Das Ruby-Tutorium

Eine Einführung in Ruby in deutscher Sprache

Marvin Gülker

Webseite des Autors: https://mg.guelker.eu/

Dieses Werk wurde mit LaTeX erstellt und aus der TeX Gyre Pagella und der TeX Gyre Heros gesetzt.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Teil I

Einführung

§ 1 Einleitung

1.1 Über dieses Buch

Dieses Tutorium ist unfertig. Es sollte einmal eine vollständige Einführung in die Programmierung anhand der Programmiersprache Ruby bieten, aber den immensen Zeitaufwand, den dieses Projekt erforderte, konnte ich irgendwann nicht mehr stemmen. Schon die Arbeit am Text allein war sehr zeitaufwendig, zudem blieb auch die Entwicklung der Programmiersprache Ruby selbst nicht stehen. Planungen für dieses Tutorium gab es mindestens schon seit 2011, ab spätestens 2012 setzte die Niederschrift ein, und seit dieser Zeit gab es erhebliche Änderungen an der Sprache: so wurden etwa Refinements eingeführt und Schlüsselwort-Argumente. Da ich Ruby nicht mehr in dem Maße einsetze wie es früher der Fall war, stieg der Aufwand zur Arbeit am Tutorium weiter. Schließlich war klar: es kann nicht zu Ende geführt werden. Daher habe ich mich entschieden, den letzten Stand dieses Tutoriums, der sich etwa auf das Jahr 2018 datieren läßt, der Öffentlichkeit unter einer freien Lizenz (CC-BY-SA) zur Verfügung zu stellen in der Hoffnung, daß jemand anderes die begonnene Arbeit fortführen und beenden kann. Die CC-BY-SA-Lizenzierung sollte das jedermann erlauben, aber ich mache besonders darauf aufmerksam, daß jeder Nutzer nach dieser Lizenz klarstellen muß, daß und welche Änderungen er gemacht hat. Ich möchte nicht ohne meine Zustimmung als Unterstützer eines von mir nicht mehr kontrollierten Werks dastehen. Dafür kann etwa der folgende Hinweis im Impressum eingesetzt werden:

> Dieses Werk basiert auf Arbeiten von Marvin Gülker aus den Jahren 2011-2018. Herr Gülker hat nichts mit den seitdem eingefügten Änderungen zu tun.

Aus dem Copyright-Vermerk (©-Zeile) im Impressum ist mein Name zu entfernen, diese Einleitung ist umzuschreiben.

Die Quellen dieses Dokuments sind in LaTeX abgefaßt und wurden jedenfalls zum Teil mit Blick auf eine Weiterverarbeitung mit dem Programm la-

texml geschrieben, sodaß neben der PDF- auch eine HTML-Version hätte entstehen können, hätte latexml nicht Probleme mit den verwendeten Tikz-Graphiken gehabt. So blieb letztlich auch die HTML-Version des Tutoriums unvollendet und neben den LaTeX-Quellen steht allein die PDF-Version zur Verfügung. Sie kann aus den Quellen wie folgt erstellt werden:

```
$ lualatex main.tex
$ biber main
$ lualatex main.tex
$ lualatex main.tex
```

Die Quellen sind auf LuaLaTeX abgestimmt. Andere Implementationen von LaTeX werden aller Voraussicht nach nicht funktionieren. Ich habe die Quellen zuletzt erfolgreich mit der in Debian 10 enthaltenen Software übersetzen können.

Das Buch ist aufgeteilt in mehrere Teile. Der erste Teil enthält Grundwissen, welches für das Verständnis des gesamten weiteren Buches essenziell ist. Er behandelt die Grundstrukturen von Computern und Programmen und führt in Form einer Art Crashkurs einmal quer durch die Programmiersprache Ruby, um dem Anfänger die grundlegenden Werkzeuge an die Hand zu geben. Im zweiten Teil werden dann einzelne wichtige Konzepte der Programmierung im Allgemeinen und mit Ruby im Besonderen vorgestellt, die nicht zwingend in der vorgegebenen Reihenfolge erarbeitet werden müssen. Wo nötig, weisen Querverweise den Weg zu vorausgesetzten Kenntnissen, die an anderer Stelle behandelt werden. Der dritte Teil schließlich führt in die Abhängigkeitenverwaltung mit RubyGems ein und stellt anhand ausgewählter Problematiken wichtige Software vor, die man als Ruby-Programmierer kennen sollte.

Ich erhebe keinen Anspruch auf Vollständigkeit. Es gibt zu viele Eventualitäten, die abzudecken den Rahmen jeder Abhandlung sprengen würde. Vielmehr soll das Tutorium den Leser in die Lage versetzen, ihm noch unbekannte Problematiken auf Basis des erarbeiteten Grundlagenwissens zu verstehen und zu recherchieren, sodaß er sie in Ruby schließlich umsetzen können sollte.

Das Tutorium setzt keine Kenntnisse in der Programmierung voraus, sehr wohl aber fundierte Kenntnisse im Umgang mit Computern. Es wird vorausgsetzt, dass der Leser mit den einschlägigen Konzepten und Begrifflichkeiten ordentlicher Nutzung von PCs vertraut sind; Erklärungen darüber, was ein Verzeichnisbaum oder ein Betriebssystem ist, würden den Rahmen des Tutoriums sprengen.

1.2 Über die Programmiersprache Ruby

Ruby ist eine objektorientierte Programmiersprache, die 1995 von dem Japaner *Yukihiro »Matz« Matsumoto* entwickelt wurde. Sie wird seitdem von einem »Core Team« genannten engen Zirkel kontinuierlich entwickelt und konnte 2013 mit Ruby 2 am 20. Geburtstag der Sprache ihren Existenzanspruch untermauern. Ihre erste Hochphase erlebte die Sprache mit dem Webframework RubyOnRails, das 2004 von *David H. Hansson* entwickelt wurde und sich bis heute anhaltender Beliebtheit erfreut, auch wenn es von anderen, vor allem JavaScript-basierten Technologien etwas in den Hintergrund gedrängt wurde.

Matsumoto-san arbeitet noch immer an der Sprache und ist darüber hinaus an einer Neu-Implementation der Sprache für Einbettungszwecke, mruby[1], beteiligt. Nach seiner Aussage soll Ruby vor allem »natürlich, nicht einfach« sein; wichtiges Design-Ziel der Sprache ist die nach wie vor unerreichte und fast intuitive Leserlichkeit der Programmiersprache für den Betrachter[**TODO: Nachweis https://www.ruby-lang.org/de/about/**]. Tafel 1.1 gibt eine Übersicht über die Geschichte von Ruby[**TODO: Nachweis http://blog.nicksieger.com/articles/2006/10/20/rubyconf-history-of-ruby/**].

Ruby wurde 2012 in Version 1.8 von der ISO standardisiert.

1.3 In diesem Buch genutzte Konventionen

Dieses Tutorium verwendet einige Konventionen, die nachfolgend erläutert werden.

- In `nichtproportionaler Schrift` gehaltene einzelne Wörter stellen Quelltext, Befehle für die Kommandozeile oder Dateinamen dar.
- Mit »…« werden in Quelltext-Beispielen bewußte Auslassungen gekennzeichnet, die sinngemäß mit Quelltext befüllt gelesen werden sollen. Ihre Erläuterung würde nur vom diskutierten Thema ablenken.
- In Groteskschrift werden Namen von Programmen und Programmbibliotheken genannt.
- Namen natürlicher Personen sind *kursiv*.

[1] `https://mruby.org/`

Datum	Bemerkung
24.02.1993	Erstidee zur Sprache von *Matsumoto-san* und *Keiju-san*.
21.12.1995	Erste öffentliche Version 0.95.
25.12.1196	Erste stabile Version 1.0.
15.03.1998	Start des Ruby Application Archive (RAA).
Dezember 1998	Eröffnung der Ruby-Talk-Mailingliste.
15.12.2001	Die »Pickaxe«, das erste nicht-japanische englische Buch über Ruby, erscheint.
04.08.2003	Version 1.8.0 erscheint; als 1.8.6 wohl die am längsten eingesetzte Ruby-Version.
13.12.2005	Veröffentlichung von RubyOnRails.
2006	Die erste »RubyKaigi« findet statt, heute wichtigste Ruby-Konferenz.
25.12.2007	Version 1.9.0 erscheint. Diese Version brach die Rückwärtskompatbilität erheblich.
April 2012	ISO-Standard für Ruby.
24.02.2013	Version 2.0.0 erscheint.
25.12.2017	Version 2.5.0 erscheint.

Tabelle 1.1: Meilensteine von Ruby

- Komponenten in Dateinamen werden für alle Betriebssysteme einheitlich mit / getrennt, auch wenn unter Windows typischerweise ein Backslash \ benutzt wird. Windows versteht aber auch den »normalen« Schrägstrich /.

- Das Zeichen »~« (Tilde) bezeichnet das Hauptverzeichnis des aktuellen Benutzers. Dies ist `/home/(nutzername)/` unter unixoiden Betriebssystemen und `C:/Users/(Nutzername)` unter Windows. Während unixoide Systeme dieses Zeichen korrekt verarbeiten, kennt Windows es nicht; dort muß der Pfad ausgeschrieben werden.

- Das Zeichen »$« bezeichnet den Shell-Prompt, d.i. die Ansammlung von Zeichen, die die Kommandozeile jedes Mal am Anfang der Zeile automatisch ausgibt, wenn man einen neuen Befehl eingeben will. Die Kommandozeilen unixoider Betriebssysteme beenden diesen Prompt typischerweise mit einem Dollar-Zeichen, weshalb ich mich für diese Konvention entschieden hat. Das Zeichen ist deshalb *nicht* miteinzugeben, sondern soll nur symbolisieren, daß hier ein Befehl auf der Kommandozeile eingegeben werden soll.

§ 2 Installation und erster Kontakt

Bevor es losgehen kann, ist eine Installation der Programmiersprache Ruby und eines Editors erforderlich. Ruby steht für alle gängigen Betriebssysteme zur Verfügung, allerdings überwiegt der Anteil der Nutzer unixoider Systeme gegenüber Windows. Ruby ist eine Programmiersprache, die üblicherweise nicht für GUI-Anwendungen herangezogen wird (obwohl das möglich ist, dazu **[TODO: Quververweis]**), sondern meist auf Servern eingesetzt wird. Server ihrerseits werden meist nicht mit Windows, sondern mit einem unixoiden Betriebssystem betrieben (vgl. Abb. 2.1). Eine ernsthafte Arbeit mit Ruby außerhalb von Nischen wie etwa mruby ist wegen des unix-zentrischen Ökosystems nur auf einem unixoiden Betriebssystem möglich. Das soll keinesfalls heißen, daß Windows zum Ausprobieren ungeeignet ist, ganz im Gegenteil. Es soll nur von vornherein klarstellen, daß eine langfristige Beziehung zur Programmiersprache Ruby kaum an einem unixoiden Betriebssystem vorbeiführt. Linux und MacOS sind deshalb schon von Haus aus gut für die Entwicklung mit Ruby geeignet, seit Windows 10 steht dank der Einführung des neuen Windows Subsystem for Linux (WSL)[1] auch Windows-Nutzern ein einfacher Weg zur Nutzung unixoider Umgebungen zur Verfügung. Ruby kann aber auch — in Grenzen — ohne das WSL direkt auf Windows betrieben werden.

2.1 Installation von Ruby

2.1.1 Ruby unter Linux

Die Installation von Ruby läuft unter den gängigen Linux-Distributionen heute unproblatisch ab. Man kann es einfach über die Paketverwaltung der Distribution installieren, das Paket heißt üblicherweise schlicht »ruby«. Auf Debian-basierten Distributionen sollte man das Metapaket »ruby-full« installieren, welches alle relevanten Einzelpakete nach sich zieht.

[1] `https://docs.microsoft.com/en-us/windows/wsl/about`

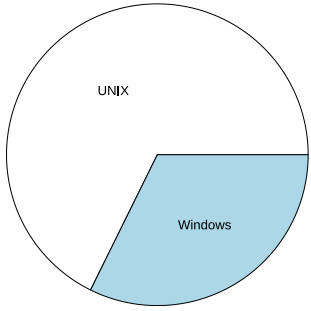

Abbildung 2.1: Betriebssysteme auf Servern, Stand 2015

2.1.2 Ruby unter MacOS

Unter Mac OS X ist Ruby bereits in einer Version vorinstalliert, die vielleicht nicht mehr ganz aktuell ist, für die Zwecke dieses Tutorials aber genügen sollte. Hier gibt es daher nichts zu tun.

2.1.3 Ruby unter Windows (ohne WSL)

Unter Windows ist, wenn nicht das WSL genutzt wird, Ruby mithilfe des RubyInstallers[2] zu installieren. Das auf dessen Webseite ebenfalls angebotene »Development Kit« wird für die Zwecke dieses Tutoriums nicht benötigt; es ist nur relevant im Zusammenhang mi C-Erweiterungen für Ruby.

> Man achte bei der Installation unbedingt darauf, unbedingt das Häkchen bei der Option »Add Ruby to the PATH environment variable« zu setzen. Dies sort dafür, dass Ihnen Ruby auf der Kommandozeile zur Verfügung stehen wird.

2.1.4 Ruby unter Windows (mit WSL)

Das WSL stellt eine Laufzeitumgebung für eine Linux-Distribution unter Windows zur Verfügung. Nach der Aktivierung des WSL wie auf dessen Webseite beschrieben[3] kann man deshalb so verfahren, als hätte man ein »echtes« Linux vor sich (→ 2.1.1).

[2] `http://rubyinstaller.org/`
[3] `https://docs.microsoft.com/en-us/windows/wsl/install-win10`

2.2 Installation des Editors

Das wichtigste Werkzeug des Programmierers ist sein Editor, mit welchem er den Quelltext bearbeitet. Bei Quelltext handelt es sich um gewöhnliche Reintext-Dokumente, sodass Sie zwar jeden beliebigen Editor verwenden können, es empfiehlt sich aber, bereits frühzeitig mit einem vernünftigen Programm zu arbeiten, denn spätestens dann, wenn die Programme länger als zwanzig Zeilen werden, wird ein Editor ohne syntaktische Kolorierung (sog. *Syntaxhighlighting*) schlicht unbenutzbar. Von weniger luxuriösen Dingen wie automatischer Einrückung einmal ganz abgesehen.

Ein gesonderter Hinweis erscheint an dieser Stelle angebracht. Quelltexte sind *Reintext-Dokumente*, keine Office-Dateien. Es ist weder möglich noch empfehlenswert, sie mit einem Office-Programm wie Word zu bearbeiten. Wer unbedingt bei der Windows-Standardausstattung bleiben will, muß stattdessen auf Notepad (dt. »Editor«) zurückgreifen.

Für den Anfang empfiehlt sich ein leichtgewichtiger, aber schon speziell auf Programmierung ausgelegter Editor wie SciTE[4], er für alle größeren Betriebssysteme zur Verfügung steht. Er bietet alle wesentlichen Funktionen eines Programmiereditors, ist aber sehr spartanisch und wird mit zunehmender Kenntnis von Programmierung unpraktisch erscheinen. Dann wird es Zeit, sich mit einem der großen Editoren auseinanderzusetzen:

- Emacs[5] ist der vom Autor bevorzugte Editor.
- Vim[6] ist der andere große Editor.

Daneben gibt es noch diverse mehr oder weniger bekannte Editoren, darunter der neuerdings sehr populäre Atom[7] oder der unter Windows-Nutzern sehr beliebte Notepad++[8]. Von schwergewichtigen sogenannten »Entwicklungsumgebungen« (*Integrated Development Environment, IDE*) wird für den Moment abgeraten. Derartige IDEs verstecken zu viel hinter vermeindlich »einfachen« Benutzeroberflächen und beeinträchtigen so das Verständnis des

[4] `http://www.scintilla.org/SciTE.html`
[5] `https://www.gnu.org/software/emacs/`
[6] `https://www.vim.org`
[7] `https://atom.io/`
[8] `https://notepad-plus-plus.org/`

Sprache. Wer erst einmal ein grundlegendes Verständnis für die Programmiersprache erworben hat, mag sich erneut mit diesem Thema auseinandersetzen.

Im Weiteren Verlauf dieses Tutoriums wird von der Nutzung von *SciTE* ausgegangen. Das Programm muß zunächst konfiguriert werden. Im Menü Options → Change Indentations Settings sind die Tabweite und die Einrückungstiefe von 8 auf 2 zu ändern und der Haken bei »Use Tabs« zu entfernen. Das ist zwar nicht strikt notwendig, allerdings entspricht es den über Jahre entwickelten Konventionen, daß Ruby-Programme mit 2 Leerzeichen und nicht mit Tabs eingerückt werden, wenn es erforderlich wird. Die Konventionen in anderen Programmiersprachen sind anders, aber darum soll es hier nicht gehen.

2.3 Das erste Programm

Nach Abschluß der Installation steht auf der Kommandozeile des Betriebssystems der Befehl `ruby` zur Verfügung (unter Windows kann die Kommandozeile über die Tastenkombination Win+R gefolgt von der Eingabe von »cmd« aufgerufen werden). Folgendes Kommando sollte nunmehr möglich sein:

```
$ ruby -v
```

Das Ergebnis sollte so oder ähnlich aussehen:

```
ruby 2.3.3p222 (2016-11-21) [x86_64-linux-gnu]
```

Erscheint stattdessen die Meldung »Kommando nicht gefunden« oder etwas ähnliches, so ist aller Wahrscheinlichkeit nach Ruby entweder nicht richtig installiert, oder die Umgebungsvariable PATH falsch gesetzt. In diesem Fall ist die Installation erneut zu überprüfen.

Für die Zwecke dieses Tutoriums empfiehlt es sich, alle erstellten Ruby-Programme in einem eigenen Verzeichnis abzulegen, etwa `~/rubytutorial` auf einem unixoiden oder `C:/rubytutorial` auf einem Windows-System. Ruby-Quelldateien enden üblicherweise mit der Dateiendung `.rb`, wobei es sich allerdings nur um eine Konvention handelt — Ruby selbst beachtet die Datei-Endung nicht.

Der Brauch verlangt, in eine neue Programmiersprache mit dem sogenannten Hallo-Welt-Programm einzuführen. Dabei handelt es sich um das einfachstmögliche Programm in einer beliebigen Programmiersprache, das nichts

weiter tut, als den Text »Hallo Welt!« auf der Kommandozeile auszugeben. Man öffne daher nun seinen Editor (etwa *SciTE*) und schreibe:

```
puts "Hallo Welt!"
```

Die Datei sollte als `hello.rb` im genannten Verzeichnis für dieses Tutorium abgespeichert werden (unter *SciTE* kann das über File → Save As erfolgen). Das Programm kann nun auf der Kommandozeile ausgeführt werden, indem nach einem Wechsel in das richtige Verzeichnis (mithilfe von `cd`) der Name der Programmdatei dem Befehl `ruby` übergeben wird:

```
$ cd rubytutorial
$ ruby hallo.rb
Hallo Welt!
```

Wie man aus diesem Beispiel ersehen kann, werden Ruby-Programme üblicherweise über die Kommandozeile gestartet, zwingend ist das aber nicht. Speziell in *SciTE* kann man auch einfach die F5-Taste drücken; das gelingt aber nur für einfache Programme, die keine Eingabe erfordern. Für diejenigen, die mit der Kommandozeile noch nicht vertraut sind, bietet Anhang **[TODO: Querverweis]** eine minimale Einführung in die Bedienung.

Es ist auch möglich, unter Windows die Dateiendung `.rb` mit dem Befehl `ruby` derart zu verknüpfen, daß beim Doppelklick auf eine solche Datei sogleich ein Kommandozeilenfenster geöffnet wird. Wenn Ruby mithilfe des Ruby-Installers installiert wurde (→ 2.1.3), dann sollte dies sogar standardmäßig der Fall sein; ausprobieren lohnt sich. So gestartete Programme schließen das Konsolenfenster aber sogleich wieder, wenn das Programm fertig durchgelaufen ist. Um dies zu verhindern, kann dem Programm der Befehl `gets` angehängt werden, der nach einer Eingabe verlangt:

```
puts "Hallo Welt!"
gets
```

Das Programm wartet bei Erreichen des Befehls `gets` dann darauf, daß der Nutzer die Eingabetaste drückt. So wird ein selbsttätiges Schließen des Fensters vermieden.

Es erscheint sachgerecht, ein Wort über den Nutzen der Kommandozeile zu verlieren. Speziell Windows-Nutzer halten die Kommandozeile oftmals

für ein Werkzeug aus der lang zurückliegenden Vergangenheit. Das ist unzutreffend. Richtig ist, daß die Kommandozeile, die lange Zeit auf Windows-Systemen ausgeliefert wurde und wird (CMD), sich seit DOS-Zeiten in der Tat nicht weiterentwickelt hat und unangenehm in der Bedienung ist. Microsoft hat aber in den letzten Jahren dazugelernt und liefert seit Windows 10 mit der Powershell eine gut benutzbare Kommandozeile aus, deren Nutzung für komplexere Aufgaben gegenüber CMD empfohlen werden kann; für unixoide Systeme war eine gute Kommandozeile schon immer ein integraler Bestandteil. Eine moderne Kommandozeile ist ein sehr nützliches Werkzeug, welches es erlaubt, langwierige Klickstrecken in graphischen Nutzeroberflächen durch einen einzigen, prägnanten Befehl zu ersetzen. Ist sie mit guten Werkzeugen versehen, so können durch die richtige Verkettung von Befehlen komplexe Aufgaben leicht automatisiert werden, ohne jedes Mal gleich zu einer komplexen Programmiersprache wie Ruby greifen zu müssen. Auf Servern, die typischerweise ein unixoides Betriebssystem haben, findet man sogar meist gar keine graphische Benutzeroberfläche vor, sondern verbindet sich mit Werkzeugen wie ssh und erhält dann Zugriff auf (nur) die Kommandozeile des Servers.

2.4 Interactive Ruby

Zusammen mit Ruby wird ein sehr hilfreiches Programm namens »interactive ruby«, kurz IRB, installiert. Dabei handelt es sich um ein Programm, welches eine Zeile Ruby-Code einliest, sofort auführt, und das Ergebnis anzeigt. Ist man sich im Unklaren über die Funktionsweise bestimmter Ausdrücke, gibt es nichts besseres, als sie in Ruhe in der IRB zu analysieren. Auch Experten benutzen die IRB noch häufig, was eindrucksvoll ihre Brauchbarkeit beweisen sollte.

Die IRB ist ein Kommandozeilenprogramm und wird durch Eingabe des Befehls `irb` gestartet. Auf jede Zeile Ruby-Code, die eingegeben wird, zeigt die IRB sogleich das Ergebnis an. Eine Beispielsitzung könnte so aussehen:

```
$ irb
irb(main):001:0> 1 + 2
=> 3
irb(main):002:0> "Hallo".gsub("l", "x")
=> "Haxxo"
irb(main):003:0> puts "Test"
```

```
Test
=> nil
irb(main):004:0> x = "Hallo"
=> "Hallo"
irb(main):005:0> y = x.gsub("l", "x")
=> "Haxxo"
irb(main):006:0> puts y
Haxxo
=> nil
irb(main):007:0> exit
```

`exit` beendet die IRB wieder.

Die IRB sollte man immer verwenden, wenn man über Code stolpert, den man nicht versteht. Es tauchen auf den Community-Seiten (→ 2.5) immer wieder Fragen auf, die leicht hätten selbst beantwortet werden können, wenn der Fragesteller die IRB bemüht hätte; dementsprechend genervt fallen oft die Antworten auf solche Anfragen aus, wenn es denn überhaupt Antworten gibt. Erst, wenn auch eine Analyse des Codes in der IRB nicht mehr ausreicht, sollte man sich an die Gemeinschaft wenden.

2.5 Hilfe, Neuigkeiten, Dokumentation

Dieses Tutorium versucht, alle für Anfänger relevanten Fragestellungen zu erklären, kann aber niemals vollständig sein und ist vielleicht nicht an allen Stellen immer gleich verständlich. Bleiben Fragen offen oder ist sonst Hilfe erforderlich, ist die erste Anlaufstelle die deutschsprachige Ruby-Mailingliste »ruby-de«. Deren Webseite kann unter `https://lists.ruby-lang.org/cgi-bin/mailman/listinfo/ruby-de` erreicht werden. Eine Mailingliste ist ein über E-Mail geführtes, öffentliches Diskussionsmedium, bei dem eine E-Mail an die Adresse der Mailingliste (in diesem Fall ruby-de@ruby-lang.org) an alle Abonennten der Mailingliste weitergeleitet wird. Die Antwort erfolgt dann wieder an die Adresse der Mailingliste, sodaß jeder Abonennt sie erhält. »ruby-de« wird öffentlich archiviert[9], eine ausführliche Benutzungsanleitung

[9] `https://lists.ruby-lang.org/pipermail/ruby-de/`

existiert ebenfalls[10]. Zur Teilnahme ist eine Einschreibung in die Liste erforderlich, die über die genannte Webseite erreicht werden kann.

»ruby-de« ist eine deutschsprachige Mailingliste. Daneben gibt es die internationale Mailingliste »Ruby-Talk«[11], die vom Prinzip her genauso funktioniert. Als ältestes Austauschmedium zur Programmiersprache Ruby ist dort die Anzahl der Teilnehmer erheblich größer als bei »ruby-de«, allerdings wird die Diskussion auch nur auf Englisch geführt.

Offizielle Neuigkeiten rund um die Programmiersprache Ruby können unter `https://www.ruby-lang.org/` abgerufen und abonniert werden.

Ruby besitzt auch eine Referenzdokumentation. Diese steht unter `https://ruby-doc.org/` für praktisch alle Versionen der Sprache zur Verfügung und sollte immer dann herangezogen werden, wenn man mehr über einen bestimmten Aspekt der Sprache wissen möchte oder sich einen Überblick über bestimmte Funktionalitäten verschaffen will. Die Referenzdokumentation beschreibt alle verfügbaren Elemente der Sprache und ist damit weit vollständiger, als eine Einführung wie dieses Tutorium es jemals sein könnte. In der praktischen Programmierung kann man auf die Referenzdokumentation schlechterdings nicht verzichten. Wo tunlich, verlinkt die HTML-Version dieses Tutoriums auf die Referenzdokumentation.

[10] `http://files.guelker.eu/misc/ruby-de.html`
[11] `https://www.ruby-lang.org/de/community/mailing-lists/`

§ 3 Theoretischer Hintergrund

Wer noch nie programmiert hat, hat möglicherweise eine fehlerhafte Vorstellung von Natur und Aufbau von Computern und der Ausführung von Programmen. Dieses Kapitel soll einige Grundfragen klären, die zwar für die Arbeit mit Ruby nicht als solches unmittelbar relevant sind, die jedoch zum Allgemeinwissen jedes Programmiers zählen (sollten). Ihre Kenntnis ermöglicht es, besser an der Diskussion anderer Programmierer teilnehmen zu können und hilft, die »Denkweise« von Computern zu verstehen.

3.1 Das Verhältnis von Hardware zu Software

3.1.1 Was ist ein Computer?

Ein Computer ist eine Maschine, die Befehle von einem Medium ausliest und anhand dieser Befehle beliebige Berechnungen durchführen kann. Die Möglichkeit, daß eine solche Maschine existieren muß, hat der Brite *Alan Turing* 1936 nachgewiesen, indem er ein theoretisches Konzept einführte, das man heute nach ihm »Turing-Maschine« nennt. Wird etwas »turing-vollständig« genannt, dann heißt dies, daß die so bezeichnete Einrichtung eine Turing-Maschine implementiert und deshalb jedes beliebige logische Problem zu lösen vermag.

Heutige Computer gehen zurück auf die Arbeit um *John von Neumann*, nach der eine Maschine zur Durchführung beliebiger Rechnungen fünf Komponenten aufweisen muß: das Speicherwerk, das Steuerwerk, das Rechenwerk, das Eingabe- und das Ausgabewerk. Man bezeichnet diese Konstruktionsweise heute als »von-Neumann-Architektur« und die in heutigen Computern verbauten Prozessoren (CPU) und Arbeitsspeicher (RAM) sind Weiterentwicklungen dieser Komponenten. Ihre Funktionsweise prägt Programme noch immer, weshalb es sinnvoll ist, ihre grundlegende Funktionsweise zu verstehen. Der *von Neumann*'sche Rechner arbeitet in einem Zyklus, der wie folgt abläuft:

1. Das Steuerwerk liest den nächsten Befehls aus dem Speicherwerk,

2. es dekodiert ihn,
3. liest danach die Befehlsparameter aus dem Speicherwerk,
4. läßt den Befehl dann durch das Rechenwerk ausführen und
5. schreibt das Ergebnis der Operation zurück in das Speicherwerk.

Danach wird wieder bei 1 begonnen, es sei denn, es gibt keine weiteren Befehle mehr. Der *von Neumann*'sche Rechner arbeitet also sequenziell, indem er ein Programm Befehl für Befehl nacheinander abarbeitet. Neuere Prozessoren besitzen meist mehrere Kerne, jedoch ändert sich dadurch hier am Grundprinzip nur wenig. Jeder Kern arbeitet für sich weiterhin sequenziell einzelne Befehle ab, sodaß maximal nur so viele Befehle echt parallel ausgeführt werden können, wie der Prozessor Kerne hat.

Computer sind heute überall. Neben dem klassischen PC oder Laptop sind Smartphones Computer, Server und Industriemaschinen, aber auch Überwachungskameras und Waschmaschinen enterhalten Computer ebenso wie Videorekorder und sogar Türklingeln und Lampen. Im Zeitalter des »Internets der Dinge« (engl. »Internet of Things, IoT«) geht der Trend dahin, jeden Alltagsgegenstand mit Computern (und einer Internetverbindung, was Sicherheitsprobleme nach sich zieht) auszustatten und ihm so neue Funktionen zu geben. Das ist es, was hinter dem Schlagwort von »smarten« Gegenständen steckt.

Entgegen anderslautenden Marketing-Bestrebungen hat sich am Grundkonzept »*von-Neumann*-Computer« auch 2018 in Zeiten von Mobile Devices, IoT und Cloud Computing nichts geändert. Am Ende jedes dieser Konzepte steht immer noch ein »normaler« *von Neumann*'scher Computer. Der Cloud-Host ist auch nur ein Server, auf dem virtuelle Maschinen laufen, im Smartphone steckt ein nach den *von Neumann*'schen Prinzipien aufgebauter ARM-Prozessor und für IoT-Geräte gilt nichts anderes. Erfreulicherweise bedeutet das für Programmierer, das die einmal gelernten Konzepte überall weiterhin Anwendung finden.

3.1.2 Was ist ein (Computer-)Programm?

Ein Computerprogramm ist eine Folge von Befehlen, die nach Aufnahme in einem maschinenlesbaren Träger bewirkt, daß ein Computer eine bestimmte Aufgabe oder ein Ergebnis anzeigt, ausführt oder erzielt [**TODO: Nachweis**].

So die rechtstechnische Definition. Pragmatischer kann man sagen, daß alles, was man auf einem Computer ausführen kann, ein Computerprogramm ist, denn wenigstens die Ausführung einer Aufgabe ist integraler Bestandteil jedes Computerprogramms. Das gilt vom Betriebssystem über die Waschmaschinensteuerung bis hin zum Webbrowser. *Keine* Computerprogramme sind bloße Daten, wie z. B. Musikdateien. Man spricht insoweit von »Software«, obwohl manche den Begriff auch synonym mit dem des Computerprogramms verwenden.

»Computerprogramm« ist ein rechtlicher Begriff. Er stammt aus dem Urheber- und Patentrecht und wird dann benutzt, wenn man über die rechtlichen Aspekte von Code sprechen möchte, insbesondere in der Diskussion mit Juristen. Bewegt man sich unter Technikern, genügt es in aller Regel, schlicht von »Programm« zu sprechen. Gemeint ist dasselbe. Diese Tutorium verwendet der Kürze halber außerhalb dieses Abschnitts nur den Begriff »Programm«.

3.1.3 Algorithmen

Während ein Programm konkrete Anweisungen an einen Computer enthält, stellt ein Algorithmus die abstrakte Vorgehensweise dar. Er ist ein rein theoretisches Konstrukt und wird üblicherweise in normaler Sprache oder in mathematischen Formeln beschrieben. Man kann daher sagen, daß ein Programm die konkrete, ausführbare Umsetzung eines Algorithmus darstellt.

Die Erforschung von spezialisierten Algorithmen stellt ein eigenes Gebiet in der theoretischen Informatik dar und ist nicht trivial. Nicht umsonst werden um — vor allem in den U. S. A. — erbitterte Rechtsstreitigkeiten um die Rechte an Videokodierungsalgorithmen geführt[**TODO: Nachweis**]. Abseits dieser theoretischen Ebene liegt aber auch jedem noch so einfachen Programm ein Algorithmus zugrunde, was der Eigenart von Computern als Turing-Maschinen geschuldet ist. Man kann insoweit zwischen universell anwendbaren Algorithmen und solchen, die nur der Lösung der konkret gestellten Aufgabe dienen, unterscheiden. Erstere zu finden erfordert einen weitaus höheren kognitiven Aufwand.

Am Unterschied zwischen Algorithmus und Programm läßt sich übrigens auch der Unterschied zwischen einem Entwickler (»Software-Architekt«) und einem Programmierer studieren. Während der Entwickler meist studierter Informatiker ist und zunächst ein theoretisches Konzept entwirft, ist der Programmierer für die Umsetzung des Konzepts in ein konkretes Programm verantwortlich. Beide Aufgaben haben ganz unterschiedliche Anforderungen und erfordern für eine qualitativ hochwertige Umsetzung entsprechende Erfah-

rung auf dem Gebiet. Laien vermischen die Konzepte meist, was dann in unsinnige politische Forderungen wie »Programmieren ab der 1. Klasse« mündet. Die Entwicklung von Software ist vor allem vom logischen Denken bestimmt, während ihre Programmierung idealerweise gut wartbaren, wiederverwertbaren sauberen und dokumentierten Quellcode hervorbringen sollte, der keine Fehler aufweisen sollte. Gute Entwickler wie auch gute Programmierer verstehen zumindest auch die Grundzüge des jeweils anderen Gebiets und sicher gibt es Leute, die beides hervorragend beherrschen. An der grundlegenden Unterscheidung ändert das nichts. Es käme ja auch niemand auf die Idee, Architekten und Bauarbeiter gleich zu behandeln und wegen eines Baubooms jedem Erstklässler Maurerstunden zu geben.

3.1.4 Maschinencode und Programmiersprachen

Jeder Prozessor besitzt eine Prozessorarchitektur, deren gängigste zurzeit die »x86_64«-Architektur ist; so gut wie jeder PC wird heute in »x86_64« ausgeführt. Andere Architekturen finden sich vornehmlich im Mobilbereich; dort nimmt »ARM« eine marktbeherrschende Stellung ein. Dennoch gibt es daneben eine Vielzahl weiterer Prozessorarchitekturen, sodaß man keinesfalls davon ausgehen sollte, dass die Welt sich in diesen beiden erschöpft.

Prozessorarchitekturen definieren einen festen Satz an Instruktionen. Alle Prozessoren einer Architektur unterstützen einen gemeinsamen Satz an Instruktionen, und einige Prozessoren darüber hinaus weitere spezifisch auf diesen Prozessor zugeschnittene Instruktionen. Alle Anweisungen, die ein Computer ausführen soll, müssen letztlich auf diese prozessorspezifischen Instruktionen heruntergebrochen werden, damit der Prozessor weiß, was er zu tun hat. Ein solchermaßen übersetztes Programm bezeichnet man als *Objektcode* oder *Maschinencode*.

Prozessoren verstehen keine allgemeinsprachlichen Anweisungen; die einzelnen Instruktionen werden letztlich numerisch kodiert. Ersetzt man die numerischen Befehlsnummern durch menschenlesbare Befehle, erhält man sogenannten *Assemblercode*. »Menschenlesbar« ist in diesem Kontext ein Euphemismus; Maschinencode ist für den durchschnittlichen Programmierer auch in Assembler-Form nicht zu entziffern, geschweige denn, daß die übergreifende Logik gedanklich erfaßt werden könnte. Die Arbeit mit Assemblercode ist ausgesprochen mühselig, und weil sich die verfügbaren Assemblersprachen mit jeder Prozessorgeneration verändern, haben in Assembler geschriebene Programme oftmals eine recht kurze Lebensdauer. Heute gibt es außerhalb sehr spezifischer Domänen (etwa Betriebssystem-Kernels) kaum noch

Anwendungsbereiche, in denen aktiv in Assembler programmiert wird. Vielmehr rekurriert man auf sogenannte *höhere Programmiersprachen*, die deshalb so genannt werden, weil sie anders als Assembler a) nicht mehr so »nahe« an der Hardware stehen und b) Konstruktionen ermöglichen, die von den unzulänglichen Fähigkeit von Assemblercode, der meist nur sehr generisch ist, abstrahieren und in diesem Sinne von ihm »weiter weg« sind. Ruby ist eine solche höhere Programmiersprache, wie eigentlich fast alle heute in Gebrauch befindlichen Programmiersprachen. Entgegen verbreiteter Meinung ist auch C nach dieser Definition eine höhere Programmiersprache. Die Übersetzung der höheren Sprache in Maschinencode nennt man *Kompilation*, und das Ergebnis ist ein *Kompilat*; handelt es sich um ein eigenständig ausführbares Programm, spricht man von einer *Binary Executable* oder auch nur kurz *Executable*.

Kompilate waren jahrelang die übliche Art und Weise, Programme an den Nutzer auszuliefern. Der Programmierer erstellte sein Programm, kompilierte es, und lieferte dem Nutzer respektive Kunden die fertige Executable. Auch heute noch wird jeder Windows-Nutzer ».exe«-Dateien kennen. Dabei handelt es sich um ebendiese Kompilate, also in Maschinensprache übersetzte Programme. Spätestens seit JavaScript haben jedoch neben diesen kompilierten Sprachen interpretierte Sprachen einen gleichberechtigten Platz gefunden. Diese Sprachen übersetzen nicht mehr das gesamte Programm im Voraus in Assemblercode, sondern erst auf dem Rechner des Nutzers (was bei diesem selbtverständlich eine Installation der betreffenden Programmiersprache voraussetzt), woraus sich eine hohe Flexibilität für den Programmierer und auch für den Nutzer ergibt. Auch Ruby ist eine interpretierte Sprache.

Bei der Interpretation wird das fertige Programm nicht mehr unverändert von der Festplatte in den Arbeitsspeicher kopiert. Vielmehr wird es zunächst vom Interpreter der Programmiersprache bearbeitet und »live« in Maschinencode übersetzt. Dieser Maschinencode liegt dann nur noch im Arbeitsspeicher vor, kann aber ansonsten genauso wie bei einer traditionell kompilierten Programmiersprache durch Übergabe der ersten Adresse des Maschinencodes im Speicher an den Prozessor weitergegeben werden.

3.1.5 Das Betriebssystem

Nach dem bereits Gesagten ist das Betriebssystem eines Computers ebenfalls nur ein Programm und nicht Hardware. Es ist jedoch insofern besonders, als daß alle Programme letztlich unter dem Betriebssystem ausgeführt werden. Nur unter außergewöhnlichen Umständen kommt man in die Lage, ohne ein

Betriebssystem arbeiten zu müssen — insbesondere dann, wenn man den Code eines Betriebssystems selbst bearbeitet. Diese Grenzfälle gehören nicht zum alltäglich erforderlichen Wissen und sollen hier daher nicht tiefer behandelt werden.

Aufgabe des Betriebssystems ist u. a. die Koordination sämtlicher anderen Programme, deren Start und Ende, die Zuteilung von Arbeitsspeicher und sonstigen Hardware-Ressourcen. Das Betriebssystem bestimmt, welches Programm in welchen Teilen an den Prozessor geschickt wird und es legt fest, wie Eingaben vom Nutzer oder aus anderer Hardware zu erhalten sind. Unter Einsatz von Treibern macht es verschiedene Hardware-Komponenten unter einer einheitlichen Schnittstelle verfügbar. Kein Teil des Betriebssystems im strengen Sinne sind graphische Benutzeroberflächen, Webbrowser und sonst alles, was ein typischer Nutzer vor sich sieht; dabei handelt sich lediglich um eine Vorauswahl von Software, die der Vertreibende für nützlich hielt und die (wie alle Programme) auf die Funktionen des eigentlichen Betriebssystem zurückgreifen. Im besonderen Falle von Windows macht Microsoft allerdings den Vertrieblern recht strenge Vorgaben, sodass Windows-Systeme sich in der Regel sehr ähnlich sehen. Apple vertreibt die Hardware für sein Betriebssystem MacOS sogar ausschließlich selbst, sodaß das Unternehmen volle Kontrolle auch über die mit dem Betriebssystem ausgelieferten Standardprogramme ausübt. Gänzlich anders dagegen im Linux-Bereich: die hinter dem Kernel stehenden Entwickler machen keine Vorgaben darüber, welche Software mit auszuliefern ist, was zusammen mit der liberalen Lizenzpolitik und den damit verbundenen geringen Kosten in der Folge dazu geführt hat, dass auf fast jedem Computer Linux läuft (mit der prominenten Ausnahme von Endnutzer-PCs, die vornehmlich historisch bedingt ist). Telefone, Kühlschränke, Waschmaschinen, Server, Supercomputer, Rechencluster: das, und viel mehr, läuft zum überwiegenden Teil mit Linux.

Startet man ein Programm, so weist man in Wirklichkeit das Betriebssystem an, den zur jeweiligen Datei auf der Festplatte gehörigen Maschinencode in den Arbeitsspeicher zu laden und an den Prozessor zu schicken. Das Betriebssystem entscheidet daher, welche Speicherbereiche einem Programm zugeordnet werden und hat die Oberhoheit darüber, wann einzelne Instruktionen des Programms an den Prozessor verschickt werden; wie genau das abläuft, kann vom Nutzer oftmals konfiguriert werden. Jedenfalls wird es nur selten vorkommen, dass ein Programm den ganzen Prozessor für sich allein beansprucht, wie es die Reinvorstellung eines Von-Neumann-Systems wäre. Es wäre unmöglich, mehrere Programme gleichzeitig zu starten, sofern man kein Mehrkern-System einsetzt. Webbrowser und Textverarbeitung würden

sich gegenseitig ausschließen. Schlimmer noch: da ja auch das Betriebssystem selbst ein Programm ist, könnte man eigentlich gar keine Programme starten, ohne das Betriebssystem »auszuschalten«. Ohne Betriebssystem gäbe es aber etwa keine graphische Bildschirmausgabe. Daher organisieren Betriebssysteme die laufenden Programme (ein laufendes Programm bezeichnet man als *Prozeß*) so, daß sie ihnen die Ressourcen der CPU wechselweise zuteilen: erst läuft Prozeß A, dann Prozeß B, dann C, dann wieder B, dann C, dann A, u. s. w. Die genaue Reihenfolge unterliegt der Einschätzung des Betriebssystems und kann durch Zuweisung von Prioritäten an Prozesse beeinflußt werden. Ganz nach Belieben schickt das Betriebssystem dann nur einen Teil der Instruktionen von Programm A an den Prozessor, bevor es sich einem anderen Programm zuwendet. Das ist für die Ausführung des Programms nicht schädlich, da das Betriebssystem, wenn es sich das nächste Mal A zuwendet, dort weitermacht, wo es aufgehört hat. Aus Sicht des Programms entsteht folglich ein kontinuierlicher Ausführungsfluß, und das sogar dann, wenn das Betriebssystem den Prozeß gänzlich suspendiert (etwa, wenn der Rechner in den Ruhezustand geschickt wird). Die Zeit, die ein Prozeß tatsächlich auf der CPU läuft, nennt man *Prozessorzeit* oder *CPU-Zeit*.

Neben der CPU-Zeit ist das Betriebssystem auch für sämtliche anderen Ressourcen verantwortlich, die einem Programm zugeordnet werden, insbesondere für den Arbeitsspeicher. Neben dem eigentlichen Maschinencode, der beim Programmstart wie beschrieben in den Arbeitsspeicher geladen werden muß, wird ein Prozeß noch weitere Speicher-Resourcen benötigen: Ergebnisse mathematischer Berechnungen oder Nutzereingaben etwa werden ebenfalls im Arbeitsspeicher abgelegt. All diesen zusätzlichen Speicheranforderungen muß das Betriebssystem auch nachkommen, oder, bei Speichermangel, sie verweigern und den gierigen Prozeß vielleicht beenden, indem es dessen weitere Ausführung aufgibt. Nachdem ein Prozeß, gleich ob durch reguläres Ende durch Abschluß des letzten Befehls oder durch »Gewalt«, beendet wurde, gibt das Betriebssystem die vom ehemaligen Prozeß belegten Ressourcen wieder frei, sodaß sie anderen Prozessen zur Verfügung stehen. Daher wird es oftmals vorkommen, daß derselbe Bereich des Arbeitsspeichers im Laufe einer PC-Sitzung von zahlreichen Programmen benutzt wird.

3.2 Die Verarbeitung von Ruby-Programmen

3.2.1 Sprachgrammatik

Die Beschreibung der Von-Neumann-Architektur (→ 3.1.1) und die Ausführungen über Maschinencode (→ 3.1.4), auf den letztlich alles hinausläuft, stellen hoffentlich klar, daß Computer per se keine verstandesbegabten Wesen sind und ein Programm nicht »verstehen« können. Vielmehr bedarf es der Compiler und Interpreter, um höhere Programmiersprachen in Instruktionssätze zu übersetzen, die dann an den Prozessor geschickt werden können. Da es sich bei Compilern und Interpretern aber auch nur um Programme handelt, die den allgemeinen Regeln für Programme unterliegen, muss die Eingabe für diese Software klar strukturierten und unverletzlichen Regeln genügen. (Normale) Programme können nur mit »deterministischen« (vorhersehbaren) Eingaben umgehen, da der Programmierer eines Compilers bzw. Interpreters nur in der Lage ist, eine endliche Anzahl von Fällen zu behandeln. Je klarer strukturiert die Programmiersprache, desto einfacher die Aufgabe, einen Compiler oder Interpreter für diese Programmiersprache zu schreiben. Daher ist allen Programmiersprachen gemeinsam, dass sie eine feste *Grammatik* oder *Syntax* aufweisen, an die der Quelltext sich zu halten hat, ansonsten eine Kompilation bzw. Interpretation scheitert. Die Grammatik von Ruby ist erfreulicherweise sehr liberal, sodass ein Ruby-Programm beinahe wie ein paar Sätze aussieht, die man auch ohne tiefergehende Programmierkenntnisse verstehen kann.

```
if x > 3
  puts "x ist größer drei."
end
```

Jede Verletzung der Grammatik verursacht in Ruby einen »SyntaxError« bei der Interpretation.

```
if alias error
  puts-> "fehler"
end

# => syntax error, unexpected keyword_alias
# => syntax error, unexpected ->, expecting end-of-input
```

Ruby besitzt eine ganze Reihe dieser syntaktischen Regeln, die mit dem Ziel möglichst guter Menschenlesbarkeit designed wurden. Das heißt nicht, dass man Ruby-Code wie Prosa schreiben könnte; wie gezeigt macht eine Verletzung der Syntax das Programm unbrauchbar. Sie sind jedoch bei näherer Betrachtung meist — nicht immer — logisch und leicht verständlich, und mit der Zeit gewöhnt man sich an sie, was in Ruby aufgrund der Design-Ziele der Sprache einfacher geht als in anderen Sprachen. Matz selbst hat diesbezüglich gesagt, er wolle die Sprache »natürlich« machen, und nicht »einfach«[1].

Die Theorie der formalen Sprachen ist schwierig, sie soll hier nicht Thema sein. Vielmehr sei dem Leser nahegelegt, sich bestehenden Code anzusehen (etwa die zahlreichen Codeschnippsel in diesem Tutorium) und dessen Struktur nachzuahmen. Über die formalmathematischen Hintergründe von Programmiersprachen (engl. »Programming Language Theory, PLT«) muss man weder als Einsteiger, noch als fortgeschrittener Programmierer Bescheid wissen. Es handelt sich dabei um Kenntnisse, die man sich später optional zulegen kann; die Erfahrung aber zeigt, dass das Gespür, welches man sich im Laufe der Zeit bei der Arbeit mit Programmen ganz von allein zulegt, die formalen Aspekte recht gut erfaßt, ohne, daß man jemals etwas von mathematischen Alphabeten, Symbolen und Worten gehört hätte. Gute Editoren sorgen durch farbliche Hervorhebung für eine gute Sichtbarkeit der syntaktisch bedeutsamenen Elemente einer Programmiersprache.

Tafel 3.1 enthält eine Auflistung aller Schlüsselwörter der Programmiersprache Ruby. Diese stehen aufgrund ihrer hervorgehobenen syntaktischen Bedeutsamkeit für eigene Bezeichner nicht zur Verfügung; eine fehlerhafte Verwendung verletzt die Grammatik der Programmiersprache Ruby.

3.2.2 Ablauf der Ausführung

Wird der Ruby-Interpreter aufgefordert, ein Programm auszuführen, durchläuft er tatsächlich drei Stufen. Zunächst wird der der Quelltext in seine Einzelteile zerlegt, dann wird er in ein leichter lesbaren Zwischenformat überführt und schließlich als Maschinencode auf der CPU ausgeführt.

1. Parsing. Wie erwähnt wird der Quelltext eingelesen und sodann anhand von Ruby formaler Grammatik in seine Einzelteile zerlegt. Durch diesen als Parsing bezeichneten Prozess ergibt sich aufgrund der möglichen Verschachtelungen der einzelnen Ausdrücke eine Baumstruktur, der sogenannte *Abstract Syntax Tree* (engl. »Abstrakter Syntaxbaum«),

[1] `https://www.ruby-lang.org/de/about/`, abgerufen am 12.06.2018.

alias	do	module	then
and	else	next	undef
begin	elsif	not	unless
BEGIN	end	or	until
break	END	redo	when
case	ensure	rescue	while
class	for	retry	yield
def	if	return	__END__
defined?	in	super	

Tabelle 3.1: Schlüsselwörter in Ruby

Literal	Bedeutung
%	String wie doppelte Anführungszeichen
%i	Array aus Symbolen
%Q	String wie doppelte Anführungszeichen
%q	String wie einfache Anführungszeichen
%r	Regulärer Ausdruck
%s	Symbol wie doppelte Anführungszeichen
%w	Array aus Strings
%x	Shell-Ausführung wie Backticks

Tabelle 3.2: %-Literale

kurz AST. Ruby enthält eine Programmbibliothek namens »ripper«, mit deren Hilfe man den AST visualisieren kann. Das ist im täglichen Gebrauch meist nicht notwendig, kann aber bei obskuren Konstrukten das Verständnis erleichtern. Dieses Ruby-Programm:

```
str = "Hello"
puts(str.gsub("l", "x"))
```

Wird von Ruby in den folgenden AST zerlegt:

```
[:program,
 [[:assign,
   [:var_field, [:@ident, "str", [1, 0]]],
   [:string_literal, [:string_content,
                      [:@tstring_content,
                        "Hello", [1, 7]]]]],
  [:method_add_arg,
   [:fcall, [:@ident, "puts", [2, 0]]],
   [:arg_paren,
    [:args_add_block,
     [[:method_add_arg,
       [:call,
        [:var_ref, [:@ident, "str", [2, 5]]],
        :".",
        [:@ident, "gsub", [2, 9]]],
       [:arg_paren,
        [:args_add_block,
         [[:string_literal,
           [:string_content, [:@tstring_content,
                              "l", [2, 15]]]],
          [:string_literal,
           [:string_content, [:@tstring_content,
                              "x", [2, 20]]]]],
         false]]]],
     false]]]]
```

2. Bytecode. Der erzeugte AST wird von Ruby anschließend in ein temporäres Zwischenformat überführt, den Bytecode. Dabei handelt es sich

um ein nicht mehr menschenlesbares Format, das jedoch im Gegensatz zum ursprünglichen Quelltext erheblich leichter zu verarbeiten und auszuführen ist. Die Erzeugung von Bytecode als Zwischenschritt zwischen dem Parsing und der Ausführung wurde erst mit der Version 1.9.0 in die Sprache Ruby eingeführt und diente vornehmlich der Erhöhung der Performanz der Sprache. In der Praxis spielt der Bytecode keine Rolle, er ist für den Programmierer wie den Nutzer vollkommen transparent. Der Vollständigkeit halber sei das obige Programm hier noch einmal als menschenlesbar annotierter Bytecode angeführt (tatsächlich besteht der Bytecode selbst nur aus Bytes ohne menschenlesbare Namen):

```
local table (size: 2, argc: 0 [opts: 0, rest: -1, post: 0,
  block: -1, kw: -1@-1, kwrest: -1])
[ 2] str
0000 trace            1                  (   1)
0002 putstring        "Hello"
0004 setlocal_OP__WC__0 2
0006 trace            1                  (   2)
0008 putself
0009 getlocal_OP__WC__0 2
0011 putstring        "l"
0013 putstring        "x"
0015 opt_send_without_block
  <callinfo!mid:gsub, argc:2, ARGS_SIMPLE>
0017 opt_send_without_block
  <callinfo!mid:puts, argc:1, FCALL|ARGS_SIMPLE>
0019 leave
```

In anderen Sprachen, etwa Python oder Elisp, kann ein Programm in Bytecode überführt und für später gespeichert werden, sodass ein späteres Ausführen des Programms möglich ist und aufgrund der einfacheren Verarbeitbarkeit des Bytecodes im Vergleich zum ursprünglichen Quelltext schneller vonstatten geht. Ruby unterstützt diese Möglichkeit (noch) nicht.

3. Ausführung. Der Bytecode ist system- und prozessorunabhängig, es handelt sich um ein internes Format des Ruby-Interpreters. Wie aber

schon im Abschnitt über Maschinencode (→ 3.1.4) beschrieben, bedingt die Ausführung von Code notwendigerweise die Überführung in prozessorabhängige Instruktionen. Dies ist der letzte Schritt. Für jede der unterstützten Betriebssysteme und Prozessorarchitekturen besitzt der Ruby-Interpreter eine Zuordnung von Bytecode-Befehlen in Prozessorinstruktionen). Er geht die einzelnen Befehle des Bytecodes durch und übersetzt jeden einzelnen in eine oder mehrere prozessorabhängige Anweisungen und sendet diese an die CPU, welche sie dann ausführt. Es gibt dabei im wesentlichen zwei Ansätze: Die *Ahead-of-time-Kompilierung*, und die *Just-In-Time-Kompilierung* (JIT). Der Unterschied besteht im Zeitpunkt, zu dem die Instruktionen an den Prozessor geschickt werden: Während im ersten Falle zunächst das gesamte Programm übersetzt und dann das Ergebnis als prozessorspezifischer Codeblob abgeschickt wird, wird im letzteren Falle bereits mit dem Absenden der prozessorspezifischen Anweisungen begonnen, bevor die Übersetzung vollständig ist, was in einem erheblichen Geschwindigkeitsvorteil bei der Ausführung resultiert. Es hat lange Zeit immer wieder Ansätze für eine JIT-Kompilierung in Ruby gegeben, aber die finale Implementation hat erst mit Ruby 2.6 in die Sprache Eingang gehalten.

3.3 Der Ablauf der Programmentwicklung

Programme entstehen nicht aus dem Nichts. Man öffnet nicht seinen Editor und schreibt irgendwelchen Code. Programme jeglicher Art haben immer den Zweck, ein Problem zu lösen, genauer: die Lösung für ein Problem (Algorithmus, → 3.1.3) in eine für den Computer verständliche Form zu bringen. Am Anfang jeglicher Software-Entwicklung steht daher die genaue Identifikation des zu lösenden Problems. Je genauer man definiert, was geschuldet ist, desto präziser kann man am Ende das Programm entwickeln.

Ist das Problem identifiziert, muß eine Lösung erarbeitet werden. Das lösungsorientierte, logische Denken ist der Schlüssel zur Software-Entwicklung überhaupt. Es ist dabei von zentraler Bedeutung, die logischen Strukturen und Abfolgen der Problemlösung zu erkennen. Da das Ausgangsproblem außerordentlich kompliziert und umfangreich sein kann (Beispiel: Chat-Client), ist es unerläßlich, es in handhabbare Unterprobleme zu zerlegen. Diese Unterprobleme führt man dann einer schrittweisen Lösung zu. Insgesamt entsteht bei diesem Vorgehen ein programmiersprachenunabhängiger Algorithmus, den man dann im letzten Schritt in ein Programm in einer bestimmten Pro-

grammiersprache umsetzt.

Dieser Dreischritt — Problemidentifizierung, Problemlösung durch Algorithmus, Umsetzung des Algorithmus — ist eine Idealvorstellung, die praktisch kaum jemals so durchgehalten werden kann und sollte. Die Aufspaltung von Algorithmusentwicklung und Programmierung ist oft genug eher hinderlich als förderlich, weil sie Zeit raubt. Für Anfänger ist die Durchführung jedes Schrittes einzeln empfehlenswert, weil sie das logische Denken schult, in der Praxis dagegen ist die theoretische Beschreibung und Lösung jedes Trivialproblems zeitraubende, unnötige Förmelei. Gewisse Probleme löst man einfach »bei der Arbeit« am Code. Manche Probleme sind aber selbst als handhabbares Unterproblem so komplex, daß eine Entwicklung direkt am Code zu minderwertigen oder gar dysfunktionalen Ergebnissen führt. In diesen Fällen sollte man sich unbedingt auf den Zwischenschritt der rein logischen Problemlösung in Form eines Algorithmus besinnen. Dies hilft auch, sein Denken von den Beschränkungen der gerade eingesetzten Programmiersprache zu befreien.

Auf der zweiten und der dritten Stufe gibt es Hilfsmittel, die die Entwicklung vereinfachen können. Einige der Hilfsmittel der zweiten Stufe werden nachfolgend vorgestellt. Es erscheint dabei der Hinweis angebracht, daß diese Hilfsmittel einander nicht ausschließen, sondern miteinander kombiniert werden können und sollen. Für alle aufgeführten Hilfsmittel gilt zudem, daß sie besonderer Software nicht bedürfen. Vielmehr lassen sich sich ganz altmodisch mit Bleistift und Papier einsetzen, wovor auch der erfahrene Programmierer nicht zurückschreckt.

- Ein gerade für Anfänger empfehlenswertes, aber auch von erfahrenen Programmierern je nach Problem nicht verschmähtes Hilfsmittel der Algorithmusplanung ist sog. **„Pseudocode“**. Dabei wird die Logik als Quellcode in einer hypothetischen, nicht real existierenden Programmiersprache niedergeschrieben, die möglichst genau zu den eigenen Gedankengängen paßt. Typischer Pseudocode sieht so aus:

  ```
  WENN die Eingabe länger als 5 Zeichen ist DANN
  gib einen Fehler aus und versuche erneut
  SONST
  Addiere alle Eingaben
  ```

  Pseudocode hat den Vorteil, daß man unabhängig von den Beschränkungen der eingesetzten Programmiersprache überlegen kann. Gerade

für Anfänger, die oft die Syntax der Programmiersprache noch nicht gemeistert haben, ist dies ein maßgeblicher Aspekt. Erst, wenn man den logischen Aufbau abgeschlossen hat, beschäftigt man sich dann damit, wie man ihn in der gewünschten Programmiersprache formuliert.

- Pseudocode ist mehr ein mentales als ein formales Hilfsmittel. Mit **mathematischen Formeln** steht ein streng formales Hilfsmittel zur Verfügung, mit dem die reine Logik abschließend und präzise erfaßt werden kann. Mancher hält die Mathematik gar für die natürliche Beschreibungssprache für Algorithmen[1, S. 1]. Ähnlich wie bei Pseudocode ist es bei Einsatz dieser Technik nicht erforderlich, sich mit den Einzelheiten der Implementierung zu befassen[Vgl. 1, S. 2]. Anders als Pseudocode zwingt die mathematische Erfassung aber dazu, wirklich jeden Aspekt logisch zu durchdenken und bietet daher eine höhere Gewähr für Richtigkeit. Allerdings liegt diese Arbeitsweise nicht jedem und setzt voraus, daß man sich in der Sprache der Mathematik sicher bewegen kann. Je stärker das zu lösende Problem einen mathematischen Schwerpunkt aufweist, desto eher lohnt sich aber der Blick auf diese Technik. Beispielsweise ließe sich ein Prüfsummenalgorithmus wohl am besten zunächst mathematisch beschreiben, bevor man ihn in Code umsetzt.

  Werden Formeln in Code umgesetzt, so empfiehlt sich deren Dokumentation. Sie erleichtert das Verständnis und die spätere Bearbeitung des Programms.

- **Flußdiagramme** dienen dazu, die Verschachtelung von Bedingungen und Wiederholungen graphisch aufzubereiten. Das Kapitel über Kontrollstrukturen (→ 5) stellt einige Flußdiagramme vor, die als Beispiel für eigene Überlegungen dienen können.

- **UML-Diagramme** sind ein Weg, um die Verhältnisse in objektorientierten Programmen (→ 6) darzustellen. Je mehr Klassen und Objekte in einer Problemlösung eine Rolle spielen, desto sinnvoller ist es, sich vor der Umsetzung in Quelltext einen Überblick über die verschiedenen involvierten Klassen zu machen.

Hilfsmittel der dritten Stufe sind typischerweise unterstützender Natur und hängen notwendigerweise von der eingesetzten Programmiersprache ab. Hierunter fallen etwa Code-Generatoren und Auto-Vervollständigung sowie überhaupt alle Werkzeuge, die die „Arbeit am Code“ komfortabler gestalten. Es

gibt ganze Sammlungen dieser Werkzeuge, die dann unter dem Begriff „integrierte Entwicklungsumgebung“ (engl. „integrated development environment“, kurz IDE) erfaßt werden. Anfänger sollten von diesen Werkzeugen Abstand halten. Sie bieten dem erfahrenen Programmierer Vereinfachungen, führen aber neue Fehlerquellen ein und stehen dem Verständnis von Programmentwicklung im Wege. Erst, wer sich in einer Programmiersprache und ihrer Umgebung sicher fühlt, kann sich diese Werkzeuge sinnvoll zu Nutze machen.

[**TODO: Weiterführende Literatur angeben**]

Die vorstehenden Erwägungen beziehen sich auf den einzelnen Programmierer. Arbeiten mehrere Programmierer an einem Projekt, gibt es eine Vielzahl von Vorschlägen dafür, wie man das Entwickler-Team bestmöglich organisiert. Sie reichen vom berüchtigten Wasserfall-Modell bis zu sog. „agilen“ Entwicklungsmethoden wie Scrum. Ihre Darstellung würde den Rahmen dieses Buches sprengen; zu all diesen Entwicklungsmodellen gibt es eigene Bücher, auf die an dieser Stelle verwiesen werden muß. Den Anfänger braucht dies nicht zu kümmern, denn es handelt sich der Sache nach gar nicht um ein Problem bei der Software-Entwicklung, sondern um ein soziologisches Problem in der Projekt- bzw. Unternehmensorganisation.

[**TODO: Weiterführende Literatur angeben**]

Ein ebenfalls nicht zu unterschätzender praktischer Aspekt ist der Einsatz sogenannter **Versionskontrollsysteme**. Mit ihnen wird erfaßt, wer wann was warum am Quelltext eines Programms geändert hat. Professionelle Programmentwicklung ist heute ohne Versionskontrolle nicht mehr denkbar. Sie ermöglicht etwa, die genaue Änderung festzustellen, die ein Fehlverhalten erstmals eingeführt hat oder nachzuhalten, wer Rechte am Code hält. Wer mehr sein möchte als ein Gelegenheitsprogrammierer, wird sich früher oder später mit dem Thema auseinandersetzen müssen. Das derzeit meistgenutzte Versionskontrollsystem ist mit hoher Wahrscheinlichkeit Git. Hierzu findet sich im Internet umfassende Literatur, sodaß die Aufnahme einer Einführung in Git in dieses Tutorium entbehrlich erschien[Hinzuweisen ist insbesondere auf 2].

[**TODO: Literaturbox?**]

Empfohlen werden kann jedem Anfänger dagegen die frühzeitige Teilnahme an quelloffenen Projekten. Viele bekannte Programme wie der Webbrowser Firefox, das Zeichenprogramm GIMP oder in diesem Fall natürlich die Programmiersprache Ruby selbst sind quelloffene Software. Ihr Quelltext ist frei im Internet verfügbar und ein Blick hinein kann nicht schaden. Wer Freude an einem Projekt findet, sollte eine Teilnahme erwägen, denn: die wenigsten Projekte weisen Mithilfe zurück. Wer allerdings gerade erst anfängt zu

programmieren, sollte sich mental darauf vorbereiten, auch mit Kritik umgehen zu müssen. Kritikfähigkeit ist ebenfalls eine unerläßliche Voraussetzungen für die Software-Entwicklung im Team, aber Empfehlungen dazu, wie man diese erwirbt, kann man schlechterdings nicht machen.

§ 4 Schnell-Einstieg

Dieses Kapitel gibt eine kurze Einführung in die Programmierung im Allgemeinen und mit Ruby im Speziellen. Es vermittelt nur die absolut notwendigen Grundlagen, um möglichst rasch mit der Erstellung eigener Programme beginnen zu können. Die hinter den hier vorgestellten Techniken stehende Theorie wird in den späteren Kapiteln erläutert, die zu lesen hiermit wärmstens ans Herz gelegt wird.

4.1 Program Flow

Ein Ruby-Programm wird sequentiell von oben nach unten ausgeführt. Der Ruby-Interpreter liest also die erste Zeile ein und führt sie aus. Danach die zweite, die dritte, u. s. w. Dieses »Abarbeiten« nennt man den *Program Flow* (engl. »Programfluss«), denn wie ein Fluss von der Quelle ins Meer fließt, »fließt« die Ausführungsmarke gleichsam den Quelltext hinunter. Der Interpreter überspringt keine Zeilen und führt keine doppelt aus, sofern es vom Programmierer nicht explizit verlangt wird (dazu später).

```
puts "Hallo!"
puts "Welt!"
puts "Wiedersehen, Welt!"
```

Diese drei Zeilen werden von oben nach unten nacheinander ausgeführt. Speichert man sie in einer Datei »hallo.rb« ab und führt sie aus, so erscheint folgendes:

```
$ ruby hallo2.rb
Hallo!
Welt!
Wiedersehen, Welt!
```

4.2 Kommentare

Kommentare sind Stellen im Quelltext, die nicht ausgeführt, sondern einfach ignoriert werden. Sie dienen dem menschlichen Bearbeiter des Programms als Hilfestellung. Quelltexte richtig zu kommentieren, ist eine Kunst, aber als Faustregel gilt: man sollte nicht kommentieren, *was* der Code tut, sondern *warum* er es tut. Man sollte also nach Möglichkeit erklären, weshalb man Code in dieser Weise geschrieben hat und nicht in einer anderen. Diese Faustregel rechtfertigt sich daraus, daß gut geschriebener Code gut leserlich und hinsichtlich des *was* deshalb selbsterklärend ist. Das *warum* befindet sich dagegen nur im Kopf des Programmierers, wenn es nicht in einem Kommentar niedergelegt wird. Kommentare sollten stets so verfaßt sein, daß eine völlig unbekannte dritte Person mit durchschnittlichen Kenntnissen in der genutzten Programmiersprache sie nachvollziehen kann.

Als Anfänger sollte man besser mehr kommentieren, um später wieder leicht in den Code »hereinzufinden«. Dem Grunde nach gilt das auch für Profis, die allerdings naturgemäß die Grundlagen bereits beherrschen und deshalb diesbezüglich Kommentare eher nervig als hilfsreich finden.

Kommentare in Ruby werden mit einem Raute-Zeichen # eingeleitet. Dieses Zeichen und alles, was darauf folgt, werden ignoriert:

```
puts "Hallo, Welt!"
# Dies ist ein Kommentar. Er wird ignoriert.
puts "Wiedersehen, Welt!"
puts "Immer noch da?" # Kommentare können auch hinter Code st
```

Die gelegentlich in diesem Tutorium verwandten Kommentare der Form `#=>` sind keine Sonderformen (für Ruby sind es ganz gewöhnliche Kommentare, die ignoriert werden, wie sonst auch), sondern sollen die Ausgabe auf der Kommandozeile oder ein Ergebnis einer Operation angeben. Man kann sie einfach wie ein »daraus folgt« lesen.

```
puts "Hallo, Welt" #=> Hallo, Welt
```

4.3 Parameter, Argumente, Klammern

»Befehle« wie `puts` (das ist nicht der korrekte Begriff, soll aber für den Moment genügen) akzeptieren *Parameter*, mit denen sie umgehen. So gibt `puts`

etwa seine Parameter auf der Kommandozeile aus, wie oben schon oftmals gezeigt wurde. Den Wert, mit welchem man den Parameter befüllt, bezeichnet man als Argument. Man sagt also: »`puts` erwartet einen Parameter«, aber »ich übergebe `puts` etwas als Argument (für seinen Parameter)«. Es gibt allerdings Leute, die den Unterschied zwischen Parameter und Argument für Haarspalterei halten und die Begriffe synonym verwenden. Dieses Tutorium versucht sie der Deutlichkeit halber zu differenzieren.

Argumente übergibt man in Ruby an einen »Befehl« wahlweise mit oder ohne Klammern. Es gilt die Faustregel: Je länger und komplizierter die Argumentenliste, desto eher sollte man Klammern benutzen. Dies dient der Lesbarkeit durch Menschen, dem Ruby-Interpreter selbst ist es gleich. Daher bewirken die beiden folgenden Zeilen dasselbe.

```
puts "Hallo, Welt!"
puts("Hallo, Welt!")
```

Ist die Argumentenliste leer, ist es Usus, die Klammern stets wegzulassen.

4.4 Strings und Integers

Die zwischen den Anführungszeichen befindlichen Texte nennt man *Strings* (engl. »Zeichenketten«). Wie schon zu erkennen war, beginnen sie immer mit einem ünd enden mit demselben Zeichen; alles dazwischen gehört zum String. Die Anführungszeichen selbst gehören nicht dazu; sie gehören zur formalen Grammatik der Sprache Ruby dienen nur der Erkennung von Strings. *Integers* sind Ganzzahlen, die man einfach hinschreiben kann, um sie zu verwenden.

```
puts "Hallo, Welt!"
puts 3
puts 145
```

Mit Ganzzahlen kann man auch rechnen:

```
puts 3 + 4   #=> 7
puts 10 * 5  #=> 50
puts 5 - 3   #=> 2
```

4.5 Arrays und Hashes

Zwei weitere sehr grundlegende und deshalb hier zu behandelnde Datentypen sind *Arrays* und *Hashes*. Bei ersteren handelt es sich um sortierte Listen anderer Daten, bei letzteren um lexikonartige Schlüssel-Wert-Listen. Gemeinsam ist beiden, dass sie eine beliebige Anzahl an Kindelementen aufnehmen und sogar ihrerseits wieder Arrays und Hashes (Schachtelung) enthalten können.

Arrays notiert man zwischen eckigen Klammern [...], Hashes zwischen geschweiften Klammern {...}, wobei die einzelnen Schlüssel-Wert-Paare durch den Operator => getrennt werden.

```
# Beispiele für Arrays
[1, 2, 3]
["a", "b", "c"]
[1, "b", "Hallo Welt", 5, [6, "ccc"]]

# Beispiele für Hashes
{"Klaus" => 4475, "Peter" => 8821}
{"Apfel" => "Rot", "Zitrone" => "Gelb"}
{7 => "a", "x" => [1, 2, "xxx"]}
```

4.6 Rückgabewerte

Jeder »Befehl« hat einen *Rückgabewert*, d. i. eine Information, die er an seinen Aufrufer zurückgibt. Der Aufrufer kann auf den Rückgabewert dann entsprechend reagieren, also etwa eine Fehlermeldung ausgeben oder weitere Bearbeitungen vornehmen. Wenn nichts zurückgegeben werden soll, wird der besondere Wert `nil` zurückgegeben, mit dem man nicht viel anfangen kann. `nil` repräsentiert das Nichts, die Abwesenheit von etwas. Ansonsten kann man es aber wie jeden anderen Wert handhaben. Beispielsweise ist der Rückgabewert von `puts` `nil`.

In Ruby unterscheiden sich die erwähnten arithmetischen Operationen von anderen »Befehlen« nur durch ihre an der gebräuchlichen mathematischen Schreibweise angelehnte Syntax, ansonsten funktionieren sie genauso: Sie akzeptieren Parameter (die mathematischen Operanden) und haben einen Rückgabewert (das Ergebnis). Das Plus addiert etwa seine beiden Operanden und gibt das Ergebnis zurück. Hier noch einmal der bereits oben gezeigte Code:

```
puts 3 + 4 #=> 7
```

Dabei wird also in Wirklichkeit der Rückgabewert von »3 + 4« (also 7) an »puts« übergeben. »puts« gibt ihn dann aus.

Ein anderes Beispiel. `sprintf` ersetzt spezielle, mit einem %-Zeichen markierte Zeichen in einem String durch seine Parameter und gibt das Ergebnis zurück. So kann man etwa eine Zahl mit führenden Nullen ausgeben:

```
puts(sprintf("Wert: %04d", 3)) #=> Wert: 0003
```

Natürlich hätte man das Beispiel auch gleich als »puts "0003"« schreiben können, aber interessant wird eine solche Konstruktion ja gerade dann, wenn man die 3 nicht statisch vorliegen hat, sondern etwa aus Nutzereingaben berechnet. `sprintf` ersetzt die Sequenz »%04d« durch eine vierstellige, mit Nullen aufzufüllende Zahl und gibt das Ergebnis (»Wert: 0003«) zurück. Dieser Rückgabewert wird dann an `puts` übergeben, der ihn ausgibt. So fügt sich eines ins andere.

Das Beispiel zeigt übrigens einen Fall, in dem man wirklich Klammern für die Argumentenliste benutzen sollte; mit Leerzeichen wäre nicht mehr klar, welches Argument für welchen »Befehl« ist.

4.7 Variablen

Müsste man alle Rückgabewerte sogleich verarbeiten, so würde der Quelltext einer Klammerwüste gleichen. Was mit zwei, drei Befehlen noch übersichtlich aussieht, wird bei hundert oder mehr zur Unleserlichkeit entstellt. Unleserliche Programme aber sind unwartbar und daher unerwünscht, und unter anderem deshalb gibt es *Variablen*[1], die Werte festhalten und für den späteren Gebrauch aufbewahren können. Die Aktion des Bindens eines Werts an eine Variable bezeichnet man als Zuweisung und sie geschieht in Ruby denkbar einfach durch ein Gleichheitszeichen nach dem gewünschten Bezeichner:

```
mein_string = "Hallo Welt!"
puts(mein_string)
```

[1] Die Schreibweise »Variabel«, die häufig bei Anfängern anzutreffen ist, ist falsch.

Dies weist der Variablen `mein_string` (der Name ist frei wählbar, darf aber nicht mit einem der in Tafel 3.1 aufgeführten Schlüsselwörtern identisch sein) den String »Hallo Welt!« zu. Durch Benutzung desselben Namens an späterer Stelle kann auf den so gespeicherten String wieder zugegriffen werden. Ein Zugriff an früherer Stelle ist wegen der oben erklärten Funktionsweise des Program Flow nicht möglich; die Zuweisung wird erst ausgeführt, wenn die Ausführung dort anlangt.

Es ist möglich und wünschenswert, Rückgabewerte in Variablen zu speichern. Hier noch einmal das Beispiel mit `sprintf`, diesmal unter Nutzung einer Variablen:

```
formatiert = sprintf("Wert: %04d", 3)
puts(formatiert) #=> Wert: 0003
```

Dies ist erheblich übersichtlicher und daher vorzugswürdig.

Anfänger mit mathematischem Hintergrund messen dem Gleichheitszeichen oft eine mathematische Bedeutung bei, d.h. sie lesen es als »linke Seite ist das gleiche wie rechte Seite«. Statements wie

```
a = 3
a = 4
puts(a) #=> 4
```

stürzen sie dann in tiefste Verwirrung, weil `a` doch nicht gleichzeitig 3 und 4 sein kann. Dem ist auch nicht so. Das Gleichheitszeichen drückt nur eine Zuweisung aus; der Program Flow arbeitet das Programm von oben nach unten durch und weist `a` erst den Wert 3 zu, und dann den Wert 4. Dabei bleibt es, und deshalb gibt puts 4 aus.

4.8 Objekte und Methoden

Ruby ist eine fast vollständig objektorientierte Programmiersprache, ohne ein minimales Grundwissen in der Objektorientierung ist daher kaum sinnvoll mit ihr umzugehen. Der Grundsatz »Alles ist ein Objekt« wird selten so konsequent verfolgt. Es ist daher angebracht, an dieser Stelle ein paar Worte zu dem Thema zu verlieren.

Alle Daten in Ruby sind Objekte. Alle Strings, alle Zahlen, überhaupt alles, mit dem man in einem Ruby-Programm umgehen kann. Objekte beherrschen Fähigkeiten (*Methoden*) und haben Eigenschaften (*Attribute*). Man kann sich Objekte vorstellen wie Billardkugeln: Man kann sie ablesen, man kann Dinge mit ihnen tun, aber wie sie von innen aussehen, ist von außen nicht erkennbar und bleibt ihr Geheimnis. Das nennt man das Geheimnisprinzip, und es ist eines der wichtigsten, wenn nicht sogar das wichtigste Prinzip in der objektorientierten Programmierung. Der Benutzer eines Objekts muß nichts über dessen interne Struktur wissen. Er sagt der Rakete nur »flieg zum Mond«, wie das Raketen-Objekt dies dann bewerkstelligt, braucht ihn nicht zu interessieren.

Bei allem, was bisher als »Befehl« tituliert wurde, handelt es sich in Wirklichkeit um Methoden verschiedener Objekte, namentlich des impliziten Main-Objekts, welches als Empfänger aller Methodenaufrufe in der obersten Programmebene (wie das schon oft gezeigte `puts`) fungiert. Auf anderen Objekten ruft man Methoden mithilfe eines Punkts auf. Strings beherrschen etwa die Methode gsub, die bestimmte Zeichen durch andere ersetzt und das Ergebnis zurückgibt; eine Anwendung könnte so aussehen:

```
str = "Hallo, Welt!"
str2 = str.gsub("l", "x")
puts str2 #=> Haxxo, Wext!
```

Hier wird zunächst das Stringobjekt »Hallo Welt!« an die Variable `str` gebunden. Anschließend wird auf demselben Objekt, referenziert über seinen Variablennamen, die Methode `gsub` aufgerufen, und dieser werden wiederum zwei Argumente mitgegeben: Der zu ersetzende Buchstabe, »l«, und sein Ersatz, »x«, beide ihrerseits String-Objekte. Das Ergebnis (noch ein String-Objekt) wird der Variablen `str2` zugewiesen und ihr Wert dann an `puts` übergeben und auf der Kommandozeile ausgegeben.

Welche Methoden ein Objekt beherrscht, bestimmt sich anhand seiner *Klasse*. `String` ist eine solche, `Integer` eine andere. Ruby enthält zahlreiche solcher Klassen, und auf einige wichtige wird in späteren Kapiteln noch eingegangen. Hier ist es nur wichtig zu behalten, daß jedes Objekt einer bestimmten Klasse angehört, die bauplanmäßig bestimmt, welche Methoden Objekte dieser Klasse beherrschen. In der Praxis wird diese Information typischerweise der Dokumentation der jeweiligen Klasse entnommen.

In Kurzschreibweise bedeutet »String#gsub«: Die Klasse `String` definiert bauplanmäßig eine Methode `gsub`, die Objekten dieser Klasse zur Verfügung steht. Diese Kurzschreibweise ist sehr präzise und wird daher sehr häufig genutzt; sie dient jedoch rein dokumantarischen Zwecken und ist selbst kein gültiger Ruby-Quellcode.

4.9 Bedingungen

Bedingungen dienen dazu, vom üblichen Konzept der sequentiellen Ausführung Zeile für Zeile abzuweichen. Sie sind eines der wichtigsten Elemente der Programmierung und ermöglichen erst Flexibilität. Angenommen, man wollte eine Datei aus dem Internet herunterladen und ausdrucken. Der hypothetische Code (also ein nicht funktionaler Entwurf) für ein solches Programm sähe so aus:

```
lade_herunter "http://example.org/datei.txt"
drucke "datei.txt"
```

Wie soll sich `drucke` verhalten, wenn der Download fehlgeschlagen ist, eine Datei »datei.txt« also gar nicht existiert? Um dies festzustellen, gibt es Bedingungen: *wenn* der Download erfolgreich war, *dann* drucke die Datei; *sonst* gib eine Fehlermeldung aus. Hypothetischer Code:

```
WENN lade_herunter "http://example.org/datei.txt"
ERFOLGREICH IST,
DANN drucke "datei.txt"
SONST fehler "Herunterladen fehlgeschlagen."
ENDE
```

Unterstellt, dass `lade_herunter` einen den Erfolg oder Misserfolg des Downloads indizierenden Rückgabewert besitzt, kann so auf die unerwünschte Situation reagiert werden.

Bedingungen notiert man in Ruby mithilfe von `if`. Ein Bedingungsblock kann beliebig viele (oder keine) `elsif`-Klauseln und optional einen `else`-Teil besitzen, wobei diese der Reihe nach abgeprüft werden. Schlagen alle Bedingungen fehl, wird der `else`-Teil ausgeführt. Der Bedingungsblock wird — wie jeder Codeblock in Ruby — mit `end` beschlossen.

```
if lade_herunter("http://example.org/datei.txt")
  drucke("datei.txt")
else
  fehler "Herunterladen fehlgeschlagen."
end
```

Der jeweils nicht der eingetretenen Bedingung entsprechende Teil wird nicht ausgeführt.

Ruby behandelt in Bedingungen alle Werte bis auf `nil` und `false` als wahr. Wenn also ein Ausdruck etwa 3 oder den String "Hallo Weltërgibt, gilt dies als wahr. Möchte man außer der Eigenschaft als »wahr« keine Information übermitteln, so kann dafür der Wert `true` genutzt werden. `true` und `false` (die sogenannten *Booleans*[2]) sind ähnlich wie `nil` besondere Werte, unterscheiden sich aber abgesehen von ihrer im Vorhinein feststehenden Bedeutung ansonsten nicht vom Rest der Sprache. Zur Illustration ein Beispiel, welches als »bedingungen.rb« abgespeichert werden könnte:

```
x = true
y = false
z = nil

if x
  puts "x ist wahr"
end

if y
  puts "y ist wahr"
else
  puts "y ist falsch"
end

if z
  puts "z ist wahr"
else
  puts "z ist falsch"
end
```

Ausführung:

[2] Nach *George Bool*, einem britischen Mathematiker.

```
$ ruby bedingungen.rb
x ist wahr
y ist falsch
z ist falsch
```

Die Ganzzahl Null (0) und der Leerstring ("") gelten in Ruby im Gegensatz zu anderen Programmiersprachen ebenfalls als wahr, genauso alle anderen leeren Objekte.

4.10 Iteratoren

Muss man eine Operation mehrmals hintereinander mit nur minimalen Änderungen ausführen, ist es sehr länglich, dieselben Instruktionen zwanzigmal und öfter hintereinander zu schreiben, was im Übrigen der Leserlichkeit schadet. Ruby bietet daher wie jede andere Programmierprache auch Wiederholungskonstrukte, die derartige Aufgaben drastisch vereinfachen und im Falle von dynamischen Grenzen (insbesondere solche, die durch Nutzereingabe bestimmt werden) überhaupt erst ermöglichen.

In Ruby werden diese Aufgaben typischerweise durch Einsatz sogenannter *Iteratoren* gelöst. Dies sind nichts weiter als Codeblöcke, die eine bestimmte Anzahl Male mit bestimmten Argumenten ausgeführt werden. So kann mithilfe von times etwa ein Codeblock beliebig oft, zum Beispiel dreimal, ausgeführt werden:

```
3.times do
  puts "Hi!"
end
#=> Hi!
#=> Hi!
#=> Hi!
```

Solche Blöcke werden mithilfe von `do`…`end` umschlossen. Sie erhalten bei bestimmten Methoden auch Argumente innerhalb vertikaler Balken `|`…`|`:

```
1.step(5, 2) do |i|
  puts i
```

```
end
#=> 1
#=> 3
#=> 5
```

Es handelt sich dabei technisch um Variablen, dementsprechend sind ihre Namen frei wählbar. Dabei ist darauf zu achten, dass diese Variablen in ihrer Gültigkeit selbstverständlich auf den jeweiligen Iteratordurchlauf beschränkt sind; welchen Wert sollte `i` im obigen Beispiel nach Ende von `step` auch haben? Dementsprechend wäre die Benutzung von `i` nach dem Block ein Fehler, den Ruby mit einem »NameError« quittieren würde.

Im Gegenatz zu anderen Programmiersprachen werden die typischen Schleifen `while`, `until` und `for` in Ruby nur selten benutzt. Stattdessen verwendet man wie gezeigt Iteratoren.

Die meisten Iteratoren sind Methoden der Klasse `Integer`, können also auf entsprechenden Objekten, den Ganzzahlen, aufgerufen werden. Wichtige Iteratormethoden von `Integer` sind:

times: Führt den Block eine bestimmte Anzahl Male aus.

upto: Zählt von einem Wert bis zu einem anderen hoch und führt entsprechend oft den Block aus.

downto: Wie upto, aber zählt herunter anstatt herauf.

step: Ähnlich wie upto, aber zählt statt immer nur 1 um den angegebenen Schritt aufwärts.

Daneben gibt es eine ebenfalls sehr wichtige Iteratormethode namens `each`, die von allen Container-Klassen, insbesondere `Array`, unterstützt wird. Sie dient dazu, die einzelnen Elemente einer Liste eines nach dem anderen durchzugehen und Code auszuführen. Dabei erhält der Block das jeweils aktuelle Element als Argument. Wenn man also etwa eine Liste auf der Kommandozeile ausgeben will, kann dies wie folgt bewerkstelligt werden:

```
# Array an Variable zuweisen
ary = [1, 2, "Hallo"]
```

```
# Array ausgeben
ary.each do |element|
  print "Array-Element: "
  puts element
end
```

Im Gegensatz zu `puts` gibt `print` keinen abschließenden Zeilenumbruch aus, sodaß ein nachfolgendes puts auf derselben Zeile weitermacht.

4.11 Nutzereingaben

Programme, die keine Nutzereingaben in irgendeiner Form annehmen, sind in ihrer Funktionalität recht beschränkt und dementsprechend selten. Ruby bietet daher eine Vielzahl von Möglichkeiten, den Nutzer nach seinen Wünschen zu fragen. Die wichtigsten sollen hier kurz vorgestellt werden.

4.11.1 Standardeingabe

Die besondere, globale Variable `$stdin` repräsentiert die Standardeingabe. Sie referenziert ein Objekt der Klasse `IO`, welches insbesondere die Methode `gets` zur Verfügung stellt, die eine Zeile mit Nutzereingaben anfordert und zurückgibt. Beispiel:

```
puts "Geben Sie Ihren Namen ein:"
name = $stdin.gets
print "Sie heißen: "
puts name
```

Ausführung:

```
$ ruby gets.rb
Geben Sie Ihren Namen ein:
Franz Meier<ENTER>
Sie heißen: Franz Meier
```

4.11.2 Kommandozeilenargumente

Einem Kommandozeilenprogramm können Kommandozeilenargumente mitgegeben werden. Dabei handelt es sich um den Teil bei dem Aufruf des Pro-

gramms, der nach dem Programmnamen erfolgt. Beispiel:

```
$ cat test/abc.txt
```

Hierbei wird das Programm `cat` mit dem Kommandozeilenargument `test/abc.txt` aufgerufen. Bei Ruby-Programmen wird alles hinter dem Dateinamen der Ruby-Quelldatei als Kommandozeilenargument an das Programm weitergereicht:

```
$ ruby meinprogramm.rb -a -b xxx.txt
```

Dies ruft den Ruby-Interpreter auf dem Programm-Quelltext in der Datei »meinprogramm.rb« auf, der auch dafür Sorge trägt, das dem in dieser Datei notierten Programm die drei Kommandozeilenargumente `-a`, `-b` und `xxx.txt` zur Verfügung stehen. Auf diese Kommandozeilenargumente kann in Ruby über das besondere Objekt `ARGV` (engl. Abkürzung für »argument values«, d. h. »Argumentenwerte«) zugegriffen werden. Dieses Objekt verhält sich im Wesentlichen wie ein Array.

Ein Beispielprogramm, das einfach nur seine Argumente wieder ausgibt, könnte so aussehen:

```
puts "Sie haben folgende Argumente angegeben:"

ARGV.each do |arg|
  print "Argument: "
  puts arg
end
```

```
$ ruby argv.rb -a -b xxx.txt
Sie haben folgende Argumente angegeben:
Argument: -a
Argument: -b
Argument: xxx.txt
```

4.11.3 Dateien

Mit dem Ende des Programms gehen alle verarbeiten Informationen verloren, wenn sie nicht dauerhaft (persistent) auf der Festplatte des Computers

gespeichert werden. Ruby bietet Zugriff auf Dateien auf der Festplatte mithilfe der Klasse `File`, deren Gebrauch sich am einfachsten an einem Beispiel illustrieren lässt.

```
File.open("datei.txt", "w") do |datei|
  datei.write "Eine Zeile"
end

File.open("datei.txt", "r") do |datei|
  puts datei.read #=> Eine Zeile
end
```

An diesem Code sind mehrerlei Dinge beachtenswert. Zunächst wird erstmalig eine Methode unmittelbar auf einer Klasse aufgerufen, nämlich `open` auf der Klasse `File`. Der naheliegende Schluß, daß es sich auch bei Klassen selbst um Objekte handelt, ist korrekt; wie erwähnt, ist in Ruby tatsächlich alles ein Objekt. Klassenobjekte sind aber ein fortgeschrittenes Thema. Es genügt zu wissen, daß Klassen Methoden wie andere Objekte auch beherrschen können.

Die Methode `open` erwartet zwei Parameter, einmal den Pfad zur zu öffnenden Datei als String, und einmal den Modus. »r« steht dabei für »read«, also Lesen; »w« für »write«, also Schreiben. Außerdem erhält diese Methode ähnlich wie die schon vorgestellten Iteratormethoden einen Block, und dieser Block erhält als Argument ein Objekt der Klasse `File`, für welches oben der passende Variablenbezeichner `datei` ausgewählt wurde. Dieses Objekt unterstützt unter anderem die Methoden `write`, die ihr Argument in die Datei schreibt, und `read`, deren Rückgabewert der Dateiinhalt als String ist. Der Rückgabewert von `open` selbst ist übrigens derselbe wie Rückgabewert des Blocks (er wird einfach durchgereicht). Die vom Dateiobjekt unterstützten Methoden hängen vom Modus ab; in eine mit »r« geöffnete Datei kann nicht geschrieben, aus einer mit »w« geöffneten Datei nicht gelesen werden.

Die seltsame Konstruktion über einen Block statt einer einfachen blocklosen Methode findet ihre Begründung darin, daß eine einmal geöffnete Datei auch wieder geschlossen werden muss. Dies kann jedoch schnell vergessen werden, weshalb Ruby dem Programmierer mit dieser Methode eine Hilfestellung bietet: nach dem Ende des Blocks wird die Datei automatisch geschlossen. Es besteht keine Gefahr, es zu vergessen.

Wenn der obige Code ausgeführt wird, wird eine Datei namens `datei.txt` auf der Festplatte in dem Verzeichnis angelegt, in welchem das Kommando `ruby` ausgeführt wird.

4.12 Eigene Klassen und Methoden

Bis jetzt sind nur bereits in Ruby enthaltene Klassen und Methoden vorgestellt worden. Ruby erlaubt es aber, auch eigene Klassen mit eigenen Methoden zu definieren, um so komplexe Programme beherrschbar zu machen, indem man sie in kleine, handhabbare Stücke aufteilt. Alle Facetten dieser sog. »objektorientierten« Programmierung vorzustellen, sprengt den Rahmen einer Kurzeinführung. Darauf soll jedoch an späterer Stelle [**TODO: Querverweis**] zurückzukommen sein.

Eine Klassendefinition beginnt mit dem Schlüsselwort `class` und endet (wie alle Codeblöcke) mit `end`. Innerhalb der Klassendefinition definiert man eine Methode für Objekte dieser Klasse mithilfe von `def`, welches ebenfalls mit `end` abzuschließen ist. Der folgende Code definiert eine Klasse »Hund«, deren Objekte jeweils über eine Methode »bellen« verfügen:

```
class Hund

  def bellen
    puts "Wau! Wau!"
  end

end
```

Ein Objekt dieser Klasse erstellt man, indem man auf der Klasse die von Ruby automatisch bereitgestellte Methode `new` aufruft; dieses Objekt kann man dann genauso verwenden wie diejenigen, die Rubys internen Klassen angehören. So kann man es etwa einer Variablen zuweisen oder die definierten Methoden auf ihm aufrufen:

```
timmy = Hund.new
timmy.bellen #=> Wau! Wau!
```

Methoden können Parameter erhalten und besitzen einen Rückgabewert. Letzterer ist schlicht das Ergebnis des letzten Ausdrucks in der Methode.

```
class Hund

  def bellen(wie_oft)
    wie_oft.times do
      puts "Wau!"
    end

    wie_oft
  end

end

timmy = Hund.new
puts timmy.bellen(3)
```

Die Ausgabe dieses Programms:

```
$ ruby hund.rb
Wau!
Wau!
Wau!
3
```

Man erkennt, daß man die Parameter einer Methode mithilfe einer geklammerten Liste von Variablennamen (eben den Parametern) hinter dem `def` angibt. Diese Variablen sind nur innerhalb der Methode gültig und werden, ganz wie bei den von Ruby vorgegebenen Methoden auch, mit den bei Aufruf übergebenen Argumenten befüllt. So wird die an »bellen« übergebene 3 an die Variable `wie_oft` zugewiesen. Innerhalb dieser Methode wird dann auf dieser 3 die bereits vorgestellte Iteratormethode `times` aufgerufen; zuletzt wird zwecks eines sinnvollen Rückgabewerts diese Variable noch einmal erwähnt. Der so geschaffene Rückgabewert wird dann an den Aufrufer zurückgegeben, sodaß er ebenfalls eine 3 zu Gesicht bekommt, die er dann ausgeben kann. Effektiv wird also dasselbe Objekt, welches als Argument an `bellen` übergeben wurde, auch als Rückgabewert genutzt. Das muss nicht immer so sein; andere Methoden haben andere Rückgabewerte. Es obliegt allein dem Schreiber der Methode, festzulegen, was sie zurückgibt.

Für den Aufrufer ist der Inhalt der Methode unwesentlich. Er muss nichts darüber wissen, wie diese Methode auf die Kommandozeile schreibt, oder

dass sie times benutzt, um etwas mehrfach auszugeben. Dies ist Ausdruck des schon erwähnten Geheimnisprinzips.

Anfänger haben oft Probleme mit dem Gültigkeitsbereich von Variablen in Methoden. Mit dem Ende der Methode verfallen die in der Methode benutzten Variablen:

```
class Hund
  def bellen(wie_oft)
    str = "Wau!"
    wie_oft.times do
      puts str
    end
  end
end

t = Hund.new
t.bellen(3)
puts str # <= Was soll das?
```

Im obigen Beispiel versucht jemand, die Variable `str` aus der Methode `bellen` nach deren Ende zu benutzen. Das ist ein Verstoß gegen das Geheimnisprinzip; der Aufrufer hat nichts darüber zu wissen, wie `bellen` funktioniert. Diese Variable verfällt mit dem Ende der Methode `bellen`, ein Versuch, später auf sie zuzugreifen führt zu einem »NameError«.

Teil II

Einzelne Konzepte

§ 5 Kontrollstrukturen

Ruby kennt wie jede andere Programmiersprache auch sog. *Kontrollstrukturen*. Kontrollstrukturen sind Anweisungen, die eine Abweichung von der normalen Ausführungsreihenfolge zur Folge haben. So kann man mit Kontrollstrukturen bestimmte Code-Abschnitte überspringen oder wiederholen lassen. Als universell verwendbares Wissen können die in diesem Kapitel vermittelten Erkenntnisse bis auf die konkrete syntaktische Umsetzung auch in anderen Programmiersprachen eingesetzt werden.

Alle Kontrollstrukturen arbeiten mit Wahrheitswerten. Ein Wahrheitswert wird in der Programmierung üblicherweise *Boolean* genannt[1] und kann nur zwei Werte annehmen: wahr oder falsch. Ruby definiert für diese beiden Grundaussagen entsprechend die Konstanten `true` für wahr und `false` für falsch. Alle anderen Werte gelten in Ruby immer als wahr, mit Ausnahme des Platzhalters für ein fehlendes Objekt, `nil`, der auch als falsch gilt.

> Diese Aussage gilt uneingeschränkt. Im Gegensatz zu anderen Programmiersprachen gelten auch der Leer-String und die Ganzzahl Null als wahr.

Man kann diese Wahrheitswerte miteinander kombinieren. Die zulässigen Kombinationen richten sich nach der boole'schen Logik, deren Operatoren Tafel 5.1 entnommen werden können. Der folgende Ausdruck ergibt nur dann wahr, wenn `mein_termin` und `termin_fällig` wahr sind und `anruf_vom_chef` falsch ist (das Ausrufungszeichen als logisches NICHT kehrt den Wahrheitsgehalt entsprechend um).

```
mein_termin && termin_fällig && !anruf_vom_chef
```

Neben den boole'schen Operatoren gibt es in Ruby natürlich auch die üblichen Vergleichsoperatoren. Diese sind als Methoden ausgestaltet und können bei Bedarf für eigene Objekte angepaßt werden (→ **[TODO: Querverweis]**).

[1] Nach *George Boole*, einem britischen Mathematiker und Pionier der binären Logik.

Operator	Bedeutung
`&&`	Logisches Und
`||`	Logisches Oder
`!`	Logisches Nicht

Tabelle 5.1: Boole'sche Operatoren

Operator	Bedeutung
`==`	Gleichheit
`!=`	Ungleichheit
`===`	Musterabgleich (→ 5.1.2)
`<`	Kleiner als
`>`	Größer als
`<=`	Kleiner oder gleich
`>=`	Größer oder gleich

Tabelle 5.2: Vergleichsoperatoren

Vergleichsoperatoren geben als Ergebnis ein Boolean zurück (oder sollten es zumindest) und können deshalb sehr gut in Wahrheitsausdrücken eingesetzt werden. Tafel 5.2 gibt eine Übersicht über alle Vergleichsoperatoren.

Im folgenden Beispiel ergibt der Ausdruck nur dann wahr, wenn `x` größer als zehn und `y` kleiner oder gleich fünf ist.

```
x > 10 || y <= 5
```

Eine wie in der Mathematik übliche und in einigen Programmiersprachen (etwa Python) mögliche Verkettung von Vergleichsoperatoren in der Art eines Ausdrucks `5 < x < 10` ist in Ruby nicht möglich und verursacht einen Fehler. Der genannte Ausdruck muß zwingend als `5 < x && x < 10` geschrieben werden.

Der Vergleichsoperator zur Prüfung auf Gleichheit ist == (zwei aufeinanderfolgende Gleichheitszeichen). Sein Gegenstück für Ungleichheit ist != (Ausrufungszeichen gefolgt von einem Gleichheitszeichen).

```
x == 10     # ist x gleich 10?
x != 10     # ist x nicht gleich 10?
!(x == 10) # Dasselbe
```

Klammern können zur Verdeutlichung der logischen Zusammenhänge benutzt werden. Ruby löst geklammerte Ausdrücke stets von innen nach außen auf.

Der Gleichheitsoperator darf nicht mit der Zuweisung durcheinander gebracht werden. Nur == führt einen Vergleich durch. Ein einzelnes Gleichheitszeichen ist immer eine Zuweisung. Konsequenterweise ist deshalb auch der Ausdruck

```
x = 5
```

immer wahr. Auf den Wert von `x` kommt es nämlich nicht an, denn `x` wird in diesem Ausdruck stets auf 5 gesetzt. Eine Zuweisung gibt den Wert rechts des Gleichheitszeichens zurück; das ist 5. 5 wird von Ruby als wahr behandelt.

Die unachtsame Verwendung von = statt == ist ein typischer Anfängerfehler. Es lohnt sich deshalb, darauf zu achten.

5.1 Bedingungen

Die wohl wichtigste Kontrollstruktur ist die Bedingung. Sie erlaubt es dem Programm, auf — oft von außen kommende — Ereignisse zu reagieren und den konkreten Programmablauf anzupassen. Dazu werden bestimmte Abschnitte im Code abhängig von der Bedingung (genauer: ihrem Wahrheitsgehalt) übersprungen oder ausgeführt. Trifft der Programmfluß auf eine solche Bedingung, wird der zur Ermittlung des Wahrheitsgehalts erforderliche Code ausgeführt und als Ergebnis entweder wahr oder falsch ermittelt. Andere Zustände existieren für eine Bedingung nicht; ultimativ müssen alle Fragen auf ein einfaches »Ja« oder »Nein« heruntergebrochen werden. Würde man die Frage, welche von vier vorgegebenen Zahlen die Größte ist, beantworten wollen, so kann dies in einem Programm deshalb nur geschehen, indem man sie nacheinander so miteinander vergleicht, daß für jeden Vergleich die Antwort nur »ja« (wahr) oder »nein« (falsch) lauten kann. Man kann den Programmfluß abhängig von seinen Bedingungen mithilfe sogenannter Flußdiagramme visualisieren. Flußdiagramme sind ein wichtiges Werkzeug der Software-Entwicklung, da man mit ihnen — unabhängig von einer konkreten Programmiersprache — den logischen Ablauf eines Programms oder eines Programm-

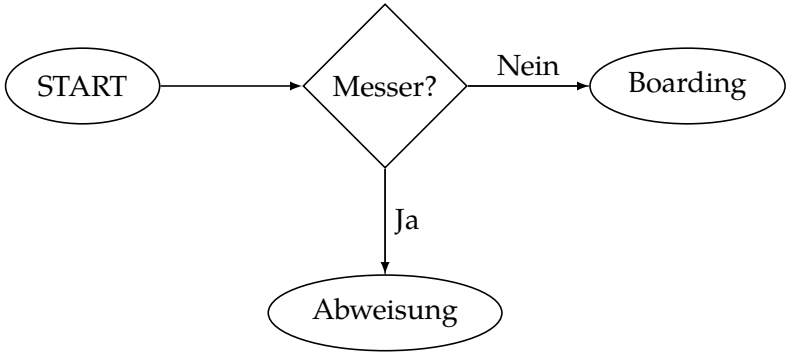

Abbildung 5.1: Messer-Prüfung

teils vorausplanen kann. Man nehme als Beispiel die Sicherheitskontrolle am Flughafen. Wer ein Messer bei sich hat, darf das Flugzeug nicht besteigen, ansonsten darf er durch. Als Flußdiagramm sähe das dann so aus wie in Abb. 5.1.

5.1.1 if

In Ruby werden Bedingungen mithilfe eines `if`-Ausdrucks beschrieben. Im folgenden Beispiel soll die Funktion `knife?` immer dann ein wahres Ergebnis liefern, wenn die untersuchte Person ein Messer bei sich hat.

```
if knife?(person)
  puts "No entrance for you."
else
  puts "Go forward."
end
```

Gibt `knife?` nun einen wahren Wert zurück, wird nur das erste `puts` ausgeführt, widrigenfalls nur das zweite. Der jeweils andere Teil wird ignoriert und spielt für den weiteren Ablauf keine Rolle mehr.

Wie das Beispiel zeigt, folgt auf `if` die zu prüfende Bedingung, die wahr oder falsch ergeben muß. Trifft sie zu, wird der unmittelbar auf die Bedingung folgende Code ausgeführt. Trifft sie nicht zu, wird der auf das optionale `else` folgende Code ausgeführt. Fehlt `else`, wird bis zum abschließenden `end` ausgeführt oder — wenn die Bedingung falsch ist — der gesamte bedingte Ausdruck bis zum `end` übersprungen. Das kann praktisch sein, wenn man

im Falle des Zutreffens einer Bedingung gar nicht mehr im aktuellen Kontext weitermachen möchte. Man kann etwa mithilfe von `raise` einen Laufzeitfehler auslösen und so effektiv die Ausführung des Programms sofort beenden (mehr zu `raise` unter **[TODO: Querverweis]**).

```
if hard_disk is_full?
  raise "Disk is full!"
end

puts "Enough space on disk, continueing..."
```

Trifft in diesem Beispiel die Bedingung zu, dann wird mit `raise` das Programm zum Absturz gebracht. Das nachfolgende `puts` wird dann gar nicht mehr ausgeführt.

Speziell für diesen.Programmiertechnik bietet Ruby eine Kurzschreibweise an, die *Postcondition*. Das heißt so viel wie »angehängte Bedingung«, was auch das Erscheinungsbild im Quelltext gut beschreibt, da das `if` mit seiner Bedingung dem bedingten Ausdruck nach- statt vorangestellt wird:

```
raise "Disk is full!" if hard_disk_is_full?
puts "Enough space on disk, continueing..."
```

Dieses Beispiel ist identisch mit dem vorangehenden. Durch den Einsatz der Postcondition ist es aber deutlich kompakter geworden: es hat statt vier Code-Zeilen nur noch zwei. Die Postcondition ist ein schönes Beispiel dafür, wie sehr Ruby den Programmierer als Design-Ziel im Auge hat. Auch läßt dieser Code sich sogar ohne tiefergehende Ruby-Kenntnisse gut verstehen, denn sein Aufbau ähnelt demjenigen der englischen Sprache.

Postconditions können nur auf eine einzelne Zeile angewandt werden und unterstützen weder `else` noch `elsif` (dazu sogleich).

Bedingungen können ineinander verschachtelt werden. Zur Veranschaulichung soll an das oben schon genutzte Beispiel der Flughafenkontrolle angeknüpft werden. Wenn der Passagier zwar kein Messer, dafür aber eine Pistole bei sich führt, sollte er den Flieger ebenfalls nicht besteigen dürfen. Dies laßt sich durch entsprechende Anpassung des Beispielprogramms erreichen. In die erste Bedingung ist im `else`-Teil die zweite Bedingung aufzunehmen:

```
if knife?(person)
  puts "No entrance for you."
```

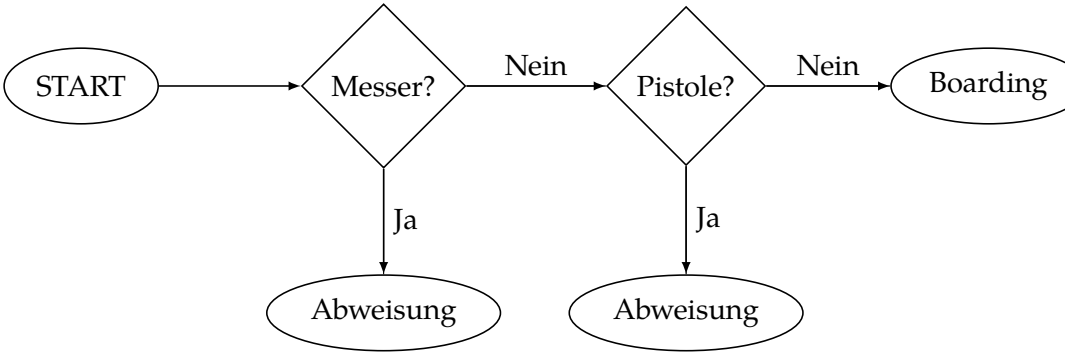

Abbildung 5.2: elsif-Konstruktion

```
else
  if gun?(person)
    puts "No entrance for you."
  else
    puts "Go forward."
  end
end
```

Während derartige Schachtelungen möglich und oft genug auch geboten sind, sind sie in der vorliegenden Situation unbefriedigend, denn logisch handelt es sich eigentlich um eine wenn-das-oder-wenn-das-sonst-Konstruktion. Sie läßt sich als Flußdiagramm wie in Abb. 5.2 darstellen.

Aus Abb. 5.2 wird ersichtlich, daß es sich eigentlich um einen geraden Gedankenfluß handelt, der vom Start bis zum Boarding fließt und nicht um eine baumartige Verschachtelung. Dies kann man in Ruby mithilfe von `elsif` abbilden. `elsif` ist gewissermaßen ein Hybrid aus `if` und `else`. Trifft die primäre, dem `if` zur Prüfung übergebene Bedingung nicht zu, dann prüft Ruby zunächst `elsif` und die ihm übergebene Bedingung. Nur, wenn auch diese nicht zutrifft, wird der `else`-Teil ausgeführt.

```
if knife?(person)
  puts "No entrance for you."
elsif gun?(person)
  puts "No entrance for you."
else
  puts "Go forward."
```

```
end
```

Das gibt den Gedankenfluß viel besser wieder und vermeidet auch die optisch unschönen Verschachtelungen. `elsif` kann beliebig oft wiederholt werden. Da man auch keine Wasserflaschen auf Flüge mitnehmen darf, bietet sich diese Prüfungsette an:

```
if knife?(person)
  puts "No entrance for you."
elsif gun?(person)
  puts "No entrance for you."
elsif water?(person)
  puts "No entrance for you."
else
  puts "Go forward."
end
```

Die gegebenen Beispiele haben didaktische Gründe. Wenn mehrere Zweige eines Bedingungskonstrukts auf dasselbe Ergebnis hinauslaufen, ist es vorzugswürdig, sie mit den in der Einleitung zu diesem Kapitel vorgestellten logischen Operatoren zu vernüpfen:

```
if knife?(person) || gun?(person) || water?(person)
  puts "No entrace for you."
else
  puts "Go forward."
end
```

5.1.2 case

Der `case`-Ausdruck ist letztlich eine besonders kompakte Form eines `if`-Ausdrucks mit vielen elsif-Klauseln. Das im vorigen Abschnitt (→ 5.1.1) genutzte Beispiel der Flughafenkontrolle kann man mit einer etwas anderen Herangehensweise auch gut als case-Ausdruck erfassen, wodurch eine noch größere Kompaktheit erreicht wird, da es dann nicht mehr erforderlich ist, für jeden Bedingungszweig die gesamte Bedingung zu wiederholen.

Im folgenden Beispiel wird unterstellt, daß die bei einer Person gefundenen Objekte einzeln auf ihren Verbotsgehalt geprüft werden. Dann ergibt sich:

```
case luggage_type
when :knife
  puts "Knifes are prohibited."
when :gun
  puts "No guns on the airplane, please."
when :water
. puts "We're very sorry, but water is not allowed."
else
  puts "Go forward."
end
```

Tatsächlich erlaubt es Ruby sogar, daß man gleichartige Bedingungszweige in ein einzelnes `when` zusammenfaßt, indem man die Probanden mit einem Komma trennt:

```
case luggage_type
when :knife, :gun
  puts "Knifes and guns are prohibited."
when :water
. puts "We're very sorry, but water is not allowed."
else
  puts "Go forward."
end
```

`case` überprüft die Gleichheit des in der erstzen Zeile genannten Objekts, indem es dieses nach und nach mit den den einzelnen `when`-Ausdrücken übergebenen Objekten vergleicht. Stellt es Gleichheit fest, wird der betreffende Code-Zweig ausgefuhrt. Ist keinerlei Gleichheit gegeben, wird — ahnlich wie bei `if` — der `else`-Zweig ausgeführt. Man kann den `case`-Ausdruck deshalb gut als einen Musterabgleich bezeichnen.

`case` stellt die Gleichheit nicht mit dem üblichen Gleichheitsoperator == fest, sondern benutzt einen eigenstandigen Operator ===. Dieser Operator macht meistens, aber nicht immer das gleiche wie ==. Die beiden wichtigsten Ausnahmen von dieser Regel sind die Klassen `Regexp` und `Class`. Reguläre Ausdrücke, die +ber `Regexp` beschrieben werden, haben ihr eigenes Kapitel ([**TODO: Querverweis**]). `Class` ist die Klasse von Klassen, z.B. `Array` oder `String`. Das erlaubt es, den `case`-Ausdruck zu benutzen, um zu überprüfen, von welcher Klasse das zu untersuchende Objekt eine Instanz ist. Beispiel:

```
case obj
when Integer
  puts "Es ist eine Ganzzahl."
when String
  puts "Es ist eine Zeichenkette."
when Array
  puts "Es ist eine Liste."
else
  puts "Es ist etwas anderes."
end
```

Der Operator === wird auf dem Objekt hinter dem `when` aufgerufen, nicht auf dem Objekt hinter dem `case`.

Wenig bekannt ist, daß es eine Sonderform von `case` gibt, die auf die Angabe eines abzugleichenden Objekts hinter `case` verzichtet. Diese Form von `case` verhält sich genauso wie ein if-elsif-Konstrukt.

```
case
when x < 5
  puts "x is smaller 5."
when x < 10
  puts "x is smaller 10."
else
  puts "x is greater 10."
end
```

5.1.3 Ternäroperator

Eine besonders kompakte Form, eine einzelne Bedingung auszudrücken, ist der aus C übernommene Ternäroperator. Mit ihm kann man ein komplettes Wenn-Dann-Sonst in eine Zeile komprimieren. Seine Syntax lautet:

```
Bedingung ? <Wahr-Code> : <Falsch-Code>
```

Das »?« ist Teil der Operator-Syntax. Endet ein Methodenname im Einzelfall mit einem »?«, so darf dieses nicht entfallen. Beispiele:

```
x == 5 ? puts("Fünf") : puts("Nicht fünf")
x.odd? ? puts("Ungerade") : puts("Gerade")
puts(x == 5 ? "Fünf" : "Nicht fünf")
```

Es ist möglich, Ternäroperatoren zu verschachteln. Davon sollte man aus Gründen der Leserlichkeit aber tunlichst Abstand nehmen.

5.2 Schleifen

Es gibt viele Situationen, in denen ein Programm mehrfach die gleiche oder eine sehr ähnliche Arbeit durchführen muß. Es wäre sehr unpraktisch, wenn man in diesen Fällen den gesamten Code, der wiederholt werden muß, vollständig kopieren und wieder einfügen müßte. Wartbarkeit etwa bei entdeckten Fehlern und Übersicht gingen verloren. Ganz unmöglich ist dieses Vorgehen, wenn die Anzahl der Wiederholungen von einer Nutzereingabe abhängt, die naturgemäß im Programm nicht im Vorhinein bekannt ist.

Diesem Problem begegnet man in der Programmierung mithilfe von *Schleifen*. Eine Schleife besteht aus einer Bedingung und einem Schleifenkörper. Abhängig von der Bedingung wird dann der Schleifenkörper noch ein weiteres Mal ausgeführt oder nicht. Mithilfe des Schlüsselwortes `break` kann jede Schleife auch sofort abgebrochen werden. Der Programmfluß wird dann hinter dem Ende der Schleife fortgesetzt; dasselbe gilt natürlich auch vom »regulären« Schleifenende durch Eintritt oder Ausbleiben der Schleifenbedingung.

Ruby kennt alle gängigen Schleifentypen, die andere Programmiersprachen auch kennen. Es ist auch keine Sünde, davon Gebrauch zu machen, solange die Übersichtlichkeit nicht leidet. Erfahrene Rubyisten bevorzugen allerdings die Verwendung von Iteratoren (→ 5.3); wer ernsthaft mit Ruby programmieren will, kommt um deren Verständnis nicht herum. Da Iteratoren aber oft intern auf Schleifen basieren, ist die Kenntnis von Rubys Schleifenkonstrukten aber dennoch unabdinglich.

5.2.1 while

Der Grundtypus aller Schleifen ist `while`. Dabei handelt es sich um eine sog. kopfgesteuerte Schleife. Dies deshalb, weil die zu überprüfende Bedingung dem Schleifenkörper vorangestellt wird, also gewissermaßen den »Kopf« bildet. Ihre Syntax lautet:

```
while <Bedingung>
```

```
  <Code>
end
```

Der Schleifenkörper einer while-Schleife wird solange immer wieder ausgeführt, wie die Schleifenbedingung wahr ist. Mit Ausfall der Bedingung endet die Schleife. Wichtig dabei: die Bedingung wird immer zu Anfang eines Schleifendurchlaufs geprüft, nicht mittendrin oder am Ende. Ändern sich während der Ausführung des Schleifenkörpers die Umstände derart, daß die Schleifenbedingung falsch würde, so führt dies nicht zum sofortigen Abbruch der Schleife (dafür kann man `break` verwenden). Vielmehr wird der aktuelle Durchlauf bis zum Ende durchgeführt. Erst vor dem nächsten Durchlauf wird die Bedingung erneut geprüft.

Für das nunmehr bereits bekannte Flughafenbeispiel (→ 5.1) gilt im Namen der Terrorabwehr, daß man alle mitgeführten Objekte zu der Kontrolle zu übergeben hat:

```
while person.has_luggage?
  luggage = person.next_piece
  case luggage.type
  when :knife, :gun
    # ...wie gehabt...
  end
end
```

Erst, wenn die Person kein Gepäck mehr in ihren Taschen hat, endet diese Schleife. Eine Darstellung als Flußdiagramm findet sich in Abb. 5.3.

5.2.2 until

`until` funktioniert spiegelbildlich zu `while`. Der Schleifenkörper wird ausgeführt, solange die Bedingung *nicht* zutrifft.

```
until person.naked?
  luggage = person.next_piece
  case luggage.type
    # ...wie gehabt...
  end
end
```

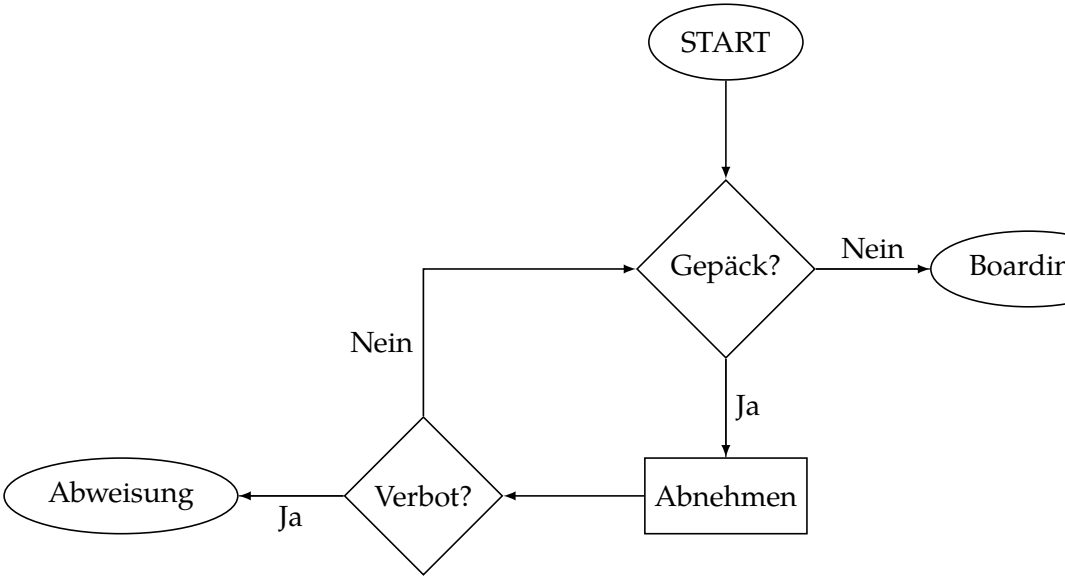

Abbildung 5.3: While-Schleife

5.2.3 Schleifen-Modifier

Sowohl `while` als auch `until` können ähnlich der Postcondition (→ 5.1.1) einer einzelnen Code-Zeile hinten angestellt werden. Der Teil vor dem Modifier wird dann als Schleifenkörper behandelt.

```
puts "Keeeeeeekse!" until give_cookies?
```

Hier lauert ein erwähnenswerter Fallstrick. Eine Variable, die erst im Modifier erstmalig auftaucht, kann im Schleifenkörper irritierenderweise noch nicht verwandt werden. Der folgende Code löst deshalb einen Fehler aus:

```
def foo
  # ...
end

# Dies verursacht einen NameError für "x"
puts(x) while x=foo
```

5.2.4 for

Oftmals ist der Grund, aus dem man eine Schleife einsetzen will, eine Liste, deren einzelne Elemente man in dieser oder jener Weise verarbeiten möchte. Listen tauchen in Ruby üblicherweise als Instanzen (→ 6.1) der Klasse Array auf, jedoch ist das nicht erforderlich. Als Liste gilt jedes Objekt, daß die Methode `each` beherrscht. Das Abgehen einer Liste Element für Element nennt man *Iteration*, und statt umständlich einen Variable als Listen-Index zu führen und in jedem Schleifendurchlauf um eins zu erhöhen, kann man in Ruby eine `for`-Schleife benutzen. Ihre Syntax lautet:

```
for <variable> in <liste>
  <Code>
end
```

So kann man etwa über die Gesamtheit der Fluggäste iterieren:

```
for passenger in passengers
  until passenger.naked?
    luggage = passenger.next_piece
    case luggage.type
      # ...wie gehabt...
    end
  end
end
```

Der Name der Schleifenvariablen ist frei wählbar. Es handelt sich um eine übliche lokale Variable.

Man beachte an obigen Beispiel, daß man Schleifen selbstverständlich auch schachteln kann.

5.3 Iteratoren und Blöcke

Die im vorigen Abschnitt (→ 5.2) vorgestellten Schleifenkonstrukte finden in Ruby praktisch nur selten Anwendung. Sehr viel häufiger werden sog. *Iteratoren* eingesetzt. Dabei handelt es sich um Methoden, die auf einer Liste operieren und für jedes Element in der Liste einen Codeblock ausgeführen. Die

Grundform einer solchen Iteratormethode ist `each`. Genau wie die for-Schleife geht `each` eine Liste Element für Element durch und führt für das Element der übergebenen Codeblock aus.

```
ary = [1, 5, 7]
ary.each do |e|
  puts e
end
```

Dieser Code gibt nacheinander jedes der Elemente in `ary` auf der Standardausgabe aus. Aus dem Beispiel werden zwei Dinge ersichtlich:

1. Codeblöcke werden mithilfe von `do...end` an eine Methode übergeben.
2. Blöcke können eigene Parameter besitzen, die zwischen zwei Balken auftauchen und von der Iteratormethode »befüllt« werden.

Während gewöhnliche Methodenparameter Informationen in die Methode »hinein« übergeben, übergeben Blockparameter Informationen aus der Methode »heraus«.

Neben `do..end` steht auch eine alternative, kürzere Syntax für die Übergabe von Blöcken zur Verfügung, die sich geschweifter Klammern {...} bedient. Der nachfolgende Code ist dem obigen Beispiel gleichwertig:

```
ary = [1, 5, 7]
ary.each{ |e| puts(e) }
```

Diese Form ist oft — gerade bei Einzeilern — kompakter und dann vorzugswürdig. Im Gegensatz zur »ausführlichen« Form mit `do...end` weisen die geschweiften Klammern aber eine höhere Bindungswirkung (*Präzedenz*) auf, d. h. sie werden von Ruby früher ausgewertet. Das zeitigt im obigen Beispiel keine Folgen, kann jedoch im Einzelfall zu Verwirrung führen. Ein Beispiel (zu `reduce` → 5.3.3):

```
ary = [1, 5, 7]
puts ary.reduce do |sum, el|
  sum + el
end
```

Führt man diesen Code aus, erhält man einen Laufzeitfehler und das Programm endet.

```
beispiel.rb:2:in `each': no block given (LocalJumpError)
from beispiel.rb:2:in `reduce'
from beispiel.rb:2:in `<main>'
```

Derselbe Code mit geschweiften Klammern funktioniert wie erwartet:

```
ary = [1, 5, 7]
puts ary.reduce {|sum, el|
  sum + el
}
#=> 13
```

Aufgrund der höheren Präzedenz ordnet Ruby den Block innerhalb von {...} dem Aufruf von `reduce` zu. Bei der Nutzung von `do...end` dagegen wird der Block fehlerhafterweise `puts` zugeordnet (welches einen mitgegebenen Block schlicht ignoriert). `reduce` steht dadurch ohne Block da. Das ist aber ein Fehler, den Ruby als »LocalJumpError« moniert und die Ausführung abbricht. Wer unbedingt auf `do...end` besteht, muß Ruby mithilfe von runden Klammern aushelfen, da runde Klammern die höchste Präzedenz überhaupt besitzen und deshalb stets zuerst ausgewertet werden:

```
ary = [1, 5, 7]
puts (ary.reduce do |el, sum|
  sum + el
end)
#=> 13
```

Dies ist aber schwer leserlicher Code von minderer Qualität. Vom Programmierer darf erwartet werden, daß er die unterschiedliche Präzedenz von {...} und `do...end` kennt; in solchen Fällen sind, wenn sie unvermeidlich sind, die geschweiften Klammern vorzugswürdig.

[TODO: Lambdas gehören in das Methodenkapitel.]

Noch kürzer als ein Einzeiler mit geschweiften Klammern ist die abgekürzte Blocksyntax mithilfe des kommerziellen Und-Zeichens &. Dieses etwas gewöhnungsbedürftige Konstrukt gehört schon fast in den Bereich der Metaprogrammierung (→ **[TODO: Querverweis]**), soll aber wegen seiner häufigen Verwendung hier diskutiert werden. Sein typischer Einsatz sieht so aus:

```
ary = [1, 2, 3, 4]
puts ary.select(&:odd?) #=> 1, 3
```

Ruby konstruiert aus dem an & übergebenen Symbol :odd? einen Block, der nichts anderes tut, als eine Methode diesen Namens auf jedem Element aufzurufen (Integer#odd? gibt für ungerade Zahlen einen wahren, sonst einen unwahren Wert zurück; zu `select` → 5.3.3). Damit sind die beiden folgenden beiden Zeilen äquivalent:

```
ary.select(&:odd?)
ary.select{|el| el.odd?}
```

5.3.1 Schleifenersatz: times, upto, downto, step

Keine Iteratormethoden im engeren Sinne — weil sie nicht auf einer Liste arbeiten — stellen die in der Abschnittsüberschrift genannten Methoden der Klasse `Integer` dar. Trotzdem ersetzen sie in der Praxis weitgehend die sonst anzuwendende while-Schleife.

Die Methode Integer#times führt einfach nur den übergebenen Block so oft aus, wie es dem Wert des Empfängers entspricht. Ganz im Sinne von Rubys Design-Zielen führt der Einsatz dieser Methode zu ausgesprochen leserlichem Code.

```
3.times do
  print "Ho! "
end
#=> Ho! Ho! Ho!
```

`times` übergibt ein optionales Blockargument, das als eine Art Schleifenzähler fungieren kann. Meist wird es nicht benötigt. Im ersten Durchlauf ist sein Wert Null, danach erhöht dieser sich jeweils um eins bis er um eins kleiner ist als der Wert des Empfängers.

```
3.times do |i|
  puts i
end
#=> 0
```

```
#=> 1
#=> 2
```

`upto` und `downto` zählen vom Wert des Empfängers aus herauf bzw. herab bis zum Wert des Arguments.

```
5.upto(7){|i| puts(i)}
#=> 5, 6, 7
7.downto(5){|i| puts(i)}
#=> 7, 6, 5
```

Sind Einer-Schritte nicht ausreichend, kann sowohl für das Herauf- wie das Herabzählen stattdessen `step` benutzt werden. Die Methode akzeptiert als zweites Argument die Schrittweite.

```
5.step(11, 2){|i| puts(i)}
#=> 5, 7, 9, 11
```

5.3.2 each und Anverwandte

`each` als Grundmethode für die Listen-Iteration wurde bereits angesprochen. Die Methode führt einfach für jedes Element in der Liste den übergebenen Codeblock aus.

```
ary = [1, 2, 3, 4]
ary.each{|el| puts(el)}
#=> 1, 2, 3, 4
```

Neben `each` gibt es einige an `each` angelehnte Methoden, die inhaltlich nicht anders als `each` agieren, sich aber in der Anzahl und/oder Art der Blockargumente unterscheiden.

Die offensichtlichste Ergänzung ist `reverse_each`, die sich so verhält, wie man es erwarten würde: sie iteriert rückwärts über die Liste.

```
ary = [1, 2, 3, 4]
ary.reverse_each{|el| puts(el)}
#=> 4, 3, 2, 1
```

Ebenfalls einfach verhält sich `each_with_index`. Die Methode funktioniert wie `each`, übergibt aber einen bei Null beginnenden Index neben dem eigentlichen Listenelement an den Block. Der höchste Wert des Indices ist demzufolge die Länge der Liste minus eins.

```
ary = [10, 15, 20]
ary.each_with_index do |el, idx|
  puts "#{el} has index #{idx}"
end
#=> 10 has index 0
#=> 15 has index 1
#=> 20 has index 2
```

Schwieriger zu verstehen sind `each_cons` und `each_slice`. `each_cons` übergibt die einzelnen Elemente mehrfach unter Verwendung einer Art Vorausschau, während `each_slice` die Liste in Einzelteile der gewünschten Länge zerlegt.

```
ary = [10, 15, 20, 25, 30]
ary.each_cons(2) do |el1, el2|
  puts "#{el1} is followed by #{el2}"
end
#=> 10 is followed by 15
#=> 15 is followed by 20
# ...
#=> 25 is followed by 30

ary.each_slice(2) do |el1, el2|
  p [el1, el2]
end
#=> [10, 15]
#=> [20, 25]
#=> [30, nil] # Fehlende Elemente werden nil!
```

5.3.3 Auf each aufbauende Methoden

Aufbauend auf `each` bietet Ruby einen ganzen Reigen von Iteratormethoden[2].

`map` wendet auf jedes Listenelement den übergebenen Codeblock an. Im Unterschied zu `each` erzeugt es aber während des Durchlaufs eine neue Liste, bei der jedes Element durch den Rückgabewert des Blocks ersetzt wurde. Dies ermöglicht eine Transformation der gesamten Liste nach einer einheitlichen Regel. So kann man beispielsweise alle Elemente einer Liste quadrieren und das Ergebnis als neue Liste erfassen:

```
ary = [2, 4, 6]
ary2 = ary.map{|el| el * el}
puts ary2 #=> [4, 16, 36]
```

`map` verändert die ursprüngliche Liste nicht. Dazu kann `map!` benützt werden.

```
ary = [2, 4, 6]
ary.map!{|el| el * el}
puts ary #=> 4, 16, 36
```

`collect` und `collect!` sind Aliase für `map` und `map!`.

Die Methoden `select` und `reject` können zur Filterung einer Liste benützt werden. Alle Elemente, für die der übergebene Block ein wahres (`select`) bzw. ein falsches (`reject`) Ergebnis ergibt, werden behalten, die übrigen Elemente werden verworfen. Wie schon zu `map!` gibt es jeweils eine mit einem Ausrufungszeichen endende Variante, die keine neue Liste zurückgibt, sondern den Empfänger bearbeitet. `delete_if` ist ein gleichwertiger Alias zu `reject!`.

```
ary = [1, 2, 3, 4]
ary.select{|el| el.odd?} #=> [1, 3]
ary.reject{|el| el.odd?} #=> [2, 4]
```

Listen kann man mithilfe der zwei Methoden `sort` und `sort_by` (die ebenfalls in der bekannten, den Empfänger modifizierenden Ausrufungszeichen-Variante zur Verfügung stehen) sortieren. `sort` kann auch ohne Block benutzt

[2] Diese werden technisch mithilfe des Moduls `Enumerable` umgesetzt. Dazu unten [**TODO: Querverweis**].

werden und wendet dann Rubys Standard-Vergleichsoperation an. Mit Block obliegt es dem Block, die zwei übergebenen Listenelemente zu vergleichen und dann einen von drei Zahlwerten zurückzugeben, der das Verhältnis der beiden Elemente angibt: -1, wenn das erste kleiner ist als das zweite, 0, wenn sie gleich sind, 1, wenn das erste größer ist als das zweite. `sort_by` funktioniert nur mit Block. Die Methode benützt immer Rubys Standard-Vergleicher, erlaubt es aber, stattdem ganzen Objekt nur bestimmte Eigenschaften zu vergleichen.

```
ary = ["a", "xx", "bbb", "c"]
ary.sort                    #=> ["a", "bbb", "c", "xx"]
ary.sort{|a, b| b <=> a} #=> ["xx", "c", "bbb", "a"]
ary.sort_by(&:length)       #=> ["a", "c", "xx", "bbb"]
```

Mithilfe von `reduce` (Alias: `inject`) kann man eine Liste auf einen einzelnen Wert reduzieren. So kann man etwa die Summe aller Listenelemente in kompakter Form berechnen, ohne wie in anderen Programmiersprachen auf eine ausdrückliche Schleifenkonstruktion zurückgreifen zu müssen. Beim Einsatz dieser Methode sollte man besonderes Augenmerk auf die Reihenfolge der Blockargumente legen: das erste Blockargument dient dem Aufbau des »Sammelwerts« und wird mit jedem Durchlauf ohne ausdrückliche Zuweisung auf den Rückgabewert des letzten Durchlaufs gesetzt, das zweite Element gibt das gerade iterierte Element an.

`map` und `reduce` treten häufig zusammen auf. Zunächst wird die Liste Element für Element verändert und danach die Liste auf einen Wert reduziert. Dieses sog. *Map-Reduce-Pattern* entstammt eigentlich — wie die meisten der in diesem Abschnitt vorgestellten Methoden — der funktionalen Programmierung.

```
ary = [2, 4, 6]
x = ary.map{|e| e * e}.reduce{|sum, e| sum + e}
puts x #=> 56
```

In diesem Beispiel wird erst jedes Element quadriert und anschließend die Summe aller quadrierten Elemente berechnet. Damit ist dieser Ruby-Code die Umsetzung dieser Formel:

$$x = \sum_{e=1}^{3} (2e)^2 \tag{5.1}$$

	all?	any?	none?
Alle Elemente wahr	true	true	false
Alle Elemente falsch	false	false	true
Einige Elemente wahr	false	true	false
Liste leer	true	false	true

Tabelle 5.3: Rückgabewert von all?, any?, none?

Vor diesem Hintergrund wird verständlich, weshalb dieses Programmiermuster der *funktionalen* Programmierung zugerechnet wird.

Es gibt noch einige speziellere Wege, um Listen auf einen Wert zu reduzieren. Die Methoden `all?`, `any?` und `none?` sind zur Verwendung in Bedingungen (→ 5.1) gedacht. Sie prüfen alle Elemente der Liste anhand des übergebenen Codeblocks durch und beantworten die gemäß ihrem Methodennamen gestellte Frage entsprechend. Tafel 5.3 gibt eine Übersicht über die verschiedenen möglichen Ergebnisse.

```
ary = [1, 2, 3, 4]
ary.all?(&:even?) #=> false
ary.any?(&:even?) #=> true
ary.none?(&:even?) #=> false
```

`find` gibt das erste Element zurück, für das der Block ein wahres Ergebnis liefert.

```
ary = [1, 2, 3, 4]
ary.find{|el| el.even?} #=> 2
```

5.3.4 Enumeratoren

Seit Version 1.9 kennt Ruby über die »gewöhnlichen« Iteratormethoden hinaus zudem noch eine Klasse Enumerable::Enumerator. `Enumerable` ist ein Modul, das hier nur als Namespace benutzt wird (zu Namespaces [**TODO: Querverweis**]) und hier nicht weiter zu interessieren braucht. Die darin definierte Klasse `Enumerator` entspricht in etwa dem, was man in anderen Programmiersprachen wie etwa C++ als Iterator bezeichnet. Instanzen dieser Klasse, die von Rubyisten gelegentlich auch als externe Iteratoren bezeichnet werden,

erlauben es, die eigentliche iterative Operation vom auszuführenden Codeblock zu trennen. Man kann das etwa benützen, um die Iteration einer Liste an einem bestimmten Punkt »einzufrieren«, um dann zu einem späteren Zeitpunkt damit fortzufahren.

Man erhält einen Enumerator, indem man entweder ausdrücklich die Methode `enum_for` auf der zu iterierenden Liste aufruft und ihr den Namen der gewünschten Iteratormethode übergibt oder indem man die Iteratormethode ohne den eigentlich erforderlichen Codeblock aufruft[3]. In beiden Fällen ist der Rückgabewert eine Instanz von `Enumerator`. Ein einzelner Iterationsschritt wird dann durch Aufruf von `next` bewirkt. Beispiel:

```
ary = [1, 5, 7]
enum = ary.enum_for(:each)
# gleichwertig: enum = ary.each
x = enum.next
puts "Erstes Element: #{x}"
x = enum.next
puts "Zweites Element: #{x}"
```

Der Rückgabewert von `next` entspricht dem, was bei regulärem Aufruf mit Block als Blockargument übergeben worden wäre, ggf. als Array. Am Ende der Iteration erzeugt Ruby einen Laufzeitfehler vom Typ `StopIteration`.

[3] Letzteres funktioniert mit den meisten, aber nicht allen Iteratormethoden, etwa nicht mit `reduce`.

§ 6 Klassen und objektorientierte Programmierung

Ruby gehört primär zu den sogenannten *objektorientierten* Programmiersprachen. Objektorientiert heißt, dass die Programmiersprache die natürliche Welt abzubilden sucht und einen möglichst »normalen« Umgang mit dieser künstlichen Welt fördern will. Die Objektorientierung ist nur eines von vielen gebräuchlichen Programmierparadigmen, stellt jedoch das aktuell am weitesten verbreitete dar. Seine Kenntnis ist für eine fachliche Konversation unabdinglich, und praktisch jedes größere Programm wird auf die Techniken der objektorientierten Programmierung zurückgreifen.

Andere Programmierparadigmen sind etwa funktionale Programmierung (Beispielsprache: Haskell) oder prozedurale Programmierung (Beispielsprache: C). Ein besser oder schlechter gibt es nicht; man sollte stets das Werkzeug nutzen, das am besten auf die gestellte Aufgabe paßt. Ruby enthält neben den objektorientierten Elementen aber auch Elemente anderer Programmierparadigmen. Insbesondere wurden zuletzt immer stärker funktionale Elemente aufgenommen.

6.1 Klassen, Instanzen, Objekte

Der Schlüssel zum Verständnis der Objektorientierung liegt in der Klassifizierung: Die Dinge werden gemäß dem Grad ihrer Beschaffenheit in Gruppen unterteilt, ganz wie man es aus der reellen Welt gewöhnt ist. Betrachtet man etwa einen Hund, so läßt er sich etwa wie folgt beschreiben:

- Eigenschaften:
  - Vier Beine
  - Faul

  - Hervorragende Nase

- Fähigkeiten:
  - Laufen
  - Bellen
  - Beißen

Katzen sind keine Hunde. Ihre Beschreibung ist anders:

- Eigenschaften:
  - Vier Beine
  - Agil
  - Scharfe Krallen

- Fähigkeiten:
  - Laufen
  - Miauen
  - Kratzen

Es handelt sich um folglich um unterschiedliche Gruppen mit unterschiedlichen Merkmalen. Schaut man aber genauer hin, so unterscheiden sich einzelne Hunde (und Katzen) allerdings auch untereinander. Nicht jeder Hund ist gleich wild, nicht jede Katze gleich verschmust. Auch sehen mitnichten alle Hunde gleich aus. Ein beliebiger Schäferhund »Lutz« mag sich ganz anders verhalten als ein anderer Schäferhund »Timmy«. Auch die einzelnen Repräsentanten einer Gruppe sind demnach nicht vollständig identisch — denn es wäre absurd anzunehmen, dass »Timmy« oder »Lutz« eine eigene Obergruppe darstellten. Sie sind die konkreten »Anwendungsbeispiele« für eine abstrakt definierbare Gruppe.

Überträgt man diese Beispiele auf die Termini der Objektorientierung, so spricht man statt von Gruppen von *Klassen* und statt von Repräsentanten einer Gruppe von *Instanzen* dieser Klasse[1].

[1] Der Begriff »Instanz« ist aus deutscher Sicht unverständlich und sprachlich verfehlt, hat sich aber so eingebürgert. Er läßt sich nur als direkte Übernahme aus dem Englischen erklären, in dem er etwa in der Bedeutung von »for instance« (»zum Beispiel«) vorkommt.

Auf die konkreten Beispiele angewandt heißt das, daß es sich bei den abstrakten Definitionen von »Hund« und »Katze« um Klassen handelt. Sie beschreiben einen Idealzustand, der es uns Menschen ermöglicht, die uns umgebenden Dinge in diese Kategorien einzuordnen, obwohl nicht jeder Repräsentant, d. h. nicht jede Instanz, wie Lutz und Timmy es sind, in allen Einzelheiten mit der abstrakten Definition »des Hundes an sich« übereinstimmt. Invertiert man die Sichtweise und betrachtet diese Systematik von den Klassen herab anstatt von den Instanzen hinauf, so spricht man insoweit auch gerne von »Bauplänen« oder »Blaupausen«, aus denen die einzelnen Instanzen konstruiert werden. So mag es etwa eine Klasse »Bohrmaschine« geben, die gleich einem Konstruktionplan festlegt, wie die einzelnen Bohrmaschinen (die Instanzen) herzustellen sind. Nichtsdestotrotz werden auch die einzelnen Bohrmaschinen sich ein wenig voneinander unterscheiden; die eine hat mehr Macken als die andere, die eine ist defekt, während die andere funktioniert. Dennoch bleiben sie eben Bohrmaschinen.

Hauptelement der Objektorientierung ist — wie der Begriff schon nahelegt — das *Objekt*. Im Grunde sind Objekte nichts weiter als die oben schon erwähnten Instanzen, also die konkrete Repräsentation eines abstrakten Schemas, mit der man tatsächlich interagieren kann. Wie ein alter Witz sagt, hat noch niemand »den Durchschnittssteuerzahler« persönlich getroffen, obwohl doch allenthalben von ihm gesprochen wird und er daher doch offenkundig eine ausgesprochen wichtige Person sein muss. Er ist mithin nur ein abstraktes Gruppenkriterium, mit dem selbst keine Interaktion möglich ist. Dies ist nur mit seinen konkreten Instanzen (einem konkreten Schreiner etwa) möglich. Konsequenterweise hat man es in der objektorientierten Programmierung häufig mit Objekten zu tun, die miteinander interagieren und weniger mit den Klassen, die die bloßen abstrakten Definitionen dieser Objekte darstellen[2]. Abbildung 6.1 zeigt zwei Klassen mit drei Instanzen.

In Ruby werden Klassen mithilfe des Schlüsselworts `class` definiert. Alles zwischen `class` und dem dazugehörigen `end` ist Teil der Klassendefinition. Ruby stellt den Klassenobjekten automatisch eine Methode (dazu sogleich) namens `new` zur Verfügung, die dazu dient, Instanzen der Klasse zu erzeugen. Beispiel:

[2] Bezeichnenderweise ist diese Sicht der Welt keine Erfindung der Informatik. Schon der griechische Philosoph *Platon* (428 v. Chr. – 348 v. Chr.) entwickelte eine darauf beruhende Erkenntnistheorie, nach der in einer überirdischen Sphäre die »Urbilder der Dinge an sich« existieren sollten, deren bloße »Abbilder« wir hier auf Erden nur zu sehen vermöchten, da ihre Reinheit und Klarheit gleich der Sonne zu grell für unseren kleinen Verstand sei. Die bekannteste Abhandlung dieser Theorie ist das platonische Höhlengleichnis.

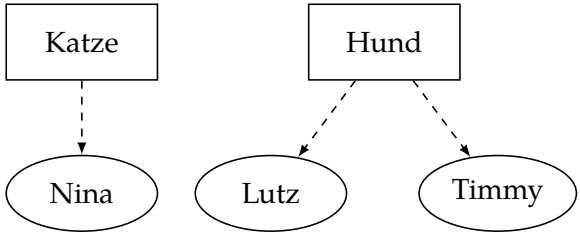

Abbildung 6.1: Zwei Klassen und drei Instanzen

```
class Dog
end

timmy = Dog.new
```

Das Beispiel zeigt die kleinstmögliche Klassendefinition und die Instanziierung eines Objekts. Zunächst wird die (leere) Klasse `Dog` (engl. »Hund«) definiert. Aus dieser wird mithilfe des von Ruby automatisch bereitgestellten `new` ein neues Objekt erstellt (instanziiert) und an die lokale Variable `timmy` gebunden.

6.2 Methoden und Attribute

Bisher ist nur besprochen worden, daß Klassen abstrakte Definitionen darstellen, die von den einzelnen Instanzen in lebendige Objekte umgesetzt werden. Es steht noch aus, zu klären, wie diese Definitionen aussehen. Dabei unterscheidet man gemeinhin zwischen *Methoden* und *Attributen*. Bei ersteren handelt es sich gewissermaßen um die »Fähigkeiten« ein Objekts, bei letzterem um die »Eigenschaften«. Allen Hunden ist gemeinsam, daß sie laufen, springen und bellen können — das sind alles Fertigkeiten, in denen der Hund aktiv von sich aus etwas tut und in den Lauf der Dinge eingreift. Dies nennt man die Methoden[3]. Im Gegensatz dazu zeugen die Attribute bloß vom Zustand eines Objekts. Ein Hund ist weiß oder schwarz (Attribut »Farbe«), gesund oder krank (Attribut »Gesundheit«), schnell oder langsam (Attribut »Geschwindigkeit«). Die Methoden können in die Attribute eingreifen und

[3] Passive Fähigkeiten sind in der Objektorientierung ein Zeichen schlechten Stils. Eine Methode »hochgehoben werden« würde als unpassend angesehen werden.

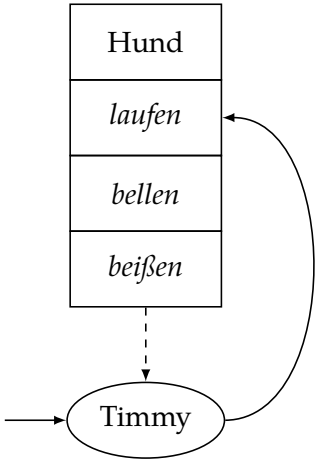

Abbildung 6.2: Nachschlagen der Methodendefinition

sie verändern: so könnte etwa die Methode »lossprinten« das Attribut »Geschwindigkeit« erhöhen.

Dies harmoniert mit dem Konzept von Klassen und Instanzen. Die abstrakt gehaltenen Klassen enthalten lediglich gerüstartig die Definitionen der Methoden und Attribute, die erst von den einzelnen Instanzen mit Leben gefüllt werden.

Während Klassen also die abstrakt-generelle Ebene darstellen, spielt sich der eigentliche Hauptteil eines Programms auf der konkret-individuellen Ebene der Instanzen ab. Die einzelnen Objekte eines Programms nutzen ihre Methoden, um miteinander zu interagieren und ihre Attribute zu verändern. Die Aktivierung einer Methode auf einem Objekt nennt man den *Methodenaufruf*. Verlangt man den Aufruf einer Methode auf einem Objekt, so prüft die Programmiersprache, ob die verlangte Methode in der Klasse des angesprochenen Objekts überhaupt definiert wurde. Trifft dies zu, wird der zur Methodendefinition gehörende Code ausgeführt. Abbildung 6.2, zeigt, wie der Methodenaufruf für die Methode »laufen« letztlich die Definition der Methode in der Klasse »Hund« des angesprochenen Hunde-Objekts »Timmy« erreicht.

Der Methodenaufruf selbst wird gemeinhin nach einem Sender-Empfänger-Modell auf der konkret-individuellen Ebene der Instanzen beschrieben. An einem Methodenaufruf sind stets zwei Objekte beteiligt: der Sender, der einem Empfänger eine Nachricht zuschickt, in der er das Verlangen nach der Ausführung einer bestimmten Methode äußert. Der minimale Inhalt der Nach-

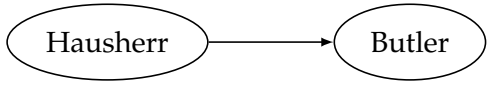

Abbildung 6.3: Einfacher Methodenaufruf

richt an das Objekt besteht in dem Namen der Methode, die aufgerufen werden soll. Die Objekte sind Instanzen verschiedener Klassen und schicken sich gegenseitig Nachrichten zu, nach denen sie handeln. Abb. 6.3 zeigt das Verhältnis beispielhaft anhand eines »Hausherr«-Objekts, das einem »Butler«-Objekt aufgibt, die Post zu holen.

Methoden werden in Ruby mithilfe von `def…end` definiert und mithilfe des Punkt-Operators aufgerufen. Vor dem Punkt steht der Empfänger des Methodenaufrufs, hinter dem Punkt der Name der aufzurufenden Methode. Auf das gezeigte Beispiel angewandt bedeutet dies:

```
# Neue Klasse "Butler"
class Butler

  # Definiere eine neue Methode namens "fetch_mail"
  # für Instanzen dieser Klasse (= Bauplanprinzip).
  def fetch_mail
    puts "Going to the post box"
    puts "Taking out the mail"
    puts "Bringing the mail back"
  end

end

# Erzeuge einen neuen Butler
hudson = Butler.new
# Schicke ihn zum Briefkasten
hudson.fetch_mail
```

Methoden müssen nicht statisch sein, sondern können mithilfe von Parametern je nach Ausgestaltung sehr unterschiedliche Aufgaben wahrnehmen. Hierauf wird im Kapitel über Methoden (→ 8) noch näher einzugehen sein.

Attribute werden in Ruby ebenfalls über besondere Methoden umgesetzt, d. h. Ruby hat keine »echten« Attribute wie andere Programmiersprachen.

Stattdessen gibt es in Ruby *Getter-Methoden* und *Setter-Methoden*, die ein Attribut auslesen oder setzen können. Gegenüber normalen Methoden gibt es nur den Unterschied, daß Rubys Syntax zum Zwecke besserer Lesbarkeit einen Aufruf erlaubt, der einer Zuweisung an eine Variable sehr ähnlich sieht. Ob die Methode aber wirklich einen Wert ausliest oder setzt, ist letztlich ihr überlassen.

Es ist möglich, Getter- und Setter-Methoden selbst nach den bereits beschriebenen Regeln zu schreiben (also mithilfe von `def`). Das ist in aller Regel aber unnötig aufwendig, weil sie für alle Attribute sehr ähnlich aussehen. Ruby bietet deshalb die Meta-Methoden der `attr`-Gruppe an, die die Definition dieser Methoden übernehmen. Dabei handelt es sich um eine einfache Form von Metaprogrammierung, mit der sich ein anderes Kapitel ausführlich beschäftigen wird **[TODO: Querverweis]**. Sie heißen im Einzelnen:

attr_reader: Diese Methode definiert einen Getter.

attr_writer: Diese Methode definiert einen Setter.

attr_accessor: Diese Methode definiert einen Getter und einen Setter.

attr: Diese Methode verhält sich im Wesentlichen so wie `attr_reader`, d. h. definiert einen Getter.

Unter Nutzung dieser Meta-Methoden könnte man beispielsweise einer Klasse `Dog` so ein Attribut `speed` (engl. »Geschwindigkeit«) verleihen:

```
# Neue Klasse "Dog"
class Dog
  # Definiert die Methoden speed (Getter) und speed= (Setter)
  attr_accessor :speed
end

# Erzeuge neue Instanz von "Dog"
timmy = Dog.new
# Weise dem Attribut "speed" mithilfe des Setters
# "speed=" den Wert 50 zu,
timmy.speed = 50
# Lese das Attribut "speed" mithilfe des Getters
# "speed" wieder aus
puts timmy.speed #=> 50
```

Das im Quelltext an `attr_accessor` übergebene `:speed` ist ein Symbol. Symbole sind letztlich eine besondere Form von Strings, die der Benennung von Dingen dienen. Zu Symbolen noch im Kapitel **[TODO: Querverweis]**.

6.3 self

Innerhalb des aktuellen Ausführungskontexts (*Binding*) kann eine Methode ohne ausdrücklichen Empfänger aufgerufen werden. Dies hat sich bereits bei Methoden wie `puts` gezeigt: auf der obersten Programmebene ist der Empfänger der Methode ein implizites Main-Objekt. Tatsächlich ist dies nur eine besondere Ausprägung der folgenden Regel: Wenn Ruby bei der Ausführung auf einen Methodenaufruf ohne ausdrücklichen Empfänger stößt (also ohne Punkt), dann wird die Methode auf `self` aufgerufen.

`self` ist eine Pseudovariable, d. h. verhält sich wie eine Variable, ohne wirklich eine zu sein. `self` verweist in Ruby stets auf das aktuell ausführende Objekt. Auf der obersten Programmebene ist dies das bereits erwähnte implizite Main-Objekt. Ruft man aber auf einem anderen Objekt eine Methode auf, dann verändert Ruby den aktuellen Ausführungskontext und legt `self` auf den Empfänger des Methodenaufrufs fest. Ruft man also auf einer Instanz der Klasse `Dog` die Methode `bark` auf, dann ändert Ruby den Ausführungskontext auf die Innenansicht der Instanz der Klasse `Dog` und `self` verweist auf die Instanz. Fortan werden Methodenaufrufe ohne Punkt nicht mehr an das implizite Main-Objekt weitergeleitet, sondern an das aktuell ausführende Objekt. Eine Klasse `Airplane` für Flugzeuge etwa kann den Startvorgang in viele Unteraufgaben zerlegen, die ihrerseits jeweils durch eine Methode repräsentiert werden. Innerhalb von Airplane#start wird das aktuell ausführende Objekt implizit als Empfänger von Methodenaufrufen angenommen, weshalb es eines expliziten Aufrufs per Punkt nicht bedarf:

```
class Airplane
  def start(runway)
    start_engine     # Gleich wie: self.start_engine
    move_to(runway)  # Gleich wie: self.move_to(runway)
    accelerate       # Gleich wie: self.accelerate
    # …
  end

  def start_engine
```

```
    # …
  end

  def move_to(runway)
    # …
  end

  def accelerate
    # …
  end
end
```

`self` kann überall eingesetzt werden, wo auch eine Variable zulässig ist. Typischerweise wird man es benutzen, um einer Methode (eines anderen Objekts) das aktuell ausführende Objekt zu übergeben oder um eine Ambivalenz wegen Namensgleichheit von Methode und lokaler Variable aufzulösen. Ein weiterer Einsatzzweck ist die Definition von Klassenmethoden, dazu später [**TODO: Querverweis**].

```
class Foo
  def bar
    # …
  end
  def baz
    bar = 3
    # …
    self.bar # erzwingt Methodenaufruf
  end
end
```

Im obigen Beispiel gibt es sowohl eine Methode Foo#bar als auch eine lokale Variable `bar` innerhalb der Methode Foo#baz. Weil bei Methodenaufrufen ohne Parameter in der Regel die Klammern weggelassen werden, ist der bloße Ausdruck bar nicht eindeutig. Er könnte sowohl die lokale Variable als auch die Methode Foo#bar meinen. Ruby entscheidet sich in einer solchen Situation stets für die lokale Variable. Ist das nicht gewünscht, so kann man entweder explizite Klammern benutzen, was bei einer parameterlosen Methode allerdings ungewöhnlich für Ruby aussieht (`bar()`) oder aber man teilt Ruby den Empfänger des Methodenaufrufs explizit mit (`self.bar`).

Bei `self` handelt es sich um ein Schlüsselwort. Es steht damit als Bezeichner für eigene Variablen und Methoden nicht zur Verfügung.

6.4 Vererbung

Unsere Wahrnehmung der Welt erschöpft sich nicht in der einmaligen Einteilung von Objekten in Gruppen, vielmehr wiederholen wir diesen Vorgang erneut und gruppieren die Gruppen in Obergruppen, welche ihrerseits wieder in Oberobergruppen zusamengefasst werden, u. s. w.Dalmatiner sind Hunde, Hunde sind Säugetiere, Säugetiere sind Tiere, Tiere sind Lebewesen, u. s. w.Ober- und Untergruppen können auch in der objektorientierten Programmierung mithilfe eines *Vererbung* genannten Mechanismus abgebildet werden, was erneut ein Beleg dafür ist, wie sehr die Objektorientierung an der menschlichen Wahrnehmung der Welt orientiert ist. Die Klassen der Objektorientierung stehen als Eltern- und Kindklassen in einem hierarchischen Verhältnis, welches der soeben erläuterten Ober/Untergruppeneinteilung entspricht.

Technisch »vererben« die Elternklassen ihren Kindklassen all ihre Methoden und Attribute. Definiert die Klasse »Hund« vier Beine für Objekte dieser Klasse, so enthält auch die von der Klasse »Hund« abgeleitete Kindklasse »Dalmatiner« dieses Attribut, ebenso wie eine weitere Kindklasse »Dackel«. Leitet man vom »Dackel« noch den »Rauhaardackel« ab, so erbt auch er das Attribut. Gleiches gilt für Methoden wie etwa »beißen«. Abbildung 6.4 zeigt eine mehrstufige Klassenhierarchie mit Instanzen der untersten Klassen.

Wird der Aufruf einer Methode von einer Instanz einer Kindklasse verlangt, so wird zunächst geprüft, ob die Kindklasse diese Methode definiert. Definiert sie sie nicht, so wird anschließend — dies ist der eigentliche Vererbungseffekt — die Elternklasse geprüft. Definiert auch diese nicht die gewünschte Methode, wird wiederum in deren Elternklasse nachgesehen u. s. w.bis schließlich das Ende der Klassenhierarchie erreicht ist. Wird die Methode auf dem Weg dorthin gefunden, wird deren Code ausgeführt, wird sie nicht gefunden, ist dies ein Fehler. Dabei ist es möglich, dass Kindklassen dieselbe Methode neu definieren, die bereits eine Elternklasse definiert hatte; zwar können alle Hunde schwimmen, der Labrador aber kann es dank der Schwimmhäute zwischen seinen Zehen besonders gut, was eine Redefinition der Methode »schwimmen« aus der Klasse »Hund« in der Klasse »Labrador« rechtfertigt. In Abb. 6.5 wird auf der linken Seite eine Methode aufgerufen, die in der Kindklasse »Labrador« neu definiert wurde — man beachte, wie der Methodenaufruf nie die in der Elternklasse definierte Methode gleichen Namens

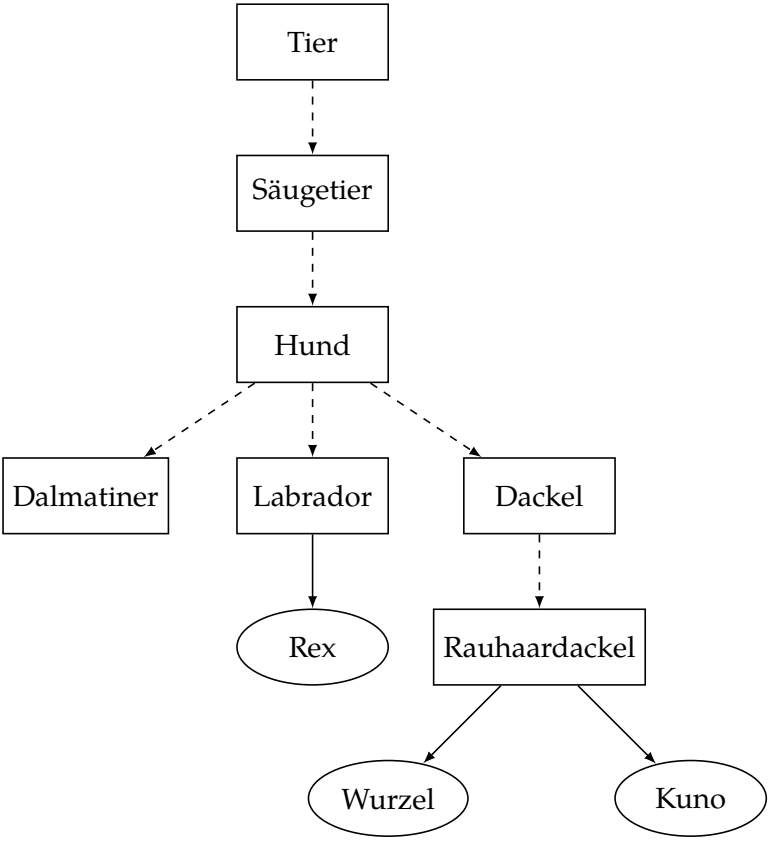

Abbildung 6.4: Verhältnis von Hunderassen

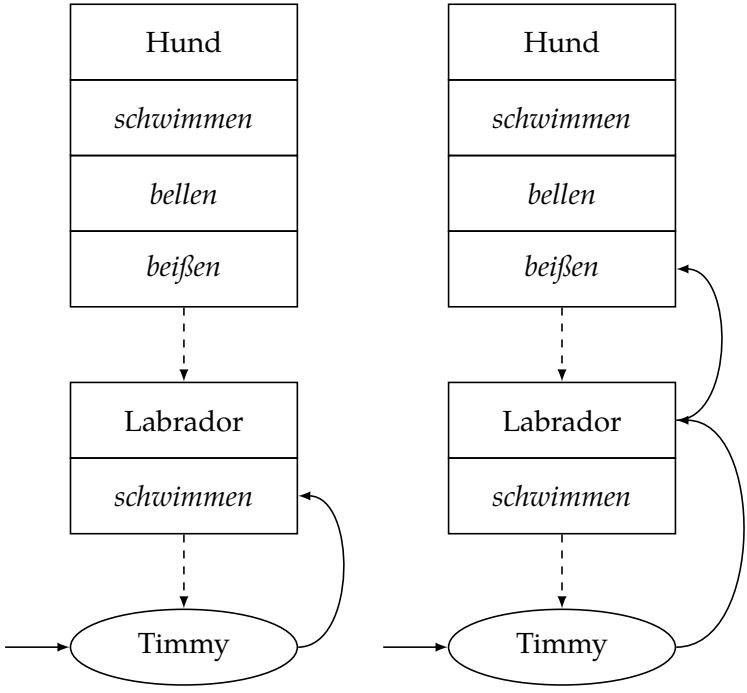

Abbildung 6.5: Methodenaufruf bei Vererbung

erreicht. Auf der rechten Seite dagegen ist der Aufruf einer in der Kindklasse nicht definierten Methode dargestellt, welcher bis zur Elternklasse durchgereicht werden muss.

Vererbung kann man in Ruby durch Einfügung des Operators < gefolgt vom Namen der Elternklasse in die Klassendefinition erreichen. Das Labrador-Beispiel sieht in Ruby wie folgt aus:

```
# Neue Klasse "Dog" mit zwei gewöhnlichen Methoden
class Dog

  def swim
    puts "swim in Dog"
  end

  def run
    puts "run in Dog"
```

```
  end

end

# Neue Klasse "Labrador", die von der Klasse "Dog" erbt.
class Labrador < Dog

  # "Labrador" überschreibt die Methode "swim", die es
  # von "Dog" geerbt hat.
  def swim
    puts "swim in Labrador"
  end

end

timmy = Labrador.new

# "run" ist vererbt worden; es wird die Methode aus der
# Elternklasse ausgeführt. "swim" dagegen wurde über-
# schrieben und nur die speziellere Fassung aus "Labrador"
# wird ausgeführt.
timmy.run  #=> run in Dog
timmy.swim #=> swim in Labrador
```

Statt eine Methode ganz zu überschreiben, kann eine Kindklasse auch auf der Methode der Elternklasse aufbauen, indem mithilfe des Schlüsselworts `super` die Elternmethode aufgerufen wird. Außerdem ist es möglich, geerbte Methoden mit `undef` gezielt aus der Kindklasse zu entfernen. Einzelheiten → 8.8.2 (super), → 8.1 (undef).

In Ruby erben im Übrigen alle Klassen, bei denen keine ausdrückliche Elternklasse angegeben wurde, automatisch von der Klasse `Object`. `Object` steht deshalb an der Spitze der regulären Vererbungshierarchie. Diese Klasse definiert einige wenige, aber wichtige Methoden wie etwa `inspect` (hübsche Beschreibung des Objekts) und `object_id` (eindeutige numerische Kennziffer des Objekts). `Object` mischt zudem das Modul `Kernel` ein (→ 9.6.1).

6.5 Geheimnisprinzip

Die objektorientierte Programmierung wird von einem als *Geheimnisprinzip* bezeichneten Leitsatz beherrscht, der verlangt, daß die einzelnen Objekte nur über die inneren Strukturen ihrer selbst Kenntnis besitzen. Im Idealfall sind die Objekte vollständig voneinander abgeschottet und kommunizieren ausschließlich über dafür vorgesehene Methoden miteinander. Es erscheint auch logisch, wenn das Katzen-Objekt nicht über den Inhalt, die Struktur und Funktionsweise des von ihm benutzten Futterautomat-Objekts Bescheid weiß, sondern sich auf die Kenntnis der offiziell zugänglichen Kommunikationswege (Methoden) beschränkt. Ebenso wissen die meisten Kfz-Halter wenig über die technische Funktionsweise ihres Wagens, sondern nutzen ihn nur über die dafür vorgesehenen Einrichtungen (Gaspedal, Bremse, Lenkrad, ...). Anders ausgedrückt: alles, was den inneren Aufbau eines Objekts und die Funktionsweise seiner Methoden betrifft, bleibt das Geheimnis des jeweiligen Objekts. Nach außen hin bekanntgegeben werden ausschließlich dafür vorgesehene Methoden. Die sich aus dieser logischen Abkapselung ergebene sehr angenehme Konsequenz ist, daß unerwartete Eingriffe in das Objekt von außen nicht möglich sind, sondern einen Fehler erzwingen. Innerhalb der Implementation eines Objekts kann man sich als Programmierer daher darauf verlassen, daß das Objekt sich nur dann verändert, wenn es in zuvor festgelegter Weise über die nach außen bekanntgegebenen Methoden angesprochen wird. Dies erleichtert die Fehlerbehebung ganz erheblich[4].

Das Geheimnisprinzip ist in seiner vollen Ausdrucksweise nicht immer durchzuhalten, dennoch ist es ein Leitsatz, an dem orientiert man üblicherweise gut organisierte Programme erstellen kann. Wie mit allen Prinzipien gilt, dass Abweichungen möglich sein müssen — diese aber einer besonderen Begründung bedürfen.

In Ruby können bereits von Haus aus Objekte nicht in die Implementation anderer Objekte hineinschauen, solange sie sich nicht besonderer Tricks aus dem Bereich der Metaprogrammierung bedienen. Darüber hinaus kennt Ruby mit `private`, `protected` und `public` besondere Schlüsselwörter, mit denen die Sichtbarkeit einer Methode nach außen hin ausdrücklich festgelegt werden kann. Dazu mehr im Kapitel über Methoden [**TODO: Querverweis**].

[4] Dennoch ist das Geheimnisprinzip nicht unbestritten. Weil es in der Praxis recht häufig vorkommt, dass Fehlerursachen sich quer über Objektgrenzen hinwegziehen, wird sogar argumentiert, das Geheimnisprinzip *erschwere* die Fehlerbehebung unter dem Gesichtspunkt, daß man die einzelnen Objekte bei der Fehlersuche doch wieder alle aufklappen müsse. Dabei handelt es sich aber wohl nicht um eine weit verbreitete Meinung.

§ 7 Variablen und Konstanten

Ruby bietet eine erhebliche Menge verschiedener Variablenarten, die ein eigenes Kapitel zu diesem Thema rechtfertigt. Allen Variablen ist gemeinsam, daß in Ruby nicht — wie z. B. in C — zwischen der Deklaration und der Definition unterschieden wird. Die erste zulässige Verwendung ist Deklaration und Definition zugleich.

Nachfolgend werden alle in Ruby möglichen Variablenarten vorgestellt. Am Ende des Kapitels faßt Tafel 7.4 die Informationen noch einmal zusammen.

7.1 Lokale Variablen

Lokale Variablen bilden den Grundtypus aller Variablen und stellen zugleich die meistgenutzte Variablenart dar. Sie sind von ihrer erstmaligen Verwendung an gültig bis zum Ende des entsprechenden Gültigkeitsbereichs. Tafel 7.1 listet die Ausdrücke auf, die in Ruby einen neuen Gültigkeitsbereich beginnen. Darunter befindet sich insbesondere nicht `if`, was gelegentlich Verwirrung stiftet.

```
def foo
  x = 3
  puts x
end
# x ist hier nicht mehr gültig

if true # Kein neuer Gültigkeitsbereich
  y = 10
```

class	def
do[1]	module

Tabelle 7.1: Gültigkeitsbereiche

```
end
puts(y) #=> 10
```

Es ist bis auf eine Ausnahme nicht möglich, Gültigkeitsbereiche zu verschachteln. Alle in 7.1 genannten Schlüsselwörter bis auf do ändern den aktuellen Gültigkeitsbreich derart, daß ein Zugriff nicht möglich ist:

```
class C
  x = 3
  def foo
    puts x # NameError
  end
end
```

Die mit do...end bzw. den geschweiften Klammern {} definierten Blöcke sind Closures (→ 8.7) und bilden damit eine Ausnahme, die vollen Zugriff auf den sie umgebenden Gültigkeitsbereich hat.

```
x = 3
1.times{x=4}
puts(x) #=> 4
```

Das kann man verhindern, indem man die ausdrücklich zu kopierenden Variablen als besondere Blockparameter mit einem Semikolon von etwaigen normalen Blockparametern abgrenzt. Gibt es keine normalen Blockparameter, darf die Liste vor dem Semikolon leer bleiben.

```
x = 3
1.times{|;x| x = 4}
puts(x) #=> 3
```

Besitzt der Block eine eigene lokale Variable mit demselben Namen wie ein umgebender Gültigkeitsbereich, dann geht diese vor (*Shadowing*). Ruby gibt in diesen Fällen eine Warnung aus, wenn es im Diagnosemodus läuft.

```
x = 3
2.times {|x| puts(x)}
puts x # Es gilt wieder das originale x
```

Ausgabe:

```
/tmp/x.rb:2: warning: shadowing outer
local variable - x
0
1
3
```

Wie dieses Beispiel für Blockparameter zeigt, gibt es zwischen Parametern — gleich, ob Block- oder Methodenparameter — und lokalen Variablen keine nennenswerten Unterschiede. Lediglich die Erstzuweisung wird bei Parametern von Ruby selbst vorgenommen statt durch den Programmierer.

Die Verwendung noch nicht zugewiesener lokaler Variablen ist unzulässig und verursacht einen NameError.

```
puts gibtsnich #=> NameError
```

Kollidiert eine lokale Variable mit einem Methodennamen im aktuellen Gültigkeitsbereich (implizites `self`, → 6.3), entscheidet Ruby unter Anwendung der folgenden Regeln:

1. Eine Zuweisung meint immer eine lokale Variable.
2. Werden Parameter übergeben, handelt es sich um einen Methodenaufruf.
3. Bei Verwendung von Klammern (auch leeren) liegt ebenfalls ein Methodenaufruf vor.
4. Sonst wird die lokale Variable benutzt.

Aus diesen Regeln folgen zwei Fälle, bei denen man den Konflikt manuell auflösen muß:

1. Eine Methode soll ohne Argumente aufgerufen werden, wenn eine gleichnamige lokale Variable existiert.
2. Eine Setter-Methode soll auf dem aktiven Objekt aufgerufen werden.

Beispiel für Fall 1:

```
class Car
  def accel
    # …Berechne aktuelle …Beschleunigung
  end
  def move_to(point)
    accel = calc_required_accel(point)
    puts "Increasing #{accel} by #{accel}"
    # …
  end
end
```

Ruby würde in `move_to` zweimal denselben Wert ausgeben, was nicht der Absicht des Programmierers entspricht. Das erste `accel` meint die Methode und ist deshalb durch `self.accel` zu ersetzen.

```
puts "Increasing #{self.accel} by #{accel}"
```

Beispiel für Fall 2:

```
class Crane
  def ypos
    @ypos ||= 0
  end
  def ypos=(val)
    if val > 100
      raise RangeError, "y is greater than max Y 100"
    end
    @ypos = val
  end
  def move_up(delta)
    ypos += delta
  end
end
```

In diesem Fall will der Programmierer den Seiteneffekt von Crane#ypos=, die Prüfung des Wertebereichs, ausnutzen und weist deshalb nicht direkt an die Instanzvariable @ypos zu (zu Instanzvariablen sogleich). Dies schlägt jedoch fehl, weil Ruby den Ausdruck `ypos += delta` als Zuweisung einer lokalen Variablen wertet. Abhilfe schafft die ausdrückliche Benutzung von `self`.

```
def move_up(delta)
  self.ypos += delta
end
```

7.2 Instanzvariablen

Instanzvariablen sind in ihrer Gültigkeit an das Objekt gekoppelt, dem sie zugeordnet sind. Verfällt das Objekt, verfallen auch sie. Haupteinsatzzweck von Instanzvariablen ist die Umsetzung von Attributen (→ 6.2).

Instanzvariablen sind mit einem führenden @-Zeichen zu versehen. Nicht zugewiesene Instanzvariablen dürfen verwandt werden und haben dann den Wert nil, allerdings gibt Ruby im Diagnosemodus dann eine Warnung aus. Die Meta-Methoden der attr-Gruppe lesen und schreiben Instanzvariablen in den von ihnen erstellten Methoden.

```
class Dog
  def initialize
    @velocity = 0
  end
  def accelerate(accel)
    @velocity += accel
  end
end
```

Weil Klassen und Module auch nur Instanzen ihrer jeweiligen Klassen `Class` und `Module` sind, kann man auch auf Klassen- bzw. Modulebene Instanzvariablen einsetzen. Diese sind dann aber keinesfalls automatisch auch für Instanzen verfügbar, da es sich ja um ganz andere Objekte handelt. Solche »Klasseninstanzvariablen« bzw. »Modulinstanzvariablen« sind praktisch, um globale, veränderliche Informationen zu erfassen, die im Zusammenhang mit der duch die Klasse bzw. das Modul abgebildeten Funktionalität stehen. So kann man beispielsweise leicht zählen, wie oft eine Klasse instanziiert wurde:

```
class Tank
  def self.count
    @count ||= 0
  end
  def self.count=(val)
```

```
    @count = val
  end
  def initialize
    self.class.count += 1
  end
end
```

7.3 Klassenvariablen

Im Grunsatz ist eine Klassenvariable sowohl innerhalb der Klasse als auch in ihren Instanzen gültig. Sie beginnt mit einem doppelten @@ und die Nutzung einer nicht zugewiesenen Klassenvariable verursacht einen NameError. Das Zählbeispiel aus Abschnitt 7.2 läßt sich mit einer Klassenvariable scheinbar einfacher wie folgt implementieren:

```
class Tank
  @@count = 0
  def initialize
    @@count += 1
  end
end
```

Leider ist damit noch nicht alles über den Gültigkeitsbereich von Klassenvariablen gesagt. Sie weisen Probleme auf, die die meisten Programmierer dazu bewegen, dieses Feature von Ruby als verunglückt anzusehen und von ihrem Gebrauch generell abzuraten. Zunächst einmal sind sie de facto eigentlich globale Variablen und unterliegen damit den für globale Variablen geltenden Vorsichtsregeln (unten → 7.4), obwohl es auf den ersten Blick nicht so aussieht. Weit schwerer wiegt jedoch, daß Klassenvariablen bei Einsatz von Vererbung einen sehr überraschenden Gültigkeitsbereich aufweisen, denn: sie vererben sich in der Klassenhierarchie *nach oben*, d. h. eine Änderung der Klassenvariable in einer untergeordneten Klasse hat Auswirkungen auf die gesamte Klassenhierarchie unterhalb der Klasse, in der die Klassenvariable erstmals definiert wurde.

Erweitert man das gegebene Zählbeispiel um Vererbung, zeigt sich das Problem:

```
class Tank
```

```
  @@count = 0
  def initialize
    @@count += 1
  end
end

class LeopardTank < Tank
  @@count = 0
  def initialize
    @@count += 1
  end
end

class OtherTank < Tank
end

l1 = LeopardTank.new
l2 = LeopardTank.new

class OtherTank
  p @@count #=> 2
end
```

@@count existiert hier nämlich keinesfalls zweimal oder gar dreimal. Es handelt sich um dieselbe Klassenvariable, die Gültigkeit für die gesamte Klassenhierarchie unter `Tank` beansprucht. Deshalb wird auch die scheinbar völlig unbeteiligte Klasse `OtherTank` in Mitleidenschaft gezogen, obwohl es in diesem Beispiel offenbar Absicht des Programmierers war, eigentlich nur die Instanzen von `LeopardTank` zu zählen.

Ein zweites Beispiel:

```
class Bridge
  @@costs_for_new = 1_000_000
  def self.costs_for_new
    @@costs_for_new
  end
end
class RailBridge < Bridge
  @@costs_for_new = 2_000_000
```

```
end
p Bridge.costs_for_new #=> 2000000
```

Durch Änderung von @@costs_for_new in der Subklasse wird die Variable innerhalb der gesamten Klassenhierarchie ab `Bridge` geändert. Aufgrund dieser Eigenart sind Klassenvariablen gut geeignet, unvorhergesehene Probleme in den eigenen Code einzufügen.

Eine Mindermeinung verteidigt die Nutzung von Klassenvariablen damit, daß die Alternative, nämlich die Nutzung von Klasseninstanzvariablen, mehr Schreibaufwand bedeute und bei richtiger Benutzung von Klassenvariablen die beschriebenen Probleme umgangen würden. Richtige Nutzung heißt nach Ansicht dieser Personen, daß man Klassenvariablen nach Berücksichtigung der für globale Variablen geltenden Vorsichtsmaßnahmen in Subklassen niemals setzen darf, da somit die beschriebenen Probleme nicht auftreten. Dieser Auffassung ist zuzugeben, daß die Fälle der Zuweisung in Subklassen oft genug gar kein Problem für Klassenvariablen sind, sondern durch Einsatz von Konstanten (→ 7.5) semantisch korrekt gelöst werden können und sollten, etwa so:

```
class Bridge
  COSTS_FOR_NEW = 1_000_000
end
class RailBridge < Bridge
  COSTS_FOR_NEW = 2_000_000
end
p Bridge::COSTS_FOR_NEW #=> 1000000
```

Damit sind aber nicht alle Fälle abgedeckt. Das oben gegebene Zählbeispiel kann mit einer Konstanten semantisch nicht akkurat gelöst werden, weil man den Wert von Konstanten nicht ändern sollte. Richtigerweise sollte man hier trotz des erweiterten Schreibaufwands auf Klasseninstanzvariablen zurückgreifen, die sich vorhersehbarer verhalten, insbesondere nicht in der gesamten Klassenhierarchie geteilt werden. Derartige Einsatzfälle sind selten genug, um den Mehraufwand von Klasseninstanzvariablen zu rechtfertigen. Weiterhin läßt die Sprache diese problematische Zuweisung gerade ausdrücklich zu, d. h. verführt geradezu dazu, auf das Vererbungsproblem zu stoßen. Weil die die detaillierte Kenntnis über den Gültigkeitsbereich von Klassenvariablen kaum von jedem Rubyisten erwartet werden kann, ist es einfacher, von ihrer Nutzung einfach ganz abzusehen. Schließlich muß man berücksichtigen, daß speziell bei Programmbibliotheken das Risiko besteht, daß aufgrund einer

Namenskollision mit einer vom Nutzer der Programmbibliothek in einer Subklasse verwandten Klassenvariable ungewollt Änderungen in der Programmbibliothek stattfinden, die zu schwer erkennbaren Fehlern führen können (auf das obige Zählbeispiel bezogen ergäbe sich dieses Problem, wenn `Tank` in dieser Form durch eine Programmbibliothek zur Verfügung gestellt würde und die beiden anderen Klassen Subklassen des Nutzers wären).

Der Einsatz von Klassenvariablen sollte daher ganz vermieden werden.

7.4 Globale Variablen

Globale Variablen gelten weltweit. Jede Änderung wirkt sich auf alle Ruby-Programme auf der ganzen Welt aus.

...das ist natürlich nicht der Fall, soll aber die Gefährlichkeit globaler Variablen verdeutlichen. Die mit einem Dollarzeichen $ beginnenden Variablen beanspruchen Geltung im gesamten Ruby-Programm, völlig unabhängig von jeglichen Gültigkeitsbereichen. Sie können ohne Zuweisung benutzt werden und haben dann den Wert nil, was im Diagnosemodus (→ 12.3.1) aber mit einer Warnung moniert wird.

```
def foo
  $myglobal = 3
end

foo
p $myglobal #=> 3
```

Globale Variablen üben eine geradezu magische Anziehungskraft auf Anfänger aus, weil man sich damit über die scheinbar so komplizierten und unnötigen Gültigkeitsbereiche hinwegsetzen kann. Es muß daher in aller Deutlichkeit gesagt werden, daß der übermäßige und unbedachte Einsatz von globalen Variablen ein zuverlässiges Signal zur Erkennung untauglicher Programmierer ist. Gültigkeitsbereiche haben nicht den Zweck, den Programmierer zu ärgern, sondern dienen dazu, logische Einheiten zu bilden, die man als Programmierer bei der Arbeit schnell vollständig überblicken kann.

Hieraus ergeben sich denn auch die Hauptgefahr von globalen Variablen. Ihr unbeschränkter Gültigkeitsbereich macht sie überall im Programm verfügbar, auch dort, wo man gar nicht mit ihnen gerechnet hätte und eigentlich eine neue Variable gleichen Namens benutzen wollte (Namenskollision).

Diese Gefahr ist bei Einsatz von Programmbibliotheken besonders hoch, da man in deren Code üblicherweise keine Einsicht nimmt. Darüber hinaus kann Code an ganz anderen Stellen im Programm eine Änderung an einer globalen Variablen vornehmen, mit der man an der konkreten, derzeit im Bearbeitung stehenden Stelle einfach nicht gerechnet hatte (Seiteneffekte). Schließlich verschmutzen sie den globalen Gültigkeitsbereich, da man sie nach dem Gebrauch nicht mehr loswird (Speicherverbrauch). Mehr als auf nil setzen kann man sie nicht, und da auch dies oft genug vergessen wird, zeigt eine abgefertigte globale Variable oft genug auf eine umfangreiche Instanz, die so dem Garbage Collector [**TODO: Querverweis**] vorenthalten wird.

Nichtsdestoweniger *gibt* es Anwendungsfälle für globale Variablen. Kosten und Nutzen globaler Variablen müssen aber in jedem Einzelfall sorgsam abgewogen werden. Für den Anfänger gilt, daß er die Finger davon lassen sollte. Jedes Konstrukt, daß man mit einer eigenen globalen Variable ausdrücken kann, kann man auch auf andere Weise realisieren. Der Einsatz eigener globaler Variablen dienen damit der Abkürzung komplexer Prozesse für den erfahrenen Programmierer.

Das vermutlich wichtigste Beispiel für den sinnvollen Einsatz einer eigenen globalen Variablen ist ein Logging-Objekt, wobei auch dessen Notwendigkeit von einigen noch bestritten wird. Vertretbar ist es bei ausreichender Dokumentation der globalen Variable aber. Die in Rubys Standardbibliothek enthaltene Bibliothek logger kann in Kombination mit einer eigenen globalen Variablen genutzt werden, um an beliebiger Stelle im Code die Ausgabe in eine Logdatei zu ermöglichen.

```
require "logger"

$log = Logger.new("/var/log/myapp.log")
# …
class Something
  def doit
    $log.info "Doing something"
    # …
  end
end
```

Eine Alternative zur globalen Variablen wäre hier die Bereitstellung eines Logging-Moduls, das als Modulmethode Logging.info anbieten würde, welches dann wiederum an die Logger-Instanz delegieren könnte. Das erscheint

unnötig aufwendig. Durch Einsatz der globalen Variablen $log wird der ohnehin oft genug lästige Logging-Code kurz und prägnant gehalten.

Bereits bei Programmstart sind von Ruby eine ganze Reihe von globalen Variablen vordefiniert. Ein Rückgriff auf diese ist stets unschädlich, denn als offiziell von Ruby bereitgestellte und dokumentierte Variablen sollten sie jedem Rubyisten bekannt sein. Die vollständige Liste aller vordefinierten globalen Variablen kann Tafel 7.2 entnommen werden. Die kryptischen Namen vieler dieser Variablen gehen auf Perl zurück, an dem Ruby sich diesbezüglich orientiert. Es gibt in der Standardbibliothek allerdings eine Programmbibliothek namens English (mit großem E), die einigen dieser Variablen eine lesbare Alternative an die Hand gibt. Auf die Dokumentation von English sei an dieser Stelle verwiesen.

Tabelle 7.2: Vordefinierte globale Variablen

Variable	Beschreibung
$!	Letzter Laufzeitfehler
$@	Backtrace des letzten Laufzeitfehlers als Array
$&	Kompletter String aus dem letzten Regexp-Match
$‘	String links vom letzten Regexp-Match
$’	String rechts vom letzten Regexp-Match
$+	Höchste Gruppe des letzten Regexp-Match
$1…$9	Match-Gruppe mit der entsprechenden Nummer
$~	MatchData-Instanz für den letzten Regexp-Match
$=	Veraltet und ohne Effekt.
$/	Eingabeseparator (Standardwert: Zeilenumbrauch)
$\	Ausgabeseparator u. a. für #print (Standardwert: nil)
$,	Ausgabefeldseparator für #print und #Arrayjoin
$;	Standardseparator für #Stringsplit
$.	Aktuelle Zeilennummer in der zuletzt gelesenen Datei
$<	Entspricht ARGF.
$>	Standardausgabe für #print(f) (Standardwert: $stdout)
$_	Zuletzt von #gets oder #readline gelesene Zeile
$0	Programmname
$*	Entspricht ARGV.
$$	Prozeßnummer des Ruby-Interpreters
$?	Endstatus des letzten Kindprozesses (threadlokal)
$:	Suchpfade für #require als Array
$”	Alle geladenen Ruby-Dateien als Array

Fortsetzung nächste Seite

Variable	Beschreibung
$-0	Alias für $/
$-a	Wahr, wenn Ruby mit »-a« aufgerufen wurde
$-d	Alias für $DEBUG
$-F	Alias für $;
$-i	Beim Soforteingabemodus die aktuelle Zeile
$-I	Alias für $:
$-l	Wahr, wenn Ruby mit »-l« aufgerufen wurde
$-p	Wahr, wenn Ruby mit »-p« aufgerufen wurde
$-v	Alias für $VERBOSE
$-w	Alias für $VERBOSE
$DEBUG	Debug-Modus aktiviert?
$LOADED_FEATURES	Alias für $"
$FILENAME	Entspricht `$<.filename`
$LOAD_PATH	Alias für $:
$SAFE	Sicherheitsstufe
$stderr	Standardfehlerausgabe
$stdin	Standardeingabe
$stdout	Standardausgabe
$VERBOSE	Diagnosemodus aktiviert?

7.5 Konstanten

Viele Programmiersprachen bieten neben Variablen auch Konstanten an. Dort unterscheiden sich Konstanten von Variablen dann vornehmlich darin, daß nach der Erstzuweisung eine Änderung der Konstanten nicht mehr möglich ist. Zwar bietet auch Ruby Konstanten an, allerdings sind diese nicht so konstant wie es die Bezeichnung vermuten läßt.

Konstanten in Ruby sind auf den Gültigkeitsbereich beschränkt, in dem sie definiert wurden. Das kann eine Klasse, ein Modul oder die oberste Ebene sein, wobei die oberste Ebene identisch ist mit einer Definition in der Klasse `Object`, die als Wurzelklasse für die meisten Ruby-Klassen dient (→ 6.4). Eine Definition innerhalb einer Methode verletzt Rubys Sprachgrammatik.

Jeder Bezeichner, der mit einem Großbuchstaben beginnt und der kein Methodenname ist, ist eine Konstante. Deshalb sind die Bezeichner von Klassen und Modulen Konstanten, auch wenn man darüber im normalen Gebrauch der Sprache nicht nachdenkt. Wichtiger sind die ausdrücklich für normale Objekte definierten Konstanten, die man qua Konvention vollständig in Groß-

buchstaben schreibt.

Der wichtigste Einsatzzweck von Konstanten ist die Festschreibung fixer, im Programmablauf unveränderlicher Werte. Sie treten dann an die Stelle sog. »magischer Zahlen« oder sonstiger »magischer Werte«, deren Bedeutung der Programmierer sonst nicht erkennen kann bzw. die man in einem Kommentar festhalten müßte. Wenn ein Kran etwa nicht über eine bestimmte Höhe hinaus gefahren werden kann (weil alle Kräne im Programm nur diese maximale Größe haben), dann ist diese statische, unveränderliche Grenze als Konstante zu definieren und nicht als magische Zahl bei der Überprüfung der Änderung der Höhe, sodaß eine »magische Zahl« 100 vermieden wird:

```
class Crane
  MAX_HEIGHT = 100

  def move_to(y)
    @y ||= 0
    raise "Max height reached" if @y > MAX_HEIGHT
                       # statt: if @y > 100
    @y = y
  end
end
```

Der Zugriff von außerhalb auf Konstanten in einer Klasse oder einem Modul ist über den Auflösungsoperator :: möglich.

```
p Crane::MAX_HEIGHT #=> 100
```

Das kann man verhindern, indem man die Konstante als privat markiert. Ein Zugriff ist dann nur noch innerhalb des betroffenen Gültigkeitsbereichs möglich, nicht mehr von außen. Es ist nicht möglich (und wäre auch nicht sinnvoll), Konstanten ähnlich wie Methoden als `protected` zu deklarieren.

```
class Crane
  MAX_HEIGHT = 100
  private_constant :MAX_HEIGHT
end
p Crane::MAX_HEIGHT
#=> private constant Crane::MAX_HEIGHT
#=> referenced (NameError)
```

Konstanten werden im Gegensatz zu Variablen von Ruby lexikalisch aufgelöst. Weil dies typischerweise für Klassen und Module relevant wird, wird das Konzept dort erläutert (→ 9.1). Die dortigen Ausführungen gelten aber für alle Konstanten. Deshalb ist das folgende nicht möglich:

```
module MyLibrary
  VERSION = "1.0.0"
end
class MyLibrary::Crane
  def initialize
    puts "Crane from library version #{VERSION}"
  end
end
MyLibrary::Crane.new
#=> /tmp/beispiel.rb:6:in `initialize': uninitialized
#=> constant MyLibrary::Crane::VERSION (NameError)
```

Wie oben angedeutet sind Konstanten in Ruby nicht wirklich konstant. Dabei ist zwischen zwei Ebenen zu unterscheiden. Auf der einen Seite ist es natürlich möglich, auf dem referenzierten Objekt Methoden aufzurufen, die das Objekt selbst ändern. Das betrifft insbesondere Strings, Arrays und Hashes. Dem kann durch Aufruf der Methode #freeze Abhilfe geschafft werden, die derartige Veränderungen unterbindet.

```
ALLOWED_COMMANDS = ["print", "exit"]
ALLOWED_COMMANDS << "hello" # OK
ALLOWED_COMMANDS.freeze
ALLOWED_COMMANDS << "goodybe"
#=> can't modify frozen Array (RuntimeError)
```

Auf der anderen Seite kann man aber auch die Konstante selbst neu zuweisen. Das ist unabhängig von #freeze, weil es die Konstante und nicht das gehaltene Objekt betrifft.

```
ALLOWED_COMMANDS = ["print", "exit"]
ALLOWED_COMMANDS << "hello" # OK
ALLOWED_COMMANDS.freeze
ALLOWED_COMMANDS = ["print", "exit", "hello", "goodbye"]
```

Konstante	Beschreibung
ARGF	Konkatenierung aller Dateien in ARGV.
ARGV	Alle Kommandozeilenargumente als Array.
DATA	Alles nach __END__ als String.
ENV	Hash-ähnliches Objekt für Umgebungsvariablen.
FALSE	Entspricht false.
NIL	Entspricht nil.
RUBY_PLATFORM	Benutztes Betriebssystem.
RUBY_RELEASE_DATE	Veröffentlichungsdatum des Interpreters.
RUBY_VERSION	Version des Interpreters.
STDERR	Terminalfehlerausgabe-Stream.
STDIN	Terminaleingabe-Stream.
STDOUT	Terminalausgabe-Stream.
TRUE	Entspricht true.

Tabelle 7.3: Vordefinierte globale Konstanten

Dies kann man nicht verhindern. Das ist aber so schlechter Stil, daß Ruby in diesem Fall eine Systemwarnung (→ 12.3.1) ausgibt.

```
beispiel.rb:4: warning: already initialized
constant ALLOWED_COMMANDS
beispiel.rb:1: warning: previous definition of
ALLOWED_COMMANDS was here
```

Obwohl möglich, sollte man daher von der Neuzuweisung von Konstanten keinen Gebrauch machen.

Ähnlich wie bei den globalen Variablen definiert Ruby bei Programmstart bereits einige Konstanten auf der obersten Ebene. Die vollständige Liste kann Tafel 7.3 entnommen werden. Erläuterungen werden in diesem Buch an der jeweils passenden Stelle vorgenommen.

Art	Beispiel	Gültigkeit	Erstwert
Lokale Variable	var	Aktueller GB	Unzulässig
Instanzvariable	@var	Aktuelle Instanz	nil
Klassenvariable	@@var	Klasse und Instanz	Unzulässig
Globale Variable	$var	Gesamtes Programm	nil
Konstante	CONST	Gesamtes Programm	Unzulässig

Tabelle 7.4: Zusammenfassung Variablen

§ 8 Methoden

Methoden sind dem Grunde nach wiederverwertbare Code-Bündel. Ihr Stellenwert und ihre Konzeption im Rahmen der objektorientierten Programmierung wurde bereits an anderer Stelle ausführlich besprochen (→ 6.2). In diesem Kapitel soll der Blick auf die technischen Möglichkeiten der Methodendefinition gelenkt werden. Ruby bietet ein reichhaltiges Repertoire an Definitionstechniken, über die man schnell den Überblick verliert.

8.1 Grundlagen

Eine Methodendefinition wird mit dem Schlüsselwort `def` eingeleitet und mit `end` beendet. Alles zwischen diesen beiden Ankerpunkten gehört zur Methode und bildet ihren Methodenkörper. Wer einen mathematischen Blickwinkel bevorzugt, wird zu sich zu recht an Funktionen erinnert fühlen, jedoch sei deutlich darauf hingewiesen, daß die Konzepte nicht deckungsgleich sind. Das liegt an den sogenannten *Seiteneffekten*: während mathemathische Funktionen nur einen Rückgabewert haben, kommt es in Ruby — wie in den meisten Programmiersprachen — oft auf einen Effekt an, der außerhalb der Berechnung des Rückgabewertes liegt. So ist etwa im Falle von `puts` der Rückgabewert (nil) sogar gänzlich uninteressant. Es kommt allein auf den Seiteneffekt, der Ausgabe des Argumente auf der Standardausgabe, an.

In Ruby hat jede Methode einen Rückgabewert. Dieser kann entweder ausdrücklich mithilfe von `return` festgelegt oder implizit dem letzten Ausdruck der Methode entnommen werden. Eine leere Methode hat den Rückgabewert nil.

```
# Rückgabewert nil
def foo
end

# Rückgabewert 3
def foo
```

```
  1 + 2
end

# Ebenso
def foo
  return 1 + 2
end
```

`return` bricht die-Ausführung der Methode unmittelbar ab. Das kann man sich zunutze machen, um sich verschachtelte Bedingungsbäume zu ersparen und den Lesefluß zn sauber zu halten.

```
def foo
  return nil if problem?
  do_something
end
```

Gibt in diesem Beispiel `problem?` einen wahren Wert zurück, greift die Bedingung (Postcondition, → 5.1.1) ein und die Methode endet mit Rückgabewert nil. Nur, wenn `problem?` einen falschen Wert zurückliefert, wird mit `do_something` fortgefahren. Mangels weiterem return-Ausdruck bildet der Rückgabewert von `do_something` in diesem Beispiel auch den Rückgabewert von `foo`.

Eine einmal definierte Methode kann mit `undef` wieder entfernt werden. Dafür besteht aber nur selten Bedarf, etwa in Vererbungsszenarien, denn man kann in Ruby eine Methode durch eine andere ersetzen, indem man einfach eine weitere Methode im selben Kontext mit demselben Namen definiert. Methodenüberladung, bei der eine Methode gleichen Namens im selben Kontext mit unterschiedlichen Bedeutungen existieren kann, wird von Ruby nicht unterstützt. Die jüngere Methode verdrängt die ältere.

```
# Erstdefinition:
def foo
  "foo"
end

# Vollständige Ersetzung (nicht: Überladung):
def foo(arg)
  puts(arg)
```

```
end

# Komplette Entfernung:
undef foo
```

Methoden sind in Ruby stets an ein bestimmtes Objekt gebunden. Der Aufruf einer Methode erfolgt in Ruby immer nach dem Schema »empfänger.methodenname«, wobei »empfänger.« samt Punkt entfallen darf, wenn der Empfänger mit dem impliziten `self` identisch ist (→ 6.3).

8.2 Methoden auf der obersten Ebene

Es ist möglich, auf der obersten Programmebene Methoden zu definieren.

```
def christmas
puts "Ho, ho, ho!"
end

christmas
```

Dieses Beispiel ist ein vollständiges Ruby-Programm. Seine Ausführung ergibt:

```
Ho, ho, ho!
```

Dies mag nach den Ausführungen zu objektorientierter Programmierung (→ 6) erstaunen, denn scheinbar wird hier eine Methode fernab einer Klasse definiert. Der erste Eindruck täuscht jedoch. In Ruby werden auf der obersten Programmebene definierte Methoden (private, dazu **[TODO: Querverweis]**) Instanzmethoden der Klasse `Object`. Das hat zwei Konsequenzen. Erstens ist `Object` die implizite Elternklasse aller anderen Klassen (Ausnahmen **[TODO: Querverweis]**; zu Vererbung **[TODO: Querverweis]**), sodaß dort definierte Methoden allen Objekten zur Verfügung stehen. Zweitens ist die oberste Programmebene der Ausführungskontext des main-Objekts. Dieses Objekt ist eine Instanz der Klasse Object. Da Methodenaufrufe ohne ausdrücklichen Empfänger an `self` geschickt werden (dazu oben schon → 6.3) und `self` auf der obersten Programmebene das main-Objekt ist, kommt in Beispielprogramm die Methode `christmas` wie erwartet zum Zuge.

Diese etwas komplizierte Logik hinter einem scheinbar so einfachen Konstrukt ist für Anfänger nicht immer leicht zu begreifen, ist aber ein schönes

Beispiel dafür, wie Ruby versucht, seine Prinzipien konsequent durchzuhalten und nicht für alles mögliche Ausnahmen von den allgemeinen Regeln zu machen. Alles in Ruby ist ein Objekt, und das gilt auch von der obersten Programmebene. Matz selbst hat gesagt, daß Ruby ganz wie der menschliche Körper außen einfach, aber innen außerordentlich komplex ist [**TODO: Nachweis**].

8.3 Methodenarten

8.3.1 Instanzmethoden

Der Regelfall der Methode ist die Instanzmethode. Als Teil einer Klassendefinition steht sie in Umsetzung des »Bauplans« den einzelnen Instanzen der Klasse zur Verfügung. Alle zwischen `class` und dem dazugehörigen `end` definierten Methoden werden, wenn keine besonderen Vorkehrungen getroffen werden, Instanzmethoden der betreffenden Klasse.

```
class Wolf
  def howl
    puts "Auuuuuuuuuu!"
  end
end
Wolf.new.howl #=> Auuuuuuuuuu!
```

8.3.2 Singleton-Methoden

Es ist möglich, einzelnen Objekten mehr Methoden an die Hand zu geben, als anderen. So ist es denkbar, daß zwei Instanzen derselben Klasse unterschiedliche Methoden erhalten können. Man kann sich vorstellen, daß der exzessive Einsatz dieses Mittels zu erheblicher Verwirrung führen würde, da scheinbar gleiches plötzlich anders ist. Bei Anwendung von Singleton-Methoden ist deshalb äußerste Vorsicht geboten. Man kann sich leicht schwer erkennbare Fehler in den Code einbauen.

Singleton-Methoden sind Instanzmethoden der *Singleton-* oder *Eigenklasse* eines Objekts. Jedes Objekt besitzt neben seiner Hauptklasse (z. B. Wolf) noch eine solche Eigenklasse, dessen einzige Instanz es ist — daher der Begriff „Singleton“. Die Eigenklasse eines Objekts kann mit der etwas sperrigen Syntax `class << objekt` geöffnet werden; die Definition ist wie üblich mit

end zu beschließen. Innerhalb dieser Definition können wie bei einer normalen Klasse Methoden definiert werden, die dann aber nur diesem einen Objekt zur Verfügung stehen.

```
class Wolf
  def howl
    puts "Auuuuuuuuuu!"
  end
end

white_fang = Wolf.new # kinda
class << white_fang
  def bite
    puts "Haps!"
  end
end
white_fang.howl #=> Auuuuuuuuuu!
white_fang.bite #=> Haps!
```

Speziell für Methodendefinitionen für nur ein Objekt steht auch eine kürzere Syntax zur Verfügung:

```
def white_fang.bite
  puts "Haps!"
end
```

In der Eigenklasse eines Objekts definierte Methoden gehen den normalen Instanzmethoden vor. Würde im obigen Beispiel die Eigenklasse von white_fang also eine eigene Version von howl definieren, würde diese anstelle der Definition in der Klasse Wolf ausgeführt.

Eigenklassen können verschachtelt werden. Ein praktischer Nutzen ist daraus aber nicht zu entnehmen, weshalb auf eine diesbezügliche Er- läuterung verzichtet wird.

8.3.3 Klassenmethoden

Die vorangegangenen Beispiele verraten, daß dies noch nicht die ganze Wahrheit ist. Denn was ist eigentlich Wolf.new? Auch hier hält Ruby an seinen Prinzipien fest. Wo Sprachen wie C++ ein eigenes Schlüsselwort benötigen,

kommt in Ruby die gewöhnliche Methodensystematik zum Einsatz. Nach den allgemeinen Regeln bedeutet der Punkt einen Methodenaufruf; ihm folgt der Name der Methode und ihm geht der Empfänger des Aufrufs voraus. Der Empfänger des Aufrufs der Methode `new` ist deshalb `Wolf`. Weil es sich dabei um eine Klasse handelt, ist der zwingende Schluß, daß Klassen Objekte sind. Quod erat demonstrandum.

In Ruby sind Klassen Instanzen der Klasse `Class`. Diese Klasse definiert dementsprechend auch eine Instanzmethode `new`, die dann den einzelnen Klassen zur Verfügung steht. In aller Regel will der Programmierer aber nicht allen Klassen gleichzeitig neue Methoden zur Verfügung stellen, wie es durch Änderung von `Class` aber der Fall wäre. Unter Anwendung der Erkenntnisse aus dem vorigen Abschnitt (→ 8.3.2) liegt die Lösung des Problems auf der Hand. Die Methoden sind als Singleton-Methoden auf dem Klassen-Objekt zu definieren. Man nennt sie dann *Klassenmethoden* und die für Singleton-Methoden geltende Syntax läßt sich entsprechend anwenden. Dazu muß man lediglich wissen, daß innerhalb einer Klassendefinition `self` auf die Klasse (das Klassenobjekt) selbst zeigt. So ergibt sich:

```
class Wolf
  class << self
    def scientific_name
      "canis lupus"
    end
  end
end
puts Wolf.scientific_name #=> canis lupus
```

Natürlich ist auch die Kurzsyntax zulässig und in aller Regel die richtige Wahl, weil kompakter.

```
class Wolf
  def self.scientific_name
    "canis lupus"
  end
end
puts Wolf.scientific_name #=> canis lupus
```

Klassenmethoden sind der einzige Fall, in dem Singleton-Methoden bedenkenlos eingesetzt werden können und sollten. Dennoch sollte man sich vor

der Definition einer Klassenmethode stets fragen, ob nicht eine Instanzmethode sinnvoller wäre. Als grobe Richtschnur ist eine Klassen- anstelle einer Instanzmethode dann angebracht, wenn sich alle Instanzen einer Klasse eine bestimmte Information teilen sollen und eine Veränderung dieser Information alle Instanzen gleichermaßen betreffen soll. Weiterhin ist eine Klassenmethode dann sinnvoll, wenn es um Informationen geht, die auch ohne Instanz verfügbar sein sollen, wie etwa im Beispiel der wissenschaftliche Name des Wolfes.

8.4 Alias

In Ruby dient das Schlüsselwort `alias` dazu, einer Methode einen neuen Namen zu geben. Dabei wird die Methode vollständig *kopiert*. Wird nach der Neubenennung die ursprüngliche Methode geändert, bleibt der Alias unberührt. Diese Technik kommt gelegentlich zum Einsatz, wenn man vor der dringenden Notwendigkeit steht, ein Problem in einer fremden Programmbibliothek schnellstmöglich beheben zu müssen und keine Zeit bleibt, auf einen Bugfix von Upstream zu warten.

```
class Foo
  def foo
    puts "foo"
  end
  alias bar foo
end

f = Foo.new
f.bar #=> foo
```

`alias_method` funktioniert genauso wie das Schlüsselwort `alias`, ist aber selbst kein Schlüsselwort, sondern eine Methode.

8.5 Parameter

Was Methoden tun, wird durch den Methodenkörper festgelegt. Das kann immer dasselbe sein, muß es aber nicht. Zwar sollte eine Methode niemals ganz unterschiedliche Dinge tun — Faustregel: eine Aufgabe pro Methode—, aber

es kann doch erforderlich sein, abhängig von äußeren Bedingungen im Einzelfall anders als üblich zu reagieren. Eine wichtige Möglichkeit zur Einführung äußerer Informationen in eine Methode sind *Parameter*. Sie werden in der Methodendefinition festgelegt und verhalten sich innerhalb der Methode wie lokale Variablen. Den vom Aufrufer im Einzelfall übergebenen Wert nennt man dann *Argument*. Ruby bietet viele verschiedene Möglichkeiten, Me- thoden mit Parametern zu versehen.

Allen Parameterarten ist gemeinsam, daß sie als Teil der Methoden-Signatur hinter dem Namen der Methode in Klammern aufgeführt werden. Beispiele hierfür finden sich in den nachfolgendn Abschnitten in ausreichender Zahl.

8.5.1 Einfache Parameter

Der wichtigste Fall sind die *einfachen* oder *ortsabhängigen* Parameter. Der Aufrufer muß die Argumente genau in der Reihenfolge übergeben, in der die Methodendefinition die Parameter angegeben hat. Das Weglassen einzelner Argumente ist in dieser Grundform nicht erlaubt, d. h. alle Parameter sind erforderlich.

```
def bark(count)
  count.times do
    puts "Wau!"
  end
end
```

Mehrere Parameter sind durch Kommata zu trennen.

```
def sum(a, b)
  a + b
end
puts sum(5, 7) #=> 12
```

8.5.2 Optionale Parameter

Parameter der vorgenannten Art (→ 8.5.1) können per Gleichheitszeichen mit Standardwerten versehen werden. Für diese Parameter muß der Aufrufer dann keine Argumente mehr übergeben. Tut er es doch, wird der übergebene Wert statt des Standardwerte benutzt.

```
def bark(count = 3)
  count.times { puts("Wau!") }
end
bark     #=> Wau! Wau! Wau!
bark(1)  #=> Wau!
```

Optionale und erforderliche Parameter können miteinander kombiniert werden. Es wird empfohlen, die erforderlichen den optionalen Parametern in der Methodendefinition voranzustellen, zwingend ist dies seit Ruby 2 aber nicht mehr.

```
def accelerate(delta, direction = :left)
  # ...
end
accelerate(50)
accelerate(32, :right)
```

8.5.3 Restparameter

Es ist zulässig, dem letzten Parameter in der Parameterliste ein Sternchen * voranzustellen. Dieser Parameter fängt dann alle Argumente auf, die sonst keinem Parameter entsprechen. So kann man Methoden schreiben, die eine variable Anzahl von Argumenten akzeptieren (*variadic parameters*).

```
def foo(req, *args)
  puts "Mandatory one: #{req}"
  args.each { |el| puts("Additional arg: #{el}") }
end
foo(3)
foo(3, 9)
foo("a", 56, "fff")
```

Ausgabe:

```
Mandatory one: 3
Mandatory one: 9
Additional arg: 9
Mandatory one: a
```

```
Additional arg: 56
Additional arg: fff
```

8.5.4 Schlüsselwort-Parameter

Die bisher vorgestellten Möglichkeiten der Parameterdefinition versagen, wenn eine Methode eine große Anzahl von Parametern erhalten soll. Das ist unschädlich, wenn es sich um eine große Zahl erforderlicher, ortsabhängiger Parameter handelt, denn eine solche Konstruktion ist schlechter Stil, die ohnehin vermieden werden sollte. Richtigerweise ist in diesen Fällen die Methode in mehrere Methoden aufzuspalten. Anders liegt der Fall, wenn es um viele optionale Parameter geht. Insbesondere bei stark abstrahierten Methoden, die unter einer einfachen API einen komplexen Prozeß abbilden, kann es erforderlich sein, dem Programmierer die Möglichkeit zu geben, auf verschiedene Aspekte dieses Prozesses Einfluß zu nehmen. Die daraus resultierende hohe Anzahl optionaler, also mit einem Standardwert versehener, Parameter ist aus Ergonomiegesichtspunkten unbefriedigend, denn es kann schnell zu Verwirrungen bei der Reihenfolge der Argument-Übergabe kommen. Standig bei der Programmierung die Dokumentation heranziehen zu müssen, um die rechte Reihenfolge festzustellen, ist ebenfalls störend, zumal sie sich womöglich in späteren Versionen der betreffenden Methode ändern könnte. Die Lösung für das Problem sind *Schlüsselwortparameter*. Bei ihnen erfolgt die Zuordnung von Argument zu Parameter nicht anhand der Position beim Methodenaufruf, sondern anhand des Parameternamens. Seit Version 2 bietet Ruby Unterstützung für dieses Konzept an, das mit Version 2.2 um die Möglichkeit erforderlicher Schlüsselwort-Parameter erweitert wurde. Syntaktisch erfolgt die Definition derartiger Schlüsselwort-Parameter durch einen dem Parameterbezeichner in der Methodendefinition nachgestellten Doppelpunkt, optional gefolgt vom Standardwert für diesen Parameter. Fehlt diese Angabe, handelt es sich um einen erforderlichen Parameter. Der Aufruf erfolgt in analoger Syntax.

```
def download(url:, timeout: 5, verify: false)
  # ...
end

download(url: "https://example.com/file.txt",
         verify: true)
```

Innerhalb des Methodenkörpers verhalten diese Parameter sich nicht anders als »gewöhnliche« Parameter auch, d. h. wie lokale Variablen.

Eine Kombination mit positionsabhangigen Parametern ist möglich. Zu kreativ sollte man damit aber nicht umgehen; oberstes Ziel muß die Lesbarkeit des Quellcodes sein.

```
def download(url, timeout: 5, verify: true)
  # ...
end

download("https://example.com/file.txt",
         verify: true)
```

Vor Einführung von Schlüsselwort-Parametern war es in Ruby üblich, denselben Effekt durch Einsatz eines positionsabhängigen Parameters, der ein Hash als Argument bekam, zu erreichen. Diese Methode trifft man auch heute noch oft an, insbesondere bei der Arbeit mit älterem Code. Das sieht meist so oder ähnlich aus:

```
def download(hsh)
  raise "Argument not a hash" unless hsh.kind_of?(Hash)
  hsh[:timeout] ||= 5
  hsh[:verify] ||= 5
  hsh[:url] || raise(ArgumentError, ":url missing")
  # ...
end

download(url: "https://example.com/file.txt",
         verify: true)
```

Der Aufruf unterscheidet sich nicht von demjenigen einer Methode mit »echten« Schlüsselwort-Parametern. Dies deshalb, weil Ruby im obigen Fall implizit geschweifte Klammern um die Argumentliste annimt, d. h. sie in ein Hash transformiert.

Effektiv muß bei dieser Variante der Programmierer das gesamte Argument-Parsing selbst übernehmen. Das ist unpraktisch, zeitraubend und fehlerträchtig, weshalb man in neuerem Code stets auf Rubys native Schlüsselwort-Parameter zurückgreifen sollte. Ein anderes gilt auch nicht für Fälle, in denen man die Schlüssel nicht von vornherein fix festlegen will. Ruby bietet

für diesen Fall eine Doppelstern-Syntax ** an, die alle genutzten Schlüsselwort-Argumente in einem Hash auffängt.

```
def save_config(**keys)
  File.open("myprog.conf", "w") do |f|
    keys.each do |k, v|
      f.puts "#{k}: #{v}"
    end
  end
end
```

8.5.5 Durchreichen und Ignorieren von Argumenten

Es kommt vor, daß man die erhaltenen Argumente samt und sonders an eine andere Methode weiterleiten oder sogar komplett ignorieren will. Für beides bietet Ruby elegante Möglichkeiten, die ohne selbstgeschriebenen Code auskommen.

Zunächst zum Ignorieren von Argumenten. Um alle übergebenen Argumente zu ignorieren, genügt ein Sternchen * als Parameterliste. Die Methode akzeptiert dann beliebig viele Argumente und ignoriert sie.

```
def ignore(*)
end
ignore(1)
ignore("a", "foo")
```

Positionsabhängige Parameter mit und ohne Standardwert kann man mithilfe des Restparameters (→ 8.5.3) durchleiten.

```
def forward(*args)
  other(*args)
end
```

Dabei wird das Sternchen beim Aufruf der Zielmethode wiederholt. Man bezeichnet es in diesem Kontext als *Splat-Operator*. Tatsächlich kann dieser auf jedes Array angewandt werden. So kann man etwa Argumentlisten aus Nutzereingaben zusammenbauen.

```
ary = ["a", 5]
mymethod(*ary)
```

Spiegelbildlich wird bei Schlüsselwort-Parametern verfahren. Es kommt hier der *Double-Splat-Operator* ** zum Einsatz.

```
def forward(**hsh)
  other(**hsh)
end
```

Er kann auf jedes Hash angewandt werden.

8.6 Blöcke

Neben regulären Parametern können in Ruby Methoden auch Blöcke akzeptieren. Diese stellen sich aus der Perspektive des Aufrufers dann als Code zwischen den Schlüsselwörtern `do` und `end` bzw. den geschweiften Klammern {} dar. Die bisher in diesem Kapitel vorgestellten Techniken sind nicht ausreichend, um selbst Methoden dieser Art zu schreiben. Dies geschieht mithilfe der nunmehr vorzustellenden Sprachelemente.

Zunächst ist aber vorauszuschicken, daß in Ruby jede Methode einen Block akzeptiert. Ohne weitere Vorkehrungen wird dieser Block jedoch ignoriert. So kann man `puts` zwar einen Block übergeben, erzielt damit aber keinen sinnvollen Effekt.

```
puts("Test"){puts "Block"} #=> Test
```

Der Codeblock wird nicht ausgeführt.

8.6.1 yield

Die wichtigste Variante zur Definition von Methoden mit Blöcken ist der Einsatz des Schlüsselwortes `yield`. `yield` gibt die Kontrolle an den der Methode übergebenen Block ab, d. h. führt ihn aus. Beispiel:

```
def foo
  puts "Vor dem Block"
  yield
  puts "Nach dem Block"
```

```
end

foo { puts "Im Block" }
#=> Vor dem Block
#=> Im Block
#=> Nach dem Block
```

`yield` kann mehrfach benutzt werden und führt den Block dann entsprechend noch einmal aus. Anwendungsfälle dafür sind selten. Haufiger ist es dagegen notwendig, dem Block Informationen aus der Methode über Blockparameter zur Verfügung zu stellen. Dies geschieht durch Übergabe von Argumenten an `yield`. In umgekehrter Richtung ist der »Rückgabewert« von `yield` derjenige des Blocks.

Früher (→ 5.3) wurde bereits die Methode Array#each vorgestellt. Es wäre ohne weiteres möglich, diese Methode unter Rückgriff auf Rubys Schleifenkonstrukte (→ 5.2) und `yield` selbst zu implementieren. Sie sähe dann in etwa so aus:

```
class Array
  def each
    index = 0
    while i < index
      yield self[i]
      i += 1
    end
  end
end
```

`yield` wird in jedem Durchlauf der Schleife einmal aufgerufen und gibt die Kontrolle an den übergebenen Codeblock ab; ferner übergibt die Methode jeweils das Element am index `i` als Argument, welches dann als Blockparameter für den Aufrufer verfügbar wird. Blockparameter funktionieren also semantisch genau wie Methodenparemeter, nur eben »umgekehrt« in dem Sinne, daß es die Methode ist, die Informationen in Richtung des Aufrufers schickt statt anders herum. Anders als bei normalen Methodenparametern behandelt Ruby Blockparameter im Hinblick auf ihre Erforderlichkeit subtil anders. So akzeptiert die oben vorgestellte Implementation von `each` es, wenn man beim Aufruf mit do...end oder geschweiften Klammern das eigentlich an `yield` übergebene Blockargument nicht abholt. Auf die Einzelheiten wird weiter unten eingegangen ([**TODO: Querverweis**]).

Ein weiteres, komplexeres Beispiel. Arrays (genauer: Klassen, die `Enumerable` einbinden, dazu **[TODO: Querverweis]**) unterstützen eine Instanzmethode `reduce`, die es erlaubt, anhand einer vom Aufrufer per Codeblock mitgeteilten Regel eine Liste auf einen einzelnen Wert zu reduzieren. Dazu erhält der Codeblock für jedes Element im Array einen Memo-Parameter und den aktuell iterierten Wert. Der Memo-Parameter wird auf den Rückgabewert des letzten Aufrufs des Codeblocks (= »Rückgabewert« von `yield` aus Sicht der Methode) gesetzt. Eine Implementation könnte damit wie folgt aussehen:

```
class Array
  def reduce(start)
    memo = start
    each do |el|
      memo = yield(memo, el)
    end
    memo # Implizites return beachten
  end
end
```

Es ist schon erstaunlich, mit wie wenigen Zeilen Code sich eine scheinbar komplexe Methode wie `reduce` in Ruby bei richtiger Anwendung der zur Verfügung stehenden Techniken realisieren laßt. Bemerkenswert an diesem Beispiel ist über das bereits Gesagte hinaus, daß `memo` in derselben Zeile genutzt und neu zugewiesen wird. Dies ist unschädlich, da der Wert von `memo` ausgelesen und an `yield` übergeben wird, bevor `memo` nach Abschluß von `yield` auf dessen Ergebnis gesetzt wird. Dieses `reduce` verhält sich genau wie sein bestehender Zwilling.

```
[1, 2, 3].reduce(0){|sum, el| sum + el} # => 6
```

8.6.2 enum_for und block_given?

Die im vorigen Abschnitt (→ 8.6.1) vorgestellte Implementatin von Array#each unterstützt nicht den von Ruby in neueren Versionen eingeführten Aufruf als Enumerator (→ 5.3.4) durch Weglassen des Blockarguments. Vielmehr verursacht ein Weglassen des Blocks einen LocalJumpError, sobald Ruby auf `yield` trifft. Das läßt sich nur verhindern, indem man den Kontrollfluß durch Einsatz einer Bedingung (→ 5.1) daran hindert, bei Nichtvorliegen einen Blocks

auf `yield` zu treffen. Hierzu dient die Methode `block_given?`, welche nur dann einen wahrenWert zurückliefert, wenn der Aufrufer einen Block übergeben hat. Dies kann man zusammen mit der Methode `enum_for` benutzen, um Rubys Verhalten bei den meisten Iteratormethoden für den Fall, daß kein Block übergeben wird, nachzuahmen. Ergänzt um die Erzeugung eines Enumerators bei Nichtübergabe eines Blocks sieht das Beispiel dann so aus:

```
class Array
  def reduce(start)
    return enum_for(__method__) unless block_given?
    memo = start
    each do |el|
      memo = yield(memo, el)
    end
    memo # Implizites return beachten
  end
end
```

`enum_for` erzeugt den externen Iterator, der aufgrund des `return`-Ausdrucks unter vorzeitigem Abbruch der Methode sofort zurückgegeben wird, wenn kein Block übergeben wurde (`block_given?`). Das Schlüsselwort `__method__` evaluiert zum Namen der aktuell ausgeführten Methode (vorliegend demnach zum Symbol :each, das man hier auch direkt hätte einsetzen können). Bei Benutzung des so erzeugten Enumerators fingiert Ruby den Block, sodaß dann `block_given?` einen wahren Wert zurückgibt.

```
en = [1, 2, 3].each
en.next #=> 1
```

8.6.3 Blockargument

Es gibt neben `yield` noch eine andere Möglichkeit, Blöcke anzunehmen: das ausdrückliche Blockargument mithilfe des kommerziellen Und-Zeichens &. Seine Anwendung funktioniert ähnlich wie das Restargument.

```
def foo(&block)
  puts "Vor dem Block"
  block.call
```

```
  puts "Nach dem Block"
end
```

Dieses Beispiel entspricht dem bereits zu `yield` gegebenen, allerdings unter ausdrücklicher Anwendung eines Blockarguments. Das &-Zeiehen in der Methodensignatur ist dabei rein syntaktischer Natur; `block` verhält sich wie eine normale lokale Variable und hält den übergebenen Block als Objekt. Es handelt sich dabei um eine Instanz der Klasse `Proc`. Mithilfe von `call` wird der Block dann ausgeführt. Ähnlich wie bei `yield` tauchen auch `call` übergebene Argumente im Block als Blockparameter auf und ist der Bückgabewert des Blocks der von `call`-

Aus diesem Beispiel wird der Wert des &-Konstrukts nicht kenntlich. Seine Vorteile spielt er aus, wenn man sich die »Objektifizierung« des Blocks zunutze macht und ihn nicht etwa ausführt, sondern auspeichert. So wird eine verzögerte Blockausführung möglich. Im Vorgriff auf Hook-Methoden [**TODO: Querverweis**] ein Beispiel anhand von `initialize`, das letztlich den Inhalt der Klassenmethode `new` bestimmt:

```
class Foo
  def initialize(&block)
    @block = block
  end
  def execute
    @block.call
  end
end

f = Foo.new{puts("Hi!")}
f.execute #=> Hi!
```

Hier wird der Block nicht sofort bei Aufruf von new ausgeführt, sondern in der Instanzvariablen `@block` [**TODO: Querverweis**] abgespeichert„ Später wird dann in der Methode execute auf dieses Block-Objekt zurückgegriffen und es ausgeführt.

Wird das Blockargument mit dem Restargument kombiniert, folgt es nach dem Restargument.

```
def foo(*args, &block)
  # ...
```

```
end
```

8.7 Anonyme Funktionen und Closures

Normalerweise tragen Funktionen in Ruby einen Namen, unter dem sie aufgerufen werden können (Methoden). Typischerweise gibt man diesen Namen als Symbol an. Es ist jedoch auch möglich, anonyme Funktionen zu definieren. Dies geschieht mithilfe der Methoden `lambda`, `proc` und `Proc.new`. Effektiv erstellen diese Methoden Blöcke; es handelt sich um Instanzen der Klasse `Proc` und damit um dieselbe Art von Objekt, die auch bei der Benutzung des Blockargements entsteht. Tatsächlich verhält sich ein durch Blockargument erstelltes Objekt genau so wie eines, das mit `proc` erstellt wurde. Auch zwischen `Proc.new` und `proc` gibt es keine Unterschiede **[TODO: Prüfen]**.

Alle Blöcke in Ruby — egal, wie sie erstellt wurden — sind *Closures*. Eine Closure ist dadurch gekennzeichnet, daß sie den Ausführungskontext, der bei ihrer Erstellung aktiv war, einschließt — daher der Name. Das ist der Grund, weshalb ein Block, der zunächst nicht ausgeführt, sondern abgespeichert wird, dennoch vollen Zugriff auf alle Variablen hat, die zum Zeitpunkt seiner Erstellung erreichbar waren. Möglich ist deshalb dieses:

```
class Foo
  # ...wie oben...
end

def foo
  x = 3
  Foo.new{puts(x)}
end

f = foo
f.execute #=> 3
```

Man sollte meinen, daß die Variable `x` mit dem Ende der Methode `foo` verfällt und der Code einen Laufzeitfehler auslöst. Das ist aber nicht der Fall, denn aufgrund der Closure-Eigenschaft des an `Foo.new` übergebenen Blocks hält Ruby die Variable weiter vor; sie steht aber natürlich wirklich nur dem Block zur Verfügung. Dieses Verhalten kann man sich in der Metaprogrammierung zunutze machen.

`lambda` und `proc` erstellen Block-Objekte; die Methoden bieten gewissermaßen die nötige Brücke dafür, da Ruby lange kein syntaktisches Konstrukt für die direkte Erstellung solcher Block-Objekte besaß. Man übergibt diesen Methoden einen Block und sie geben ihn ohne weitere Ausführung als Instanz der Klasse `Proc` zurück. Diese läßt sich wie jedes andere Objekt auch in einer Variablen speichern.

```
proc1 = proc   {|x| p(x)}
proc2 = lambda {|x| p(x)}
```

Die Ausführung des Procs erfolgt über eine von drei Varianten, die alle gleichwertig sind; übergebene Argumente tauchen als Blockargumente auf. Bereits vorgestellt wurde die Methode `call`. Daneben kann man eckige Klammern wie bei der Array- oder Hash-Indexierung benutzen oder die verunglückte Punktklammersyntax:

```
proc1.call(5) #=> 5
proc2.call(5) #=> 5
proc1[5]      #=> 5
proc2[5]      #=> 5
proc1.(5)     #=> 5
proc2.(5)     #=> 5
```

Bei der Behandlung der Argumente zeigt sich der Unterschied zwischen `proc` und `lambda`. Während ersteres Unterschiede zwischen der übergebenen und der Anzahl der erforderlichen Argumente ignoriert, verlangt letzteres ganz wie eine richtige Methode eine genaue Übereinstimmung, d. h. wo `proc` überschüssige Argumente verwirft und fehlende Argumente als nil annimmt, erzeugt `lambda` einen ArgumentError. Aufgrund dieser Unterschiede hat es sich eingebürgert, `lambda` dann zu benutzen, wenn man den Aspekt als anonyme Funktion in den Vordergrund stellen möchte und `proc` dann, wenn es mehr um die Eigenschaft als »objektifizierter« Block geht.

```
proc1.call      #=> nil
proc2.call      #=> ArgumentError
proc1.call 5, 6 #=> 5
proc2.call 5, 6 #=> ArgumentError
```

Zwischen den mit `lambda` und `proc` erstellten Procs gibt es einen weiteren Unterschied, der mit der Behandlung des Schlüsselworts `return` zusammenhängt. Ein mit `lambda` erstellter Proc verhält sich so, wie man es von einer (anonymen) Methode erwarten würde: der Block wird beendet mit dem `return` übergebenen Argument als Rückgabewert. Bei mit `proc` erstellten Procs führt `return` zu einem Laufzeitfehler[1].

```
proc1 = lambda{ return }
proc2 = proc{ return }

def foo(f)
  f.call
  puts "In foo!"
end

foo(proc1) #=> In foo!
foo(proc2) #=> LocalJumpError
```

Seit Ruby 1.9 existiert mit dem Lambda-Operator `->` auch syntaktisch eine Möglichkeit, Instanzen von `Proc` zu erhalten. Dieser Operator soll angeblich wie ein griechisches Lambda-Zeichen λ als ASCII-Art aussehen. Dem Autor gelingt es seit Jahren nicht, ein solches in dieser Strichkonstruktion zu erkennen. Seine Form, die wohl eher einem nach rechts zeigenden Pfeil ähnelt, hat diesem Operator die Spitznamen »Picky Lambda« und »Stabby Lambda« eingebracht. Seine Anwendung entspricht dem Aufruf von `lambda`. Das obige Beispiel 2 noch einmal als Picky Lambda:

```
block2 = -> x {p(x)}
```

Wohl die Mehrheit der Rubyisten hält diesen Operator für den Türöffner Rubys zur funktionalen Programmierung. Nach hiesiger Ansicht verletzt der Operator Rubys Design-Ziel des leicht menschenlesbaren Quelltexts und sollte aus der Sprache gestrichen werden. Der Einsatz der Methode `lambda` ist vorzugswürdig. Wer eine Kurzsyntax will, kann sich eine solche leicht unter Verwendung eines richtigen Lambda-Zeichens schaffen:

[1] Das war in früheren Versionen von Ruby anders. Dort wurde die umgebende Methode beendet — eine schnell übersehene Fehlerquelle.

```
module Kernel
  alias  lambda
end

block2 = {|x| p(x)}
```

8.8 Methodenauflösung und super

8.8.1 Die Auflösung von Methodenaufrufen

[TODO: Schreiben, insb. mit Vererbung und Mixins]

8.8.2 super

Das Schlüsselwort `super` stellt eine besondere Form des Methodenaufrufs dar. Es kann nur innerhalb einer Methode verwandt werden und ruft dort, wo es benutzt wird, die gleichnamige Methode im nachfolgenden Element der Vorfahrenskette auf. Anders ausgedrückt, stößt `super` die Auflösung des Methodenaufrufs neu an und klammert dabei alle Elemente in der Vorfahrenskette bis einschließlich dem Aufrufer aus. `super` darf an beliebiger Stelle beliebig oft im Methodenkörper auftreten.

Das wichtigste Einsatzgebiet für `super` ist die Verfeinerung von Methoden bei Vererbung. So ist es naheliegend, daß die Methode Labrador#swim auf der allgemeineren Methode Dog#swim aufbaut. Technisch erfolgt die Umsetzung dann mithilfe von `super`:

```
class Dog
  def swim
    puts "Basic swimming"
  end
end
class Labrador < Dog
  def swim
    super
    puts "Better swimming"
  end
end
Labrador.new.swim
```

```
#=> Basic swimming
#=> Better swimming
```

super ist aber unabhängig vom Einsatz von Vererbung; es funktioniert mit Mixins (→ 9.2) genauso gut, denn es operiert nicht auf der Elternklasse, sondern wie beschrieben auf der Vorfahrenskette. Dazu ein komplexeres Beispiel:

```
module M
  def foo
    puts "M#foo"
  end
end
class A
  include M
  def foo
    puts "A#foo"
    super
  end
end
class B < A
  def foo
    puts "B#foo"
    super
  end
end

p B.ancestors
B.new.foo
```

Ausgangspunkt ist der Aufruf von B#foo. Diese Methode wird bei der Auflösung des Methodenaufrufs zuerst gefunden, da B sich am Anfang der Vorfahrenskette befindet. super stößt dann die Suche wieder an; gefunden wird dann A#foo. Da auch dieses super aufruft, wird erneut weitergesucht. Nach A folgt M in der Vorfahrenskette aufgrund des Aufrufs von include. Die Ausgabe dieses Programms lautet entsprechend:

```
[B, A, M, Object, Kernel, BasicObject]
B#foo
A#foo
M#foo
```

Bei der Vermischung von Vererbung, Mixins und `super` sollte man einen guten Überblick über die Vorfahrenskette behalten und penibel darauf achten, an welcher Stelle in einer Methode man `super` aufruft. Wenn man die Vorfahrenskette statt nur mit `include` auch noch mit `prepend` modifiziert, dann kann man ausgesprochen schwer erfaßbare Aufrufungswege erhalten.

`super` akzeptiert Argumente ähnlich wie ein gewöhnlicher Methodenaufruf. Werden ausdrücklich Argumente übergeben, so werden sie benutzt, um die Parameter der nächsthöheren Methode zu füllen. Ein Unterschied ergibt sich aber bei Weglassen von Argumenten: in diesem Fall werden automatisch alle Parameter der aktuellen Methode als Argumente für die übergeordnete Methode benutzt. Will man das verhindern, muß man ausdrücklich eine leere Argumentliste übergeben. Beispiel:

```
class Foo
  def foo(*args)
    p args
  end
end
class Bar < Foo
  def foo(arg)
    super       # Implizit
    super(arg) # Ausdrücklich
    super()     # Keine Argumente
  end
end
Bar.new.foo(3)
#=> [3]
#=> [3]
#=> []
```

Eine für die übergeordnete Methode unzulässige Argumentenzahl führt wie sonst auch zu einem Laufzeitfehler (ArgumentError). Fehlt eine übergeordnete Methode ganz, dann erzeugt Ruby gleichfalls einen Laufzeitfehler:

```
class Foo
  def problem
    puts "Going to blow up"
    super
  end
```

Name/ Schlüsselwort	Zugriff von... außen	anderer Instanz	bei Vererbung
public	Ja	Ja	Ja
protected	Nein	Ja	Ja
private	Nein	Nein	Ja

Tabelle 8.1: Methoden-Sichtbarkeiten

```
end

Foo.new.problem
```

Ausgabe:

```
Going to blow up
/tmp/x.rb:4:in `problem': super: no superclass method
`problem' for #<Foo:0x0055f32d2b7ca0> (NoMethodError)
from /tmp/example.rb:8:in `<main>'
```

8.9 Methoden-Sichtbarkeit

Aus dem Geheimnisprinzip (→ 6.5) folgt, daß Objekte ihre interne Organisation dem Aufrufer gegenüber nicht bekanntgeben sollten. Methoden dienen in Ruby aber auch dazu, komplexen Code in kleinere, handhabbare und damit wartbare Stücke aufzuteilen. Es wäre verfehlt, aus der Kollision von Geheimnisprinzip und Wartbarkeit den Schluß zu ziehen, daß eines von beiden bei der Arbeit an komplexen Problemen aufzugeben sei und auf diese Weise riesige Methoden produzierte oder zu viele Interna in Form kleiner Methoden preisgäbe. Der Konflikt wird durch den Einsatz verschiedener Methoden-Sichtbarkeiten aufgelöst. Es ist möglich, bestimmte Methoden dem Zugriff von außerhalb des Objekts zu entziehen.

Die in Ruby verfügbaren Methoden-Sichtbarkeiten können Tafel 8.1 entnommen werden. Alle drei sind Schlüsselwörter und ändern bei Einsatz ohne weitere Argumente im aktuellen Gültigkeitsbereich die Methoden-Sichtbarkeit für alle danach definierten Methoden. Fehlt eine Sichtbarkeitsangabe, nimmt Ruby `public` an.

Typischerweise trifft man diese Schlüsselwörter innerhalb einer Klassendefinition an:

```
class Plane
  def travel(destination)
    # ...
  end

  private

  def move_to(runway)
    # ...
  end

  def accelerate(delta)
    # ...
  end
end
```

Während `travel` der impliziten public-Sichtbarkeit unterliegt, sind `move_to` und `accelerate` private Methoden. Sie können — im Gegensatz zu `travel` — nicht ohne Rückgriff auf Tricks der Metaprogramming **[TODO: Querverweis]** aufgerufen werden.

```
a380 = Plane.new
a380.travel(:FRA)    # OK
a380.accelerate(10) # NoMethodError
```

Ein Aufruf dieser Methoden innerhalb von `travel` ist dagen möglich und sinnvoll. Dabei spielt es keine Rolle, ob für den Aufruf das implizite `self` genutzt wird oder `self` ausdrücklich als Empfänger genannt wird. Ausschlaggebendes Kriterium ist allein, ob der aktuelle Gültigkeitsbereich der »Innenansicht« des Empfängers entspricht.

Anwendungsfälle für geschützte Methoden (`protected`) sind selten. Der Unterschied zu privaten Methoden ist, daß ein Aufruf der Methode durch andere Instanzen derselben Klasse (sowie deren Kindklassen) möglich bliebt bleibt, wohingegen die Klasse bei privaten Methoden keine Rolle spielt. `private` ist damit strenger als `protected`. Im Zweifel sollte aufgrund des Geheimnisprinzips der strengeren Form — also privaten Methoden — der Vorzug eingeräumt werden.

[TODO: Anwendungsbeispiel protected]

Seit Version 2.1 besteht die Möglichkeit, den Sichtbarkeits-Umschaltern auch ausdrücklich den Namen der Methode zu übergeben, deren Sichtbarkeit ge-

ändert werden soll. In diesem Fall beschränkt die Sichtbarkeitsänderung sich auf diese eine Methode; nachfolgende Methodendefinitionen unterliegen wieder der aktuell geltenden Sichtbarkeit (sind also im Zweifel öffentlich, s. o.),

```
def foo
  # ...
end
private :foo
```

Seit derselben Version gibt der `def`-Ausdruck den Namen der definierten Methode als Symbol zurück. Zulässig ist deshalb auch folgendes:

```
private def foo
  # ...
end
```

Mit Ruby 2.6 wurde für die Nutzungder der Sichtbarkeitsumschalter in der vorbezeichneten Weise der Rückgabewert von nil auf den Namen der veränderten Methode geändert. Damit werden Konstruktionen möglich, die man bisher eher aus Java als aus Ruby kannte und die nur das Ergebnis von Metaprogrammierung sein können. Wenn `insecure` eine selbst geschriebene Meta-Methode ist, dann kann sie nunmehr wie folgt angewandt werden:

```
insecure private def foo
  # ...
end
```

Es bleibt abzuwarten, ob sich diese Möglichkeit durchsetzen wird.

§ 9 Module

Module dienen der Abgrenzung größerer Code-Teile von anderen. Dazu werden Module im Wesentlichen auf zwei Arten eingesetzt: als Namespaces und als Mixins. Diese beiden hauptsächlichen Nutzungsarten werden nachfolgend zuerst vorgestellt, bevor ein Blick auf andere Varianten geworfen wird.

Syntaktisch wird ein Modul ganz ähnlich wie eine Klasse definiert: auf das Schlüsselwort `module` folgt der Name des Moduls, der mit einem Großbuchstaben beginnen muß. Alles bis zum zugehörigen `end` ist Teil des Moduls. Ähnlich wie Klassen sind Module in Ruby offen, d. h. können nach ihrer erstmaligen Definition später erneut geöffnet und modifiziert werden.

9.1 Namespaces

Wer Code von Dritten bezieht oder mit ihnen teilt, läuft Gefahr, eine Namenskollision auszulösen. So könnte in einer Programmbibliothek (die man in Ruby meist als RubyGem bezieht, dazu [**TODO: Querverweis**]) zur Berechnung von graphischen Primitiven eine Klasse `Plane` (Fläche) definiert sein. Beschäftigt sich das eigene Hauptprojekt mit Flugzeugen, würde die Einbindung dieser Programmbibliothek zur Namenskollision führen. Aufgrund von Rubys offenen Klassen würden beide Klassen `Plane` in eine vereinigt und es käme zu unvorhergesehenen Verhaltensweisen.

Dieses Problem wird durch Einsatz von *Namespaces* gelöst. Dazu definiert man in Ruby einfach die gewünschten Klassen *innerhalb* eines Moduls. Während für Endnutzerprogramme mangels Wiederverwertungsabsicht dies meist nicht notwendig ist, sollte man den Code von Programmbibliotheken generell innerhalb eines Moduls definieren, das den Namen der Programmbibliothek trägt. So wird das Risiko von Namenskonflikten für potentielle Nutzer der Programmbibliothek minimiert. Im Beispiel von gerade sollte, wenn die Programmbibliothek mygraphics heißt, das entsprechende Namespace-Modul `MyGraphics` heißen.

```
module MyGraphics
```

```
  class Plane
  end
end
```

Derart in einem Modul definierte Klassen erreicht man von außen über den Auflösungs-Operator :: (zwei aufeinanderfolgende Doppelpunkte).

```
circle = MyGraphics::Plane.new
```

Innerhalb des Moduls `MyGraphics` ist dagegen die ausdrückliche Auflösung nicht notwendig, da Ruby anhand des Kontexts den Klassennamen richtig zuordnen kann.

```
module MyGraphics
  class Plane
  end
  class Rectangle
    def initialize
      @plane = Plane.new # meint MyGraphics::Plane
    end
  end
end
```

Es ist Konvention, unterschiedliche Klassen in unterschiedlichen Dateien zu definieren, was aufgrund von Rubys offenen Modulen kein Problem ist. Jede Quelltextdatei der Programmbibliothek mygraphics kann einfach das Modul `MyGraphics` wieder öffnen und eine neue Klasse hinzufügen. Allerdings empfinden manche Programmierer es als störend, wenn dabei jedes Mal die gesamte Klasse nach der Konvention innerhalb des Moduls eingerückt werden muß. Natürlich kann man die Konvention in diesem Fall einfach übergehen und `module` und `class` ohne Einrückung auf dieselbe »Ebene« setzen, aber Ruby bietet eine elegantere Möglichkeit an. Der Auflösungsoperator kann auch direkt hinter dem Schlüsselwort `class` oder `module` eingesetzt werden, um eine Klasse oder ein Modul in ein (anderes) Modul »hineinzudefinieren«. Das setzt allerdings voraus, daß das betreffende Modul bereits vorher definiert worden ist, d. h. eine implizite Moduldefinition findet nicht statt.

```
module MyGraphics
```

```
end

class MyGraphics::Plane
end
```

Wer auf diese Technik zurückgreift, muß wissen, daß Rubys Konstantenauflösung lexikalisch ist. Das bedeutet, daß nur auf die Struktur des Quelltexts geachtet wird und nicht auf dessen inhaltliche Bedeutung. Folgendes ist deshalb nicht möglich:

```
module MyGraphics
  class Plane
  end
end
class MyGraphics::Rectangle
  def initialize
    @plane = Plane.new #=> NameError
  end
end
```

Zeile 7 dieses Codes wird beim Ablauf einen NameError verursachen, weil Ruby `Plane` nicht zuordnen kann. Die Auflösung muß ausdrücklich durch Einsatz des Auflösungsoperators durchgeführt werden.

```
class MyGraphics::Rectangle
  def initialize
    @plane = MyGraphics::Plane.new
  end
end
```

Das ist eine häufige Fehlerquelle. Welche Gültigkeitsbereiche von Ruby bei der impliziten Konstantenauflösung durchsucht werden, kann mithilfe der (Meta-)Methode Module::nesting herausgefunden werden.

```
module MyGraphics
  class Rectangle
    p Module.nesting
```

```
    #=> [MyGraphics::Rectangle, MyGraphics]
  end
end
```

Wird ein Namespace häufig gebraucht, ist das ständige Ausschreiben störend. Es gibt zwei Möglichkeiten, mit diesem Problem umzugehen. Die erste besteht darin, die gewünschte Klasse per Zuweisung aus dem Namespace »herauszuziehen«.

```
Plane = MyGraphics::Plane.new
plane = Plane.new
```

Das ist aber nur bei einzelnen Klassen praktikabel, da es sonst schnell unübersichtlich wird. Werden aus dem Namespace mehrere Klassen benötigt, kann auf `include` zurückgegriffen werden.

```
class Foo
  include MyGraphics

  def foo
    plane = Plane.new
  end
end
```

Im Gegensatz zur Konstantenauflösung arbeitet `include` nicht lexikalisch. Es handelt sich um eine Methode (Module#include) und ihr Einsatz verändert den Empfänger, bei dem es sich meistens um das implizite `self` (→ 6.3) handeln wird.

`include` kann mit einer Ausnahme nur im Kontext von Klassen und Modulen eingesetzt werden (für den Grund → 9.5). Diese Ausnahme ist das main-Objekt. Es definiert eine Singleton-Methode `include`, die sich so verhält, wie man es erwarten würde. Der Inhalt des übergebenen Moduls wird Teil des Main-Objekts. Wichtig: Weil es nur ein einzelnes globales Main-Objekt gibt, wirkt ein `include` auf der obersten Ebene immer global für alle nachfolgend geladenen Dateien. Derartige Inkludierungen sollten deshalb auf die Hauptdatei von Endnutzerprogrammen beschränkt werden und dürfen nicht mit der Funktionalität von `require` ([**TODO: Querverweis**]) verwechselt werden.

9.2 Mixins

Ein *Mixin* ist die Abkapselung einer bestimmten Funktionalität unabhängig von einer Klasse in einem eigenständigen Modul. Das Modul kann dann von Klassen, aber auch von anderen Objekten eingebunden werden, um der betreffenden Klasse bzw. dem betreffenden Objekt die in dem Mixin gespeicherte Funktionalität hinzuzufügen. Technisch geschieht dies durch die Definition von Instanzmethoden in einem Modul, die anders als per Mixin überhaupt nicht aufgerufen werden können. Diese Instanzmethoden von Modulen dürfen nicht mit Modulmethoden verwechselt werden (zu diesen unten → 9.4).

```
module Runnable
  def run_with_log
    # ...
  end
  def run_in_parallel
    # ...
  end
  def noop_run
    # ...
  end
end
```

Das Beispiel zeigt zunächst eine wichtige Konvention für die Benennung von Mixins auf. Ein Mixin kapselt eine bestimmte Funktionalität, die typischerweise später verschiedenen Objekten zugeordnet werden soll. Deshalb wählt man als Namen für ein Mixin ein Adjektiv, das auf »-able« endet. Die beiden wichtigen von Ruby selbst bereitgestellten Mixins `Comparable` und `Enumerable` halten sich selbst an diese Konvention. Sodann definiert das Modul `Runnable` verschiedene Arten, etwas auszuführen, nämlich mit Logging, parallel, etc. Auffällig ist, daß die eigentliche Grundfunktionalität, nämlich was genau wie auszuführen ist, fehlt. Das ist kein Zufall, sondern typisch für Mixins. Sie sind ihrer Natur nach zwar abstrakt von bestimmten Klassen, aber ein Objekt muß oft bestimmte Bedingungen erfüllen, um sinnvoll um das betreffende Mixin erweitert zu werden. Im Beispiel wird das Mixin `runnable` höchstwahrscheinlich eine Methode `run` voraussetzen, die es aufruft (zu Threads, die parallele Code-Ausführung ermöglichen, **[TODO: Querverweis]**):

```
module Runnable
  def run_in_parallel
    Thread.new { run }
  end
end
```

Es wird de facto also in dem Mixin eine Methode aufgerufen, die im konkreten Kontext des Mixins gar nicht definiert ist. Das ist in Ordnung, wenn die Methode beim einbindenden Objekt gefunden werden kann. Es ist sogar so üblich, daß die bereits erwähnten, von Ruby mitgelieferten Mixins `Enumerable` und `Comparable` bestimmte Methoden im einbindenden Objekt voraussetzen (Details weiter unten).

Die Einbindung eines Mixins auf drei Arten erfolgen, die sehr ähnlich sind: per `include`, `extend` und `prepend`. Alle drei Methoden ändern die für die Methodenauflösung wichtige Vorfahrenskette (→ 8.8; parallel hierzu lesen, falls noch nicht geschehen). Das Vorgehen erinnert nicht von ungefähr an das bei Vererbung (→ 6.4). Tatsächlich kann man Mixins benutzen, um die in Ruby eigentlich nicht vorhandende Mehrfachvererbung, bei der von mehreren Klassen gleichzeitig geerbt wird, nachzubauen. Das verfehlt allerdings den Zweck von Mixins, denn wie bereits erwähnt sollte die in einem Mixin implementierte Funktionalität abstrakt von irgendeiner Klasse sein. Ist das nicht der Fall, sollte man zu Vererbung greifen, die Ruby als Einfachvererbung ja durchaus ermöglicht; die Fälle notwendiger Mehrfachvererbung sind selten genug. Die Kontrollfrage muß insoweit lauten, ob man die im Mixin gespeicherte Funktionalität *jedem* Objekt zuordnen können soll, sofern dieses Objekt nur bestimmte Bedingungen erfüllt.

Das Schlüsselwort `super`, das normalerweise im Vererbungskontext Anwendung findet, kann auch mit Mixins benutzt werden. Dazu → 8.8.2.

9.2.1 include

Die mit Abstand wichtigste Art, ein Mixin einzubinden, ist `include`. Es handelt sich um dasselbe `include`, das schon bei Namespaces erwähnt wurde und oft genug kommt es vor, daß ein Modul gleichzeitig als Mixin und als Namespace dient. Das Modul wird an die Vorfahrenskette angehängt, sodaß effektiv der »Instanzteil« des Moduls Teil der einbindenden Klasse wird und die Instanzmethoden des Mixins zu Instanzmethoden der Klasse. Das oben vorgestellte Modul `Runnable` läßt sich daher so einbinden (zu `system` **[TODO: Querverweis]**):

```
class Command
  include Runnable

  def initialize(str)
    @cmd = str
  end

  def run
    system(@cmd)
  end
end

cmd = Command.new("rm -rf /")
cmd.run_in_parallel
```

Obwohl `run_in_parallel` nie für die Klasse `Command` definiert wurde, steht die Methode den Instanzen dieser Klasse durch das Mixin zur Verfügung.

Stellt man fest, daß Teile eines Mixins unpassend sind, kann man die Methode in der einbindenden Klasse einfach übschreiben. In der Klasse definierte Methoden genießen Vorrang, denn `include` hängt an die Vorfahrenskette an. In der Klasse selbst definierte Methoden gehen vor.

```
class Command
  include Runnable
  p ancestors

  def initialize(str)
    @cmd = str
  end

  def run
    system(@cmd)
  end

  def run_with_log
    puts "My run with log!"
  end
end
```

```
cmd = Command.new("rm -rf /")
cmd.run_with_log
```

Ausgabe:

```
[Command, Runnable, Object, Kernel, BasicObject]
My run with Log!
```

9.2.2 prepend

prepend funktioniert ähnlich wie include, hängt das Modul aber nicht an die Vorfahrenskette an, sondern stellt es ihr voran. Die im Mixin definierten Methoden genießen so den Vorrang vor den in der Klasse definierten. Hauptanwendungsfall für diese Art von Programmierung dürften Plugins sein, d. h. ein Hauptprogramm kann vom Endnutzer durch Einbindung eines Plugin-Moduls erweitert und verbessert werden. Das Modul wird durch das Plugin-Interface dann per prepend eingebunden, sodaß die Funktionalität des Hauptprogramms hinter dem Plugin zurücktritt.

Angenommen, ein Hauptprogramm sieht so aus:

```
class App
  PLUGDIR = File.join(ENV["HOME"],
            ".myapp",
            "plugins").freeze
  module Plugins
  end

  def self.all_plugins
    Dir.glob("#{PLUGDIR}/*.rb").sort.each do |file|
      require(file)
    end
    Plugins.constants.map{|c| Plugins.const_get(c)}
  end
  def self.load_plugins
    all_plugins.each{|pl| prepend(pl)}
  end
  def doit
    puts "Done"
```

```
  end
end

App.load_plugins
p App.ancestors
a = App.new
a.doit
```

Die Details, wie `all_plugins` die Module von der Festplatte lädt, brauchen hier nicht zu interessieren. Grob gesagt werden durch Einsatz von Metaprogrammierung ([**TODO: Querverweis**]) alle Dateien im Verzeichnis `~/.myapp/plugins`, die auf »`.rb`« enden, als Ruby-Code ausgeführt und anschließend alle unterhalb von `App::Plugins` definierten Module in einem Array gesammelt. `load_plugins` fügt diese Module dann per `prepend` als Mixin der Hauptklasse `App` hinzu.

Wenn es keine Plugin-Module gibt, ist die Ausgabe des obigen Programms:

```
[App, Object, Kernel, BasicObject]
Done
```

Jedoch könnte man nun ein Plugin als Modul definieren, daß die Methode `doit` überschreibt. Dazu müßte man folgende Quelltextdatei in `~/.myapp/plugins/myplugin.rb` ablegen:

```
module App::Plugins::DoItBetter
  def doit
    puts "Doing it better!"
  end
end
```

Dieses Modul wird als Mixin dann in `load_plugins` per `prepend` eingebunden. Weil es am Anfang der Vorfahrenskette eingeschoben wird, wird die Methode `doit` im Plugin-Mixin bei der Methodenauflösung zuerst gefunden und anstelle der in der Klasse `App` definierten Methode ausgeführt.

```
[App::Plugins::DoItBetter, App, Object, Kernel, BasicObject]
Doing it better!
```

9.2.3 extend

Schließlich ist es möglich, mithilfe von `extend` gezielt ein einzelnes Objekt um den Inhalt eines Mixins zu erweitern. Dies entspricht letztlich einem `include` in der Eigenklasse des Objekts (→ 8.3.2). Anwendungsfälle sind selten und treten wohl vor allem in der Metaprogrammierung auf. Dort instanziiert man dann direkt die Klasse `Object` und erweitert das betreffende sonst nur sehr generische Objekt.

```
obj = Object.new
obj.extend(MyMixin)
p obj.singleton_class.ancestors
# => [#<Class:#<Object:0x0055c702d5cf40>>,
      MyMixin, Object, Kernel, BasicObject]
```

`extend` funktioniert mit allen Objekten. Weil Module natürlich auch Objekte sind, sieht man gelegentlich noch dieses Programmiermuster:

```
module Log
  def log(msg)
    puts "#{Time.now} #{msg}"
  end
  extend self
end
```

Der Aufruf von `extend self` macht hier die Instanzmethode `log` auch als Modulmethode verfügbar, denn `self` zeigt ähnlich wie bei Klassendefinitionen innerhalb der Moduldefinition auf das Modul selbst. Man kann danach sowohl `Log.log` aufrufen als auch erst das Modul `Log` per `include` einbinden und dann die Instanzmethode `log` aufrufen. Dieser Weg ist aber nur sinnvoll, wenn alle Methoden eines Moduls in dieser Weise verfügbar gemacht werden sollen. Ansonsten sollte man stattdessen `module_function` benutzen (→ 9.4).

9.3 Refinements

Rubys offene Klassen erlauben es, jederzeit Änderungen an bestehenden Klassen vorzunehmen. Hauptzweck dieser als *Monkeypatching* (wörtl. »Affenflicken«) bezeichneten Technik ist es, Fehler in Programmbibliotheken zu korrigieren,

für die der Autor der Programmbibliothek noch kein Update bereitgestellt hat. Das Problem dieser Technik ist, daß der Code sehr schnell sehr unübersichtlich wird und man sich nicht mehr sicher sein kann, dass eine Klasse sich wirklich so verhält, wie es von ihrem Haupt-Quelltext aus aussieht, denn ein Monkeypatch wirkt sich ab seiner Anwendung immer auf das gesamte Programm aus — darunter möglicherweise an Stellen, mit denen man nicht gerechnet hat. Diesem Problem entgegnet man mit *Refinements*, die in modernem Ruby-Code Monkeypatches weitgehend ersetzen. Bei Einsatz von Refinements kann man sich wieder sicher sein, daß der Hauptklassencode überall auch so ausgeführt wird, wie man es erwartet; es sei denn, mithilfe von `using` wurde deutlich sichtbar ein Refinement aktiviert. Refinements dienen damit vor allem als Programmierer-Lesehilfe für den Quelltext und der Vermeidung von Fehlern, die aufgrund der programmweiten Auswirkungen von Monkeypatches entstehen.

Refinements sind mit Ruby 2 in die Sprache eingeführt und erst langsam stabilisiert worden. Selbst mit Ruby 2.5.1 sind noch immer nicht alle Aspekte endgültig festgelegt; so weist die Dokumentation ausdrücklich darauf hin, daß das Zusammenspiel von Refinements mit `send` [**TODO: Querverweis**] noch nicht abschließend geklärt ist. Auch wenn sie gegenüber Monkeypatches vorhersehbarer sind, sollte mit ihnen möglichst sparsam umgegangen werden. Der korrekte Weg, Fehler in Drittbibliotheken zu beheben, ist es, diese Fehler an den Autor der Drittbibliothek zu melden und dann dessen Update der Drittbibliothek abzuwarten und einzuspielen. So profitieren auch andere Nutzer der Drittbibliothek von der Fehlerkorrektur.

Angenommen, eine Programmbibliothek stellt eine Klasse `Spreadsheet` zur Verfügung, in der eine Methode `subtract` existiert, um die einzelnen angegebenen Zellen sukzessive voneinander abzuziehen:

```
# spreadsheet.rb in Drittbibliothek
class Spreadsheet
  # ...
  def subtract(cells)
    cells.map(&:val).inject(cells.shift) do |val, memo|
      memo - val
    end
  end
end
```

Diese Methode hat einen Bug. Der Programmierer hat die Blockparameter

von Enumerable#inject verwechselt. Weil Subtraktion nicht dem Kommutativgesetz folgt, ergibt diese Methode falsche Ergebnisse. Setzt man die Programmbibliothek produktiv (= in einem am Markt befindlichen Produkt) ein und ein Kunde beschwert sich, dann will man das Problem schnellstmöglich beheben. Besucht der Autor der Programmbibliothek aber momentan die RubyKaigi und schließt einen längeren Japan-Urlaub an, dann kommt ein Update möglicherweise nicht mehr rechtzeitig, um Schadensersatzansprüche des Kunden abzuwenden. Eine rasche Lösung muß her, und dafür setzt man ein Refinement ein.

Ein Refinement wird mithilfe von Module#refine definiert. Nur Klassen können Gegenstand eines Refinements sein. Das Refinement selbst ist ein anonymes Modul, auf welches `self` innerhalb des an `refine` übergebenen Blocks zeigt.

```
module FixSubtractionInSpreadsheet
  refine Spreadsheet
    def subtract(cells)
      cells.map(&:val).inject(cells.shift) do |memo, val|
        memo - val
      end
    end
  end
end
```

Der Einsatz erfolgt dann mithilfe der Methode `using`. Diese kann entweder auf der obersten Ebene eingesetzt werden oder innerhalb einer Klasse oder eines Moduls. In allen Fällen wirken Refinements lexikalisch ab Aufruf von `using` und enden mit dem — lexikalischen — Ende des momentenen Gültigkeitsbereichs. Bei Neuöffnung einer Klasse oder in anderen Dateien wird das Refinement nicht angewandt. Dies dient der Lesbarkeit des Quelltexts, da der Einsatz eines Refinements unbedingt gut sichtbar sein muß.

```
# spreadsheet_gui.rb in eigenem Programm
class SpreadsheetGUI
  using FixSubtractionInSpreadsheet
  # ...
  def preview_subtraction(cells)
    render_formula @spreadsheet.subtract(cells)
  end
```

```
end
```

Damit verwendet `preview_subtraction` jetzt die Version ohne Bug und ein Notfall-Release der Tabellenkalkulations-GUI kann angestoßen und an den Kunden verteilt werden. Dennoch ist der Fehler selbstverständlich an den Autor der Spreadsheet-Bibliothek zu melden, damit er nach seiner Rückkehr aus Japan das Problem beheben kann. Idealerweise stellt man ihm das Refinement als Patch direkt zur Verfügung.

9.4 Modulmethoden und -funktionen

Module sind wie alles in Ruby Objekte. Es handelt sich um Instanzen der Klasse `Module`. Für sie gelten deshalb keine besonderen Regeln bei der Definition von Methoden. Definiert man eine Singleton-Methode (→ 8.3.2) auf einem Modul, spricht man von *Modulmethoden*. Sie verhalten sich analog zu Klassenmehoden (→ 8.3.3). Ein typischer Anwendungsfall für solche Modulmethoden ist die Definition einer Setup-Methode, die sich darum kümmert, daß für die Nutzung einer Programmbibliothek alle notwendigen Anstalten getroffen werden. Solche Methoden hat der Nutzer der Programmbibliothek in aller Regel zu Beginn seines Programms aufzurufen.

```
module MyGraphics
  def self.setup(compat:)
    # ...
  end
end

# Im Hauptprogramm dann:
MyGraphics.setup(compat: "2.5.0")
```

Ein Sonderfall der Modulmethoden sind die Modulfunktionen. Eine Methode kann zur Modulfunktion geändert werden, indem man ihren Namen an die (Meta-)Methode Module::module_function übergibt. Die Methode wird dadurch noch einmal in die Eigenklasse des Moduls kopiert. Im Ergebnis steht die Methode dann sowohl direkt über das Modul als Methodenempfänger wie auch bei Verwendung des Moduls als Mixin (→ 9.2) zur Verfügung.

Modulfunktionen werden eingesetzt, wenn ein Modul nur in sich geschlossene Methoden bereitstellen und nicht primär als Namespace (→ 9.1) dienen soll. Ein Beispiel aus Rubys stdlib bildet die Programmbibliothek fileu-

tils. Diese stellt Methoden bereit, die den UNIX-Kommandozeilenwerkzeugen zur Verarbeitung von Dateien und Verzeichnissen dienen (z. B. `mkdir`, `ln_s`, `rm_rf`, etc.). Sie enthält zahlreiche Ausdrücke wie diesen:

```
module FileUtils
  def mkdir(list, options = {})
    # ...
  end
  module_function :mkdir
end
```

Dieser Code entspricht semantisch folgendem Code:

```
module FileUtils
  def mkdir(list, options = {})
    # ...
  end
  def self.mkdir(list, options = {})
    # Gleicher Inhalt
  end
  private :mkdir
end
```

`module_function` ermöglicht somit die Verwendung von `mkdir` sowohl über das Modul `FileUtils` als Empfänger wie auch per Mixin. Letzteres ist besonders praktisch, wenn man shell-artige Skripte schreiben möchte und wird insbesondere vom Automationswerkzeug Rake, das ebenfalls Teil der stdlib ist, genutzt.

```
FileUtils.mkdir "mydir"
# Oder gleichwertig:
include Fileutils
mkdir "mydir"
```

Soll eine große Zahl Methoden zu Modulfunktionen gemacht werden, kann man `module_function` ähnlich wie die Sichtbarkeitsmodifikatoren (→ 8.9) benutzen und die zu ändernde Methode weglassen. In diesem Fall werden alle nachfolgenden Methoden von vornherein als Modulfunktionen definiert.

Erwähnenswert ist, dass die Methoden kopiert werden und nicht bloß eine Weiterleitung stattfindet. Wenn nach Anwendung von `module_function` eine der beiden Methoden geändert wird, bleibt die andere davon unberührt.

9.5 Verhältnis von Class zu Module

Die in Ruby für Klassen zuständige Klasse `Class` ist eine Kindklasse von `Module`. Klassen dürfen deshalb in den meisten Fällen wie Module behandelt werden. Sie können lediglich nicht als Argument für Methoden wie `include` benutzt werden, die explizit ein Modul erwarten. Ein solcher Versuch löst zur Laufzeit einen für Ruby seltenen TypeError aus.

```
class Foo
  module Bar
  end
end # OK

include Foo #=> TypeError
```

9.6 Wichtige vordefinierte Module

Ruby bringt bereits ab Werk eine Reihe von Modulen mit. Auf die wichtigsten von ihnen soll nachfolgend kurz eingegangen werden.

9.6.1 Kernel

Das Modul `Kernel` nimmt eine Sonderstellung ein. Es definiert grundlegende Methoden, die überall in einem Ruby-Programm von Nutzen sein können. `Kernel` ist ein Mixin und wird von der allen (normalen) Klassen zugrundeliegenden Klasse `Object` [**TODO: Querverweis OOP**] eingemischt, d. h. die dort definierten Methoden stehen allen Objekten zur Verfügung. Eine weitere Besonderheit ist, daß die in `Kernel` definierten Methoden ausnahmslos privat (→ 8.9) sind, sodaß ein Aufruf nur innerhalb der einzelnen Klassen möglich ist.

In `Kernel` sind etwa die Ausgabemethoden `puts`, `print` und `p` definiert, aber auch die für Laufzeitfehler [**TODO: Querverweis**] wichtige Methode `raise` und die zur Erstellung anonymer Methoden relevante Methode `lambda`.

Außerdem werden einige für typische Textverarbeitungsskripte wichtige Methoden wie `readline` oder `sub` definiert und natürlich die für die Einbindung von Programmbibliotheken unentbehrliche Methode `require`. Schon diese Übersicht zeigt, daß `Kernel` ein weites Spektrum an Anwendungsfällen abdeckt und eine Vorstellung der einzelnen Methoden an dieser Stelle nicht sinnvoll ist. Auf einzelne Methoden wird deshalb an inhaltlich entsprechender Stelle eingegangen. Merken sollte man sich, daß hinter einer Methode, die scheinbar überall verfügbar ist und die nie mit einem ausdrücklichen Empfänger aufgerufen wird, sehr häufig das Modul `Kernel` steckt. Das ist wichtig, um die zur Methode gehörige Dokumentation zu finden. Vieles, was in anderen Programmiersprachen ein Schlüsselwort ist, ist in Ruby eine Methode von `Kernel`.

`Kernel` ist entgegen seines Namens auch nur ein normales Modul. Es ist deshalb auch ohne weiteres möglich, es um neue Methoden zu erweitern, die entsprechend überall verfügbar sind.

9.6.2 Math

Das Modul `Math` enthält zahlreiche mathematische Funktionen, darunter etwa die trigonometrischen Funktionen Sinus, Cosinus und Tangens, diverse Logarithmusfunktionen und andere. Außerdem definiert es die Konstanten Math::PI und Math::E für die Kreiszahl π und die Euler'sche Zahl e.

Die im Modul `Math` definierten Methoden sind sämtlich Modulfunktionen, d. h. das Modul kann auch per `include` eingebunden werden.

```
Math::PI       #=> 3.1415...
Math.cos(1.0) #=> 0.5403...
include Math
cos(1.0)       #=> 0.5403...
```

9.6.3 Enumerable

Das Mixin `Enumerable` stellt einen ganzen Satz von Methoden zur Arbeit mit Containern bereit. Die inkludierende Klasse muß lediglich eine Methode `each` definieren, die alle Objekte im Container nacheinander an den Block übergibt. Durch Einbindung von `Enumerable` werden darauf aufbauend dann mit Stand Ruby 2.3.3 ganze 53 Methoden verfügbar, darunter insbesondere viele Methoden, die für einen funktionalen Programmierstil wichtig sind

(`map`, `reduce`, etc.).

Ruby bietet selbst nur die generischen Container-Klassen `Array` und `Hash` an, die zwar für viele Aufgaben geeignet, aber nicht immer performant sind. Es gibt deshalb gute Gründe, bestimmte Konzepte für Container selbst zu implementieren, um so die maximale Performanz zu erreichen. Solche selbstgeschriebenen Container-Klassen sollten immer `Enumerable` einbinden, um dem Benutzer das gewohnte API für die Arbeit mit Containern anzubieten. Wenn im Einzelfall eine Implementation in `Enumerable` ungeeignet ist, kann die spezifische Methode in der eigenen Klasse überschrieben werden. Aber auch abseits von spezifischen Containern kann sich im Einzelfall die Gelegenheit bieten, über ein Objekt zu iterieren. Die für Dateien zuständige Klasse `IO` etwa inkludiert ebenfalls `Enumerable` und definiert `each` so, daß bei jedem Blockaufruf die nächste Zeile in der Datei übergeben wird.

Praktisch wohl keine gute Idee (weil Rubys Arrays performanter sind), aber didaktisch als Beispiel für die Nutzung von `Enumerable` sinnvoll ist die Implementation einer einfach verlinkten Liste. Dabei verweist jedes Listenelement auf das gehaltene Objekt sowie auf das nächste Listenelement. `each` wird dann so definiert, daß es die Liste Element für Element abgeht, dabei den Verweisen auf das jeweils nächste Listenelement folgt und das jeweils gehaltene Objekt an den Block übergibt.

```
class LinkedListNode
  include Enumerable

  attr_accessor :value
  attr_reader   :next

  def initialize(value = nil)
    @value = value
    @next = nil
  end

  def add(next_value)
    @next = LinkedListNode.new(next_value)
  end

  # TODO: while-Schleife!
  def each
```

```
    node = self
    loop do
      yield(node.value)
      break unless node.next
    end
  end
end

# Schönen Namen für den Container
LinkedList = LinkedListNode

# Benutzung
l = LinkedList.new(5)
l.add(5).add(10)
l.reduce(0){|sum|, el| sum + el} #=> 20
```

9.6.4 Comparable

Viele Objekte können miteinander verglichen werden. Ruby bietet dazu die Operator-Methoden <, <=, <=>, >= und > sowie die Methoden #between?, #clamp und #eql? [**TODO: Querverweis**]. Diese kann man selbst implementieren, aber da sie letztlich alle logisch voneinander abhängen, ist das unnötige Arbeit. Ruby wäre nicht „der beste Freund eines Programmierers“, wenn es für diese eintönige Aufgabe nicht eine Lösung anbieten würde. In diesem Fall ist die Lösung das Mixin `Comparable`. Der Programmierer definiert nur noch eine Methode <=>, die übrigen Operator-Methoden stellt dann `Comparable` bereit. Lediglich die Methode #eql? nimmt eine Sonderstellung ein. Diese Methode wird nicht von `Comparable` bereitgestellt, allerdings ist #eql? bereits in der Klasse `Object` als Alias (→ 8.4) auf die Methode #== definiert. Weil ein Alias allerdings die neue Methode nicht als Weiterleitung, sondern als Kopie anlegt, kommt es durch Einbindung von `Comparable` über <=> bzw. auch ganz generell bei Definition der Operator-Methode <=> zu einer Diskrepanz zwischen #eql? und #==, die meist nicht gewünscht ist. Rubys Dokumentation empfiehlt daher zurecht, daß, wer `Comparable` einbindet oder #== selbst überschreibt, den Alias zu #eql? noch einmal neu anlegen sollte.

Die Methode #equal? sollte im Gegensatz zu #eql? niemals überschrieben werden. Diese Methode vergleicht die Objekt-Identität (#object_id) und ist damit Teil von Rubys Metaprogrammierungs-Funktionalität.

<=> nimmt das zu vergleichende Objekt entgegen und muß einen von vier Werten zurückgeben:

- -1, wenn self kleiner ist als das Argument.
- 1, wenn self größer ist als das Argument.
- 0, wenn self gleich dem Argument ist.
- nil, wenn die beiden Objekte nicht vergleichbar sind.

Die oben vorgestellte einfach verlinkte Liste könnte so etwa anhand der Länge der Liste verglichen werden:

```
class LinkedListNode
  include Enumerable
  include Comparable
  # ...wie gehabt...
  def <=>(other)
    return nil unless other.kind_of?(self.class)
    count <=> other.count
  end
  alias eql? ==
end
```

Nachdem zunächst sichergestellt wird, daß hier nicht Äpfel mit Birnen verglichen werden — Rückgabewert nil, wenn das Argument keine Liste ist — wird die Länge der Listen verglichen. Hierbei wird auf die — vom ebenfalls inkludierten `Enumerable` zur Verfügung gestellte — Methode `count` zurückgegriffen. Um ein etwas längliches if/else abzukürzen, wird für den eigentlichen Vergleich der Längen, bei dem ja nur noch Integers beteiligt sind, auf die <=>-Methode der Klasse `Integer` zurückgegriffen. Dieser Kunstgriff spart Code und ist weniger fehleranfällig als die manuelle Implementation, aber nicht in allen Fällen möglich.

Wiederholungsfragen und -aufgaben

1. Wofür benötigt man Namespaces?
2. Was ist ein Monkeypatch, was ein Refinement? Welchem Zweck dienen diese Institute?
3. Wofür benötigt man `include`?
4. Lesen Sie sich über binäre Suchbäume ein, z. B. in der Wikipedia. Implementieren Sie diese Datenstruktur in einer Klasse `BinaryTree` in Ruby und setzen Sie dabei `Enumerable` und `Comparable` ein, um die binären Suchbäume iterier- und vergleichbar zu machen.

§ 10 Zahlen

Zahlen gehören zu den grundlegenden Datentypen, mit denen jede Programmiersprache umgehen können muß. Ruby macht da keine Ausnahme.

Alle Zahlen in Ruby sind vollwertige Objekte und eine Instanz einer der numerischen Klassen der Programmiersprache. Diese Klassen sind in einer Hierarchie angeordnet, die ihre Wurzel in der Klasse `Numeric` hat und die in Abb. 10.1 dargestellt ist. Hierbei ist auf eine kürzlich erfolgte Änderung hinzuwiesen: mit Ruby 2.4 sind die Klassen `Bignum` und `Fixnum` mit `Integer` vereinigt worden.

10.1 Ganzzahlen

10.1.1 Allgemeines

Eine Ganzzahl (*Integer*) ist in Ruby einfach durch Einfügen der Zahl als Literal in den Code erreichbar. Da ein entsprechendes Beispiel geradezu trivial wäre, soll stattdessen darauf aufmerksam gemacht werden, daß Zahlenliterale mithilfe von Unterstrichen aufgetrennt werden können, ohne daß die Unterstriche den Zahlwert beeinflussen würden; tatsächlich betreffen sie nur die Sprachgrammatik und sind in der späteren Zahl nicht mehr sichtbar. Ihr Einsatz bietet sich besonders für große Zahlen an, bei denen der Unterstrich üblicherweise als Tausendertrenner benutzt wird:

```
num = 1_000_000 # Eine Million
puts num        #=> 1000000
```

Wissenschaftliche Notation in der Form xen ist zulässig; das entspricht $x \cdot 10^n$. Sie ist für Integers aber unüblich und findet meist nur bei Floats Verwendung.

```
puts 1e6 #=> 1000000
```

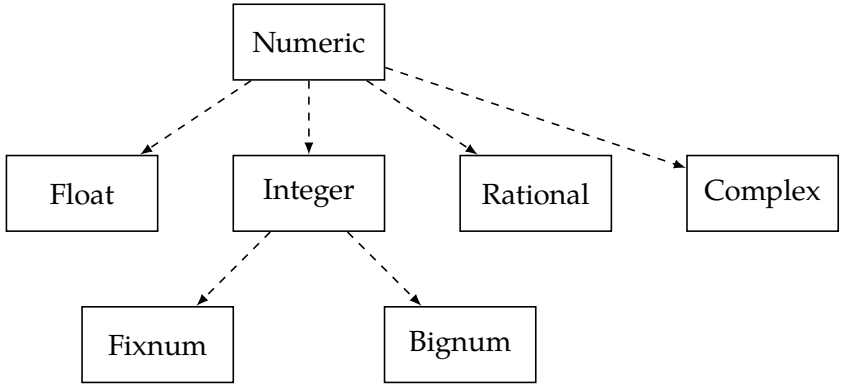

Abbildung 10.1: Zahlenklassen in Ruby

Normale Integer-Literale beziehen sich stets auf Zahlen zur Basis 10, wie es im natürlichen Sprachgebrauch üblich ist. Je nach Kontext kann das unpraktisch sein, weshalb Ruby die Möglichkeit bietet, Binär- (Basis 2), Hexadezimal- (Basis 16) und Oktalzahlen (Basis 8) zu notieren. Binärzahlen werden durch das Präfix 0b gekennzeichnet, Hexadezimalzahlen durch 0x. Oktalzahlen werden dagegen nur über eine führende Null gekennzeichnet. Soweit aus Gründen der Einheitlichkeit sinnvoll, kann eine Dezimalzahl mithilfe des Präfixes 0d ausdrücklich als solche gekennzeichnet werden. Alle nachfolgenden Beispiele notieren dieselbe Zahl:

```
puts 14      #=> 14
puts 0d14    #=> 14
puts 0xE     #=> 14
puts 0b1110  #=> 14
puts 016     #=> 14
```

Insbesondere bei Oktalzahlen sollte man genau hinschauen. 016 ist nämlich eben nicht 16, sondern 14.

10.1.2 Fixnum und Bignum

Bis Ruby 2.4 wurden Integers in Ruby nach ihrem Speicherbedarf in Fixnums und Bignums unterschieden. Fixnums waren nur solche Integers, die sich in einem Wort — der einfachsten Dateneinheit eines Prozessors — minus 1 Bit repräsentieren ließen. Ein weiteres Bit muß für das Vorzeichen abgezogen

werden. Da heute gängige PC-Prozessoren typischerweise eine Wortbreite von 64 Bit besitzen, war die größtmögliche Fixnum-Zahl als $2^{62} - 1$ zu bestimmen. 2^{62} war die erste Bignum-Zahl.

Mit Ruby 2.4 sind die Klassen Fixnum und Bignum als Interna entfallen und Rubys öffentliches API wurde dem Standard ISO/IEC 30170:2012 angepaßt. Diese Änderung hat insbesondere ältere, ungewartete C-Extensions getroffen, die seit Ruby 2.4 nicht mehr funktionfähig sind. Ruby-Code, der eigentlich ohnehin nie direkt Gebrauch von Fixnum und Bignum machen sollte, blieb dagegen von der Änderung weitestgehend verschont, da seit Ruby 2.4 Fixnum und Bignum als Konstanten beibehalten wurden, die allerdings nurmehr auf die Klasse Integer verweisen.

Die Logik von Fixnum und Bignum spielt allerdings auch weiterhin eine Rolle. Fixnums waren und sind *Immediate Values*, d. h. intern übergibt Ruby keine Referenzen, sondern das Objekt selbst. Wichtigste Konsequenz hieraus ist, daß Fixnums keine Eigenklasse besitzen; jede einzelne Zahl im Fixnum-Bereich ist also gewissermaßen bereits ein Singleton. Das ist auch logisch. Es würde keinen Sinn ergeben, zwischen 5 und 5 zu unterscheiden, denn es gibt nun einmal nur eine 5. Bignums dagegen werden bei Bedarf erzeugt und sind vollwertige Objekte. Aus Konsistenzgründen besitzen allerdings auch Bignums keine Eigenklasse. Wer genau hinschaut, kann den Unterschied aber noch erkennen. Fixnums besitzen wegen ihrer Eigenschaft als „Immediate Values“ immer dieselbe Objekt-ID, Bignums dagegen nicht.

```
f1 = 5
f2 = 5
b1 = 2 ** 62
b2 = 2 ** 62
p f1.object_id #=> 11
p f2.object_id #=> 11
p b1.object_id #=> 4728268340344O
p b2.object_id #=> 47282683403360
p f1.class     #=> Fixnum (oder neu: Integer)
p b1.class     #=> Bignum (oder neu: Integer)
```

Es gehört zu den Kuriositäten von Ruby, daß die Objekt-ID eines Fixnums immer $2n + 1$ ist, wobei n den Wert des Fixnums darstellt. Bignums haben dagegen reguläre Objekt-IDs. Die Objekt-ID von nil ist übrigens 8 (früher: 4), von false 0 und von true 20.

10.1.3 Konvertierung

Es gibt verschiedene Möglichkeiten, Integers in Strings und umgekehrt zu konvertieren. Die wichtigste Art der Konvertierung wird mithilfe der Methoden String#to_i und Integer#to_s durchgeführt, die jeweils das tun, was man ihrem Namen nach von ihnen erwarten kann. Beide Methode akzeptieren ein Argument, mit dem man die Zahlenbasis angeben kann, um so etwa mit Hexadezimalzahlen arbeiten zu können. Als Zahlen ungültige Strings gelten als Null.

```
p 28.to_s        #=> "28"
p 28.to_s(16)    #=> "1c"
p "28".to_i      #=> 28
p "28".to_i(16)  #=> 40
p "foo".to_i     #=> 0
```

Tatsächlich ist die Konvertierung nach Integer nicht auf Strings beschränkt. Jede beliebige Klasse kann durch Definition einer Methode #to_i eine Konvertierungsmöglichkeit zu Integer eröffnen. Es handelt sich um eine ganz gewöhnliche Methode.

```
class Line
  def initialize(length)
    @length = length
  end
  def to_i
    @length
  end
end

Line.new(15).to_i #=> 15
```

Klassen, die nicht nur nach Integer konvertibel sein sollen, sondern sogar wie ein Integer behandelt werden sollen, definieren darüber hinaus die Methode #to_int. Dabei handelt es sich um einen Aspekt von Rubys *Duck Typing*, dazu [**TODO: Querverweis**].

Eine andere Möglichkeit, ein Objekt nach Integer zu konvertieren, ist die Methode #Integer. Der wichtigste Unterschied zu #to_i ist, daß #Integer Präfixe honoriert, die die Basis der Zahl angeben. Die unterstützten Präfixe sind

dieselben, die auch bei der Eingabe im Quellcode direkt genutzt werden können.

```
p Integer("10")   #=> 28
p Integer("0x10") #=> 16
p Integer("0b10") #=> 2
```

Ein weiterer Unterschied ist, daß #Integer bei ungültigem Argument einen ArgumentError auslöst. Das kann wichtig sein, um fehlerhafte Nutzereingaben zu erkennen.

```
input = read_from_user
# Nutzer gibt "foo" ein
p input.to_i      #=> 0 # Ups!
p Integer(input) #=> ArgumentError # Besser
```

Auch #Integer akzeptiert als zweites Argument die ausdrückliche Angabe der Basis. Kommt es zum Konflikt mit dem im String vorhandenen Präfix, wird ein ArgumentError ausgelöst.

Für die Konvertierung von Ganzzahlen in Strings stellen die beiden Methoden #printf und #sprintf diverse Optionen bereit, die zur Formatierung der Zahlen benützt werden können. Diesbezüglich sei auf deren Dokumentation verwiesen; nachfolgend nur ein kurzes Beispiel dazu, wie man eine bestimmte Anzahl von Nullen hinter dem Dezimaltrenner erreichen kann.

```
printf("%.3f", 1.4) #=> 1.400
```

10.1.4 Rechenoperationen

Rubys Integers unterstützen Addition, Subtraktion, Multiplikation und Division mithilfe der üblichen (Ersatz-)Operatoren +, -, * und /. Darüber hinaus unterstützt Ruby Potenzierung mithilfe eines Doppelstern-Operators ** und Modulorechnung (Restberechnung) mithilfe des Prozent-Operators %.

```
puts 1 + 3  #=> 4
puts 6 - 1  #=> 5
puts 2 * 4  #=> 8
puts 8 / 2  #=> 4
```

```
puts 2 ** 3 #=> 8
puts 5 % 2  #=> 1
```

Die Division durch Null ist unzulässig und verursacht einen ZeroDivisionError.

Rubys Division von ganzen Zahlen ist die in vielen Programmiersprachen übliche und letztlich auf Prozessorinstruktionen zurückgehende Ganzzahldivision. Wenn das Ergebnis nicht als ganze Zahl darstellbar ist, wird der Teil hinter dem Komma „abgeschnitten“ (in Wirklichkeit tritt er bei Durchführung der Berechnung auf dem Prozessor nicht einmal — auch nicht temporär — auf). Der verbleibende Rest kann mithilfe des Modulo-Operators ermittelt werden. Beispielsweise entsteht bei der Division von 9 durch 4 die Ganzzahl 2 und ein Rest von 1.

```
puts 9 / 4 #=> 2
puts 9 % 4 #=> 1
```

Ist die Ganzzahldivision unterwünscht, muß man auf andere numerische Klassen zurückgreifen. Ruby bietet dafür Float (→ 10.2) und Rational (→ 10.3).

Die Mißachtung der Regeln der Ganzzahldivision ist ein typischer Anfängerfehler.

10.1.5 Bitoperationen

Für die Arbeit mit binären Daten nutzt man in Ruby vorwiegend Strings. Soweit allerdings einzelne Bits bearbeitet werden müssen, muß man hierfür mangels Alternativen auf Integers zurückgreifen. Ruby unterstützt in dieser Hinsicht alle wichtigen Operationen auf Bitmustern. Ein Bitmuster ist letztlich nichts anderes als ein Integer, dessen Zahlwert nicht von Interesse ist — es kommt allein auf die Werte der einzelnen Bits im Arbeitsspeicher (→ 3.1.1) an.

Der Zugriff auf die einzelnen Bits erfolgt ähnlich wie bei Arrays durch Indexierung mithilfe von eckigen Klammern. Rubys Integers werden als „Little Endian“ gespeichert, sodaß die Einerstelle (Least Significant Bit, LSB) mit Index Null am Anfang steht, d. h. die Zahl gewissermaßen verkehrt herum gespeichert wird. Grund dafür ist das Design heutiger Prozessoren, die so performanter arbeiten können.

```
p 0b1100[0] #=> 0
p 0b1100[1] #=> 0
p 0b1100[2] #=> 1
p 0b1100[3] #=> 1
```

Ein Schreibzugriff ist wegen der Unveränderbarkeit von Integern nicht möglich. Hierfür ist auf Binäroperationen zurückzugreifen. Ruby unterstützt das binäre UND (AND), das binäre ODER (OR) und das binäre Exklusiv-Oder (XOR). Bei Durchführung einer solchen Operation wird das gesamte übergebene Bitmuster auf das gesamte Bitmuster des Empfängers angewandt. Wer nur ein einzelnes Bit kippen möchte, muß das deshalb mithilfe einer entsprechenden Binäroperation durchführen. Jedes Bit im Empfänger wird also einzeln mit dem korrespondieren Bit im Operator verglichen. Es gilt:

- Das binäre UND ergibt 1 für zwei Bits, wenn beide Bits 1 sind, sonst 0.
- Das binäre ODER ergibt 1, wenn wenigstens eines der beiden Bits 1 ist, sonst 0.
- Das binäre Exklusiv-ODER ergibt 1, wenn nur einer der beiden Operanden 1 ist, sonst 0.

Das normale und das Exklusiv-ODER unterscheiden sich daher, wenn beide Operanden 1 sind. Das normale ODER ergibt hier 1, das Exklusiv-ODER 0. Tafel 10.1 faßt alle möglichen Bitoperationen zusammen.

Die einzelnen Operatoren in Ruby lauten & (AND), | (OR) und ^ (XOR). Sie dürfen nicht mit den logischen Operatoren && und || und nicht mit dem Potenzierungsoperator ** verwechselt werden.

```
printf "%04b", 0b1100 & 0b0110 #=> 0100
printf "%04b", 0b1100 | 0b0110 #=> 1110
printf "%04b", 0b1100 ^ 0b0110 #=> 1010
```

Ruby unterstützt daneben noch einige weitere Bitoperationen. Mit ~ (Tilde) kann man ein Bitmuster vollständig invertieren (Komplement). Da Rubys Integers beliebig groß werden können, merkt Ruby sich in diesem Fall, daß eine unendliche Anzahl von Einsen am Anfang steht, bei der Ausgabe dargestellt durch zwei Pünktchen.

Op.1	Op.2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabelle 10.1: Bitoperationen

```
printf "%04b", ~0b1100 #=> ..10011
```

Praktisch wichtiger sind die Bitshift-Operatoren ⟨⟨ und ⟩⟩. Das Bitmuster wird dabei um die im Operanden angegebene Anzahl von Stellen in die Richtung verschoben, in die der Operand zeigt. Auf der anderen Seite rücken Nullen nach.

```
printf "%04b", 0b0110 >> 1 #=> 0011
printf "%04b", 0b0110 >> 2 #=> 0001
printf "%04b", 0b0110 >> 3 #=> 0000
printf "%04b", 0b0110 << 1 #=> 1100
printf "%04b", 0b0110 << 2 #=> 11000
printf "%04b", 0b0110 << 3 #=> 110000
```

10.2 Fließkommazahlen

10.2.1 Allgemeines

Ruby bietet zur Arbeit mit Kommazahlen die Klasse Float an. Solche Floats kann man in Ruby einfach durch Benutzung eines (US-amerikanischen) Dezimalpunkts erzeugen. Wissenschaftliche e-Notation wird wie bei Integern (→ 10.1.1) zur Basis 10 unterstützt. Auch die Trennung mit Unterstrichen ist möglich.

```
num = 4.5
num = 4.5e4 # 45000
num = 45_000.5
```

Seit längerem unzulässig ist das Weglassen der führenden Null, wenn Null gemeint ist. Das ist ein Unterschied zu einigen anderen Programmiersprachen.

```
num = .5
#=> SyntaxError: no .<digit> floating literal
#=> anymore; put 0 before dot
```

10.2.2 Rechenoperationen

Floats unterstützen die gleichen mathematischen Operationen wie Integers (→ 10.1.4), allerdings gibt es einige Besonderheiten.

Zunächst ist relevant, daß die Division keine Ganzzahl, sondern wieder ein Float hervorbringt, bei dem der Bruchteil hinter dem Komma berücksichtigt wurde. Zur Aktivierung der Float-Division genügt es, wenn einer der beiden Operanden ein Float ist.

```
puts 1.0 + 3.0  #=> 4.0
puts 9.0 / 4.0  #=> 2.25
puts 9.0 / 4    #=> 2.25
```

Auch ist die Division durch Null zulässig. Sie ergibt unendlich, entweder positiv oder negativ. Unendlich steht auch als Konstante Float::INFINITY zur Verfügung. Mit unendlich kann man sogar rechnen, die Ergebnisse sind aber immer wieder bloß unendlich.

```
puts 4.0 / 0.0           #=> Infinity
puts -4.0 / 0.0          #=> -Infinity
puts Float::INFINITY     #=> Infinity
puts Float::INFINITY - 1 #=> Infinity
```

Floats besitzen noch einige andere mathematische Unarten. So unterscheiden sie zwischen +0 und -0. Einige Operationen ergeben zudem den Sonderwert NaN („Not a Number“, engl. „Keine Zahl“), der auch als Konstante Float::NAN abgerufen werden kann.

```
p -0.0       #=> -0.0
p 0.0        #=> 0.0
p 0.0 / 0.0  #=> NaN
p Float::NAN #=> NaN
```

Während sich normale Floats in Größenvergleichen wie erwartet verhalten und das sogar für Float::INFINITY (positiv wie negativ) gilt, ist der Vergleich mit NaN schwierig. NaN ist nie größer oder kleiner einer beliebigen Zahl, sondern immer anders — jegliche Größenvergleiche mit NaN ergeben daher false und ein Gleichheitstest mit NaN auch. Vergleicht man NaN mit NaN, ist das Ergebnis laut Ruby-Dokumentation implementierungsabhängig; Rubys kanonische Implementation, der MRI, gibt jedenfalls in Version 2.3.3 false zurück. Der Vergleich von -0 mit +0 ist immer wahr.

10.2.3 Konvertierung

Integers und Strings können mithilfe von #to_f in ein Float konvertiert werden. Ähnlich wie schon bei Integern führen ungültige Zahlwerte in Strings zum Wert Null und ebenso wie bei Integer gibt es eine Methode #Float, die in diesem Fall einen ArgumentError auslöst.

```
puts "foo".to_f    #=> 0.0
puts Float("foo") #=> ArgumentError
```

Wenn man zwei Integers nach den Float-Regeln dividieren will, konvertiert man mithilfe von Integer#to_f daher einfach wenigstens einen der beiden Operanden zu Float.

```
puts x / y.to_f
```

Umgekehrt kann man einen Float mit #to_i und #to_s in einen Integer oder einen String umwandeln. Bei der Umwandlung in ein Integer mit #to_i wird der Teil hinter dem Komma abgeschnitten (nicht gerundet).

```
p 3.6.to_s #=> "3.6"
p 3.6.to_i #=> 3
```

Dies vermeidet der Einsatz der Method #round, die so rundet, wie man es erwartet. #round akzeptiert sogar noch einige Argumente, um statt auf ganze Zahlen auf eine bestimmte Anzahl Nachkommastellen zu runden und/oder den Umgang mit einem halben Wert zu kontrollieren (standardmäßig wird .5 mathematisch richtig aufgerundet).

	3.7	3.5	3.1	-3.7	Effekt
floor	3	3	3	-4	Nach unten abschneiden
ceil	4	4	4	-3	Nach oben abschneiden
truncate	3	3	3	-3	Zur Null abschneiden
to_i	3	3	3	-3	Wie truncate
round	4	4	3	-4	Runden

Tabelle 10.2: Abschneiden von Floats

```
p 3.6.round #=> 4
p 3.5.round #=> 4
p 3.4.round #=> 3
p 3.13.round(1) #=> 3.1
```

Daneben gibt es noch die Methoden #ceil, #floor und #truncate, die in bestimmte Richtungen abschneiden. Siehe dazu Tafel 10.2.

10.2.4 Genauigkeit von Floats

Bei der Nutzung von Floats ist Vorsicht angebracht. Sie bieten sich nur scheinbar zur Rechnung mit beliebigen Zahlen an, denn sie haben ihre Tücken:

```
puts 9.4 - 0.2 #=> 9.200000000000001
```

Das Beispiel zeigt: Floats sind *ungenau*. Rubys Floats sind, wie der Klassenname schon andeutet, *Fließkommazahlen*, implementiert nach dem Standard IEEE 754 mit mit *double precision*. Diese Fließkommazahlen können viele Dezimalzahlen nicht präzise darstellen, sondern nur annähernd. Das wird deutlich, wenn man eine scheinbar präzise Zahl wie oben 0.2 einmal mit mehr als der von Ruby normalerweise angezeigten Anzahl Nachkommastellen ausgibt:

```
printf "%.30f", 0.2
#=> 0.200000000000000011102230246252
```

Für den Anfänger gilt deshalb die Faustregel:

> Keine Floats benutzen, wenn es auf präzise Berechnungen ankommt!

Das betrifft insbesondere, aber nicht nur Berechnungen mit Geldbeträgen. **[TODO: Anekdote Überweisung mit Subpfennigbeträgen]** An anderen Orten haben Fließkommazahlen dagegen wegen ihrer hohen Performanz in der Berechnung ihren rechten Platz. Die meisten Positionsberechnungen in Videospielen etwa benötigen kaum absolute Präzision; hier sind die Ungenauigkeiten weit hinter dem Komma meist vernachlässigbar. Sind präzise Berechnungen notwendig, sollte man auf Rational (→ 10.3) oder auf die Bibliothek bigdecimal aus der stdlib zurückgreifen.

Wegen der fehlenden Genauigkeit sollte man Floats auch niemals auf Gleichheit testen, sondern nur, ob sie innerhalb einer bestimmten Toleranzschwelle liegen. Mathematisch ist das nichts anderes als dies:

$$|x - y| \leq \delta \tag{10.1}$$

Dabei gibt δ die Toleranzschwelle an. In Ruby sähe das wie folgt aus, wobei man überlegen sollte, ob man die Operation nicht in eine eigene Methode auslagert:

```
x = 9.2
y = 9.4 - 0.2
delta = 0.001
p x == y                  #=> false (!)
p (x - y).abs <= delta #=> true
```

Natürlich sind Floats nicht unvorhersehbar ungenau, denn das würde der Funktionsweise von Computern widersprechen (→ 3.1.1). Die in IEEE 754 festgelegten Einzelheiten sind indessen kompliziert und für den Anfänger kaum von Interesse. Wer das Thema später vertiefen will, findet in **[TODO: Verweis Goldberg]** aber eine gute Erläuterung.

10.3 Rationale Zahlen

Die Probleme mit der Genauigkeit von Fließkommazahlen (→ 10.2.4) haben zur Einführung der Klasse Rational in die Programmiersprache Ruby geführt. Ihre Instanzen repräsentieren rationale Zahlen als Bruch mit Zähler und Nenner und sind deshalb im Rahmen dessen, was rationale Zahlen abbilden können, immer exakt. Ungenauigkeiten wie mit Fließkommazahlen sind ausgeschlossen. Rationals sind deshalb bei Berechnungen, die Genauigkeit erfor-

dern, die richtige Wahl. Sie sind dafür allerdings auch weniger performant als Floats.

Rationals können mithilfe des divisionsähnlichen r-Literals erstellt werden. Ruby kürzt den Bruch automatisch auf die kleinstmögliche Kombination. Unterstriche sind wie üblich zur Zahlentrennung möglich. Rational verbietet eine Division durch Null; die Zahl Null kann aber mit einem Zähler von Null für jeden anderen beliebigen Nenner dargestellt werden (Rational kürzt automatisch auf $\frac{0}{1}$).

```
p 2/3r       #=> (2/3)
p 3/6r       #=> (1/2)
p 1/100_000 #=> (1/100000)
p 0/10r      #=> (0/1)
```

Mithilfe der Methode #Rational kann man Rationals auch aus Variablen erstellen, die Zähler und Nenner angeben, sowie aus Strings, Integers und Floats. Mit #to_r unterstützen die anderen numerischen Klassen eine Konvertierung in ein Rational. Wird von Float konvertiert, wird die Ungenauigkeit der Fließkommazahl in den Rational übernommen (aber bei Berechnungen nicht verschlimmert).

```
p Rational(2, 3)  #=> (2/3)
p Rational(5)     #=> (5/1)
p Rational("2.5") #=> (5/2) # Exakt, weil String
p 10.to_r         #=> (10/1)
p 0.2.to_r        #=> (2589569785738035/281474976710656)
```

Umgekehrt können Rationals auch in andere Zahlentypen konvertiert werden. Bei einer Konvertierung nach Float geht die Präzision verloren, bei einer Konvertierung nach Integer die Dezimalstellen.

```
p 1/1r.to_i   #=> 1
p 1/10r.to_i  #=> 0
p 92/10r.to_f #=> 9.2
```

Die mit Rationals durchgeführten Rechenoperationen sind mathematisch exakt. Beispiel im Vergleich mit Float:

```
i=0.0
```

```
1000.times{i+=0.1}
puts i #=> 99.9999999999986

i=0/1r
1000.times{i+=1/10r}
puts i #=> (100/1)
```

Rational unterstützt dieselben Rundungs- und Abschneidemethoden wie Float (vgl. Tafel 10.2), arbeitet dabei jedoch präzise. Die Ergebnisse sind ihrerseits entweder Integers oder Rationals, jedenfalls aber genau.

```
p Rational("5.1234").round     #=> 5
p Rational("5.1234").round(2) #=> (128/25)
p Rational("5.1234").ceil      #=> 6
```

10.4 Komplexe Zahlen

Ruby unterstützt auch den Umgang mit komplexen Zahlen. Primär erstellt man sie über die algebraische Form, die Ruby als Literal und über die Methode #Complex unterstützt. Die Literale für rationale und komplexe Zahlen können kombiniert werden.

```
c = 3+4i           #=> (3+4i)
c = Complex(3, 4) #=> (3+4i)
c = 5/1r+11/5ri    #=> ((5/1)-(11/5)*i)
```

Daneben kann man komplexe Zahlen auch als Koordinaten in Rechteck- oder Polarform referenzieren.

```
c = Complex.rect(3, 1)          #=> 3+1i
c = Complex.polar(3, Math::PI) #=> (-3+0.0i)
```

Rubys Complex-Zahlen besitzen keine eigene Präzision; sie delegieren die Genauigkeit der Darstellung an die zugrundeliegenden Zahlenklassen. Handelt es sich um Floats, treten bei der späteren Verarbeitung die bereits behandelten Probleme auf (→ 10.2.4).

```
c = Complex(5.0, 2.2)     #=> (5.0+2.2i)
c = Complex(5/1r, 11/5r) #=> ((5/1)+(11/5)*i)
c = 5.0+11/5ri            #=> (5.0-(11/5)*i)
```

Die anderen Zahlentypen können mit #to_c in eine komplexe Zahl konvertiert werden, deren imaginärer Teil dann freilich Null ist.

```
c = 4.to_c #=> (4+0i)
```

Ruby unterstützt Rechenoperationen auch mit komplexen Zahlen, einschließlich derjenigen Operationen, die spezifisch nur für komplexe Zahlen existieren, z. B. Konjugation. Für Details sei auf die Dokumentation von Rubys Complex-Klasse verwiesen.

```
p 5+2i.conjugate #=> (5-2i)
```

§ 11 Strings und Symbole

Der Einsatz von Zeichenketten (engl. *Strings*) und den anverwandten Symbolen gehören zu den Grundkenntnissen, die man in der Programmierung mit Ruby beherrschen muß. Dieses Kapitel vertieft die in der Einführung angerissenen Problematiken und soll einen Überblick über alle relevanten Problemstellungen geben.

11.1 Erstellung

Strings in Ruby können auf verschiedene Arten erstellt werden. Die letztlich erstellten Strings unterscheiden sich in ihrer Natur aber nicht mehr voneinander, d. h. einem einzelnen String kann nicht angesehen werden, auf welche Art und Weise er erstellt worden ist. Die beiden Grundtypen der Erstellung von Strings sind die Erstellung mit doppelten oder einfachen Anführungszeichen:

```
str = "Text"
str = 'Text'
```

Die beiden Techniken unterscheiden sich leicht bei der Verarbeitung des Inhalts des String-Literals. Grob gesagt ist die String-Erstellung mit einfachen Anführungszeichen etwas »rudimentärer« als die mit doppelten Anführungszeichen. Dabei geht es im Wesentlichen um zwei Unterschiede: die Behandlung von Escape-Sequenzen und die Zulässigkeit von Interpolation.

11.1.1 Interpolation

Zunächst zur Interpolation. Sie ist nur bei Strings mit doppelten Anführungszeichen zulässig und ermöglicht es, innerhab des String-Literals beliebigen Code — auch mehrzeilig — auszuführen, dessen Ergebnis dann nach automatischem Aufruf der Methode #to_s dann in den String übernommen wird.

Der auszuführende Code befindet sich dabei in einer besonderen Klammerung, die durch die Zeichen #{...} begrenzt wird. Dazu einige Beispiele mit ansteigender Komplexität:

```
n = 3
ary = [1, 2, 3]
str = "Es ist #{5 + 2} Uhr"
str = "n ist #{n}"
str = "Zählung: #{ary.join(", ")}"
str = "Nicht empfehlenswert: #{class A
  def to_s
    "Schwer zu lesen"
  end
end
A.new}"
```

Innerhalb von Interpolationen können die Anführungszeichen beliebig verschachtelt werden, allerdings verliert man dann schnell die Übersicht.

11.1.2 Escape-Sequenzen

Escape-Sequenzen sind Zeichenkombinationen, die ein anderes Zeichen ergeben, als sie vordergründig darstellen. Dies dient der Darstellung sonst nicht menschenlesbar darstellbarer Zeichen, z.B. einem Zeilenumbruch oder dem Steuerzeichen zum Läuten der Terminal-Glocke (BEL). Escape-Sequenzen beginnen in Ruby mit einem Backslash \. Sie sind bis auf wenige Ausnahmen nur in String-Literalen mit doppelten Anführungszeichen möglich und werden in solchen mit einfachen Anführungszeichen ignoriert, d. h. dort werden die Zeichen 1:1 übernommen. Beispiel:

```
str1 = "String mit\nZeilenumbruch"
str2 = 'Kein\nZeilenumbruch'
puts str1
puts str2
```

Ergebnis:

```
String mit
Zeilenumbruch
Kein\nZeilenumbruch
```

Sequenz	ISO 646	Bedeutung
\'	'	Hochkomma maskieren
\"	"	Anführungszeichen maskieren
\#	n/a	Gatter maskieren (keine Interpolation)
\a	BEL	Terminal-Glocke läuten
\b	BS	Rückschritt
\t	TAB	Horizontaler Tabulator
\n	LF	Zeilenvorschub
\v	VT	Vertikaler Tabulator
\f	FF	Seitenvorschub
\r	CR	Wagenrücklauf
\e	ESC	Beginn einer ANSI-Sonder-Sequenz
\s	SP	Leerzeichen
\c?	DEL	Löschzeichen
\\	n/a	Rückwärtiger Schrägstrich
nnn	n/a	Byte als Oktalzahl *nnn*
\x*nn*	n/a	Byte als Hexadezimalzahl *nn*
\u*nnnn*	n/a	Unicode-Codepunkt als Hexadezimalzahl *nnnn*
\u*nnnn*...	n/a	Mehrere Unicode-Codepunkte

Tabelle 11.1: Escape-Sequenzen

Tafel 11.1 listet die wichtigsten Escape-Sequenzen auf. In String-Literalen mit einfachen Anführungszeichen sind von ihnen nur zulässig »\\« und »\'«.

String-Literale können mehrzeilig sein. Ein Zeilenumbruch kann dabei entweder ausdrücklich durch »\n« markiert werden (dazu schon das Beispiel oben) oder wird durch Erreichen des Zeilenendes markiert. (Nur) in String-Literalen mit doppelten Anführungszeichen möglich ist, einen derart mehrzeiligen String durch Nutzung eines Backslashes in einen einzeiligen zu verwandeln.

```
str = "Zeile 1.
Zeile 2."
str = 'Zeile 1.
Zeile 2.'
str = "Zeile 1.\
Immer noch Zeile 1."
str = 'Zeile 1.\
```

```
Zeile 2.'
```

11.1.3 Heredocs

Für mehrzeilige Strings sollte man aber auf eine ganz andere Technik zurückgreifen, nämlich auf sog. *Heredocs* (engl. für »Hier-Dokumente«). Diese werden durch den Operator ⟨⟨ eingeleitet und sehen dem Grunde nach so aus:

```
str =<<EOF
Erste Zeile.
Zweite Zeile.
EOF
```

Hinter ⟨⟨ folgt ein beliebig wählbarer Bezeichner, im Beispiel das oft gewählte »EOF«. Großbuchstaben für diesen Bezeichner sind Konvention. Das Heredoc endet dort, wo dieser Bezeichner in einer Zeile allein für sich steht; alles dazwischen — einschließlich der Zeilenumbrüche auch der letzten Zeile — wird Teil des Strings.

Heredocs dieser Art verhalten sich wie String-Literale mit doppelten Anführungszeichen, d. h. Interpolation und Escape-Sequenzen sind möglich. Heredocs können aber auch ausdrücklich als String-Literale mit doppelten oder einfachen Anführungszeichen gekennzeichnet werden und haben dann entsprechende Effekte.

```
n = 1
str1 = <<"EOF"
n ist #{n}
EOF
str2 = <<'EOF'
n ist #{n}
EOF
puts str1
puts str2
```

Ausgabe:

```
n ist 1
n ist #{n}
```

Soll der String noch verarbeitet werden, können nach dem Bezeichner noch Methoden aufgerufen werden.

```
n = 1
str1 = <<EOF.upcase
n ist #{n}
EOF

puts str1 #=> N IST 1
```

Heredocs können eingerückt werden, um dem programmatischen Kontext besser gerecht zu werden. Mit einem Minuszeichen - vor dem Bezeichner darf der Bezeichner, mit einer Tilde ~ das gesamte Heredoc eingerückt werden. Im letzten Falle wird von jeder Zeile der Weißraum entfernt, der der am wenigsten eingerückten Zeile entspricht. Im folgenden Beispiel sind die beiden erstellten Strings »str1« und »str2« deshalb identisch. Ohne den Modifikator - bzw. ~ hätte das jeweils terminierende »EOF« ohne führende Leerzeichen am Zeilenanfang stehen müssen.

```
def foo
  str1 = <<-EOF
Zeile 1.
Zeile 2.
  EOF

  str2 = <<~EOF
  Zeile 1.
  Zeile 2.
  EOF
  puts str1, str2
end
foo
```

Endlich können mehrere Heredocs gleichzeitig benutzt werden. Sie sind dann einfach hintereinander zu definieren, müssen aber unterschiedliche terminierende Bezeichner aufweisen.

```
puts(<<STRA, <<STRB)
String 1
```

```
STRA
String 2
STRB
```

11.1.4 Chars

Für einzelne Zeichen (engl. »chars«) besitzt Ruby keine eigene Klasse, allerdings gibt es aus historischen Gründen die Möglichkeit, ein einzelnes Zeichen mithilfe des Fragezeichen-Literals als String zu erhalten. Der praktische Nutzen dieser Funktionalität ist indessen beschränkt.

```
puts ?a #=> "a"
puts ?ä #=> "ä"
```

11.2 Zeichensätze

Das Thema Zeichensätze (engl. *charset* oder unpräziser *encoding*) gehört zu den schwierigeren Themen bei der Programmierung mit Ruby. Es handelt sich um die größte Neuerung, die mit Version 1.9 in die Programmiersprache eingeführt wurde und die für die meisten Inkompatibilitäten mit Programmen vor der Einführung verantwortlich sein dürfte. Ein rudimentäres Verständnis dieser Funktionalität ist für ein erfolgreiches Arbeiten mit der Programmiersprache Ruby schlechthin unabdingbar, da man sich rasch mit ansonsten unverständlichen Fehlermeldungen konfrontiert sieht, insbesondere dem unter Anfängern berüchtigten Encoding::CompatibilityError.

Für das Verständnis von Zeichenkodierung sind einige theoretische und terminologische Ausführungen unvermeidlich, da das Thema sonst nicht adäquat beschrieben werden kann (→ 11.2.1). Sodann soll die konkrete Umsetzung in Ruby beschrieben werden (→ 11.2.2).

11.2.1 Hintergrund

Bevor zu den technischen Einzelheiten der Zeichenspeicherung übergeleitet werden kann, muß die Vorfrage geklärt werden, *welche* Zeichen man eigentlich im Computer speichern möchte. Diese Menge an ausgewählten Zeichen, der sogenannte *Zeichensatz* (engl. *charset*), kann je nach Bedürfnis sehr unterschiedlich ausfallen. So ist denkbar, daß man sich auf die in Westeuropa üblichen lateinischen Zeichen nebst Zahlen und ein paar Sonderzeichen wie das

Euro-Zeichen beschränkt, daß die chinesischen Zeichen genügen oder daß man mit allen Zeichen der Erde umgehen können möchte. Der historisch bedeutsame frühe Zeichensatz für die U. S. A., *ASCII*[1], umfaßt 128 Zeichen; der heute maßgebliche Weltzeichensatz *Unicode* in der derzeit aktuellen Version 11 umfaßt 137.374 Zeichen.

Zeichen, wie sie von Menschen zu Kommunikation eingesetzt werden, sind für Computer allerdings erst einmal nicht verständlich. Im (Arbeits-)Speicher werden lediglich Bytes abgelegt. Ein Byte besteht aus acht Bit, also Werten, die nur Null oder Eins sein können. Ein Byte bietet damit $2^8 = 256$ verschiedene Zustände, die zur Speicherung von Informationen genutzt werden können.

Aufgabe der *Zeichenkodierung* (*encoding*[2]) ist es nun, jedem Zeichen im gewählten Zeichensatz eine bestimmte Anordnung dieser Bits zuzuweisen. Können alle Zeichen eines Zeichensatzes mit den acht Bit eines Bytes abgebildet werden, so spricht man von einem *Single-Byte-Encoding*, anderenfalls von einem *Multi-Byte-Encoding*. Weil 256 Möglichkeiten für die mittlerweile über 100.000 Zeichen, die Unicode definiert, bei weitem nicht ausreichen, sind alle modernen Zeichenkodierungen Multi-Byte-Encodings.

Das bedeutsamste Single-Byte-Encoding ist allerdings das mit dem korrespondierenden Zeichensatz gleichnamige ASCII (formal genauer: ANSI X3.4-1986) von 1963, das 1986 zuletzt geändert wurde. Hierbei handelt es sich wegen der hohen weltweiten Verbreitung, die auf die damalige dominierende Rolle der Vereinigten Staaten von Amerika im Informationstechnologiesektor zurückgeht, um eine Art Mindeststandard. Fast alle Systeme, auch alte, unterstützen wenigstens diese Zeichenkodierung. ASCII spezifiziert 128 Zeichen nebst ihren Kodierungen, indem es die ersten 7 Bit eines Bytes nutzt. Es verbleiben damit 128 weitere vom ASCII-Standard freigelassene und das 8. Bit ausnutzende Speichermöglichkeiten in einem Byte. In Europa hat man diese freien Speichermöglichkeiten genutzt, um jeweils länderspezifische Zeichen zu kodieren. Daraus entstanden die Zeichenkodierungen der ISO-8859-Gruppe, darunter insbesondere ISO-8859-1 für Westeuropa, das die freien Plätze für Umlaute, Eurozeichen und ähnliches nutzt. Diese Zeichenkodierung hielt um die Jahrtausendwende herum Einzug insbesondere in Microsoft Windows und ist dort bis heute die Standardkodierung. Dies ist vor dem Hintergrund neuerer moderner Kodierungen (dazu sogleich) ein Anachronismus, der Entwicklern heute viel unnötige Arbeit verursacht.

[1] American Standard Code for Information Interchange.

[2] Der Begriff ist mehrdeutig. In anderen Kontexten kann er eine von der hier beschriebenen Funktion abweichende Bedeutung haben; z. B. spricht man auch davon, daß Base64, ein Standard zur Kodierung von E-Mails, ein Encoding sei.

Die Arbeit mit den 8-Bit-Kodierungen war im asiatischen Raum schon immer schwierig (dort entstanden vor dem Hintergrund der dortigen sehr großen Zeichensätze gänzlich andere Zeichenkodierungen wie Big5 und Shift-JIS), führte im Zuge der fortschreitenden Globalisierung aber auch in Europa und den Vereinigten Staaten von Amerika zu Problemen. Nachrichten, die etwa nach Polen oder Rußland verschickt wurden, mußten in einer dort gebräuchlichen Zeichenkodierung verfaßt werden, die mit der lokal genutzten inkompatibel war. Dieses Problem löst nun Unicode und die ihm zugeordneten Zeichenkodierungen der UTF-Gruppe. Von diesen Kodierungen sind drei gebräuchlich:

- UTF-8 hat sich als Standard in praktisch allen Belangen außer Windows-Desktop-Anwendungen durchgesetzt.
- UTF-16 ist der speziell von Microsoft Windows propagierte Standard.
- UTF-32 wird bisher eher selten genutzt, hat aber Zukunftspotential.

Alle drei Zeichenkodierungen bilden den gesamten Unicode-Zeichensatz ab, allerdings in unterschiedlicher Weise. Allen gemeinsam ist aber, daß es sich um Multi-Byte-Encodings handelt, da die Speichermöglichkeiten eines einzelnen Byte für die große Menge an Unicode-Zeichen einfach nicht ausreichen.

UTF-8 hat den praktischen Vorteil, daß die ersten 128 Zeichen identisch wie bei ASCII kodiert werden und es damit rückwärtskompatibel zu reinem ASCII-Text ist. Erst die weiteren Zeichen werden durch Einsatz mehrerer Bytes kodiert, sodaß es sich bei UTF-8 um eine Zeichenkodierung mit variabler Länge handelt. UTF-16 verwendet dagegen für alle Zeichen mindestens zwei Byte. Die meistgebrauchten Zeichen aus dem Unicode-Standard (»Basic Multilingual Plane«, BMP) passen auch tatsächlich in genau zwei Byte, sodaß es sich, solange die Anwendung sich auf diese Untermenge von Zeichen beschränkt, um eine Zeichenkodierung mit vorhersehbaren Speicherbedarf handelt. Allerdings ist das BMP oft nicht ausreichend, sodaß es zur Nutzung von mehr Bytes kommt. Erst UTF-32 mit seinen vier Byte gewährleistet einen einheitlichen Speicherbedarf für alle Zeichen, der allerdings entsprechend erheblich ist.

Unicode definiert strenggenommen keine Zeichen, sondern *Codepoints*. Der Unterschied besteht darin, daß manche Zeichen aus mehreren Codepoints zusammengesetzt werden können. Codepoints werden für die allgemeine Diskussion in einer besonderen Schreibweise angegeben, die auf den Unicode-

Standard Bezug nimmt. So bedeutet U+0041 den Codepoint mit dem Namen »LATIN CAPITAL LETTER A«, der für den Buchstaben A steht. U+00DF steht für »LATIN SMALL LETTER SHARP S«, das deutsche scharfe ß. U+221A normiert das mathematische Wurzelzeichen, »SQUARE ROOT«, √. In dieser Notation steht »U« für Unicode, das +-Zeichen ist nur ein Trenner, und danach folgt die Position im Unicode-Standard als Hexadezimalzahl, aufgefüllt auf vier Zeichen mit führenden Nullen.

Die Möglichkeit, ein Zeichen aus mehreren Codepoints zusammenzusetzen, betrifft insbesondere akzentuierte Zeichen. So kann Beispielsweise das Zeichen »ä« aus dem Codepunkt für »a« (U+0061) und dem Kombinationszeichen für die Diäräse »¨« (U+0308) zusammengesetzt werden. Einige Zeichen stehen allerdings sowohl als einzelner Codepoint wie auch als Kombination mehrerer Codepoints zur Verfügung. »ä« ist etwa auch als einzelner Codepoint U+00E4 definiert. Muß man Unicode-Strings miteinander vergleichen, müssen sie daher *normalisiert* werden, d. h. in eine definierte Form gebracht werden, damit die für den Nutzer gleich aussehenden Texte auch programmatisch als gleich und nicht wegen des Unterschieds zwischen U+0061/U+0308 einerseits und U+00E4 andererseits als unterschiedlich gewertet werden. Während die meisten Betriebssysteme wohl wenn möglich die einfachen Codepoints vorziehen, präferiert speziell MacOS die kombinierte Form. Hierbei handelt es sich um ein beliebtes Fettnäpfchen in Unicode-Anwendungen.

Das, was der Nutzer am Ende zu sehen bekommt, ist im Übrigen nicht einmal ein Codepoint. *Schriftarten* (engl. *Fonts*) entscheiden, wie die visuelle Gestaltung eines Codepoints aussieht und ordnen jedem Codepoint eine *Glyphe* zu. Weil Unicode sehr umfangreich ist, setzen nur wenige Schriftarten wirklich alle in Unicode spezifierten Codepoints um. Unterstützt die vom Nutzer eingesetzte Schriftart die verwandten Codepoints nicht, sieht der Nutzer ein Ersatzzeichen — oft eine weiße Box mit dünnem Rahmen und der Codepoint-Nummer (beliebt bei Browsern) oder das Unicode-Ersatzzeichen U+FFFD (ein eingekästeltes Fragezeichen).

11.2.2 Umsetzung in Ruby

Das von Ruby eingesetzte System zum Umgang mit Zeichenkodierungen ist ein Spezifikum der Programmiersprache. Jeder String in Ruby umfaßt sowohl die eigentlichen Bytes wie auch die Zeichenkodierung. Das Encoding kann mithilfe von #encoding abgerufen werden. Mithilfe von #force_encoding wird die Kodierungsmarkierung geändert, *ohne* daß die eigentlichen Bytes geändert werden. So können mit einer falschen Encoding-Markierung eingelesene

Strings korrigiert werden. Die *Bytes* ändert man durch Transkodierung mithilfe von #encode, das auch als den Empfänger ändernde Methode #encode! zur Verfügung steht. #valid_encoding schließlich prüft, ob die Bytes des Strings für die Kodierungsmarkierung Sinn ergeben. Das folgende Beispiel verdeutlicht den Zusammenhang der Methoden:

```
# Ersterstellung erfolgt mit UTF-8
str = "Bär"
puts str.encoding         #=> UTF-8

# Transkodierung auf ISO-8859-1
str.encode!("ISO-8859-1")
puts str.encoding         #=> ISO-8859-1
puts str.valid_encoding?  #=> true

# Austausch der Kodierungs-Markierung
str.force_encoding("UTF-8")
puts str.valid_encoding?  #=> false
```

Besonderes Augenmerk sollte auf das Ergebnis der letzten Zeile gelegt werden. #force_encoding ändert die Kodierungs-Markierung, läßt die in `str` enthaltenen Bytse jedoch unberührt. Deshalb ist das Ergebnis ein String, dessen Bytes UTF-8 notieren, der aber fälschlicherweise mit ISO-8859-1 markiert ist. Diese Situation sollte nach Mögichkeit vermieden werden, da sie zu verwirrenden Fehlern führen kann.

In diesen Zusammenhang gehört auch der folgende ungeliebte Fehler. Liest man einen String, der Bytes in UTF-8-Notation enthält, fehlerhafterweise als ISO-8859-1 ein (= falsche Kodierungsmarkierung), kann es irritierenderweise dazu kommen, daß der resultierende String auch gültiges ISO-8859-1 ist, aber dort etwas ganz anderes bedeutet. Ein solcher String übersteht eine Prüfung mit #valid_encoding?. Transkodiert man den fehlerhaft markierten String anschließend nach UTF-8 entsteht ein Fehler, den wohl jeder schon einmal auf einer Webseite gesehen hat:

```
str = "Bär" # Erstellung als UTF-8
puts str.encoding #=> UTF-8

# Fehlerhafte Markierung als ISO-8859-1
str.force_encoding("ISO-8859-1")
```

```
puts str.valid_encoding? #=> true # Ups!

# Transkodierung des vermeindlichen ISO-8859-1
# nach UTF-8
str.encode("UTF-8")
puts str #=> "BÃ¤r"
```

Statt dem erwarteten „Bär“ kommt „BÃ¤r“ heraus. Der Zeichensalat ergibt keinen Sinn, ist aber gültiges UTF-8 (bei ¤ handelt es sich um ein Platzhaltersymbol für Währungen). Wer mit dieser Art von Fehler konfrontiert ist, wird sich auf eine oft längliche Suche danach machen müssen, an welcher Stelle ein String mit der falschen Zeichenkodierung markiert wurde. Ruby versucht, diese Art Fehler möglichst frühzeitig zu einem Laufzeitfehler aufzuwerten, indem der Versuch, Strings mit unterschiedlichen Kodierungs-Markierungen zu kombinieren, einen Encoding::CompatibilityError auslöst:

```
str1 = "Bär".encode("ISO-8859-1")
str2 = "Straße" # UTF-8
str1 + str2
#=> Encoding::CompatibilityError: incompatible
#=> character encodings: ISO-8859-1 and UTF-8
```

Wie sich schon aus den Ausführungen zum theoretischen Hintergrund ergibt (→ 11.2.1) bilden nicht alle Zeichenkodierungen alle Zeichensätze ab. Das Zeichen »ä« ist etwa nicht im vom asiatischen Big5 abgebildeten Zeichensatz enthalten. Der Versuch, es dorthin zu transkodieren, muß scheitern.

```
str = "Bär" # UTF-8
str.encode("Big5")
#=> Encoding::UndefinedConversionError:
#=> U+00E4 from UTF-8 to Big5
```

Im Hinblick auf den historisch bedeutsamen ASCII-Zeichensatz und seine Kodierung muß unterschieden werden. Das ursprüngliche ASCII spezifiziert wie bereits erwähnt nur die Bedeutung der ersten sieben Bit eines Bytes. Diese Zeichenkodierung kennt Ruby als US-ASCII. In einem als US-ASCII markierten String sind deshalb nur Bytes im Bereich 0-127 (letztes Bit nicht gesetzt) zulässig. Darüber hinaus gibt es aber eine Pseudo-Zeichenkodierung namens ASCII-8BIT für die Arbeit mit Binärdaten, da Ruby keine eigene Klasse zur Arbeit mit binären Daten besitzt. ASCII-8BIT verhält sich für alle Bytes mit

Kodierung	Feststellung	Standard
Quelltext-Kodierung	`__ENCODING__`	UTF-8
Umgebungs-Kodierung	`Encoding.locale_charmap`	Systemabhängig
Dateisystem-Kodierung	`Encoding.find("filesystem")`	Systemabhängig
Externe Kodierung	`Encoding.default_external`	Umgebungs-Kodg.
Interne Kodierung	`Encoding.default_internal`	nil

Tabelle 11.2: Zeichenkodierungen

ungesetztem letzten Bit wie US-ASCII, erlaubt aber auch Bytes mit gesetztem letzten Bit, ohne diesen eine Bedeutung beizulegen.

```
str1 = "\226" # Außerhalb von ASCII
str2 = "\226"
str1.force_encoding("US-ASCII")
str2.force_encoding("ASCII-8BIT")
puts str1.valid_encoding? #=> false
puts str2.valid_encoding? #=> true
```

ASCII-8BIT besitzt einen gleichwertigen Alias namens BINARY, dem der Vorzug eingeräumt werden sollte, um Mißverständnissen von vornherein aus dem Weg zu gehen. Geht es um Text in ASCII-Kodierung, sollte man als Kodierung US-ASCII wählen, geht es um Binärdaten, dann sollte man BINARY wählen.

Wie aber kommt nun die Kodierungs-Markierung zum String? Dafür existieren viele Möglichkeiten. Zunächst einmal kann wie schon beschrieben per String#force_encoding die Kodierungs-Markierung ausdrücklich gesetzt werden. Ruby versieht aber schon von Haus aus Strings aus diversen Quellen mit ebenso diversen Markierungen. Eine Übersicht verschafft Tafel 11.2, die freilich noch einiger Erläuterung bedarf.

Da ist zunächst die *Quelltext-Kodierung*. Dies ist die Zeichenkodierung, in der das Ruby-Programm selbst geschrieben ist. Diese Zeichenkodierung ist standardmäßig UTF-8, worauf der eigene Editor bei der Arbeit dementsprechend eingestellt werden sollte. Es ist aber möglich, mithilfe eines besonderen „magischen“ Kommentars die Quelltext-Kodierung zu ändern. Der Kommentar selbst muß in ASCII kodiert sein, die erste Zeile eines Skripts — bei Verwendung eines Shebangs (→ 12.6) kann es auch die zweite sein — darstellen und die Zeichenfolge »coding:« gefolgt von einem Abstand und dem Namen der gewünschten Zeichenkodierung ohne Rücksicht auf Groß- oder

Kleinschreibung enthalten. Weitere Zeichen davor oder danach werden ignoriert. Beispiele für solche Kommentare:

```
# coding: iso-8859-1
# Encoding: US-ASCII
# -*- coding: utf-8 -*-
```

Mit den *Umgebungs-* und *Dateisystem-Kodierungen* hat man in Ruby eher selten zu tun. Diese hängen von den Systemeinstellungen des Nutzers ab und geben an, was dessen präferierte Zeichenkodierung für Kommandozeilenprogramme und die Kodierung von Dateipfaden auf der Festplatte ist. Ersteres wird in Ruby über die externe Kodierung (dazu sogleich) gelöst, letzteres erfolgt sogar vollautomatisch. **[TODO: PRÜFEN]**. Eine Rolle können die Dateisystem- und Umgebungs-Kodierungen etwa bei den Werten im ENV-Objekt (→ 12.2) spielen.

Die *externe* und *interne* Zeichenkodierung sollen die Arbeit für den Programmierer vereinfachen, wenn es um die Interaktion mit der Umgebung außerhalb des Programms geht. Der Sache nach handelt es sich um Besonderheiten bei der Ein- und Ausgabe, allerdings gebietet der Zusammenhang eine Erklärung an dieser Stelle statt im entsprechenden Kapitel **[TODO: Querverweis]**, auf das für den Grundlagen zu IO verwiesen wird. Jedes IO-Objekt in Ruby besitzt sowohl eine externe als auch eine interne Zeichenkodierung. Die externe Zeichenkodierung gibt an, in welcher Kodierung die Daten in der Umgebung außerhalb des Programms (also etwa in einer Datei) vorliegen. Mangels besonderer Angabe beim Öffnen des IO-Objekts wird diese Zeichenkodierung auf `Encoding.default_external` gesetzt, dessen Standardwert seinerseits von der Programmumgebung abhängt. Das externe Encoding dient damit der richtigen Markierung der eingelesenen Strings mit der korrekten Zeichenkodierung (vgl. zu diesem Problem schon oben). Darauf baut nun in einem zweiten Schritt das interne Encoding auf: Ruby transkodiert die eingelesenen Daten automatisch von der externen in die interne Zeichenkodierung, bevor das Ruby-Programm die Daten überhaupt zu Gesicht bekommt. Das interne Encoding eines IO-Objekts ist bei fehlender Angabe `Encoding.default_internal`, und dieses ist standardmäßig nil, d. h. standardmäßig findet keine Transkodierung statt. Der gesamte Prozeß funktioniert auch in umgekehrter Richtung, wobei allerdings an die Stelle des internen Encodings die Kodierungs-Markierung des auszugebenden Strings tritt. Unterscheidet diese sich vom externen Encoding des Ziel-IO-Objektes, so transkodiert Ruby bei der Ausgabe automatisch. Abbildung 11.1 verdeut-

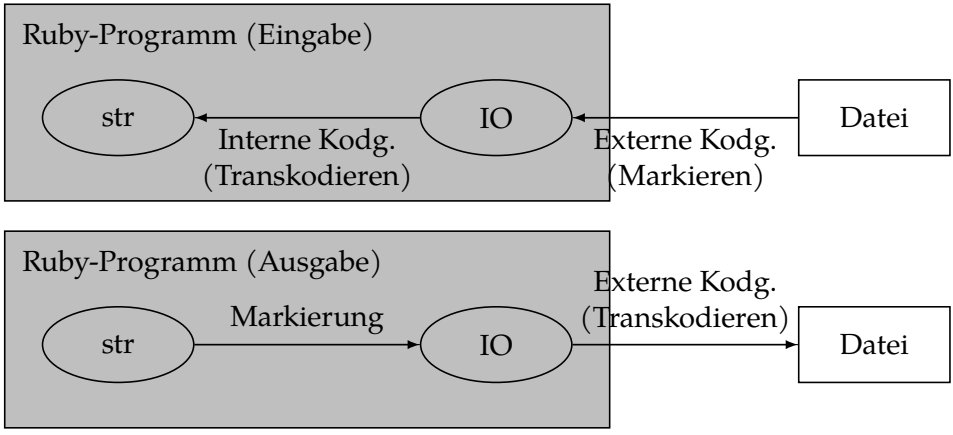

Abbildung 11.1: Externe und interne Kodierung

licht das Konzept, Tafel 11.3 listet alle von Ruby unterstützen Zeichenkodierungen auf.

Die interne und die externe Standardkodierung können auch ausdrücklich gesetzt werden, allerdings rät die Ruby-Dokumentation davon ab und verweist stattdessen auf den Kommandozeilenschalter -E des Ruby-Interpreters, den der Nutzer setzen solle. Als Argument wird angeführt, daß Strings vor und nach dem Setzen dieser Werte andere Kodierungs-Markierungen erhalten, was zu schwierig zu entdeckenden Fehlern führen könne. Die Empfehlung erweist sich nach hiesiger Auffassung nur als teilweise richtig, noch dazu aus einem anderen Grund. Zunächst greift das vorgebrachte Argument mit den Kodierungs-Markierungen bei richtiger Anwendung der Methoden nicht durch, denn wenn die Änderung der Standardkodierungen direkt und einmalig bei Programmstart vorgenommen wird, werden die befürchteten Überraschungseffekte nicht auftreten. Auch ist die Verwendung des -E-Schalters vergleichsweise kompliziert umzusetzen, da der Nutzer entsprechend instruiert oder ein Wrapper-Skript geschrieben werden muß. Speziell im Hinblick auf die interne Standardkodierung ist die Empfehlung der Nutzung von -E außerdem irreführend: sie suggeriert, daß der Nutzer sich die interne Kodierung aussuchen könne. Das ist aber unzutreffend, denn dafür müßte der Nutzer Kenntnis über die Programminterna haben, widrigenfalls er eine möglicherweise unzureichende Zeichenkodierung wie US-ASCII festlegt. Dem manuellen Setzen der internen Standardkodierung bei Programmstart steht deshalb nichts entgegen. Anders liegt der Fall für die externe Standardkodierung.

Der Programmierer hat in der Regel keine gesicherte Kenntnis vom System des Nutzers, da sein Programm portabel zwischen verschiedenen Systemen und Nutzern sein soll. Der Nutzer dagegen kennt sein System oder sollte es zumindest kennen. Es ist daher nur sachgerecht, dem Nutzer die Entscheidung darüber aufzuerlegen, anzugeben, welche Zeichenkodierung in seiner Umgebung aktiv ist. Von einem manuellen Setzen sollte daher abgesehen werden. Glücklicherweise ist Ruby meistens von sich aus in der Lage, durch Analyse der Programmumgebung — etwa durch Auswertung der Umgebungsvariable »LANG« — die externe Standardkodierung bereits ohne Nutzerintervention richtig zu erkennen. Wenn diese Erkennung im Einzelfall einmal versagen sollte, ist ein Rückgriff durch den Nutzer auf den Kommandozeilenschalter -E richtig und empfehlenswert.

Bei einzelnen IO-Operationen kann die gewünschte externe und interne Zeichenkodierung auch abweichend von den Standardwerten festgelegt werden. Der praktisch wichtigste Fall ist das Öffnen einer Datei. Dazu wird das Modus-Argument um die Angabe der externen und optional der internen Zeichenkodierung durch Doppelpunkte erweitert:

```
File.open("datei.txt", "r:ISO-8859-1:UTF-8")
```

Hier folgt auf den Modus (»r«, also lesend) erst die externe Zeichenkodierung (die Datei enthält also Text mit der Kodierung ISO-8859-1) und dann die interne Zeichenkodierung (beim Auslesen wird also automatisch nach UTF-8 transkodiert). Wird die interne Zeichenkodierung weggelassen, gilt Encoding::default_internal, was wie bereits beschrieben standardmäßig nil ist, sodaß keine Transkodierung stattfindet. Einzelheiten gehören in das Kapitel über die Ein- und Ausgabe.

[**TODO: Querverweis**]

Tabelle 11.3: Verfügbare Zeichenkodierungen

Name	Alias für	ASCII*	Bemerkungen
646	US-ASCII	Ja	
ANSI_X3.4-1968	US-ASCII	Ja	
ASCII	US-ASCII	Ja	
ASCII-8BIT	—	Ja	Binär-»Encoding«
Big5	—	Ja	
Big5-HKSCS	—	Ja	
Big5-HKSCS:2008	Big5-HKSCS	Ja	
Big5-UAO	—	Ja	

*ASCII-kompatible Kodierung?

Fortsetzung nächste Seite

Name	Alias für	ASCII*	Bemerkungen
BINARY	ASCII-8BIT	Ja	Binär-»Encoding«
CP1250	Windows-1250	Ja	
CP1251	Windows-1251	Ja	
CP1252	Windows-1252	Ja	
CP1253	Windows-1253	Ja	
CP1254	Windows-1254	Ja	
CP1255	Windows-1255	Ja	
CP1256	Windows-1256	Ja	
CP1257	Windows-1257	Ja	
CP1258	Windows-1258	Ja	
CP437	IBM437	Ja	
CP50220	—	Nein	
CP50221	—	Nein	
CP51932	—	Ja	
CP65000	UTF-7	Nein	
CP65001	UTF-8	Ja	
CP737	IBM737	Ja	
CP775	IBM775	Ja	
CP850	—	Ja	
CP852	—	Ja	
CP855	—	Ja	
CP857	IBM857	Ja	
CP860	IBM860	Ja	
CP861	IBM861	Ja	
CP862	IBM862	Ja	
CP863	IBM863	Ja	
CP864	IBM864	Ja	
CP865	IBM865	Ja	
CP866	IBM866	Ja	
CP869	IBM869	Ja	
CP874	Windows-874	Ja	
CP878	KOI8-R	Ja	
CP932	Windows-31J	Ja	
CP936	GBK	Ja	
CP949	—	Ja	
CP950	—	Ja	
CP951	—	Ja	
csWindows31J	Windows-31J	Ja	
ebcdic-cp-us	IBM037	Nein	
Emacs-Mule	—	Ja	
EUC-CN	GB2312	Ja	
EUC-JIS-2004	—	Ja	

*ASCII-kompatible Kodierung?

Fortsetzung nächste Seite

Name	Alias für	ASCII*	Bemerkungen
EUC-JISX0213	EUC-JIS-2004	Ja	
EUC-JP	—	Ja	
euc-jp-ms	eucJP-ms	Ja	
EUC-KR	—	Ja	
EUC-TW	—	Ja	
eucCN	GB2312	Ja	
eucJP	EUC-JP	Ja	
eucJP-ms	—	Ja	
eucKR	EUC-KR	Ja	
eucTW	EUC-TW	Ja	
external	—	—	Externes Encoding
filesystem	—	—	Systemabhängig
GB12345	—	Ja	
GB18030	—	Ja	
GB1988	—	Ja	
GB2312	—	Ja	
GBK	—	Ja	
IBM037	—	Nein	
IBM437	—	Ja	
IBM737	—	Ja	
IBM775	—	Ja	
IBM850	CP850	Ja	
IBM852	—	Ja	
IBM855	—	Ja	
IBM857	—	Ja	
IBM860	—	Ja	
IBM861	—	Ja	
IBM862	—	Ja	
IBM863	—	Ja	
IBM864	—	Ja	
IBM865	—	Ja	
IBM866	—	Ja	
IBM869	—	Ja	
internal	—	—	Internes Encoding
ISO-2022-JP	—	Nein	
ISO-2022-JP-2	—	Nein	
ISO-2022-JP-KDDI	—	Nein	
ISO-8859-1	—	Ja	
ISO-8859-10	—	Ja	
ISO-8859-11	—	Ja	
ISO-8859-13	—	Ja	
ISO-8859-14	—	Ja	

*ASCII-kompatible Kodierung?

Fortsetzung nächste Seite

Name	Alias für	ASCII*	Bemerkungen
ISO-8859-15	—	Ja	
ISO-8859-16	—	Ja	
ISO-8859-2	—	Ja	
ISO-8859-3	—	Ja	
ISO-8859-4	—	Ja	
ISO-8859-5	—	Ja	
ISO-8859-6	—	Ja	
ISO-8859-7	—	Ja	
ISO-8859-8	—	Ja	
ISO-8859-9	—	Ja	
ISO2022-JP	ISO-2022-JP	Nein	
ISO2022-JP2	ISO-2022-JP-2	Nein	
ISO8859-1	ISO-8859-1	Ja	
ISO8859-10	ISO-8859-10	Ja	
ISO8859-11	ISO-8859-11	Ja	
ISO8859-13	ISO-8859-13	Ja	
ISO8859-14	ISO-8859-14	Ja	
ISO8859-15	ISO-8859-15	Ja	
ISO8859-16	ISO-8859-16	Ja	
ISO8859-2	ISO-8859-2	Ja	
ISO8859-3	ISO-8859-3	Ja	
ISO8859-4	ISO-8859-4	Ja	
ISO8859-5	ISO-8859-5	Ja	
ISO8859-6	ISO-8859-6	Ja	
ISO8859-7	ISO-8859-7	Ja	
ISO8859-8	ISO-8859-8	Ja	
ISO8859-9	ISO-8859-9	Ja	
KOI8-R	—	Ja	
KOI8-U	—	Ja	
locale	—	—	Systemabhängig
macCentEuro	—	Ja	
macCroatian	—	Ja	
macCyrillic	—	Ja	
macGreek	—	Ja	
macIceland	—	Ja	
MacJapan	MacJapanese	Ja	
MacJapanese	—	Ja	
macRoman	—	Ja	
macRomania	—	Ja	
macThai	—	Ja	
macTurkish	—	Ja	
macUkraine	—	Ja	

*ASCII-kompatible Kodierung?

Fortsetzung nächste Seite

Name	Alias für	ASCII*	Bemerkungen
PCK	Windows-31J	Ja	
Shift_JIS	—	Ja	
SJIS	Windows-31J	Ja	
SJIS-DoCoMo	—	Ja	
SJIS-KDDI	—	Ja	
SJIS-SoftBank	—	Ja	
stateless-ISO-2022-JP	—	Ja	
stateless-ISO-2022-JP-KDDI	—	Ja	
TIS-620	—	Ja	
UCS-2BE	UTF-16BE	Nein	
UCS-4BE	UTF-32BE	Nein	
UCS-4LE	UTF-32LE	Nein	
US-ASCII	—	Ja	Echtes 7-Bit-ASCII.
UTF-16	—	Nein	
UTF-16BE	—	Nein	
UTF-16LE	—	Nein	
UTF-32	—	Nein	
UTF-32BE	—	Nein	
UTF-32LE	—	Nein	
UTF-7	—	Nein	
UTF-8	—	Ja	
UTF-8-HFS	UTF8-MAC	Ja	
UTF-8-MAC	UTF8-MAC	Ja	
UTF8-DoCoMo	—	Ja	
UTF8-KDDI	—	Ja	
UTF8-MAC	—	Ja	
UTF8-SoftBank	—	Ja	
Windows-1250	—	Ja	Die Windows-
Windows-1251	—	Ja	Encodings
Windows-1252	—	Ja	sind mit
Windows-1253	—	Ja	den ISO-
Windows-1254	—	Ja	Encodings
Windows-1255	—	Ja	weitest-
Windows-1256	—	Ja	gehend
Windows-1257	—	Ja	identisch.
Windows-1258	—	Ja	
Windows-31J	—	Ja	
Windows-874	—	Ja	

*ASCII-kompatible Kodierung?

11.3 Wichtige Methoden

Ruby besitzt umfangreiche Möglichkeiten, um Zeichenketten zu verarbeiten. Hier soll ein kurzer Überblick genügen; für umfassende Informationen kann die Referenz der Programmiersprache Ruby herangezogen werden[3]

11.3.1 Größe

Strings verfügen über eine bestimmte Länge, die in Zeichen angegeben wird. Diese kann mit der Methode #length oder ihrem Alias #size abgerufen werden. Die von der genutzten Zeichenkodierung abhängige, genau belegte Anzahl an Bytes im String ist dagegen Gegenstand der Methode #bytesize.

```
puts "Bär".length   #=> 3
puts "Bär".bytesize #=> 4
```

11.3.2 Methoden zum Umgang mit Unicode

Die für Rubys Zeichenkodierungssystem wesentlichen Methoden #encode, #force_encoding und #valid_encoding? wurden bereits oben unter → 11.2.2 vorgestellt. Sie transkodieren einen String in eine andere Zeichenkodierung (#encode), ändern die Kodierungsmarkierung (#force_encoding) und prüfen einen String auf seine Gültigkeit in Bezug auf die Zeichenkodierung, mit der er markiert ist (#valid_encoding?). Darüber hinaus bietet Rubys String-Klasse aber noch einige Methoden mehr.

#ascii_only? überprüft die im String enthaltenen Bytes darauf, ob sie ausschließlich im Bereich des ASCII-Zeichensatzes liegen und ignoriert dabei die Kodierungsmarkierung. Die Methode mit dem etwas kryptischen Namen #b, der wohl vornehmlich dem schnellen Tippen dienen soll, kopiert den Empfänger und markiert die sodann zurückgegebene Kopie mit der Pseudo-Zeichenkodierung BINARY. Das kann praktisch sein, um auf einfache Weise bereits mit einer Zeichenkodierung versehene Strings in Binärdaten »umzuwandeln«. Am String selbst ändert sich dabei selbstverständlich nichts.

```
puts "abc".ascii_only? #=> true
puts "Bär".ascii_only? #=> false
puts "Bär".b.encoding  #=> ASCII-8BIT (= BINARY)
```

[3] Siehe `https://ruby-doc.org/core/String.html`.

Das oben (→ 11.2.2) erwähnte Fettnäpfchen im Bezug auf Zeichen, die bei gleichem Aussehen aus unterschiedlichen Unicode-Codepoints zusammengesetzt sind, kann und sollte man in Ruby unter Verwendung der Methode #unicode_normalize lösen. Hierbei sollte man es sich zur Gewohnheit machen, Daten, die nicht-binäre Nutzereingaben enthalten, unbedingt sogleich zu normalisieren, um sich innerhalb des Programms stets auf einen definierten Zustand verlassen zu können.

Der Unicode-Standard definiert vier Normalisierungsalgorithmen, NFC, NFD, NFKC und NFKD, die die verschiedenen Möglichkeiten der Normalisierung von Unicode-Strings abbilden. Dabei sind zwei Fragem zu unterscheiden:

1. Soll für Kombinationszeichen die Variante mit einem einzelnen Codepoint (Komposition) oder die Variante mit mehreren Kombinations-Codepoints (Dekomposition) bevorzugt werden?

2. Sollen Zeichengruppen und Spezialzeichen in »normale« Zeichen umgewandelt werden (Kompatibilität mit Nutzereingaben) oder nicht (keine Kompatibilität)?

Während die erste Frage sich damit beschäftigt, ob Zeichen wie »ä« als einzelner Codepoint U+00E4 (Komposition) oder als zwei separate Codepoints U+0061/U+0308 (Dekomposition) dargestellt werden sollen, betrifft die zweite Frage spezielle Zeichen wie etwa Ligaturen. Die Ligatur »ff« verfügt über den eigenen Codepoint U+FB00. Das ist mißlich, denn semantisch handelt es sich ja nach wie vor um zwei separate Buchstaben »f«, die nur aus optischen Gründen besonders dargestellt werden. Etwas ähnliches kann man auch von den römischen Zahlzeichen sagen, die auch über eigene Codepoints verfügen (etwa U+2167 für die römische Zahl VIII für 8). Werden solche Spezialzeichen zerlegt, werden sie »kompatibel« zu gewöhnlichen Zeichen und damit durchsuchbar; wer nach »f« sucht, wird auch in der Ligatur »ff« fündig.

Die Normalisierungen NFC und NFD beantworten die zweite Frage beide mit »keine Kompatibilität«, d. h. fassen die entsprechenden Codepoints nicht an. Das ist im Zweifel die richtige Variante, denn während die (De-)Komposition ohne semantischen Bedeutungsverlust durch Anwendung des jeweils anderen Normalisierungs-Algorithmus rückgängig gemacht werden kann[4],

[4] Allerdings ist nicht garantiert, daß genau derselbe Codepoint genutzt wird wie in der Ausgangszeichenkette. Es kann auch ein anderer, aber semantisch gleichwertiger Codepoint herauskommen.

	Komposition	Kompatibilität
NFD	Dekomposition	Unverändert
NFC	Komposition	Unverändert
NFKD	Dekomposition	Zerlegen
NFKC	Komposition	Zerlegen

Tabelle 11.4: Unicode-Normalformen

ist das Aufbrechen der erwähnten Spezialzeichen für eine Kompatibilität mit den Algorithmen NFKD und NFKC destruktiv. Ist die römische Zahl VIII einmal in die Codepoint-Folge V+I+I+I zerlegt, kann sie nicht mehr zurückverwandelt werden.

Durch alle denkbaren Kombinationen der Antworten auf diese beiden Fragen entstehen die $2^2 = 4$ Normalformen, die in Tafel 11.4 zusammengefaßt sind.

Rubys Methode String#unicode_normalize kann mit allen Normalisierungsalgorithmen umgehen. Wird keiner angegeben, wird standardmäßig NFD angewandt.

```
str = "B\u00e4\uFB00" # Bäff mit Ligatur
p str.codepoints #=> [66, 228, 64256]

# Mit ff-Ligatur, mit Einzel-Codepoint ä
str2 = str.unicode_normalize
p str2.codepoints #=> [66, 228, 64256]

# Mit ff-Ligatur, mit Multi-Codepoint ä
str3 = str.unicode_normalize(:nfd)
p str3.codepoints #=> [66, 97, 776, 64256]

# Ohne ff-Ligatur, mit Einzel-Codepoint ä
str4 = str.unicode_normalize(:nfkc)
p str4.codepoints #=> [66, 228, 102, 102]
p str4            #=> "Bäff"

# Ohne ff-Ligatur, mit Multi-Codepoint ä
str5 = str.unicode_normalize(:nfkd)
p str5.codepoints #=> [66, 97, 776, 102, 102]
```

```
p str5              #=> "Bäff"
```

11.4 Symbole

Ein Symbol ist ein programmweit eindeutiger Name und eine Instanz der Klasse Symbol. Anfänger tun sich mit dem Verständnis von Symbolen traditionell schwer. Häufig werden Sie mit Variablen verwechselt oder mit Strings. Ersteres ist falsch, letzteres nur zum Teil berechtigt. Eine Variable hält einen Wert, ein Symbol *ist* der Wert. Verwirrend ist, daß man in der Metaprogrammierung Symbole benützt, um auf Variablen zuzugreifen ([**TODO: Querverweis**]). Doch darf das nicht darüber hinwegtäuschen, daß Symbole selbst — anders als Variablen — Objekte sind. Richtig ist aber, daß Symbole in der Handhabung den Strings ähnlich sind[5]. So unterstützt die Klasse Symbol etwa auch Manipulationsmethoden wie #upcase. Deren Einsatz ist aber nicht notwendigerweise guter Stil; als Anfänger sollte man davon Abstand halten.

Symbole werden über ein eigenes Literal definiert, das mit einem Doppelpunkt beginnt. Leer- und Sonderzeichen können mit einer an String-Literale angelehnten Syntax verwandt werden. Escape-Sequenzen verhalten sich in diesem Fall wie bei Strings.

```
sym1 = :mysymbol
sym2 = :"symbol with\ttab"
sym3 = :'symbol with space'
```

Strings können in Symbole, Symbole in Strings umgewandelt werden.

```
p :mysmbol.to_s      #=> "mysmbol"
p "mysymbol".to_sym  #=> :mysymbol
```

Die Methode String#to_sym war lange Jahre ein oft unterschätztes Speicherleck für Ruby-Programme, denn Symbole wurden nicht vom Garbage Collector ([**TODO: Querverweis**]) eingesammelt. Mit immer neuen Eingaben entstanden dann immer neue Symbole und die Symboltabelle wuchs und wuchs, bis das Betriebssystem das Programm wegen Speichermangels abschoß. Seit Ruby 2.0 [**TODO: Prüfen**] ist das Problem entschärft worden.

[5] Sie sind sich sogar so ähnlich, daß mehrfach der Vorschlag zur Diskussion stand, die Klasse Symbol von String erben zu lassen. Bisher ist der Vorschlag aber noch jedes Mal abgelehnt worden.

Symbole können jetzt vom Garbage Collector freigegeben werden. Dennoch ist der Einsatz von String#to_sym in aller Regel aus semantischen Gründen nicht empfehlenswert (s.o.).

Anders als bei Strings ist jedes Symbol *eindeutig*, d. h. jedes Mal, wenn man das Doppelpunkt-Literal verwendet, greift man auf *dasselbe* Objekt zu. Dies läßt sich demonstrieren:

```
p :foo.object_id  #=> 1237788
p :foo.object_id  #=> 1237788
p "foo".object_id #=> 47318292339580
p "foo".object_id #=> 47318292304260

p :foo.equal?(:foo)   #=> true
p "foo".equal?("foo") #=> false
```

Wichtigster Einsatzzweck für Symbole abseits der Metaprogrammierung ist als Schlüssel für Hashes. Beide Verwendungen werden in den jeweiligen Kapiteln beschrieben. Abseits hiervon sind Einsatzzwecke rar; es gilt die Grundregel, daß man immer dann, wenn man etwas eindeutig benennen will, ein Symbol, und immer dann, wenn die Eingabe potentiell verändert werden soll, einen String benützen sollte.

11.5 Reguläre Ausdrücke

Reguläre Ausdrücke (engl. »regular expressions«, kurz »RegExp«) sind ein möchtiges Werkzeug zur Extraktion von Informationen aus Texten. Anhand eines vorgegebenen Musters *engl.* (*pattern*) durchsuchen sie einen String und versuchen die Stelle zu finden, an der das Muster auf den String paßt (*match*). Ruby verfügt über eingebaute exzellente Unterstützung nicht nur für perl-kompatible Reguläre Ausdrücke (*PCRE*), sondern stellt darüber hinaus noch eigene Elemente zur Verfügung. Möglich wird dies durch Einsatz der mächtigen RegExp-Engine Onigmo, bei der es sich um eine Ruby-eigene Abspaltung von Oniguruma handelt.

Über Reguläre Ausdrücke allein sind Bücher geschrieben worden, auf die für den professionellen Umgang mit ihnen verwiesen werden muß. Hierbei verdient das speziell auf Rubys Reguläre Ausdrücke zugeschnittene Buch [**TODO: Titel?**] von *Nádasi-Donner* besondere Erwähnung. Dieser Abschnitt kann nur die Grundlagen vermitteln.

Reguläre Ausdrücke sind in Ruby (wie alles andere sonst auch) Objekte; sie sind Instanzen der Klasse Regexp. Zur Erstellung bietet Ruby eine auf Schrägstrichen basierende Literal-Syntax an, welche sich im Hinblick auf Interpolationen und Escape-Sequenzen so verhält wie String-Literale, die per doppelten Anführungszeichen erstellt werden. Lediglich Schrägstriche im Literal müssen durch Voranstellung eines rückwärtigen Schrägstrichs \ maskiert werden. Enthält ein regulärer Ausdruck zahlreiche Schrägstriche, kann man eine alternative Literal-Syntax auf Basis des Prozentzeichens benützen: %r{...}, wobei statt der geschweiften Klammern auch andere Begrenzer zulässig sind. Daneben besteht die Möglichkeit, durch Aufruf der Methode Regexp::new auch ausdrücklich eine Instanz zu erzeugen, was insbesondere bei der Erstellung von Regulären Ausdrücken aus Nutzereingaben sinnvoll sein kann. Regexp::new kompiliert den übergebenen String als Regulären Ausdruck.

```
# Normal:
r = /abc/
r = Regexp.new("abc")
r = %r{abc}
r = %r!abc!

# Mit Schrägstrich:
r = /<br\/>
r = %r{<br/>}

# Mit Escape-Sequenz und Interpolation:
s = "str"
r = /abc\t#{s}/
```

Es gibt zwei primäre Wege, einen String gegen einen regulären Ausdruck zu prüfen: die Methode #match und den besonderen Match-Operator =~. Sowohl die Methode #match als auch der Match-Operator sind sowohl auf der Klasse String als auch auf der Klasse Regexp definiert, sodaß es nicht darauf ankommt, welches Objekt einer Musterprüfung Sender und welches Empfänger ist. Der Operator =~ gibt bei einem Treffer die Position zurück, an dem das Muster beginnt, widrigenfalls nil. #match gibt im Erfolgsfall ein MatchData-Objekt (sonst: nil) zurück. Darauf wird später zurückzukommen sein (→11.5.2). Für den Moment genügt zu wissen, daß MatchData#to_s den Teil eines Strings ausgibt, der auf das angeforderte Muster paßt. Der Rückgabe-

wert von #match kann damit gut an #puts übergeben werden.

Das nachfolgende, einleitende Beispiel illustriert den Einsatz von #match und demonstriert zugleich die Nützlichkeit regulärer Ausdrücke. Gegeben sei ein String, der neben Buchstaben und Satzzeichen nur eine einzige Ziffer enthält. Aufgabe ist es, die Ziffer aus dem String zu extrahieren. Die herkömmliche Lösung dieses Problems wäre eine Schleife, die den String Zeichen für Zeichen abtastet, bis die Ziffer gefunden wäre. Eine einigermaßen idiomatische Lösung in Ruby würde so aussehen:

```
string = "Es ist jetzt 4 Uhr."
digits = %w[0 1 2 3 4 5 6 7 8 9]
puts string.chars.find{|c| digits.include?(c)} #=> 4
```

Der Code zeigt die Ausdruckskraft von Rubys Syntax (viele andere Programmiersprachen können die Aufgabe nicht als Zweizeiler lösen), ist aber trotzdem alles andere als verständlich. Dagegen findet sich dieselbe Aufgabe mit einem regulären Ausdruck verständlicher und kürzer gelöst:

```
string = "Es ist jetzt 4 Uhr."
puts string.match(/\d/) #=> 4
```

Erweitert man die Aufgabe dahingehend, daß nicht nur eine Ziffer, sondern eine ganze Zahl gefunden werden soll, dann ist die Lösung ohne reguläre Ausdrücke nicht mehr ohne erhebliche zusätzliche Komplexität durchführbar. Dagegen wird die Lösung mit regulären Ausdrücken um gerade einmal ein einziges Zeichen länger:

```
string = "Es ist jetzt 14 Uhr."
puts string.match(/\d+/) #=> 14
```

Reguläre Ausdrücke sind damit eine Möglichkeit, komplexe Musterabgleiche in einer außerordentlich präzisen Form zu beschreiben.

11.5.1 Aufbau

Ein regulärer Ausdruck besteht aus der Zusammensetzung von vier verschiedenen Elementen:

- Normalen Zeichen und Escape-Sequenzen,

^ $ ( ) [ ] { } | . ? + * \

Tabelle 11.5: Sonderzeichen in regulären Ausdrücken

- Gruppenzeichen und Zeichenklassen,
- Ankerzeichen,
- Wiederholungszeichen und die Gruppe.

11.5.1.1 Normale Zeichen und Escape-Sequenzen

Normale Zeichen sind alle Zeichen, die nicht in eine der anderen genannten Zeichengruppen fallen. Praktisch bedeutet das vor allem, daß die gewöhnlichen Buchstaben, Ziffern und die meisten Satzzeichen in regulären Ausdrücken keine besondere Bedeutung aufweisen. Die normalen Zeichen müssen (sofern kein anderer Befehl eingreift) genau so im abzugleichenden String vorkommen.

```
r1 = /ro/
r2 = /rt/
s = "Strohhalm"
s =~ r1 # Trifft zu
s =~ r2 # Trifft nicht zu
```

Wie bereits erwähnt unterstützen reguläre Ausdrücke alle Escape-Sequenzen, die auch normale, mit doppelten Anführungszeichen erstellte Strings unterstützen (siehe Tafel 11.1). Dazu kommt die Unterstützung der Maskierung des Schrägstrichs mithilfe von \/. Weiterhin müssen alle Zeichen in einem regulären Ausdruck, die nicht zu den normalen Zeichen gehören, mit einem vorangestellten rückwärtigen Schrägstrich maskiert werden, wenn die ihnen zugewiesene besondere Bedeutung im Einzelfall nicht erwünscht ist. Tafel 11.5 listet alle diese Zeichen zwecks Überblicks auf. Auf die Funktion der einzelnen Zeichen wird nachfolgend an passender Stelle eingegangen.

```
# Beispiel: eckige Klammer als Teil des Musters selbst.
str = "[ANN] Besonders wichtige Mitteilung"
puts str.match(/\[.+\]/) #=> [ANN]
```

11.5.1.2 Gruppenzeichen und Zeichenklassen

Obwohl sie ähnlich wie Escape-Sequenzen aussehen, haben Gruppenzeichen eine andere Funktion. Während Escape-Sequenzen der Darstellung nicht darstellbarer Zeichen dienen (dazu →11.1.2), haben Gruppenzeichen den Zweck, quasi stellvertretend für eine ganze Gruppe von Zeichen zu stehen. Auf ein Gruppenzeichen paßt folglich immer jedes Mitglied der vertretenen Gruppe. Die vollständige Liste der Gruppenzeichen kann Tafel 11.6 entnommen werden. Ein Beispiel eines Gruppenzeichens (`\d`) findet sich weiter oben im einführenden Beispiel zu regulären Ausdrücken.

Gruppenzeichen sind die Kurzschreibweise von besonders häufig benützten Zeichenklassen. Diese funktionieren konzeptionell genauso wie die Gruppenzeichen: an der Position, an der eine Zeichenklasse eingesetzt wird, ist jedes Zeichen zulässig, das durch die Zeichenklasse vertreten wird. Der Unterschied zu den Gruppenzeichen liegt darin, daß bei den Zeichenklassen der Programmierer festlegt, welche Zeichen von der Zeichenklasse vertreten werden. Bei den Gruppenzeichen dagegen ist dies von der Programmiersprache Ruby selbst festgelegt und kann auch nicht verändert werden. Tafel 11.6 führt in der zweiten Spalte („Langform") die einem Gruppenzeichen entsprechende Zeichenklasse auf.

Eine Zeichenklasse wird durch eine eckige Klammer eingeleitet. Alle Zeichen zwischen der öffnenden und der schließenden eckigen Klammer werden durch diese Zeichenklasse vertreten. Ist das erste Zeichen einer Zeichenklasse dagegen ein Spitzdach ^, dann wird die Zeichenklasse *negiert*, d. h. sie vertritt alle Zeichen *außer* den in ihr aufgelisteten Zeichen. Beispiele:

```
str1 = "Bar"
str2 = "Bär"
str3 = "Bor"
r1   = /B[aä]r/
r2   = /B[^aä]r/

str1 =~ r1 # Trifft zu
str2 =~ r1 # Trifft zu
str3 =~ r1 # Trifft nicht zu

str1 =~ r2 # Trifft nicht zu
str2 =~ r2 # Trifft nicht zu
str3 =~ r2 # Trifft zu
```

Mithilfe des Bindestrichs können Zeichenklassen abgekürzt geschrieben werden. Der Bindestrich wird von Ruby intern durch alle Zeichen zwischen dem Zeichen vor und nach dem Bindestrich ersetzt. Deshalb kann man die Zeichenklasse [0123456789] kürzer als [0-9] schreiben (wobei man dann natürlich auch gleich \d verwenden könnte). [0-9a-z] ist identisch mit [0123456789abcdefghijklmnopqrstuvwxyz]. Ist ein Bindestrich als zu vertretendes Zeichen gewünscht, ist es mit \ zu maskieren, z. B. paßt die Zeichenklasse [,;!\-_] auf diverse Sonderzeichen.

Die herkömmlichen Gruppenzeichen decken nur den ASCII-Zeichenbereich ab (zu Zeichensätzen →11.2). Mit zunehmendem Einsatz von Unicode haben diese sich mehr und mehr als unzulänglich herausgestellt. So erfaßt das Gruppenzeichen \w zwar den Buchstaben a, nicht aber den Buchstaben ä. Schon für deutschsprachige Texte ist das ärgerlich und machte einen Rückgriff auf die Langschreibweise der Zeichenklassen notwendig: statt \w mußte man schreiben: [a-zA-ZäöüÄÖÜß0-9_]. POSIX hat deshalb die in Tafel 11.7 aufgeführten Kurzschreibweisen für Zeichenklassen mit Unterstützung für Nicht-ASCII-Zeichen eingeführt. Diese Kurzschreibweisen sind an den Unicode-Standard angelehnt, in dem jedem Zeichen auch eine Kategorie zugewiesen ist. So ist gewährleistet, daß auch bei Weiterentwicklung des Unicode-Standards neue Zeichen korrekt erfaßt werden. Die Kurzschreibweise [[:digit:]] etwa erstellt eine Zeichenklasse, welche alle Zeichen vertritt, die in Unicode mit der Kategorie „Nd“ (Ziffern) versehen sind. Ein Beispiel zur Verarbeitung der in E-Mails gängigen Betonung durch Asterisken *:

```
str = "Ich halte das *für besonders wichtig*."
r1 = /\*[[:alnum:]]\*/
r2 = /\*\w\*/
puts str.match(r1) #=> für besonders wichtig
puts str.match(r2) #=> (Nichts, paßt nicht)
```

11.5.1.3 Anker

Ein regulärer Ausdruck macht zunächst keinen Unterschied danach, an welcher Stelle im String er paßt. Keinesfalls sollte man den Fehler machen und glauben, daß ein regulärer Ausdruck immer an den Anfang eines Strings gebunden wäre. Deshalb ist der folgende Ausdruck auch wahr:

```
"Dort entlang!" =~ /!/ #=> 12
```

Gruppenzeichen	Langform	Vertretene Zeichen
.	—	Jedes beliebige Zeichen.
\d	[0-9]	Alle ASCII-Ziffern.
\D	[^0-9]	Alles außer \d.
\h	[0-9a-fA-F]	Hexadezimalziffern.
\H	[^0-9a-fA-F]	Alles außer \h.
\s	[\t\r\n\f\v]	ASCII-Weißraum.
\S	[^ \t\r\n\f\v]	Alles außer \s.
\w	[a-zA-Z0-9_]	Alle ASCII-Buchstaben u. -ziffern.
\W	[^a-zA-Z0-9_]	Alles außer \w.

Tabelle 11.6: Gruppenzeichen in regulären Ausdrücken

POSIX-Klasse	Vertretene Zeichen
`[[:alnum:]]`	Alphanumerische Zeichen
`[[:alpha:]]`	Alphabetische Zeichen
`[[:blank:]]`	Leer- und Tabulatorzeichen
`[[:cntrl:]]`	Steuerzeichen
`[[:digit:]]`	Ziffern
`[[:graph:]]`	Nicht-Weißraum-Zeichen
`[[:lower:]]`	Kleinbuchstaben
`[[:print:]]`	Wie [[:graph:]], aber mit dem Leerzeichen
`[[:punct:]]`	Satzzeichen
`[[:space:]]`	Jeglicher Weißraum ([[:space:]] u. a. m.)
`[[:upper:]]`	Großbuchstaben
`[[:xdigit:]]`	Hexadezimalziffern.
Von Ruby unterstützte Nicht-POSIX-Klassen:	
`[[:ascii:]]`	Alle ASCII-Zeichen.
`[[:word:]]`	Zeichen der Unicode-Kategorien „Letter“, „Mark“, „Number“, „Connector_Punctuation“

Tabelle 11.7: POSIX-Zeichenklassen

Anker	Beschreibung
^	Paßt auf den Zeilenanfang
$	Paßt auf das Zeilende
\A	Paßt auf den String-Anfang
\z	Paßt auf das String-Ende
\Z	Paßt auf das String, akzeptiert aber noch \n danach.

Tabelle 11.8: Anker in regulären Ausdrücken

Es kommt allerdings häufig vor, daß man die Geltung eines regluären Ausdrucks auf den Anfang eines Strings oder doch wenigstens einer Zeile (denn man sollte nicht vergessen, daß Strings durchaus mehrzeilig sein können) beschränken möchte. Dafür kommen Anker zum Einsatz. Die wichtigsten können Tafel 11.8 entnommen werden. Es gibt noch einige mehr, die aber den Rahmen dieses Tutoriums sprengen würden.

Das Linux-Programm 1(dmesg) gibt die Log-Meldungen des Linux-Kernels aus. So eine Meldung sieht zum Beispiel so aus:

```
[    0.214442] pci 0000:00:00.0: [1011:0a3e] type 00 class 0x0b1
```

In eckigen Klammern am Anfang (und nur dort) befindet sich ein Zeitstempel, der die Anzahl der Sekunden seit Systemstart angibt. Wenn man den Zeitstempel auslesen will, bietet sich folgender Code an:

```
puts str.match(/^\[.*\]/) #=> [    0.214442]
```

Wenn reguläre Ausdrücke zur Validierung von Nutzereingaben (Beispiel: gültige E-Mail-Adresse) benützt werden, kann die nachlässige Verwendung der zeilenbasierten Anker ^ und $ zu korrupten Daten und nachfolgend zu einer Sicherheitslücke führen. Wenn ein Nutzer etwa als E-Mail-Adresse »johndoe@example.com\nMalicious Data« angibt, dann würde die Eingabe bei Verwendung des Ausdrucks `/^.+@.+\$/` als gültig erkannt und akzeptiert. Das Problem wird umgangen durch Einsatz der string-basierten Anker: `/\A.+@.+\z/`.

11.5.1.4 Wiederholungszeichen und Gruppen

Es gibt in regulären Ausdrücken vier Wiederholungszeichen:

- * wiederholt den vorangegangenen Ausdruck nullmal bis beliebig häufig.
- + wiederholt den vorangegangen Ausdruck einmal bis beliebig häufig.
- ? Null- oder einmal. Diesen Operator betrachtet man am besten als Kennzeichner für einen optionalen Teil des Musters.
- {n,m} wiederholt den vorangegangenen Ausdruck mindestens n-mal, höchstens m-mal. Diese Wiederholung kennt noch Varianten:
  - {n} genau n-mal.
  - {n,} mindestens n-mal bis beliebig oft.
  - {,m} nullmal bis höchstens m-mal.

Wiederholungen sind nützlich, weil sie (ähnlich wie Schleifen in der Programmierung) den Programmierer davon befreien, alle seine Anweisungen mehrfach schreiben zu müssen.

```
"Raaaaaah!".match(/a+/) #=> aaaaaa
"Wal".match(/Wah?l/)    #=> Wal
```

Wiederholungszeichen werden meistens gebraucht in Kombination mit Gruppen. Eine Gruppe wird durch runde Klammern (...) festgelegt und gruppiert die darin enthaltenen Muster zu einem einzigen Ausdruck. Innerhalb einer Gruppe kann man den Alternierungsoperator | benützen. Er verknüpft die Ausdrücke auf seinen beiden Seiten mit einem logischen ODER, d. h. entweder die eine oder die andere Seite muß zutreffen. Tatsächlich kann man den Alternierungsoperator auch ohne Gruppen verwenden, aber weil er sich dann auf den gesamten regulären Ausdruck bezieht, ist er in dieser Form meist wenig nützlich.

```
"Fußball"  =~ /u(ss|ß)b/  #=> 1
"Fussball" =~ /u(ss|ß)b/  #=> 1

"Fussball" =~ /u(ss|ß)?b/   #=> 1
"Fuball"   =~ /u(ss|ß)?b/   #=> 1
```

Auf diese Weise kann man also auch größere Teile des Musters variieren, während Zeichenklassen nur auf ein einziges Zeichen beschränkt sind.

11.5.2 Die Abgleichsmethoden

Wie bereits erwähnt stellt Ruby verschiedene Methoden zum Musterabgleich zur Verfügung. Die einfachste Möglichkeit, einen Musterabgleich nach den beschriebenen Regeln durchzuführen, ist der Operator =~ . Er gibt im Erfolgsfalle die Position zurück, an der das Muster auf den String paßt; man rufe sich in Erinnerung, daß 0 (Null) in Ruby als wahr gilt. Paßt der reguläre Ausdruck nicht, wird nil zurückgegeben.

```
"Taxi" =~ /T/ #=> 0
"Taxi" =~ /U/ #=> nil
```

Demgegenüber gibt die Methode #match eine Instanz der Klasse MatchData zurück. Diese Klasse stellt Methoden zur genaueren Untersuchung des Prüfungsergebnisses zur Verfügung. Die wichtigsten Methoden werden hier beispielhaft vorgestellt; im Übrigen sei auf die Dokumentation verwiesen.

```
str = "[ALERT] Important log message"
md  = str.match(/^\[(\w+)\]/)

# Der []-Operator gibt die bezeichnete Gruppe zurück.
# Gruppe 0 ist das gesamte Muster, -1 die letzte Gruppe.
p md[0] #=> "[ALERT]"
p md[1] #=> "ALERT"
# MatchData#pre_match und MatchData#post_match
# geben zurück, was sich vor oder nach dem Match
# befindet.
p md.pre_match #=> ""
p md.post_match #=> " Important log message"
```

Ein praktisch wichtiger Anwendungsfall für reguläre Ausdrücke ist die Substitution von Mustern mit einem anderen. Dafür bietet String die zwei Methoden String#sub und (wichtiger) String#gsub an. Während letztere alle Vorkommen eines Musters im String ersetzt, ersetzt erstere nur das erste. Beide Methoden nehmen als erstes Argument das zu ersetzende Muster und als zweites die neue Fassung an. In der neuen Fassung kann mithilfe von \n auf den Inhalt der n-ten Gruppe des aktuellen Musterabgleichs zugegriffen werden (n=0 bezieht sich auf das gesamte Muster). Diese Funktionalität ist *keine* Escape-Sequenz. Sie wird nicht von Rubys Grammatik bereitgestellt, sondern

von #sub bzw. #gsub implementiert. Daraus folgt insbesondere, daß der rückwärtige Schrägstrich auch bei der Methode ankommen muß und nicht vom Parser entfernt werden darf. Am einfachsten erreicht man das, indem man das Argument für die neue Fassung in einfache statt doppelten Anführungszeichen setzt.

```
"Aiiiiii".sub(/i+/, 'n') #=> "An"
"An Ab Am".gsub(/A\w/, '*\0*') #=> "*An* *Ab* *Am*"
```

Das Problem mit den Schrägstrichen in #(g)sub gehört zu den berüchtigten Unebenheiten in Ruby. Wie viele rückwärtige Schrägstriche bräuchte man, um einen rückwärtigen Schrägstrich einzufügen? Die Antwort lautet: nicht weniger als vier. Daran ändert wegen der besonderen Semantik des rückwärtigen Schrägstrichs auch der Einsatz einfacher Anführungszeichen statt doppelter nichts. Zu ergründen, welcher Schrägstrich nun was bedeutet, sei dem geneigten Leser zum Selbststudium überlassen.

```
puts ">> X <<".sub(/X/, '\\\\') #=> >> \ <<
```

Ganz umgehen kann man solche Probleme durch Einsatz der Blockform der genannten Methoden, die dafür allerdings auch nicht über die Gruppensyntax verfügt[6] Obiges Beispiel benötigt dann nur noch den einen zusätzlichen Schrägstrich für den Parser.

```
puts ">> X <<".sub(/X/){ "\\" } #=> >> \ <<
```

Oft kommt es vor, daß man prüfen will, ob ein Muster auf einen String paßt, um im Erfolgsfalle den String zu verarbeiten. Beispielsweise könnte es erforderlich sein, aus einer Logdatei bestimmte Zeilen auszufiltern und diese in ihre Einzelteile zu zerlegen. Da #match nil zurückgibt, wenn das Muster nicht paßt, ließe sich das ohne zusätzlichen Musterabgleich wie folgt realisieren:

```
loglines.each do |line|
  if md = line.match(/^<myprog> (\[\w+\]) (.*)$/
    puts "Loglevel #{md[1]} with message #{md[2]}"
  end
end
```

[6] Allerdings gibt es einen Weg, das nachzustellen. Dazu unten im Text.

Jedoch hat dieses Vorgehen den Nachteil, daß die Zeile recht lang wird und zudem eine Zuweisung in der Bedingung vorkommt. Zuweisungen in Bedingungen sind so weit wie möglich zu vermeiden, um nicht in Gefahr zu geraten, die Operatoren = und == miteinander zu verwechseln. Solcher Gebrauch wie hier wäre zwingend mit einem klarstellenden Kommentar zu versehen, daß das einfache Gleichheitszeichen beabsichtigt ist. Ärgerlicherweise verfügt Ruby sogar über einen Operator für Musterabgleiche, =~, der allerdings kein MatchData-Objekt zurückgibt (und selbst wenn, bestünde das Zuweisungsproblem weiterhin) und für diesen Einsatzzweck deshalb scheinbar ungeeignet ist. Doch gibt es einen Trick. Ruby setzt für jeden durchgeführten Musterabgleich einige globale Variablen. Unter Rückgriff auf diese kann das obige Beispiel mit dem Operator =~ umgeschrieben werden:

```
loglines.each do |line|
  if line =~ /^<myprog> (\[\w+\]) (.*)$/
    puts "Loglevel #{$1} with message #{$2}"
  end
end
```

Die globalen Variablen $1 bis $9 enthalten den Inhalt der entsprechenden Gruppen des letzten Musterabgleichs, $& (nicht $0 — das ist etwas ganz anderes[7]!) enthält das gesamte Muster. Es werden noch einige weitere globale Variablen gesetzt, die in Tafel 11.9 aufgeführt werden. Diese Gruppe globaler Variablen sollte jedem Ruby-Programmierer bekannt sein. Mit ihrer Hilfe kann man auch in der Blockform von #(g)sub die Substitution von Gruppentext durchführen:

```
"An Ab Am".gsub(/A\w/){"*#{$&}*"} #=> "*An* *Ab* *Am*"
```

Wiederholungsfragen und -aufgaben

1. Was ist eine Interpolation?
2. Ein Kunde hat sich beim Support beschwert, daß er eine Mahnung ohne vorangehende Rechnung bekommen hat. Außerdem sei auf der Mahnung seine Adresse („Lönsstraße 15“ in Dortmund) mit

[7] Siehe →12.3.4.

Variable	MatchData	Bedeutung
$~	md	Das MatchData-Objekt des Abgleichs.
$&	`md[0]`	Gesamter passender Text.
$1...$9	`md[n]`	Text in Gruppe »n«.
$+	`md[-1]`	Text in der letzten Gruppe.
$‘	`md.pre_match`	Text vor dem Muster.
$’	`md.post_match`	Text nach dem Muster.

Tabelle 11.9: Globale Variablen bei regulären Ausdrücken

unverständlichen Zeichen geschrieben (ein Photo zeigt „LÃ¶nsstraÃ?e 15“) Die Rechnungsabteilung teilt mit, die automatisert aufgebene Rechnung sei in der Tat als unzustellbar zurückgekommen, allerdings erst, nachdem man die Mahnung bereits (ebenfalls automatisiert) verschickt hatte. In der Rechnungsabteilung werden ausschließlich Windows-PCs eingesetzt. Wo vermuten Sie das Problem?

3. Was bedeuten die globalen Variablen $4, $ und $’?

4. Die Betreffzeilen von E-Mails an Mailing-Listen, die Veröffentlichungen oder Ereignisse ankündigen, beginnen meistens mit der Zeichenkette „[ANN]“, „[ANK]“ oder „[ann]“. E-Mails, die nichts mit Ankündigungen zu tun haben, weisen diese Kennzeichnung nicht auf. Antworten haben generell ein führendes „Re:“ im Betreff. Schreiben Sie ein Programm, daß aus der unten angegebenen Liste von E-Mail-Betreffzeilen alle Ankündigungen herausfiltert. Antworten auf Ankündigungen sind auszulassen.

```
[ANN] RDL 2.2.0 release
How does the class name magic with Class.new?
Re: How does the class name magic with Class.new?
[ANN] hoe 3.17.2 Released
Re: [ANN] hoe 3.17.2 Released
[ANK] Deutschsprachige Ruby-Mailingliste eröffnet
Re: [ANK] Deutschsprachige Ruby-Mailingliste eröffnet
ruby rookie needs help with "ri" command
Problems with using [...] character classes in regexp
```

§ 12 Die Programmumgebung

Auf einem Computer laufen typischerweise dutzende Programme nebeneinander ab, wenn auch echte Gleichzeitigkeit nur bei Computern mit mehreren Kernen möglich ist (→ 3.1.1). Diese Prozesse können sich auf verschiedene Arten beeinflussen. Ruby bringt dazu die notwendigen Bordmittel mit, die das Thema dieses Kapitels bilden.

12.1 Kommandozeilenargumente

Wenn man auf der Kommandozeile ein Programm ausführt, so kann man ihm zusätzliche Argumente mitgeben. In der Einleitung (→ 2.3) konnte man dies beim Aufruf des Ruby-Interpreters beobachten, etwa hier:

```
$ ruby hallo.rb
```

Hier wird dem Programm ruby das Kommandozeilenargument »hallo.rb« übergeben. ruby erkennt, daß es sich hierbei um einen Dateinamen handelt und verarbeitet den Inhalt dementsprechend als Ruby-Code. Kommandozeilenargumente müssen aber keinesfalls immer Dateinamen sein. Der Schalter »-w« etwa schaltet Warnungen für schlechten Programmierstil ein (Diagnosemodus).

```
$ ruby -w /tmp/beispiel.rb
/tmp/beispiel.rb:3: warning: mismatched
indentations at 'end' with 'if' at 2
```

Aus Sicht des Programms ruby wurden zwei Kommadozeilenargumente übergeben, »-w« und »hallo.rb«. Es ist üblich, Schalter, die die Ausführung eines Programms verändern, mit einem Minuszeichen zu beginnen, zwingend ist das aber nicht. Es liegt letztlich beim aufgerufenen Programm, etwas aus den ihm übergebenen Argumenten zu machen. Das UNIX-Programm date erkennt eine Zeitformatangabe etwa anhand eines Pluszeichens.

```
$ date +%Y
2018
```

Bevor ein Programm allerdings die Kommandozeilenargumente zu Gesicht bekommt, werden sie durch die Kommandozeile selbst (Shell) verarbeitet. Einzelheiten unterscheiden sich von Shell zu Shell, auf eine Besonderheit soll aber wegen der besonderen Relevanz an dieser Stelle eingegangen werden. Praktisch alle Shells verarbeiten gewisse Sondersequenzen und ändern den Programmaufruf entsprechend. Unter Bash kann man etwa mithilfe von »!!« das zuvor ausgeführte Kommando wiederholen. Dieser Verarbeitungsschritt geht das aufgerufene Programm nichts an und es gibt auch keine Möglichkeit, eine solche Vorverarbeitung durch die Kommandozeile festzustellen; dies schon allein deshalb, weil es eine Unmenge verschiedener Shells gibt. Das ist aber auch nciht nötig, denn die richtige Benutzung der Shell darf vom Endnutzer erwartet werden.

Die wichtigste Ausprägung der Vorverarbeitung durch die Shell wird beim *Quoting* sichtbar. Viele Shells verarbeiten mit einem Dollarzeichen beginnende Worte als Variablennamen, z. B. $LANG, und ersetzen diesen Teil des Kommandos durch den Wert der Variablen:

```
$ echo $LANG
de_DE.UTF-8
```

Um diese Sonderbehandlung zu unterbinden, wird das Sonderzeichen in den meisten Shells durch Einsatz von Anführungsstrichen, engl. »quotation marks«, maskiert. Man bezeichnet ihren Einsatz daher als (*Quoting*):

```
echo '$LANG'
$LANG
```

Viele Shells unterscheiden zwischen einfachen und doppelten Anführungszeichen; Einzelheiten müssen der Dokumentation der benutzten Shell entnommen werden. Wichtig ist hier nur, daß das aufgerufene Programm, im Beispiel echo, nur das Ergebnis der Verarbeitung zu Gesicht bekommt.

Besondere Relevanz erhält dieses Verhalten im Falle von Leerzeichen. Praktisch alle Shells behandeln das Leerzeichen als Argument-Separator (so wie in Ruby das Komma). Will man also etwa eine Datei mit Leerzeichen im Namen an den Ruby-Interpreter übergeben, muß das Leerzeichen maskiert werden. Weil der Ruby-Interpreter davon ausgeht, daß nur das erste Kommandozeilenargument, das nicht mit einem Minus beginnt, der Name der auszuführenden Datei ist, würde er den Dateinamen in diesem Fall falsch erkennen.

```
$ ruby 'Meine Datei.rb'
```

Die Anführungszeichen maskieren das Leerzeichen, und »Meine Datei.rb« wird ruby als *einzelnes* Argument übergeben, und nicht etwa als zwei (»Meine« und »Datei.rb«). Die Vorverarbeitung durch die Shell bleibt dem Programm verborgen.

Speziell bei Ruby-Skripten muß zwischen den Kommandozeilenargumenten für den Ruby-Interpreter selbst (ruby) und den Kommandozeilenargumenten für das Ruby-Skript unterschieden werden. Die Grenze wird beim ersten erkannten Dateinamen gezogen, gleich ob die Datei nun existiert oder nicht.

```
$ ruby -w skript.rb -d all
```

In diesem Beispiel ist »-w« ein Argument für den Ruby-Interpreter, ebenso wie »skript.rb«. Diese Argumente sind aus dem Ruby-Skript heraus nicht erreichbar. Erst alles, was auf den Skript-Dateinamen folgt, kann verarbeitet werden. ruby reicht nämlich nur diese Kommandozeilenargumente an das Skript weiter. Was im Beispiel »skript.rb« nun mit »-d« und »all« anfängt, bleibt ihm überlassen.

Der Zugriff auf die Skript-Argumente erfolgt über die Konstanten ARGV und ARGF. Eine möglicherweise erwartete Konstante ARGC existiert nicht. Sie ist unnötig, denn ARGV ist ein Array, dessen Länge ganz normal per Aufruf von length::festgestellt werden kann.

> ARGV enthält nicht den Namen des Skripts als erstes Argument. Im obigen Beispiel ist das erste Element von ARGV deshalb »-d«. Das ist ein Unterschied zu C. Der Skriptname ist über $0 verfügbar (unten → 12.3.4).

Ein einfaches Skript »args.rb«, das einfach nur alle seine Kommandozeilenargumente ausgibt, mag deshalb so aussehen:

```
for arg in ARGV
  puts "Argument: #{arg}"
end
```

Ausgabe:

```
$ ruby -w args.rb -d all
Argument: -d
Argument: all
```

In Rubys Standardbibliothek befindet sich eine Programmbibliothek namens optparse, deren Einsatz sich bei komplexeren Argumentstrukturen dringlichst empfiehlt, da die manuelle Verarbeitung von Kommandozeilenargumenten schnell erhebliche Ausmaße annehmen kann. Details sind der Dokumentation von optparse zu entnehmen; zur Nutzung von Programmbibliotheken [**TODO: Querverweis**].

Mit ARGF verhält es sich etwas komplizierter. Dieses Kuriosum arbeitet *auf* ARGV, ersetzt es also nicht. ARGF ist ein IO-ähnliches Objekt (zu IO [**TODO: Querverweis**]), ds den Inhalt aller in ARGV benannten Dateien aneinandergereiht enthält. Immer, wenn eine Datei vollständig ausgelesen wurde, wird sie automatisch aus ARGV entfernt. Der aktuelle Dateiname steht als ARGF.filename::zur Verfügung; die vordefinierte globale Variable $< ist mit ARGF gleichwertig. Vor der Benutzung von ARGF sind Argumente, die sich nicht auf Dateien beziehen, aus ARGV zu entfernen, sonst kommt es zu einem Laufzeitfehler. Das UNIX-Werkzeug cat kann man damit etwa so nachbauen:

```
ARGV.reject!{|el| el.start_with?("-")}
puts ARGF.read
```

Sinnvollerweise sollten die mit »-« beginnenden Kommandozeilenargumente aber nicht wie hier verworfen, sondern — idealerweise mit optparse — verarbeitet werden, bevor mit dem Auslesen der Dateien begonnen wird.

12.2 Umgebungsvariablen

Jeder Prozeß besitzt eine Umgebung, in der verschiedene Variablen gesetzt sein können. Diese Umgebung wird bei der Erstellung von Prozessen an Kindprozesse vererbt. Erstmalig festgelegt wird sie bei der Anmeldung des Benutzers auf der Shell. Einzelheiten sind von der eingesetzten Shell abhängig. Die aktuell aktive Umgebung kann mithilfe des UNIX-Werkzeugs env untersucht werden.

Innerhalb von Ruby steht die Programmumgebung über die globale Konstante ENV zur Verfügung. Diese verhält sich wie ein Hash, dessen Schlüssel die Namen der Umgebungsvariablen und dessen Werte deren Werte sind. Ei-

ne typischerweise zur Verfügung stehende Umgebungsvariable ist »HOME«; sie gibt das Hauptverzeichnis des aktiven Benutzers an.

```
puts "Your home directory is #{ENV["HOME"]}."
#=> Your home directory is /home/quintus.
```

ENV ist schreibbar. Geänderte Werde sind im gesamten Programm sichtbar und werden an Kindprozesse **[TODO: Querverweis]** vererbt.

Ebenfalls über ENV verfügbar ist die wichtige Umgebungsvariable »LANG«. Sie gibt die vom Benutzer bevorzugte Sprache an. Ruby verfügt in der Grundausstattung aber nicht über Werkzeuge zur Verarbeitung von »LANG« oder den anverwandten »LC«-Umgebungsvariablen. Für die Verarbeitung dieser sogenannten *Locales* muß auf externe Programmbibliotheken wie i18n[1] oder r18n[2]. zurückgegriffen werden, wenn man nicht wirklich alles selbst implementieren will. Die Nutzung externer Programmbibliotheken wird später beschrieben **[TODO: Querverweis]**.

Die Zeichenkodierung (→ 11.2) der Werte in ENV ist ein etwas diffiziles Thema. Sie hängt von verschiedenen Faktoren ab und kann unter anderem auch das in Ruby sonst nur selten benutze Dateisystem- oder Locale-Encoding sein. Verallgemeinernd läßt sich sagen, daß Ruby jedenfalls sicherstellt, daß die von ENV zurückgegebenen Werte zwar keine leicht vorhersehbare Zeichenkodierung haben, aber jedenfalls die genau zurückgebenen Bytes in der zurückgegebenen Zeichenkodierung gültig sind. Anders ausgedrückt: wer mit Nicht-ASCII-Werten in ENV umgehen will, dem ist ein ausdrücklicher Aufruf von String#encode angeraten. Insbesondere hat weder die externe noch die interne Standardkodierung Einfluß auf die von ENV zurückgegene Zeichenkodierung.

```
str = ENV["MYSTR"].encode("UTF-8")
```

12.3 Umgebungsbezogene globale Variablen

Von den diversen globalen Variablen, die Ruby bei Programmstart definiert (s. Tafel 7.2), besitzen einige einen besonderen Bezug zur Programmumgebung. Die wichtigsten von ihnen werden nachfolgend vorgestellt.

[1] https://github.com/svenfuchs/i18n
[2] https://github.com/ai/r18n

12.3.1 $VERBOSE

Die globale Variable $VERBOSE zeigt an, ob Ruby im Diagnosemodus ausgeführt wird, d. h. ob für einige fragwürdige, aber zulässige Konstruktionen bei der Ausführung Warnungen angezeigt werden. $VERBOSE kann dabei drei Werte annehmen: true, false und nil. Zwischen false und nil gibt es einen winzigen Unterschied. In beiden Fällen werden Warnungen der Kernel-Methode #warn nicht mehr angezeigt. Warnungen aber, die auf der C-Ebene von Ruby mithilfe von rb_warn() ausgegeben werden (»Systemwarnungen«), werden nur dann unterdrückt, wenn $VERBOSE nil ist. Demzufolge ergibt sich für $VERBOSE das in Tafel 12.1 zusammengefaßte Bild.

Die Differenzierung zwischen zwei Arten von Warnungen ist Gegenstand laufender Kritik und es ist nicht unwahrscheinlich, daß sie irgendwann aufgegeben wird. Sie leuchtet auch nicht ein, denn die der Warnung übergeordnete Stufe ist nicht eine andere Art von Warnung, sondern eine Exception [**TODO: Querverweis**]. Schreibt man C-Extensions, sollte man sich auf die Nutzung von rb_warning() beschränken.

$VERBOSE wird in der Regel vom Aufrufer durch Übergabe einer der Kommandozeilenschalter -w, -W oder -v gesetzt, deren genaue Effekte ebenfalls Tafel 12.1 entnommen werden können; ohne besondere Vorkehrungen hat $VERBOSE bei Programmstart den Wert false. Die Variable darf aber auch vom Programm selbst gesetzt werden. Das ist insbesondere dann sinnvoll, wenn man Warnungen unterdrücken will, die aus dem Code benutzter Drittsoftware herrühren. So kann man seinen eigenen Quelltext »-w-fähig« machen, wie gern gesagt wird, ohne die ganze Zeit mit der mäßigen Qualität einer benutzten Programmbibliothek behelligt zu werden. Im Code sieht das dann meist so aus:

```
_v, $VERBOSE = $VERBOSE, nil
require "drittsoftware"
$VERBOSE, _v = _v, nil
```

Zu #require [**TODO: Querverweis**].

12.3.2 $DEBUG

Im Gegensatz zu $VERBOSE ist $DEBUG eine globale Variable, die man nicht selbst setzen sollte. Sie ist true, wenn Ruby im Debug-Modus läuft, was durch Übergabe des Kommandozeilenschalters »-d« erreicht werden kann. Das ist

$VERBOSE	Schalter	Ausgegeben werden
true	-w / -W	Alle Warnungen.
false	keiner / -W1	Nur Systemwarnungen (rb_warn()).
nil	-W0	Keine Warnungen.

Tabelle 12.1: Wirkung von $VERBOSE

sehr praktisch, um etwa die Ausgabe von Debugging-Informationen auch tatsächlich auf das eigentliche Debugging zu beschränken (eine Möglichkeit, von der viel zu selten Gebrauch gemacht wird). Dies ließe sich sogar in eine eigene Methode extrahieren:

```
def debug(str)
  $stderr.puts("Debug: #{str}") if $DEBUG
end
debug "Debugging-Info"
```

Ruby selbst mißt $DEBUG nur einen Effekt zu, der aber für das Debugging von Problemen mit Exceptions sehr wertvoll ist. Ruby gibt nämlich bei aktivem $DEBUG für jeden ausgelösten Laufzeitfehler eine Information mit Ursprungsdatei und -zeile aus — und zwar auch dann, wenn der Fehler mithilfe von `rescue` aufgefangen wird [**TODO: Querverweis**]. So kann man ungewollt aufgefangene Laufzeitfehler entdecken.

12.3.3 $SAFE

Ruby bietet mit der globalen Variable $SAFE einen rudimentären Sicherheitsmechanismus an. Die Idee dahinter ist, daß potentiell gefährliche Daten markiert werden (»tainting«) und bei entsprechendem $SAFE-Level in bestimmten Methoden nicht verarbeitet werden dürfen. Dies löst dann einen SecurityError aus. Die Nutzung von $SAFE hat sich jedoch nicht durchgesetzt und es gibt Pläne, $SAFE mit Ruby 3 ganz zu entfernen. Von einer Erläuterung wird daher ebenso abgesehen wie von der Beschreibung des temporär in Ruby eingeführten Trust-Systems, das bereits wieder gestrichen worden ist. Bei Bedarf können weitere Informationen der [**TODO: Pickaxe, S. 425ff.**] entnommen werden.

12.3.4 $0

Da ARGV in Ruby keine Möglichkeit bietet, auf den Programmnamen zuzugreifen, muß es dafür andere Mechanismen geben. Ruby stellt dafür die globale Variable $0 und das Schlüsselwort __FILE__ zur Verfügung. Beide erfüllen eine ganz ähnliche Funktion, unterscheiden sich aber minimal. $0 ist der Name, unter dem das Programm aufgerufen wurde und entspricht insoweit dem `argv[0]` von C. An $0 kann zugewiesen werden; hierdurch ändert sich der Name des Programms in der Prozeßliste, wie sie etwa von ps(1) angezeigt wird. __FILE__ dagegen enthält immer den Namen der Datei, in der es verwendet wird. Bei Programmen, die aus mehreren Quelltextdateien bestehen — also besonders bei Verwendung von #load oder #require, dazu **[TODO: Querverweis]** —, können die Werte auseinanderfallen. Daraus hat sich das folgende Idiom entwickelt:

```
if $0 == __FILE__
  start_program
end
```

Dieser Trick erlaubt es, dieselbe Datei sowohl als eigenständiges Programm wie auch als per #require ladefähige Programmbibliotek zu benutzen.

12.4 Informationen über den Ruby-Interpreter

Die vordefinierten Konstanten RUBY_VERSION, RUBY_RELEASE_DATE und RUBY_PLATFORM geben Aufschluß über den genauen Ruby-Interpreter, der das Skript ausführt. Sie geben Informationen über die genaue Versionsnummer, das Veröffentlichungsdatum und das Betriebssystem, auf dem Ruby derzeit läuft. Exemplarisch lauten die Werte auf dem Debian-9-System des Verfassers:

```
p RUBY_VERSION       #=> 2.3.3
p RUBY_RELEASE_DATE  #=> 2016-11-21
p RUBY_PLATFORM      #=> x86_64-linux-gnu
```

Tafel 12.2 listet verschiedene möglichen Werte von RUBY_PLATFORM auf ohne Anspruch auf Vollständigkeit auf; je nach genutztem C-Compiler und Betriebssystem können auch andere Werte auftreten. Es soll aber nicht unerwähnt bleiben, daß Ruby-Code in aller Regel plattformunabhängig ist und

Wert	Bedeutung
bccwin	Windows, kompiliert mit Borland C
bsd	BSD-Familie
cygwin	Windows, kompiliert mit Cygwin
darwin	MacOS
java	JVM (JRuby)
linux	Linux-Familie
mingw	Windows, kompiliert mit MinGW im normalen Modus
mswin	Windows, kompiliert mit MSVC
msys	Windows, kompiliert mit MinGW im MSYS-Modus
solaris	Solaris-Familie

Tabelle 12.2: Mögliche Werte von RUBY_PLATFORM

man sich mit der Überprüfung dieser Variablen daher sehr zurückhaltend geben sollte. Die Überprüfung von RUBY_VERSION hat dagegen bei der Erstellung von Code, der mit möglichst vielen Versionen von Ruby lauffähig sein soll, seine einschränkungslose Berechtigung. Ein sinnvoller, außerakademischer Einsatzzzweck für die Überprüfung von RUBY_RELEASE_DATE dagegen ist dem Verfasser bisher nicht bekannt geworden.

12.5 An Skript angehängte Daten

Es gibt ein kaum bekanntes Feature von Ruby, welches man benutzen kann, um größere Mengen von Informationen direkt in einem Skript zur Verfügung zu stellen, ohne auf externe Medien wie insbesondere Dateien zurückgreifen zu müssen. Das besondere Schlüsselwort __END__ markiert, wenn es allein in einer Zeile steht, das Ende eines Ruby-Skripts. Der Interpreter bricht die Ausführung des Skripts (ohne Fehlerstatus) ab, wenn er auf diese Zeile trifft. Ruby legt jedoch eine Konstante DATA an, wenn es erkennt, daß das Skript eine solche Markierung enthält. Diese Konstante verweist auf ein geöffnetes File-Objekt, welches das Ruby-Skript selbst im Lesemodus geöffnet hat und dessen Leseposition (Cursor) sich am Anfang der Zeile nach der __END__-Markierung befindet. Fehlt die Markierung, ist DATA gar nicht definiert und der Zugriff verursacht einen NameError.

Im nachfolgenden Beispiel wird diese Technik benutzt, um bei Übergabe des Schalters »-v« auf der Kommandozeile einen längeren Copyright-Hinweis nicht direkt im Code als String hinterlegen zu müssen.

```
if ARGV.include?("-v")
  puts DATA.read
end

puts "Starting program"
# …
__END__
This program is Copyright (C) 2018 John Doe.
All Rights Reserved.

You may only distribute, modify, or otherwise make use
of this programme under the terms of the license that
has been granted to you by the author. Violating these
terms makes you liable to any damage arising from it,
including but not limited to additional licensing fees.
```

Diese Technik kann mit der in der Standardbibliothek enthaltenen Programmbibliothek yaml kombiniert werden, um maschinenlesbare Informationen zusammen mit dem Skript auszuliefern, z. B. eine Standardkonfiguration.

```
require "yaml"

defaultconfig = YAML.load(DATA.read)
p defaultconfig
# …
__END__
option1: true
option2: 42
```

Ausgabe:

```
{"option1"=>true, "option2"=>42}
```

12.6 Shebang

Die sog. Shebang-Zeile ist ein Spezifikum unixoide Systeme und eigentlich nicht ruby-spezifisch. Da sie aber häufig Verwendung in Ruby-Programmen

findet, soll sie hier erläutert werden.

Auf unixoiden Systemen kann jede Textdatei durch Setzen des Executable-Bits in den Zugriffsrechten als ausführbares Programm markiert werden. Beginnt die erste Zeile dieses Programms mit der Zeichenkombination #!, dem sog. *Shebang* oder *Hashbang*, gibt der Rest der Zeile den Dateipfad zu dem Programm an, das zur Verarbeitung der Textdatei genutzt werden soll. Für Ruby-Quelltexte ist das der Ruby-Interpreter. Ist dieser in `/usr/bin/ruby` installiert, so könnte man eine Datei mit folgendem Inhalt anlegen:

```
#!/usr/bin/ruby
puts "Hallo!"
```

Die Datei kann dann wie folgt als ausführbar markiert und sodann als eigenständiges Programm benutzt werden:

```
$ chmod +x hallo.rb
$ ./hallo.rb
Hallo!
```

Ist das Verzeichnis, in dem die Programmdatei abgelegt wurde, Teil der Umgebungsvariablen »PATH«, kann das »./« entfallen. Aus Sicht von Ruby ergeben sich bei Einsatz eines Shebangs übrigens keine Besonderheiten. Da er mit dem Kommentarzeichen beginnt, ignoriert der Ruby-Interpreter die gesamte Shebang-Zeile als Kommentar. Hierin dürfte auch der Grund zu erblicken sein, weshalb so viele Skriptsprachen sich für das Rautezeichen # als Kommentarzeichen entscheiden.

Will man das Programm verteilen, ist die feste Angabe des Pfads zum Ruby-Interpreter unpraktisch, denn möglicherweise hat der Nutzer seinen Ruby-Interpreter anderswo, z. B. unter `/usr/local/bin/ruby` (typisch für Selbstkompilation), installiert. Gängige UNIX-Systeme bieten zur Lösung des Problems ein Programm namens env an, das sich meist in `/usr/bin/env` befindet und die PATH-Umgebungsvariable nach dem übergebenen Kommandozeilenargument durchsucht und dieses dann als Programm ausführt. Es ist deshalb nicht ungewöhnlich, Shebang-Zeilen der folgenden Art zu benutzen:

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

Freilich hilft dies nicht, wenn env nicht in `/usr/bin/env` liegt oder gar nicht installiert ist. Das ist aber unwahrscheinlich genug, um das Problem ignorieren zu können, zumal man wohl erwarten darf, daß sich in `/usr/bin/env`

dann wenigstens ein Symlink auf den richtigen Installationsort von env befindet

§ 13 Nebenläufigkeit

Ein Ruby-Programm wird, wenn keine weiteren Vorkehrungen getroffen werden, bei der Ausführung streng chronologisch Zeile für Zeile abgearbeitet. Methodenaufrufe und Bedingungen verändern zwar die Position, ab der Code ausgeführt wird, ändern aber nichts daran, daß weiterhin nur eine Anweisung nach der anderen ausgeführt wird. Sind alle Anweisungen eines Programms abgearbeitet, dann ist es beendet.

Die in diesem Kapitel vorgestellten Funktionalitäten verändern diesen Mechanismus. Sie erlauben die gleichzeitige Ausführung mehrerer Anweisungen.

13.1 Threads

13.1.1 Allgemeines

Threads sind die einfachste (und meistgenutzte) Art, verschiedene Aufgaben parallel durchzuführen. Alle Ruby-Programme laufen zunächst mit nur einem Thread, dem Haupt-Thread. Mithilfe von Thread::new kann man einen neuen Thread erstellen, der dann weitgehend parallel zum Haupt-Thread läuft. Der Programmfluß wird also aufgespalten.

```
puts "Im Hauptthread"
Thread.new do
  sleep 3
  puts "Im Nebenthread"
end
puts "Immer noch im Hauptthread"
sleep 10
puts "Ende"
#=> Im Hauptthread
#=> Immer noch im Hauptthread
#=> Im Nebenthread
```

```
#=> Ende
```

Wenn die letzte Anweisung des Haupt-Threads ausgeführt wurde, beendet der Ruby-Interpreter alle anderen Threads zwangsweise. Will man dagegen auf die Beendigung eines Threads warten, so muß man Thread#join einsetzen. Die Methode blockiert den Programmfluß solange, bis der entsprechende Thread beendet wurde.

```
puts "Im Hauptthread"
t = Thread.new do
  sleep 3
  puts "Im Nebenthread"
end
puts "Immer noch im Hauptthread"
t.join # Ohne diese Zeile würde das puts im
       # Thread nie ausgeführt, weil der
       # Haupt-Thread vorher zu Ende ist.
```

#join gibt das Ergebnis des letzten Ausdrucks im Ziel-Thread zurück. Das kann man sich zu Nutze machen, um eine Anzahl Aufgaben parallel durchzuführen und dann die Ergebnisse zu sammeln. Ein Beispiel bildet das Herunterladen von Dateien. Wenn die Bandbreite des Internetanschlusses nur groß genug ist, dann ist das parallele Herunterladen mehrerer Dateien schneller als das sequentielle.

```
def download_file(url)
  # ... lädt Datei herunter ...
  # ... gibt die Größe zurück...
end

files = %w[http://... ... ...]
threads = files.map do |url|
  Thread.new do
    download_file(url)
  end
end

total = threads
  .map(&:join)
```

```
  .reduce(0){|sum, el| sum + el}
puts "Gesamt: #{total} Bytes"
```

Fallen während der Verarbeitung neue Aufgaben an, kann man ein Queue-Modell implementieren, das Aufgaben auf eine feste Anzahl an Arbeiter-Threads verteilt. Ruby bietet dafür in der Standardbibliothek die Klasse Queue an (→ [**TODO: Querverweis**]).

Mithilfe von Thread::kill kann man einen Thread vor Ende der regulären Laufzeit beenden. Da dies den Thread in einem undefinierten Zustand läßt, sollte man hiervon nur dann Gebrauch machen, wenn man von dem Thread kein Ergebnis mehr erwartet. Die Instanzmethode Thread#kill erreicht dasselbe.

```
t = Thread.new{sleep(5); puts("Hi")}
Thread.kill(t) # oder t.kill
sleep 10
# Nichts passiert
```

Rubys Threads sind seit Version 1.9 echte Threads auf Betriebssystemebene, d. h. sie werden in den Werkzeugen des Betriebssystems (etwa ps(1) unter unixoiden Systemen) entsprechend angezeigt. Daraus folgt, daß das Betriebssystem darüber (mit-)bestimmt, welcher Thread wann läuft. Bis einschließlich Version 1.8 implementierte Ruby dagegen dieses sog. **Scheduling** noch selbst. Man sprach von „green threads“ und bemängelte, daß Rubys Algorithmus zur Zeitverteilung nicht immer optimal und hinter denen der Betriebssysteme zurückblieb. Außerdem verhinderten „green threads“ die wirklich „gleichzeitige“ Ausführung von Code. Durch den Wechsel auf Betriebssystem-Threads ist diese Beschränkung zwar dem Grunde nach weggefallen, dennoch ist eine „echte“ Parallelität im Sinne einer wirklich zeitgleichen Ausführung von zwei oder mehr Anweisungen weiterhin nur beschränkt möglich. Hardware-bedingte Grundvoraussetzung für eine solche echte Parallität ist zunächst die Präsenz von wenigstens zwei Prozessorkernen (→3.1.1). Auf nur einem einzigen Kern kann man zwar weiterhin Threads einsetzen. Diese laufen aufgrund der Geschwindigkeit heutiger Prozessoren auch aus Nutzerperspektive quasi gleichzeitig, aber tatsächlich werden sie nur nur in schneller Folge abwechselnd abgearbeitet. Neben dieser hardware-seitigen Beschränkung, die für alle Programmiersprachen gleichermaßen gilt, gibt es noch eine andere, ruby-spezifische Beschränkung: den **Global VM Lock**, kurz **GVL**[1].

[1] Bis einschließlich Ruby 1.8 sprach man vom „Global Interpreter Lock“ (GIL). Die Bezeich-

Rubys kanonische Implementierung, der MRI, ist aus historischen Gründen an vielen Stellen nicht in der Lage, mit echter Parallelität mehrerer Anweisungen umzugehen. Um trotzdem definierte Ergebnisse zu erreichen, blockiert (engl. „lock“) die Ruby-VM die Ausführung anderer Threads des Programms während bestimmter (zahlreicher) Operationen. So wird erreicht, daß faktisch immer nur ein Thread eine Anweisung ausführt. Diese Beschränkung aufzuheben gehört zu den Daueraufgaben des Core-Teams, das damit Ruby-Version für Ruby-Version Fortschritte macht, d. h. mit jeder neuen Ruby-Version benötigen weniger Operationen den GVL. Voraussichtlich mit Ruby 3 wird er ganz abgebaut sein und damit echte Parallelität möglich. Bis dahin bleibt, wenn solche erforderlich ist, nur der Rückgriff auf andere Ruby-Implementationen als den MRI: weder Rubinius noch JRuby unterliegen einer derartigen Beschränkung, weil sie von Anfang an mit Blick auf echte Parallelität programmiert wurden. Bei diesen Ruby-Implementationen ist echte Parallelität damit nur durch die Hardware beschränkt.

Ruby bietet einige einschränkte Möglichkeiten, auf das Scheduling Einfluß zu nehmen. Die wichtigste ist die Kombination aus Thread::stop und Thread#run. Andere Methoden, auf den Scheduler Einfluß zu nehmen, existieren zwar, geben aber keine Erfolgsgarantie. Insoweit sei auf die Dokumentation von Thread verwiesen. Es ist daher empfehlenswert, sich nicht darauf zu verlassen, daß die eigenen Threads in einer bestimmten Reihenfolge zum Zuge kommen. Die Bedingungen, unter denen der Scheduler sich für den einen oder anderen Thread entscheidet (oder sogar, bei Zutreffen der genannten zwei Bedingungen, sich für eine gleichzeitige Ausführung entscheidet) sind zahlreich, betriebssystemspezifisch und auf Ruby-Ebene nicht vorhersehbar.

Mit Thread::stop wird der aktuelle Thread angehalten. Er wird aus der Liste ablaufbereiter Threads entfernt und wird nicht weiter abgearbeitet, bis er durch Aufruf von Thread#run wieder in Gang gesetzt wird. Diesbezüglich muß darauf hingewiesen werden, daß #run den Thread lediglich wieder in die Liste ablaufbereiter Threads eingepflegt. Er wird nicht notwendigerweise sofort abgearbeitet. Wann es dazu kommt, bestimmt der Scheduler nach seinen Regeln (s.o.).

```
t = Thread.new do
  puts "Im Nebenthread"
  Thread.stop
```

nung wird von langjährigen Rubyisten auch heute noch verwandt. Gemeint ist dasselbe Konzept.

```
  puts "Wieder im Nebenthread"
end

sleep 2
puts "Im Hauptthread"
t.run
t.join # Blockt, bis t fertig ist
#=> Im Nebenthread
#=> Im Hauptthread
#=> Wieder im Nebenthread
```

13.1.2 Geteilte Ressourcen und Synchronisation

Alle Threads teilen sich den globalen Zustand eines Ruby-Programms: sie haben gleichermaßen Zugriff auf alle globalen Variablen, auf alle Klassen und auf die Programmumgebung. Für den an Thread::new übergebenen Block gelten die üblichen Regeln: er hat Zugriff auf alles, was in seinem Gültigkeitsbereich liegt (Closure). Dieser geteilte globale Zustand ist Vor- und Nachteil von Threads zugleich: vorteilhaft ist, daß es auf diese Weise sehr einfach ist, zwischen verschiedenen, weitgehend gleichzeitig ablaufenden Teilen eines Programms Informationen auszutauschen; ein Zugriff auf eine mehreren Threads zur Verfügung stehende Variable genügt. Nachteilhaft ist dagegen, daß der umfangreiche, gleichberechtigte Zugriff Synchronisationsprobleme mit sich bringt. Was soll passieren, wenn ein Thread gerade mitten in einer Schreiboperation auf eine solche geteilte Ressource unterbrochen und an seiner Statt ein Thread ausgeführt wird, der von dieser Ressource liest? Der lesende Thread erhielte eine unvollständige Information. Weil nicht immer vorhersehbar ist, welcher Thread wann auf eine geteilte Ressource zugreift, müssen deshalb hierfür Kontrollmechanismen eingesetzt werden. Kommt es wegen mangelnden Einsatzes solcher Mechanismen zu einem unkontrollierten Zugriff auf die geteilte Ressource, spricht man von einer **Race Condition**, weil mehrere Threads „um die Wette“ fahren, wenn sie auf die geteilte Ressource zugreifen. Ein einfaches Beispiel einer solchen Race Condition sieht so aus:

```
str = ""
t1 = Thread.new{ 1000.times { str << "a" } }
t2 = Thread.new{ 1000.times { str << "b" } }
```

```
t1.join
t2.join
puts str
```

In welcher Reihenfolge »str« die Buchstaben »a« und »b« enthalten wird, ist in diesem Beispiel unvorhersehbar, weil der Zugriff auf die geteilte Ressource »str« nicht kontrolliert wird. Während man das in diesem Beispiel recht einfach sehen kann, sind Race Conditions in der Praxis weitaus schwieriger zu erkennen. Durch Race Conditions eingeführte Fehler gehören wegen ihrer Unvorhersehbarkeit zu den besonders schwierig zu behebenden Fehlern, weil ihre Diagnose Schwierigkeiten bereitet. So kann es passieren, daß das Programm ohne erkennbaren Grund manchmal korrekte und manchmal fehlerhafte Ergebnisse liefert.

Man sollte sich beim Einsatz von Threads also von vornherein Gedanken darüber machen, welche Ressourcen mehreren Threads zur Verfügung stehen müssen. Je weniger das sind, desto besser. Doch läßt sich das oft nicht vermeiden, sodaß man auf die angesprochenen Kontrollmechanismen zurückgreifen muß. Hierfür bietet Ruby Mutexen und Condition-Variables.

13.1.2.1 Mutexen

Die Funktionsweise einer **Mutex** läßt sich am besten anhand eines Vergleichs darstellen. Angenommen, jemand ist Eigentümer eines Gartens mit einem Geräteschuppen. Dieser Geräteschuppen enthält verschiedene Materialien zur Gartenarbeit, allerdings ist er so schmal, daß nur eine Person gleichzeitig hineingeht. Wenn sich eine zweite Person dazuquetschte, würde alles umfallen und durcheinander geraten. Um das zu verhindern, schließt man, solange man nach dem richtigen Werkzeug (z. B. einer Schaufel) sucht, die Schuppentür von innen ab. Hat man das Werkzeug gefunden, öffnet man die Tür wieder und geht hinaus. Wenn jemand gewartet hat, kann er (erst) jetzt in den Schuppen. Dennoch wurde während der Suche nach der Schaufel die Gartenarbeit aller anderen Personen im Garten nicht aufgehalten (solange sie eben nicht in den Schuppen mußten).

Die Schuppentür bzw. ihr Schloß in diesem Beispiel ist die Mutex, die arbeitenden Personen im Garten die Threads, der Geräteschuppen eine geteilte Ressource. Eine Mutex[2] kann nur von einem einzigen Thread aktiviert (geschlossen) werden. Solange dieser Thread die Mutex hält, müssen alle anderen Threads, die ebenfalls gerade diese Mutex haben müssen, warten. Mute-

[2] Der Begriff ist ein Kofferwort aus. engl. „mutual exclusion“, also Ausschließlichkeit.

xen können damit genutzt werden, um mehrere Threads aufeinander abzustimmen oder beim Zugriff auf geteilte Ressourcen sicherzustellen, daß niemals mehr als ein Thread auf einmal auf eine Ressource zugreift. Das obige Beispiel sieht als Ruby-Code dann so aus:

```
hut = %w[Schaufel Harke Dünger]
mutex = Thread::Mutex.new

t1 = Thread.new do
  mutex.synchronize do
    sleep 5 # Simuliere Arbeit
    puts hut.index("Schaufel")
  end
end

sleep 1

t2 = Thread.new do
  mutex.synchronize do
    puts hut.index("Harke")
  end
end

t1.join
t2.join
#=> 0
#=> 1
```

»t1« bekommt in diesem Beispiel zuerst Zugriff auf die Mutex und schließt sie durch Aufruf von Mutex#synchronize. Diese Methode gibt die Mutex erst mit Ende des übergebenen Blocks wieder frei. Das hier künstlich eingefügte #sleep für 5 Sekunden verhindert, daß »t2« die Mutex erhalten kann, wenn dieser Thread seinerseits #synchronize aufruft. Mutex#synchronize blockt den Programmfluß solange, bis der Thread, der die Mutex geschlossen hat, diese wieder öffnet. Wenn mehrere Threads auf die Mutex warten, kann aus Sicht des Ruby-Programmierers nicht vorhergesagt werden, welcher Thread zuerst zum Zuge kommt; die anderen müssen weiter warten.

Auch das obige Beispiel mit der Race Condition kann man durch Einsatz einer Mutex in eine vorhersehbare Ausführungsreihenfolge überführen:

```
str = ""
m = Mutex.new
t1 = Thread.new do
  m.synchronize { 1000.times { str << "a" } }
end
t2 = Thread.new do
   m.synchronize { 1000.times { str << "b" } }
end
t1.join
t2.join
puts str
```

Auf diese Weise bleibt zwar immer noch offen, ob zuerst 1000 Buchstaben »a« oder 1000 Buchstaben »b« an »str« angefügt werden, aber jedenfalls ist garantiert, daß es nicht mehr durcheinander geschieht.

Thread::Mutex bietet neben der Blockmethode #synchronize auch noch die zwei ausdrücklichen Methoden #lock und #unlock an, bei denen der Programmierer aber selbst darauf Acht geben muß, die geschlossene Mutex wieder zu öffnen, widrigenfalls es zu einem sog. **Deadlock** kommt. Dabei warten mehrere Threads auf die Mutex, die aber kein Thread mehr freigibt. Das Programm wäre in einem dauerhaften Wartezustand gefangen. Ruby hat als Gegenmaßnahme einen Detektor für solche Probleme, der einfache Deadlock-Situationen erkennt und das Programm beendet. Zur Illustration diene folgender Beispielcode:

```
m = Mutex.new
t = Thread.new do
  sleep 1
  m.lock # t kann die Mutex nicht mehr haben...
end
m.lock # ...weil der Haupt-Thread sie nicht hergibt.
t.join
# => No live threads left. Deadlock? (fatal)
```

Dieser Schutzmechanismus greift aber schon bei komplexeren Situationen nicht mehr zuverlässig ein. Daher sollte, wenn möglich, #synchronize der Vorzug eingeräumt werden. Auf diese Weise kann man zwar immer noch ein Deadlock herbeiführen, aber jedenfalls nicht mehr durch einen vergessenen

Aufruf von #unlock.

Mit Thread::Mutex#try_lock steht schließlich noch eine Methode zur Verfügung, die bei einem vergeblichen Zugriff auf die Mutex nicht wie #lock und #synchronize wartet, sondern sofort false zurückgibt. Gelingt der Zugriff dagegen — kann also die Mutex geschlossen werden —, dann gibt die Methode true zurück. Diese Methode sollte dann eingesetzt werden, wenn der Thread in der Zeit, in der er sonst auf die Mutex warten würde, noch etwas anderes sinnvolles erledigen kann.

13.1.2.2 Condition-Variables

Condition-Variables stellen ein Supplement zu Mutexen dar. Normalerweise ist der Programmierer gut beraten, die durch Mutexen begrenzten Regionen ausschließlichen Zugriffs so klein wie möglich zu halten, damit andere Threads nicht unnötig aufgehalten werden. Nicht immer ist das aber in dieser Idealform möglich. Eine Condition-Variable ermöglicht in diesen (und anderen) Fällen eine präzise(re) Steuerung des Programmflusses. Dies funktioniert so, daß ein Thread zunächst ausdrücklich eine Condition-Variable anfordert; er *wartet*. Die dazu genutzte Methode ConditionVariable#wait gibt die übergebene Mutex frei. Ein anderer Thread muß die Condition-Variable dann signalisieren; dafür wird ConditionVariable#signal oder ConditionVariable#broadcast verwandt. Erstere Methode weckt den ersten Thread, der auf die Condition-Variable gewartet hat auf, was diesen veranlaßt, sogleich die Mutex zu schließen und dann die Programmfluß fortzusetzen. Letztere weckt alle Threads auf, die auf eine Condition-Variable warten (allerdings kann natürlich immer noch nur ein Thread die Mutex schließen — die anderen Threads warten dann nicht mehr auf die Condition-Variable, sondern auf die Mutex).

Anwendungsbeispiele für Condition-Variables sind selten. Wenn ein Programm die dafür notwendige Komplexität erreicht, greift man meist gleich ganz auf eine Programmbibliothek zurück, die die lästige manuelle Verwaltung von Threads vor dem Programmierer versteckt. Eine davon, Eventmachine, wird im dritten Teil dieses Tutoriums vorgestellt (**[TODO: Querverweis]**). Nachfolgend wird das Beispiel aus Rubys Dokumentation wiedergegeben.

```
mutex = Mutex.new
resource = ConditionVariable.new
```

```
a = Thread.new {
    mutex.synchronize {
      # Thread 'a' benötigt die Ressource
      resource.wait(mutex)
      # 'a' kann die Ressource jetzt nutzen
    }
}

b = Thread.new {
    mutex.synchronize {
      # Thread 'b' ist fertig mit der Ressource
      resource.signal
    }
}
```

13.1.3 Thread-lokale Variablen

Ein eher ungewöhnliches Feature von Ruby sind thread-lokale Variablen. Diese wurden mit Version 1.9 allerdings unter Änderung der dafür bislang benützten Methode Thread::[] von den (nicht minder ungewöhnlichen) fiberlokalen Variablen (**[TODO: Querverweis]**) weitgehend abgelöst. Verblieben für thread-lokale Variablen sind die Methoden Thread#thread_variable_get und Thread#thread_variable_set.

13.2 Fibers

13.3 Prozesse

Wiederholungsfragen und -aufgaben

1. Folgendes Programm enthält eine Race Condition. Finden und beheben Sie sie.

   [TODO: Programm schreiben]

2. Was ist ein Deadlock?

Teil III

Drittbibliotheken

§ 14 Einführung

[TODO: Schreiben: stdlib, RubyGems, RDoc]

§ 15 Standardbibliothek

[TODO: Schreiben. Verschiedene wichtige Libs der stdlib vorstellen.]

Anhang

Literatur

[1] Leslie Lamport. “If You’re not Writing a Program, Don’t Use a Programming Language”. british. In: *Bulletin of EATCS* 125 (2018). URL: http://bulletin.eatcs.org/index.php/beatcs/article/view/539 (besucht am 28. 07. 2018).

[2] *Pro Git*. Amerikanisch. 2. Aufl. Apress, 2014. URL: https://git-scm.com/book/ (besucht am 27. 08. 2019).