Ruby Grundlagen

PDF zum Buch

Rapid Web Development mit Ruby on Rails

Ralf Wirdemann und Thomas Baustert
www.b-simple.de
Hamburg

22. Juni 2006

Inhaltsverzeichnis

Kapitel 1

Ruby Grundlagen

Dieses PDF dient als Ergänzung zum Buch *Rapid Web Development mit Ruby on Rails, Wirdemann, Baustert, Hanser 2006*. Es führt in die Programmierung mit der Sprache Ruby ein. Es stellt keine allumfassende Beschreibung der Sprache dar, das vermittelte Wissen schafft aber eine solide Basis und ist mehr als ausreichend, um Ruby (on Rails) Programme zu verstehen und zu schreiben. Die vorliegende Beschreibung basiert auf der Ruby Version 1.8.2.

1.1 Online-Dokumentation und Bücher

Informationen zur Sprache sowie Dokumentation der Core- und Standard-API finden Sie unter *www.ruby-lang.org*. Für einen tieferen Einstieg empfehlen wir Ihnen die folgenden Bücher:

- *Dave Thomas: Programming Ruby*, Second Edition, Pragmatic Bookshelf, 2005.
  Das Standardwerk unter den Ruby-Bücher und uneingeschränkt empfehlenswert.
- *Hal Fulton: The Ruby Way*, Sams, 2001
  Das Buch ist nicht mehr auf dem neuesten Stand der Ruby Sprache, bietet aber über 100 Lösungsbeispiele und ist daher interessant.

1.2 Einführung

Ruby ist eine reine objektorientierte, dynamisch typisierte Sprache. Ruby Programme werden nicht (wie z.B. in Java) in ein Binärformat übersetzt, sondern direkt von einem Interpreter verarbeitet. Die Sprache wurde bereits 1995 von Yukihiro Matsumoto veröffentlicht und ist neben Smalltalk, Python, u.a. vor allem durch Perl beeinflusst.

Alles in Ruby ist ein Objekt, es gibt keine primitiven Typen (wie z.B. in Java). Ruby bietet neben der Objektorientierung unter anderem Garbage Collection, Ausnahmen (Exceptions), Reguläre Ausdrücke, Introspektion, Code-Blöcke als Parameter für Iteratoren und Methoden, die Erweiterung von Klassen zur Laufzeit, Threads und vieles mehr. Ruby-Programme sind aufgrund ihrer Einfachheit und klaren Syntax leicht zu verstehen und zu warten.

1.3 Ruby-Programme

Ruby-Programme werden in Dateien mit der Endung `.rb` gespeichert. Programme können Klassen (vgl. 1.15), Module (vgl. 1.16) oder einfach nur Ruby-Code enthalten. Im Folgenden ist das allseits bekannte *Hello World*-Programm angegeben:

```
# hello.rb
puts "Hello World!"
```

Wird der Code in einer Datei mit Namen `hello.rb` gespeichert, erfolgt der Aufruf wie folgt:

```
> ruby hello.rb
> Hello World!
```

Unter Windows können Sie ggf. über Dateiassoziationen die Ausführung über den Explorer erlauben. Unter Linux/Unix können Sie die *Shebang* Zeile je nach Betriebssystem nutzen:

```
#!/usr/local/bin/ruby
puts "Hello World!"

#!/usr/bin/env ruby
puts "Hello World!"
```

Der Aufruf erfolgt dann direkt über das Programm:

```
> chmod 744 hello.rb
> ./hello.rb
```

Anweisungen in Ruby können mit einem Semikolon enden, müssen es aber nicht. Ruby ist eine zeilenorientierte Sprache, d.h. eine Anweisung endet ohne Semikolon am Ende der Zeile. Es sei denn, der Interpreter kann erkennen, dass die Anweisung auf der nächsten Zeile fortgeführt wird. Die Ausgaben der folgenden Anweisungen sind identisch. Die letzte Anweisung erzeugt einen Fehler:

```
puts "Hello World!";
puts "Hello World!"
puts "Hello" \
  " World!";
puts "Hello" +
  " World!";
```

```
puts "Hello"    # Die Anweisung ist hier beendet.
  + " World!";  # Eine neue Anweisung, die nicht
                # interpretiert werden kann
```

Mehrere Anweisungen können in einer Zeile durch das Semikolon getrennt angegeben werden. Dies ist aber eher unüblich, weil es den Lesefluss stört:

```
# geht, aber unschön
a = 42; b = 15; c = a + b

# besser
a = 42
b = 15
c = a + b
```

In Ruby hat sich eine Einrückung von zwei Leerzeichen (keine Tabs) durchgesetzt. Dies ist eher eine Konvention als eine Empfehlung und sollte daher tunlichst eingehalten werden:

```
# sehr gut eingerückt
while line = file.readline
  if !comment_line(line)
    lines.add(line)
  end
end

# oh, oh, da bekommen Sie mit der Community Ärger!
while line = file.readline
    if !comment_line(line)
        lines.add(line)
    end
end
```

Ruby bietet eine ganze Reihe von Standardtypen, wie Zahlen, Strings, reguläre Ausdrücke, Arrays, Hashes, usw. Alle diese Element werden über Klassen (vgl. 1.15) oder Module (vgl. 1.16) bereitgestellt, die nicht explizit in die Programme (die Dateien) eingebunden werden müssen. Sie stammen aus der Core-Bibliothek und stehen überall im Programm automatisch zur Verfügung.

Des weiteren wird über die Standard-Bibliothek eine Reihe weiterer Klassen und Module, wie z.B. `Date`, `Logger`, `Test::Unit` usw. bereitgestellt. Diese sowie selbst entwickelte müssen explizit über das Schlüsselwort `require` in jedes Programm eingebunden werden. Dazu ist der Name der Datei mit oder ohne Endung (`.rb`) hinter `require` anzugeben:

```
require 'date'      # date.rb mit Klasse Date
require 'my_class'  # my_class.rb mit Klasse MyClass
require 'my_module' # my_modul.rb mit Modul MyModul
```

Handelt es sich bei dem Namen nicht um einen absoluten Pfad, wird die Datei in allen Standard-Verzeichnissen von Ruby gesucht. Diese sind in der globalen

Variable `$:` enthalten[1]. Die Namen aller in einem Programm geladenen Klassen und Module können über die globale Variable `$"` ausgegeben werden.

1.4 Kommentare

Eine Kommentarzeile beginnt in Ruby mit einer Raute (`#`). Der Kommentar kann am Beginn oder Ende der Zeile stehen.

```
# die folgende Zeile ist auskommentiert:
# a = b - c
a = b + c # ein Kommentar am Ende
```

Ein Blockkommentar beginnt mit `=begin` und endet mit `=end`. Die Schlüsselwörter müssen ohne Leerzeichen am Beginn einer Zeile stehen:

```
=begin
  def my_method
    ...
  end
=end
```

1.5 Zahlen

Ruby unterstützt Ganz- und Fließkommazahlen. Da es in Ruby keine primitiven Typen gibt, sind alle Zahlen Objekte. Ganzzahlen im Bereich von -2^{30} bis $+2^{30}$ (bzw. -2^{62} bis $+2^{62}$ auf 64bit Maschinen) werden als Typ `FixNum` definiert, alle darüber hinaus sind vom Typ `BigNum`. Die Typzuordnung und -konvertierung erfolgt automatisch und die Größe einer Zahl ist letztlich nur durch den Hauptspeicher bestimmt:

```
value = 42 # FixNum
big_value = 123456789012345678901234567890 # BigNum
```

Zahlen können auch in den Zahlensystemen Hexadezimal, Oktal oder Binär angegeben werden:

```
# 42
0x2A
0052
b101010
```

Entsprechende mathematische Operatoren stehen zur Verfügung. Das Hoch- und Herunterzählen wird über die Operatoren += und −= ermöglicht. Die aus C und Java bekannten Operatoren ++ und −− gibt es in Ruby nicht:

```
a = 2
b = 3
```

[1] Die Pfade erhalten Sie u.a. über den Aufruf *ruby -e "puts $:"* auf der Kommandozeile

```
c = a + b   # 5
c = a - b   # -1
c = a / b   # 0
c = 2.0 / b # 0.666666666666667
c = a * b   # 6
c = a**b    # 2*2*2 = 8

a += 1      # a = 3
a -= 1      # a = 2
a++         # geht nicht in Ruby
```

`FixNum` und `BigNum` erben von der Basisklasse `Integer`. Diese stellt hilfreiche Methoden zur Verfügung, die mit Blöcken (siehe Abschnitt 1.14) kombiniert werden:

```
1.upto(3) { |i| puts i }    # 1 2 3
3.downto(1) { |i| puts i }  # 3 2 1
0.step(10,2) { |i| puts i } # 0 2 4 6 8 10
3.times { puts "42" }       # 42 42 42
```

Fließkommazahlen werden in Ruby durch die Klasse `Float` repräsentiert. Wie in allen Sprachen unterliegen Fließkommazahlen auch in Ruby dem Problem von Rundungsfehlern. Für exakte Berechnungen z.B. bzgl. Beträgen empfiehlt sich daher die Verwendung der Klasse `BigDecimal` aus der Ruby-Standard-Bibliothek. Diese kann gegenüber `Float` beliebig genaue Fließkommazahlen darstellen und umgeht das Problem von Rundungsfehlern.

1.6 Strings

Strings werden in Ruby innerhalb von einfachen oder doppelte Hochkommata gesetzt. Die Hochkommata können jeweils innerhalb der anderen vorkommen:

```
str = "Hallo"           # Hallo
str = "Hallo 'Thomas'"  # Hallo 'Thomas'
str = 'Hallo'           # Hallo
str = 'Hallo "Thomas"'  # Hallo "Thomas"
```

Strings können auch über die Literale `%q` und `%Q` erzeugt werden. Dies ist dann sinnvoll, wenn innerhalb des Strings viele Hochkommata oder andere Zeichen vorkommen, die andernfalls zu escapen wären. Durch `%q` wird ein String in einfachen Hochkommata und durch `%Q` ein String in doppelten Hochkommata eingeschlossen. Der Text wird bei der Definition durch Trennsymbole begrenzt, die bis auf alphanumerische Zeichen alle Zeichen annehmen können:

```
%q{ein String durch geschweiften Klammern begrenzt}
%q(ein String durch runde Klammern begrenzt)
%Q$ein String durch Dollarzeichen begrenzt$
```

Bei `%Q` werden Ausdrücke der Form `#{Ausdruck}` (s.u.) ersetzt, bei `%q` nicht:

Tabelle 1.1: Escape-Zeichen in Strings mit doppelten Hochkomma

\a	Klingelton	\s	Leerzeichen	\C-x	Control-x
\b	Backspace	\t	Tabulator	\M-x	Meta-x
\e	Escape	\v	Vertikaler Tabulator	\M-\C-x	Meta-Control-x
\f	Formfeed	\nnn	Octal nnn	\x	x
\n	Neue Zeile	\xnn	Hexadezimal xnn	#{code}	Code
\r	Return	\cx	Control-x		

```
puts %q{result: #{42.0/3} EUR}  # result: #{42.0/3} EUR
puts %Q{result: #{42.0/3} EUR}  # result: 14.0 EUR
```

Im Falle der geschweiften, runden und eckigen Klammern wird der String durch das Gegenstück der Klammer begrenzt. Bei allen anderen Zeichen durch das erneute Vorkommen des Zeichens. Der String kann auch über mehrere Zeilen angegeben werden, wobei Ruby die führenden Leerzeichen nicht löscht:

```
s = %Q@ein String über mehrere
  Zeile mit "" und '' und durch
    einen Klammeraffen begrenzt@
puts s
=>
ein String über mehrere
  Zeile mit "" und '' und durch
    einen Klammeraffen begrenzt

puts s.inspect
=>
"ein String \374ber mehrere\n  Zeile mit \"\" und '' ...
... und durch \n    einen Klammeraffen begrenzt"
```

Das Ergebnis eines Ausdruck kann über `#{Ausdruck}` in einen String eingefügt werden. Das funktioniert jedoch nur innerhalb doppelter Hochkommata:

```
"Ergebnis #{42*7/100} %"       # Ergebnis #{2.94} %
"Die Anwort ist: #{answer(x)}" # Die Anwort ist: 42
```

Sonderzeichen werden, wie auch aus C und Java bekannt, mit einem Slash escaped. Eine Liste aller Sonderzeichen ist in Tabelle 1.1 aufgeführt:

```
"Ein Slash: \\"      # Ein Slash: \
"Er rief: \"Ralf!\"" # Er rief: "Ralf!"
'War\'s okay?'       # War's okay?
"Neue\nZeile"        # Neue
                     # Zeile
'Neue\nZeile'        # Neue\nZeile
```

Die Inhalte zweier Strings werden über die Methode == verglichen. Hingegen vergleicht die Methode `equal?`, ob es sich um dieselbe String-Instanz handelt (vgl. Abschnitt 1.15.5):

```
s1 = "Thomas"
s2 = "Thomas"
s3 = "Ralf"
s1 == s2      # => true
s1 == s3      # => false
s1.equal? s1  # => true
s1.equal? s2  # => false
s1.equal? s3  # => false
```

Strings können über die Methoden + und << konkateniert werden. Durch die Verwendung von * ist auch eine Multiplizierung möglich:

```
"Thomas" + "/Ralf"  # Thomas/Ralf
s = "Thomas"
s << " und Ralf"    # Thomas und Ralf
"Ralf " * 2         # Ralf Ralf
```

Die Klasse String bietet eine Fülle von Methoden, die keine Wünsche lassen. Im Folgenden sind einige exemplarisch aufgeführt:

```
s = "Thomas und Ralf"
s[3]      # 109
s[3].chr  # m
s[7,3]    # und
s[0..6]   # Thomas

"Thomas und Ralf".delete("a")         # Thoms und Rlf
"Thomas und Ralf".delete("aou")       # Thms nd Rlf

"Thomas und Ralf".gsub("und", "oder") # Thomas oder Ralf
"Thomas und Ralf".gsub(/[aou]/, "$")  # Th$m$s $nd R$lf

"Thomas und Ralf".index('a')          # 4
"Thomas und Ralf".index('a',5)        # 12

"Thomas und Ralf".split     # ["Thomas", "und", "Ralf"]
```

Für die Konvertierung von Strings in eine Ganz- oder Fließkommazahl gibt es zwei Möglichkeiten. Sicherer ist die Verwendung der Kernel-Methoden `Integer` und `Float`, da diese im Fehlerfall eine Ausnahme werfen. Alternativ bietet die Klasse `String` die Methoden `to_i` und `to_f`, die weniger restriktiv sind:

```
 "42".to_i      # => 42
  nil.to_i      # =>  0
"42x".to_i      # => 42
Integer("42")   # => 42
Integer(nil)    # =>  0
Integer("42x")  # => ArgumentError
```

1.7 Bereiche

Ruby bietet durch die Klasse `Range` die Definition von Bereichen. Ein Bereich wird durch einen Start- und Endwert definiert, aus denen sich auch die Zwischenwerte ergeben. Je nach Angabe von zwei oder drei Punkten zwischen den Werten, ist der Endwert im Bereich enthalten oder nicht:

```
inclusive = (1..10)    # 1, 2, 3, ..., 10
exclusive = (1...10)   # 1, 2, 3, ..., 9
letters = ('a'..'z')   # a, b, c, ..., z
words = ('aaa'..'aaz') # aaa, aab, aac, ..., aaz
```

Über die Methode === wird geprüft, ob ein Wert innerhalb des Bereichs liegt:

```
letters = ('a'..'z')
letters === 'k'   # true
letters === '@'   # false

values = -12.5..+12.5
values === -7.45  # true
values === 42     # false
```

Bereiche kommen in Schleifen vor, können innerhalb von Bedingungen angegeben werden oder finden als Index in Arrays Verwendung:

```
for i in (1..42) do
  puts i
end

while c = gets
  case c
  when '0'..'9' then puts "Ziffer"
  when 'a'..'z' then puts "Buchstabe"
  end
end

a = [1, 2, 3, 4, 5, 6]
puts a[1..3]  # 2 3 4
puts a[1...3] # 2 3
```

1.8 Variablen und Konstanten

In Ruby enthält jede Variable eine Referenz auf ein Objekt. Ein nicht initialisierte Variable liefert `nil`. Da jede Variable eine Referenz auf ein Objekt ist, erfolgt die Übergabe von Objekten an Methoden per Referenz und nicht als Wert.

Lokale Variablen müssen bei ihrer Definition initialisiert werden. Sie sind nur innerhalb des umschließenden Blocks (Programm, Methode, Block) gültig:

```
def add(a, b)
```

Tabelle 1.2: Namensbeispiele

Variablen, Methoden und Parameter	Globale Variablen	Konstanten, Klassen- und Modulnamen	Instanz- und Klassenvariablen
length	$Logger	Maxlng, MAXLNG	@name
calculate_rate	$context	ProjectTask	@@counter

```
  c = a + b
end

puts c  # Fehler, c unbekannt
```

Globale Konstanten werden außerhalb eines Moduls oder einer Klasse definiert und sind überall im Programm sichtbar. Der Zugriff auf eine nicht initialisierte globale Variable liefert `nil`:

```
$Logger = Logger.new("rails")
...
$Logger.info("hello!")
```

Instanz- und Klassenvariablen werden in Abschnitt 1.15 beschrieben.

Konstanten müssen bei ihrer Definition initialisiert werden. Sie können außerhalb oder innerhalb von Klassen und Modulen definiert werden, jedoch nicht in Methoden. Der Wert einer Konstanten kann nach der Initialisierung verändert werden [2]:

```
MAX_LNG = 1024
...
MAX_LNG = 42
```

1.9 Namen und Symbole

In einem Ruby-Programm werden Namen für Variablen, Konstanten, Methoden, Klassen und Module vergeben. Ruby unterscheidet die Bedeutung eines Namens anhand des ersten Zeichens.

Lokale Variablen und Methodenparameter müssen mit einem Kleinbuchstaben oder einem Unterstrich beginnen. Klassen- und Instanzvariablen beginnen mit zwei bzw. einem Klammeraffen (`@`). Klassen- und Modulnamen sowie Konstanten beginnen mit einem Großbuchstaben. Globale Variablen beginnen mit einem Dollarzeichen (`$`). Für alle gilt, dass die weiteren Zeichen aus Buchstaben, Ziffern und dem Unterstrich bestehen dürfen. Tabelle 1.2 zeigt einige Beispiele.

Methoden beginnen mit einem Kleinbuchstaben, gefolgt von Buchstaben, Ziffern, oder dem Unterstrich.

Ruby definiert eine Reihe von reservierten Namen, die in Tabelle 1.3 aufgeführt sind.

[2] Dies führt allerdings zu einer Warnung durch den Interpreter

Tabelle 1.3: Reservierte Namen

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

In Ruby werden anstelle von Strings häufig Symbole verwendet. Ein Symbol wird durch einen führenden Doppelpunkt (`:`) gekennzeichnet:

```
string  = "Thomas"
symbol1 = :"Ralf"
symbol2 = :my_method
```

Symbole sind gegenüber Strings atomar, d.h. es gibt für ein und dasselbe Symbol nur genau eine Instanz. Egal, wo im Programm das Symbol auch referenziert wird, es handelt sich immer um dasselbe Symbol (dieselbe Instanz). Für ein und denselben String (z.B. ”ruby”) werden an unterschiedlichen Stellen im Programm immer unterschiedliche Instanzen erzeugt. Theoretisch würden also für 1.000.000 ”ruby”-Strings auch 1.000.000 Instanzen erzeugt. Bei der Verwendung von `:ruby` aber nur ein einziges:

```
"ruby".object_id        # 443352
"ruby".object_id        # 443332

:ruby.object_id         # 880910
:ruby.object_id         # 880910
```

Symbole werden daher überall dort bevorzugt, wo keine neue Instanz benötigt wird, z.B. als Schlüssel in einer Hash (vgl. 1.12.2) oder bei der Angabe von Namen:

```
hash1 = { :name => "Thomas", ... }
hash2 = { :name => "Ralf", ... }

has_one :motor
```

Der Vergleich zweier Symbole ist gegenüber Strings schneller, weil nur die Referenzen verglichen werden müssen. Hingegen werden bei zwei Strings deren Inhalte auf Gleichheit überprüft.

1.10 Bedingungen

Bedingungen können in Ruby auf unterschiedliche Weise definiert werden.

1.10.1 If und Unless

Jede `if`-Anweisung ist mit einem `end` abzuschließen. Dem `if` können optional beliebig viele `elsif`-Anweisungen und optional eine `else`-Anweisung folgen. Der Ausdruck in der Bedingung wird auf wahr oder falsch geprüft:

```
x = 5
if x == 0
  puts "0"
elsif x < 0
  puts "kleiner 0"
else
  puts "größer 0"
end
```

`Nil` oder ein zu `nil` ausgewerteter Ausdruck gilt dabei als falsch:

```
x = nil
if x
  puts "nicht angesprungen!"
else
  puts "angesprungen!"
end
# => "angesprungen!"
```

Wird die Anweisung in derselben Zeile wie die Bedingung angegeben, müssen sie durch das Schlüsselwort `then` bzw. den Doppelpunkt `:` getrennt werden:

```
if x == 0 then puts "0"
elsif x < 0 then "kleiner 0"
else "größer oder kleiner 0"
end

if x == 0 : puts "0"
elsif x < 0 : "kleiner 0"
else "größer oder kleiner 0"
end
```

Für kurze Einzeiler kann `if` auch hinter der Anweisung angegeben werden:

```
x = 1 if x > MAX
x = 0 if x > 0 && x < MAX
```

Für die Verknüpfung von logischen Ausdrücken stehen die Operatoren `&&` für Und und `||` für Oder bereit. Die Auswertung erfolgt *lazy*, d.h. ein Teil des Ausdrucks wird nur ausgewertet, falls das Gesamtergebnis noch nicht eindeutig ist. Beide Operatoren liefern dabei nicht `true` oder `false` zurück, sondern ein Objekt:

```
puts "Thomas" || "Ralf"  # "Thomas"
puts "Thomas" && "Ralf"  # "Ralf"
```

Dies vereinfacht u.a. die Zuweisung. Statt des folgenden Ausdrucks:

```
if params[:name]
  n = params[:name]
else
  n = "Defaultname"
end
```

erfolgt die Zuweisung einfach wie folgt:

```
n = params[:name] || "Defaultname"
```

Statt die Bedingung einer `if`-Anweisung durch `!` zu negieren, bietet Ruby die `unless`-Anweisung. Gegenüber `if` ist nur die optionale Angabe des `else` Zweigs möglich. Ein `elsif` o.ä. existiert hier nicht:

```
# der Code ...
if !x
  ...
else
  ...
end

# ... ist identisch mit diesem:
unless x
  ...
else
  ...
end
```

Auch die `unless`-Anweisung kann am Ende der Zeile stehen:

```
y = 1 unless y > MAX
```

Neben der Verwendung als Kontrollstruktur im Programm können `if` und `unless` auch direkt zur Auswertung von Ausdrücken verwendet werden:

```
c = if a > b then a; else b; end
z = unless x > y then x; else y; end
```

1.10.2 Ternary Operator

Für Schreibfaule bietet Ruby den aus C und Java bekannten Ternary-Operator:

```
puts sex == 0 ? "männlich" : "weiblich"
```

1.10.3 Case

Ruby bietet mit der `case`-Anweisung neben `if` ein weiteres Programmkonstrukt für den Kontrollfluss. Die Anweisung ist in `case` und `end` eingeschlossen. Sie kann beliebig viele `when`-Zweige und einen `else`-Zweig als Default besitzen:

```
case name
when "Thomas" : puts "ein Programmierer"
```

```
when "Ralf"   : puts "ein anderer Programmierer"
else puts "und was machst Du?"
end

case c
when '0'..'9' then puts "Ziffer"
when 'a'..'z' then puts "Buchstabe"
else puts "unbekanntes Zeichen #{c}."
end
```

Als Bedingung hinter `when` können ein oder mehrere Ausdrücke angegeben werden. Dies schließt Bereiche und reguläre Ausdrücke ein. Beginnt der Block zu `when` in der nächsten Zeile, ist die Angabe von `then` oder `:` nicht notwendig:

```
case c
when 'a'..'z', 'A'...'Z'
  puts "Buchstabe"
else
  puts "unbekanntes Zeichen #{c}."
end
```

Die `case`-Anweisung kann auch zur Auswertung eines Ausdruck verwendet werden. In diesem Fall entfällt die Angabe eines Wertes hinter `case`:

```
type = case
       when '0'..'9' === c : "Ziffer"
       when 'a'..'z' === c : "Buchstabe"
       else "unbekanntes Zeichen #{c}."
       end
```

1.11 Schleifen und Iteratoren

Für das Iterieren über eine Menge von Werten bietet Ruby Schleifen und Iteratoren. Im Zusammenspiel mit Arrays und Hashes werden Iteratoren eher eingesetzt, weil sie sich als kürzer, wartbarer und verständlicher erweisen. Für alle Fälle stehen aber auch Schleifen zur Verfügung.

1.11.1 For, While und Until

Ruby bietet mit `for`, `while` und `until` drei verschiedene Schleifenkonstrukte. Die `for`-Schleife iteriert über die Elemente eines Bereichs, Arrays oder auch Hashes. Bei letzterem ist die Reihenfolge jedoch nicht vorhersehbar:

```
for i in 1..10 do
  puts i
end

=>
1
```

```
2
...

for i in [42, "Ralf", -3.14]
  puts i
end

=>
42
Ralf
-3.14

for i in { :a=> "Thomas", :b=>"Ralf" }
  puts "#{i[0]} => #{i[1]}"
end

=>
b => Ralf
a => Thomas
```

Die `while` Schleife wird durchlaufen, solange die Bedingung wahr ist. Wird ein Ausdruck in der Bedingung zu `nil` oder `false` ausgewertet, gilt die Bedingung als falsch:

```
while condition do
  # Block wird ausgeführt, solange condition true ist
end
```

Soll die Schleife durchlaufen werden, solange die Bedingung `nil` oder `false` ist, bietet sich die Verwendung der `until`-Schleife an:

```
until condition do
  # Block wird ausgeführt, solange condition false ist
end
```

Für kurze Ausdrücke können die Schleifen auch nach dem Ausdruck angegeben werden:

```
a = 42
a -= 1 while a > 0
a += 1 until a > 42
```

1.11.2 Break, Next und mehr

Ruby bietet für die Steuerung der Schleife aus dem Schleifenblock heraus insgesamt vier Möglichkeiten. Diese werden anhand des folgenden Beispiel einer `for`-Schleife beschrieben, gelten aber auch für `while`, `until` oder die Iteration per `each`.

Die angegebene Schleife soll fünf Mal durchlaufen werden und bei jeder Iteration wird der Schleifenindex ohne Zeilenumbruch ausgegeben. Hat der Index den

Wert 3 erreicht, wird ein Zeilenumbruch ausgegeben und der Kontrollbefehl (z.B. `break`) ausgeführt. Eine ggf. erneute Ausführung des `if` Blocks wird verhindert, indem `active` auf `false` gesetzt wird:

```
active = true
for i in 1..5
  print i
  if i == 3 && active
    active = false
    print "\n"
    break; # next, redo, retry
  end
  print ","
end;
print "*\n"
```

Durch Angabe des Schlüsselworts `break` wird die Schleife verlassen und der Programmfluss nach der Schleife fortgesetzt. Die Ausgabe der Schleife lautet:

```
1,2,3
*
```

Bei der Verwendung von `next` wird die nächste Iteration der Schleife erzwungen (ähnlich dem `continue` aus Java). Die Ausgabe der Schleife lautet in diesem Fall:

```
1,2,3
4,5,*
```

Das Schlüsselwort `redo` führt zu einem erneuten Durchlaufen der Schleife, ohne den Schleifenindex zu erhöhen. Somit wird folgende Ausgabe getätigt:

```
1,2,3
3,4,5,*
```

Durch die Angabe von `retry` wird die Schleife erneut von Beginn an gestartet, daher lautet die Ausgabe:

```
1,2,3
1,2,3,4,5,*
```

1.11.3 Iteratoren

Das Iterieren über einen Container, wie z.B. Array oder Hash erfolgt in Ruby typischerweise nicht durch Schleifen, sondern durch Iteratoren in Verbindung mit Codeblöcken. Der Block enthält den Code zur Verarbeitung des Elements und wird an entsprechende Instanzmethoden des Containers übergeben. Die Methoden iterieren über jedes Element des Containers und übergeben dieses an den Codeblock. Ein häufig verwendeter Vertreter dieser Methoden ist `each`:

```
# möglich, aber untypisch
for i in 1...array.length do
  process array[i]
```

```
end

# eleganter und besser
array.each do |i|
  process i
end
```

Durch die Verwendung von Codeblöcken und Iteratoren wird eine klare Trennung der Verantwortlichkeiten geschaffen. Der Iterator ist für die Lieferung des jeweils nächsten Elements zuständig, aber von der Art Verarbeitung befreit. Der Codeblock hingegen ist rein für die Verarbeitung des Elements zuständig, muss sich aber nicht um dessen Beschaffung kümmern. Auf diese Weise kann das Iterieren über die Elemente des Arrays mit einer beliebigen Verarbeitung kombiniert werden.

Im Gegensatz dazu enthält der Code unter Verwendung der Schleife sowohl die Iteration wie auch die Verarbeitung. Beides muss für jede Art der Verarbeitung neu geschrieben werden.

1.12 Arrays und Hashes

Ruby bietet mit `Array` und `Hash` zwei Container als Standardtypen, die in diesem Kapitel näher beschrieben werden.

1.12.1 Array

Arrays sind in Ruby vom Typ der Klasse `Array`. Das Erzeugen eines Arrays geschieht typischerweise durch die Angabe von Werten innerhalb eckiger Klammern `[]`. Alternativ wird ein Array über `new` erzeugt:

```
array = Array.new
array[0] = 42
empty = []
numbers = [1, 7, 12, 42]
list = ["Thomas", 42, [3,7], buffer, sum(1,2)]
```

Die Größe eines Arrays muss beim Erzeugen nicht angegeben werden. Arrays sind in ihrer Länge nicht begrenzt und werden bei Bedarf automatisch erweitert. Die Länge eines Arrays wird über die Methode `length` geliefert:

```
array = Array.new
array[0]  = 10
array[1]  = 11
...
array.length
```

Auf die Objekte im Array wird mit der Methode `[]` zugegriffen. Dabei beginnt die Indizierung, wie auch aus C und Java, bekannt bei 0. In Ruby können darüber hinaus auch negative Indizes verwendet werden. Die Indizierung beginnt dabei mit -1 für das letzte Element, -2 für das vorletzte usw.:

```
numbers = [1, 7, 12, 42]
numbers[0]  # 1
numbers[3]  # 42
numbers[4]  # nil
numbers[-1] # 42
numbers[-2] # 12
```

Die Methode `[]` kann zusätzlich zum Index die Anzahl der zurückzuliefernden Elemente annehmen. Hierdurch wird ein Teilbereich des Arrays als neues Array zurückgeliefert:

```
numbers = [1, 7, 12, 42]
numbers[0,2]  # [1, 7]
numbers[2,1]  # [12]
numbers[2,0]  # []
numbers[-2,2] # [12, 42]
numbers[0,10] # [1, 7, 12, 42]
```

Die Angabe eines Bereichs als Parameter ist ebenfalls möglich. Auch in diesem Fall wird ein Teilbereich des Arrays als neues Array zurückgeliefert:

```
numbers = [1, 7, 12, 42]
numbers[0..1]   # [1, 7]
numbers[0..2]   # [1, 7, 12]
numbers[0..10]  # [1, 7, 12, 42]
numbers[-3..-1] # [7, 12, 42]
```

Die Art der Indizierung lässt sich auch auf die Zuweisung anwenden und ermöglicht so eine flexible Art, Arrays zu füllen. Entstehen bei der Zuweisung von Werten an ein Array Lücken, so werden diese mit `nil` gefüllt:

```
a = [1,2,3,4,5,6]
a[2] = [3,2,1]
a.inspect   # [1, 2, [3, 2, 1], 4, 5, 6]

a = [1,2,3,4,5,6]
a[1..4] = [0,0,0]
a.inspect   # [1, 0, 0, 0, 6]

a = [1,2,3,4,5,6]
a[1..4] = 0
a.inspect   # [1, 0, 6]

a = []
a[0]  = 10
a[2]  = 11
a.length    # 3
a.inspect   # [10, nil, 11]
```

Das Hinzufügen von Werten an das Ende des eines bestehenden Arrays erfolgt über die Methode <<:

```
a = [ 1, 2 ]
a << 3 << 4 << [5, 6] # [1, 2, 3, 4, [5, 6]]
```

Das Iterieren über die Inhalte eines Arrays erfolgt in Ruby in der Regel über Iteratoren in Verbindung mit Codeblöcken (vgl. Abschnitt 1.11.3). Ein häufig verwendeter Vertreter dieser Methoden ist `each`:

```
array.each do |i|
  puts i
end
```

Arrays aus Strings können auch elegant über die Notation per `%w{}` und `%W{}` erzeugt werden. Erstere führt keine Ersetzungen vor, letztere schon. Leerzeichen in einem Wort sind durch einen Slash anzugeben:

```
favorite = "Ruby"

a = %w{ Java Ruby Visual\ Basic #{favorite} }
=> [ "Java", "Ruby", "Visual Basic", "#{favorite}"]

b = %W{ Java Ruby Visual\ Basic "#{favorite}" }
=> [ "Java", "Ruby", "Visual Basic", "Ruby"]
```

1.12.2 Hash

Eine Hash wird in Ruby durch die Standardklasse `Hash` repräsentiert. Eine Hash wird typischerweise durch die Angabe von Paaren aus Schlüssel und Wert innerhalb geschweifter Klammern {} erzeugt. Alternativ ist die Erzeugung durch die Klassenmethode `new` möglich:

```
members = Hash.new
members[id] = "Thomas"
empty = {}
hash = { "x" => 199, 123 => [1,2,3],
         :z => { "a"=>1, "b"=>2 } }
```

Auch eine Hash wird nach Bedarf in der Größe erweitert und ist lediglich durch den Speicher begrenzt. Die Länge einer Hash-Instanz kann über die Methode `length` ermittelt werden. Der Zugriff auf einen Wert in der Hash erfolgt analog dem Array über die Methode `[]`. Im Gegensatz zum Array kann der Schlüssel aber von jedem Typ sein, wobei Strings und Symbole die Regel sind:

```
hash = { "x" => 199, 123 => [1,2,3],
         :z => { "a"=>1, "b"=>2 } }
hash["x"]       # 199
hash[123]       # [1,2,3]
hash[:z]        # {a=>1, b=>2}
hash[:z]["b"]   # 2
hash["a"]       # nil
hash.length
```

Das Iterieren über die Inhalte einer Hash erfolgt in Ruby in der Regel über Iteratoren in Verbindung mit Codeblöcken (vgl. Abschnitt 1.11.3). Ein häufig verwendeter Vertreter dieser Methoden ist `each`:

```
hash.each do |key,value|
  puts value[key]
end
```

Um in einer Hash den Wert zu einem Schlüssel zu finden, werden die Methoden `hash` und `eql?` des Schlüssel verwendet. Die Methode `hash` liefert den Hashwert als Ganzzahl (`Fixnum`) der Instanz und `eql?` vergleicht den übergebenen Schlüssel mit dem in der Hash.

Wird eine eigene Klasse als Schlüssel definiert, die direkt von `Object` erbt, so ist neben `hash` auch `eql?` zu überschreiben. Andernfalls erfolgt der Vergleich auf Basis der Objekt-Id (vgl. Abschnitt 1.15.5). Es wird dann nur derselbe Schlüssel gefunden, nicht aber der Schlüssel mit gleichem Wert. Außerdem ist sicherzustellen, dass der Hashwert eines Schlüssels sich nach der Erzeugung nicht ändert. Geschieht dies doch, ist nach jeder Änderung die Methode `Hash#rehash` aufzurufen:

```
class MyKey
   attr_reader :name
   def initialize(name)
      @name = name
   end

   def eql? obj                       # ohne eql?
      obj.kind_of?(self.class) &&     # liefert
        obj.name == @name             # hash[key2]
   end                                # nil statt 42

   def hash
      @name.hash
   end
end

key1=MyKey.new("Thomas")
key2=MyKey.new("Thomas")
hash = { :key1 => 42 }
hash[key1]  # => 42
hash[key2]  # => 42
```

Etwas Vorsicht ist bei der Verwendung von Symbolen und Strings als Schlüssel geboten. Symbole und Strings sind unterschiedliche Typen und werden als Schlüssel in der Hash unterschiedlich behandelt. Ein Symbol `:a` und ein String `"a"` als Schlüssel verweisen daher auf zwei verschiedene Werte:

```
h = {}
h["a"] = 42
h["a"]  # 42
h[:a]   # nil
```

```
h[:a]  = 11
h["a"]  # 42
h[:a]   # 11
```

Hingegen erlauben die von Rails vordefinierten Hashes, wie z.B. `@params` oder `@session` den Zugriff über Strings oder Symbole, wodurch der Zugriff vereinfacht wird. Wir empfehlen Ihnen aufgrund der unter Abschnitt 1.9 genannten Punkte die Verwendung von Symbolen als Schlüssel einer Hash.

1.13 Methoden

Eine Methodendefinition beginnt mit dem Schlüsselwort `def`, gefolgt vom Methodennamen, optionalen Parametern und dem abschließenden `end`:

```
def methodenname(param1, param2, ...)
  ...
  return x
end
```

Methoden, die außerhalb einer Klasse definiert sind, werden automatisch private Instanzmethoden der Klasse `Object` (vgl. Abschnitt 1.15.5).

Da Ruby eine dynamisch typisierte Sprache ist, entfallen bei der Methodendefinition die Typangaben für Parameter und Rückgabewert. Das Schlüsselwort `return` kann entfallen, da Ruby immer das Ergebnis des letzten Ausdrucks als Wert zurückliefert. Auch die Klammern bei einer Methodendefinition oder einem Methodenaufruf sind optional:

```
def add a, b
  a + b
end

puts add 1, 2
```

Da Klammern aber häufig helfen, den Code lesbarer zu machen, sollte auf Klammern nur in einfachen Ausdrücken verzichtet werden. Auch der Ruby Interpreter kann in Schwierigkeiten geraten, wenn allzu sehr mit Klammern gespart wird. Sollte einmal die Meldung `warning: parenthesize argument(s) for future version` erscheinen, so empfiehlt der Interpreter, in Zukunft mehr Klammern zu verwenden, damit er nicht ins Stolpern gerät:

```
puts add 1, 2  # möglich, aber ggf. Warnung
puts add(1, 2) # eindeutig und besser
```

Klammern sollten immer direkt nach dem Methodennamen angegeben werden und nicht durch ein Leerzeichen davon getrennt sein. Andernfalls erzeugt der Interpreter auch hier eine Warnung der Art: `warning: don't put space before argument parentheses`.

In Ruby ist es nicht möglich, zwei Methoden mit demselben Namen in einem Namensraum (z.B. Klasse oder Modul) zu definieren, d.h. es gibt kein Überladen von Methoden. Es besteht aber die Möglichkeit, ein und dieselbe Methode mit einer unterschiedlichen Zahl von Argumenten aufzurufen. Zum einen bietet Ruby die Defaultbelegung von Argumenten an, wobei diese nur am Ende der Parameteraufzählung angegeben werden dürfen:

```
def method1(a, b=1) ...
def method2(a, b=1, c=2) ...
def method3(a, b=1, c) ... # Fehler! c ohne Vorbelegung

method1(1)
method1(1,2)
...
```

Zum anderen kann die Methode auch gleich mit einer variablen Anzahl von Parametern definiert werden. Dazu erhält die Methode genau einen Parameter, der mit einem Stern (`*`) beginnen muss:

```
def foo(*vargs) ...
def foo(name, *vargs) ...

foo("Thomas", 42, 43, 44)
foo("Thomas", 42)
foo("Ralf")
```

Methoden können mehr als einen Rückgabewert haben. Hierdurch wird die elegantere und flexible Zuweisung von Rückgabewerten an Variablen ermöglicht:

```
def double_trouble
  return "Ralf", "Thomas"
end

name1, name2 = double_trouble
```

Die Anzahl der Variablen muss dabei nicht zwangsläufig der Anzahl der zurückgelieferten Werte entsprechen. Ein schönes Beispiel hierfür ist die Methode `split` der Klasse `String`:

```
h       = "16:42:23".split(':')  # ["16", "42", "23"]
h,      = "16:42:23".split(':')  # "16"
h, m    = "16:42:23".split(':')  # "16", "42"
h, m, s = "16:42:23".split(':')  # "16", "42", "23"
```

In Ruby werden Methoden per Konvention häufig mit einem Ausrufungs- oder Fragezeichen beendet. Erstere modifizieren die Instanz, auf der sie aufgerufen werden. Eine gleichnamige Methode ohne Ausrufungszeichen liefert hingegeben eine modifizierte Kopie und ändert die Instanz selbst nicht:

```
s = " Thomas "
t = s.strip
puts s  # => " Thomas "
```

```
puts t  # => "Thomas"
s.strip!
puts s  # => "Thomas"
```

Methoden mit einem Fragezeichen am Ende prüfen einen Zustand und liefern immer einen boolschen Wert zurück:

```
a = [ 1, 2, 3 ]
a.include? 2  # => true
a.include? 42 # => false
```

Die Konvention sollte auch für eigene Methoden eingehalten werden. Dies fördert die Einheitlichkeit und das Lesen und Verstehen von Ruby-Programmen.

Methoden können auch Codeblöcken übergeben werden (vgl. Abschnitt 1.14).

1.14 Codeblöcke

Ruby ermöglicht die Definition von Codeblöcken. Ein Codeblock enthält Anweisungen und Ausdrücke und ist in geschweifte Klammern ({}) oder in `do` und `end` eingeschlossen. Ein Block wird ausschließlich als ein Argument eines Methodenaufrufs definiert. Der Start des Blocks durch { oder `do` muss in der selben Zeile, wie der Methodenaufruf stehen.

```
my_method { puts "Meine Welt" }

your_method do
  puts "Deine Welt"
end
```

An eine Methode kann zur Zeit nur ein Block übergeben werden. Der Block ist immer als letzter Parameter anzugeben, wird aber in der Methodendefinition nicht als Parameter definiert:

```
def execute(count)
  puts count
  yield
end

execute(42) { puts "ausgeführt!" }

=>
42
ausgeführt!
```

Ein Codeblock wird innerhalb der Methode durch `yield` ausgeführt. An `yield` übergebene Parameter werden in der Reihenfolge ihrer Angabe an den Block weitergereicht. Der Block liefert das Ergebnis des letzten Ausdrucks im Block an die Methode zurück. Im folgenden Beispiel erhält die Methode `operation` zwei Werte und den Operator in Form eines Blocks übergeben. Der Befehl `yield` ruft

den Block auf und übergibt die Werte `a` und `b` als Argumente. Der Block nimmt die Werte in `x` und `y` entgegen und führt die Anweisung aus:

```
def operation(a,b)
  yield a, b
end

operation(42,24) { |x,y| x + y }   # => 66
operation(42,24) { |x,y| x - y }   # => 18
```

Codeblöcke werden in Ruby im Zusammenhang mit Iteratoren über Arrays oder Hashes verwendet (vgl. Abschnitt 1.11.3). Ruby bietet hier entsprechende Methoden, die einen Codeblock als Parameter annehmen und für jedes Element im Array oder Hash diesen Block aufrufen. Ein typisches Beispiel ist die Methode `each`:

```
[1,2,3].each do |x|
  puts x
end

=>
1
2
3

{ :ralf => 1, thomas => 2 }.each do |key, value|
  puts "#{key} = #{value}"
end

=>
ralf = 1
thomas = 2
```

Ein Codeblock ist an sich einfacher Ruby-Code, kann aber explizit in eine Instanz der Klasse `Proc` gewandelt werden. Auf diese Weise ist die Zuweisung an Variablen möglich. Im folgenden Beispiel wird die Art der Operation (+, -, usw.) als Block bei der Instanziierung der Klasse `Operator` übergeben. Auf diese Weise können unterschiedliche Instanzen von `Operator` unterschiedliche Operationen ausführen. Die Konvertierung des Codeblocks erfolgt in diesem Fall durch die Kennzeichnung des letzten Parameters durch ein kaufmännisches Und (`&`):

```
class Operator
  def initialize(name, &operation)
    @name = name
    @op   = operation
  end
  def execute(a, b)
    @op.call(a,b)
  end
end

plus  = Operation.new("+") { |x,y| x + y }
```

```
minus = Operation.new("-") { |x,y| x - y }
plus.execute(42,24)   # 66
minus.execute(42,24)  # 18
```

Im Beispiel ist pro Operator keine neue Klasse (PlusOperator, MinusOperator, usw.) notwendig. Ohne Codeblöcke und nur mit der Klasse `Operator` müßte eine Klasse `PlusOperation`, `MinusOperation` usw. erstellt werden. Mit Codeblöcken entfällt diese Notwendigkeit.

Eine zweite Möglichkeit ist, den Block explizit in eine Instanz von `Proc` zu konvertieren. Bei der Übergabe an eine Methode ist der Block dann durch ein kaufmännischen Und (`&`) als solcher zu kennzeichnen:

```
block = Proc.new { |x| puts x }
[1,2,3].each &block
```

Die dritte und letzte Möglichkeit erzeugt einen Block über die Kernel-Methode `lambda`:

```
block = lambda { |x| puts x }
[1,2,3].each &block
```

Im Unterschied zu `Proc.new` wird eine durch `lambda` erzeugte Blockinstanz wie ein Block einer Methode interpretiert und kann per `return` verlassen werden:

```
def method_proc
  p = Proc.new { return 42 }
  p.call
  17
end

def method_lambda
  p = lambda { return 42 }
  p.call
  17
end

method_proc     # => 42
method_lambda   # => 17
```

Im folgenden Beispiel ist der den Block umgebende Kontext, im Falle von `Proc.new` nicht mehr vorhanden. Ein `return` erzeugt daher einen `LocalJumpError`. Im Falle von `lambda` und dem dadurch existierenden (Methoden-)Block führt das `return` zu keinem Fehler:

```
def new_Proc(&block)
  block
end

block = new_Proc { return 42 }
block.call  # => unexpected return (LocalJumpError)

lmd = lambda { return 42 }
```

```
block = new_Proc &lmd
block.call  # 42
```

Ein Codeblock merkt sich den Kontext in dem er definiert wurde. Dies schließt Konstanten, Instanz-, Klassen- und lokale Variablen ein. Bei jeder Ausführung des Blockes steht dieser Kontext zur Verfügung. Dieses Prinzip wird als *Closure* bezeichnet. Im folgenden Beispiel werden auch bei der Anwendung des Blocks über `use_block` die bei der Erzeugung zugewiesenen Werte (`Bibi Blocksberg` und `123`) verwendet:

```
class BlockCreate
  def create_block
    @name = "Bibi Blocksberg"
    value = 123
    block = lambda { puts "#{@name}, #{value}" }
  end
end

class BlockUse
  def use_block
    @name = "Graf Dracula"
    value = 4711
    yield
  end
end

block = BlockCreate.new.create_block
block.call              # Bibi Blocksberg, 123
user = BlockUse.new
user.use_block &block   # Bibi Blocksberg, 123
```

Besteht die Notwendigkeit zu prüfen, ob ein Codeblock an eine Methode übergeben wurde, steht hierfür die Methode `block_given?` zur Verfügung:

```
def dump(array, &block)
  unless block_given?
    block = Proc.new { |x| puts x }
  end
  array.each &block
end

a = [65, 66, 67]
dump(a)                     # 65 66 67
dump(a) { |x| puts x.chr }  # A B C
```

1.15 Klassen und Instanzen

Ruby ist eine objektorientierte Sprache und bietet daher das Konzept der Klassen und Instanzen. Klassen sind im Gegensatz zu anderen Sprachen in Ruby nach

ihrer Definition offen für Erweiterungen und Änderungen. Des weiteren können Klassendefinitionen Ruby-Code enthalten, der bei der Interpretation der Klassendefinition ausgeführt wird. Klassen sind in Ruby ebenfalls Objekte. Jede Klasse in Ruby ist eine Instanz der Klasse `Class`, die direkt von `Module` erbt.

1.15.1 Klassen

Eine Klassendefinition in Ruby beginnt mit dem Schlüsselwort `class` gefolgt vom Klassennamen und einem abschließenden `end`. Klassennamen beginnen in Ruby immer mit einem Großbuchstaben. Besteht der Name aus mehreren Wörtern, beginnt jedes Wort mit einem Großbuchstaben.

```
class Project
end

class WebApplicationProject
end
```

In der Regel wird eine Klasse in genau einer Datei gespeichert. Anders als z.B. in Java, besteht keine Notwendigkeit, die Datei entsprechend dem Klassennamen zu benennen. Es hat sich etabliert, den Dateinamen komplett klein zu schreiben und einzelne Wörter durch einen Unterstrich (_) zu trennen:

```
# file: web_application_project.rb
class WebApplicationProject
end
```

Eine Instanz einer Klasse wird über den Aufruf der Methode `new` erzeugt. Die Methode erzeugt zunächst mit Hilfe der Methode `allocate` Speicher für das Objekt. Anschließend ruft sie, wenn vorhanden, die Methode `initialize` auf. Diese ist vom Entwickler zu definieren und dient der Initialisierung des Objekts. Beim Aufruf von `new` übergebene Argumente werden dabei an `initialize` weitergereicht:

```
class Project
  def initialize(name)
    @name = name
  end
end

project = Project.new("Ruby lernen")
```

Die Methode `initialize` kann nur einmal in einer Klasse definiert werden. Sollen Instanzen mit einer unterschiedliche Anzahl von Werten initialisiert werden, so sind die optionalen Parameter mit einem Defaultwert zu belegen:

```
class Project
  def initialize(name, lead = "unknown", start = nil)
    @name = name
    @lead = lead
    @start = start
```

```
  end
end

project = Project.new("Ruby lernen")
project = Project.new("Ruby lernen", "Ralf")
project = Project.new("Ruby lernen", "Ralf", "01.01.2006")
```

Instanzmethoden einer Klasse werden wie gewöhnliche Methoden definiert (vgl. Abschnitt 1.13). Sie haben Zugriff auf alle Instanzvariablen, Klassenvariablen und Konstanten der Klasse. Mit Hilfe des Befehls `alias_method` können für Instanzmethoden einer Klasse Aliases vergeben werden. Unter dem neuen Namen wird eine Kopie der Methode abgelegt. Änderungen an der alten Methode haben somit keine Auswirkung auf die neue:

```
module Kernel
  alias_method :orig_system, :system

  def system(*args)
    code = orig_system(*args)
    puts "system code: #{code}"
                code
  end
end
```

Instanzvariablen beginnen in Ruby mit dem Klammeraffen (`@`), die folgenden Zeichen können beliebig gewählt werden [3]. Eine Instanzvariable wird nicht (wie z.B. in Java) in der Klasse und außerhalb von Methoden definiert[4].

Die Definition einer Instanzvariablen erfolgt in Ruby innerhalb einer Instanzmethode. Die Variable wird beim ersten Vorkommen in einer Methode angelegt und steht danach in allen folgenden Instanzmethoden zur Verfügung:

```
class Timer
  @start_time = 0 # keine Instanzvariable!

  def start
    # erzeugt Instanzvariable @start_time
    @start_time = Time::now
  end

  def stop
    # kennt nur @start_time aus start.
    @delay = Time::now - @start_time
  end
end
```

Das erste Vorkommen muss dabei nicht, wie oben zu sehen, zwangsläufig in der Methode `initialize` geschehen. Die Methode `initialize` dient nur der op-

[3] Mit der Ausnahme, dass einem `@` keine Ziffer folgen darf.
[4] Die Definition ist zwar möglich, führt aber eine Klassen-Instanzvariable ein. Diese Art ist für Metaklassenprogrammierung nützlich (vgl. *Dave Thomas: Programming Ruby*).

tionalen Initialisierung von Variablen, vor der ersten Nutzung in den Instanzmethoden.

Wie der Name schon sagt, sind Instanzmethoden und Instanzvariablen Elemente einer Instanz der Klasse. Innerhalb der Klasse kann auf die eigene Instanz mit dem Schlüsselwort `self`[5] zugegriffen werden. Da Ruby für jede Instanzmethode oder Instanzvariable implizit `self` verwendet, kann die Angabe aber entfallen.

Ausnahme bildet der Zugriff auf eine Setter-Methode, andernfalls wird die Anweisung als die Zuweisung an eine lokale Variable gewertet:

```
class Project
  attr_writer :name

  def init
    name = "Rails"      # lokale Variable name
    self.name = "Ruby"  # Aufruf Setter Methode name=
  end
end
```

Per Default sind Instanz- wie auch Klassenvariablen private Elemente der Instanz, auf die von außen weder lesend noch schreibend zugegriffen werden kann. Für jedes Attribut muss es eine entsprechende Getter- und/oder Setter- Methode geben:

```
class Project
  def initialize
    @count = 0
    @name  = ""
  end
  def name
    @name
  end
  def name=(n)
    @name = n
  end
end

prj = Project.new("Java lernen")
prj.count = 99               # Zugriff verweigert
puts prj.count               # Zugriff verweigert
prj.name = "Ruby lernen"     # setze neuen Namen
puts prj.name                # => "Ruby lernen"
```

Ruby wäre aber nicht Ruby, wenn dies nicht auch einfacher ginge. Statt für jede Variable explizit eine Getter- und Setter-Methode zu definieren, wird dies von Ruby übernommen. Dazu werden die gewünschten Variablennamen hinter der vordefinierten Methode `attr_reader` bzw. `attr_writer` aufgeführt. Alternativ kann der Aufruf durch die Methode `attr` und `attr_accessor` vereinfacht

[5] Analog `this` in Java

werden. In allen Fällen wird automatisch auch die Instanzvariable (hier `@name`) erzeugt:

```
class Project
  attr_reader :name
  attr_writer :name
  # Alternative: erzeugt Getter und mit true auch Setter:
  #     attr :name, true
  # Alternative: erzeugt Getter und Setter:
  #     attr_accessor :name
end

prj = Project.new
prj.name = "Ruby lernen"  # @name = "Ruby lernen"
puts prj.name             # "Ruby lernen"
```

1.15.2 Sichtbarkeit von Methoden

Mit Ausnahme der privaten Methode `initialize` sind alle Methoden einer Klasse per Default öffentlich. Die Sichtbarkeit einer Methode kann durch die Angabe eines der drei Schlüsselwörter `public`, `protected` oder `private` festgelegt werden. Die Schlüsselwörter leiten bei jeder Angabe einen Bereich ein, in dem die entsprechende Sichtbarkeit gilt. Eine Art *package private*, wie z.B. in Java gibt es aufgrund fehlender Packages in Ruby nicht:

```
class Project
  # public per Default
  def description
  end

protected
  def method_protected1
  end
  def method_protected2
  end

public
  def method_public1
  end
  def method_public2
  end

private
  def method_private1
  end
  def method_private2
  end
end
```

Alternativ können die Methodennamen auch als Parameter an Methoden zur Zugriffskontrolle übergeben werden. Die Angabe muss am Ende der Klasse und nach den Methodendefinitionen erfolgen:

```
class Project
  def Project.class_method
  end

  def method1
  end

  def method2
  end

  def method3
  end

  def method4
  end

  private_class_method :class_method1
  protected :method2, :method3
  private   :method4
end
```

Außerhalb der Klassen sind nur die durch `public` definierten Methoden sichtbar. Innerhalb kann die Klasse auf alle Methoden ihrer Superklassen zugreifen, auch auf die privaten. Wird innerhalb der Klasse eine Instanz der Klasse selbst oder einer Superklasse erzeugt, so kann die Klasse in diesem Fall nur auf die `public`- und `protected`-Methoden zugreifen. Der Zugriff auf die privaten Methoden ist nur über `self` erlaubt. Da eine Instanz erzeugt wird, die `self` kapselt, kann von außen nicht auf private Methoden zugegriffen werden. Auch nicht, wenn die erzeugte Instanz von der Klasse selbst ist. Die Einschränkung der Sichtbarkeit von Methoden gilt auch für Module (vgl. Abschnitt 1.16):

```
class Project
public
  def pub
  end
protected
  def prot
  end
private
  def priv
  end
end

class RailsProject < Project
  def call
    pub     # erlaubt (self.pub)
    prot    # erlaubt (self.prot)
```

```
    priv    # erlaubt (self.priv)
    p = Project.new
    p.pub   # erlaubt (p.pub)
    p.prot  # erlaubt (p.prot). Zwar protected
            # aber innerhalb von Superklasse.
    p.priv  # nicht erlaubt, da nicht self.priv!
  end
end
```

Wie in Abschnitt 1.15.4 beschrieben, können Subklassen die Sichtbarkeit von Methoden ihrer Superklasse ändern.

1.15.3 Klassenvariablen und Klassenmethoden

Klassenvariablen werden in Ruby durch zwei vorangestellte Klammeraffen (`@@`) definiert. Sie müssen bei ihrer Definition initialisiert werden und sind innerhalb der Klasse und allen Subklassen sichtbar. Eine Klassenvariable wird von allen Instanzen der Klasse geteilt und eine Änderung ist sofort in alle Instanzen der Klasse sichtbar:

```
class Project
  @@counter = 0

  def initialize(name)
    @@counter += 1
    @name = name
  end

  def Project.counter
    @@counter
  end

end

p1 = Project.new("Java")
p2 = Project.new("Ruby")
Project.counter   # 2
Project::counter  # 2
```

Eine Klassenmethode erhält zur Unterscheidung gegenüber der Instanzmethode als Präfix den Klassenname oder alternativ `self` gefolgt von einem Punkt.

```
class Project
  def Project.category_of?(category)
    ...
  end

  def self.counter(category)
    ...
  end
end
```

Das Schlüsselwort `self` bezieht sich hierbei auf die Klasse und nicht auf eine Instanz der Klasse. Eine Klasse wird in Ruby selbst als ein Objekt definiert und besitzt daher das Schlüsselwort `self`.

Der Zugriff auf die Klassenmethode erfolgt außerhalb der Klasse über den Präfix des Klassennamens oder über zwei Doppelpunkte (`::`). Die Sichtbarkeit von Klassenmethoden ist identisch zur der von Instanzmethoden:

```
Project.category_of?(...)
Project::counter
```

Klassenmethoden können auch innerhalb der Klassendefinition, ausgeführt werden. Stößt der Interpreter auf eine Klassendefinition, so führt er eine innerhalb des Klassenblocks angegebenen Klassenmethode aus:

```
class Project
  def Project.category_of?(category)
    puts "category: #{category}"
  end

  category_of :webapp
end

Project.new
# category: webapp
```

Der Aufruf von Klassenmethoden innerhalb einer Klassendefinition, wird Ihnen bei der Arbeit mit Rails noch sehr häufig begegnen.

1.15.4 Vererbung

Als objektorientierte Sprache bietet Ruby die Möglichkeit der Vererbung. Die Subklasse erbt dabei alle Variablen, Konstanten und Methoden. Dies gilt auch für die Methode `initialize`:

```
class Project
  def initialize(name)
    @name = name
    ...
  end
  ...
end

class RailsProject < Project
  # keine explizite Definition von initialize notwendig
  ...
  def rails_name
    "Rails: #{@name}"  # Zugriff auf Intanzvariable
                       # der Superklasse
  end
end
```

```
rp = RailsProject.new("Web-Projekt")
rp.rails_name   # "Rails: Web-Projekt"
```

Die Subklasse hat Zugriff auf alle Methoden der Superklasse inklusive der privaten. Die Subklasse kann dabei die Sichtbarkeit der Methoden für ihre eigenen Instanzen ändern:

```
class Project
  def foo
    ...
  end
  private :foo
end

class RailsProject < Project
  public :foo
end

p1 = Project.new.foo      #  geht nicht, da private
p2 = RailsProject.new.foo #  geht, da public
```

Wird eine Methode der Superklasse überschrieben und muss aus der gleichnamigen Methode der Subklasse die Methode der Superklasse aufgerufen werden, erfolgt der Aufruf über `super`:

```
class Project
  def to_s
  end
end

class RailsProject < Project
  def to_s
    s = super   # ruft to_s von Project
    ...
  end
end
```

Eine Mehrfachvererbung wird in Ruby nicht unterstützt. Durch das Einbinden von Modulen in Klassen ist aber eine Art Mehrfachvererbung zu erreichen (vgl. Abschnitt 1.16.2)

1.15.5 Klasse Object

Die Klasse `Object` ist in Ruby die Basisklasse aller Klassen und Module sie wird in diesem Abschnitt etwas näher betrachtet.

Der Interpreter erzeugt beim Start eines Programms eine Instanz von `Object`, das Top-Level-Objekt sozusagen. Alle Methoden, die außerhalb von Klassen und Modulen definiert werden, sind automatisch Methoden dieses Objekts und können überall im Programm verwendet werden.

Da ein Instanz der Klasse `Object` auch eine Reihe von Standardmodulen einbindet (vgl. Abschnitt 1.16), stehen die Methoden dieser Module ebenfalls überall im Ruby-Programm zur Verfügung. Ein Beispiel eines solchen Moduls ist `Kernel`. Dieses definiert u.a. die häufig verwendete Methode `puts` zur Ausgabe von Meldungen auf der Console oder auch `require` zum Einbinden von Modulen in das Programm (vgl. Abschnitt 1.3).

Jedes Objekt (jede Instanz einer Klasse) hat in Ruby eine eindeutige Id, die über die Methode `object_id` ermittelt werden kann. Es gibt keine zwei Objekte, die dieselbe Id erhalten[6]:

```
s1 = "Thomas"
s2 = s1
s1.object_id  # => 21277436
s2.object_id  # => 21277436
s2 = "Ralf"
s2.object_id  # => 21277499
```

Der Typ eines Objekts kann über die Methode `class` ermittelt werden. Um zu prüfen, ob ein Objekt zu einer Klasse gehört, steht die Methode `instance_of?` zur Verfügung. Dagegen wird bei `kind_of?` auch die Superklasse und ggf. eingebundene Module (vgl. Abschnitt 1.16) berücksichtigt:

```
class Project
end

class RubyProject < Project
end

class RailsProject < RubyProject
end

ruby  = RubyProject.new

ruby.class                      # RubyProject
ruby.instance_of? Project       # false
ruby.instance_of? RubyProject   # true
ruby.instance_of? RailsProject  # false
ruby.kind_of? Project           # true
ruby.kind_of? RubyProject       # true
ruby.kind_of? RailsProject      # false
```

Die Klasse `Object` stellt für eine String-Repräsentation eines Objekts die Methode `to_s` zur Verfügung. Per Default gibt sie den Klassenname und die Objekt-Id zurück. Subklassen überschreiben die Methode ggf., erzeugen aber häufig kein gut lesbares Format. Hierfür ist in Ruby eher die Methode `inspect` geeignet. Wird sie nicht überschrieben, ruft sie jedoch `to_s` auf:

```
a = [ "Thomas", "Ralf", "David" ]
puts a.to_s     # => ThomasRalfDavid
```

[6] Mit Ausnahme von kleinen Zahlen, `true`, `false` und `nil`.

```
puts a.inspect  # => ["Thomas", "Ralf", "David"]
```

Die Prüfung auf `nil` kann in Ruby über `obj == nil` oder die Instanzmethode `nil?` erfolgen:

```
if obj == nil ...
if obj.nil? ...
```

Für die Vergleiche von Objekten stehen in Ruby eine Reihe von Methoden mit unterschiedlichem Verhalten zur Verfügung. Die Methode == prüft die Gleichheit zweier Objekte, z.B. den Inhalt zweier Strings. In den meisten Fällen verhält sich die Methode `eql?` gleich. Sie wird u.a. für den Vergleich der Schlüssel einer Hash verwendet (vgl. Abschnitt 1.12.2). Für Zahlen wird im Vergleich zu == aber zusätzlich auf Typgleichheit geprüft:

```
42 == 42.0    # => true
42.eql? 42.0  # => false
```

Im Gegensatz dazu vergleicht die Methode `equal?` die Objekt-Id der beiden Objekte:

```
a = "Thomas"
b = "Thomas"
a == b      # => true
a.eql? b    # => true
a.equal? b  # => false
a.equal? a  # => true
```

Als weitere Variante steht die Methode === zur Verfügung. Per Default liefert sie das selbe Ergebnis wie ==. Sie wird in Subklassen häufig überschrieben und prüft, ob ein Objekt in einer Menge enthalten ist. Die `case`-Anweisung (vgl. Abschnitt 1.10.3) oder Bereiche (vgl. Abschnitt 1.7) sind Beispiele.

Eine besonders interessante Methode stellt `method_missing` dar. Ruft der Interpreter auf einem Objekt eine Methode auf, die nicht existiert, wirft er zunächst keine Ausnahme. Er führt erst `method_missing` aus, die per Default die Ausnahme wirft. Subklassen von `Object` können also die Methode überschreiben und für ein anderes Verhalten sorgen.

1.15.6 Erweiterung von Klassen

Ruby erlaubt Klassen und Module zur Laufzeit zu erweitern. Stößt der Interpreter auf eine Klassendefinition, prüft er ob diese Klasse bereits existiert. Ist dies der Fall, werden alle Klassen- und Instanzmethoden, Variablen und Konstanten der neuen Definition zur existierenden Klasse hinzugefügt.

Soll z.B. das Datum N Tage von heute an ermittelt werden, wäre eine erste Lösung die Definition einer Modulmethode (vgl. Abschnitt 1.16). Dadurch ist die Implementierung gekapselt und kann ausgetauscht werden, ohne alle Stellen im Programm ausfindig machen zu müssen:

```
module DateUtil
```

```
  def DateUtil.date_days_from_now(days)
    DateTime::now + days
  end
end

DateUtil.date_days_from_now(10)
DateUtil.date_days_from_now(42)
```

Eine Alternative bietet die Erweiterung der Klasse `Integer`. Die Methode `days` liefert hierbei die Zahl selbst zurück und soll deutlich machen, dass Tage gemeint sind. Die Methode `from_now` liefert das Datum N Tage von heute zurück. Dadurch lässt sich der Programmcode viel eleganter und verständlicher ausdrücken, eben *the ruby way*[7]:

```
class Integer
  def days
    self
  end
  def from_now
    DateTime::now + self
  end
end

10.days.from_now
42.days.from_now
```

Die Erweiterung von Klassen ist auch im Zusammenspiel mit Unit-Tests von Nutzen. Denkbar ist z.B. die Erweiterung der Klasse `Date` als eine Art Mock- Objekt[8], so dass sie für Testfälle ein konstantes Datum liefert. Andernfalls ist das Ergebnis des Tests womöglich abhängig vom Tag der Ausführung oder durch andere Art und Weise simuliert werden.

1.16 Module

Ruby bietet die Definition von Modulen, die für zwei Dinge gut sind. Zum einen führen Module einen Namensraum ein unter dem Module, Klassen und Methoden zusammengefasst werden. Zum anderen können sie als *Mixin* in Klassen eingefügt werden. Module sind Instanzen `Module`, die direkt von `Object` erbt. Genau wie Klassen sind Module offen für Erweiterungen.

1.16.1 Namensraum

Häufig besteht die Notwendigkeit, eine Sammlung von Methoden unter einer logischen Einheit zusammenzulegen. Beispielsweise eine Reihe von Hilfsmethoden im Umgang mit Strings:

[7] Die Erweiterung von Klassen wird auch von Rails bevorzugt. Für das Beispiel sieht die Umsetzung im Detail anders aus.
[8] Eine Art Stellvertreter

```
module StringUtil
  def StringUtil.arrayToString(array)
    ...
  end
  ...
end

a = [1, 2, 3]
s = StringUtil.arrayToString(a) # => "[1,2,3]"
```

Eine Moduldefinition beginnt mit dem Schlüsselwort `module` gefolgt vom Modulnamen und einem abschließenden `end`. Genau wie Klassen beginnen Module immer mit einem Großbuchstaben. Besteht der Name aus mehreren Wörtern, beginnt jedes Wort mit einem Großbuchstaben. Die Vergabe der Dateinamen ist analog der von Klassen.

Module können auch Konstanten enthalten. Wie auch bei Klassen muss beim Zugriff außerhalb des Moduls der Modulname als Präfix verwendet werden:

```
module StringUtil
  DEFAULT_SEPARATOR = ','
  ...
end

StringUtil::DEFAULT_SEPARATOR
```

Modulmethoden erhalten als Präfix den Modulnamen. Dadurch ist die Methode Teil des Namensraumes zum Modul und wird von einer Methode mit demselben Namen unterschieden:

```
module A
  def A.foo
  end
end

module B
  def B.foo
  end
end

A.foo
B.foo
```

Module können auch Klassen oder Module selbst enthalten, um diese mit einem Namensraum zu versehen:

```
module XML
  class Document
  end

  class Tag
  end
```

```
  class Attribute
  end
end

t = XML::Tag.new("<person>")
a = XML::Attribute.new("name","Ralf")
```

1.16.2 Mixin

Module sind keine Klassen und haben daher auch keine Instanzen. Sie können aber dennoch Instanzmethoden definieren. Dies mag im ersten Moment sinnlos erscheinen, da keine Instanzen existieren, auf denen diese Methoden aufgerufen werden können:

```
module LogUtil
  def log(msg)
    ...
  end
end

x = LogUtil.new       # geht nicht, da von Modulen keine
x.log(a)              # Instanzen erzeugt werden können

LogUtil.log(a)        # geht nicht, da Instanzmethoden
                      # des Moduls und keine Modulmethode
# und nun?
```

Die Bedeutung von Instanzmethoden eines Modul wird im Zusammenspiel mit Klassen deutlich. Module können als sogenannte *Mixin* mit `include` [9] in Klassen eingebunden werden. Dadurch werden die Instanzmethoden eines Moduls zu Instanzmethoden der Klasse. Die Methoden verhalten sich sozusagen wie Instanzmethoden einer Superklasse:

```
class Project
  include LogUtil

  def initialize(name)
    log("Projekt #{name} erzeugt")
                          # log wird zur Instanz-
    ...                   # methode der Klasse
  end
end
```

Module definieren häufig Instanzmethoden, die für viele Klassen von Nutzen sind. Ein gutes Beispiel hierfür bieten Standardmodule, wie z.B. `Comparable`. Es stellt ein Reihe von Vergleichsmethoden, wie z.B. <, >, == usw. zur Verfügung.

[9] Der Befehl `include` lädt im Gegensatz zu `require` (vgl. Abschnitt 1.3) das Modul nicht in die Klasse. Stattdessen wird eine Referenz auf das Modul abgelegt. Änderungen am Modul wirken sich somit auf alle Klassen aus, die das Modul eingebunden haben.

Alle diese Methoden nutzen intern die Vergleichsmethode <=>. Diese wiederum liefert je nachdem, ob eine Instanz kleiner, gleich oder größer einer anderen ist, -1, 0 oder 1 zurück.

Die Methode <=> ist selbst aber gar nicht im Modul enthalten. Sie wird jeweils von der Klasse bereitgestellt, die das Modul einbindet. Denn nur die Klasse weiß, wie zwei ihrer Instanzen zu vergleichen sind. Beispiele solcher Klassen sind `String`, `Time` oder `Fixnum`. Durch das Einbinden des Moduls stehen in diesen Klassen neben der eigenen <=> automatisch die Vergleichsmethoden <, > und == zur Verfügung. Sie müssen nicht in jeder Klasse neu definiert werden.

Eine Klasse kann beliebig viele Module einbinden. Hierdurch ist auch eine Art Mehrfachvererbung möglich. Existieren Instanzmethoden der Klasse und eingefügter Module mit gleichem Namen, unterscheidet Ruby diese wie folgt. Zuerst wird das Vorhandensein einer Instanzmethode der Klasse selbst geprüft. Existiert keine solche Methode, wird im zweiten Schritt in den eingebundenen Modulen nachgeschaut. Dabei wird mit dem zuletzt eingefügten begonnen. Ist die Instanzmethode auch hier nicht definiert, beginnt die Suche in der Superklasse.

Um Instanzmethoden eines Modul nicht nur innerhalb einer Klasse oder eines Modul, sondern überall verwenden zu können, müssen diese in Modulmethoden gewandelt werden. Hierzu steht die Methode `module_function` zur Verfügung:

```
module LogUtil
  def log(msg)
    ...
  end

  module_function :log
end

# als Instanzmethode ...
include LogUtil
log("...")

# ... oder Modulmethode
LogUtil.log("...")
```

1.17 Ausnahmebehandlung

Ausnahmen werden in Ruby durch die Klasse `Exception` oder eine ihrer Unterklassen repräsentiert. Das Werfen einer Ausnahme erfolgt in Ruby durch das Schlüsselwort `raise`. Dabei werden der Typ der Ausnahme und die Fehlermeldung als Parameter übergeben. Wird die Ausnahme weggelassen, so verwendet Ruby automatisch den Typ `RuntimeError`:

```
raise IOError, "buffer ist leer"
raise "buffer ist leer"
```

Zum Fangen einer Ausnahme dient das Schlüsselwort `rescue`. Es wird innerhalb eines in `begin` und `end` eingeschlossenen Blocks angegeben:

```
begin
  # hier wird ggf. eine Ausnahme geworfen
  ...
rescue IOError => ioe
  # Ausnahme fangen
  ...
end
```

Innerhalb einer Methodendefinition kann `rescue` auch direkt und ohne `begin` und `end` verwendet werden:

```
def read
  # hier wird ggf. eine Ausnahme geworfen
  ...
rescue IOError => ioe
  # Ausnahme fangen
  ...
end
```

Für einen einzeiligen Ausdruck kann es auch direkt hinter dem Ausdruck vorkommen.

```
42 / 0 rescue puts "ups!"
```

Die Instanz der gefangenen Ausnahme wird der hinter `rescue` angegebenen Variablen zugewiesen. Der Fehlertext wird über die Methode `message` oder `to_s` ausgegeben. Der Ablauf der Methodenaufrufe bis zur Stelle der Ausnahme, steht über die Methode `backtrace` zur Verfügung. Eine gefangene Ausnahme kann bei Bedarf erneut per `raise` geworfen werden:

```
begin
  # hier wird ggf. eine Ausnahme geworfen
  ...
rescue IOError => ioe
  puts "Exception: #{ioe.message}"
  puts ioe.backtrace
  raise ioe
end
```

Wird keine Ausnahmeklasse hinter `rescue` angegeben, verwendet Ruby implizit die Klasse `StandardError`:

```
begin
  # hier wird ggf. eine Ausnahme geworfen
  ...
rescue
  # fängt StandardError und alle Subklassen
  puts "Exception: #{$!}"
end
```

Zum Fangen einer Ausnahme können auch mehrere `rescue`-Anweisungen angegeben werden. So ist eine Unterscheidung der Ausnahmeart und ein entsprechendes Reagieren möglich. Damit Ruby die geworfene Ausnahme der richtigen `rescue`-Anweisung zuordnen kann, vergleicht es von oben nach unten im Quellcode jeweils den Typ der angegebenen Ausnahme mit dem der geworfenen. Entspricht der Ausnahmetyp einer `rescue`-Anweisung der geworfenen Ausnahme oder einem Supertyp so verzweigt Ruby in diesen `rescue`-Block. Daher müssen Subklassen immer vor ihrer Superklasse (z.B. `IOError` vor `StandardError`) angeben werden. Andernfalls wird der `rescue`-Block der Subklassen niemals angesprungen:

```
begin
  # hier wird ggf. eine Ausnahme geworfen
  ...
rescue EOFError
  # Subklasse von IOError, muss somit vor IOError
  # angegeben werden
rescue IOError
  # Subklasse von StandardError, muss somit vor
  # StandardError angegeben werden
rescue
  # würde EOFError und IOError fangen, wenn diese
  # nicht (wie hier) vorher angegeben sind
end
```

Um sicherzustellen, dass ein bestimmter Code am Ende des Blocks ausgeführt wird, kann das Schlüsselwort `ensure` angegeben werden:

```
def read_config
  f = nil
  begin
    f = File.open("config.cfg")
    ...
  rescue
    puts "Fehler: #{$!}"
  else
    # keine Ausnahme, weiter geht's
    ...
  ensure
    # wird ausgeführt, egal ob eine Ausnahme
    # gefangen wurde oder nicht.
    f.close if f
  end
end
```

Fehler in Systemfunktionen (z.B. `open`) werden in Ruby über eine der `Errno`-Klassen repräsentiert, die von `SystemCallError` erben.

Neben den vordefinierten Ausnahmen können auch eigene Ausnahmen definiert werden. Dabei sollte von `StandardError` oder einer Subklasse von `StandardError` geerbt werden. Andernfalls wird die Ausnahme nicht per

Default gefangen, sondern muss immer explizit hinter der `rescue`-Anweisung angegeben werden:

```
class ParseException < StandardException
  attr :position
  def initialize(pos)
    position = pos
  end
end
...
rails ParseException.new(42), "unexpected '@'"
```

Im Gegensatz zu Java gibt es in Ruby keine *Checked Exceptions*. Daher werden Methoden, die Ausnahmen werfen nicht explizit durch Angabe der Ausnahmeklassen in der Methodendefinition gekennzeichnet.

1.18 Ein- und Ausgabe

Die Ein- und Ausgabe von Daten erfolgt in Ruby über die Klasse `IO`. Diese stellt alle möglichen Methoden zum Lesen und Schreiben aus und in einen Datenstrom zur Verfügung. Im einfachsten Fall sieht der Aufruf zum Lesen von Zeilen aus einer Datei z.B. wie folgt aus:

```
IO.foreach("config.cfg") { |line| puts line }
```

Normalerweise wird am häufigsten mit Dateien als Ein- oder Ausgabemedium gearbeitet, so dass die Klasse `File` die meiste Verwendung findet. Diese erbt von `IO` und stellt zusätzliche Methoden für die Arbeit mit Dateien bereit. Ein typischer Code für das Lesen aus einer Datei könnte wie folgt aussehen:

```
file = File.open("config.cfg")
lines = file.readlines
file.close

lines.each do |line|
  puts line
end
```

Eine bessere Alternative stellt die Verwendung eines Blocks (vgl. 1.14) dar, da in diesem Fall die Datei beim Verlassen des Blocks automatisch geschlossen wird:

```
File.open("config.cfg") do |file|
  file.readlines.each do |line|
    puts line
  end
end
# automatisches close
```

Das Schreiben erfolgt im Prinzip auf die gleiche Art mit Hilfe der Methode `write`. Ein Zeilenumbruch muss in diesem Fall explizit angegeben werden:

```
File.open("config.cfg", "w") do |file|
  lines.each do |line|
    file.write(line + "\n")
  end
end
```

Eine formatierte Ausgabe kann über die von der Sprache `C` bekannte Methode `printf` erfolgen.

```
File.open("config.cfg", "w") do |file|
  file.printf "<%04d>\n", 42  # => "<0042>"
end
```

Die Klasse `File` stellt eine Reihe hilfreicher Klassenmethoden für die Arbeit mit Dateien zur Verfügung:

```
File.exist?("config.cfg")
File.file?("config.cfg")
File.directory?("config.cfg")
File.dirname("/home/thomas/config.cfg") # "/home/thomas"
File.basename("config.cfg")             # "config.cfg"
File.basename("config.cfg", ".cfg")     # "config"
File.extname("config.cfg")              # ".cfg"
# usw.
```

1.19 Reguläre Ausdrücke

Das folgende Kapitel behandelt reguläre Ausdrücke in Ruby. Das Kapitel ist keine Einführung in das Thema reguläre Ausdrücke. Es werden daher nur die wesentlichen Aspekte bei der Verwendung in Ruby beschrieben.

Reguläre Ausdrücke werden eingesetzt, um zu prüfen, ob ein String einem definierten Muster entspricht. Im Gegensatz zu vielen anderen Sprachen sind reguläre Ausdrücke in Ruby über die Klasse `Regexp` Teil der Standardtypen. Eine Instanz der Klasse wird explizit durch Aufruf des Konstruktors oder implizit durch die Angabe von `/Muster/` oder `/%r{Mustern}/` erzeugt:

```
re = Regexp.new("Java|Ruby")
re = /Ruby/
re = %{Ruby}
```

Der Vergleich des Musters mit einem String erfolgt über die Methoden `=~` oder `!~`. Erstere liefert die Position in der Zeile, an der das Muster übereinstimmt, oder `nil`, wenn keine Übereinstimmung vorliegt. Letztere prüft auf das nicht Vorhandensein und liefert daher `true`, wenn keine Übereinstimmung vorliegt, andernfalls `false`:

```
if "It's Ruby" =~ /Ruby/  # => 5
  puts "Ruby unterstützt reguläre Ausdrücke!"
end
```

```
# => Ruby unterstützt reguläre Ausdrücke!

if "It's Java" !~ /Ruby/  # => true
  puts "Sie verwenden aber nicht Ruby!"
end
# => Sie verwenden aber nicht Ruby!
```

Die folgende Hilfsmethode kennzeichnet die Stelle im String durch <>, an der ein Muster gefunden wurde [10]. Sie verwendet dazu globale Variablen, die den Bereich vor der Übereinstimmung, die Übereinstimmung und den Bereich nach der Übereinstimmung ausgeben. Die globalen Variablen werden automatisch bei einem Treffern gesetzt. Wir gehen später näher darauf ein:

```
def show_re(str, re)
  if str =~ re
    puts "#{$`}<#{$&}>#{$'}"
  else
    puts = "no match"
  end
end

show_re("I liebe Ruby!", /liebe/)  # => I <liebe> Ruby!
show_re("I liebe Ruby!", /hasse/)  # => no match
```

Sonderzeichen

In einem regulären Ausdruck werden die Zeichen `*`, `+`, `?`, `^`, `$`, `\`, `|`, `.`, `(`, `)`, `{`, `}`, `[` und `]` als Sonderzeichnen interpretiert. Alle anderen Zeichen stehen für sich selbst. Um das Sonderzeichnen als gewöhnliches Zeichen im Muster zu interpretieren, ist es per Backslash zu escapen:

```
show_re("a+b=c", /a+b/)  # => no match
show_re("a+b=c", /a\+b/) # => <a+b>=c
```

Wiederholungen

Wiederholung von Teilen des Musters werden durch Sonderzeichen definiert. Dabei steht + für ein oder mehrmaliges, `*` für kein oder mehrmaliges und `?` für kein oder einmaliges Vorkommen des Teilbereichs:

```
show_re("a",   /ab+/)  # => no match
show_re("ab",  /ab+/)  # => <ab>
show_re("abb", /ab+/)  # => <abb>
show_re("a",   /ab*/)  # => <a>
show_re("ab",  /ab*/)  # => <ab>
show_re("abb", /ab*/)  # => <abb>
show_re("a",   /ab?/)  # => <a>
show_re("ab",  /ab?/)  # => <ab>
show_re("abb", /ab?/)  # => <ab>b
```

[10] Die Idee wurde *Dave Thomas: Programming Ruby* entnommen.

Darüber hinaus kann die Anzahl der erlaubten Wiederholungen exakt festgelegt werden. Dazu ist die Mindest- und Maximalzahl hinter dem Teilbereich des Musters über geschweifte Klammern anzugeben:

```
show_re("ab",   /ab{2}/)    # => no match
show_re("abb",  /ab{2}/)    # => <abb>
show_re("abbb", /ab{2}/)    # => <abb>b
show_re("ab",   /ab{2,}/)   # => no match
show_re("abb",  /ab{2,}/)   # => <abb>
show_re("abbb", /ab{2,}/)   # => <abbb>
show_re("ab",   /ab{1,2}/)  # => <ab>
show_re("abb",  /ab{1,2}/)  # => <abb>
show_re("abbb", /ab{1,2}/)  # => <abb>b
```

Anker

Um ein Muster am Beginn oder Ende einer Zeile zu finden, sind die Zeichen ^ für Beginn und $ für Ende der Zeile zu verwenden:

```
s = "Ruby by Matz\nMatz ist cool!"
show_re(s, /\bby/)    # => Ruby <by> Matz\nMatz ist cool!
show_re(s, /\Bby/)    # => Ru<by> by Matz\nMatz ist cool!
show_re(s, /^Ruby/)   # => <Ruby> by Matz\nMatz ist cool!
show_re(s, /^Matz/)   # => Ruby by Matz\n<Matz> ist cool!
show_re(s, /Matz/)    # => Ruby by <Matz>\nMatz ist cool!
show_re(s, /^Matz/)   # => Ruby by Matz\n<Matz> ist cool!
show_re(s, /Matz$/)   # => Ruby by <Matz>\nMatz ist cool!
```

Durch Angabe des Zeichens \A wird das Muster nur am Beginn des gesamten Textes und nicht am Beginn einzelner Zeilen gefunden. Über die Zeichen \Z und \z wird ein Muster am Textende erkannt. Endet der Text mit einem Zeilenumbruch, wird das Muster nur über \Z erkannt:

```
s = "Ruby by Matz\nMatz ist cool!"
show_re(s, /\ARuby/)  # => <Ruby> by Matz\nMatz ist cool!
show_re(s, /\AMatz/)  # => no match
show_re(s, /Matz\z/)  # => no match
show_re(s, /Matz\Z/)  # => no match
show_re(s, /cool!\z/) # => Ruby by Matz\nMatz ist <cool!>
show_re(s, /cool!\Z/) # => Ruby by Matz\nMatz ist <cool!>

s = "Ruby by Matz\nMatz ist cool!" + "\n"
show_re(s, /cool!\z/) # => no match
show_re(s, /cool!\Z/) # => Ruby by Matz\nMatz ist <cool!>
```

Um im Muster Wortgrenzen zu definieren, bzw. diese auszuschließen, sind die Zeichen \b und Zeichen \B zu verwenden:

```
s = "Ruby by Matz\nMatz ist cool!"
show_re(s, /\bby/)    # => Ruby <by> Matz\nMatz ist cool!
show_re(s, /\Bby/)    # => Ru<by> by Matz\nMatz ist cool!
```

Tabelle 1.4: Zeichenklasse

Zeichenklasse	Abkürzung	Bedeutung
[0-9]	\d	Eine Ziffer
^0-9]	\D	Alle Zeichen außer Ziffern
A-Za-z_0-9]	\w	Ein Wortzeichen
^A-Za-z_0-9]	\W	Alle Zeichen außer Wortzeichen
\f\n \r\s\t]	\s	Ein Leerzeichen
^\f\n \r\s\t]	\S	Alle Zeichen außer Leerzeichen
	\b	Wortgrenze
	\B	Keine Wortgrenze

Zeichenklassen

Um ein Zeichen gegen eine Menge von Zeichen zu prüfen, können Zeichenklassen definiert werden. Alle Zeichen der Klasse werden dazu in eckige Klammern gefasst. Sonderzeichen, wie z.B. `*`, `+` und `?` werden innerhalb der Klammer zu gewöhnlichen Zeichen:

```
show_re("123",   /[0-9]?/)  # => <1>23
show_re("a+b-c", /[*\/+-]/) # => a<+>b-c
```

Die Negierung erfolgt über das Zeichen `^` direkt am Start. Dadurch werden nur Zeichen gefunden, die nicht in der Klasse enthalten sind:

```
show_re("123",   /[0-9]/)   # => <1>23
show_re("123",   /[^0-9]/)  # => no match
```

Für einige Zeichenklasse existieren Abkürzungen. Die Angabe von `\s` definiert z.B. alle Leerzeichen, wie Leerzeichen, Tabulator oder Zeilenumbruch, etc. Die Angabe von `\w` definiert alle Buchstaben, Ziffern und den Unterstrich, d.h. alle Wortzeichen. Eine Übersicht ist in Tabelle 1.4 gegeben.

```
show_re("42a",    /\d/)   # => <4>2a
show_re("foo42!", /\w/)   # => <f>oo42!
show_re("foo42!", /\w+/)  # => <foo42>!
```

Gruppering und Auswahl

Die Gruppierung von Teilen des Musters wird durch runde Klammern ermöglicht, Alternativen durch den senkrechten Strich `|`:

```
show_re("ababc", /ab+/)
show_re("ababc", /(ab)+/)

show_re("abc", /a|bc+/)   # => <a>bc
show_re("abc", /(a|b)c+/) # => a<bc>
show_re("ac", /(a|b)c+/)  # => <ac>
show_re("bc", /(a|b)c+/)  # => <bc>
```

Tabelle 1.5: Globale Variablen und MatchData

Variable	MatchData	Bedeutung
$~	MatchData	Alle Information zur Übereinstimmung
$&	MatchData[0]	Übereinstimmung
$`	MatchData#pre_match	Teil vor der Übereinstimmung
$'	MatchData#post_match	Teil nach der Übereinstimmung
$+	-	Letzte Gruppe der Übereinstimmung
$1 bis $9	MatchData[1-9]	Erste bis neunte Gruppe der Übereinstimmung

Globale Variablen und MatchData

Die Übereinstimmung eines Strings mit einem Mustern kann auch über die Methode `match` der Klasse `Regexp` erfolgen. Diese liefert im Gegensatz zu `=~` oder `!~` im Erfolgsfall eine Instanz der Klasse `MatchData`, andernfalls `nil` zurück. Hierin enthalten sind alle Informationen über die Übereinstimmung. Bei Bedarf kann pro Prüfung eine eigene Instanz verwendet werden:

```
re = /(\d).*(\d).*(\d)/
md = re.match("a=1; b=2; c=3; # yo!")
md[0]          # => 1; b=2; c=3
md[1]          # => 1
md[2]          # => 2
md[3]          # => 3
md.pre_match   # => a=
md.post_match  # => ; # yo!
$+             # => 3
$1             # => 1
$2             # => 2
```

Bei jeder Übereinstimmung werden auch eine Reihe von globalen Variablen gesetzt. Die Instanz von `MatchData` steht z.B. über die globale Variable `$~` zur Verfügung. Weitere Variablen enthalten einzelne Werte der Instanz. Eine Übersicht finden Sie in Tabelle 1.5. Beachten Sie, dass die globalen Variablen bei jeder neuen Prüfung neu gesetzt werden.

1.20 Nicht behandelte Bestandteile

Die Sprache Ruby bietet noch einiges mehr. Wir haben in diesem Kapitel nur die aus unserer Sicht wichtigsten Themen für die Programmierung mit Ruby (on Rails) kurz beschrieben. Nicht behandelt wurde die folgenden Themen:

- Reflection
- Prozess und Threads
- Grafische Frontends
- Ruby Safe

- Verteilte Objekte
- Erweiterung durch eigene C-Programme
- Installation und Verwaltung mit RubyGem
- Dokumentation mit RDoc
- Werkzeuge: Interactive Ruby, Debugger, Profiler

Weitere tiefer gehende Informationen finden Sie in den unter 1.1 aufgeführten Ressourcen.