast < ActiveRecord::Base
  belongs_to(:station)
  ...

```

Nun kann das Programm über HTTP Verbindungen zu anderen Orten im Internet aufnehmen und so auch via REST mit der AOL API kommunizieren.

AOL Video Search befragen

Um mit AOL Video Search in Verbindung zu treten, brauchen Sie zunächst einmal Ihren eingangs erhaltenen Schlüssel und eine Webadresse, unter der Sie die Entwicklerschnittstelle erreichen können. Es ist keine schlechte Idee, beides als Konstante festzulegen. Zumindest der URL der AOL-API macht den Eindruck, als könne er sich irgendwann einmal ändern: <http://beta.searchvideo.com/apis3>. Diese Adresse finden Sie natürlich auch in der recht guten Dokumentation der API auf http://dev.aol.com/aol_video.

Beispiel 5-8 : Schlüssel und URL als Konstante in broadcast.rb

```
...
class Broadcast < ActiveRecord::Base

  belongs_to(:station)

  AOL_API_URL = 'http://beta.searchvideo.com/apisv3'
  AOL_DEVELOPER_ID = 'xxxxxxxxxxxxxxxxxx'
  ...

```

Beachten Sie, dass Sie beim Eintragen Ihres eigenen Schlüssels in AOL_DEVELOPER_ID die Groß- und Kleinschreibung berücksichtigen. Nur mit einem korrekten Schlüssel ist der Zugriff auf die AOL API möglich.

Da die Kommunikation mit AOL Video Search über REST erfolgt, bietet sich eine Methode an, die einen URL nach API-Vorschrift erzeugt und diesen als HTTP-Request an AOL schickt. Wie diese Vorschrift aussieht, können Sie der Dokumentation zur API entnehmen.

Mit REST Suchanfragen starten

Um nach Videos zu suchen, können Sie mit REST die Methode truveo.videos.getVideos der AOL API aufrufen. Sie erwartet als Parameter Details zur Suche, wie viele Ergebnisse zurück gegeben werden sollen und ab welcher Position dies geschehen soll.

Eine Erklärung zu den zuletzt genannten Angaben: Die AOL Video Search kann durchaus hunderttausende Videos zu einer Suche finden. Damit nicht alle Ergebnisse in einer Megabyte großen XML-Datei an den anfragenden Client gesendet werden muss, können Sie über diese Werte paginierte Ergebnisse erhalten. Nicht zu vergessen: Der Entwicklerschlüssel muss ebenfalls in den Anfrage-URL. So ergibt sich folgender Query-String, der an die in AOL_API_URL angehangen werden muss:

```
?method=truveo.videos.getVideos&query=dogs&results=10&start=50&appid=xxxxxx
```

Diese Anfrage fordert AOL Video Search auf, Videos herauszusuchen, die in Titel, Beschreibung oder Stichwörtern das Wort *dogs* enthalten. Zehn Stück sollen es sein, allerdings beginnend ab dem 50. gefundenen Video.

Über die Formulierung der Suchanfrage, die weit über einen Suchbegriff hinausgeht, gibt es auf den Dokumentationsseiten bei AOL ein eigenes Kapitel. Das ist angemessen, schließlich gibt es einige Optionen, die Sie dabei einfließen lassen können.

TVsendl soll es dem Anwender ermöglichen, nach Videos bei bestimmten Anbietern zu suchen. So ist eine Suche nach *dogs* ausschließlich bei YouTube, dem wohl größten Video-Sharer unter der Sonne, möglich. In der Suche wird das über *Channels* gelöst.

REST. Was ist das eigentlich?

REST steht für *REpresentational State Transfer* und bezeichnet ein Modell, das beschreibt, wie vernetzte Systeme miteinander funktionell verbunden werden können.

Im Word Wide Web erfreut sich REST immer größerer Beliebtheit als Standard für die Verwendung von Webservices. Dabei handelt es sich um Angebote von Websites, Teile ihrer Funktionalität über das herkömmliche *Hyper Text Transfer Protocol (HTTP)* fernsteuern zu können. Bei TVsendr wird beispielsweise die AOL Video Search ferngesteuert.

REST ist gerade für Neueinsteiger in die Welt der Webservices ideal, da oft herkömmliche URLs ausreichen, um entfernte Webservice-Funktionen bedienen zu können. Grundvoraussetzung dafür ist die Bildung eines URLs nach den Vorschriften des Webservices. Wenn man will, kann man sogar sagen, dass das gesamte Web durch die intensive Verlinkung der Websites untereinander eine einzige, riesige REST-Anwendung ist.

REST-APIs im Web können durch folgende Punkte charakterisiert werden:

- In jeder REST-Anfrage sind alle Informationen enthalten, die der Webservice braucht, um die Anfrage korrekt zu bearbeiten. Zwischenspeicherlösungen zwischen zwei Anfragen, wie zum Beispiel Cookies, sind dabei nicht vorgesehen. Allerdings werden Sie dennoch von einigen Webservices genutzt. In TVSendr wird aus diesem Grund beispielsweise bei jeder und nicht nur bei der ersten Anfrage der API-Key mitgesendet.
- Das HTTP sieht für die Übertragung verschiedene Modi vor. Die bekanntesten sind GET und POST, weitere sind PUT und DELETE. Durch die Benutzung dieser HTTP-Request-Methoden, die auch *HTTP-Verben* genannt werden, können Informationen oder andere Ressourcen des Webservices angefordert (GET), erzeugt (POST), verändert (PUT) oder gelöscht (DELETE) werden. TVSendr ruft lediglich Informationen ab, ohne selbige verändern zu wollen. Daher kommt hier nur GET als HTTP-Request-Methode in Frage.
- REST-Anfragen betreffen Ressourcen des Webservices. Werden Daten zurückgeliefert, so bilden diese die Repräsentation dieser Ressourcen. Meist werden diese Daten als XML oder HTML formatiert. TVSendr empfängt die Repräsentation der Daten von AOL Video Search als XML.
- Die Interaktion zwischen einer Webanwendung und einem Webservice wird grundsätzlich durch den Client initiiert. Nur wenn TVSendr einen HTTP-Request an die API von AOL Video Search sendet, erhält die Anwendung auch ihre gewünschten Daten.



Die Beliebtheit von REST schlägt sich mittlerweile auch in Ruby on Rails nieder. Allerdings nicht im Zusammenhang mit Webservices, sondern bei der Benutzung von Controllern. Wie Sie wissen, basiert REST auf GET, POST, PUT und DELETE. Die Aktivitäten, die durch diese HTTP-Verben ausgeführt werden, passen ideal zu CRUD-Anwendungen, wie sie beispielsweise durch Scaffolding erzeugt werden können. Die Ressourcen in Rails-Anwendungen sind dabei Datensätze.

Die Zukunft von Rails wird so aussehen, dass Sie Controller über den URL nicht mehr in Verbindung mit einer Action und gegebenenfalls einer Datensatz-ID aufrufen müssen, sondern nur noch mit der ID. Was mit dieser Ressource (Datensatz) geschehen soll, wird durch die dabei verwendete HTTP-Request-Methode an die Anwendung übermittelt, die wiederum die erforderliche Action auslöst.

RESTful Rails, wie diese Entwicklung genannt wird, sorgt auch dafür, dass eine Action clientabhängige Daten ausgeben kann. So kann eine Action je nach Art des anfragenden Clients einerseits Daten für den Webbrower rendern, andererseits aber auch als XML ausgeben, was wiederum jede Rails-Anwendung als Webservice in Erscheinung treten lässt.

Diese Entwicklung begann mit der Veröffentlichung von Rails 1.2 und wird sich sicher in kommenden Versionen verfestigen.

Außerdem sollen möglichst nur Videos zurückgeliefert werden, die direkt im Browser mit einem Flash-Player abgespielt werden können. Diese Einschränkung ist erforderlich, da AOL auch Videos findet, die erst herunter geladen werden müssen, um abgespielt werden zu können. Zu diesem Zweck stehen Ihnen *Format-Filter* zur Verfügung.



Leider bedeutet die Tatsache, dass Sie nur nach Flash-Videos suchen lassen, nicht automatisch, dass auch alle Videos direkt in TVsendlern abgespielt werden können. Urheberrechtliche Einschränkungen oder spezielle Nutzerwünsche sorgen dafür, dass Ihnen manches Mal nur ein Link zur Quellseite zur Verfügung gestellt wird. Eine Funktionalität, mit der man dieses Manko umgehen kann, ist derzeit nicht in der AOL Video Search API enthalten.

Mit einem weiteren Filter können Sie bestimmen, dass Sie nur frei verfügbare Videos verwenden möchten. Sie können einen Type-Filter benutzen, um dies sicherzustellen.

Fügen Sie Filter, getrennt durch ein Leerzeichen, dem Suchbegriff oder den Suchbegriffen hinzu. Dabei gilt stets das Format: <Filtername>:<Filterwert>. Eine Suche nach frei verfügbaren dogs-Videos bei Google Video im Flash-Format sähe demnach als *REST-Request* so aus:

```
?method=truveo.videos.getVideos&query=dogs type:free format:flash channel:  
"Google Video"&results=10&start=50&appid=xxxxxxx
```

Der Wert von query ist nicht ganz sauber. Leerzeichen und Anführungszeichen haben in Query-Strings nichts zu suchen und müssen *escaped* werden. Ein Leerzeichen wird zu %20, ein Anführungszeichen metamorphosiert zu %22.

```
?method=truveo.videos.getVideos&query=dogs%20type:free%20format:flash%20channel:  
%22Google Video%22&results=10&start=50&appid=xxxxxxx
```

Darum müssen Sie sich aber nicht selbst kümmern. Das macht die Klassenmethode `URI.escape` für Sie. Und damit können Sie mit der Implementation der Methode `Broadcast#get_videos` beginnen.

Beispiel 5-9 : `Broadcast#get_videos`

```
def self.get_videos(search, channel, per_page, page)
  url = AOL_API_URL +
    "?method=truveo.videos.getVideos" +
    "&query=" + URI.escape("#{search} type:free format:flash channel:\#{channel}\\" +
  "") +
    "&results=#{per_page.to_i}" +
    "&start=#{(page.to_i - 1) * per_page.to_i}" +
    "&appid=#{AOL_DEVELOPER_ID}"
  data = Broadcast.get_xml(url)
  return data
end
```

Wie Sie sehen, erwartet die Klassenmethode `Broadcast#get_videos` vier Parameter: Den Suchbegriff, den gewünschten Channel, die Anzahl der Ergebnisse pro Seite und für welche Seitennummer die Ergebnisse zurückgegeben werden sollen. Da die REST-Anfrage die Indexnummer des ersten Videos erwartet und mit Seitennummern nicht umgehen kann, erfolgt eine Umrechnung.

Die fertig zusammen gesetzte Webadresse wird nun der Methode `Broadcast#get_xml` übergeben. In dieser Klassenmethode, die wir nun quelltexten werden, wird XML Simple um tatkräftige Unterstützung gebeten. Diese Bibliothek soll die ebenfalls in `get_xml` angeforderten XML-Daten in eine Ruby-freundlichere Struktur, bestehend aus Arrays und Hashes, umwandeln.

Damit Sie verstehen, wie XML Simple arbeitet, lohnt zunächst ein Blick in beispielhafte und verkürzte XML-Daten, die von AOL erzeugt werden und Berliner Lokalkolorit beinhalten.

Beispiel 5-10 : Berlin-Videos im XML-Format

```
<?xml version="1.0"?>
<Response xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://beta.
searchvideo.com" xsi:schemaLocation="http://beta.searchvideo.com apiv3.xsd">
  <method>truveo.videos.getVideos</method>
  <query>berlin type:free format:flash channel:"YouTube"</query>
  <VideoSet>
```

Beispiel 5-10 : Berlin-Videos im XML-Format (Fortsetzung)

```
<totalResultsAvailable>124</totalResultsAvailable>
<totalResultsReturned>20</totalResultsReturned>
<firstResultPosition>0</firstResultPosition>
<title>Videos that match your query: berlin type:free format:flash channel:
"YouTube"</title>
<rssUrl>http://beta.searchvideo.com/rss?query=berlin type:free format:
flash channel:"YouTube"&showAdult=1</rssUrl>
<embedTag><embed src="http://developer.searchvideo.com/apps/listWidget/listWidget.
swf?query=berlin type:free format:flash channel:
"YouTube"&resultsnum=20" type="application/x-shockwave-flash" width="600" height="100">
</embed></embedTag>
<Video>
  <id>1593972341</id>
  <title>S8 am Ostkreuz nach Zeuthen</title>
  <videoUrl>http://beta.searchvideo.com/
rd?i=1604546326&a=f328127a912d194c8bf189f398b4d4fa&p=2</videoUrl>
  <channel>YouTube</channel>
  <channelUrl>http://www.youtube.com</channelUrl>
  <dateFound>Wed, 21 Feb 2007 22:03:21 -0500</dateFound>
  <textRelevancy>0.7866</textRelevancy>
  <vRank>0.188026</vRank>
  <description>Ein Zug der BR 480 verlässt den Bahnhof Ostkreuz nach Zeuthen.</
description>
  <referrerPageUrl>http://www.youtube.com</referrerPageUrl>
  <category>Travel</category>
  <tags>Berlin, ostkreuz, s-bahn</tags>
  <thumbnailUrl>http://thumbnail.search.aolcdn.com/truveo/images-thumbnails/C4/7F/
C47F23AD80F471.jpg</thumbnailUrl>
  <videoResultEmbedTag><embed src="http://developer.searchvideo.com/apps/
videoWidget/videoWidget1.swf?query=id:1593972341" type="application/x-shockwave-
flash" width="400" height="92"></embed></videoResultEmbedTag>
  <videoPlayerEmbedTag><object width="425" height="350">
<param name="movie" value="http://www.youtube.com/v/Gb40DtGkAE"></param>
<param name="wmode" value="transparent"></param><embed src="http://www.youtube.com/v/
Gb40DtGkAE" type="application/x-shockwave-
flash" wmode="transparent" width="425" height="350"></embed></object></
videoPlayerEmbedTag>
  <formats>flash</formats>
</Video>
<Video>
  <id>2072451617</id>
  <title>Wolf fish (Steinbitt) at KaDeWe in Berlin.</title>
  <videoUrl>http://beta.searchvideo.com/
rd?i=2078550032&a=f328127a912d194c8bf189f398b4d4fa&p=3</videoUrl>
  <channel>YouTube</channel>
  <channelUrl>http://www.youtube.com</channelUrl>
  <dateFound>Fri, 23 Feb 2007 05:13:17 -0500</dateFound>
  <textRelevancy>0.6963</textRelevancy>
  <vRank>0.188026</vRank>
  <description>My son, David and I, eat Wolf fish (Steinbitt) at KaDeWe in Berlin.
</description>
```

Beispiel 5-10 : Berlin-Videos im XML-Format (Fortsetzung)

```
<referrerPageUrl>http://www.youtube.com</referrerPageUrl>
<category>Travel</category>
<tags>ridanos</tags>
<thumbnailUrl>http://thumbnail.search.aolcdn.com/truveo/images-thumbnails/46/CD/
46CDE5FF8626D1.jpg</thumbnailUrl>
<videoResultEmbedTag><embed src="http://developer.searchvideo.com/apps/
videoWidget/videoWidget1.swf?query=id:2072451617" type="application/x-shockwave-
flash" width="400" height="92"></embed></videoResultEmbedTag>
<videoPlayerEmbedTag><object width="425" height="350">
<param name="movie" value="http://www.youtube.com/v/sN8vkjbf0c"></param>
<param name="wmode" value="transparent"></param><embed src="http://www.youtube.com/v/
sN8vkjbf0c" type="application/x-shockwave-
flash" wmode="transparent" width="425" height="350"></embed></object></
videoPlayerEmbedTag>
<formats>flash</formats>
</Video>
</VideoSet>
</Response>
```

XML ist wie HTML ein Format zum Auszeichnen von Informationen. Jedes kleine bisschen Information ist von einem Tag umgeben. Diese Tags wiederum sind von weiteren Tags umgeben, wodurch sich eine hierarchische, baumähnliche Struktur ergibt.

Um an den Inhalt des Elements totalResultsAvailable zu gelangen, welches die Anzahl aller gefundenen Videos beinhaltet, müssen Sie zunächst das übergeordnete Element VideoSet ansteuern. Innerhalb von VideoSet gibt es darüber hinaus noch mehrere Video-Elemente, die jeweils ein gefundenes Video repräsentieren. Ein Video-Element enthält beispielsweise den Titel eines Videos. Um also an diese Information zu gelangen, müssen Sie den Pfad <VideoSet>-<Video>-<title> entlang wandern; für den Titel des ersten Video-Elements innerhalb von VideoSet steuern Sie natürlich auch das erste Video-Element an.

XML Simple macht das ganz ähnlich, nutzt dafür aber Ruby-Objekte. XML Simple parst XML-Daten und gibt als Ergebnis ein Objekt zurück, in dessen Attributen das gesamte Dokument enthalten ist. Um dieses Objekt nach dem Titel des ersten Videos zu befragen, müssen Sie auch einen Weg beschreiten, der allerdings mit Hashes und Arrays gepflastert ist.

```
xml_obj['VideoSet'][0]['Video'][0]['title'][0]
```

In diesem Beispiel hat XML Simple sein Ergebnis in `xml_obj` abgeladen. Auch hier geht der Weg zunächst zum VideoSet-Element, welches hier als Hash-Schlüssel vorliegt. Der Wert dieses Hash-Schlüssels ist ein Array, dessen Elemente alle VideoSet-Elemente der XML-Daten auf der obersten Ebene repräsentieren. Da es hier nur ein VideoSet-Element gibt, ist das erste, erreichbar durch den Index 0, für uns von Interesse.



Bitte beachten Sie, dass Hash-Schlüssel, die Sie für die Wegbeschreibung in dem von XML Simple zurück gegebenen Objekt verwenden, den korrespondierenden Tags der XML-Daten entsprechen müssen. Dies trifft insbesondere auf die Groß- und Kleinschreibung zu.

Nun folgt der nächste Schritt. Durch ['VideoSet'][0] haben Sie nun Zugriff auf alle in <VideoSet> enthaltenen Video-Elemente. Hierbei gilt ein weiteres Mal: Durch die Kombination aus Hash-Schlüssel und Array-Index können Sie an den Inhalt des jeweiligen Elements gelangen.

Der Inhalt eines Video-Elements besteht unter anderem aus einem title-Element. Da es hier nur eins gibt, ist [:title][0] der letzte Schritt auf dem Weg zur Erlangung des Titels des ersten Videos des von AOL zurück gegebenen XML-Datenmaterials.



Statt mit [:title][0] können Sie natürlich auch mit [:title].first auf den Titel eines Videos zugreifen. Array#first eignet sich gut, wenn Sie genau wissen, dass es nur ein Element der entsprechenden Sorte gibt.

Mit diesem Wissen sind Sie nun für die Methode Broadcast#get_xml gerüstet. Sie erhält den in get_videos generierten REST-URL.

Beispiel 5-11 : Broadcast#get_xml

```
def self.get_xml(url)

  # Data-Objekt initialisieren
  data = Hash.new
  data[:count] = 0
  data[:items] = Array.new

  # XML-Daten holen
  xml = Net::HTTP.get_response(URI.parse(url))

  # HTTP-Response-Code prüfen
  data[:http_code] = xml.code.to_i
  return data unless data[:http_code] == 200

  # XML Parsen
  xml_obj = XmlSimple.xml_in(xml.body.to_s)
  data[:count] = xml_obj['VideoSet'][0]['totalResultsAvailable'][0].to_i
  return data if data[:count] == 0

  # Informationen auslesen
  xml_obj['VideoSet'][0]['Video'].each { |xml_video|
    new_video = Hash.new
    new_video[:video_id] = xml_video['id'][0].to_s
```

```

Beispiel 5-11 : Broadcast#get_xml (Fortsetzung)

    new_video[:title] = xml_video['title'][0].to_s
    new_video[:source_url] = xml_video['videoUrl'][0].to_s
    new_video[:thumbnail_url] = xml_video['thumbnailUrl'][0].to_s
    new_video[:description] = (xml_video['description'][0].to_s) if xml_
video['description']
    new_video[:video_html] = (xml_video['videoPlayerEmbedTag'][0].to_s) if xml_-
video['videoPlayerEmbedTag']
    data[:items] << new_video
}

return data

end

```

In dieser Methode wird zunächst ein Hash-Objekt erzeugt, dass all die Daten aufnehmen soll, die in den nächsten Zeilen generiert werden. Dabei wird der Schlüssel count, der gleich die Anzahl der gefundenen Videos speichern wird, mit 0 initialisiert und in items ein Array erzeugt, welches sich den Daten der gefundenen Videos aufnehmen wird.

Im nächsten Schritt nimmt Net::HTTP#`get_response` Verbindung zur AOL-REST-Schnittstelle auf. Das Ergebnis ist `xml`, ein Objekt der Klasse `HTTPResponse`. Elementare Attribute dieses Objekts sind `code` – durch den erfahren Sie, ob die Anfrage erfolgreich war – und `body` – hier steckt XML drin.

Ist der *HTTP-Status-Code* der Response alles, nur nicht 200 (was für eine astreine Übertragung steht), dann bricht die Methode vorzeitig ab, so dass es nicht mehr zum Auswerten der XML-Daten kommt. Sie können in diesem Falle davon ausgehen, dass ein Fehler bei der REST-Anfrage oder während der Datenübertragung aufgetreten ist.

Sollte der HTTP-Code 200 sein, lohnt das Auslesen des Datenmaterials. Dazu übergeben Sie `XmlSimple#xml_in` den Inhalt von `xml.body`. Sogleich macht XML Simple daraus die bereits weiter oben vorgestellte Hash-Array-Kombi, die das XML-Dokument Ruby-konform zur Verfügung stellt. Mit `each` können Sie dann über alle Video-Elemente iterieren, und alles aus ihnen heraus holen, was Sie interessiert: die AOL-interne Video-ID, der Titel, die Quell-URL, unter der das Video im Internet zu finden ist, die Thumbnail-URL, die ein kleines Vorschaubild bereithält, die Beschreibung und den HTML-Code, mit dem Sie das Video samt Flash-Player in TVsendr einbauen können – sofern vorhanden.

All diese Daten werden temporär in den Hash `new_video` gepackt und anschließend als neues Element dem Array `data[:items]` angehängt.

Auf Basis der Anzahl der gefundenen Videos lässt sich nun bestimmen, wie viele Seiten nötig sind, um dem Benutzer von TVsendr alle Videos zur Verfügung zu stellen.

```

def self.get_videos(search, channel, per_page, page)

  url = AOL_API_URL +
    "?method=truveo.videos.getVideos" +
    "&query=" + URI.escape("#{search} type:free format:flash channel:\\" +
  "#{channel}\") +
    "&results=#{per_page.to_i}" +
    "&start=#{(page.to_i - 1) * per_page.to_i}" +
    "&appid=#{AOL_DEVELOPER_ID}"
  data = Broadcast.get_xml(url)

  if data[:http_code] == 200
    data[:page_count] = data[:count] / per_page
    data[:page_count] += 1 if data[:count] % per_page > 0
  end

  return data
end

```

Hierbei wird die in data[:count] vorliegende Zahl durch die Menge an Videos pro Seite geteilt. Sollte dabei ein Division mit Rest (und Rest heißt hier wirklich Rest und nicht REST) durchgeführt worden sein, erhöht sich die Seitenzahl um 1, damit auch die restlichen Videos auf der letzten Seite des Suchergebnisses angezeigt werden können.

Damit wäre die komplette Funktionalität zum Abrufen von Informationen über Videos bereits implementiert. Obwohl – ich kann Ihnen ja viel erzählen, oder? Ich bin mir sicher, Sie würden das eben Implementierte gern mal ausprobieren. Auf den ersten Blick erscheint das etwas schwierig, fehlt doch noch der ganze Controller-Code, der die Daten von Broadcast#get_videos in die Ihrer Anwendung befördert. Aber wie Sie sich vielleicht schon denken können: Es gibt da was.

Wunderwerkzeug Rails-Console

Die Rails-eigene Console ist ein äußerst nützliches Werkzeug während der Entwicklungs- und Testphase von Rails-Anwendungen. Mit der Rails-Console können Sie sich direkt in das Herz Ihrer Applikationen vordringen und dort so tun, als wären Sie ein Controller. Sie hat grundlegende Ähnlichkeiten mit *Interactive Ruby (irb)*, ist aber spezialisiert auf die Anwendung, in deren Verzeichnis die Rails-Console aufgerufen wird.



Verwechseln Sie die Rails-Console nicht mit der, die Ihnen in RadRails zur Verfügung steht. Beide haben nichts miteinander zu tun.

Sie starten die Rails-Console über die Kommandozeile. Wechseln Sie vorab in das Wurzelverzeichnis der Anwendung, für die die Rails-Console gültig sein soll. Bei mir ist das C:\InstantRails\rails_apps\tvsendl. Geben Sie dann ein:

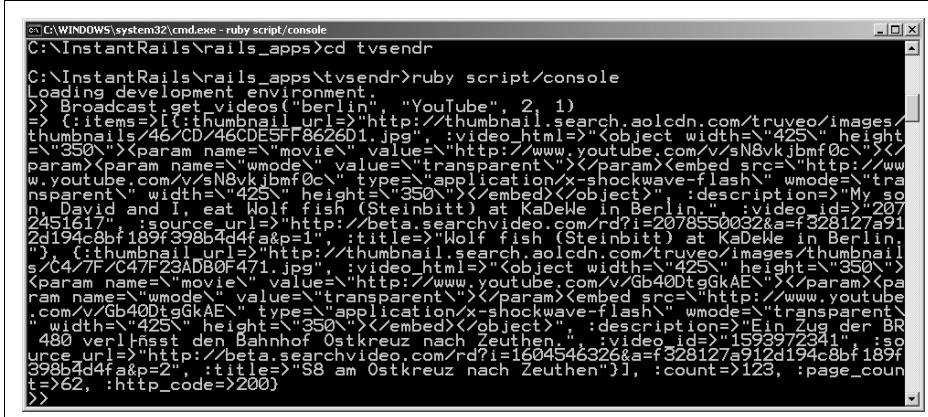
```
ruby script/console
```

Sollten Sie kein Windows-User sein, können Sie sich das vorangestellte ruby und damit den expliziten Aufruf des Interpreters sparen. Ihr System erkennt in diesem Fall, womit das Skript interpretiert werden muss.

Wie bereits erwähnt, können Sie jetzt so tun, als wären Sie ein Rails-Controller. Somit können Sie beispielsweise unsere eben fertig gestellte Klassenmethode Broadcast#get_videos aufrufen.

```
Broadcast.get_videos('berlin', 'YouTube', 2, 1)
```

Bestätigen Sie Ihre Eingabe mit *Enter* und schon schickt Broadcast.get_videos eine Anfrage an AOL Video Search ab. Sobald das Ergebnis da ist, wird es mithilfe von XML Simple umgewandelt und in ein Hash-Objekt verpflanzt.



```
C:\WINDOWS\system32\cmd.exe - ruby script/console
C:\InstantRails\rails_apps\tvsendl>cd tvsendl
C:\InstantRails\rails_apps\tvsendl>ruby script/console
Loading development environment
=> Broadcast.get_videos("berlin", "YouTube", 2, 1)
=> :items=>[{:thumbnail_url=>"http://thumbnail.search.aolcdn.com/truveo/images/thumbnails/46/CD/46CDE5F8626D1.jpg", :video_html=><object width="425" height="350"><param name="movie" value="http://www.youtube.com/v/sN8vkjbmf0c"></param><param name="wmode" value="transparent"></param><embed src="http://www.youtube.com/v/sN8vkjbmf0c" type="application/x-shockwave-flash" wmode="transparent" width="425" height="350"></embed></object>, :description=>"My son, David and I, eat Wolf fish (Steinbitt) at KaDeWe in Berlin.", :video_id=>"2072451617", :source_url=>"http://beta.searchvideo.com/rd?i=2078550032&a=f328127a912d194c8bf189f398b4d4fa&p=1", :title=>"Wolf fish (Steinbitt) at KaDeWe in Berlin."}, {:thumbnail_url=>"http://thumbnail.search.aolcdn.com/truveo/images/thumbnails/C4/7F/C47F23ADBF471.jpg", :video_html=><object width="425" height="350"><param name="movie" value="http://www.youtube.com/v/Gb40DtgGkAE"></param><param name="wmode" value="transparent"></param><embed src="http://www.youtube.com/v/Gb40DtgGkAE" type="application/x-shockwave-flash" wmode="transparent" width="425" height="350"></embed></object>, :description=>"Ein Zug der BR 480 verlässt den Bahnhof Ostkreuz nach Zeuthen.", :video_id=>"1592972341", :source_url=>"http://beta.searchvideo.com/rd?i=1604546326&a=f328127a912d194c8bf189f398b4d4fa&p=2", :title=>"S8 am Ostkreuz nach Zeuthen"}], :count=>123, :page_count=>62, :http_code=>200]
=>
```

Abbildung 5-4: Broadcast#get_videos im ersten Test

Die Ausgabe von Broadcast#get_videos präsentiert sich zwar als buntes Sammelsurium von allerlei Zeichen. Wenn Sie aber genau hinsehen, erkennen Sie die Struktur der Daten wieder.



Sollten Sie während Ihrer Arbeit mit der Rails-Console Veränderungen am Quelltext vornehmen, der unmittelbaren Einfluss auf Ihre Tests in der Console haben soll, müssen Sie die Console anweisen, alle Klassen neu zu laden. Erzeugen Sie auch neue Objektinstanzen, so denn vorhanden, da sie sonst den alten Quelltext nutzen.

Sie können nun die Rails-Console mit der Eingabe von exit wieder schließen – allerdings nur, wenn Sie diesen kleinen wertvollen Helfer nicht vergessen und immer an sie denken, wenn Sie .

Informationen zur Sendung

Broadcast liefert nun eine Liste mit Videos. Was ist aber, wenn unsere Anwendung nur die Informationen zu einem konkreten Video benötigt? Dann ist die Zeit für Broadcast#get_video gekommen.

```
Beispiel 5-12 : Broadcast#get_video
def self.get_video(video_id)
  url = AOL_API_URL +
    "?method=truveo.videos.getVideos" +
    "&query=" + URI.escape('id:' + video_id.to_s) +
    "&appid=#{AOL_DEVELOPER_ID}"
  data = Broadcast.get_xml(url)

  video = data[:items][0]
  video[:http_code] = data[:http_code]
  return video
end
```

Um diese Methode zu nutzen, genügt die eindeutige ID, die AOL jedem gefundenen Video zuteilt und derer TVsendl bereits durch Broadcast#get_videos habhaft wurde. Auch in diesem Falle führen Sie die Funktion truveo.videos.getVideos der AOL API via REST aus. Diesmal besteht die Suchanfrage jedoch ausschließlich aus der ID des gewünschten Videos, welche durch id: als eine solche gekennzeichnet wird.

Das Ergebnis der Anfrage wird danach so umgebaut, dass der Rückgabewert das Video und den HTTP-Code der Anfrage enthält.

Abschließend fügen Sie dem Model Broadcast noch eine Methode hinzu, die ein Array möglicher Kanäle ausgibt, das später als Listbox in der View-Ebene in Erscheinung tritt.

Kanalliste

Auch Broadcast#channels sollte als Klassenmethode implementiert werden. Ansonsten hätten Sie nur Zugriff auf die Liste, wenn eine Instanz des Models existierte.

```
Beispiel 5-13 : Broadcast#channels liefert eine Liste der möglichen Quellen
```

```
def self.channels
  return [
    '[Alle Quellen]',
```

Beispiel 5-13 : Broadcast#channels liefert eine Liste der möglichen Quellen (Fortsetzung)

```
'YouTube',
'MYSPACE',
'Dailymotion',
'Google Video',
'IFILM',
'Veoh'
]
end
```

Natürlich gibt Broadcast#channels nicht alle Quellen aus, die AOL Video Search anzapft. Jedoch sind *YouTube*, *MySpace*, *Dailymotion*, *Google Video*, *IFILM* und *Veoh* die zurzeit ergiebigsten Quellen für unser Anliegen. Sicher kommen mit der Zeit noch einige interessante Video-Seiten dazu. Anzunehmen wäre beispielsweise die bessere Berücksichtigung für AOLs eigene YouTube-Kopie namens *Uncut*.

Zurück zur Gegenwart. In dieser sollten Sie Broadcast#get_videos noch eine Zeile Code spendieren, damit die Methode mit einer Channel-Angabe wie [Alle Quellen] auch zurecht kommt.

Beispiel 5-14 : Bei [Alle Quellen] entfällt die Channel-Angabe

```
def self.get_videos(search, channel, per_page, page)

  channels = '' if channel == '[Alle Quellen]'

  url = AOL_API_URL +
  ...
```

Das war's. Mehr müssen die beiden Models von TVsendr nicht können. Und das bedeutet, dass Sie sich mit dem Wissen über eine fertige Programmlogik, die ein Mix aus eigenem Quelltext und der von Ruby on Rails und AOL zur Verfügung gestellten Funktionalität ist, frohen Mutes an die Gestaltung der Benutzeroberfläche machen können.

Das Grundlayout

Sie haben durch den Scaffold-Code-Generator nicht nur einen funktionstüchtigen Controller erhalten, sondern auch einige Action-spezifische Views. Auf denen können Sie nun aufbauen. Ruby on Rails hat Ihnen sogar einen Layout-Datei inklusive CSS-Datei spendiert. Die RHTML-Datei heißt *stations.rhtml* – so, wie der dazugehörige Controller – und befindet sich in *app/views/layouts*. Öffnen Sie diese Datei, damit Sie Änderungen am optischen Grundgerüst Ihrer Anwendung vornehmen und einige weitere Dateien einbinden können.

Im Gegensatz zum Layout von Picsblog, wo jede Unterseite in das gleiche Grundgerüst gezwängt wurde, müssen wir hier berücksichtigen, dass unterschiedliche Seiten

auch unterschiedliche Strukturen haben werden. Beispielsweise werden wir gleich die View *edit.rhtml* so umbauen, dass mit ihr die Auswahl der Sendungen eines Senders stattfinden kann. Hingegen braucht *show.rhtml* eine ganz andere Struktur. Diese View wird einmal zuständig für die Präsentation der Sender sein. Das bedeutet: Die Festlegung von Spalten und deren Breiten etwa soll uns erst in den jeweiligen Views interessieren.

Beispiel 5-15 : stations.rhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
  <title>TVsendr: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
  <%= stylesheet_link_tag('layout') %>
  <%= javascript_include_tag(:defaults) %>
</head>
<body>
  <div id="page">
    <div id="header">
      <% if flash[:notice] %>
        <div id="notice"><p><%= flash[:notice] %></p></div>
      <% end %>
      <h1>TVsendr</h1>
    </div>
    <div id="content">
      <%= yield %>
    </div>
    <div id="footer">
      TVsendr basiert auf <%= link_to('Ruby on Rails', 'http://www.rubyonrails.org') %>
    </div>
  </div>
</body>
</html>
```

Fangen wir oben an. Zunächst hat sich der Titel des RHTML-Dokuments verändert. Der Scaffold-Generator hat automatisch den Namen des Controllers, welche zu diesem Layout gehört, in den title-Tag gesetzt. Jetzt befindet sich dort korrekt der Name der gesamten Applikation.

Viel wichtiger ist aber die Integration der CSS-Datei *layout.css*, die wir gleich noch erstellen. Außerdem neu mit dabei: Die Dateien der beiden JavaScript-Bibliotheken *Prototype* und *Scriptaculous*. Mit *javascript_include_tag(:defaults)* gelingt Ihnen das Einbinden der beiden Experten für Ajax, visuelle Effekte und neuartige Steuer-elemente mit einer Zeile.

Im body-Teil der Layoutdatei sind lediglich ein paar Ergänzungen hinzugekommen, die der Oberfläche von TVsendr Struktur geben sollen. Ein ähnliches Muster kennen Sie bereits von Picsblog. Im *Header* befindet sich der Titel der Seite, zudem wird `flash[:notice]` nun nur noch angezeigt, wenn es auch wirklich was zum Anzeigen gibt. Darüber hinaus wird `yield` im Content-Bereich die gerenderten Views einsetzen und in Footer weist ein kurzer Text auf den Kraftstoff hin, der in den Adern dieser Anwendung fließt.

layout.css

Mit `layout.css` wird das Grundgerüst komplettiert. Erstellen Sie die Datei in `app/public/stylesheets` und befüllen Sie selbige mit nachfolgendem Inhalt, den wir später noch ergänzen werden:

Beispiel 5-16 : layout.css

```
#page {  
    width: 950px;  
    margin: 0 auto;  
}  
  
#footer {  
    clear: both;  
}
```

Diese paar Zeilen CSS sorgen dafür, dass der Darstellungsbereich der Anwendung 950 Pixel breit ist und in die Mitte des Browsers gerückt werden soll. Der Footer wird schon einmal darauf vorbereitet, dass im Content-Bereich wild umher floating div-Elemente vorkommen können. Aber das kennen Sie bereits von Picsblog.

Im nächsten Schritt soll es um die Erweiterung von `edit.rhtml` gehen. Dabei fällt auch etwas für `layout.css` ab.

Auf dem Weg zum Programmchef

Der Aufruf der `edit`-Action des Stations-Controllers hat zur Folge, dass der Benutzer von TVsendr sich dem wohl komplexesten Teil der Anwendung gegenüber sieht:



Damit Sie auch sehen und nachvollziehen können, was hier gleich passiert, empfehle ich Ihnen, einen neuen Sender anzulegen, den Sie direkt nach seiner Erzeugung bearbeiten möchten.

Und so soll die View aussehen: Links oben soll der jetzt bereits existierende Bereich zum Bearbeiten der Senderdetails erscheinen, direkt darunter der per Drag and Drop sortierbare Programmplan. Auf der rechten oberen Seite soll der Benutzer die

Möglichkeit haben, sich Sendungen aus dem von AOL zur Verfügung gestellten Archiv herauszusuchen. Unter dem Bereich, in dem die Suchergebnisse gelistet werden, soll zudem Platz geschaffen werden für eine Vorschaufunktion. Sie soll die Entscheidung bezüglich der Aufnahme einer Sendung in das TV-Programm erleichtern. So viel zur Theorie. Hier kommt die Praxis:

Beispiel 5-17 : edit.rhtml

```
<div id="edit_left">
  <div id="station_details">
    <h1>Editing station</h1>
    <% form_tag :action => 'update', :id => @station do %>
      <%= render :partial => 'form' %>
      <%= submit_tag 'Edit' %>
    <% end %>
  </div>

  <div id="station_schedule">
    <h2>Programmplan</h2>
  </div>

  <%= link_to 'Show', :action => 'show', :id => @station %> |
  <%= link_to 'Back', :action => 'list' %>
</div>

<div id="edit_right">
  <div id="broadcast_search">
    <h2>Sendungen auswählen</h2>
  </div>

  <div id="broadcast_preview">
    <h2>Vorschau</h2>
  </div>
</div>
```

Wie Sie sehen, teilt sich die edit-View grundsätzlich in zwei Spalten, `edit_left` und `edit_right`. Damit die beiden, wie es sich für Spalten gehört, auch nebeneinander positioniert werden, muss `layout.css` ergänzt werden. Dabei wird auch die Breite festgelegt und zudem ein zehn Pixel breiter Abstand zwischen den beiden Spalten.

Beispiel 5-18 : Ergänzungen in layout.css

```
#edit_left {
  width: 400px;
  float:left;
}

#edit_right {
  width: 540px;
  float: right;
  margin-left: 10px;
}
```

Die Grundstruktur in *edit.rhtml* ist damit geschaffen. Kommen wir zu den Details und beginnen wir mit dem Erstellen des Suchformulars und der Anzeige der Suchergebnisse.

The screenshot shows a web page titled "TVsendr". On the left, there is a section titled "Editing station" containing fields for "Name" (with value "fghfh") and "Description" (with value "fhfghh"). Below these fields is an "Edit" button. To the right, there are two sections: "Sendungen auswählen" (Select programs) and "Vorschau" (Preview). Under "Programmplan", it says "Show | Back" and "TVsendr basiert auf Ruby on Rails.". The entire interface is contained within a light gray border.

Abbildung 5-5: Der *edit*-View nimmt Formen an

Brauchbare Sendungen gesucht

Mit der Implementation der Suchfunktionalität begegnen Sie zum ersten Mal Rails' Ajax-Funktionalitäten. Denn die Ergebnisse der Suche sollen nicht wie bei herkömmlichen Websites üblich erst nach einem kompletten Seiten-Reload verfügbar sein, sondern mittels Ajax in die bereits bestehende Webseite gesetzt werden.

Ruby on Rails stellt Ihnen zu diesem Zweck eine große Anzahl an Helpern zur Verfügung. Ein ganz wichtiger, wenn es um Ajax-basierte Formulare gehen soll, ist `form_remote_tag`. Das *remote* weist hier eindeutig darauf hin, dass Ajax dahinter steckt.

Ein ajaxifiziertes Formular

Dieser Helper funktioniert grundsätzlich wie sein Ajax-freier Verwandter `form_tag`. Er erzeugt ebenfalls ein HTML-form-Tag, sorgt aber durch automatisches Hinzufügen von JavaScript-Code dafür, dass der Formularinhalt nicht direkt per HTTP an den Server geschickt, sondern von JavaScript verarbeitet wird.

In der JavaScript-Abteilung Ihrer Anwendung wartet Prototype bereits auf die Eingaben und schickt sie per asynchroner Datenübertragung und mit Hilfe der XMLHttpRequest-Schnittstelle Ihres Browsers an den Server. Die genaue Adresse, hier eine bestimmte Action des Controllers, erhält XMLHttpRequest aus Ihren Angaben bei `form_remote_tag`.

Es ist wirklich wahr: Sollten Sie schon einmal Anwendungen auf Ajax-Basis entwickelt haben, kennen Sie vielleicht den Stress, den so ein Vorhaben mit sich bringt. Da muss zunächst eine Ajax-Engine implementiert werden, die muss irgendwie die Daten erhalten, das Formular muss daran gehindert werden, den herkömmlichen Weg zu beschreiten und nach dem Absenden die Seite neu zu laden. Dann muss das Ergebnis der Datenübertragung mit einer JavaScript-Callback-Funktion abgefangen werden und in die Webseite integriert werden, und so weiter und so weiter. Alles Schnee von gestern. Rails macht das zwar nicht anders, aber dafür selbstständig, komplett intern und mit Hilfe der ohnehin großartigen Prototype-Bibliothek. Sie bekommen davon fast gar nichts mit.

Daher: Sollten Sie beim Stichwort XMLHttpRequest nur Bahnhof verstanden haben, so ist das gar nicht schlimm. Denn da, wo ein Bahnhof ist, gibt es meistens auch Schienen und so lange Ruby darüber rollt, ist das kein Problem.

Es gibt einen wichtigen Unterschied zwischen `form_tag` und `form_remote_tag`: Bei `form_remote_tag` müssen Sie das Ziel des Formulars als Hash angeben und diesen dann in der Option `url` übergeben. Aber auch für `form_remote_tag` gilt: nicht den schließenden `form`-Tag am Ende des Formulars vergessen!

Beispiel 5-19 : edit.rhtml erhält ein Ajax-basiertes Formular

```
...
<h2>Sendungen auswählen</h2>
<%= form_remote_tag(:url => {:action => :search_broadcasts}) %>
  <%= text_field("broadcast_search", "search") %>
  <%= select("broadcast_search", "channel", Broadcast.channels) %>
  <%= submit_tag("Suchen") %>
</form>
</div>
...
```

Damit das Formular auch Formular ist, muss es natürlich noch Eingabeelemente bekommen. Hier handelt es sich um ein Ihnen bereits bekanntes Eingabefeld, das durch `text_field` generiert wird, sowie um ein `select`-Element, das durch einen gleichnamigen Helper zu HTML wird. Den Inhalt der Liste erhält `select` durch ein Array, welches Sie im dritten Parameter des Helpers angeben müssen. Hier kommt unsere eben noch implementierte Methode `Broadcast#channels` zum Einsatz. Da sie bereits ein Array liefert, muss der Controller gar nicht erst belastet werden. Und so können Sie auch direkt in einer View auf Inhalte der Model-Schicht zugreifen.

Der Blick in den Quelltext, den all die Helper erzeugen, lohnt sich. Sie sehen hierbei vor allem, wie Ruby on Rails dank `form_remote_tag` ein normales HTML-Formular Ajax-basiert gestaltet und wie der `select`-Helper für jeden Kanal ein `option`-Element erzeugt hat.

Beispiel 5-20 : Das Ajax-Formular in HTML und JavaScript

```
...
<div id="broadcast_search">
  <h2>Sendungen auswählen</h2>

  <form action="/stations/search_broadcasts" method="post" onsubmit="new Ajax.
Request('/stations/search_broadcasts', {asynchronous:true, evalScripts:
true, parameters:Form.serialize(this)}); return false;">
    <input id="broadcast_search_search" name="broadcast_
search[search]" size="30" type="text" />
    <select id="broadcast_search_channel" name="broadcast_search[channel]">
      <option value="[Alle Quellen]">[Alle Quellen]</option>
      <option value="YouTube">YouTube</option>
      <option value="MYSPACE">MYSPACE</option>
      <option value="Dailymotion">Dailymotion</option>
      <option value="Google Video">Google Video</option>
      <option value="IFILM">IFILM</option>
      <option value="Veoh">Veoh</option></select>
      <input name="commit" type="submit" value="Suchen" />
    </form>
  </div>
...
```

Im eben verwendeten `form_remote_tag` haben wir vereinbart, dass sich die Action `search_broadcasts` um die Formulareingaben kümmern soll. Bei der Implementation dieser Methode können Sie so herangehen, als wäre sie gar nicht durch ein ajaxifiziertes Formular aufgerufen worden.

Beispiel 5-21 : StationsController#search_broadcasts

```
def search_broadcasts
  @search = params[:broadcast_search][:search]
  @channel = params[:broadcast_search][:channel]
  if params[:broadcast_search][:current_page]
    @current_page = params[:broadcast_search][:current_page].to_i
  else
    @current_page = 1
  end

  @videos = Broadcast.get_videos(
    @search,
    @channel,
    20,
    @current_page
  )
end
```

In `search_broadcast` steht bislang nur längst Bekanntes. Über die Parameter, die die Action aus dem Formular über das Objekt `broadcast_search` erhält, werden Objekte erzeugt, mit denen `Broadcast#get_videos` gefüttert werden kann. Das Ergebnis der

Abfrage wird in @videos gespeichert. Sollte keine Angabe zur momentan angezeigten Seite der Suchergebnisse vorliegen, wird für current_page der Wert 1 angenommen, was der ersten Seite entspricht.

Viel interessanter ist die Antwort auf die Frage, wie denn das Suchergebnis in die Webseite eingesetzt werden soll. Und, um die Beantwortung noch schwieriger zu machen, definieren wir gleich mehrere Stellen in *edit.rhtml*, die als Ergebnis der Suche verändert werden sollen.

Dabei handelt es sich um einen Bereich, der Informationen zu einer fehlgeschlagenen Suche oder aber die Menge der gefundenen Video anzeigen soll. Ein weiterer Bereich listet die Suchergebnisse, ein dritter soll abhängig vom Suchergebnis eine Navigation bereitstellen, mit deren Hilfe man durch die einzelnen Seiten des Suchergebnisses durchforsten kann. Wechseln Sie also wieder zu *edit.rhtml* und ergänzen Sie die View um folgende Zeilen, direkt unter dem Ajax-Formular.

Beispiel 5-22 : Drei Bereiche sollen das Suchergebnis in edit.rhtml darstellen

```
...
</form>

<div id="broadcast_search_results_count"></div>
<div id="broadcast_search_results" style="display: none"></div>
<div id="broadcast_search_results_navigation" style="display: none"></div>
</div>
...

```

Wie Sie sicher erkennen, sind broadcast_search_results und broadcast_search_results_navigation zunächst einmal ausgeblendet. Sie sollen nur bei Bedarf eingeblendet werden. Ja, auch das muss StationsController#search_broadcasts managen.

RJS – JavaScript auf Ruby-Art

Sollten Sie bereits Erfahrungen mit JavaScript und DOM haben, können Sie sicher schon ahnen, auf was das alles hier hinaus läuft. Falls nicht, verrate ich es Ihnen: In der Action search_broadcasts werden wir nun (mit Ruby) JavaScript-Code schreiben, der als Reaktion auf die Suchanfrage an den Browser geschickt wird. Dort angekommen verrichtet er clientseitig all die Arbeiten, die ich eben angekündigt habe: Die Anzeige des Suchergebnisses inklusive Veränderungen des Inhalts an gleich drei Positionen in der View ohne Seite-Reload und das Ein- beziehungsweise Ausblenden benötigter oder nicht benötigter Bereiche der View.

Beispiel 5-23 : search_broadcasts, zweiter Teil.

```
...
20,
@current_page
```

Beispiel 5-23 : search_broadcasts, zweiter Teil. (Fortsetzung)

```
)  
  
    render(:update) { |page|  
  
      if @videos[:http_code] == 200  
        page.replace_html(  
          'broadcast_search_results_count',  
          :inline => "<p>Es wurden <b><%= @videos[:count] %></b>  
Sendung<%= 'en' unless @videos[:count] == 1 %> gefunden</p>"  
        )  
      else  
        page.replace_html(  
          'broadcast_search_results_count',  
          :inline => "<p>Es trat ein Fehler bei der Daten&uuml;bertragung auf.</p>"  
        )  
      end  
  
      if @videos[:count] > 0  
        page.replace_html(  
          'broadcast_search_results',  
          :partial => 'broadcast_search_result',  
          :collection => @videos[:items]  
        )  
        page.replace_html(  
          'broadcast_search_results_navigation',  
          :partial => 'broadcast_search_results_navigation',  
          :locals => {  
            :videos => @videos,  
            :search => @search,  
            :channel => @channel,  
            :current_page => @current_page  
          }  
        )  
        page.show('broadcast_search_results')  
        page.show('broadcast_search_results_navigation')  
        page.visual_effect(  
          :highlight,  
          'broadcast_search_results'  
        )  
      else  
        page.hide('broadcast_search_results')  
        page.hide('broadcast_search_results_navigation')  
      end  
    }  
  end
```

Mit `render(:update)` legen Sie zunächst fest, dass Sie die momentan im Browser angezeigte Webseite nicht neu laden möchten, sondern lieber aktualisieren. Was dabei geschehen soll, definieren Sie innerhalb eines Blocks, den Sie der `render`-Methode anhängen. Netterweise wird Ihnen bei dieser Gelegenheit in der Blockvariable, hier `page`, eine Referenz auf die zu aktualisierende Seite übergeben.

Mit Hilfe diverser Methoden, die alle JavaScript generieren, können Sie nun in Ruby JavaScript-Befehle schreiben. Ein sehr beliebter *JavaScript-Generator* ist `replace_html`. Bei dieser Methode geben Sie an, dass der Inhalt eines bestimmten HTML-Elements der Webseite gegen einen anderen ausgetauscht werden soll. Der erste Parameter von `replace_html` muss die ID des betreffenden Elements beinhalten.

Der Rest stimmt mit den Parametern überein, die Sie bereits von der `render`-Methode kennen. So können Sie beispielsweise verfügen, dass ein gerendertes Partial den Inhalt des im ersten Parameter angegebenen Elements ersetzen soll.

Zunächst überprüft die Methode `search_broadcasts` aber, ob die Anfrage an die AOL API erfolgreich verlief. In diesem Fall kann in der Inhalt des div-Elements mit der id `broadcast_search_results_count` gegen eine entsprechende Information über die Anzahl gefundener Videos ausgetauscht werden. Andernfalls findet dort die traurige Botschaft über die missglückte Suche Platz.

Beide Botschaften sind in HTML verfasst und werden als *Inline-HTML* in die Seite gesetzt. Die Verwendung von Inline-HTML ist dann nützlich, wenn es sich einfach nicht lohnt, wegen wenigen Worten ein eigenständiges Partial zu schreiben.



Gehen Sie sparsam mit Inline-HTML um. Es gibt Menschen, die meinen, dass das gegen das MVC-Pattern verstößt, da etwas, was ins View gehört im Controller vorkommt. Unter uns: So ganz unrecht haben die nicht. Letztlich sind aber Sie der Entwickler – und Sie entscheiden, was wie wo hin muss.

Im weiteren Verlauf der Methode werden zwei Partials gerendert und in die beiden div-Elemente `broadcasts_search_results` beziehungsweise `broadcasts_search_results_navigation` eingesetzt. Wie bereits erwähnt, erfolgt die Parameterübergabe so, wie Sie es von der `render`-Methode kennen. Ein Array voller Videos beispielsweise wird über `collection` an das Partial übergeben. Und dieses wird anschließend für jedes einzelne Element gerendert.

Neu für Sie könnte die Übergabe einzelner Variablen an ein Partial sein. Dies ist beim Partial `broadcast_search_results_navigation` der Fall. Die hier übergebenen *Locals* stehen innerhalb des Partials als lokale Variablen namens `videos`, `search`, `channel` und `current_page` zur Verfügung. Wozu die vier Objekte verwendet werden, sehen Sie gleich.

Zunächst möchte ich Sie noch auf die beiden Methoden `show` und `hide` hinweisen. Sie leisten genau das, was ihre Bezeichner vermuten lassen: Sie verstecken und zeigen Elemente der Webseite an. Auch hier erfolgt die eindeutige Zuordnung über die ID des Elements.

Und dann gibt es da noch den JavaScript-Generator namens `visual_effect`. Hier wird JavaScript generiert, mit dem die Bibliothek *Scriptaculous* angesprochen wird. Visuelle Effekte sind wichtige Elemente in der modernen Programmiererei mit Ajax und sind damit alles andere als pure Spielerei und Effekthascherei.

Oft verpasst ein Benutzer Änderungen auf der Oberfläche einer Website, weil vieles mit Ajax nebenbei und im Hintergrund passiert. Wenn der veränderte Bereich aber durch ein kurzes Aufblitzen auf sich aufmerksam macht, dann merkt das der Benutzer. Und genau das passiert hier. Der Effekt `highlight` wird hier auf das Element mit der ID `broadcast_search_results` angewendet. Wann immer also neue Suchergebnisse eingetroffen sind, leuchten selbige kurz auf.



Es gibt neben `highlight` noch eine Menge weiterer Effekte in der *Scriptaculous*-Bibliothek. Mehr dazu finden Sie auf der Website des Projekts, <http://script.aculo.us>. Hier gibt es auch eine Übersicht über alle standardmäßig enthaltenen Effekte inklusive Live-Demo: <http://wiki.script.aculo.us/scriptaculous/show/CombinationEffectsDemo>. Weitere Effekte lernen Sie auch im Verlauf dieses Kapitels noch kennen.

Bevor Sie das Ajax-Formular, das JavaScript-gesteuerte Einsetzen neuer Inhalte in die Oberfläche von TVsendr und den spektakulären visuellen Effekt genießen können, müssen noch fix zwei Partials geschrieben werden.

Darstellung eines Suchergebnisses

Das erste, `broadcast_search_result`, legt fest, wie jedes einzelne Suchergebnis auszusehen hat. Das nötige Datenmaterial erhält das Partial über das Array `@videos[:items]`.



Hier heißt es besonders aufpassen. Normalerweise sind Sie gewöhnt, dass Sie innerhalb eines Partials über eine lokale Variable, die genau so heißt, wie das Partial selbst, auf ein Element der Collection zugreifen können. Das ist hier anders! Jedes Element von `@videos[:items]` ist ein Hash, dessen Schlüssel-Wert-Paare Informationen über ein Video enthalten. Handelt es sich bei den Einzelementen der Collection um Hashes, so werden die Hash-Schlüssel automatisch zu lokalen Variablen, die jeweils den Wert des dazu gehörigen Hash-Wertes annehmen. Die lokale Variable, die den Partial-Namen trägt, können Sie hier nicht nutzen.

Die Rails-Entwickler sind sich noch nicht ganz einig, ob das ein Feature oder ein Fehler des Frameworks sein soll. Ich persönlich tendiere zu Fehler, schließlich ist das eben beschriebene Verhalten eine Ausnahme, deren Grund schwer zu erkennen ist.

Legen Sie in `app/views/stations` eine neue Datei namens `_broadcast_search_result.rhtml` an. Vergessen Sie dabei den führenden Unterstrich im Dateinamen nicht!

Beispiel 5-24 : `_broadcast_search_result.rhtml`

```
<%= image_tag(thumbnail_url, :style => 'float:left') %>
<h3><%= h(title) %></h3>
<p>
<%= link_to_remote(
  'Vorschau',
  :url => {
    :action => :preview_broadcast,
    :video_id => video_id
  },
  :update => :broadcast_preview_player
) %>
| <%= link_to_remote(
  'In Programm aufnehmen',
  :url => {
    :action => :apply_broadcast,
    :video_id => video_id
  }
) %>
<%= " | (mit Player)" if video_html %>
</p>
<p><%= h(description) %></p>
<div style="clear: both"></div>
```

Das Partial beginnt mit der Anzeige des Thumbnails. Das ist Aufgabe des Helpers `image_tag`. Per `style`-Attribut wird das Bild so eingestellt, dass es von Text umflossen werden kann. Zum Beispiel vom Titel des Videos, der dank `float: left` nicht unter dem Thumbnail erscheint, sondern rechts daneben.

Es folgen zwei Links, auf die ich gleich noch genauer eingehen werde. Vorab nur soviel: Der erste Link initiiert die Vorschau im unteren rechten Bereich der Seite. Der zweite Link startet die Aufnahme des des jeweiligen Videos in den Programmplan des Senders.

Anschließend folgt noch ein Hinweis, der darüber informiert, ob ein Video einen eigenen Player mitbringt. Ist dies der Fall, so kann das Video innerhalb von TVsendr abgespielt werden. Das einzige Kriterium dafür: in `video_html` muss etwas enthalten sein. Ist dies der Fall, erscheint der Hinweis.

Durch Suchergebnisse navigieren

Das zweite Partial, was zum Anzeigen der Suchergebnisse gebraucht wird, speichern Sie als `_broadcast_search_results_navigation.rhtml` ab. Es rendert die Navigationsleiste unterhalb der Suchergebnisanzeige.

Beispiel 5-25 : _broadcast_search_results_navigation.rhtml

```

<p>
<% if current_page > 1 %>

<%= link_to_remote(
  '[<<]',
  :url => {
    :action => 'search_broadcasts',
    'broadcast_search[search]' => search,
    'broadcast_search[channel]' => channel,
    'broadcast_search[current_page]' => 1
  }
) %>

<%= link_to_remote(
  '[<]',
  :url => {
    :action => 'search_broadcasts',
    'broadcast_search[search]' => search,
    'broadcast_search[channel]' => channel,
    'broadcast_search[current_page]' => current_page - 1
  }
) %>

<% end %>

<%= " Seite <b>#{current_page}</b> von <b>#{videos[:page_count]}</b> " %>

<% if current_page < videos[:page_count] %>

<%= link_to_remote(
  '[>]',
  :url => {
    :action => 'search_broadcasts',
    'broadcast_search[search]' => search,
    'broadcast_search[channel]' => channel,
    'broadcast_search[current_page]' => current_page + 1
  }
) %>

<%= link_to_remote(
  '[>>]',
  :url => {
    :action => 'search_broadcasts',
    'broadcast_search[search]' => search,
    'broadcast_search[channel]' => channel,
    'broadcast_search[current_page]' => videos[:page_count]
  }
) %>

<% end %>
</p>
```

Je nach Sachlage zeigt die Navigation Links zur *ersten*, zur *vorhergehenden*, zur *nächsten* und zur *letzten* Seite der Suchergebnisse an. Alle Navigationslinks haben das gleiche Ziel: StationsController#search_broadcasts.

Wie Sie wissen, haben wir diese Methode eben so geschrieben, dass Sie auf das Absenden des Suchformulars reagieren kann. Suchergebnisse werden stets als Hash-Objekt an eine Action übergeben. Damit search_broadcast mit den Links in *_broadcast_search_results_navigation.rhtml* etwas anfangen kann, werden die nötigen Parameter so verschickt, als würden sie vom Suchformular stammen. Also statt search wird der Parameter broadcast_search[search] verwendet.

Im Gegensatz zum Absenden des Suchformulars wird bei den Navigationslinks allerdings current_page inklusive numerischen Wert mitgesendet. Klar, mit dieser Zahl wird bestimmt, welche Seiten als nächstes angezeigt werden soll. Und das macht eine Navigation schließlich erst zu einer Navigation.

Jetzt ist der spannende Augenblick gekommen, in dem TVsendr erstmals einen Hauch seines Könnens preisgeben darf. Suchen Sie doch einmal nach Videos. Geben Sie einen ertragreichen Suchbegriff ein und betätigen Sie den Suchen-Button oder die Enter-Taste. Sogleich müssten Sie Thumbnails mit Titel und Beschreibung sehen, die auf einem kurz aufflackernden Hintergrund liegen. Mehr aber leider noch nicht. Eine Vorschau ist beispielsweise nicht enthalten. Noch nicht.

Programmvorschau

Im Partial *_broadcast_search_result* haben Sie jedem Suchergebnis zwei Links spendiert, mit denen ein Video angesehen oder in den Programmplan aufgenommen werden kann. Eine kleine Rückschau:

```
...
<%= link_to_remote(
  'Vorschau',
  :url => {
    :action => :preview_broadcast,
    :video_id => video_id
  },
  :update => :broadcast_preview_player
) %>
| <%= link_to_remote(
  'In Programm aufnehmen',
  :url => {
    :action => :apply_broadcast,
    :video_id => video_id
  }
) %>
...
...
```

Zur Methode `link_to_remote` ist zunächst einmal folgendes zu sagen: Sie können sie genau so benutzen wie `form_remote_tag`, nur eben ohne Formular. Die Parameter sind bei beiden Methoden gleich. Und deshalb gibt es auch die `update`-Option sowohl bei `link_to_remote` als auch `form_remote_tag`. Mit dieser Option ist es möglich, direkt in der Helper-Methode festzulegen, an welcher Stelle der Website das gerenderte Ergebnis der dazugehörigen Action platziert werden soll. Die Anfrage an eben diese Action wird, wie auch hier der kleine Zusatz *remote* erahnen lässt, via Ajax durchgeführt.

Sie erinnern sich? Eben haben Sie beispielsweise durch `replace_html` auf Controller-Seite festgelegt, wo der Ort des Geschehens sein soll. Jetzt lernen Sie mit der `update`-Option eine Alternative kennen, die zudem auf Controller-seitiges RJS verzichtet. Das bedeutet, dass Sie die Action nahezu wie gewohnt verfassen können.

Bevor Sie dies tun, müssen Sie allerdings `edit.rhtml` um den Bereich erweitern, in dem das Video inklusive Player erscheinen soll. In der `update`-Option ist er bereits als `broadcast_preview_player` benannt.

Beispiel 5-26 : Platz für den Vorschau-Player

```
<div id="broadcast_search_results_count"></div>
<div id="broadcast_search_results" style="display: none"></div>
<div id="broadcast_search_results_navigation" style="display: none"></div>
</div>

<div id="broadcast_preview">
  <h2>Vorschau</h2>
  <div id="broadcast_preview_player">
    <p>Bitte w&uuml;hlen Sie eine Sendung aus.</p>
  </div>
</div>
</div>
```

Hier haben wir die Gelegenheit und den Platz genutzt, um den TVsendl-Benutzer zu informieren, warum in diesem Bereich noch nichts zu sehen ist. Sobald ein Video zur Vorschau ausgewählt ist, macht der Schriftzug dem Player natürlich Platz.

Der Link, der die Vorschau verursacht, enthält den Namen der Action, die das Video abspielen soll und darüber hinaus die Video-ID des Filmchens. Auf Basis der Video-ID, die im Controller als Parameter `video_id` vorliegt, werden in der Action `preview_broadcast` mit Hilfe von `Broadcast#get_video` alle erforderlichen Daten von AOL abgerufen.

Beispiel 5-27 : StationsController#preview_broadcast

```
def preview_broadcast
  @video = Broadcast.get_video(params[:video_id])
  render :action => :view_broadcast, :layout => false
end
```

Nachdem die Daten in @video abgelegt sind, kommt die dazugehörige View ins Spiel. Wie gesagt, auch wenn hier Ajax maßgeblich beteiligt ist, verhält sich die Action wie bei herkömmlichen HTTP-Requests mit Seiten-Reload. Bis auf eine Ausnahme: Die Option layout.

Im vierten Kapitel haben Sie erfahren, wie Sie ein Layout festlegen, dass grundsätzlich für alle Actions gilt, es sei denn, sie stehen in einer Ausnahmeliste. Hier eine kleine Erinnerungshilfe aus PostsController:

```
layout('standard', :except => :list_newest)
```

Nun sehen Sie, wie man es auch machen kann: Wenn Sie Ruby on Rails auffordern, eine bestimmte View zu rendern, so können Sie durch die Option layout verhindern, dass dabei das Standard-Layout des Controllers mitgerendert wird.

Das Layout darf deshalb in preview_broadcast nicht mitgerendert werden, weil es ja bereits im Browser existiert. Die Action preview_broadcast soll also nur den Teil liefern, der erneuert werden soll. Durch die update-Option haben Sie zudem festgelegt, wo dieser Bereich innerhalb der Webseite steckt.



Partials werden grundsätzlich ohne Layout gerendert. Bei ihnen entfällt also das explizite Abschalten des Layouts.

Sicher ist Ihnen aufgefallen, dass hier *view_broadcast.rhtml* und nicht *preview_broadcast.rhtml*, wie es die Action vermuten lassen würde, gerendert werden soll. Das liegt in der weisen Voraussicht begründet, dass Sie diese View später noch einmal nutzen werden. Nämlich dann, wenn das komplette Fernsehprogramm abgespielt werden soll. Wir betrieben hier quasi View-Sharing.

Aus diesem Grunde enthält *view_broadcast.rhtml*, wie Sie gleich sehen werden, auch eine kleine Besonderheit. Neben dem Titel, der Beschreibung und dem Video soll der Vorschaubereich auch einen Link beinhalten, der die Sendung in den Programmplan aufnimmt. Einen solchen Link kennen Sie bereits von den Suchergebnissen.

Selbstverständlich soll dieser Link nur dann angezeigt werden, wenn *view_broadcast.rhtml* als Folge der Action preview_broadcast gerendert wird. Im Zuschauermodus wäre er fehl am Platze.

Innerhalb einer View können Sie über controller.action_name heraus finden, welche Action die View aufgerufen hat.

Beispiel 5-28 : view_broadcast.rhtml

```
<% if @video[:title] %>  
<h3><%= h(@video[:title]) %></h3>
```

Beispiel 5-28 : view_broadcast.rhtml (Fortsetzung)

```
<% if @video[:video_html] %>
  <%= @video[:video_html] %>
<% else %>
  <p>Leider kann das Video hier nicht angezeigt werden. Klicken Sie <%= link_
to('hier', @video[:source_url],
:target => '_blank') %> um es trotzdem anzusehen.</p>
<% end %>

<p><b>Inhalt: </b><%= h(@video[:description]) %></p>

<% if controller.action_name == 'preview_broadcast' %>
  <p><%= link_to_remote(
    'In Programm aufnehmen',
    :url => {
      :action => :apply_broadcast,
      :video_id => @video[:video_id]
    }
  ) %></p>
<% end %>

<% else %>
  <%if @video[:http_code] == 200 %>
    <p>Diese gew&auml;hlte Sendung ist nicht mehr im Angebot.</p>
  <%else %>
    <p>Es trat ein Fehler bei der Daten&uuml;bertragung auf.</p>
  <% end %>
<% end %>
```

Diese View berücksichtigt, ob beim Abfragen der Video-Informationen ein Übertragungsfehler aufgetreten ist. In diesem Falle wird eine entsprechende Meldung angezeigt. Darüber hinaus trägt sie der Tatsache Rechnung, dass nicht alle von AOL gefundenen Videos in externen Seiten wiedergegeben werden dürfen. Sollte dies der Fall sein, stellt die View zumindest den Link zu der Seite zur Verfügung, unter der das Video dennoch in Augen- und Ohrenschein genommen werden kann.

Ich schlage vor, Sie suchen sich ein paar Videos aus, die einen Player mitbringen – eine entsprechende Notiz ist bekanntlich bei jedem Suchergebnis zu sehen – und bewundern Ihre eben implementierte Vorschau-Funktion. Ja, jetzt ist der Moment erreicht, in dem zum ersten Mal bewegte Bilder zu sehen sind. Das ist wie damals. 1928 auf der Berliner Funkausstellung. Vielleicht erinnern Sie sich ja noch?

Aber die Zeiten haben sich geändert. Heute können Sie Ihr Programm selbst zusammenstellen. Damit dieser Satz auch für TVsendlr gilt, implementieren Sie nun die Action `apply_broadcast`.

Eine Sendung ins Programm nehmen

Wie Sie wissen, soll der Programmplan von TVsendr kein starres Gebilde sein. Sie sollen per Drag and Drop entscheiden können, welcher Programmpunkt an welcher Stelle des Programms stehen soll. Damit das gelingt, müssen wir einige Vorbereitungen treffen.

Im weiteren Verlauf werden Sie ein *Sortable* nutzen. Das ist ein sehr nützliches Steuerelement aus Script.aculo.us' Repertoire. Ein Sortable macht eine Gruppe von Elementen mit der Maus sortierbar.

Sortables sind standardmäßig auf li-Elemente spezialisiert, die sich naturgemäß innerhalb eines ul- oder ol-Elements befinden. Das passt uns ganz gut, schließlich ist ein Programmplan ja irgendwie auch eine Liste. Erhält das umgebende ul-Element zudem ein ID-Attribut, steht dem Einsatz eines Sortables fast nichts mehr im Wege und schon in Kürze können Sie die einzelnen Programmpunkte über den Bildschirm schieben. Die li-Elemente werden durch die Anwendung generiert, das dazugehörige ul-Element werden wir fest in edit.rhtml integrieren.

Und noch etwas muss im Zusammenhang mit dem Sortable in *edit.rhtml* getan werden. Aus Gründen, die Sie gleich erfahren, müssen wir innerhalb des Dokuments Platz für JavaScript-Code reservieren. Diesen JavaScript-Code müssen Sie natürlich nicht selbst schreiben. Er wird von Ruby on Rails generiert und sorgt dafür, dass die li-Elementen auch wirklich sortierbar werden. Lediglich den Platz zu definieren bleibt Ihnen überlassen.

Beispiel 5-29 : Vorbereitungen für den sortierbaren Programmplan

```
...
<div id="station_schedule">
  <h2>Programmplan</h2>
  <ul id="station_schedule_list">
  </ul>
  <div id="station_schedule_list_js">
  </div>
</div>

<%= link_to 'Show', :action => 'show', :id => @station %> |
...
...
```

Die Action `apply_broadcast` beinhaltet zwei Teile. Im ersten wird dafür gesorgt, dass die neue Sendung in der Datenbanktabelle gespeichert wird. Der zweite erledigt die Darstellung der Sendung im Sendeplan.

Vielleicht erinnern Sie sich noch, wie Sie in Picsblog Kommentare gespeichert haben: Zunächst wurde der Beitrag ermittelt, zu dem der Kommentar gehört, dann wurde der Kommentar mittels << dem Beitrag zugeordnet. Dank der Assoziationen kümmerte sich Rails um den ganzen internen Datenbankzinober, der dafür ansonsten nötig gewesen wäre. So funktioniert das auch hier mit neuen Programmpunkten.

Allerdings gibt es ein Problem. Die Action `apply_broadcast` weiß nicht, welcher Sender gerade bearbeitet wird. Sie kann zwar einen neuen Programmfpunkt in der Datenbank speichern, aber nicht zuordnen. Das liegt daran, dass seit dem Aufruf der `edit`-Action im Browser diverse Ajax-Requests an den Server gingen, bei denen die ID des Senders nie mitgesendet wurde, so dass sie nun auch nicht `apply_broadcast` übermittelt werden kann.

Um dieses Manko zu beheben, können Sie beim Aufrufen der `edit`-Action die ID des Senders als Session-Variable speichern. Sie steht dann auch nach diversen Anfragen an den Server zur Verfügung, denn das ist die Natur von Sessions. Bevor Sie also `apply_broadcast` implementieren, nehmen Sie noch eine kleine Veränderung an `StationsController#edit` vor. Schließlich ist das die einzige Action während der Programmplanung, welche die ID des Senders erhält.

Beispiel 5-30 : StationsController#edit

```
def edit
  @station = Station.find(params[:id])
  session[:station_id] = params[:id]
end
```

In `apply_broadcast` können Sie nun `session[:station_id]` auslesen, um zu erfahren, welcher Sender die neue Sendung erhalten soll.

Beispiel 5-31 : StationsController#apply_broadcast, 1. Teil

```
def apply_broadcast
  # Sender
  @station = Station.find(session[:station_id])

  # Sendung
  @video = Broadcast.get_video(params[:video_id])
  @broadcast = Broadcast.new
  @broadcast.title = Broadcast.get_video(params[:video_id])[:title]
  @broadcast.station_id = session[:station_id]
  @broadcast.video_id = params[:video_id]
  @broadcast.position = 9999

  # Sendung dem Sender hinzufügen
  @station.broadcasts << @broadcast
  ...

```

Wie Sie sehen, wird hier eine Instanz des `Broadcast`-Models gebildet, die den Titel der Sendung erhält, wiederum mit Hilfe einer kleinen Nachfrage an AOL, die ID von Sender und Video und eine recht hohe Positionsangabe. Die liegt deshalb bei 9999, weil wir bestimmt haben, dass sich die Reihenfolge der Sendungen eines Senders aus deren Positionswert herleiten lässt. Mit 9999 stellen Sie sicher, dass die neue Sendung zunächst definitiv ans Ende des Programmplans gesetzt wird.

Beispiel 5-32 : StationsController#apply_broadcast, 2. Teil

```
...
# Veränderungen anzeigen
render(:update) { |page|
  page.insert_html(
    :bottom,
    'station_schedule_list',
    :partial => 'station_schedule_item',
    :object => @broadcast
  )

  page.visual_effect(
    :pulsate,
    "item_#{@broadcast.id}"
  )

  page.replace_html(
    :station_schedule_list_js,
    sortable_element(
      :station_schedule_list,
      :url => { :action => :order_schedule}
    )
  )
}
```

Was render(:update) bewirkt, wissen Sie bereits. Neu ist die Methode insert_html. Im Gegensatz zu replace_html ersetzt sie nicht den Inhalt eines HTML-Elements, sondern ergänzt ihn. Wo die Ergänzung statt finden soll, ob ober- (top) oder unterhalb (bottom) des bereits bestehenden Inhalts oder ob direkt vor (before) oder direkt nach (after) dem HTML-Element, legen Sie im ersten Parameter fest. Der Rest von insert_html funktioniert wie replace_html. Es folgt also das betreffende Element und dann das, was gerendert werden soll.

In unserem Fall soll das gerenderte Partial station_schedule_item der Liste stations_schedule_list hinzugefügt werden. Durch die Verwendung eines optischen Effekts, hier pulsate, macht der neu hinzugefügte Programmfpunkt auf sich aufmerksam.

Schließlich wird der oben angesprochene JavaScript-Code für das Sortable erzeugt. Dieser Vorgang ist immer dann nötig, wenn ein neues Element in die Programmliste station_schedule_list aufgenommen wird. Ansonsten könnten Sie das neue Element im Gegensatz zu den bestehenden nicht per Drag and Drop sortieren.

Der aus Basis der erweiterten Liste generierte Code wird in den extra dafür geschaffenen Bereich station_schedule_list_js platziert. Während dieses Vorgang wird der JavaScript zugleich durch den Browser ausgeführt, so dass die Programmliste direkt nach dem Hinzufügen der neuen Sendung (wieder) komplett sortierbar ist.

Die Methode `sortable_element` erwartet als Parameter das Element, welches die zu sortierenden enthält. In unserem Falle handelt es sich um `station_schedule_list`. Über die Option `url` geben Sie eine Action an, die durch das Sortable und per Ajax immer dann aufgerufen wird, wenn sich etwas an der Reihenfolge der einzelnen Elemente des Sortables etwas geändert hat. Die Action `order_schedule` erhält dabei automatisch Informationen darüber und kann die Positionsdaten in die Datenbank schreiben.

Bevor Sie Ihren Programmplan sehen und sortieren können, müssen Sie allerdings noch ein Partial verfassen. Es beinhaltet ein neues `li`-Element, was per `insert_html` eingefügt wird, und heißt `station_schedule_item`. Erstellen Sie eine entsprechende, mit Unterstrich beginnende und mit `.rhtml` endende Datei in `app/views/stations`.

Wenn Sie noch einmal den Code in `apply_broadcast` betrachten, werden Sie feststellen, dass bei `insert_html` mit `object` eine bislang unbekannte Option enthalten ist. Mit `object` können Sie wie mit `collection` arbeiten. Allerdings wird das Partial nur einmal und nur für das angegebene Objekt gerendert. Die Konvention, dass dieses Objekt innerhalb des Partials über eine lokale Variable verfügbar ist, die so heißt, wie das Partial selbst, gilt auch hier.

Beispiel 5-33 : _station_schedule.item.rhtml

```
<% item = station_schedule_item %>
<li id="item_<%= item.id %>">
  <h3><%= item.title %></h3>
  <p><%= link_to_remote(
    'Vorschau',
    :url => {
      :action => :preview_broadcast,
      :video_id => item.video_id
    },
    :update => :broadcast_preview_player
  ) %>
  | <%= link_to_remote(
    'L&ouml;schen',
    :url => {:action => :delete_broadcast, :id => item.id}
  ) %></li>
```

Da dieses Partial mit `station_schedule_item` einen recht langen Namen hat und das einen recht langen Bezeichner der lokalen Variable zur Folge hat, wird der Inhalt von `station_schedule_item` am Anfang des Templates an die handlichere Variable `item` übergeben.

Anschließend wird ein `li`-Element mit dem Titel der Sendung und zwei Links als Inhalt erzeugt. Ganz wichtig hierbei: Jedes `li`-Element muss ein ID-Attribut besitzen, das nach der Regel `<irgendeinString>_<ID_des_Datensatzes>` gebildet wird. Nur so kann Ihre Liste als Sortable verwendet werden. Ein ID-Attribut, das nur die Datensatz-ID als Wert hat, ist nicht zulässig.

Das Partial enthält zudem zwei Links. Den einen kennen Sie schon, er startet die Vorschau des Videos. Der andere sorgt sich um die Entfernung einer Sendung aus dem Programmplan. Um StationsController#delete_broadcast kümmern wir uns gleich, zunächst sollten Sie aber mal den Programmplan mit Sendungen füllen.

Das sollte schon ganz brauchbar funktionieren. Es gibt jedoch ein Problem: Beim Neuladen der Seite sind alle Einträge wieder verschwunden. Zumindest optisch, denn wir haben bislang nichts unternommen, was die Liste des Programmplans beim Aufrufen der edit-Action mit den bereits bestehenden Daten füllt. Das lässt sich aber durch ein paar wenige Ergänzungen in *edit.rhtml* ändern.

Hierbei kommt das eben entwickelte Partial zum Einsatz, denn mit seiner Hilfe können Sie natürlich auch bestehende Sendungen rendern.

Beispiel 5-34 : edit.rhtml

```
...
<div id="station_schedule">
  <h2>Programmplan</h2>
  <ul id="station_schedule_list">
    <%= render(
      :partial => 'station_schedule_item',
      :collection => @station.broadcasts
    ) %>
  </ul>
  <div id="station_schedule_list_js">
    <%= sortable_element(
      :station_schedule_list,
      :url => { :action => :order_schedule }
    ) %>
  </div>
</div>
...
...
```

Damit die Liste bestehender Sendungen von Beginn an sortiert werden kann, müssen Sie natürlich auch hier aus dem ul-Element ein Sortable machen. Der Aufruf des JavaScript-Generators sortable_element erfolgt dabei, wie Sie sicher sehen, in aus dem Controller bekannter Weise. Lediglich auf den Receiver page müssen Sie verzichten. Das war's. Wenn Sie jetzt einen Sender bearbeiten möchten, erscheint der komplette Programmplan auf der Seite. Für den Fall, dass dieser bereits etwas zu lang geraten ist, gibt es ein Mittel.

Sendungen aus dem Programm streichen

Die Action delete_broadcast besteht wie das Pendant apply_broadcast aus zwei Teilen. Zunächst wird die Sendung aus der Datenbanktabelle entfernt, dann aus der sortierbaren Liste station_schedule_list.

Wie Sie im Partial `station_schedule_item` sehen können, erhält die Action die ID des Datensatzes, die Sie nicht mit der Video-ID verwechseln dürfen.

Beispiel 5-35 : StationsController#delete_broadcast

```
def delete_broadcast
  @id = params[:id]
  Broadcast.destroy(@id)

  render(:update) { |page|
    page.visual_effect(
      :squish,
      "item_#{@id}",
      :afterFinish => "function(){Element.remove(\"item_#{@id}\")}"
    )
  }
end
```

Zunächst wird der Datensatz mit der übergebenen ID aus der Datenbanktabelle `broadcasts` gelöscht. Anschließend wird das dazugehörige `li`-Element aus der Programmliste entfernt. Das könnte man ganz einfach mit `page.remove` erledigen. Aber hier soll es etwas effektvoller sein.

Der visuelle Effekt `squish` soll das Element langsam aus der Liste ausblenden. Allein dieser Effekt löscht es aber nicht, sondern macht es nur unsichtbar. Daher bleibt es Ihre Aufgabe, das Element zu entfernen – natürlich erst dann, wenn das optische Feuerwerk beendet ist.

Das erreichen Sie durch die Nutzung von `afterFinish`. Bei dieser Option können Sie JavaScript-Code hinterlegen, der ausgeführt wird, wenn der Effekt `squish` optisch nichts mehr zu bieten hat.

Den Programmablauf festhalten

Sie wissen bereits, dass das Sortable mittels Ajax die Action `order_schedule` aufruft, sobald sich etwas an der Reihenfolge der Sendungen des Programmplans getan hat. Dadurch können Sie jede Veränderung sofort in der Datenbank speichern. Das Sortable übergibt der Methode einen Parameter, der genau so heißt, wie das Element, welches die zu sortierenden Elemente umhüllt.

Dabei handelt es sich um ein Array von Datensatz-IDs, wobei diese in der Reihenfolge aufgeführt sind, wie sie auch in der Liste vorkommen.

Beispiel 5-36 : StationsController#order_schedule

```
def order_schedule
  params[:station_schedule_list].each_with_index { |id, position|
    Broadcast.update(id, :position => position)
```

Beispiel 5-36 : StationsController#order_schedule (Fortsetzung)

```
    }
    render(:nothing => true)
end
```

Mithilfe von Array#each_with_index können Sie über das Array station_schedule_list iterieren. Mithilfe des Index' eines jeden Elements, den Sie durch die Blockvariable position erhalten, können Sie die Position einer Sendung in der Programmliste feststellen und mit der Methode update in die Datenbanktabelle broadcasts schreiben. Der zu verändernde Datensatz wird dabei durch die ID des jeweiligen Videos bestimmt.

Abgesehen von der optischen Aufmachung, die natürlich noch auf der Agenda steht, ist der Teil von TVsendr, mit dem Sie ein Programm zusammenstellen können, fertig. Wobei, so ein paar Kleinigkeiten gibt's ja immer.

Feinschliff im Programmeditor

Ich habe zwei Dinge zu bemerken. Erstens ist eine Liste mit 20 Suchergebnissen, vor allem, wenn sie ausschweifende Beschreibungstexte beinhalten, viel zu lang. Der Vorschaubereich rutscht dabei in die Tiefen der Belanglosigkeit ab, während Programmplan und Senderdetails oben bleiben. Zweitens bin möglicherweise nicht nur ich verunsichert, ob ein gestarteter Suchvorgang auch wirklich sucht. Schließlich dauert dieser ein paar Augenblicke und währenddessen tut sich nichts auf der Webseite. Ohne Ajax war das anders. Als Seiten-Reloads noch auf der Tagesordnung standen, merkte man, ob gerade etwas passiert oder nicht.

Das erste Problem zu lösen ist recht simpel. Lassen Sie uns den Anzeigebereich der Suchergebnisse einfach auf eine gewisse Höhe beschränken und zudem festlegen, dass, wenn die Suchergebnisse mehr Platz benötigen als Sie ihnen zubilligen, eine Scrollleiste eingeblendet werden soll. Beide Festlegungen können Sie mit CSS regeln und in layout.css platzieren. Der entsprechende Selektor heißt #broadcast_search_results.

Beispiel 5-37 : layout.css: Eine scrollbare Ergebnisanzeige

```
...
#broadcast_search_results {
  height: 300px;
  overflow: auto;
}
...
```

Auch für das zweite Problem gibt es eine Lösung. Eine, die mittlerweile Standard bei allem ist, was mit Ajax lädt. Es handelt sich dabei um so genannte *Aktivitätsindikatoren*. Die kennen Sie bestimmt schon. Meist treten sie als in einem Kreis angeordnete Striche auf, wobei einer nach dem anderen wie bei einer Laola im Stadion aufleuchtet.

Im Internet gibt es bereits fertige Grafiken für diese Zwecke. Ich schlage vor, Sie suchen sich zum Beispiel eine attraktive Indikator-Animation auf <http://www.nappy-fab.com/ajax-indicators/> aus und speichern Sie als *indicator.gif* im Verzeichnis *public/images*.

Diese Animation soll nun genau neben dem Suchen-Button in *edit.rhtml* platziert werden. Dabei soll sie zunächst nicht sichtbar sein. Erst, wenn eine Suchanfrage läuft, wird die Animation eingeblendet. Sie verschwindet wieder, sobald Suchergebnisse eingetroffen sind. So weiß jeder Benutzer, dass da was im Hintergrund läuft.

Beispiel 5-38 : Integrieren von indicator.gif in edit.rhtml

```
...
<%= text_field("broadcast_search", "search") %>
<%= select("broadcast_search", "channel", Broadcast.channels) %>
<%= submit_tag("Suchen") %>
<%= image_tag(
  'indicator.gif',
  :id => 'search-indicator',
  :style => 'display:none'
) %>
</form>
...
```

Dass die Grafik zunächst nicht sichtbar ist, liegt am Inhalt der *style*-Option. Durch sie wird später ein gleichnamiges Attribut im *img*-Element erzeugt. Beachten Sie auch, dass die Angabe der Bilddatei ohne Pfad auskommt. Rails sucht in solchen Fällen im Verzeichnis *public/images* nach dem angegebenen Dateinamen. Außerdem ist die Vergabe einer ID sehr wichtig. Über diese können Sie das Element ansprechen und mit JavaScript ein- oder auch ausblenden.

Genau das soll passieren, wenn das Suchformular abgeschickt wird beziehungsweise Suchergebnisse eingetroffen sind. Für diese Zwecke kennt *form_remote_tag* die beiden Optionen *loading* und *complete*. Ihnen können Sie JavaScript-Code zuweisen, der beim jeweiligen Ereignis ausgeführt wird. Somit können Sie *form_remote_tag* der Videosuche wie folgt ergänzen:

```
...
<h2>Sendungen ausw&uuml;hlen</h2>
<%= form_remote_tag(
  :url => {:action => :search_broadcasts},
  :loading => "Element.show('search-indicator')",
  :complete => "Element.hide('search-indicator')"
) %>
<%= text_field("broadcast_search", "search") %>
<%= select("broadcast_search", "channel", Broadcast.channels) %>
<%= submit_tag("Suchen") %>
...
```

Der JavaScript-Code blendet mit `Element.show` das HTML-Element mit der ID `search-indicator` ein, sobald die Ajax-Transaktion beginnt. Endet sie, blendet `Element.hide` das selbe Element wieder aus. Diese Technik sollten Sie nicht nur bei TVsendr, sondern stets nutzen, wenn Sie via Ajax mit dem Server kommunizieren, das für den Benutzer aber nicht ersichtlich ist.

Mit diesen beiden Maßnahmen enden unsere Arbeiten am Programmeditor. Er ist schließlich zumindest funktionell voll einsatzfähig. Das kann man von der Oberfläche, auf der die Sender ausgestrahlt werden sollen, wahrlich nicht behaupten.

Wir bauen uns einen Fernseher

Als Basis für Ihr Fernsehgerät im Browser soll die Action und das View `show` herhalten. Beide existieren dank Scaffolding bereits, sind aber bislang als Bauteile für ein virtuelles TV-Gerät eher ungeeignet. Aber das kann sich schnell ändern.

Die Grundstruktur der Oberfläche besteht aus einem Bereich im oberen Teil des Seiten, der Sendernamen und -beschreibung anzeigt. Darunter folgt links eine eher kleine Spalte mit der Programmvorstellung, rechts daneben sollen die Sendungen zu sehen sein. Auch hierbei ist Platz für ein paar erläuternde Worte, die aus Titel und Beschreibung einer jeden Sendung bestehen. Somit steht folgendes Grundgerüst fest.

Beispiel 5-39 : show.rhtml

```
<div id="show_top">
  <h1><%= @station.name %></h1>
  <p><%= @station.description %></p>
</div>

<div id="show_left">
  <div id="show_schedule">
    <h2>Programm</h2>
  </div>

  <div id="show_other_stations">
    <h2>Weitere Sender</h2>
  </div>

  <%= link_to 'Neuer Sender', :action => 'new' %> |
  <%= link_to 'Edit', :action => 'edit', :id => @station %> |
  <%= link_to 'Back', :action => 'list' %>
</div>

<div id="show_right">
  <div id="show_player">
  </div>
</div>
```

Zugegeben, so richtig viel übrig geblieben ist nicht von der ursprünglichen *show.rhtml*. Aber dafür funktioniert die Anzeige der Senderinformationen schon tadellos – ohne, dass *StationsController#show* auch nur minimal verändert wurde.

Bevor *show.rhtml* weitere beeindruckende Highlights implantiert bekommt, gilt es noch, *layout.css* zu erweitern. Die beiden Spalten *show_left* und *show_right* müssen noch nebeneinander angeordnet werden, wobei sie 10 Pixel zwischen sich lassen sollen.

Beispiel 5-40 : layout.css

```
...
#show_left {
    width: 200px;
    float:left;
}

#show_right {
    width: 740px;
    float: right;
    margin-left: 10px;
}
```

Um in *show.rhtml* das Programm des angezeigten Senders zu listen, bedarf es ebenfalls keinerlei Änderungen an *StationsController#show*. Dank Assoziationen können Sie bekanntlich über *@station.broadcasts* auf die Sendungen zugreifen. Es empfiehlt sich, den Programmplan als Liste zu implementieren.

Beispiel 5-41 : Programmvorstellung in show.rhtml

```
...
<div id="show_schedule">
    <h2>Programm</h2>
    <ul>
        <% for broadcast in @station.broadcasts %>
        <li><%= link_to_remote(
            h(broadcast.title),
            :url => {
                :action => :show_broadcast,
                :video_id => broadcast.video_id
            },
            :update => 'show_player'
        ) %></li>
        <% end %>
    </ul>
</div>
...
```

Der Programmplan besteht somit aus li-Elementen, die jeweils einen Link enthalten. Jeder Link verweist auf die Action `show_broadcast` und übermittelt zudem die AOL-eigene ID des entsprechenden Videos. Das Ergebnis dieses Ajax-Calls ist der Austausch des Inhalts des div-Elements `show_player` – so steht es in der Option `update`.

Unter dem elektronischen Programmführer soll eine Liste weiterer Sender erscheinen. Auch hier bedienen wir uns der beliebten linkhaltigen ul-/li-Kombination.

Beispiel 5-42 : show.rhtml zeigt weitere Sender an

```
...
<div id="show_other_stations">
  <h2>Weitere Sender</h2>
  <ul>
    <% for station in Station.find(:all, :order => 'name ASC') %>
      <li><%= link_to(
        h(station.name),
        :id => station.id
      ) %></li>
    <% end %>
  </ul>
</div>

<%= link_to 'Neuer Sender', :action => 'new' %> |
...

```

Um alle Sender, die TVsendlr kennt, aufzulisten, können Sie direkt in der View auf das Model `Station` zugreifen und die Klassenmethode `find` anwenden. Sie liefert in diesem Falle alle Sender in alphabetischer Reihenfolge. Da der Aufruf eines neuen Senders wieder über die Action `show` läuft, in der Sie sich gerade befinden, genügt als Link-Ziel allein die ID des gewünschten Senders.

Fehlt jetzt noch was? Ach ja, der Mattscheiben-Bereich der virtuellen Glotze. Um ihn zu implementieren, sind zwei Schritte nötig: Der erste befördert die erste Sendung des Programmplans des gewählten Senders direkt beim Laden der `show`-Action in die gleichnamige View. Im zweiten müssen wir dafür sorgen, dass die nachfolgenden Sendungen auf Wunsch angezeigt werden.

Da es für den zweiten Schritt schon entsprechende Links in der linken Spalte der Webseite gibt, fangen wir einfach damit an. Die Action `show_broadcast` wird durch jeden dieser Links aufgerufen. Da die Video-ID als Parameter vorliegt und mit `view_broadcast.rhtml` bereits eine passende View existiert, ist das eine Angelegenheit von vier Zeilen Ruby-Code. Da `show_broadcast` eine View rendert, die per JavaScript in die Seite gesetzt wird, ist es dabei wieder erforderlich, das Rendern des Layouts zu verhindern. Als View kommt `view_broadcast.rhtml` zur erneuten Verwendung.

Beispiel 5-43 : StationsController#show_broadcast

```
def show_broadcast
  @video = Broadcast.get_video(params[:video_id])
  render(:action => 'view_broadcast', :layout => false)
end
```

Der zweite Schritt kann damit als erledigt angesehen werden. Aktualisieren Sie Ihren Browserinhalt und klicken Sie auf einen der Sendungs-Links in der Programmliste.

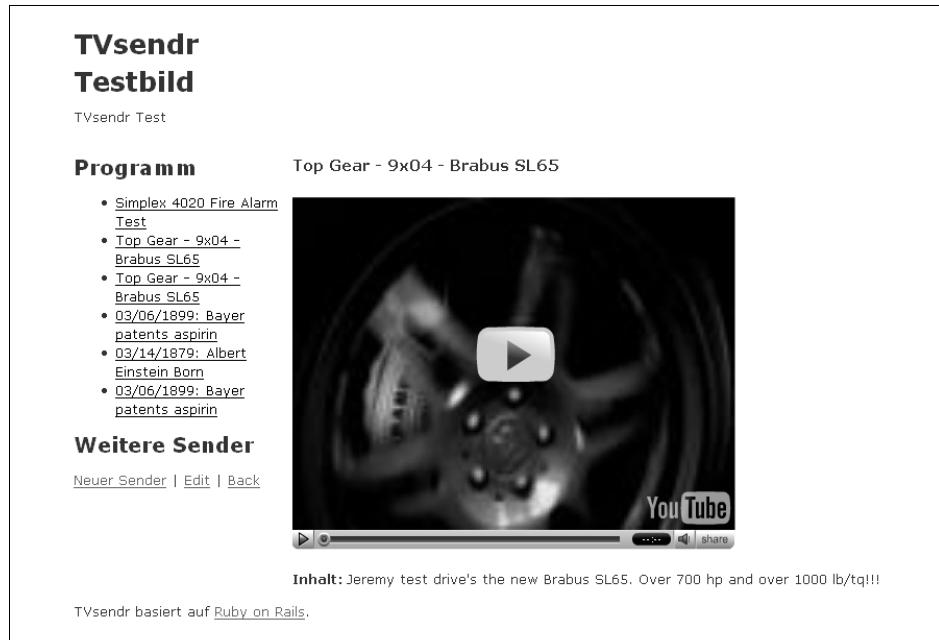


Abbildung 5-6: Fast fertig: TVsendr zeigt erste Sendungen

Um automatisch die erste Sendung beim Anzeigen der show-Action zu laden, müssen Sie StationsController#show beauftragen, herauszufinden, wie denn die Video-ID der fraglichen Sendung lautet – sofern dieser Sender überhaupt schon ein Programm hat.

Beispiel 5-44 : StationsController#show wird erweitert

```
def show
  @station = Station.find(params[:id])
  if @station.has_broadcasts?
    @broadcast = @station.broadcasts.find(:first)
    @video = Broadcast.get_video(@broadcast.video_id)
    @video_content = render_to_string(:action => 'view_broadcast', :layout => false)
  end
end
```

Interessant in dieser Erweiterung der show-Action ist zunächst einmal die Methode `Station#has_broadcasts?`. Sie steht Ihnen dank der `has_many`-Assoziation zur Verfügung und gibt `true` zurück, wenn ein Sender über Sendungen verfügt. Sollte ein anderes Model als Broadcast mit Station verbandelt sein, so würde sich selbstverständlich der Methodenbezeichner diesem Umstand anpassen und etwa `has_comments?` heißen.

Noch ein interessanter Punkt ist die Methode `render_to_string`. Sie funktioniert komplett wie die Ihnen bekannte `render`-Methode, gibt aber ein String-Objekt zurück, das hier als `@video_content` angesprochen werden kann. `Render_to_string` ändert aber nichts daran, dass die Action `show` die gleichnamige View inklusive Layout rendernt. Das Standardverhalten wird also nicht beeinflusst. Jedoch steht Ihnen in `@video_content` eine weitere gerenderte View zur Verfügung. Und die bauen wir jetzt noch in `show.rhtml` ein.

Beispiel 5-45 : show.rhtml lädt die erste Sendung

```
...
<div id="show_right">
  <div id="show_player">
    <% if @video %>
      <%= @video_content %>
    <% else %>
      <p>Dieser Sender strahlt zurzeit kein Programm aus.</p>
    <% end %>
  </div>
</div>
```

Hier wird überprüft, ob `@video` etwas beinhaltet. Das ist dann der Fall, wenn `StationsController#show` ein mit mindestens einer Sendung gefülltes Programm gefunden hat. Ist das nicht der Fall, wird der Hinweis ausgegeben, dass der Sender programmlos ist.

Mit diesen Codezeilen funktioniert nun auch der eigentliche Fernsehbetrieb Ihrer Anwendung. Durch simple Klicks auf die Sendungen des Programmplans können Sie nun Ihren eigenen Fernsehsender sehen – oder anderen zur Verfügung stellen.

Damit Dritte nicht von der schlichten Optik abgeschreckt werden, sollen abschließende kosmetische Behandlungen mit CSS das Projekt `TVsendr` beschließen.

Ein TV-Gerät mit Style

Dank Scaffolding existiert bereits eine CSS-Datei neben `layout.css` namens `scaffold.css`. Sie befindet sich im Verzeichnis `public/stylesheets`. Sie können diese Datei gern benutzen, schließlich ist sie auch schon in der Layout-Datei eingebunden.

Die erste optische Maßnahme betrifft sämtliche Überschriften des Projekts von h1 bis h3. Die folgenden CSS-Eigenschaften sollen sie etwas ansehnlicher gestalten.

```
h1, h2, h3 {  
    margin: 12px 0;  
    padding: 5px 0;  
    font-family: tahoma, arial, helvetica, sans-serif;  
    color: #B33;  
    border-bottom: 1px solid #F22;  
}
```

Alle Überschriften erhalten einen einheitlichen Außen- und Innenabstand, lediglich in der Schriftgröße, die jeweils in der voreingestellten Größe verbleibt, unterscheiden sie sich noch. Allen gemein ist ein dezentes Rot und eine Unterstreichung. Im Header befindet sich auch ein h1-Element, welches den Schriftzug TVsendr unterschiedlich dazu darstellen soll.

```
#header h1 {  
    font-family: "times new roman", "times", serif;  
    font-size: 60px;  
    border: 0;  
    color: #BBB;  
}
```

Zugegeben, *Times New Roman* ist kein Ausbund an Attraktivität. Doch ab einer bestimmten Größe – und 60 Pixel ist nicht gerade klein – ist sie einigermaßen erträglich.

Um einzelne Bereiche insbesondere in *show.rhtml* und *edit.rhtml* voneinander zu trennen, sollen folgende Eigenschaften helfen. Sie beschreiben Hintergrund, Innenabstand, Abstand nach unten und die Beschaffenheit der Umrandung.

```
.box {  
    padding: 10px;  
    border: 1px solid #F22;  
    margin-bottom: 10px;  
    background: #FFE;  
}
```

Damit .box wirken kann, müssen Sie nun noch alle Bereiche, die diesen Stil übernehmen sollen, mit dem Attribut class und dem Wert box ausstatten. Zum Beispiel so:

Beispiel 5-46 : edit.rhtml: broadcast_preview wird in Form gebracht

```
...  
<div id="broadcast_preview" class="box">  
    <h2>Vorschau</h2>  
    <div id="broadcast_preview_player">  
        <p>Bitte w&auml;hlen Sie eine Sendung aus.</p>  
    </div>  
</div>  
</div>
```

Neben broadcast_preview sollten Sie in *edit.rhtml* auch die div-Elemente mit den IDs station_details, station_schedule und broadcast_search derartig ergänzen. In *show.rhtml* stünde dies show_top, show_schedule, show_other_stations und show_player gut zu Gesicht.

Weiter geht's mit den Drag and Drop Elementen in *edit.rhtml*. Sie basieren auf li-Elementen, die von dem ul-Element station_schedule_list umgeben werden. Mit den folgenden Eigenschaften wandeln Sie die eher schlichte Optik der HTML-Liste so um, dass daraus kleine Kästen werden. Die lassen sich viel einfacher draggen und droppen..

```
#station_schedule_list {  
    list-style: none;  
    margin: 0;  
    padding: 0;  
}  
  
#station_schedule_list li {  
    padding: 5px;  
    margin: 5px 0;  
    background: #EEE;  
    border: 1px dashed #222;  
}  
  
#station_schedule_list h3, #station_schedule_list p {  
    margin: 0;  
}
```

Jede Sendung im Programmplan wird nunmehr durch einen Kasten mit grauem Hintergrund und einem ein Pixel dicken, dunklen Rand dargestellt.

Ähnliches passiert mit dem Bereich, in dem die Suchergebnisse in *edit.rhtml* dargestellt werden sollen. Hier empfiehlt sich besonders die Definition eines Abstandes zwischen den aufgelisteten Videos und dem Rand, zumal auf der rechten Seite unter Umständen eine Scrollleiste vorhanden ist. Ein Abstand würde auch den Vorschaubildern nicht schaden, damit Videotitel, Links und Beschreibung nicht direkt am Bild kleben.

```
#broadcast_search_results {  
    padding: 5px;  
    border: 1px solid #222;  
    background: #EEE;  
}  
  
#broadcast_search_results img {  
    margin: 0 5px 5px 0;  
}
```

Was das Menü zum Durchblättern der Suchergebnisse betrifft, so soll auch dieses eine Umrandung und einen farblich abgesetzten Hintergrund erhalten. Ein zartes Lindgrün soll es hier sein.

Da die Navigation direkt an den Bereich, der die Suchergebnisse listet, angrenzt, können Sie hier auf den oberen Rand verzichten. Den stellt schon `#broadcast_search_result`.

```
#broadcast_search_results_navigation {  
    padding: 5px;  
    border: 1px solid #222;  
    border-top-width: 0;  
    background: #EEB;  
}  
  
#broadcast_search_results_navigation p {  
    margin: 0;  
    padding: 0;  
}
```

Die beiden Listen in `show.rhtml`, welche das Programm und andere Sender anzeigen, erhalten ebenfalls ihr eigenes Styling. Dies beinhaltet den Wegfall der Bullets und den Verlust der Einrückung nach rechts. Damit die einzelnen Bestandteile der Listen dennoch auseinander gehalten werden können, soll zwischen den einzelnen `li`-Elementen jeweils eine Lücke von 6 Pixeln klaffen.

```
#show_other_stations ul, #show_schedule ul {  
    margin: 0;  
    padding: 0;  
    list-style-type: none;  
}  
  
#show_other_stations ul li, #show_schedule ul li {  
    margin-bottom: 6px;  
}
```

Ebenfalls in `show.rhtml` befindet sich ein `div` mit der ID `show_top`. Zur Erinnerung: Hier werden die Informationen zum Sender angezeigt. Damit diese Box nicht zu viel Platz wegnimmt, können Sie die standardmäßige Ausdehnung des `p`-Elements eingämmen, indem Sie Innen- und Außenabstand auf 0 setzen.

```
#show_top p {  
    margin: 0;  
    padding: 0;  
}
```

Zum Schluss soll natürlich auch der Footer mit einigen CSS-Eigenschaften bedacht werden. Entsprechend dimensionierte Innen- und Außenränder, sowie eine horizontale Linie oberhalb des dort enthaltenen Textes soll ihn vom Rest der Anwendung abgrenzen. Darüber hinaus soll die wichtige Botschaft zentriert dargestellt werden.

```
#footer {  
    margin-top: 10px;  
    padding: 15px;  
    text-align: center;  
    border-top: 1px solid #000;  
}
```

Diese moderne Art der Fußpflege und all die vorhergehenden Eigenschaften haben zur Folge, dass aus dem schlichten, durch Scaffolding erzeugten Äußen von TVsendr mit einfachen Mitteln eine halbwegs brauchbare Oberfläche geworden ist.

The screenshot shows the TVsendr website. At the top, there's a header with the logo 'TVsendr' and a sub-section 'RailsTV' with the subtitle 'Aktuelles zu Ruby on Rails, der Nummer 1 unter den Frameworks für Web-Applikationen. Screencasts Video-Podcasts und Live-Mitschnitte von Kongressen und Meetings.' Below this is a sidebar with a 'Programm' section containing links to 'diango: Web Development for Perfectionists with Deadlines', 'HTML/CSS/JavaScript is Local Part 2', and 'Ruby on Rails, Vim, and rails.vim'. Another sidebar lists 'Weitere Sender' with links to 'BerlinTV', 'RailsTV', and 'The Simpsons'. The main content area has a heading 'HTML/CSS/JavaScript is Local Part 2' with a screenshot of a Safari browser window showing the URL 'http://www.rubyonrails.org/down'. Below the browser screenshot is a text block: 'Rails is low on dependencies and prides itself on most everything you need in the box. To get started with Ruby, the language, and RubyGems, the packa...'. A video player is embedded in the page, showing a thumbnail for a video titled 'Rails installieren - Ruby modules' with a play button and a progress bar at 0:30. Below the video player is the caption 'Inhalt: Andrew shows a small demo with ruby on rails'. At the bottom of the page, a footer note says 'TVsendr basiert auf Ruby on Rails.'

Abbildung 5-7: TVsendr ist einsatzbereit

Sollte es Ihnen so gehen wie mir, dann werden Sie sicher viel Spaß an TVsendr haben. Während der Entwicklung dieses Projekts gab es oft Situationen, in denen ich einfach nicht weiterkam – weil im Vorschau-Player gerade etwas Sehenswertes lief.

Abspann

Wenn Sie sich dennoch noch für ein paar Minuten losreißen können, dann empfiehlt sich die Eindeutschung der Anwendung. Durch das Scaffolding zu Beginn des Entwicklungsprozesses sind viele Button- und Linkbeschriftungen noch auf Englisch. Jetzt, wo die Anwendung steht, ist der ideale Zeitpunkt zur Übersetzung ins Deutsche.

Und wenn Sie dann noch, analog zu Picsblog, die Standard-Route in *config/routes.rb* auf den Controller *stations* lenken, ist auch die letzte Kleinigkeit in Bezug auf TVsendr erledigt. Danach können Sie sich einen Fernsehabend gönnen.

Ich hab noch einen Tipp für Sie. Einer Schwierigkeit, der Sie im Zusammenhang mit Ajax-Applikationen immer wieder begegnen werden, ist das Fehlen von direkt sichtbaren Fehlermeldungen. Wenn Sie innerhalb Ihrer Anwendung einen Link beispielsweise mit `link_to_remote` platzieren, ein Klick auf selbigen aber zu rein gar nichts führt, ist guter Rat teuer.

Oder ganz und gar kostenlos. Mit dem freien Firefox-Addon *Firebug* können Sie direkt in Ihre Ajax-Calls blicken. Der in Firebug eingebaute *XMLHttpRequest-Spion* zeigt Ihnen dabei, welche Parameter Ihre Anwendung an den Server geschickt hat und welche Antwort daraufhin zurückgekommen ist. In eben dieser Antwort kann der Schlüssel des Problems liegen. Dieser kann sogar eine von Ruby on Rails erzeugte Fehlermeldung sein, die Ihnen ohne Firebug entgehen würde.

Sie bekommen Firebug unter <http://www.getfirebug.com>. Die Website enthält zudem einen umfangreichen Überblick über das, was Firebug noch kann.

Zusammenfassung

Damit endet ein Kapitel, das Ihnen speziell den Umgang mit Ajax in Ruby on Rails näher bringen wollte. Sie haben gesehen, welche Möglichkeiten Ihnen zur Verfügung stehen, Daten ohne Reload der Webseite an den Server zu senden und von ihm zu empfangen. Durch die perfekte Integration von Prototype und Script.aculo.us in Ruby on Rails ist die Entwicklung Ajax-basierter Anwendungen ein Kinderspiel. Zeitgemäße Standards bei der Gestaltung von Weboberflächen wie Drag and Drop oder das Anzeigen von inhaltlichen Veränderungen durch visuelle Effekte sind nahtlos in Ruby on Rails integriert.

Darüber hinaus haben Sie erfahren, wie Scaffolding besonders den Anfangszeitraum der Entwicklung einer Webapplikation extrem beschleunigt und dem Begriff Rapid Prototyping alle Ehre macht. Sie haben auch gesehen, wie Sie über REST einen Webservice fernbedienen und XML-Daten auswerten können. Mit diesem Wissen können Sie schon sehr brauchbare eigene Rails-Anwendungen schreiben.

Sollten Sie nun planen, mit TVsendr online zu gehen, so können Sie all die modernen, hier verwendeten Technologien wie Ajax, Webservices, REST, Ruby on Rails durch die Nutzung eines weiteren derzeitigen Trend noch aufwerten: Platzieren Sie gut sichtbar den Hinweis, dass sich Ihr Angebot noch im Beta-Stadium befindet. Das ist ein Muss bei Applikationen dieser Art. Eine Anleitung zum Veröffentlichen einer Rails-Anwendung finden Sie übrigens im nächsten Kapitel. Dort erfahren Sie, wie Sie Ihre Rails-Anwendung auf einen Webserver transferieren und dort funktionstüchtig machen.

Falls Sie derartiges nicht vorhaben, nutzen Sie doch TVsendr, um Ihren eigenen Ruby-on-Rails-Kanal zu gründen. Denn auch bei YouTube und Google Video gibt es interessante Screen- und Videocasts, sowie Live-Mitschnitte von Kongressen und Konferenzen zum Thema Ruby on Rails.

Rails-Anwendungen veröffentlichen

In diesem Kapitel:

- Bevor es losgeht
- Ruby und Rails installieren
- FastCGI installieren
- Eine Anwendung auf dem Server einrichten
- Auf dem Weg zum Profi
- Zusammenfassung

Toll. Gerade eben in Kapitel 5 haben Sie eine Rails-Anwendungen programmiert, die vor technischen Raffinessen nur so strotzt. *Scaffolding*, *REST*, *Webservices*, *Ajax*, Effekte. Und die Anwendung läuft auch tadellos, allerdings nur auf Ihrem Rechner. Doch was nützen die schönsten Anwendungen, die Sie mit *Ruby on Rails* erstellen, wenn sie keiner außer Ihnen sieht und verwenden kann? Wenig, sehr wenig. Darum: Befreien Sie Ihre Rails-Anwendungen aus der Enge Ihres Rechners, entlassen Sie sie ins große World Wide Web.

Nun ist das nicht ganz einfach, weil das Angebot an Hosting-Angeboten mit Unterstützung für Ruby on Rails zum Zeitpunkt, da dieses Buch entsteht, noch etwas zu wünschen übrig lässt. Aber das wird sich ganz bestimmt bald ändern, denn die Nachfrage nach Angeboten mit Ruby on Rails wird in hohem Maße zunehmen.

Bis dahin bleibt Ihnen wohl nicht anderes übrig, als die Angebote ausländischer, insbesondere US-amerikanischer Unternehmen wahrzunehmen oder sich für einen der wenigen kleinen Rails-Hoster unserer Breitengrade zu entscheiden. Eine entsprechende Übersicht finden Sie im Anhang.

Doch es gibt noch eine weitere Möglichkeit, wie Ihnen die folgenden Seiten beweisen möchten. Dort erfahren Sie, wie Sie recht preiswert an Ihren eigenen Rails-Webserver kommen. Am Beispiel eines virtuellen Servers aus dem Hause STRATO, den man Ihnen dort für knapp unter zehn Euro im Monat überlässt, möchte ich Ihnen zeigen, wie Sie Ihre Rails-Anwendungen veröffentlichen können. Der genannte virtuelle Server ist zwar wie so viele grundsätzlich Ruby- und Rails-frei, aber mit wenig Aufwand und der Gottesgabe Root-Zugang können Sie das selbst ändern.

Anhand einer kompletten Schritt-für-Schritt-Anleitung werden Sie hier nachvollziehbar lesen können, wie Sie *Ruby*, *RubyGems* und *Rails* auf dem entfernten Server installieren und wie Sie den dort werkenden *Apache* um *FastCGI* erweitern, was Ihre Rails-Anwendungen schön schnell machen wird. Anschließend soll es darum gehen, am Beispiel von TVsendr zu zeigen, wie Sie Ihre Anwendung auf dem Server nutzbar machen können.

Das alles funktioniert natürlich nicht nur mit dem hier verwendeten *V-PowerServer A* von STRATO. Bei allen Servern mit Root-Zugang und Linux als Betriebssystem sind die gleichen oder ähnlichen Arbeitsschritte möglich. Sollte Ihr Server mit *SuSE 9.3* laufen und mit *Plesk* administrierbar sein, werden Sie kaum Unterschiede in der Installation feststellen.

Bevor es losgeht

Sollten Sie vor der Entscheidung stehen, sich einen eigenen Server zuzulegen, um darauf Ihre Rails-Anwendungen zu platzieren, sollten Sie bedenken, dass Sie den Server, und sei es nur ein virtueller wie hier, selbstverantwortlich betreiben. Das bedeutet auch, dass Sie Ihren Server ständig pflegen und gegen allerlei Bedrohungen aus den zweilichtigen Bereichen des Webs schützen müssen. Eine gewisse Grundkenntnis der Materie wäre keine all zu schlechte Voraussetzung für den Betrieb eines eigenen Servers. Wägen Sie also gut ab und entscheiden Sie sich im Zweifel für einen der Anbieter, die Ihnen fertige Hosting-Angebote mit Rails-Unterstützung bieten. Auf eine entsprechende Liste mit Links im Anhang dieses Buches sei an dieser Stelle noch einmal hingewiesen. Sollten Sie sich für den Server entscheiden, bleibt mir noch diese Anmerkung: Das Operieren am offenen Herzen eines Webserver, und nicht anderes werden Sie gleich tun, ist natürlich immer mit kleinen bis mittleren Risiken verbunden und bedarf einer gewissen Kenntnis über das Innenleben desselben. Sollten Sie sich diesbezüglich etwas unsicher fühlen, empfehle ich Ihnen dringend ein Backup des gesamten Systems. Bei STRATO und vielen anderen Anbietern erfolgt das täglich automatisch, so dass Sie zur Not auf den Stand des Vortages mit wenigen Klicks zurückkehren können.

Die folgende Anleitung geht davon aus, dass Ihr Server einwandfrei läuft und bereits über eine korrekt konnektierte Domain mit physischem Hosting verfügt. Falls nicht, sollten Sie gegebenenfalls noch fix eine Domain bestellen und diese beispielsweise über Plesk für Ihren Server verfügbar machen.

Außerdem benötigen Sie noch etwas Software, damit das Fernbedienen Ihres entfernten Servers leicht von der Hand geht und Sie Ihre Anwendung überspielen können. Sollten Sie Windowsianer sein, wäre *WinSCP* eine gute Wahl. Damit können Sie per *SSH (Secure Shell)* auf Ihren Server zugreifen, als ob Sie direkt vor ihm sitzen würden. Vor allem ermöglicht WinSCP Dateitransfers zwischen Ihrem Rechner und dem Webserver. Sie bekommen das Freeware-Programm und Informationen dazu

auf der Website <http://www.winscp.net>. Die Download-Page <http://winscp.net/eng/download.php> hält eine multilinguale Version bereit, die es Ihnen ermöglicht, WinSCP mit deutschsprachiger Oberfläche nutzen zu können.

Um noch komfortabler auf den Server zugreifen zu können, brauchen Sie noch *PuTTY*, ebenfalls ein SSH-Client, der allerdings wegen seiner Fähigkeiten als Terminal geschätzt wird. Die Projektseite <http://www.chiark.greenend.org.uk/~sgtatham/putty/> stellt Ihnen PuTTY als *putty.exe* auf der Download-Seite zur Verfügung. Speichern Sie *putty.exe* am besten unter C:\Programme\PuTTY und schon sind Sie startklar.

Ruby und Rails installieren

Einige der nun folgenden Schritte werden Ihnen bereits bekannt vorkommen, wenn Sie im ersten Kapitel dieses Buches Kapitel 1 der Anleitung gefolgt sind, welche Ihrer Ubuntu-Installation Rails beibrachte. Andere werden neu sein, da Sie nur einen Webserver betreffen, der produktiv und nicht zum Entwickeln eingesetzt wird.

Öffnen Sie zunächst WinSCP und starten Sie eine neue Sitzung. Setzen Sie bei Rechnername Ihren Domainnamen, bei Benutzername root und bei Passwort Ihr Root-Passwort ein. Letzteres entnehmen Sie dem Kundenservicebereich von STRATO unter *Serverkonfiguration → Serverdaten*. Anschließend nimmt WinSCP die Verbindung auf und zeigt Ihnen die Verzeichnisstruktur Ihres Servers ausgehend vom Verzeichnis *root* an.

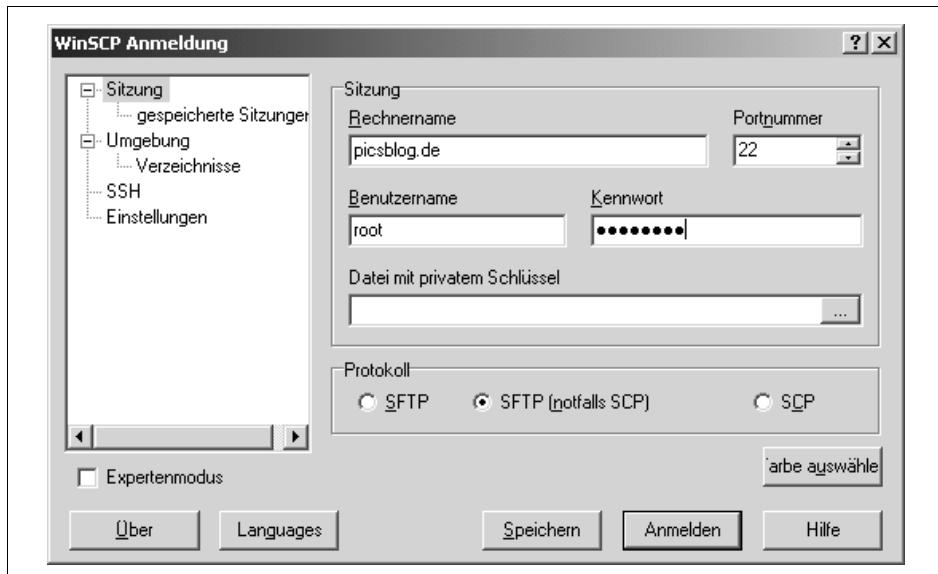


Abbildung 6-1: WinSCP für den Zugriff auf den Server vorbereiten

Klicken Sie nun auf *Befehle* → *in PUTTY öffnen* und geben Sie dort ebenfalls Ihr Root-Passwort ein. Lassen Sie während der gesamten Installation WinSCP und PuTTY laufen.

Als erstes installieren Sie den Ruby-Interpreter. Um die aktuellste Version zu erhalten, gehen Sie auf die Ruby-Website <http://www.ruby-lang.org> und klicken Sie auf *Downloads*. Hier finden Sie in der Rubrik *Ruby-Quellcode* den Link zur aktuellen Ruby-Version, hier 1.8.5. Kopieren Sie den Link in die Zwischenablage. Wechseln Sie dann zu PuTTY. Hier können Sie mit dem Befehl `wget` Dateien von externen Servern herunterladen. Geben Sie `wget` ein und setzen Sie ein Leerzeichen. Anschließend können Sie mit einem Rechtsklick auf das PuTTY-Terminal den Inhalt der Zwischenablage einfügen. Letztlich ergibt das folgende Zeile:

```
wget ftp://ftp.ruby-lang.org/pub/ruby/ruby-1.8.5.tar.gz
```

Sobald Sie dies mit *Enter* bestätigen, wird Ruby herunter geladen. Anschließend muss das Archiv entpackt, kompiliert und installiert werden. Geben Sie dazu nacheinander folgende Zeilen ein, wobei Sie nach jeder Betätigung der *Enter*-Taste der Dinge harren sollten, die auf Ihrem Bildschirm geschehen. Beachten Sie, dass die Befehle für die Ruby-Version 1.8.5 gelten und passen Sie sie gegebenenfalls an eine mittlerweile aktuellere Version an.

```
tar -xvzf ruby-1.8.5.tar.gz  
cd ruby-1.8.5  
.configure  
make  
make install  
cd
```

Anschließend können Sie mit der Eingabe von `ruby -v` überprüfen, ob die Installation erfolgreich war. Das ist dann der Fall, wenn Ihnen Ruby verrät, welche Version nun installiert ist.



Sollten Probleme bei der Installation von Ruby aufgetreten sein, überprüfen Sie, ob auf Ihrem System die Programme `gcc`, `zlib` und `make` installiert sind. Um das zu tun, geben Sie `yast` ein und schauen Sie unter Software->Install and Remove Software nach. Sollte eine der Komponenten fehlen, können Sie diese hier installieren und den Vorgang erneut starten.

Als nächstes steht *RubyGems* auf dem Programm. Auch hier laden Sie per `wget` die aktuelle Version in Quelltext-Form herunter. Den URL dazu finden Sie auf <http://rubyforge.org/projects/rubygems>. Entscheiden Sie sich hier für das `.tgz`-Archiv. Anschließend können Sie RubyGems über eine mitgelieferte Ruby-Datei installieren.

```
wget http://rubyforge.org/frs/download.php/17190/rubygems-0.9.2.tgz  
tar -xvzf rubygems-0.9.2.tgz  
cd rubygems-0.9.2  
ruby ./setup.rb  
cd
```

Auch die erfolgreiche Installation von RubyGems können Sie durch die Abfrage der Versionsnummer überprüfen. Geben Sie dazu `gem -v` ein. Sollte das von Erfolg gekrönt sein, können Sie Rails installieren.

```
gem install rails --include-dependencies
```

Und wie Sie den Erfolg der Installation prüfen können, ahnen Sie sicher bereits: `rails -v`. Nun fehlt noch Rails' Verbindung zu MySQL.

```
gem install mysql
```

Damit wäre Ihr Server schon bestens geeignet, um auf ihn Rails-Anwendungen zu entwickeln. Da das aber nicht sein Zweck sein soll, gibt es noch ein paar Dinge zu tun, damit Sie ihn produktiv einsetzen können.

FastCGI installieren

Als nächstes steht eine Erweiterung für die Webserver-Software Apache auf dem Programm. Mit *FastCGI* ist es möglich, die üblichen Zugriffe auf Apache über *CGI (Common Gateway Interface)* zu beschleunigen. Davon profitieren Ihre Rails-Anwendungen, weil Sie schneller ausgeführt werden können.

Auch für FastCGI brauchen Sie zunächst dessen Quelltext. Den erhalten Sie auf der Projektseite von FastCGI, die Sie unter <http://www.fastcgi.com> erreichen. Im Abschnitt *Servers* finden Sie unter *Apache* einen Download-Link, der in der Rubrik *current* steht. Daraufhin öffnet sich eine Liste, aus der Sie den Link zu *fcgi-2.4.0.tar.gz* in die Zwischenablage kopieren. Sollte zwischenzeitlich eine andere Version aktueller sein, dann ändert sich der Dateiname entsprechend.

Laden Sie die Datei wieder mit `wget` in PuTTY herunter. Anschließend folgt die Ihnen bereits bekannte Prozedur des Entpackens, Konfigurierens und Installierens.

```
wget http://www.fastcgi.com/dist/fcgi-2.4.0.tar.gz
tar -xvzf fcgi-2.4.0.tar.gz
cd fcgi-2.4.0
./configure
make
make install
cd
```

Damit der Apache FastCGI nutzen kann, müssen Sie noch ein passendes Modul installieren, *mod_fastcgi*. Das können Sie auch von *fastcgi.com* beziehen. Oder Sie nutzen die komfortablere Variante über den Paketmanager von *yast*. Der erledigt das mit folgender Eingabe in PuTTY:

```
yast -i apache2-mod_fastcgi
```

Das Modul, das den Apache befähigt, eine FastCGI-Schnittstelle zur Verfügung zu stellen, ist damit installiert. Allerdings weiß Apache noch nichts davon. Daher müssen Sie das Modul in eine Liste von Modulen eintragen, die beim Start des Webservers automatisch geladen werden.

Wechseln Sie dazu zu WinSCP und navigieren Sie durch die Verzeichnisstruktur Ihres Servers zu `/etc/sysconfig`. Klicken Sie mit der rechten Maustaste auf die Datei `apache2` und wählen Sie *Bearbeiten*. Anschließend suchen Sie nach einer Zeile, die mit `APACHE_MODULES=` beginnt. Hier ist die besagte Liste, der Sie ganz am Ende und unbedingt vor dem schließenden Anführungszeichen und nach einem trennenden Leerzeichen das Modul `fastcgi` hinzufügen. Das könnte anschließend so aussehen:

```
APACHE_MODULES="mod_perl access actions alias auth auth_
dbm autoindex cgi dir env expires include log_
config mime negotiation setenvif ssl userdir php4 php5 perl python suexec /usr/lib/
apache2-prefork/mod_frontpage.so rewrite fastcgi"
```

Zum Abschluss der Installation von FastCGI müssen Sie Ruby erweitern, um eben diese Funktionalität auch in Ruby nutzen zu können.

```
gem install fcgi
```

Nun müssen Sie noch den Apache neu starten. Das erledigen Sie beispielsweise über Plesk. Loggen Sie sich ein und klicken Sie im Hauptmenü auf Server. In der anschließenden Ansicht gibt es die Abteilung Starten / Stoppen des Servers mit entsprechenden Icons darunter. Klicken Sie auf Neu starten und warten Sie etwas ab, damit Ihr Server eben dieses tun kann. Spätestens, wenn Ihre Domain wieder erreichbar ist, können Sie weitermachen.

Dieses Weitermachen besteht nun darin, Ihre Rails-Anwendung zu installieren. Denn Ihr Webserver ist nun fähig, Anwendungen mit Ruby on Rails auszuführen. Alles, was Sie bisher in diesem Kapitel getan haben, brauchen Sie bei zukünftigen Rails-Anwendungen, die Sie auf Ihrem Server installieren möchten, nicht mehr tun.

Alles, was jetzt folgt, hingegen schon. Übrigens: Beim Neustarten des Servers ist die Verbindung von WinSCP und PuTTY hopps gegangen. Die können Sie nun wieder etablieren. Schließen Sie dazu das PuTTY-Fenster und klicken Sie in WinSCP auf Verbinden. Anschließend können Sie PuTTY über *Befehle->in PuTTY öffnen* wieder aktivieren.

Eine Anwendung auf dem Server einrichten

Ich empfehle Ihnen grundsätzlich, einen zentralen Ort auf Ihrem Server einzurichten, an dem Sie Ihre sämtlichen Rails-Anwendungen, die auf dem Server laufen sollen, unterbringen. Wie wäre es beispielsweise mit `/srv/www/rails`? Navigieren Sie in WinSCP zu `/srv/www` und erstellen Sie mit einem Rechtsklick auf die freie, weiße Fläche mit der Wahl des Menüpunkts *Neu → Verzeichnis...* ein Verzeichnis namens `rails`. Wechseln Sie in selbiges.

Kopieren Sie dann von Ihrer lokalen Festplatte das komplette Verzeichnis Ihrer Anwendung. In diesem Beispiel ist es das Verzeichnis `tvsendl`. Wenn Sie WinSCP mit Norton-Commander-gleicher Ansicht installiert haben, navigieren Sie im linken

Teil der WinSCP-Oberfläche zum Beispiel in das Verzeichnis *rails_apps* von InstantRails. Per Drag and Drop können Sie dann das Verzeichnis *tvsendl* auf den Server ziehen.

Ist das erfolgt, müssen Sie Ihre Anwendung an die neue Umgebung anpassen. Zunächst bedeutet das, Datenbanken zu erstellen und die Datenbankkonfiguration in *database.yml* anzupassen.

Datenbank einrichten

Beim Erstellen einer neuen Datenbank ist Ihnen Plesk behilflich. Wechseln Sie dort zur Domain, unter der TVsendr erreicht werden soll und klicken Sie dann in der Rubrik *Dienste* auf *Datenbanken*. Anschließend können Sie eine neue Datenbank hinzufügen. Hier soll uns eine genügen, die den Namen tvsendr tragen soll. Wählen Sie als Typ MySQL aus.

Anschließend fügen Sie einen neuen Datenbankbenutzer hinzu und merken sich dessen Benutzernamen und Passwort. Sobald der neue Benutzer angelegt ist, können Sie dessen Daten in *database.yml* eintragen. Navigieren Sie dazu in WinSCP innerhalb Ihrer Rails-Anwendung auf dem Server ins Verzeichnis *config* und öffnen Sie *database.yml* zur Bearbeitung.

Es genügt, wenn Sie die Abschnitte *development* und *production* an den neuen Datenbanknamen, den neuen Benutzer und das Passwort anpassen. Beide Abschnitte können die gleiche Datenbank verwenden.

Beispiel 6-1 : database.yml auf dem Webserver

```
development:
  adapter: mysql
  database: tvsendr
  username: tvsendr
  password: xxxxxxxxxxxx
  host: localhost
  socket: /var/lib/mysql/mysql.sock
...
production:
  adapter: mysql
  database: tvsendr
  username: tvsendr
  password: xxxxxxxxxxxx
  host: localhost
  socket: /var/lib/mysql/mysql.sock
```

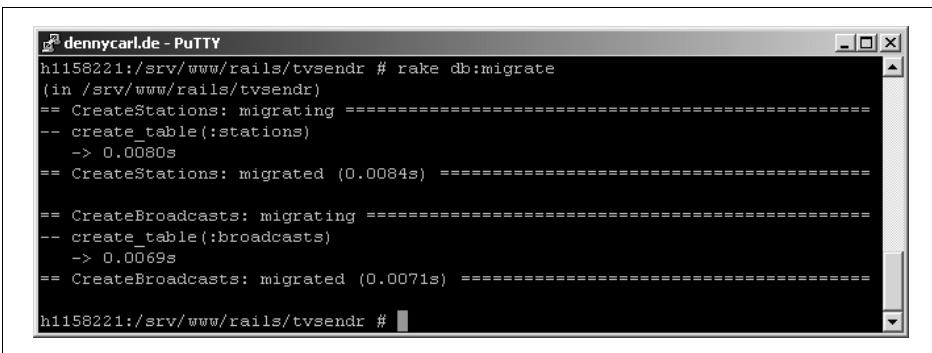
Beachten Sie bitte die jeweils letzte Zeile der beiden Abschnitte. Mit dieser wird sichergestellt, dass Ihre Anwendung mit der MySQL-Datenbank des Servers Kontakt aufnehmen kann. Der Pfad zu *mysql.sock* kann auf Ihrem Server möglicherweise anders lauten. Überprüfen Sie ihn, wenn Sie beim ersten Start der Anwendung eine entsprechende Fehlermeldung von Rails erhalten.

Sobald Sie die Datei gespeichert und wieder geschlossen haben, wird es zum ersten Mal Ernst. Die Datenbankverbindung ist konfiguriert, also können Sie die Datenbanktabellen, die die Anwendung braucht, einrichten. Aber das geht blitzschnell, denn Sie haben die dafür nötigen Informationen in Migrations festgehalten.

Da Sie hier nicht mehr mit RadRails arbeiten, müssen Sie das Migrieren eigenhändig starten. Wechseln Sie in PuTTY in das Anwendungsverzeichnis und führen Sie rake mit dem Befehl db:migrate aus.

```
cd /srv/www/rails/tvsendr  
rake db:migrate
```

Ob diese Aktion erfolgreich verläuft, können Sie während des Migrierens in PuTTY mitverfolgen. Drei Tabellen, posts, comments und users müssten dabei erzeugt werden.



```
dennycarl.de - PuTTY  
h1158221:/srv/www/rails/tvsendr # rake db:migrate  
(in /srv/www/rails/tvsendr)  
== CreateStations: migrating =====  
-- create_table(:stations)  
  -> 0.0080s  
== CreateStations: migrated (0.0084s) =====  
  
== CreateBroadcasts: migrating =====  
-- create_table(:broadcasts)  
  -> 0.0069s  
== CreateBroadcasts: migrated (0.0071s) =====  
h1158221:/srv/www/rails/tvsendr #
```

Abbildung 6-2: Die Datenbanktabellen existieren nun auch auf dem Server

Sollte der Erfolg ausgeblieben sein, überprüfen Sie zunächst die Richtigkeit der in *database.yml* gemachten Angaben und ob Ihre MySQL-Datenbank überhaupt läuft. Das können Sie via Plesk checken.

Kurz vor dem ersten Start

Wenn Sie mit Plesk eine neue Domain einrichten, so erzeugt Plesk Daten, die den Apache-Webserver beim Aufrufen dieser Domain auf ein bestimmtes Verzeichnis des Webservers lenkt. Bei der Domain *tvsendr.de* ist das beispielsweise */var/www/vhosts/tvsendr.de/httpdocs*. Da sich ihre Rails-Anwendung aber ganz woanders befindet und die öffentlich zugänglichen Dateien zudem in einem Unterverzeichnis namens *public* und nicht *httpdocs* abgelegt werden, müssen Sie den Pfad noch umbiegen.

Klicken Sie sich in WinSCP dazu in das Verzeichnis */var/www/vhosts/<domain-name>/conf*. Hier befindet sich die Datei *httpd.include*. Öffnen Sie diese zur Bearbeitung. Hier steht eine ganze Menge drin, aber nur wenige Zeilen sind nun wirklich

von Interesse. Es handelt sich dabei um den Eintrag DocumentRoot und um ein Directory, das den gleichen Pfad konfiguriert. Ändern Sie beide Pfade so ab, dass sie die Position des *public*-Verzeichnisses Ihrer Anwendung beschreiben.



Ganz wichtig: Das *public*-, nicht das Wurzelverzeichnis Ihrer Anwendung ist hier gefragt!

Außerdem müssen Sie innerhalb des Directory-Abschnitts, dessen Pfad Sie gerade geändert haben, die Zugriffserlaubnis für dieses Verzeichnis definieren. Ein idealer Ort ist dafür eine Zeile vor dem Ende des Abschnitts. Achten Sie darauf, dass innerhalb der Datei mehrere Directory-Abschnitte existieren können.

Beispiel 6-2 : httpd.include

```
<VirtualHost 85.214.80.222:80>
    ServerName tvsendlr.de:80
    ServerAlias www.tvsendlr.de
    UseCanonicalName Off
    DocumentRoot /srv/www/rails/tvsendlr/public
    ...
    <Directory /srv/www/rails/tvsendlr/public>
        <IfModule sapi_apache2.c>
            AddType text/plain .php .php4 .php3 .phtml
            php_admin_flag engine off
        </IfModule>
        ...
        Allow from all
        AllowOverride all
        Order allow,deny
    </Directory>
    ...
</VirtualHost>
```

Speichern Sie die Datei anschließend. Nun müssen Sie nur noch die Verbindung zwischen Ihrer Anwendung und der FastCGI-Schnittstelle des Apache-Servers konfigurieren. Wechseln Sie dazu in WinSCP in das *public*-Verzeichnis Ihrer Anwendung, also nach */srv/www/rails/tvsendlr/public*. Hier finden Sie die Datei *dispatch.fcgi*. Öffnen Sie diese zur Bearbeitung.

Hier müssen Sie die erste Zeile anpassen. Dort ist noch der Aufenthaltsort des Ruby-Interpreters in Bezug auf InstantRails vermerkt. Noch viel schlimmer: Es ist eine Windows-Pfadangabe, mit der Ihr Linux-Server nun überhaupt nicht klar kommt. Daher: Tauschen Sie die erste Zeile – und es muss dabei zwingend auch die erste Zeile bleiben und darf nicht wegen einer Leerzeile zur zweiten werden – gegen den Aufenthaltsort des Ruby-Interpreters auf Ihrem Server aus.

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

Speichern und schließen Sie *dispatch.fcgi* wieder. Zum Abschluss müssen Sie nun noch dafür sorgen, dass Apache beim Zugriff auf Ihre Anwendung eben diese *dispatch.fcgi* verwendet. Rails hat bei der Generierung der Anwendung auf Ihrem Rechner bereits gute Vorarbeit dafür geleistet und eine *.htaccess*-Datei spendiert. Sie befindet sich ebenfalls im *public*-Verzeichnis und sollte von Ihnen nun zur Bearbeitung geöffnet werden.

Sie finden in ihr im unteren Bereich eine Zeile, die mit `RewriteRule` beginnt und zudem *dispatch.cgi* enthält. Damit wird Apache mitgeteilt, dass er *dispatch.cgi* verwenden soll, wenn jemand auf die Anwendung zugreift. Hier soll aber *dispatch.fcgi* zum Einsatz kommen – sonst wäre die ganze Arbeit bezüglich der Einrichtung von FastCGI umsonst gewesen. Fügen Sie also das *f*, das den kleinen aber bedeutenden Unterschied macht, in die Dateinamenerweiterung ein und speichern Sie die Datei. Sie kann danach geschlossen werden.



Möchten Sie die CGI-Schnittstelle Ihres Apache-Servers trotzdem nutzen, sollten Sie natürlich den Dateinamen so belassen. In diesem Falle müssen Sie aber den Ort des Ruby-Interpreters in *dispatch.cgi* analog zu *dispatch.fcgi* anpassen.

Zum Abschluss müssen Sie noch einige Nutzungsrechte für einige Dateien und Verzeichnisse setzen. Dies geschieht, indem Sie in WinSCP mit der rechten Maustaste auf eine Datei oder ein Verzeichnis klicken und dann den Menüpunkt *Eigenschaften* ansteuern. Dort ist der Bereich *Rechte* für Sie interessant.

Damit *dispatch.fcgi* ausführbar ist, vergeben Sie für diese Datei den Oktalwert 0755. Die Verzeichnisse */tmp* und */log* Ihrer Rails-Anwendung brauchen 0777 – allerdings für alle Inhalte. Klicken Sie daher im Eigenschaftsdialog die Checkbox *Gruppe, Eigentümer und Rechte rekursiv setzen* an.

Damit wären alle Schritte getan. Der Server ist Ruby-fähig, Apache hat eine FastCGI-Schnittstelle, Ihre Anwendung ist darauf eingerichtet, die nötigen Datenbanktabellen existieren, Rechte sind vergeben – also eigentlich kann der erste Testlauf beginnen.

Starten Sie dazu Ihren Server neu und rufen Sie anschließend die Domain auf, die Sie gerade für Ihre Rails-Anwendung konfiguriert haben. Wenn anschließend alles wie gewünscht läuft, können Sie Ihre Anwendung von der *development*- in die *production*-Umgebung switchen. Nehmen Sie daher noch einmal via WinSCP Kontakt mit Ihrem Server auf und öffnen Sie noch einmal */var/www/vhosts/<domain-na.me>/conf/httpd.conf*. Setzen Sie unter den drei vorhin eingefügten Zeilen noch eine weitere:

```
...
AllowOverride all
Order allow,deny
SetEnv RAILS_ENV development
</Directory>
...
```

Speichern Sie die Datei wieder und starten Sie ein letztes Mal den Server neu. Ihre Rails-Anwendung befindet sich danach in dem Modus, der für den Betrieb im Web gedacht ist. Somit kann TVsendr auf die Öffentlichkeit losgelassen werden.

Auf dem Weg zum Profi

Sie haben gerade gesehen, wie Sie Ihre Anwendung manuell auf den Server bringen und einrichten können. Und Sie haben gesehen, dass es funktioniert. Wenn Sie allerdings von nun an häufiger diesen Vorgang durchführen werden, weil sie Gefallen an Rails gefunden und den Kopf voller Ideen haben, oder wenn Sie eine Anwendung veröffentlichen möchten, die sich noch in der aktiven Weiterentwicklungsphase befindet, dann sollten Sie den Vorgang der Veröffentlichung auf einem Webserver automatisieren.

Das Standard-Werkzeug in Rails-Kreisen ist dabei *Capistrano*. Besonders in Verbindung mit der Versionsverwaltung *Subversion* zeigt Capistrano seine Qualitäten. Die Software nimmt selbständig Verbindung zum Server auf und richtet Ihre Anwendung anhand eines so genannten Rezepts dort ein oder aktualisiert sie mit aktuellen Quelltexten.

Sollten Sie neugierig geworden sein, so besuchen die Projektseite von Capistrano, zu finden unter <http://rubyforge.org/projects/capistrano>. Eine ausführliche Beschreibung und Einführung in die Prinzipien von Capistrano gibt es auf <http://manuals.rubyonrails.org/read/book/17>.

Zusammenfassung

Mit dem Veröffentlichen einer Rails-Anwendung endet dieses Buch. Hier ist also erst einmal Endstation für Ihre erste Reise mit Ruby on Rails. Hinter Ihnen liegt eine ausführliche Einführung in Ruby und Ruby on Rails, in die geniale Sprache und das nicht minder geniale Framework. Sie haben Ruby-Anwendungen entwickelt, die außerhalb und innerhalb eines Webbrowsers laufen, eine komplexer als die andere. Und mit dem Wissen, das Sie dabei gesammelt haben, können Sie viele neue Reisen mit Ruby on Rails starten. Denn das ist hier nur die Endstation des Buches. Von hier aus vielen aber viele Schienenstränge zu neuen, aufregenden Webapplikationen mit Ruby on Rails.

Im Anhang dieses Buches finden Sie diverse Ressourcen, die Reisetipps und Streckenempfehlungen enthalten. Explizit möchte ich Sie dabei schon jetzt auf die Website zu diesem Buch hinweisen. Auf <http://www.praxiswissen-ruby-on-rails.de> finden Sie weiterführende und aktuelle Informationen zum Thema Ruby on Rails. Und wenn Sie Fragen zum Buch haben, sind Sie dort ebenfalls richtig. Ich freue mich auf Ihre, verzeihen Sie mir ein letztes schienenenorientiertes Wortspiel, Bahnpost.

Ruby Kurzreferenz

Die Ruby-Kurzreferenz soll Ihnen einen kompakten Überblick über die Syntax von Ruby geben und Ihnen Nutzungsmöglichkeiten einzelner Methoden aufzeigen.

Folgende Punkte sind grundsätzlich bei der Programmierung mit Ruby zu beachten:

- Anweisungen werden durch Zeilenumbruch oder innerhalb einer Zeile durch Semikolon getrennt
- Bezeichner beginnen mit Buchstabe oder Unterstrich (`_`), danach folgen weitere Buchstaben, Ziffern oder Unterstriche. Groß- und Kleinschreibung werden unterschieden.
- Kommentare: von `#` bis Zeilenende

Ausdrücke

Ein Ausdruck ist eine beliebig komplexe Verknüpfung von Literalen, Variablen und Methodenaufrufen durch Operationen.

Literale

Literalen sind alle *wörtlich* gemeinten Einzelwerte.

Strings

"In doppelten Anführungszeichen werden Escape-Sequenzen und eingebettete Ausdrücke ausgewertet"
'In einfachen Anführungszeichen werden nur \' und \\ umgewandelt.'
%Q(Wie doppelte Anführungszeichen)
%q(Wie einfache Anführungszeichen)

Wichtige Escape-Sequenzen:

- "\n" (Zeilenvorschub, auch allgemeiner Zeilenumbruch),
- "\r" (Wagenrücklauf),
- "\t" (Tabulator)

Eingebettete Ausdrücke werden bei doppelten Anführungszeichen ausgewertet und in den String geschrieben:

"Beliebiger Text #{beliebig komplexer Ausdruck} Text"

Zahlen

- Dezimale Ganzzahlen (Fixnum): -100, -3, 0, 1024
- Beliebig große Ganzzahlen (Bignum): 1000000000000, 5 * 10 ** 100
- Optionales Tausendertrennzeichen _: 1_000_000 # 1000000
- Fließkommazahlen (Float): -3.5678, -2.0, 0.0, 1.4567

Bereiche (Ranges)

Bereiche geben eine Menge an Zahlen oder Buchstaben an, wobei zwischen zwei Range-Typen unterschieden wird:

```
0..4 # 0, 1, 2, 3, 4  
0...4 # 0, 1, 2, 3  
'a'..'d' # "a", "b", "c", "d"  
'a'...'d' # "a", "b", "c"
```

Sonstige Klassen

- true – wahre logische Aussage (TrueClass)
- false – falsche logische Aussage (FalseClass)
- nil – leeres Element (NilClass)
- :sym – Symbol (eindeutiges Element ohne konkreten Wert)

Variablen und Konstanten

Variablen sind Stellvertreter für Objekte. Über den Variablenbezeichner können Objekte auch nach ihrer Erstellung angesprochen werden.

```
variable = Ausdruck
```

Bildungsvorschriften

- Variablennamen beginnen mit Kleinbuchstaben oder Unterstrich: test, _test, _test123
- Globale Variablen, die im gesamten Skript gelten, beginnen mit \$: \$global = "Test"
- Instanzvariablen innerhalb eines Objekts beginnen mit einem @-Zeichen: @text = "Erde"
- Klassenvariablen sind in der Klasse und allen Instanzen verfügbar und beginnen mit zwei @-Zeichen: @@text = "Mars"
- Großbuchstaben kennzeichnen Konstanten, deren Wert sich nach der Definition nicht mehr ändert: G = 9.81, PI = 3.1415926

Arrays

Arrays können in sich mehrere Objekte vereinen. Über einen numerischen Index kann jedes Element angesprochen werden.

```
arr = ['Wert1', 'Wert2', ..., 'WertN']
arr[0] # erstes Element
arr[n] # Element Nr. n-1
arr[-1] # letztes Element
```

Hashes

Hashes ähneln Arrays. Sie besitzen jedoch keine numerischen Index. Der Zugriff auf einzelne Objekte erfolgt durch einen Schlüssel. Hashes bestehen aus *Schlüssel-Wert-Paaren*.

```
hash = {'Schluessel1' => 'Wert1', 'Schluessel2' => 'Wert2', ...}
hash['Schluessel1'] # => 'Wert1'
```

Als Schlüssel kommen meist Objekte der Klassen String oder Symbol zum Einsatz.

Operatoren

Die wichtigsten Operatoren 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)
- && (logisches Und)
- || (logisches Oder)
- ... (Bereich inklusive Endwert), ... (Bereich ohne Endwert)
- ?: (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)

Kontrollstrukturen

Kontrollstrukturen ändern den linearen Programmablauf, in dem sie ihn verzweigen oder Teile mehrfach durchlaufen.

Fallunterscheidungen

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 und führt bei einem positiven Ergebnis den dazugehörigen Quellcode aus.:

```
case var when
Wert1:
    # Anweisungen für Wert1
```

```

Wert2, Wert3:
    # Anweisungen für Wert2 oder Wert3
    ...
else
    # Anweisungen für alle anderen Werte (Default)
end

```

Schleifen

Mit Schleifen können Teile des Quelltextes mehrmals ausgeführt werden.

`while` – führt Anweisungen aus, solange ein Ausdruck true ist:

```

while Ausdruck
    # Schleifenrumpf (Anweisungen)
end

```

Fußgesteuerte Schleife – Sie wird mindestens einmal ausgeführt und prüft erst dann die 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` – führt eine Schleife über Listenelemente durch, beispielsweise über Array- oder Range-Objekte.

```

for var in liste
    # var nimmt nacheinander den Wert jedes Elements an
end

```

Klassen und Objekte

Nahezu alles in Ruby ist ein Objekt. Dementsprechend groß ist der Stellenwert, den die objektorientierte Programmierung in dieser Programmiersprache hat.

Definition einer Klasse

```

class Klassenname
    # Akzessoren-Methoden
    # Konstruktor ...
    # Methoden ...
end

```

Vererbung

```
class Klassenname < Elternklasse
  # Zusätze und Unterschiede programmieren
end
```

Akzessoren-Methoden

Durch Akzessor-Methoden ist es möglich, auf Instanzvariablen von außerhalb eines Objekts lesend und / oder schreibend zuzugreifen.

```
attr_reader :var1, ...      # @var1 ist als var1 von außen lesbar
attr_writer :var1, ...      # @var1 ist als var1 von außen veränderbar
attr_accessor :var1, ...    # @var1 ist als var1 von außen les- und veränderbar
```

Konstruktor

Durch den Konstruktor, eine automatisch als private markierte Methode namens initialize, können Sie Werte bei der Erzeugung einer Instanz übergeben und Instanzvariablen initialisieren.

```
def initialize(var1, var2, ...)
  @instanzvar1 = var1
  ...
end
```

Instanzmethoden

Methoden einer Klasse sind grundsätzlich öffentlich, so dass man bei Instanzen der Klasse auf sie Methode zugreifen kann. Als Rückgabewert wird dabei der zu letzt ausgewertete Ausdruck genutzt. Bei der Nutzung von return in Verbindung mit einem Rückgabewert wird der Durchlauf der Methode abgebrochen.

```
def methodename(var1, ...)
  ...
  # Rückgabewert:
  Ausdruck
  # Sofortiger Abbruch mit Rückgabewert:
  return Ausdruck
end
```

Aufruf von außen:

```
obj = Klassenname.new;
obj.methodename
```

Klassenmethoden

Im Gegensatz zu Instanzmethoden sind Klassenmethoden auch ohne Instanzbildung durch direkten Zugriff auf die Klasse nutzbar.

```
def self.methodename(var1, ...)
  ...
end
```

Aufruf von außen:

Klassenname.methodenname

Ruby-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 setzen sie explizit private).

Array

Liste beliebig vieler beliebiger Objekte mit numerischem Index

Konstruktoraufrufe

- `Array.new []`
- `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 Einzelement
- `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

Konstruktorauftruf

- `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

Konstruktorauftruf

- `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` – Aufrunden auf nächsthöhere Ganzzahl
- `float.floor` – Nachkommastellen abschneiden
- `float.to_i` – Umwandlung in Ganzzahl

Hash

Liste aus Schlüssel-Wert-Paaren

Konstruktoraufrufe

- `hash = Hash.new` # leerer Hash
- `hash = Hash.new(Standardwert)` # leerer Hash
- Implizit: `hash = {Schlüssel1 => Wert1, ...}`

Instanzmethoden

- `hash[Schlüssel]` – Zugriff auf einzelnes Element
- `hash.has_key?(Schlüssel)` – true, wenn Schlüssel existiert, ansonsten false
- `hash.size` – Anzahl der Paare
- `hash.invert` – Schlüssel und Werte vertauschen (Achtung: Datenverlust bei vormals gleichen Werten, nun aber gleichen Schlüsseln!)
- `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

I0

Allgemeine Ein- und Ausgabeklasse

Instanzmethoden

- `io.print(...)` – Text ausgeben
- `io.puts(...)` – Text zeilenweise ausgeben
- `io.printf(format, ...)` – Elemente gemäß `format` ausgeben
- `io.getc` – ein Zeichen einlesen
- `io.gets` – eine Zeile einlesen
- `io.read` – Eingabestrom bis Dateiende (EOF) einlesen

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

String

Beliebige Zeichenkette (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-Variablen 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 enthaltenen Buchstaben in Klein-schreibung um
- `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.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

Konstruktorauftrag

- `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.

Weitere Informationen

- Mehr zur Syntax von Ruby und den eingebauten Klassen erfahren Sie auf <http://www.ruby-doc.org>.

Ruby on Rails-Kurzreferenz

Nachfolgend erhalten Sie einen kompakten Überblick über die Handhabung der wichtigsten Features von Ruby on Rails. Damit bieten Ihnen die folgenden Seiten eine praktische Kurzreferenz für vieles, was Sie in diesem Buch bei *Picsblog* und

TVsendl kennen. Und die versprochenen Befehle für den Fall, dass Sie nicht mit RadRails arbeiten, aber dennoch Controller generieren oder Datenbanktransformationen migrieren möchten, gibt's natürlich auch.

Rails-Applikation erzeugen

Um eine neue Anwendung mit Ruby on Rails zu schreiben, ohne dabei RadRails zu nutzen, wechseln Sie in das gewünschte Projektverzeichnis und geben Sie folgendes in eine Kommandozeile ein:

```
rails anwendungsname
```

Dieser Aufruf erzeugt ein Unterverzeichnis `anwendungsname` und dort eine Verzeichnisstruktur für die Anwendung. Mögliche Optionen sind:

--database=

Spezifiziert das verwendete Datenbanksystem, beispielsweise mysql (Standard), postgresql, sqlite

--ruby-path=

Pfad, in dem sich der Ruby-Interpreter befindet.

Generatoren

Generatoren erzeugen fertigen Quelltext. Sie müssen im Wurzelverzeichnis einer Rails-Applikation wie folgt aufgerufen werden.

```
ruby script/generate model ModelName
```

Ein Model namens `ModelName` wird erzeugt in `app/models/model_name.rb`

```
ruby script/generate controller ControllerName
```

Ein Controller namens `ControllerNameController` wird erzeugt in `app/controllers/controller_name_controller.rb`.

```
ruby script/generate controller ControllerName action1 action2
```

Ein Controller namens `ControllerNameController` wird erzeugt in `app/controllers/controller_name_controller.rb`. Darüber hinaus erhält der Controller zwei leere Actions inklusive Views.

```
ruby script/generate scaffold ModelName ControllerName
```

Erzeugt ein Model namens `ModelName` und einen Controller namens `ControllerNameController`, mit allem, was für eine CRUD-Anwendung benötigt wird, inklusive Views. Damit Scaffolding funktioniert, muss bereits eine zum Model passende Datenbanktabelle existieren, da deren Spaltenbezeichner benötigt werden.

```
ruby script/generate migration MigrationTitel
```

Erzeugt eine Migration in `app/db/migrations/xxx_migration_titel.rb`.

Mögliche Optionen für Generatoren sind:

--pretend

Generator ausführen, ohne Änderungen vorzunehmen

--force

Überschreiben bereits vorhandener Dateien erwünscht

--skip

Bereits existierende Dateien werden beim Generieren übersprungen.

Skripte

Neben Generatoren finden Sie im *script*-Verzeichnis einer Rails-Anwendung weitere Skripte, mit denen Sie weitere Aufgaben erledigen können.

`ruby script/about`

Es werden Informationen angezeigt, die Auskunft über die aktuelle Programmierumgebung geben. Zum Beispiel: Verwendete Ruby-, RubyGems- und Rails-Versionen, Absoluter Pfad der Anwendung

`ruby script/console`

Interaktive Rails-Console, ähnlich `irb`, jedoch im Kontext der Rails-Anwendung.

`ruby script/destroy GeneratorName`

Entfernt den Generator `GeneratorName` aus dem System.

`ruby script/server`

Der Anwendungsserver (hier: Mongrel) wird gestartet. Die Rails-Applikation steht anschließend unter `http://127.0.0.1:3000` zur Verfügung.

Rake

Mit dem Tool *Rake* erledigen Sie Aufgaben, die um Ihre Rails-Anwendung herum anfallen und nur mittelbar mit ihr zu tun haben. Führen Sie `rake` stets im Wurzelverzeichnis Ihrer Rails-Anwendung aus.

`rake db:migrate`

Führt noch anstehende Migrations nacheinander aus. Über `VERSION=` können Sie bestimmen, bis zu welcher Version Ihrer Datenbanktransformationsvorschriften migriert werden soll. `RAILS_ENV=` entscheidet, in welcher Datenbank (*development*, *test* oder *production*) die betreffende Tabelle sich befindet oder befinden soll.

`rake log:clear`

Log-Dateien werden auf 0 Bytes verkleinert, ihr Inhalt geht dabei verloren.

`rake stats`

Gibt statistische Daten zu Ihrer Rails-Anwendung aus, zum Beispiel Anzahl der verwendeten Models oder die Anzahl der Quelltextzeilen in Controllern.

```
rake tmp:cache:clear
```

Löscht den Cache der Rails-Anwendung in *tmp/cache*

```
rake tmp:sessions.clear
```

Löscht alle Dateien in *tmp/sessions*

Migrations

Mit Migrations können Sie Datenbanktabellen beschreiben, erstellen, verändern und löschen, Datensätze erstellen, verändern und löschen, sowie SQL-Statements ausführen.

`create_table` erzeugt eine neue Datenbanktabelle, auf die in einem Block sofort zugriffen werden kann. Dort können mit `column` Felder erstellt werden. Die Art eines Feldes bestimmen Sie dabei durch die Werte `:string`, `:text`, `:integer`, `:float`, `:datetime`, `:timestamp`, `:time`, `:date`, `:binary` und `:boolean`.

```
create_table(:pictures) { |t|
  t.column(:title, :string)
  t.column(:description, :text)
  t.column(:created_at, :datetime)
}
```

Weitere Methoden in Migrations:

```
drop_table(:pictures)
add_column(:pictures, :url, :string)
rename_column(:pictures, :description, :content)
change_column(:pictures, :content, :string)
remove_column(:pictures, :url)
```

Model

Im *Model-View-Controller-Pattern*, welches Ruby on Rails zu Grunde liegt, ist die Model-Schicht dafür vorgesehen, die Programmlogik der Software aufzunehmen. Insbesondere die Kommunikation mit einer Datenbank, das Lesen und Schreiben von Dateien und Berechnungen zählen zu den Hauptaufgaben eines Models. Konventionen und ausgeklügelte Technologien erleichtern das Entwickeln des Models.

Models werden in Rails in *app/models* gespeichert. Der Dateiname besteht aus dem Model-Bezeichner und der Endung *.rb*.

Model-Instanzen erzeugen

Es gibt drei verschiedene Möglichkeiten, eine Modelinstanz zu erzeugen. Das Ergebnis ist entweder ein leeres Objekt, ein Objekt mit Daten, die bei der Erzeugung in der

Datenbank gespeichert werden und ein Objekt mit Daten, die bei der Erzeugung in der Datenbank gespeichert werden, wenn sie dort noch nicht existieren.

```
obj = Book.new
obj = Book.create(:title => 'Tirol', :description => 'Ein Bildband.')
obj = Book.find_or_create_by_title(:title => 'Tirol', :description =>
    'Ein Bildband.')
obj = Book.find_or_create_by_title_and_description(:title => 'Tirol', :
    description => 'Ein Bildband.')
```

Automatische Zuordnungen

Models werden durch Klassen implementiert. Erbt eine Modelklasse von ActiveRecord::Base gelten folgende Zusammenhänge, die das Zusammenspiel zwischen einer Datenbanktabelle und dem Model betreffen.

- Modelklassen spiegeln Datenbanktabellen wieder
- Jede Instanz einer Modelklasse referenziert eine Zeile der Datenbanktabelle
- Akzessoren einer Modelklasseninstanz stellen die Spalten der Datenbanktabelle dar

Diese Zuordnungen zwischen Datenbanktabelle und Model existieren automatisch, wenn

- der Klassenname des Models ein Substantiv in der Einzahl ist
- der Name der Datenbanktabelle die Mehrzahl des Modelnamens ist

Assoziationen

Um Zusammenhänge zwischen mehreren Models und Datenbanktabellen herstellen zu können, stehen Ihnen Assoziationen zur Verfügung, mit denen unterschiedliche Relationen realisiert werden können.

```
belongs_to AnderesModelInEinzahl
has_one AnderesModelInEinzahl
has_many AnderesModelInMehrzahl
has_and_belongs_to_many AnderesModelInMehrzahl
```

Um zwei Models mit has_and_belongs_to_many zu verknüpfen ist eine zusätzliche Datenbanktabelle nötig, deren Bezeichnung sich aus den Namen der zu verknüpfenden Datenbanktabellen zusammensetzt, wobei die Reihenfolge alphabetisch sein muss.

Validierungsmethoden

Mit Validatoren können Sie Regeln für die Daten eines Models definieren, die für das Speichern eines Datensatzes erfüllt werden müssen.

```
validates_presence_of(:title, :url)
validates_presence_of(:name, :on => :update)
validates_uniqueness_of(:username, :case_sensitive => false)
validates_confirmation_of (:email)
```

Datensätze finden

Mit der Klassenmethode `find` eines jeden Models können Sie alle Datensätze eines Models, mehrere Datensätze, die einem Kriterium entsprechen oder einen bestimmten Datensatz ermitteln. Dabei gibt `find` das Ergebnis als Modelinstanz oder als Array von Modelinstanzen zurück.

```
Post.find(1)
Post.find(10,25,67)
Post.find(:first)
Post.find(:first, :order => "date DESC", :offset => 5)
Post.find(:all, :order => "date DESC", :offset => 5, :limit => 10)
```

Controller

Controller steuern im MVC-Pattern von Rails die verfügbaren Models und initiieren die Darstellung von Daten im Browser. Sie basieren auf folgende Grundregeln:

- Controller-Dateien werden als `<controller>.rb` in `app/views/controller` gespeichert. Rails generiert automatisch die Klasse `<ControllerName>Controller`.
- Alle öffentlichen (*public*) Methoden eines Controllers sind *Actions* und können über einen URL der Form `/<controller>/<action>` aufgerufen werden, es sei denn in `config/routes.rb` ist etwas anderes vereinbart.
- Parameter, zum Beispiel der Inhalt eines Formulars oder Bestandteile des *Query-Strings* des URLs, liegen in einem Hash-Objekt vor, das als `params` angeprochen werden kann.
- Alle Instanzvariablen, die in einer Action definiert werden, sind in der View, die die Action rendern soll, verfügbar.
- Standardmäßig rendert eine Action eine View, die den Namen der Action trägt

Nutzen von Views

- Es gibt verschiedene Varianten von Views, die innerhalb einer Action mit der Methode `render` genutzt werden können. Bei `:action`, `:partial` und `:inline` kann das Template `EmbeddedRuby` enthalten.

```
render(:action => :edit) # edit.rhtml
render(:action => :edit, :id => 5, :layout => false) # edit.rhtml ohne Layout
render(:partial => :item, :collection => items) # _item.rhtml für jedes Element in items
render(:partial => :item, :object => data) # _item.rhtml für das Objekt data
render(:text => 'Guten Tag!', :layout =>
  'welcome') # Rendern von Text mit Layout welcome
render(:inline => "Moin, <% @name %>") # Rendern eines Inline-Templates
render(:nothing) # Kein Rendern erforderlich
```

Mit `render(:update)` können Sie Inline-JavaScript mit Rails erzeugen, mit dem Sie den Inhalt der im Browser angezeigten Seite manipulieren können. Die erforderlichen Vorgänge werden als Block angehangen, eine Referenz auf die Webseite liegt als Blockvariable vor.

```
render(:update) { |page|
  page.replace_html(:name, @name)
}
```

Inline RJS

Um mit der `render`-Variante `:update` Inline RJS zu realisieren, stehen neben anderen diese Methoden zur Verfügung. Die folgenden Methoden können auch innerhalb eines RJS-Templates (zum Beispiel `update.rjs`) verwendet werden. Das betreffende Element der Webseite wird über den Wert seines ID-Attributs angesprochen, dass außer bei `insert_html` und `visual_effect` stets an erster Stelle der Parameterliste aufgeführt wird.

```
page.replace_html(:title, @title)
page.insert_html(:bottom, :user_list, @new_user)
page.insert(:top, :charts, :partial => :charts_item, :object => @new_entry)
page.show(:edit_button)
page.hide(:edit_button)
page.remove(:edit_button)
```

Mit `visual_effect` können Sie optische Effekte auf ein HTML-Element anwenden.

```
page.visual_effect(:highlight, "entry_#{@id}")
```

Mit `sortable_element` können Sie eine Gruppe von HTML-Elementen, standardmäßig `li`-Elemente, per Drag and Drop sortierbar machen. Nach jedem Sortiervorgang wird eine Action aufgerufen, der die Sortierreihenfolge übergeben wird.

```
page.sortable(:shopping_list, :url => {:action => sort_shopping_list});
```

Bitte beachten Sie, dass Sie zur Nutzung von RJS und Inline RJS JavaScript-Dateien benötigen, die Sie mit `javascript_include_tag(:defaults)` in den head-Bereich eines Layouts einbinden können.

Session

Um Daten über mehrere Requests festzuhalten, die ob der Natur von HTTP sonst verlorengehen würden, können *Sessions* genutzt werden. Dafür steht das Hash-Objekt `session` zur Verfügung.

```
session[:post_id] = @post.id
@post_id = session[:post_id]
render(:text => 'Login erfolgreich') if session[:user_id]
```

Flash

Mit dem Hash-Objekt `flash` steht Ihnen eine weitere, Session sehr ähnliche Technologie zur Verfügung, die Daten jedoch maximal bis nach dem nächsten Request speichert. Mit `flash.now` werden die Daten bis vor dem nächsten Request aufbewahrt. Besonders gebräuchlich ist `flash` als Speicher für Hinweismeldungen an den Benutzer, konkret im Schlüssel `:notice`.

```
flash[:notice] = 'Virus wurde erfolgreich von der Platte geputzt.'  
flash.now[:notice] = 'Diese Meldung steht nur für diesen Request zur Verfügung.'
```

Views

Mit *Views* sind in Rails verschiedene Arten an Templates gemeint, die durch den Controller oder aus einem View heraus gerendert werden können. Sie werden in `app/views` gespeichert. Sie tragen als Dateiendung `.rhtml` (HTML-Basis), `.rjs` (JavaScript-Basis) und `.rxml` (XML-Basis).

EmbeddedRuby

Innerhalb einer View können Sie mit ERb Ruby-Code platzieren, der beim Rendern ausgeführt wird. Um ERb zu nutzen, muss der Ruby-Code zwischen zwei Tags gesetzt werden.

```
<% Ruby-Code wird ausgeführt %>  
<%= Ruby-Code wird ausgeführt und das Ergebnis ausgegeben %>
```

Die Variante `<%= %>` wird genutzt, um beispielsweise die Rückgabewerte von Helpern oder weitere Templates in die View zu setzen.

```
<% if @sun.shining? %>  
  <%= render(:partial => :sunny) %>  
<% else %>  
  <%= render(:partial => :cloudy) %>  
<% end %>
```

Helper

Ihnen stehen diverse Helper zur Verfügung, die das Schreiben insbesondere von HTML-Views und -Tags mittels Ruby ermöglichen. Bitte beachten Sie, dass alle Helper in Views nur mittels EmbeddedRuby genutzt werden können. Umgeben Sie die Helper also mit `<%= %>`.

```
javascript_include_tag(:defaults)  
stylesheet_link_tag('layout')  
image_tag('rails.png', :alt => 'Rails Logo')  
link_to('Hier klicken!', :controller => 'clicks', :action => 'here')  
link_to('Hier klicken!', :controller => 'clicks', :action => 'here', :confirm =>  
  'Wirklich?')  
form_tag(:action => :change, :id => @post_id)  
text_field(:post, :message)
```

```
text_area(:post, :long_message)
hidden_field(:post, :secret_message)
password_field(:post, :top_secret_message)
datetime_select(:baby, :created_at)
```

Ajax-Helper

Zu `link_to` und `form_tag` gibt es Varianten, die ihre Aufgabe ohne Reload auf Basis von Ajax erledigen können.

```
link_to_remote('Daten anzeigen', :url => {:action => :show_data})
link_to_remote('Daten anzeigen', :url => {:action => :show_data}, :update => :data)
link_to_remote(
  'Daten anzeigen',
  :url => {:action => :show_data},
  :update => :data,
  :loading => 'Element.show('indicator')',
  :complete => 'Element.hide('indicator')
)
form_remote_tag(:url => {:action => :update, :id => @post.id})
```

Die beiden Helper `link_to_remote` und `form_remote_tag` unterscheiden sich in ihren Parametern nur in einem Punkt: Der erste Parameter von `link_to_remote`, der beispielsweise den Linktext beinhaltet, fehlt bei `form_remote_tag`.

Weitere Informationen

Ausführliche Informationen zur Syntax von Ruby on Rails finden Sie im Internet unter <http://api.rubyonrails.org>. Die in Ruby on Rails genutzten JavaScript-Bibliotheken *Prototype* und *Script.aculo.us* finden Sie im Internet unter <http://www.prototypejs.org> beziehungsweise <http://script.aculo.us>.

Konventionen beim Schreiben von Ruby-Code

Es gibt eine geschriebene und einige ungeschriebene Gesetze beim Verfassen von Ruby-Code, die Sie beachten sollten. Während einige Konventionen bindend für die fehlerfreie Ausführung des Quelltextes sind, markieren andere lediglich eine bessere Lesbarkeit. Wichtig sind beide Bereiche.

Kommentare

Kommentare beginnen mit einem Hash-Zeichen (#) und sollten über einen Code-Zeile platziert werden

```
# Grillgut prüfen
@meat = nil if @meat.color == :black
```

Ein Ausdruck pro Zeile

Verwenden Sie stets nur einen Ausdruck pro Zeile. In Ausnahmefällen, die jedoch die Lesbarkeit des Codes verschlechtern, können mehrere Ausdrücke durch Semikolons getrennt werden.

```
# optimal
car.accelerate
speed_camera.take(car)
# auch erlaubt
car.accelerate; speed_camera.take(car)
```

Leerzeichen bei Zuweisungen, Vergleichen und Operatoren

Bei einer Wertzuweisung, sei es an eine Variable oder an einen Hash-Schlüssel, sollte vor und nach = beziehungsweise => jeweils ein Leerzeichen gesetzt werden. Dies gilt auch bei Vergleichsoperatoren wie == oder <=, booleschen Verknüpfungen wie && und Rechenoperatoren.

```
@boss = Person.find(0)
bee[:output] = 50
bird.do(:action => 'chirp')
'Lang lebe der König' if king.dead? == true
a = b + c unless b < c && c +1 == 6
```

Einrückungen

Einrückungen, beispielsweise um den Inhalt eines Blocks optisch leichter wahrnehmen zu können, sollten immer durch zwei Leerzeichen vorgenommen werden, jedoch nicht durch Tabs.

```
children.each { |child|
  child.cry
}
```

Groß- und Kleinschreibung von Bezeichnern

Ruby erkennt anhand der Benutzung von Groß- und Kleinschreibung von Bezeichnern, um welche Art von Objekten es sich jeweils handelt.

```
# Variable
bear = 'Bruno'
# Konstante
BEAR = 'Bruno'
# Symbol
:bear
# Methode
def get_bear
# Klasse
class Bear
# Klasse (Beispiel für CamelCase)
class BearData
```

Blöcke

Blöcke werden einer Methode angehangen. Sie werden durch {} oder begin und end begrenzt. Der Beginn eines Block muss zusammen mit einer oder mehreren eventuell vorhandenen Blockvariablen in einer Zeile liegen. Der Quelltext des Block beginnt in der folgenden Zeile.

```
@girl_friends.each { |cutie|  
  love_letter.send_to(cutie.address)  
}
```

Globale, lokale, Instanz- und Klassenvariablen

Die Schreibweise eines Variablenbezeichners legt seine Sichtbarkeit fest.

```
# global  
$name = 'Paulchen'  
# lokal  
name = 'Fritz'  
# Instanzvariable  
@name = 'Hans'  
# Klassenvariable  
@@name = 'Paula'
```

Struktur einer Klasse

Die Deklaration von Akzessoren, Assoziationen und Validierungsregeln sollten in einer Rails-Klasse stets zu Beginn der Klassendefinition notiert werden.

```
class Tea  
  belongs_to(:drinks)  
  validates_presence_of(:sugar, :spoon)  
  attr_reader(:temperature)  
  ...  
end
```

Quellen im Internet

Das Internet hält viele Websites bereit, die sich mit Ruby beschäftigen. Neben Seiten, die sich rein der Dokumentation von Ruby und Ruby on Rails verschrieben haben, finden Sie auch viele Entwicklerblogs, in denen Sie von praktischen Erfahrungen anderer Entwickler profitieren können. Nutzen Sie die Angebote, um ihr hier erworbene Grundwissen in Sachen Ruby und Ruby on Rails auszubauen.

Ruby Programming Language

Die Website von Ruby erreichen Sie unter <http://www.ruby-lang.org>. Dies ist ein guter Startpunkt für alle, die Ruby noch nicht kennen oder ihr Wissen vertiefen

möchten. Neben dem Download von Ruby und einer umfangreichen Dokumentation erhalten Sie Informationen, an welchen Stellen und wie Sie Hilfe bei Problemen mit der Programmierung mit Ruby bekommen. Außerdem erfahren Sie hier stets Neuigkeiten rund um Ruby.

Ruby Forum

Das Ruby Forum, welches Sie unter der Adresse <http://www.ruby-forum.com> besuchen können, ist wohl das am stärksten frequentierte Forum für Ruby webweit. Es bietet Ihnen auch den Zugang zur sehr beliebten Ruby-Mailingliste.

O'Reilly Ruby

Unter <http://www.oreillynet.com/ruby> finden Sie ständig aktuelle Informationen, Tipps und Anleitungen zu diversen Aspekten der Entwicklung mit Ruby. Aber auch Rails-Themen werden hier behandelt.

[rubyforen.de](http://www.rubyforen.de)

Auf der Website <http://www.rubyforen.de> finden Sie eine der wenigen gut besuchten Ruby-Foren in deutscher Sprache. In der Kategorie Ruby on Rails widmet man sich speziell den Sorgen und Problemen von Rails-Entwicklern.

Ruby on Rails

Hinter <http://www.rubyonrails.org> verbirgt sich die Website von Ruby on Rails. Sie ist für Rails-Entwickler stets ein guter Anlaufpunkt. Die aktuellste Rails-Version und umfangreiche Informationen im Rails Wiki stehen Ihnen hier zur Verfügung.

api.rubyonrails.org

Sollten Sie während der Entwicklung mit Ruby on Rails einmal nicht weiterkommen, sei Ihnen <http://api.rubyonrails.org> wärmstens empfohlen. Hier finden Sie alle Standardmethoden des Rails-Frameworks in einer umfangreichen Dokumentation, sortiert nach Methodenbezeichner oder Klassenzugehörigkeit.

PlanetRubyOnRails

Diese Website, abrufbar unter <http://www.planetrubyonrails.com>, hat es sich zur Aufgabe gemacht, RSS-Feeds von diversen Rails-affinen Websites zu bündeln und gemeinsam zu präsentieren. Das Ergebnis ist ein breitgefächter Themenmix. Schwarz hinterlegte Beiträge deuten dabei auf besonders wertvolle Inhalte hin.

RubyForge

Rubyforge, <http://www.rubyforge.org>, sammelt für Sie freie Ruby- und Rails-Projekte, die Sie nutzen können. Dabei handelt es sich sowohl um komplett Anwendungen als auch Bibliotheken und Erweiterungen. Auch der Anteil an Quelltexten zur Verwendung mit Ruby on Rails ist erfreulich hoch. Insgesamt sind auf RubyForge knapp 3000 Projekte versammelt, unter anderem InstantRails, Rails, Ruby-Gems, Rake und Mongrel.

RadRails

Unter <http://www.radrails.org> finden Sie die Website der IDE, die in diesem Buch zum Einsatz kommt. Hier erfahren Sie stets, ob es etwas Neues gibt in Sachen RadRails. Der Editor, dem ein fulminanter Durchbruch in der Rails-Entwickler-Welt gelang, wird auch in Screencasts vorgestellt.

TextMate

Mac OS Benutzer, die mit Rails entwickeln, schwören auf TextMate (<http://www.macromates.com>) als Editor. Mit seinen Leistungsmerkmalen ist er mittlerweile Vorbild für diverse Nachahmer-Programme, zum Beispiel unter Windows (Intype, <http://www.intype.info>). TextMate ist kostenpflichtig.

Praxiswissen Ruby on Rails

Auf der Website zu diesem Buch finden Sie neben begleitenden Informationen, den Quellcode der hier behandelten Programme ständig viele weitere Links, Tipps und Tricks rund um die Entwicklung mit Ruby on Rails. Außerdem können Sie hier Picsblog und TVsendlr als Live-Demo ausprobieren. Der Webserver hinter der Adresse <http://www.praxiswissen-ruby-on-rails.de> und der Autor dieses Buches erwarten vorfreudig Ihren Besuch.

Ruby on Rails Hosting-Anbieter

Die Anzahl hiesiger Anbieter von Hosting-Lösungen mit Ruby-on-Rails-Unterstützung ist noch überschaubar, während der Markt in Übersee kontinuierlich und mit sinkenden Preisen wächst.

Deutsche, Österreichische und Schweizer Webhoster

- GPcom media – <http://www.gpcom.de>
- heininger web-hosting – <http://www.heininger.at>
- mydotter Webservice – <http://www.mydotter.net>

- Netzallee – <http://www.netzallee.de>
- nine.ch – <http://www.nine.ch>
- null2.net – <http://www.null2.at>
- Railshoster – <http://www.railshoster.de>
- studio78.at Webhosting – <http://www.studio78.at>
- tibit.de Webhosting – <http://www.tibit.de>

US-amerikanische Webhoster

- BlueHost – <http://ruby.bestadvise.info>
- Hosting Rails – <http://www.hostingrails.com>
- Media Temple – <http://www.mediatemple.net>
- Site5 – <http://www.site5.com>
- Speedy Rails – <http://www.speedyrails.com>
- TextDrive – <http://www.textdrive.com>
- VPSLink – <http://www.vpslink.com>

Index

Symbole

! 50
- 30, 62
\$_ 147
% 30
%Q 41
%q 41
%w{} 62
& 62
&& 75
* 30, 62
** 30
+ 30, 38, 62
.. 58
... 58
/ 30
< 74
<- 74
<< 38, 65, 232
= 47, 64, 69
== 74
==== 60
> 74
>= 74
@content_for_layout 195
@session 225
| 62
|| 75

A

abs 36, 149
Action 160
Action Controller 160
Action Mailer 160
Action View 159
Action Web Service 160
action_name 283
Active Record 159
Active Support 160
Addition 30
Ajax 272
Ajax-Helpe 334
Akzessoren 113
ancestors 107
Anführungszeichen 39
AOL Instant Messenger 247
AOL Video Search 246, 247
AOL-Scrennamen 247
Array 61
Arrays 317
Assoziationen 159, 255
attr_accessor 116
attr_reader 115
attr_writer 116
Ausdruck 28
Ausrufungszeichen 50

B

basename 204
begin 59
belongs_to 228
Bereiche 316
Bignum 31, 321
Block 90
Blöcke 336
break 86

C

Capistrano 313
capitalize 45
Captcha 233
case 77, 318
ceil 36
CGI 307
chomp 57
Class 111
class 94
clear 66
close 143
collect 93
column 190
Components 238
concat 38, 64
Console (Rails) 162, 265
Controller 331
count 43
create_table 189
created_at 227
CRUD 161

D

database.yml 170, 182, 309
datetime_select 198
def 99
default 69
default= 69
delete 46, 66, 70, 218
delete_at 66
destroy 217
Developer Key 246
Dir 322
dispatch.fcgi 311
Dispatcher 160
Division 30
DOM 275

downcase 45
downto 84
DRY 157
dump 142
dup 126

E

each 91, 237
each_key 92
each_value 92
each_with_index 141, 291
Eclipse 21
Editor
 Eclipse 21
 FreeRIDE 21
 RadRails 21
 SciTE 21
Einrückungen 335
else 72, 144
elsif 72
EmbeddedRuby 160, 333
Empfänger 104
empty? 66, 69
end 59
Endlosschleife 319
ensure 144
environment.rb 170, 185
Environments 162
ERb 160
error_messages_for 252
eRuby 160
Escape 316
escape 260
e-Schreibweise 32
Exception-Behandlung 144
Exceptions 144
exclude_end? 59
exists? 206

F

FalseClass 42
FastCGI 307
File 143, 218, 322
file_field 198
Filter 224
find 208, 213
find_all 95
Firebug 302

first 59, 64
Fixnum 31, 323
flash 201
flash.now 201
flash.now[:notice] 200
Float 33, 323
floor 36
for 85, 319
form_remote_tag 272
form_tag 197
Format-Filter 259
Framework 152
FreeRIDE 21
Full-Stack-Framework 157
Fußgesteuerte Schleife 319

G

Gem 24
Generatoren 161, 327
german_datetime 235
get_response 264
gets 56, 147
Getter 113

H

h 211
habtm 255
has_and_belongs_to_many 255
has_key? 69
has_many 228
has_many_and_belongs_to_many 228
has_one 228
has_value? 69
Hash 67, 323
Hashes 317
Helper 235, 333
HomerOnDemand 246
HTML 277
 Inline-HTML 277
 Scriptaculous 278
HTTP 258
HTTPResponse 264

I

id2name 51
if 72, 318
if/else 318
image_tag 210

include 131
include? 42, 60, 64
index 43, 64, 70
initialize 111
inspect 113
InstantRails 15
Integer 324
Interactive Ruby 29
IO 324
irb (Interactive Ruby) 29
is_a? 129

J

JavaScript 275
javascript_include_tag 269
JavaScript-Generator 277
join 67

K

keys 71
Klassen
 Array 61
 Bignum 31
 Class 111
 Exception 144
 FalseClass 42
 File 143, 218
 Fixnum 31
 Float 33
 Hash 67
 index 219
 LoginGenerator 221
 Mixins 130
 Module 130
 new 68
 NilClass 44
 Numeric 107
 Range 58
 String 37
 Symbol 51
 TrueClass 42

Klassenattribute 112
Klassenmethoden 112
Kommandozeileninterpreter 29
konkatenieren (Strings) 38
Konstante 50
Konstruktoraufrufe 321

L

last 59, 64
layout 196
length 43, 63, 69
link_to 211
link_to_remote 282
load 142
Locomotive 17
loop 86

M

main 105
map 93
map! 93
Marshal 142
Marshalling 142
Matsumoto, Yukihiro 2
Matz 2
merge 70
merge! 70
Meta-Programmierung 158
Methoden 99
methods 104
Migration 187
mkdir_p 206
mod_fastcgi 307
Model 329
Model View Controller 155
Module 130
 Comparable 131
 Marshal 142
Modulo 30
Mongrel 20
Multiplikation 30
MVC 155
myMusic 246
MySpace 246
MySQL 20
mysql 183

N

Namensräume 133
Net 256
new 62, 111, 143
next 89
nil 44
NilClass 44

now 203
Numeric 107

O

Object 104, 324
object_id 49
Objektrelationales Mapping 157
OOP 5
Operatoren 317
 - 30, 62
 % 30
 & 62
 && 75
 * 30, 38, 62
 ** 30
 + 30, 38, 62
 / 30
 < 74
 <- 74
 << 38, 65
 == 74
 > 74
 >= 74
 | 62
 || 75
 and 75
 or 75
 or 75
original_filename 203
ORM 157

P

paginate 212
pagination_links 212
PaginationHelper 212
Paginator 212
params 200
Partial 209
Plesk 304
pop 65
Potenz 30
print 57
private 120
protected 120
public 120
push 65
puts 55
PuTTY 305

R

RadRails 21, 52
RAILS_ENV 313
RAILS_GEM_VERSION 185
RAILS_ROOT 206
Rails-Console 162, 266
RailsLiveCD 18
Rake 190, 328
rand 36
Range 58
Ranges 316
Receiver 104
redirect_to 200
reject 95
render 200, 235, 276
render_to_string 297
replace_html 277
REpresentational State Transfer 258
Request 160
require 146
rescue 144
respond_to? 94, 104
REST 257
RESTful Rails 259
return 100
reverse 47, 66
rhtml 170
rindex 44, 64
RJS 275
round 36
Router 160
RubyGems 19, 24

S

save 200
scaffold 249
Scaffold-Generator 249
Scaffolding 161, 249
Schleifen 82, 319
Schnittstelle 246
SciTE 21
Scriptaculous
 highlight 278
select 95, 273
Server 304
Sessions 225, 286
Setter 113
shift 65, 70
Sichtbarkeitsbereiche 120

size 43, 63, 69
Skripte 328
slice 45, 64
slice! 66
sort 66, 96
Sortable 285
sortable_element 288
split 62
squeeze 47
SSH 304
step 84
STRATO 303
strftime 235
String 37, 325
stylesheet_link_tag 194
submit_tag 198
Subtraktion 30
Subversion 313
succ 36, 42
super 128
superclass 127
Superklasse 127
SuSE 304
swapcase 45
Symbol 51

T

Tausender trennzeichen 32
text_area 231
text_field 197
Time 203, 326
times 83
to_a 71
to_f 35, 42
to_i 35, 42
to_s 37, 51, 67, 71
to_str 37
Top-Level 105
tr 45
TrueClass 42
Turing 233
TVGaga 246

U

Uncut 268
uniq 65
uniq! 65
unless 73
unshift 65

until 87
upcase 45
Update von Rails 185
update_attributes 216
upto 84
URI 256

V

valid? 204
validates_presence_of 204, 252
validates_uniqueness_of 252
Validierung 204
Validierungsregeln 252
values 71
values_at 64
Variable 48
Variablen 316
Variablennamen 317
Video-Element 263
Views 333
visual_effect 278

W

when 77
while 87
WinSCP 304

X

XML 260
XML Simple 248, 260
xml_in 264
XMLHttpRequest 273

Y

yield 195
YouTube 246
Yukihiro Matsumoto 2

Z

Zahlen 316

Über den Autor

Denny Carl ist seit 2000 als selbständiger Softwareentwickler, Webdesigner und Autor in Berlin tätig. Seitdem entwickelte er eine Vielzahl an Computerspielen, Info- und Edutainmentprodukte und datenbankgestützte Software für kleine bis mittlere Unternehmen. Sein Schwerpunkt liegt jedoch bei der Gestaltung und Entwicklung standardkonformer, dynamischer Webseiten unterschiedlichster Art. Er veröffentlicht regelmäßig Beiträge zu den Themen Webdesign und Webdevelopment in Zeitschriften, Newslettern, bei mehreren Online-Portalen und in seinem Weblog www.devblog.de. In seiner Freizeit genießt er die vielen angenehmen Vorteile eines Wohnsitzes in der deutschen Hauptstadt und dabei jede Minute ohne Tastaturkontakt. Im O'Reilly Verlag hat er bereits das Buch *Praxiswissen Ajax* veröffentlicht.

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 Birk, die Überschriftenschrift ist die Adobe Myriad Condensed, und die Nichtproportionalschrift für Codes ist LucasFont's TheSansMono Condensed.

