

# 1 Ruby stellt sich vor

*For an instant I thought Ruby was reading my mind . . .  
if matz can do that, it IS a powerful language!  
Hal Fulton*



Dieses Kapitel enthält neben nicht technischen Hintergrundinformationen zu Ruby einige Codebeispiele aus unterschiedlichen Anwendungsgebieten, die als Appetithäppchen gedacht sind und zeigen sollen, wie kompakt und elegant Ruby-Code sein kann. Abschließend stellen wir die wichtigsten Codekonventionen vor, die in der Ruby-Gemeinschaft Anwendung finden.

## 1.1 Über Ruby

Was ist Ruby? Die kurze Antwort lautet: Ruby ist eine sehr mächtige objektorientierte Skriptsprache mit einer einfachen Syntax. Nun etwas länger: Man nehme das Beste von Smalltalk, Perl, Python, Eiffel, Scheme, CLU und Lisp und mische mehrmals stark.

*Ruby > (Smalltalk +  
Perl)/2*

Ruby wurde (und wird noch) von Yukihiro »matz« Matsumoto entwickelt und ist als Open Source frei verfügbar. Die Sprache ist sehr vielseitig einsetzbar: von einfachen Skripten zum Filtern von Textdateien bis zum kompletten OO-Programm, von grafischen Frontends bis hin zu Webanwendungen. Und in seiner japanischen Heimat ist Ruby schon populärer als Python.

Ruby zieht Programmierer aus den verschiedensten Gründen an. Die meisten erfreuen sich an Rubys elegantem Aufbau, seiner sauberen Syntax, den mächtigen eingebauten Funktionen oder den aus Smalltalk stammenden Iteratoren. Andere sind davon begeistert, wie schnell man GTK+-Anwendungen schreiben kann.

Manche kommen aus der Perl-Welt, weil Klassen und Objekte in Ruby viel natürlicher zu bearbeiten sind als in Perl und die Programme übersichtlich und gut lesbar bleiben. Der Umstieg ist einfach, weil viele syntaktische Elemente von Perl übernommen wur-

den. Durch die konsequente Anwendung objektorientierter Prinzipien sind sie in Ruby oft mächtiger geworden. Ein starkes Plus ist auch die Unterstützung von internationalen Zeichensätzen, insbesondere der japanischen Schrift.

Ruby ist eine gut ausbalancierte und natürlich wirkende Sprache, da matz stets bemüht ist, folgende Eigenschaften von Ruby herauszuarbeiten:

*least surprise !=  
no surprise*

- ❑ **konsequent:** Eine kleine Menge von Regeln sollte die ganze Sprache definieren. Ruby vertritt das Prinzip der kleinsten Überraschung und der geringsten Anstrengung.
- ❑ **kompakt:** Es gibt keine überflüssigen Sprachelemente. Der Autor von Perl, Larry Wall, meint, dass Ruby deshalb weniger »ausdrucksstark« ist. Man bedenke jedoch auch, dass Rubine (in der Regel) klar und Perlen undurchsichtig sind.
- ❑ **flexibel:** Eine Sprache sollte genug Spielraum bieten, so dass man alles mit ihr machen kann. Dabei sollten einfache Aufgaben auch einfach zu lösen sein und komplexe Probleme zumindest überhaupt lösbar sein. matz meint jedoch, dass auf Grund der guten OO-Eigenschaften von Ruby schwierige Probleme nicht unbedingt schwer zu lösen sein müssen.
- ❑ **anpassungsfähig:** Computersprachen sollen Menschen helfen, sich auf das Problem zu konzentrieren und nicht mit der Sprache zu kämpfen, nur weil sich ein Algorithmus in eben dieser nicht adäquat formulieren lässt.

*i - )*

matz hat sich beim Design von Ruby von folgenden Grundregeln leiten lassen:

- ❑ **Lerne von so vielen Sprachen wie nur möglich.**  
Man muss nicht alles selbst erfinden und kann sich von anderen Sprachdesignern inspirieren lassen.
- ❑ **Höre auf dein Herz.**  
Der Autor ist der erste Benutzer der Sprache. Wenn sie für seine Zwecke passend gestaltet ist, wird sie wahrscheinlich auch für andere nützlich sein.
- ❑ **Sei nicht zu mutig.**  
Wenn man will, dass eine Sprache den größtmöglichen Verbreitungsgrad findet, sollte man keine Extravaganzen einbauen. Natürlich muss man ein paar neue Dinge in einer Sprache einführen, aber es sollten nicht zu viele sein.

Zudem ist Ruby sehr portabel und auf den unterschiedlichsten Rechnerplattformen einsetzbar. Die dynamische Typisierung trägt dazu bei, dass Entwürfe mit einem minimalen Aufwand erstellt werden können und bei Bedarf auch zu größeren Programmen ausgebaut werden können, ohne dass sprachbedingte Hürden zu überwinden sind.

Da Ruby Perl ähnelt, wollte matz auch den Namen eines Juwels wählen und entschied sich für den Geburtsstein eines Kollegen. Erst später bemerkte matz, dass der Rubin (er steht für Juli) der Nachfolger (!) von Perl (Juni) ist.

;-)

Das Ergebnis ist eine Programmiersprache, die leicht zu erlernen ist und in der man gerne programmiert.

## 1.2 Appetithappen

Die folgenden Appetithappen sollen einen Eindruck davon vermitteln, wie sich Ruby-Code »anfühlt«. Im Anschluss (Abschnitt 1.3) fassen wir die wichtigsten Codekonventionen und Syntaxregeln für das Arbeiten mit Ruby zusammen.

Auch wenn die Anweisungen erst später im Buch erklärt werden, kann man beim Durchlesen oft schon erraten, was mit dem Code bezweckt wird. Noch mehr wird man allerdings von unseren Häppchen haben, wenn man die Beispiele auf einem Rechner ausprobiert, was wir im nächsten Kapitel beschreiben. Zusätzlich zu den lokalen Testmöglichkeiten gibt es noch Clemens' RubyChannel, eine Website, auf der ein Ruby-Interpreter zur Verfügung steht, mit dem viele Codebeispiele getestet werden können.

[www.ruby.ch](http://www.ruby.ch)

- Obligatorisches Kultprogramm (Unix-Syntax):

*Hello World!*

```
ruby -e 'puts "Hallo zukuenftiger Rubyist!"'
```

- Mehrfachzuweisungen für Vertauschungen:

*Intuitive Syntax*

```
x, y = 2, 3          # x == 2 und y == 3
x, y = y, x          # x == 3 und y == 2
```

- Primzahlen finden; der Algorithmus ist langsam, aber kurz.

*Kompakter Code*

```
for i in 2..100 do
  puts i unless (2..i-1).find{|j| i % j == 0}
end
#-> 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
     53 59 61 67 71 73 79 83 89 97
```

*Beliebig genaue  
Ganzzahlarithmetik*

- **Beliebig große ganze Zahlen sind eingebaut und werden automatisch verwendet.**

```
def fak(n)
  f = 1
  (2..n).each { |i| f *= i }
  f
end
puts fak(gets.to_i)

# Eingabe: 100
#-> 93326215443944152681699238856...
```

*Nützliche Einzeiler*

- **Wir haben einige nützliche Einzeiler zusammengestellt, die in der gezeigten Schreibweise auf einem Unix-Rechner funktionieren. In einer DOS-Konsole müssen die dort gültigen Syntaxregeln beachtet werden.**

```
# Zeilen zählen
ruby -e 'print readlines.length'

# ersetze foo durch bar in C-Dateien
# Sicherungskopie in .bak-Dateien
ruby -i.bak -pe 'gsub "foo", "bar"' *.ch]

# Zeilen 'in place' nummerieren
ruby -pi -e 'printf "%d\t", $.;' thgTTg.txt
```

*Flexible Arrays*

- **Arrays und Hashtabellen haben in Ruby dynamische Größen und können Elemente verschiedener Typen aufnehmen.**

```
array = [1, 'Alligator', 643.001]
pcSpeedHash = {
  'Duck2'=>750, 'Rockaplan'=>25, 'ZX81'=>0.6 }
pcSpeedHash['Ananke']=1700
```

*Iteratoren*

- **Iterieren über Aufzählungen aller Art.**

```
# Bereiche
(5..10).each { |z| puts z**3 }
#-> 125 216 343 512 729 1000
```

```
# Hashtabellen
pcSpeedHash.sort.map{ |key, val|
  "#{key}: #{val} MHz"
} #-> ['Ananke: 1700 Mhz', 'Duck2: 750 MHz',
      'Rockaplan: 25 MHz', 'ZX81: 0.6 MHz']
```

- **Worthäufigkeiten zählen.**

*Flexible Zuweisungen*

```
freq = Hash.new(0)
open("dings.rb").read.scan(/\w+/{ |wort|
  freq[word] += 1
}
freq.sort.each { |wort, zahl|
  puts "#{wort} - #{zahl}"
}
```

- **Reguläre Ausdrücke in ganzer Pracht. Wir hatten eine Liste von Zitaten in der Form »Autor: Zitat« und erzeugen daraus die L<sup>A</sup>T<sub>E</sub>X-Anweisungen, die wir für dieses Buch verwenden.**

*Reguläre Ausdrücke*

```
readlines.each{ | line |
  puts line.gsub('(.*)?: (.*)',
    '\\begin{rquote}{\1}\2\\end{rquote}')
```

- **»Die Stringmethode hatte doch ein 'case' im Namen ...«**

*Reflexion ist eingebaut*

```
".methods.grep(/case/)
#-> ["upcase!", "upcase", "swapcase!", ...]
```

...»und wie hieß gleich noch mal die Error-Klasse?«

```
ObjectSpace.each_object(Class) { |c|
  puts c if c.to_s[/error/i]
} #-> SystemStackError, LocalJumpError,
      EOFError, IOError, RegexpError, ...
```

- **Ruby ist eine dynamische Sprache! Methoden und Variablen können während der Laufzeit hinzugefügt und geändert werden, sogar für einzelne Objekte:**

*Singleton-Methoden*

```

class A
  def method_missing(methId)
    print methId.id2name, " gibt es nicht.\n"
  end
end
a=A.new
a.hello           #-> hello gibt es nicht.

# diese Singleton-Methode gibt es nur für a
def a.hello
  p "hier schon"
end
a.hello           #-> hier schon
A.new.hello       #-> hello gibt es nicht.

```

*Lesbare Syntax*

- **Quicksort von Tony Hoare in Ruby:**

```

def quicksort( xs )
  return xs if xs.size <= 1
  m = xs[0] # Split-Element
  quicksort(xs.select { |i| i < m } ) +
  xs.select { |i| i == m } +
  quicksort(xs.select { |i| i > m } )
end
quicksort([13, 11, 74, 69, 0])
#-> [0, 11, 13, 69, 74]

```

*Netzwerkzugriff*

- **Netzwerkzugriff über Standardprotokolle, z. B. per ftp:**

```

require 'net/ftp'
ftp = Net::FTP.open('ftp.netlab.co.jp')
ftp.login
ftp.chdir('pub/lang/ruby')
puts ftp.dir
ftp.quit

```

*Threads – auch unter  
DOS*

- **Threads stehen ebenfalls zur Verfügung, um Aufgaben parallel zu erledigen. Dieses Beispiel holt zwei Webseiten gleichzeitig:**

```

require 'net/http'
url1="www.approximity.com"
url2="www.ruby.ch"

```

```

def getPage(url)
  h = Net::HTTP.new(url, 80)
  puts "Hole: #{url}"
  resp, data = h.get('/', nil )
  puts "Habe #{url}:  #{resp.message}"
end

thread1=Thread.new { getPage(url1) }
thread2=Thread.new { getPage(url2) }

thread1.join
thread2.join

```

- Mit dem so genannten **Marshalling** können in Ruby fast alle Objekte in Binärstreams/Strings transformiert werden. Diese werden dann in Instanzen der Klasse String gespeichert.

*Marshalling*

```

include Marshal
a = 25.6;
pt = Struct.new('Point', :x,:y);
x = pt.new(10, 10)
y = pt.new(20, 20)
rt = Struct.new('Rectangle', :origin,:corner);
z = rt.new(x, y)
c = Object.new
thing = [a, x, z, c, c, "fff"];
representation = dump(thing);

p thing
p representation
p load(representation)

```

- **Client/Server-Kommunikation ist mit Distributed Ruby kein Problem.** Ein einfacher Server ist schnell geschrieben:

*Verteiltes Ruby*

```

# server.rb
require 'drb'
class TestServer
  def doit; "Hallo, verteilte Welt"; end
end
ts = TestServer.new

```

```

DRb.start_service(
  'druby://approximity.com:9000', ts)
DRb.thread.join

# client.rb
require 'drb'
DRb.start_service
cl = DRbObject.new(nil,
  'druby://approximity.com:9000')
p cl.doit

```

Wir haben den TestServer auf `approximity.com` eingerichtet, so dass nach dem Start des Clients der Text »Hallo, verteilte Welt« erscheinen sollte.

*Closures*

- So genannte Closures (siehe Abschnitt 9.4) binden die Umgebung, in der ihre Definition ausgeführt wird.

```

def addierer(incr); proc { |n| n + incr }; end
ad = addierer(3)
p ad.call(5)      # => 8

```

- Wenn man sich wirklich, wirklich Mühe gibt, kann man auch kryptisches Ruby erzeugen. Das folgende Beispiel sollte man genau so eingeben, wie es hier gezeigt ist:

```

# ruby.rb
(n=${0}.to_i).times{|i|s=(n/2-i).abs;puts" "*s+"#"(n-2*s)}

# Konsole:
> ruby ruby.rb 10

```

- Wir haben die Häppchen mit »Hallo Welt« begonnen. Es gibt noch ein weiteres Programm [Bottles], das anscheinend in jeder Sprache programmiert worden ist:

*Ex und hopp!*

```

99.downto(0) { |x|
  u = "#{x > 0 ? (x) : 'No more'}" +
    " bottle#{x != 1 ? 's' : ''} of beer"
  w = u + " on the wall, "
  puts w + u +
    ". Take one down, pass it around. " + w
}

```

Man kann die Flaschen auch objektorientiert leeren:

*Für den Connaisseur*

```
class Wall
  def initialize num; @beers = num; end
  def beer?; @beers > 0; end
  def takeBottle; @beers -= 1; end
  def bottles
    @beers == 1 ? "bottle" : "bottles"
  end
  def count
    @beers > 0 ? @beers.to_s : "No more"
  end
  def bob
    "#{count} #{bottles} of beer"
  end
end

wall = Wall.new 99
while wall.beer?
  print "#{wall.bob} on the wall, " +
    "#{wall.bob}. Take one down, "
  wall.takeBottle
  puts "pass it around. " +
    "#{wall.bob} on the wall."
end
```

## 1.3 Codekonventionen und grundlegende Syntax

Die folgenden Informationen sollte man beim Lesen und Schreiben von Ruby-Programmen im Hinterkopf behalten:

- Anweisungen werden durch ein Semikolon (;) voneinander getrennt. Wenn eine Zeile nur eine Anweisung enthält, kann das abschließende »;« entfallen.
- Wenn Argumente bei Methodenaufrufen in Klammern gesetzt werden, schließt die öffnende Klammer unmittelbar an den Methodennamen an.

```
puts ("hallo") # nicht gut
```

```
puts("hallo") # ok
puts "hallo"  # ok
```

- ❑ Operatoren werden (wegen der besseren Lesbarkeit) in der Regel von Leerzeichen umgeben.
- ❑ Der Operator = ist eine syntaktische Alternative für den Aufruf von Setter-Methoden, die Zuweisungen im weitesten Sinne vornehmen, siehe Abschnitt 15.2.
- ❑ Das Suffix ? zeigt an, dass die Methode eine Ja-/Nein-Frage beantwortet und folglich entweder true oder false zurückliefert.

```
0.zero?           #-> true
1.zero?           #-> false
(0.3 - 0.1 - 0.2).zero? #-> false
0.3 - 0.1 - 0.2   #-> -2.775557562e-17
```

- ❑ Ein abschließendes ! zeigt an, dass die Methode »gefährlich« ist, weil sie zum Beispiel ein bestehendes Array modifiziert, ohne vorher eine Kopie zu erzeugen.

```
x = ['Stefan', 'Armin', 'Clemens']
x.sort #-> ["Armin", "Clemens", "Stefan"]
x      #-> ["Stefan", "Armin", "Clemens"]
x.sort! #-> ["Armin", "Clemens", "Stefan"]
x      #-> ["Armin", "Clemens", "Stefan"]
```



Wir verstoßen bei unseren Codebeispielen gelegentlich aus layout-technischen Gründen gegen den guten Geschmack und packen mehrere Befehle in eine Zeile. Manchmal »mogeln« wir auch, indem wir den Code etwas kleiner schreiben und sogar in den Rand hinein ragen lassen, wenn wir keine zufrieden stellende Möglichkeit gefunden haben, einen Zeilenumbruch einzufügen. 