Learn to Program - Programmieren lernen

Chris Pine
übersetzt von Anja Stiedl

Inhalt

Ein guter Start für angehende Programmierer

Es begann wohl in 2002. Ich habe darüber nachgedacht, Programmieren zu unterrichten und dass Ruby eine tolle Sprache ist, mit dem Programmieren zu beginnen. Viele waren begeistert von Ruby, weil es so mächtig war, sehr elegant und das Programmieren unheimlich viel Spaß machte. Darüber hinaus fand ich, es wäre ein toller Einstieg in die Programmierung.

Unglücklicherweise gab es damals noch kaum Dokumentation über Ruby auf Anfänger-Niveau. In der Community wurde öfter von einem Tutorial "Ruby für Neulinge" gesprochen, d.h. im Allgemeinen ein Tutorial, wie Programmierung unterrichtet werden sollte. Je mehr ich darüber nachdachte, desto mehr konnte ich darüber mitreden (was mich selbst sehr erstaunte). Und schließlich kam ich zu dem Entschluss, dass ich das Tutorial schreiben sollte, statt immer nur darüber zu reden. Also habe ich es getan!

Und es war nicht sehr gut. Ich hatte viele theoretische Ideen, aber ein tolles Tutorial für Nicht-Programmierer zu schreiben, war in der tat viel schwieriger, als ich es vorher angenommen hatte. (Es erschien mir zwar gut, aber ich wußte ja auch schon, wie man programmiert.)

Mir kam zu Hilfe, dass es leicht war, mich zu kontaktieren und ich viele Rückmeldungen bekam, wo die Leser hängen blieben und Fragen ungeklärt blieben. Wann immer mehrere Leser an derselben Stelle Probleme hatten, habe ich diese neu geschrieben. Es war viel Arbeit, aber langsam wurde es besser.

Nach ein paar Jahren war das Tutorial richtig gut. :-) Sogar so gut, dass ich es für fertig erklärte, und mich selber anderen Themen widmen wollte. Genau zu diesem Zeitpunkt erhielt ich die Möglichkeit, das Tutorial in ein Buch umzuschreiben. Da es ja praktisch fertig war, dachte ich, es wäre kein Problem. Ich müßte nur ein paar Stellen verschönern, mehr Übungen einbauen, vielleicht auch mehr Beispiele, ein paar zusätzliche Kapitel, es von 50 weiteren Test-Lesern ausprobieren lassen...

Ich brauchte ein weiteres Jahr, aber nun denke ich, es ist wirklich, wirklich gut geworden durch die vielen Hundert tapferen Menschen, die mir beim Schreiben geholfen haben.

Was ihr hier findes, ist das Original-Tutorial, quasi unverändert seit 2004. Für die neuste und beste Version rate ich euch zum Buch.

Anregungen für Dozenten

Es gibt ein paar grundlegende Prinzipien, die ich versucht habe umzusetzen. Ich denke, dass sie den Lernprozess erleichtern; Programmieren lernen ist an und für sich schon schwer genug. Wenn man Programmieren unterrichtet oder den Lernprozess

eines Lernenden in den Programmier-Himmel begleitet, so helfen diese Ansätze sicher.

Erstens trenne ich verschiedene Konzepte soweit dies möglich ist, damit der Lernende zu jedem Zeitpunkt nur ein Konzept erarbeiten muss. Dies ist am Anfang schwer, mit Übung aber immer leichter. Manche Sachverhalte müssen vor anderen vermittelt werden, doch ich war erstaunt, wie wenig Hierarchie sich aus den Abhängigkeiten ergab. Schließlich musste ich ich mir eine Reihenfolge der Themen überlegen und ich habe sie so angeordnet, dass jedes neue Thema sich aus dem vorherigen ergab.

Ein weiterer Ansatz war, dass ich immer nur eine Möglichkeit, etwas zu tun, vorgestellt habe. Das ist ein ganz klarer Vorteil in einem Tutorial für reine Programmier-Anfänger. Einerseits weil das Erlernen eines Weges immer leichter ist, als das Erlernen von zwei Wegen, andererseits -und das ist wohl der größere Vorteil- weil ein Programmierer um so kreativer und überlegter vorgehen muss, je weniger Möglichkeiten er kennt. Da eine Hauptaufgabe der Programmierung im Lösen von Problemen besteht, ist es wichtig, dies immer und auf jedem Level zu fördern.

Ich habe versucht, Programmier-Konzepte quasi huckepack mit den Konzepten zu vermitteln, die jeder Mensch intuitiv hat; also die Ideen dahinter so darzustellen, dass die Intuition die Arbeit übernimmt und nicht das Tutorial. Die objektorientierte Programmierung verfolgt diesen Ansatz ohnehin schon. Somit kann ich recht schnell Begriffe wie 'Objekte' und 'Arten von Objekten' verwenden, meist lasse ich sie unterschwellig einfließen. Ich mache bewusst keine Aussagen wie "alles in Ruby ist ein Objekt" oder "Zahlen und Zeichenketten sind Arten von Objekten", weil diese Aussagen einem Programmier-Anfänger nicht sagen. Stattdessen spreche ich über Zeichenketten (nicht "Zeichenketten-Objekte") und irgendwann schwenke ich um zu "Objekten", was einfach nur bedeutet wie "irgendwelche Dinge in unseren Programmen". Die Tatsache, dass all diese Dinge in Ruby Objekte sind, unterstützt sehr das unauffällige Einbringen von Fachbegriffen.

Obwohl ich eigentlich die typische objektorientierte Sprache vermeiden wollte, wollte ich doch sicherstellen, dass der Leser, wenn er einen Begriff lernt, den richtigen lernt. (Ich will nicht, dass er es zweimal lernen muss.) So spreche ich von Zeichenketten, nicht von Text. Auch Methoden müssen irgendwie benannt werden, also nenne ich sie "Methoden".

In Bezug auf die Übungen, denke ich, dass ich schon ein paar gute integriert habe, aber man kann nie zu viele haben. Ehrlich gesagt, glaube ich dass ich die Hälfte der zeit damit verbracht habe, lustige, interessante Übungen zu entwickeln. Langweilige Aufgaben töten die Freude am Programmieren, während die perfekte Übung im neuen Programmierer ein Jucken erzeugt, an dem er einfach kratzen muss. Kurz gefasst kann man gar nicht zu viel Zeit in die Suche nach den richtigen Übungen stecken.

Über das Original

Die Seiten des Original-Tutorials (des englischen Originals!) werden mit natürlich mit einem großen Ruby-Programm generiert. :-) Alle Beispiele werden automatisch gestartet und die dargestellt Ausgabe ist die die Ausgabe, die sie erzeugten. Ich denke, das ist die beste, leichteste und sicherlich die coolste Art sicherzustellen, dass der dargestellte Code genau so funktioniert, wie ich es beschrieben habe. So besteht keine Gefahr, dass ich etwas falsch kopiert habe oder vergessen habe, etwas zu testen; es ist alles getestet! (Kommentar: dies gilt nicht für die deutsche Übersetzung. Sie ist ein reines pdf und kann durchaus Fehler enthalten. Bitte einfach mitteilen, wenn ihr welche findet: dann können wir es gemeinsam verbessern. - Danke!)

Danksagung

Schließlich möchte ich noch allen in der Ruby-Talk Mailing Liste danken für ihre Anregungen und Ermutigungen danken, ebenso allen Test-Lesen für ihre Hilfe, dieses Buch viel besser zu machen als ich es alleine geschafft hätte, meiner geliebten Ehefrau dafür, dass sie mein Haupt-Testleser/Tester/Versuchskaninchen/Muse ist, Matz für diese wundervolle Sprache und den pragmatischen Programmierern, dass sie mir davon erzählt haben - und dass sie mein Buch herausgegeben haben!

Wenn du Fehler oder Rechtschreibfehler entdeckst oder irgendwelche Kommentare oder Vorschläge hast oder etwa gute Übungen, die ich integrieren könnte, dann lass es mich wissen.

Vorwort zur deutschen Übersetzung

Dies ist die deutsche Übersetzung des Tutorials **Learn To Program** von **Chris Pine**.

Das Tutorial richtet sich an reine Programmieranfänger, die sich mit ein paar Grundkonzepten der Programmierung vertraut machen wollen. Es setzt nichts voraus, erklärt jeden Schritt im Detail.

Für Leser, die bereits Vorkenntnisse im Programmieren haben, kann es aus diesem Grunde langwierig erscheinen, da viele Fachbegriffe oder Sprachkonstrukte aus anderen Programmiersprachen leicht übertragen werden können.

Verwendet wird die Programmiersprache Ruby, heutzutage in aller Munde durch RubyOnRails. Diese Sprache bietet einen optimalen Einstieg in die Programmierung, da sogenannte Vorwärtsbezüge, also sowas wie "das erklären ich später", großteils vermieden werden können, wodurch Beispiele klein und übersichtlich bleiben, und sich folglich der Leser auf das aktuelle Thema konzentrieren kann. Zudem sind in Ruby viele sehr moderne Programmierparadigmen implementiert und können schnell erlernt werden.

Ich habe mich zur Übersetzung entschlossen, da ich in der Erwachsenenbildung wiederkehrend mit reinen Programmier-Anfängern zu tun habe, denen Anleitungen auf Englisch eine zusätzliche Hürde bereiten. Das Tutorial spiegelt viele Grundlagen-Aspekte wider, die ich in meinen Schulungen integriere. Zudem bietet es schrittweises Vorgehen, detaillierte Beschreibungen und praktische Beispiele zum Ausprobieren, sowie Übungsaufgaben zur Intensivierung und Festigung.

Ein Wort zu deutschen Sonderzeichen: immer wieder gibt es Probleme mit den Umlauten (ä, ö, ü, Ä, Ö, Ü) und dem scharfen s (ß). Wenn du diese Fehlermeldung erhältst:

```
invalid multibyte char (US-ASCII)
```

so füge in deinen Programmen als erste Zeile ein:

```
# encoding: utf-8
```

Damit sagst du dem Ruby-Interpreter, dass das Programm internationale UTF-8-Codierung der Zeichen verwendet, und er kann damit umgehen. Alternativ kannst du diese Zeichen vermeiden, indem du an ihrer Stelle 'ae', 'oe' und so weiter verwendest.

Wer Ideen oder Anregungen zum Tutorial oder zur Übersetzung hat, oder sich austauschen möchte über Erfahrungen in der Erwachsenenbildung, kann gerne mit mir in Kontakt treten.

Nun wünsche ich viel Spaß und Erfolg mit Ruby!

Anja Stiedl

Aller Anfang ist schwer...

Wenn man einen Computer programmiert, muss man mit ihm in einer Sprache sprechen, die er versteht: einer Programmiersprache. Es gibt inzwischen sehr viele Sprachen und viele davon sind phantastisch. In diesem Tutorial habe ich meine Lieblingssprache gewählt, *Ruby*.

Außer dass sie meine liebste Sprache ist, ist Ruby auch noch die einfachste Sprache, die ich kenne (und ich kenne einige). Und dies ist der eigentliche Grund dafür, dass ich dieses Tutorial schreibe: ich habe mich nicht erst entschlossen, ein Tutorial zu schreiben und danach meine Lieblingssprache Ruby als Thema gewählt, sondern ich fand Ruby so leicht, dass ich beschlossen habe, es sollte darüber ein Tutorial für Anfänger geben. Rubys Einfachheit hat dieses Tutorial initiiert, nicht nicht Tatsache, dass es mein Favorit ist. (Ein ähnliches Tutorial für die Sprachen C++ oder Java würde Hunderte von Seiten beanspruchen.) Aber denke nicht, dass Ruby eine reine Anfängersprache ist, nur weil sie leicht ist! Sie ist die stärkste und mächtigste Programmiersprache auf professioneller Ebene, die es je gab.

Wenn man etwas in einer menschlichen Sprache schreibt, so nennt man dies Text. Wenn man etwas in einer Programmiersprache schreibt, so heißt dies Code. Ich integriere viele Code-Beispiele in dieses Tutorial, meist sind es komplette Programme, die du auf deinem Computer ablaufen lassen kannst. Um die Programm-Beispiele lesbarer zu gestalten, habe ich verschiedene Formatierungen verwendet:

Alles, was du als Programm-Code in eine Ruby-Datei eingibst, hat diese Darstellung,

```
Ruby-Programm
```

was das Programm beim Ablauf anzeigt, ist so formatiert,

```
Programm-Ausgabe
```

und deine Eingaben wiederum werden so dargestellt:

```
Programm-Ausgabe
```

Wenn du über etwas stolperst, was du nicht verstehst, oder wenn du eine Frage hast, die nicht beantwortet wurde, so schreib dir dies auf, und lies weiter! Wenn deine Frage bis zum letzten Kapitel nicht geklärt ist, so findest du dort einige Quellen, bei denen du nachfragen kannst. Da draußen gibt es viele tolle Leute, die dir gerne helfen; du musst nur wissen, wo du sie findest.

Aber jetzt solltest du Ruby erstmal installieren.

Windows-Installation

Die Windows-Installation von Ruby ist ein Kinderspiel. Erst einmal lädt man den Ruby Installer herunter. Wahrscheinlich sind verschiedene Versionen zur Auswahl. Dieses Tutorial basiert auf Version 1.8.4, also solltest du mindestens diese verwenden. (Ich würde die neuste verwenden!) Danach startest du das Installationsprogramm. Es wird dich nach dem Installationspfad fragen. Wenn es keinen speziellen Grund dagegen gibt, verwende am Besten das vorgeschlagene Installationsverzeichnis.

Um zu programmieren, musst du Programme schreiben und Programme starten. Dafür benötigst du einerseits einen Editor und zum anderen eine Kommandozeile.

Der Ruby-Installer liefert auch einen kleinen Editor, der SciTE heißt: der Scintilla Text Editor. Du kannst ihn im Startmenü starten. Wenn du deinen Code mit farbiger Unterstützung haben willst, lade dir auch diese Dateien herunter und speichere sie in deinem SciTE-Ordner (c:/Ruby/scite bei der Standard-Installation):

- Globale Eigenschaften
- Ruby Eigenschaften

Im Startmenü findest du unter Zubehör die Kommandozeile. Du musst zu dem Verzeichnis navigieren, in dem du deine Programme speichern willst. Verwende cd .., um eine Ebene höher zu gelangen, und cd ordnername, um in einen gewünschten Ordner zu wechseln. Um alles im aktuellen Verzeichnis zu sehen, tippe ls /ad.

... und damit hast du alles und bist bereit, Programmieren zu lernen!

Macintosh-Installation

Wenn du Mac OS X 10.2 (Jaguar) hast, dann ist Ruby bereits auf deinem System! Was könnte leichter sein? Unglücklicherweise kannst du Ruby auf Mac OS X 10.1 oder früher nicht nutzen, soweit ich weiß.

Um zu programmieren, musst du Programme schreiben und Programme starten. Dafür benötigst du einerseits einen Editor und zum anderen eine Kommandozeile.

Deine Kommandozeile findest du in der Terminalanwendung (unter Anwendungen / Zubehör).

Als Texteditor kannst du jeden beliebigen Editor verwenden, den du hast oder magst. Wenn du TextEdit nimmst, stelle sicher, dass du die Programme als reinen Text speicherst! Sonst wird es *nicht funktionieren*. Andere Möglichkeiten sind der Emacs, vi, pico, die du alle von der Kommandozeile aus erreichst.

... und damit hast du alles und bist bereit, Programmieren zu lernen!

Linux-Installation

Erst einmal solltest du überprüfen, ob Ruby schon installiert ist. Tippe which ruby. Wenn es irgendwas wie "/usr/bin/which: no ruby installed" (=kein Ruby installiert) sagt, musst du Ruby herunterladen, ansonsten überprüfe einmal, welche Version installiert ist, indem du ruby -v tippst. Wenn sie älter ist als der letzte stabile Build auf der eben erwähnten Download-Seite, willst du vielleicht einen Upgrade durchführen.

Wenn du root bist, brauchst du wahrscheinlich keine Hinweise, wie du Ruby installierst, ansonsten frag am Besten deinen Systemadministrator, es für dich zu installieren. (Auf diese Weise kann jeder im System es nutzen.)

Sonst kannst du es installieren, dass nur du es nutzen kannst. Schiebe die herunter geladene Datei in ein temporäres Verzeichnis, wie etwa $HOME/tmp. Lautet der Name ruby-1.6.7.tar.gz, so öffnest du die Datei mit tar zxvf ruby-1.6.7.tar.gz. Wechsel in das entstandene Verzeichnis (im Beispiel cd ruby-1.6.7).

Passe deine Installation an durch ./configure --prefix=$HOME. Danach tippe make und dein Ruby-Interpreter wird erzeugt werden. Im Anschluss tippe make install, was den Interpreter installiert.

Schließlich willst du noch $HOME/bin zu deinem system pfad hinzufügen. Dafür musst du %HOME/.bashrc anpassen. (Log dich dann aus und wieder ein, damit die Änderungen übernommen werden.) Teste als Abschluß noch deine Installation ruby -v. Wenn du nun mit der installierten Version einverstanden bist, kannst du die Dateien in $HOME/tmp löschen (oder dorthin verschieben, wo du sie für gewöhnlich aufbewahrst).

... und damit hast du alles und bist bereit, Programmieren zu lernen!

Zahlen

Nun, da alles richtig installiert ist, wollen wir ein ersten Programm schreiben! öffne deinen bevorzugten Editor und schreibe das folgende:

```
puts 1+2
```

Speichere das Programm (ja, das ist schon ein Programm!) als *calc.rb* (die Dateiendung .rb bezeichnet normalerweise Ruby-Programme). Nun starte dein Programm durch den Aufruf ruby calc.rb in der Kommandozeile. Es sollte eine 3 auf den Bildschirm schreiben.

Schau, programmieren ist gar nicht schwierig, oder etwa doch?

Einführung in puts

Also, was geschieht in dem Programm? Ich bin sicher, dass du erraten hast, was hinter 1+2 steckt... unser Programm ist dasselbe wie:

```
puts 3
```

puts schreibt auf den Bildschirm, was dahinter eingegeben wird.

Integer und Float

In den meisten Programmiersprachen (und Ruby ist keine Ausnahme) werden ganze Zahlen, also Zahlen ohne Komma, als *Integer* bezeichnet und Gleitkommazahlen, also Zahlen mit Komma heißen *Float* (oder genauer: floating point numbers).

Hier sind ein paar Integer-Zahlen:

```
5

-205

99999999999999999999999999999999

0
```

und hier ein paar Float-Zahlen:

```
54.321
```

```
0.0001

-205.3884

0.0
```

In der Praxis verwenden Programmierer meist keine Floats, sondern Integers. (Schließlich will niemand 7,4 Emails lesen oder 1,8 Webseiten ansehen oder 5,24 Lieblingslieder anhören...) Floats werden eher für akademische Zwecke verwendet (physikalische Experimente und so) und für 3-dimensionale Grafiken. Selbst die meisten kaufmännischen Programme verwenden Integers: man muss nur die Anzahl von Cent verwalten.

Einfache Arithmetik

Jetzt haben wir alle Grundlagen für einen einfachen Taschenrechner kennen gelernt. (Taschenrechner verwenden immer Floats, wenn du also willst, dass dein sich Computer wie ein Taschenrechner verhält, solltest du also auch Floats verwenden.) Zur Addition und Subtraktion verwenden wir + und -, wie gesehen. Zur Multiplikation verwenden wir * und zur Division /. Die meisten Tastaturen haben diese Tasten beim Nummernblock auf der rechten Seite der Tastatur. Wenn du eine kleine Tastatur oder ein Laptop hast, erhältst du / als Umschalt-7.

Versuchen wir, unser *calc.rb*-Programm ein wenig zu erweitern. Tipp das folgende ab und starte es.

```
puts 1.0 + 2.0

puts 2.0 * 3.0

puts 5.0 - 8.0

puts 9.0 / 2.0
```

(Die vielen Leerzeichen im Programm sind unbedeutend, sie machen es nur leichter lesbar.) Dies wird vom Programm zurückgegeben:

```
3.0

6.0

-3.0

4.5
```

Nun, das war nicht zu überraschend. Versuchen wir es mit Integers:

```
puts 1 + 2

puts 2 * 3

puts 5 - 8

puts 9 / 2
```

Fast dasselbe, stimmt's?

```
3

6

-3

4
```

Uuups... außer dem letzten! Immer wenn du mit Integers rechnest, bekommst du Integers als Antwort. Wenn dein Computer nicht die *richtige* Antwort darstellen kann, rundet er ab. (Damit ist 4 natürlich die richtige Antwort für 9/2, aber vielleicht nicht das, was du erwartet hast.)

Vielleicht wunderst du dich, welchen Sinn diese Art der Integer-Division hat. Angenommen, du gehst ins Kino, hast aber nur 20 Euro dabei. Sagen wir, ein Film kostet 8 Euro. Wie viele Filme kannst du sehen? 20/8... 2 Filme. In diesem Fall ist 2,5 sicher nicht die richtige Antwort. Du darfst sicher keinen halben Film ansehen... manche Dinge sind einfach nicht teilbar!

Nun experimentiere mit ein paar eigenen Programmen herum! Wenn du komplexere Ausdrücke schreiben willst, kannst du runde Klammern verwenden, z.B.

```
puts 5 * (12-8) + -15

puts 98+(59872 / (13*8)) * -52
```

ergibt:

```
5
```

```
-29802
```

Selber ausprobieren

Schreib ein Programm, das dir sagt...

- wie viele Stunden in einem Jahr sind?
- wie viele Minuten in einem Jahrzehnt sind?
- wie viele Sekunden du alt bist?
- wie viele Tafeln Schokoladen wirst du in deinem Leben essen? (Vorsicht: dieses Programm kann etwas länger brauchen...)

Hier ist eine schwierigere Frage:

- Wenn ich 1031 Sekunden alt bin, wie alt bin ich dann?

Wenn du genug mit Zahlen gespielt hast, schauen wir uns Buchstaben und Zeichenketten an...

Zeichenketten

Bisher haben wir alles über Zahlen gelernt, doch wie sieht's aus mit Buchstaben? Worten? Text? Eine Gruppe von Buchstaben in einem Programm werden als Zeichenkette oder *String* bezeichnet. (Du kannst dir gedruckte Buchstaben wie Perlen auf einer Kette aufgefädelt vorstellen.)

Hier sind ein paar Strings:

```
'Hallo.'

'Ruby ist super.'

'5 ist meine Glückszahl... und deine?'

'Snoopy sagt #%^?&*@! wenn er sich den Zeh anstößt.'

'     '

''
```

Wie du sehen kannst, können Strings auch Satzzeichen, Ziffern, Symbole und Leerzeichen enthalten... mehr als nur Buchstaben. Der letzte String enthält gar nichts; wir nennen ihn den leeren String oder Leerstring. Mit Zahlen haben wir puts verwendet, versuchen wir das auch mal mit ein paar Strings:

```
puts 'Hallo, Welt!'

puts ''

puts 'Auf Wiedersehen.'
```

ergibt:

```
Hallo, Welt!



Auf Wiedersehen.
```

Das hat ja gut hingehauen. Nun versuch es mit ein paar eigenen Strings!

Rechnen mit Strings

Ebenso wie man mit Zahlen rechnen kann, kann man auch mit Strings rechnen! Nun, jedenfalls so ähnlich.... auf jeden Fall kannst du Strings addieren. Addieren wir mal zwei Strings und schauen wir, was puts daraus macht:

```
puts 'Ich mag'+'Apfelkuchen'
```

```
Ich magApfelkuchen
```

Uuups! Da habe ich das Leerzeichen zwischen 'Ich mag' und 'Apfelkuchen' vergessen. Leerzeichen zählen eigentlich nicht, doch innerhalb von Strings sind sie wichtig. (Stimmt schon, was man so sagt: Computer machen nicht das, was du willst, sondern nur das, was du ihnen sagst.) Versuchen wir es noch einmal:

```
puts 'Ich mag ' + 'Apfelkuchen.'

puts 'Ich mag' + ' Apfelkuchen.'
```

```
Ich mag Apfelkuchen.

Ich mag Apfelkuchen.
```

(Wie du siehst, macht es keinen Unterschied, zu welchem String wir das Leerzeichen hinzugefügt haben.) Du kannst Strings addieren, aber auch multiplizieren! (Natürlich mit einer Zahl.) Pass auf:

```
puts 'zwinker ' * 4
```

```
zwinker zwinker zwinker zwinker
```

Wenn man darüber nachdenkt, macht dies Sinn. Immerhin bedeutet 7*3 dasselbe wie 7+7+7, also bedeutet 'muh'*3 dasselbe wie 'muh'+'muh'+'muh'

12 und '12'

Bevor wir weitermachen, sollten wir uns den Unterschied zwischen Zahlen und Ziffern verdeutlichen. 12 ist eine Zahl, während '12' eine Kette von zwei Zeichen ist, die zufällig Ziffern darstellen.

Spielen wir mal damit herum:

```
puts  12  +  12

puts '12' + '12'

puts '12  +  12'
```

```
24

1212

12  +  12
```

Und wie sieht's damit aus?

```
puts  2  *  5

puts '2' *  5

puts '2  *  5'
```

```
10

22222

2  *  5
```

Dies sind ziemlich eingängige Beispiele. Nun, wenn man nicht so vorsichtig trennt zwischen Strings und Zahlen, stolpert man leicht über...

Probleme

Hier hast du vielleicht schon das ein oder andere ausprobiert, das nicht funktionierte. Ansonsten sind hier ein paar Beispiele:

```
puts '12' + 12

puts '2' * '5'
```

```
#<TypeError: can't convert Fixnum into String>
```

Hmmm... eine Fehlermeldung. Die Ursache ist, dass man eine Zahl nicht zu einem String addieren kann, oder einen String mit einem anderen String multiplizieren kann. Es macht nicht mehr Sinn, als dies hier:

```
puts 'Betty' + 12

puts 'Fred' * 'John'
```

Sei dir auch immer im klaren über diese Feinheit: du kannst in einem Programm 'Schwein'*5 schreiben, weil es bedeutet, dass 5 mal der String 'Schwein' aneinander gereiht wird. Nun kannst du aber nicht 5*'Schwein' schreiben, weil es bedeuten würde, dass die Zahl 5 'Schwein'-mal verwendet werden soll, was einfach Quatsch ist.

Zum Schluss: was ist, wenn ich im Programm Wie geht's? schreiben will? Wir können dies probieren:

```
puts 'Wie geht's?'
```

Es funktioniert nicht; ich versuche nicht einmal, es zu starten. Der Computer denkt, dass der String vorbei sei. (Da ist ein Texteditor mit Syntax-Unterstützung nett.) Also, wie sagen wir dem Computer, dass wir im String bleiben wollen? Wir müssen dem Apostroph seine Bedeutung als Sonderzeichen nehmen, unter Programmierern sagt man auch: *wir müssen das Apostroph escapen*, nämlich so:

```
puts 'Wie geht\'s?'
```

```
Wie geht's?
```

Der Backslash ist das sogenannte *Escape*-Zeichen. In anderen Worten, wenn du einen Backslash und einen anderen Buchstaben hast, dann werden sie manchmal in ein anderes Zeichen umgewandelt. Die einzigen Zeichen, für die dies Sinn macht, sind das Apostroph und der Backslash selbst. (Mal drüber nachdenken: der Backslash escaped sich selbst.) Hier ein paar Beispiele:

```
puts 'Wie geht\'s?'

puts 'backslash am Ende vom String  \\'

puts 'auf\\ab'

puts 'auf\ab'
```

```
wie geht's?

backslash am Ende vom String  \

auf\ab

auf\ab
```

Weil der Backslash ein 'a' nicht escaped, aber sich selbst escaped, sind die beiden unteren Zeilen identisch. Sie schauen im Code nicht gleich aus, aber für den Computer haben sie dieselbe Bedeutung. Wenn du weitere Fragen hast, lies einfach weiter! Ich konnte auf dieser Seite sicherlich nicht alle Fragen beantworten.

Variablen und Zuweisung

Wenn wir bisher einen String oder eine Zahl mit puts ausgegeben haben, war die Information im Anschluss weg. Dies bedeutet, dass wir Text, den wir doppelt ausgeben wollen, auch doppelt eintippen müssen:

```
puts '...das kannst du zweimal sagen...'

puts '...das kannst du zweimal sagen...'
```

```
...das kannst du zweimal sagen...

...das kannst du zweimal sagen...
```

Es wäre nett, wenn wir es nur einmal tippen müssen und dann wieder darauf zurückgreifen können... es irgendwo speichern können. Nun, das können wir natürlich - ansonsten hätte ich das Thema gar nicht angesprochen!

Um den String im Speicher deines Computers zu speichern, musst du dem String einen Namen geben. Programmierer bezeichnen dies als die Zuweisung und der Name heißt Variable. Solch eine Variable kann quasi jede Reihe von Buchstaben und Ziffern sein, jedoch muss das erste Zeichen ein Kleinbuchstabe sein. Versuchen wir nun dies mit dem letzten Programmbeispiel, ich nenne den String mit dem Namen meinString (obwohl ich genauso auch meinEigenerKleinerString oder heinrichDerAchte hätte nehmen können).

```
meinString = '...das kannst du zweimal sagen...'

puts myString

puts myString
```

```
...das kannst du zweimal sagen...

...das kannst du zweimal sagen...
```

Wenn man etwas nun mit meinString macht, so tut man es stattdessen mit '...das kannst du zweimal sagen...'. Man kann sich vorstellen, dass die Variable meinString

auf '...das kannst du zweimal sagen...' zeigt. Hier ist ein etwas interessanteres Beispiel:

```
name = 'Patricia Rosanna Mildred Oppenheimer'

puts 'Mein Name ist '+ name +'.'

puts 'Wow! '+ name +' ist wirklich ein langer Name!'
```

```
Mein Name ist Patricia Rosanna Mildred Oppenheimer.

Wow! Patricia Rosanna Mildred Oppenheimer ist wirklich
ein langer Name!
```

Ebenso wie wir einer Variablen ein Objekt zuweisen können, können wir auch ein neues zuweisen. (Deshalb heißen sie Variablen: denn das, worauf sie zeigen, kann sich ändern.)

```
komponist = 'Mozart'

puts komponist+ ' war unbekannt zu seiner Zeit.'

komponist = 'Beethoven'

puts 'Ich persönlich bevorzuge '+ komponist
```

```
Mozart war unbekannt zu seiner Zeit.

Ich persönlich bevorzuge Beethoven.
```

Natürlich kann man Variablen verschiedene Objekte zuweisen, nicht nur Strings:

```
var = 'nur ein anderer '+'String'

puts var

var =5* (1+2)

puts var
```

```
nur ein anderer String

15
```

Tatsächlich können Variablen quasi auf alles zeigen... außer auf andere Variablen. Was passiert also, wenn wir es versuchen?

```
var1 = 8

var2 = var1

puts var1

puts var2

puts ''

var1 = 'acht'

puts var1

puts var2
```

```
8

8

acht

8
```

Wenn wir die Variable var2 auf die Variable var1 zeigen lassen, so zeigt sie stattdessen auf die Zahl 8 (genauso wie var1 darauf zeigt). Dann lassen wir var1 auf 'acht' zeigen, aber weil var2 niemals wirklich auf var1 zeigte, zeigt var2 weiterhin auf 8.

Da wir nun Variablen, Zahlen und Strings kennen, mischen wir dies mal alles!

Mischen wir es

Wir haben uns verschiedene Objekte angesehen (Zahlen und Buchstaben) und wir haben Variablen auf sie zeigen lassen; als nächstes wollen wir sie nett miteinander kombinieren.

Wir haben gesehen, dass das folgende nicht funktioniert, wenn wir 25 drucken wollen, weil man keine Zahlen und Buchstaben addieren kann:

```
var1 = 2

var2 = '5'

puts var1 + var2
```

Teil des Problems ist, dass dein Computer nicht weiß, ob du versuchst, 7 zu bekommen (2+5), oder ob du 25 bekommen wolltest ('2'+'5').

Bevor wir all diese Variablen addieren können, müssen wir wissen, wie wir entweder aus var1 einen String machen können, oder aus var2 eine Integer-Variable.

Umwandlungen

Um ein Objekt in einen String zu verwandeln, müssen wir einfach .to_s dahinter schreiben:

```
var1 = 2

var2 = '5'

puts var1.to_s + var2
```

```
25
```

Entsprechend gibt uns to_i die Integer-Version eines Objektes, to_f ergibt die Float-Version. Schauen wir uns etwas genauer an, was diese drei Methoden machen (und was sie nicht machen):

```
var1 = 2
```

```
var2 = '5'

puts var1.to_s + var2

puts var1 + var2.to_i
```

```
25

7
```

Beachte, dass obwohl wir die String-Version von var1 ausgeben, var1 immer auf 2 zeigt, und nie auf '2'. Außer wenn wir var1 explizit ein neues Objekt zuweisen (wofür das = Zeichen verwendet wird), zeigt die Variable für die Laufzeit des Programms immer auf 2.

Nun probieren wir ein paar interessantere (und ein paar komische) Umwandlungen:

```
puts '15'.to_f

puts '99.999'.to_f

puts '99.999'.to_i

puts ''

puts '5 ist meine Lieblingszahl!'.to_i

puts 'Wer fragt nach 5?'.to_i

puts 'Es war deine Mutter.'.to_f

puts ''

puts 'stringy'.to_s

puts 3.to_i
```

```
15.0

99.999

99

```

```
5

0

0.0


stringy

3
```

Dies hat wahrscheinlich etwas überrascht. Das erste ist völlig klar: 15.0. Danach haben wir den String '99.999' in ein Float verwandelt und in einen Integer. Das Float hat getan, was wir erwarteten, das Integer verhielt sich wie immer, abgerundet.

Als nächstes hatten wir ein paar Beispiele eigenartiger Strings, die in Integer umgewandelt wurden. to_i ignoriert das erste Zeichen, das es nicht versteht und von da an den Rest des Strings. Entsprechend wurde der erste in 5 umgewandelt, doch die anderen wurden völlig ignoriert, weil sie mit Buchstaben begannen... also wählt der Computer die Null.

Schließlich haben wir noch zwei Umwandlungen, die nichts bewirkten. Das haben wir so erwartet.

Nochmal puts

Da gibt's noch etwas eigenartiges an meiner Lieblingsmethode... schau mal hier:

```
puts 20

puts 20.to_s

puts '20'
```

```
20

20

20
```

Warum drucken alle drei Zeilen dasselbe aus? Bei den beiden letzten ist es klar, da '20' dasselbe ist wie 20.to_s. Aber was ist mit der ersten, die Integerzahl 20? Was

bedeutet es denn, die Zahl 20 zu schreiben? Wenn du eine 2 und eine 0 auf ein Stück Papier schreibst, schreibst du einen String, keine Zahl. Der Integer 20 ist die Anzahl an Fingern und Zehen, es ist keine 2 gefolgt von einer 0.

Hier liegt das große Geheimnis hinter unserem Freund puts: bevor puts etwas ausgibt, verwendet es to_s, um eine String-Version des Objektes zu erhalten. Tatsächlich steht das s in puts auch für String: puts bedeutet put String (dt.: gib String).

Das mag jetzt noch nicht zu aufregend wirken, doch in Ruby gibt es viele viele verschiedene Objekte (du wirst sogar noch deine eigenen schreiben!) und es ist wirklich wichtig zu wissen, was passiert, wenn du puts auf irgendein komisches Objekt anwendest, wie vielleicht das Bild Deiner Großmutter oder eine Musikdatei oder so. Aber das kommt später...

In der Zwischenzeit schauen wir uns ein paar weitere Methoden an und schreiben lustige Programme...

Die Methoden gets und chomp

Da puts put String heißt, kannst Du sicher erraten, was gets bedeutet. Und ebenso, wie puts nur Strings ausspuckt, nimmt gets nur Strings an. Und woher bekommt es sie?

Von dir! ... zumindest von deiner Tastatur. Weil deine Tastatur Strings erzeugt, funktioniert das wunderbar. Genau genommen sitzt gets da und wartet darauf, dass du die Enter-Taste drückst. Versuchen wir es:

```
puts gets
```

```
hallo Echo!

hallo Echo!
```

Natürlich wird genau das, was du tippst (diese Schrift!), wiederholt werden. Starte dieses Programm ein paar Mal und probiere verschiedene Texte aus. Jetzt können wir interaktive Programme schreiben! In diesem hier gibst du deinen Namen ein und es wird dich begrüßen:

```
puts 'Hallo, wie heißt Du?'

name = gets

puts 'Du heißt '+name+'? Was für ein schöner Name!'
```

```
puts 'Schön, dich zu treffen, ' +name+ '. :-)'
```

```
Hallo, wie heißt Du?


Chris

Du heißt Chris

? Was für ein schöner Name!

Schön, dich zu treffen, Chris

. :-)
```

Hmmm... offensichtlich wurden bei meiner Eingabe der Buchstaben C, h, r, i, s und der Enter-Taste sowohl die Buchstaben, wie auch das Enter in die Variable übernommen.

Glücklicherweise gibt es hierfür die Methode chomp. Sie entfernt alle Enter-Zeichen am Ende des Strings. Schreiben wir das Programm ein wenig um und verwenden wir chomp:

```
puts 'Hallo, wie heißt Du?'

name = gets.chomp

puts 'Du heißt '+name+'? Was für ein schöner Name!'

puts 'Schön, dich zu treffen, ' +name+ '. :-)'
```

```
Hallo, wie heißt Du?


Chris

Du heißt Chris? Was für ein schöner Name!

Schön, dich zu treffen, Chris. :-)
```

Viel besser! Beachte, dass die Variable name auf gets.chomp zeigt. Deshalb müssen wir chomp nicht wiederholen, name ist bereits ge-chomp-t.

Selber ausprobieren

- Schreibe ein Programm, das eine Person nach dem ersten und zweiten Vornamen und dem Nachnamen fragt. Danach soll es diese Person grüßen, mit dem vollen Namen.

- Schreib ein Programm, das eine Person nach ihrer Lieblingszahl fragt, dann 1 dazu zählt und das Ergebnis als größere und bessere Lieblingszahl vorschlägt. (Aber taktvoll!)

Wenn du diese beiden Programme beendet hast, ebenso wie weitere, die du schreiben möchtest, lass uns mehr Methoden kennenlernen, und mehr über Methoden.

Mehr über Methoden

Bisher haben wir ein paar Methoden kennengelernt, puts, gets und so weiter (Spezial-Quiz: Liste alle Methoden auf, die du gesehen hast! Es sind genau 10; die Auflösung gibt es später.) und doch haben wir noch nicht darüber gesprochen, was Methoden denn eigentlich sind. Wir wissen was sie tun, aber wir wissen nicht, was sie sind.

Aber das ist genau das, was sie sind: Dinge, die etwas tun. Wenn Objekte (wie Strings, Integers oder Floats) in der Programmiersprache die Nomen sind, dann sind Methoden die Verben. Und wie im Deutschen, kann man kein Verb verwenden ohne ein Nomen zu haben, das das Verb ausführt. Zum Beispiel ist 'ticken' nicht etwas, das einfach passiert; eine Uhr muss es tun. Im Deutschen heißt das: "die Uhr tickt." In Ruby heißt es uhr.tickt (unter der Annahme, dass uhr ein Ruby-Objekt ist). Programmierer sagen, dass wie die tickt-Methode des Uhr-Objektes aufrufen.

Hast du das Quiz gelöst? Gut. Ich bin sicher, dass du dich an die Methoden puts, gets und chomp erinnert hast, weil wir sie behandelt haben. Du hast wahrscheinlich auf die Umwandlungsmethoden gefunden, to_s, to_iund to_f. Und, hast du die weiteren 4 erkannt? Nichts anderes als unsere alten Mathe-Freunde +, -, *, /!

Ebenso, wie ich sagte, dass jedes Verb ein Nomen braucht, so braucht jede Methode ein Objekt. Normalerweise findet man leicht heraus, welches Objekt eine Methode ausführt: sie ist nämlich genau vor dem Punkt angegeben, wie im uhr.tickt-Beispiel oder in 101.to_s. Manchmal ist es aber nicht so ersichtlich, wie etwa bei den arithmetischen Methoden. Wie sich herausstellt, ist 5+5 eine Abkürzung für 5.+5. Zum Beispiel:

```
puts 'hallo '.+ 'Welt'

puts (10.*9).+ 9
```

```
hallo Welt

99
```

Das sieht nicht schön aus und wir werden es nie wieder so schreiben; doch es ist wichtig zu verstehen, was wirklich passiert. (Mein Computer hat mir hierbei eine Warnung gegeben: warning: parenthesize argument(s) for future version (=Argumente klammern für zukünftige Versionen). Er hat das Programm problemlos ausgeführt, jedoch hat es mir mitgeteilt, dass es Probleme hat, herauszufinden, was ich meine und ich solle doch in Zukunft mehr Klammern verwenden.) Damit ist auch leichter zu

verstehen, warum wir 'Schwein'*5 machen können, nicht aber 5*'Schwein': 'Schein'*5 sagt, dass 'Schwein' die Multiplikation ausführen soll, doch 5*'Schwein' sagt, dass 5 multiplizieren soll. 'Schwein' weiß, wie es 5 Kopien von sich selbst herstellt und addiert; währen 5 Probleme hat, 'Schwein' Kopien von sich selbst herzustellen und zu addieren.

Und natürlich müssen wir puts und gets erklären. Wo sind ihre Objekte? Im Deutschen kann man manchmal das Nomen weglassen, zum Beispiel, wenn ein Bösewicht schreit "Stirb!", so ist das implizierte Nomen die Person, die er anschreit. Wenn ich in Ruby schreibe puts 'sein oder nicht sein', so ist dies eine Abkürzung für self.puts 'sein oder nicht sein'. Aber was ist self? Das ist eine besondere Variable, die auf das Objekt zeigt, in dem du gerade drin bist. Wir wissen noch nicht, wie man in einem Objekt drin ist, aber wir werden es noch herausfinden und bis dann befinden wir uns einfach mal in einem großen Objekt... in dem ganzen Programm! Und glücklicherweise hält das Programm ein paar Methoden für uns bereit, wie etwa puts und gets. Schau mal:

```
unglaublichLangerVariablenNameDerNurAuf3Zeigt =3

puts unglaublichLangerVariablenNameDerNurAuf3Zeigt

self.puts
unglaublichLangerVariablenNameDerNurAuf3Zeigt
```

```
3

3
```

Wenn du das jetzt noch nicht ganz verstanden hast, so ist das auch o.k. Wichtig ist die Erkenntnis, dass jede Methode von einem Objekt ausgeführt wird, auch wenn kein Punkt davor steht. Wenn das klar ist, kann es weitergehen!

Lustige String-Methoden

Lass uns neue lustige String-Methoden kennen lernen. Du musst sie dir nicht alle merken, du kannst ja auf dieser Seite nachschlagen, wenn du sie brauchst. Ich möchte dir nur ein wenig von dem zeigen, was Strings tun können. Tatsächlich kann ich mir selber nicht einmal die Hälfte aller verfügbaren String-Methoden merken, aber auch das ist in Ordnung, denn es gibt tolle Seiten im Internet, die alle Methoden listen und erklären. (Am Ende des Tutorials zeige ich dir, wo du sie findest.) Ehrlich gesagt, will ich mir auch nicht alle Methoden merken, das wäre ja wie alle Wörter im Wörterbuch zu kennen. Ich kann auch sehr gut Deutsch sprechen ohne jedes Wort zu kennen...

und das ist ja auch der Grund dafür, dass es Wörterbücher überhaupt gibt: dass man nicht alle Wörter kennen muss! Unsere erste String-Methode dreht den String um:

```
var1 ='stop'

var2 ='kannst du den Satz rückwärts lesen?'

puts var1.reverse

puts v2.reverse

puts var1

puts var2
```

```
pots

?nesel sträwkcür ztaS ned ud tsnnak

stop

kannst du den Satz rückwärts lesen?
```

Du siehst: reverse dreht nicht den Original-String um; es macht nur eine neue Rückwärts-Version von ihm. Darum ist var1 immer noch 'stop', auch nach dem Aufruf von reverse. Eine andere String-Methode ist length, die uns die Anzahl der Zeichen im String mitteilt, inklusive Leerzeichen:

```
puts 'wie heißt du?'

name = gets.chomp

puts 'In deinem Namen sind '+name.lengths+' Zeichen,
'+name+'!'
```

```
wie heißt du?

Christopher David Pine

#<TypeError: can't convert Fixnum into String>
```

Oha! Da lief etwas schief, und es scheint, dass es irgendwo nach der Zeile name = gets.chomp passiert sein muss... siehst du das Problem? Schau mal, ob du es selbst beheben kannst! Das Problem liegt bei length: es gibt uns eine Zahl, doch wir wollen einen String. Ganz leicht: to_s anfügen (und Daumen drücken):

```
puts 'wie heißt du?'

name = gets.chomp

puts 'In deinem Namen sind '+name.length.to_s+' Zeichen, '+name+'!'
```

```
wie heißt du?

Christopher David Pine

In deinem Namen sind 22 Zeichen, Christopher David Pine!
```

Huch, das ist mir neu... Hinweis: das ist die Anzahl an Zeichen in meinem Namen, nicht die Anzahl an Buchstaben (zähl' sie!). Ich schätze mal, du kannst selber ein Programm schreiben, das einzeln nach dem ersten und zweiten Vornamen, sowie dem Nachnamen fragt, die Längen zusammenzählt... hey, probier' das doch aus! Mach nur, ich warte hier auf dich.

Hast du's geschafft? Gut. Hübsches kleines Programm, nicht wahr? Nach ein paar weiteren Kapiteln wird du fasziniert sein, was du alles kannst!

Es gibt auch ein paar String-Methoden, die Klein- und Großbuchstaben im String ändern können. upcase wandelt jeden Kleinbuchstaben in einen großen um und lowcase wandelt große Buchstaben in kleine. swapcase wandelt kleine in große und gleichzeitig große in kleine Buchstaben um. Und capitalize funktioniert wie downcase, doch das erste Zeichen wird in einen Großbuchstaben verwandelt (wenn es ein Buchstabe ist).

```
letters = 'aAbBcCdDeE'

puts letters.upcase

puts letters.downcase
```

```
puts letters.swapcase

puts letters.capitalize

puts ' a'.capitalize

puts letters
```

```
AABBCCDDEE

aabbccddee

AaBbCcDdEe

Aabbccddee

 a

aAbBcCdDeE
```

Nette Standard-Aufgaben. Wie man aus der puts ' a'.capitalize Zeile ersehen kann, wird das erste Zeichen in einen Großbuchstaben verwandelt, nicht der erste Buchstabe. Ebenso ist zu bemerken, dass die Variable letters nicht verändert wird. Dieses Verständnis ist wichtig. Es gibt Methoden, die das zugehörige Objekt verändern können, aber die werden wir erst später kennenlernen.

Die letzten lustigen Methoden, die wir uns ansehen, übernehmen die Formatierung von Text. Die erste, center, fügt Leerzeichen an den Anfang und das Ende des Strings, um ihn zentriert darzustellen. Ebenso wie wir bisher puts mitteilen mussten, was denn gedruckt werden soll oder +, was addiert werden soll, müssen wir dieser Methode sagen, auf welche Breite der String zentriert werden soll. Wenn wir also ein Lied zentriert haben wollen, würde ich es so machen:

```
breite = 50

puts('Hast Du etwas Zeit für mich?'.center(breite))

puts('Dann singe ich ein Lied für
Dich'.center(breite))

puts('Von 99 Luftballons'.center(breite))

puts('Auf ihrem Weg zum Horizont'.center(breite))
```

```
puts('Denkst      Du      vielleicht      grad\'      an
mich'.center(breite))

puts('Dann      singe      ich      ein      Lied      für
Dich'.center(breite))

puts('Von 99 Luftballons'.center(breite))

puts('Und dass sowas von sowas kommt'.center(breite))
```

```
          Hast Du etwas Zeit für mich?

        Dann singe ich ein Lied für Dich

                Von 99 Luftballons

            Auf ihrem Weg zum Horizont

        Denkst Du vielleicht grad' an mich

        Dann singe ich ein Lied für Dich

                Von 99 Luftballons

          Und dass sowas von sowas kommt
```

Ich glaube, es wird deutlich, wie die center-Methode funktioniert: durch die Variable breite wird die Gesamtbreite bestimmt, die in jeder Zeile verwendet wird. Dass ich diesen Wert in einer Variablen abgespeichert habe, liegt an meiner Faulheit... und Faulheit ist nicht unbedingt eine schlechte Sache in der Programmierung. Angenommen, ich beschließe später, dass ich das Lied breiter formatiert haben möchte, so muss ich nur eine Variable ändern, anstatt in jeder Zeile des Liedes den Wert zu ändern. Bei einem langen Text, kann mir dies viel Zeit ersparen. In der Programmierung ist diese Art der Faulheit eine echte Tugend!

Es fällt bei dem Text auf, dass nicht jeder Zeile exakt zentriert ist und ein Textverarbeitungsprogramm dies wohl schöner gemacht hätte. Wenn du wirklich perfekte Zentrierung (und vielleicht eine hübschere Schriftart) möchtest, dann musst du eine Textverarbeitung verwenden. Ruby ist ein tolles Werkzeug, aber kein Werkzeug ist perfekt für jeden Job.

Zwei weitere Methoden zur Formatierung von Strings heißem ljust und rjust, was für left justify (=Linksausrichtung) und right justify (=Rechtsausrichtung) steht. Sie

verhalten sich ähnlich wie center, doch füllen sie den String rechts oder links mit Leerzeichen auf. Schauen wir uns die drei im Vergleich an:

```
breite = 40

str = '--> text <--'

puts str.ljust(breite)

puts str.rjust(breite)

puts str.center(breite)

puts str.ljust(breite/2) + str.rjust(breite/2)
```

```
--> text <--
                                --> text <--
                 --> text <--
--> text <--                    --> text <--
```

Selber ausprobieren

- Schreib ein Programm chef.rb. Es fragt erst unhöflich, was du willst. Es wiederholt deine Antwort in Großbuchstaben und feuert dich im Anschluss, z.B.

```
Was wollen Sie?

eine Gehaltserhöhung

Was soll das heißen: "EINE GEHALTSERHöHUNG"??? - Sie
sind gefeuert!!!
```

- Schreib ein Inhaltsverzeichnis (für ein Ruby-Tutorial ;-), das den Inhalt folgendermaßen formatiert:

```
  Inhaltsverzeichnis

Kapitel 1: Zahlen   Seite 1
```

Mathematik

(Dieser Abschnitt ist optional. Es wird von einem gewissen mathematischen Grundwissen ausgegangen. Wenn es dich nicht interessiert, kannst du gleich mit dem nächsten Kapitel weitermachen. Häufig ist aber ein kurzer Blick auf die Zufallszahlen hilfreich.) Es gibt nicht annähernd so viele mathematische Methoden wie String-Methoden (und dennoch kenne ich sie nicht alle auswendig). Hier betrachten wir erst ein paar arithmetische Methoden, einen Zufallszahlengenerator und das Mathematik-Objekt Math mit seinen trigonometrischen und transzendenten Methoden.

Mehr Arithmetik

Die anderen beiden arithmetischen Methoden sind `**` (Exponent) und `%` (Modulo). Wenn man in Ruby “fünf zum Quadrat” sagen will, schreibt man 5**2. Man kann für den Exponenten auch Floats verwenden, die Quadratwurzel von 5 ist dann also: 5**0.5. Der Modulo-Operator gibt uns den Rest einer Division, wenn man zum Beispiel 7 durch 3 teilt, lautet das Ergebnis 2 Rest 1. Schauen wir uns das im Programm an:

```
puts 5**2

puts 5**0.5

puts 7/3

puts 7%3

puts 365%7
```

```
25

2.23606797749979

2

1

1
```

In der letzten Zeile sieht man, dass ein Nicht-Schaltjahr eine gewisse Anzahl von Wochen hat, plus einen Tag. Wenn Dein Geburtstag in einem Jahr an einem Dienstag ist, fällt er im nächsten Jahr auf einen Mittwoch. Man kann den Operator auch mit Floats verwenden, du kannst ja mal damit rumspielen... Eine letzte Methode möchte ich noch ansprechen, bevor wir uns Zufallszahlen zuwenden: abs. Sie berechnet uns den Absolutbetrag einer Zahl:

```
puts ((5-2).abs)

puts ((2-5).abs)
```

```
3

3
```

Zufallszahlen

Ruby hat einen ganz netten Zufallszahlengenerator. Die Methode, um eine Zufallszahl zu bekommen, heißt rand. Ruft man rand ohne Parameter auf, erhält man eine Zahl größer oder gleich 0.0 und kleiner 1.0. Wenn man rand einen Integer als Argument gibt (etwa 5), so erhält man eine Integer-Zufallszahl größer gleich 0 und kleiner 5, also eine der fünf Zahlen 0, ... 4.

Schauen wir uns rand in Aktion an:

```
puts rand

puts rand

puts rand

puts rand(100)

puts rand(100)

puts rand(100)

puts rand(1)

puts rand(1)

puts rand(1)

puts
rand(99999999999999999999999999999999999999999999999
999)

puts   'Der   Wetterbericht   meldet'+rand(101).to_s+'%
Regen,'

puts 'aber dem kann man eh nicht trauen!'
```

```
0.866769322351658

0.155609260113273

0.208355946789083

61

46

92

0

0

0

893478923874674465676512321164914962197628367462738 6

Der Wetterbericht meldet 45% Regen,

aber dem kann man eh nicht trauen!
```

Beachte, dass man mit rand(101) eine Zahl zwischen 0 und 100 erhält, rand(1) immer 0. Unverständnis über die möglichen Rückgabewerte ist der größte Fehler, den man mit rand machen kann. Ich hatte mal einen CD-Player, der bei zufälliger Wiedergabe alle Lieder abspielte, außer dem letzten... (ich frage mich, was passiert wäre, wenn ich eine CD mit nur einem Lied eingelegt hätte?)

Manchmal möchte man dieselben Zufallszahlen in derselben Reihenfolge bei zwei verschiedenen Abläufen des Programms bekommen. (Zum Beispiel habe ich mal Zufallszahlen verwendet, um eine zufällige Welt in einem Computerspiel zu verwenden. Wenn ich eine Welt fand, die mir gefiel, wollte ich sie speichern, oder einem Freund schicken.) Dafür muss man einen so genannten seed setzen, was man mit srand macht.

```
srand 1776

puts rand(100)

puts rand(100)

puts rand(100)
```

```
srand 1776

puts rand(100)

puts rand(100)

puts rand(100)
```

```
24

35

36

24

35

36
```

Verwendet man srand, wiederholt es die Zufallszahlen. Willst du wieder ganz andere Zahlen erhalten (wie wenn du nie srand verwendet hast), verwende srand(0). Damit wird ein neuer komischer Seed verwendet, der sich unter anderem aus der Zeit im Computer in Millisekunden berechnet.

Das Objekt Math

Schließlich betrachten wir das Math Objekt und fangen sofort mit Beispielen an:

```
puts (Math::PI)

puts (Math::E)

puts (Math.cos(Math::PI/3))

puts (Math.tan(Math::PI/4))

puts (Math.log(Math::E**2))

puts ((1+Math.sqrt(5))/2)
```

```
3.14159265358979
```

```
2.71828182845905

0.5

1.0

2.0

1.61803398874989
```

Dir fällt sicher sofort die ::-Notation auf. Die Erklärung dieses sogenannten Scope-operators liegt außerhalb dieses Tutorials. Es reicht, wenn ich betone, dass Math::PI genau das ist, was du erwartest: die Zahl PI.

Wie du siehst, bietet Math alle Möglichkeiten eines anständigen Taschenrechners, und mit den Floats kann man ziemlich genau rechnen.

Und damit *fließen* wir zum nächsten Thema...

Ablaufsteuerung

Hier werden wir nun einige Dinge zusammenführen. Auch wenn dieser Abschnitt kürzer und leichter ist als das Kapitel Mehr über Methoden, eröffnet es uns eine ganz große Welt von Programmier-Möglichkeiten. Nach diesem Kapitel kannst du echte interaktive Programme schreiben. Bisher hatten wir Programme, die in Abhängigkeit von der Tastatur-Eingabe verschiedene Dinge sagen, doch nach diesem Kapitel können sie auch noch unterschiedliche Dinge tun. Jedoch müssen wir vorher lernen, Objekte zu vergleichen. Wir brauchen ...

Vergleichs-Operatoren

Wir beeilen uns in diesem Teil, damit wir schnell zum nächsten Abschnitt kommen, in den all die coolen Dinge geschehen. Also, um zu überprüfen, ob ein Objekt kleiner oder größer einem anderen ist, verwenden wir `>` und `<`, also so:

```
puts 1>2

puts 1<2
```

```
false

true
```

Kein Problem. Wenn wir herausfinden wollen, ob ein Objekt größer-oder-gleich oder kleiner-oder-gleich einem anderen Objekt ist, verwenden wir `>=` und `<=`, z.B.:

```
puts 5>=5

puts 5<=4
```

```
true

false
```

Und schließlich, ob etwas gleich oder ungleich ist, verwenden wir `==` und `!=`. Dabei ist es wichtig, den Unterschied zwischen `=` und `==` zu verstehen: `=` sagt, dass eine

Variable auf ein Objekt zeigen soll (Zuweisungsoperator), == fragt "Sind diese beiden Objekte gleich?"

```
puts 1==1

puts 2!=1
```

```
true

true
```

Natürlich kann man ebenso Strings vergleichen. Wenn Strings verglichen werden, wird ihre lexikographische Ordnung verglichen, also ihre Reihenfolge im Wörterbuch:

```
puts 'cat'<'dog'
```

```
true
```

Aber Achtung: normalerweise sortieren Computer die Kleinbuchstaben nach den Großbuchstaben ein. (Weil sie ihre Buchstaben so sortiert haben.) Dies bedeutet, dass 'Zoo' vor 'albern' kommt. Wenn du also wissen willst, welches Wort in einem echten Wörterbuch zuerst kommen würde, musst du sicherstellen, dass beide Wörter erst in Großbuchstaben verwandelt und danach erst verglichen wurden (oder beide in Kleinbuchstaben).

Eine letzte Anmerkung: die Vergleichsmethoden geben uns nicht die Strings 'true' und 'false' zurück, sondern besondere Objekte true und false. (Natürlich ergibt true.to_s wiederum den String 'true'.) true und false werden oft verwendet in...

Verzweigungen

Verzweigungen sind ein einfaches Konzept, aber sehr wirkungsvoll. Ich denke sogar, dass es so einfach ist, dass ich es nicht erst erkläre, sondern einfach mal zeige:

```
puts 'Hallo, wie heißt du?'

name=gets.chomp
```

```
puts 'Hallo, '+name+'.'

if name == 'Chris'

  puts 'Toller Name!'

end
```

```
Hallo, wie heißt du?

Chris

Hallo, Chris.

Toller Name!
```

Aber wenn wir einen anderen Namen eingeben...

```
Hallo, wie heißt du?

Hillary

Hallo, Hillary.
```

... und das nennt man Verzweigung. Wenn die Bedingung, die nach dem if steht, wahr ist, wird der folgende Code bis zum end ausgeführt, sonst nicht. Ganz einfach.

Ich habe den Code zwischen dem if und dem end bewusst eingezogen, weil ich denke, dass es so übersichtlicher wirkt. So ziemlich alle Programmierer machen dies so, unabhängig von der verwendeten Programmiersprache. Der Nutzen mag in diesem kleinen Beispiel noch nicht sichtbar sein, aber in komplexeren Anwendungen macht diese Einrückung einen deutlichen Unterschied.

Oftmals möchte man ein Programm schreiben, das etwas tut, wenn die Bedingung wahr ist, und etwas anderes, wenn sie falsch ist. Dafür gibt es das Schlüsselwort else:

```
puts 'Ich kann hellsehen! Sag mir deinen Namen:'

name = gets.chomp

if name == 'Chris'
```

```
  puts 'Du wirst reich sein und Erfolg haben.'
else
  puts 'Deine Zukunft ist... oh, es ist schon spät...'
  puts 'Ich muss jetzt wirklich gehen!'
end
```

```
Ich kann hellsehen! Sag mir deinen Namen:
Chris
Du wirst reich sein und Erfolg haben.
```

... und mit einem anderen Namen:

```
Ich kann hellsehen! Sag mir deinen Namen:
Hillary
Deine Zukunft ist... oh, es ist schon spät...
Ich muss jetzt wirklich gehen!
```

Verzweigung ist im Code wie eine Abzweigung: nehmen wir den Pfad für name == 'Chris' oder den anderen?

Und wie bei Bäumen können hier Zweige auch wieder neue Zweige haben:

```
puts 'Hallo und willkommen im Kurs.'
puts 'Ich bin Frau Mahlzahn und wie heißt du?'
name = gets.chomp

if name==name.capitalize
  puts 'Setz dich doch, '+name+'.'
```

```
else
  puts name+'? Du meinst sicher: '+name.capitalize
    puts 'Weißt du jetzt, wie man deinen Namen
schreibt?'
  reply = gets.chomp

  if reply.downcase=='ja'
    puts 'Ok, setz dich.'
  else
    puts 'RAUS HIER!'
  end
end
```

```
Hallo und willkommen im Kurs.
Ich bin Frau Mahlzahn und wie heißt du?
chris
chris? Du meinst sicher: Chris
Weißt du jetzt, wie man deinen Namen schreibt?
ja
Ok, setz dich.
```

Und wenn ich den Namen groß schreibe:

```
Hallo und willkommen im Kurs.
Ich bin Frau Mahlzahn und wie heißt du?
Chris
```

```
Setz dich doch, Chris.
```

Manchmal kann es verwirrend sein, zu überblicken, welche ifs, elses und ends zusammen gehören. Ich schreibe sie deshalb immer gleichzeitig. Als ich also das obige Programm geschrieben habe, sah es erst einmal so aus:

```
puts 'Hallo und willkommen im Kurs.'

puts 'Ich bin Frau Mahlzahn und wie heißt du?'

name = gets.chomp


if name==name.capitalize

else

end
```

Dann füge ich Kommentare ein, das sind Bereiche, die der Compiler ignorieren wird:

```
if name == name.capitalize

  #  sie ist freundlich.

else

  #  sie wird sauer.

end
```

Alles nach dem #-Zeichen wird als *Kommentar* betrachtet (außer man ist in einem String). Danach ersetze ich die Kommentare mit funktionierendem Code. Manche Programmierer lassen gerne die Kommentare. Ich bin persönlich der Ansicht, dass gut geschriebener Code für dich selbst sprechen soll. Früher habe ich mehr Kommentare geschrieben, doch seit ich fließend Ruby 'spreche', werden es immer weniger. Manchmal finde ich sie auch verwirrend. Dies ist natürlich eine persönliche Einschätzung, und du findest sicher deinen eigenen Stil! Mein nächster Schritt sieht also so aus:

```
puts 'Hallo und willkommen im Kurs.'

puts 'Ich bin Frau Mahlzahn und wie heißt du?'

name = gets.chomp


if name==name.capitalize

  puts 'Setz dich doch, '+name+'.'

else

  puts name+'? Du meinst sicher: '+name.capitalize

    puts 'Weißt du jetzt, wie man deinen Namen
schreibt?'

  reply = gets.chomp


  if reply.downcase=='ja'

  else

  end

end
```

Du siehst: ich habe wieder erst mal das if-else-end fertig geschrieben. Das hilft mir, die Kontrolle über den Code zu bewahren. Es macht auch die Arbeit leichter, weil man sich immer auf kleine Bereiche konzentrieren kann, wie etwa das programmieren des if-Zweiges. Ein anderer Vorteil dieser Methode liegt darin, dass der Computer den Code zu jedem Zeitpunkt versteht und ausführen könnte. Jede der unfertigen Versionen, die ich vorgestellt habe, würde laufen. Sie sind nicht fertig, aber lauffähig. So kann man testen, während man programmiert, was hilft, Eindrücke vom Programm zu gewinnen. Wenn es all meine Tests erfüllt, weiß ich, dass ich fertig bin! Diese Tipps helfen dir, Verzweigungen zu programmieren, aber ebenso bei anderen Kontroll-Strukturen, wie den ...

Schleifen

Oft soll der Computer dasselbe immer und immer wiederholen: das ist ja auch das, worin Computer angeblich so gut sind!

Wenn du dem Computer sagst, er soll immer wiederholen, musst du ihm auch sagen, wann er damit aufhören soll! Computer langweilen sich nie, wenn du ihm also nicht sagst aufzuhören, wird er nie aufhören. Um dies zu vermeiden, sagen wir dem Computer, er solle wiederholen, solange eine Bedingung erfüllt ist. Das funktioniert so ähnlich wie die if-Anweisung:

```
command=''
while command !='Ende'
  puts command
  command = gets.chomp
end
puts 'Komm mal wieder!'
```

```


Hallo?
Hallo?
Wer ist da?
Wer ist da?
Schön dich zu treffen!
Schön dich zu treffen!
Oh, wie niedlich!
Oh, wie niedlich!
Ende
Komm mal wieder!
```

Und das ist eine Schleife! (Vielleicht ist dir die Leerzeile am Anfang der Ausgabe aufgefallen: die stammt vom ersten puts vor dem ersten gets. Wie würdest du das Programm abwandeln, damit sie nicht da ist? Probier es aus! Funktioniert ansonsten Dein Programm genauso wie das ursprüngliche Programm?)

Mit Schleifen kann man viele interessante Dinge tun, wie du dir sicher vorstellen kannst. Sie können aber auch Probleme bereiten, wenn du einen Fehler gemacht hast. Was passiert, wenn dein Computer in eine Endlos-Schleife geraten ist? In diesem Fall drücke einfach die Tastenkombination Strg-C. Bevor wir mit Schleifen rumspielen, lernen wir noch ein paar Dinge, die den Job leichter machen:

Ein bisschen Logik

Schauen wir noch einmal auf unser erstes Verzweigungs-programm. Was passiert, wenn meine Frau heimkommt, das Programm ausprobiert und es sagt ihr nicht, was für einen schönen Namen sie hat? Ich möchte sie nicht kränken (oder gar auf dem Sofa schlafen), so sollten wir es umschreiben:

```
puts 'Hallo, wie heißt du?'

name=gets.chomp

puts 'Hallo, '+name+'.'

if (name=='Chris' or name=='Katy')

  puts 'Toller Name!'

end
```

```
Hallo, wie heißt du?

Katy

Hallo, Katy.

Toller Name!
```

Viel besser. Dafür haben wir den or-/oder-Operator verwendet. Die anderen logischen Operatoren lauten and/und und not/nicht. Es ist immer hilfreich, zur Verdeutlichung Klammern zu verwenden. Schauen wir uns das einmal an:

```
ichBinChris    =true
```

```
ichBinLila    =false
ichEsseGerne  =true
ichEsseSteine =false
```

```
puts (ichBinChris  and ichEsseGerne)
puts (ichEsseGerne and ichEsseSteine)
puts (ichBinLila   and ichEsseGerne)
puts (ichBinLila   and ichEsseSteine)
puts
puts (ichBinChris  or ichEsseGerne)
puts (ichEsseGerne or ichEsseSteine)
puts (ichBinLila   or ichEsseGerne)
puts (ichBinLila   or ichEsseSteine)

puts (not ichBinLila)
puts (not ichBinChris)
```

```
true
false
false
false

true
true
true
```

```
false

true

false
```

Die einzige Operation, die verwirren könnte, ist or (dt.: oder): in der natürlichen Sprache verwendet man 'oder' in der Bedeutung 'das eine oder das andere, aber nicht beide!'. Wenn zum Beispiel deine Mutter sagt: ï¿½Was willst du als Nachtisch: Eis oder Kuchen?' so heißt dies nicht, dass du beides haben kannst... in der Computersprache heißt dieser Ausdruck aber 'das eine oder das andere oder beide' oder anders ausgedrückt 'mindestens eines'. Und darum sind Computer viel cooler als Mütter...

Selber ausprobieren

- Taube Oma Schreib ein 'Taube-Oma'-Programm. Auf alle deine Eingaben reagiert sie mit 'WAS? ICH VERSTEHE NICHT! SPRICH LAUTER', außer du rufst es (=schreibst es in Großbuchstaben). Wenn du schreist, kann sie dich hören (denkt sie jedenfalls) und antwortet immer mit 'NICHT SEIT 1938'.
  Um das Programm wirklich realistisch zu gestalten, soll die Oma jedes Mal ein anderes Jahr nennen, zufällig zwischen 1930 und 1950.
  Das Gespräch kann nur mit einem 'AUF WIEDERSEHEN' beendet werden.
  Tipp 1: Vergiss die chomp-Methode nicht.
  Tipp 2: überlege Dir, welche Teile des Programms wiederholt werden sollen. All das sollte in der Schleife stehen.

- Liebe taube Oma:
  Die Oma möchte nicht, dass du gehst und stellt sich taub, wenn du 'AUF WIEDERSEHEN' sagst. Du musst es drei mal hintereinander sagen, damit das Programm endet. Wandel das Programm entsprechend ab.

- Schaltjahr
  Schreib ein Programm, das nach einem Start- und einem Endjahr fragt und alle Schaltjahre im angegebenen Zeitraum ausgibt. Es gilt:
  - ist ein Jahr durch 4 teilbar, ist es ein Schaltjahr
  - aber: ist es durch 100 teilbar, ist es doch kein Schaltjahr
  - aber: ist es durch 400 teilbar, so ist es wiederum ein Schaltjahr

(Ja, das ist verwirrend, aber nicht halb so schlimm, wie wenn wir den Juli mitten im Winter hätten, und das würde passieren, wenn das Schaltjahr nicht so berechnet werden würde.)

Wenn du das geschafft hast, gönn' dir eine Pause! Du hast schon viel gelernt. Glückwunsch! Erstaunt es dich, wie viel du einem Computer schon anschaffen kannst? Noch ein paar Kapitel und du kannst so gut wie alles programmieren - ehrlich!

Sieh dir nur mal an, was du mit Verzweigungen und Schleifen alles bewirken kannst!

Wir lernen nun ein Objekt, das Listen anderer Objekte kontrolliert, kennen: ein Array.

Arrays und Iteratoren

Schreiben wir ein Programm, das uns so viele Wörter eingeben lässt (ein Wort pro Zeile), wie wir wollen und erst abbricht, wenn wir in einer Leerzeile die Enter-Taste drücken. Danach wiederholt es alle Wörter alphabetisch sortiert. OK?

Also,...

Du weißt, dass ich nicht glaube, dass wir das machen können. Wir brauchen einen Weg, um eine unbekannte Anzahl an Wörtern zu speichern und sie zu organisieren, dass sie nicht mit unseren anderen Variablen vermischt werden. Wir brauchen eine Art Liste. Wir brauchen *Arrays*.

Ein Array ist nur eine Liste im Computer. Jedes einzelne Feld in der Liste kann wie eine Variable verstanden werden: du kannst sehen, auf welches Objekt jedes Feld zeigt und du kannst das Feld auf ein anderes Objekt zeigen lassen. Schauen wir uns ein paar Arrays an:

```
[]

[5]

['Hallo', 'Oma']

geschmack='vanille'                # das ist kein Array!!!

[89.9, geschmack, [true, false]]  # aber das hier...
```

Im ersten Beispiel haben wir ein leeres Array, dann ein Array mit einer einzelnen Zahl, schließlich ein Array, das auf zwei Strings zeigt. Danach haben wir eine einfache Zuweisung und schließlich ein Array, das auf drei Objekte zeigte, von denen das letzte wiederum ein Array ist. Denk dran, Variablen sind keine Objekte und unser Array zeigt auf einen Float-Wert, einen String und auf ein anderes Array. Selbst, wenn wir die Variable geschmack auf ein anderes Objekt zeigen lassen, ändert dies nichts am Inhalt des Arrays.

Damit wir einzelne Objekte in einem Array ansprechen können, ist jedem Feld ein Index zugewiesen. Programmierer (und auch viele Mathematiker) beginnen beim Zählen mit 0, und somit ist der erste Index im Array auch 0. Hier sieht man, wie die einzelnen Felder des Arrays angesprochen werden:

```
namen = ['Ada', 'Belle', 'Chris']

puts namen
```

```
puts namen[0]

puts namen[1]

puts namen[2]

puts namen[3]  # nicht im Array!
```

```
Ada

Belle

Chris

Ada

Belle

Chris

nil
```

Wir sehen also in der ersten Zeile, dass puts namen alle Felder des Arrays ausdruckt. Daraufhin verwenden wir namen[0], um das erste Feld anzusprechen und namen[1] für das zweite... ich denke mal, das ist verwirrend, aber man gewöhnt sich daran. Man muss sich nur daran gewöhnen, dass das Zählen mit 0 beginnt, und aufhören, die Wörter 'erstes' und 'zweites' zu verwenden. Ein paar Beispiele, wie du diese Zählweise in deinen Alltag übernehmen kannst: Wenn du bei einem 5-Gänge-Menü sitzt, beginne mit dem 0-ten Gang und arbeite dich zum 4-ten Gang vor. Deine Finger an der rechten Hand erhalten die Zahlen 0 bis 4. Meine Frau und ich jonglieren mit Keulen. Im Moment schaffen wir die Keulen 0 bis 5, doch hoffentlich beherrschen wir auch bald Keule 6. Das würde bedeuten, dass wir mit 7 Keulen jonglieren. Dass es dir ins Blut übergegangen ist, merkst du, wenn du über das nullte Ding sprichst... ja, das Wort gibt es wirklich: frag irgendeinen Programmierer oder Mathematiker!

Im Beispiel versuchen wir schließlich puts namen[3], um mal auszuprobieren, was passiert. Hast du einen Fehler erhalten? Manchmal stellt man ja eine Frage, die keinen Sinn macht (zumindest für den Computer), dann bekommt man einen Fehler. Andererseits fragt zwar eine korrekte Frage, doch die Antwort ist nichts. Was ist im Feld 3? Nichts. Was ist namen[3]? nil: Rubys Ausdruck für 'nichts'. nil ist ein besonderes Objekt, das ausdrücken soll 'kein anderes Objekt'.

Wenn dir nun die komische Nummerierung in den Arrays zu viel wird, kann ich dich beruhigen: wir verwenden oft andere Methoden, mit denen wir und darüber überhaupt keine Gedanken machen müssen, wie ...

Die Methode each

each ermöglicht es uns, etwas (was auch immer du willst) mit allen Objekten eines Arrays zu tun. Wenn wir also etwas nettes über alle Sprachen in einem Array sagen wollen, ginge das so:

```
sprachen=['Englisch', 'Deutsch', 'Ruby']

sprachen.each do |spr|

  puts 'Ich finde '+spr+' klasse!'

  puts 'Und Du?'

end

puts 'Und c++?'

puts 'ähem ...'
```

```
Ich finde Englisch klasse!

Und Du?

Ich finde Deutsch klasse!

Und Du?

Ich finde Ruby klasse!

Und Du?

Und c++?
```

ähem ... Was ist passiert? Wir sind durch alle Felder des Arrays durchgegangen ohne Indexnummern zu verwenden, was nett ist. übersetzt ins Deutsche bedeutet das obige Programm so etwas wie: für jedes Objekt im Array 'sprachen', lass die Variable 'spr' auf das Objekt zeigen und führe dann das aus, was ich dir sage, bis du auf das Wort end stößt.

(Kurze Info am Rande: c++ ist eine andere Programmiersprache. Sie ist schwieriger zu erlernen und ein Programm in c++ ist meist länger als dasselbe Programm in Ruby.)

Jetzt denkst du vielleicht: "dies ist irgendwie sehr ähnlich der Schleife, die wir im letzten Abschnitt kennengelernt haben" und ja, das ist ähnlich! Ein wichtiger Unterschied ist, dass each eine Methode ist während while und end keine Methoden sind, sondern Schlüsselwörter der Sprache Ruby, wie auch do, if und else oder das = oder die Klammern, vielleicht wir die Satzzeichen im Deutschen.

Aber nicht each. each ist nur eine Methode, die das Array bereitstellt. Methoden (wie each), die wie Schleifen wirken, werden auch als Iteratoren bezeichnet. Beachte, dass bei Iteratoren immer ein do...end folgt, bei while und if kam kein do vor; ein do kommt nur mit Iteratoren vor.

Hier ist noch ein kleiner Iterator, doch diesmal ist es keine Array-Methode, sondern eine Integer-Methode:

```
3.times do

  puts 'Hurra!'

end
```

```
Hurra!

Hurra!

Hurra!
```

Mehr Array-Methoden

Wir haben each kennengelernt, doch es stecken noch mehr Methoden in Arrays... fast so viele wie in den Strings! In der Tat sind einige von ihnen identisch zu den String-Methoden (wie length, reverse, + und *), außer, dass sie auf den Feldern eines Arrays statt auf den Buchstaben eines Strings arbeiten. Andere (wie last und join) sind spezifisch für Arrays. Wiederum anderen (wie push und pop) verändern sogar das Array. Und so wie bei Strings musst du sie dir nicht alle merken, es genügt, dass du weißt, dass es sie gibt und wo du nachsehen kannst (nämlich hier).

Zuerst schauen wir uns to_s und join an. join wirkt so ähnlich wie to_s, fügt aber einen String zwischen die Objekte ein:

```
essen=['Schokolade', 'Kuchen', 'Karamell']


puts essen

puts essen.to_s

puts essen.join(', ')

puts essen.join(' und ')

puts essen.join(' und ') +': lecker!'
```

```
Schokolade

Kuchen

Karamell

SchokoladeKuchenKaramell

Schokolade, Kuchen, Karamell

Schokolade und Kuchen und Karamell

Schokolade und Kuchen und Karamell: lecker!
```

Du siehst, dass puts Arrays anderes behandelt als andere Objekte: es ruft puts auf jedem Objekt einzeln auf. Und 200 mal puts auf ein leeres Array anwenden, gibt einfach gar nichts aus. (200 mal nichts tun heißt: nichts tun.) Wende mal puts auf ein Array an, das andere Arrays enthält: tut es, was du erwartest? Schauen wir nun push, pop und last an. push und pop bewirken das Gegenteil von einander, so wie + und -. push fügt ein Objekt an das Ende eines Arrays an, und pop entfernt das letzte Feld eines Arrays (und sagt dir, was es war). last ist insofern ähnlich zu pop, dass es sagt, was das letzte Element ist, entfernt es aber nicht. Nochmal: push und pop verändern das Array!

```
favoriten=[]

favoriten.push '99 Luftballons'

favoriten.push 'irgendwie irgendwo irgendwann'
```

```
puts favoriten[0]
puts favoriten.last
puts favoriten.length

puts favoriten.pop
puts favoriten
puts favoriten.length
```

```
99 Luftballons
irgendwie irgendwo irgendwann
2
irgendwie irgendwo irgendwann
99 Luftballons
1
```

Selber ausprobieren

- Schreib mal das Programm, das zu Beginn dieses Kapitels beschrieben wurde. Tipp: es gibt eine Array-Methode, die Arrays sortiert: sort. Nutze sie!

- Und schreib das Programm mal ohne die Methode sort. Ein Großteil der Programmierung beschäftigt sich mit dem Lösen von Problemen: also übe dich darin!

- Schreibe das Inhaltsverzeichnis-Programm neu (aus dem Kapitel über Mehr über Methoden). Beginnen das Programm nun mit einem Array, das die benötigte Information über das Inhaltsverzeichnis (Kapitelüberschriften, Seitenzahlen und

so weiter) enthält. Dann drucke die Information im Array schön formatiert als Inhaltsverzeichnis aus.

Bisher haben wir sehr viele Methoden kennengelernt. Es ist an der Zeit, dass wir eigene schreiben!

Eigene Methoden schreiben

Wir haben gesehen, dass Schleifen und Iteratoren uns die Möglichkeit bieten, dasselbe immer und immer wieder zu tun. Manchmal will zwar man dasselbe öfter machen, aber an unterschiedlichen Stellen im Programm. Stellen wir uns vor, wir würden einen Fragebogen für einen Psychologie-Studenten schreiben. Bei den Studenten, die ich kannte, und den Fragebögen, die sie mir gegeben haben, würde dies etwa so aussehen:

```
puts 'Hallo und danke, dass du dir die Zeit nimmst'
puts 'und mich bei diesem Experiment unterstützt.'
puts 'Diese Umfrage geht um deine Meinung über '
puts 'italienisches Essen. Denke einfach nur an '
puts 'italienisches Essen und versuche, ganz ehrlich'
puts 'zu antworten, entweder mit "ja" oder "nein".'
puts 'Meine Umfrage hat nicht mit Bettnässen zu tun.'
puts


# Wir stellen ein paar Fragen, ignorieren aber die
Antworten.
richtig = false
while (not richtig)
  puts 'Magst du Spaghetti Bolognese?'
  antwort = gets.chomp.downcase
  if (antwort=='ja' or antwort=='nein')
    richtig=true
  else
    puts 'Bitte antworte mit "ja" oder "nein"'
  end
```

```
end

richtig = false
while (not richtig)
  puts 'Magst du Maccaroni mit Käse?'
  antwort = gets.chomp.downcase
  if (antwort=='ja' or antwort=='nein')
    richtig=true
  else
    puts 'Bitte antworte mit "ja" oder "nein"'
  end
end

# Diesmal merken wir uns die Antwort!
richtig = false
while (not richtig)
  puts 'Nässt du dein Bett?'
  antwort = gets.chomp.downcase
  if (antwort=='ja' or antwort=='nein')
    richtig=true
    if (antwort=='ja')
      bettnaesser=true
    else
      bettnaesser=false
    end
```

```
  else
    puts 'Bitte antworte mit "ja" oder "nein"'
  end
end

richtig = false
while (not richtig)
  puts 'Magst du Penne arrabiata?'
  antwort = gets.chomp.downcase
  if (antwort=='ja' or antwort=='nein')
    richtig=true
  else
    puts 'Bitte antworte mit "ja" oder "nein"'
  end
end

richtig = false
while (not richtig)
  puts 'Magst du Calzone?'
  antwort = gets.chomp.downcase
  if (antwort=='ja' or antwort=='nein')
    richtig=true
  else
    puts 'Bitte antworte mit "ja" oder "nein"'
  end
```

```
end


# und hier kommen noch weitere Fragen...

puts

puts 'Auswertung'

puts 'Danke, dass du dir Zeit für die Umfrage genommen
hast.'

puts 'Tatsächlich hat die Untersuchung aber nichts mit
'

puts 'italienischem Essen zu tun. Es ging ums
Bettnässen.'

puts 'Alle anderen Fragen sollten nur die
Aufmerksamkeit '

puts 'ablenken, um ehrliche Antworten zu erhalten.'

puts 'Vielen Dank nochmals.'

puts

puts bettnaesser
```

```
Hallo und danke, dass du dir die Zeit nimmst

und mich bei diesem Experiment unterstützt.

Diese Umfrage geht um deine Meinung über

italienisches Essen. Denke einfach nur an

italienisches Essen und versuche, ganz ehrlich

zu antworten, entweder mit "ja" oder "nein".

Meine Umfrage hat nicht mit Bettnässen zu tun.


Magst du Spaghetti Bolognese?
```

```
ja
Magst du Maccaroni mit Käse?
nein
Nässt du dein Bett?
nein
Magst du Penne arrabiata?
ja
Magst du Calzone?
nein


Auflösung
Danke, dass du dir Zeit für die Umfrage genommen hast.
Tatsächlich hat die Untersuchung aber nichts mit
italienischem Essen zu tun. Es ging ums Bettnässen.
Alle anderen Fragen sollten nur die Aufmerksamkeit
ablenken, um ehrliche Antworten zu erhalten.
Vielen Dank nochmals.


false
```

Dies war ein ziemlich langes Programm, mit vielen Wiederholungen. (Alle diese Teile Code über das italienische Essen waren identisch und die Frage übers Bettnässen war nur ein bisschen anders.) Wiederholung ist eine schlechte Angewohnheit. Andererseits können wir dies nicht in eine große Schleife oder Iterator packen, da wir ja manchmal etwas zwischen den Fragen auch machen wollen. In solchen Situationen braucht man Methoden. Und die schreibt man so:

```
def bellen
```

```
  puts 'wau '
end
```

Wie?... unser Programm hat gar nicht gebellt. Und warum nicht? Weil wir ihm nicht gesagt haben, dass es bellen soll. Wir haben ihm gesagt, *wie* man bellt, jedoch nie, dass es bellen soll. Versuchen wir es noch einmal:

```
def bellen
  puts 'wau '
end


bellen
bellen
puts 'wuff wuff wuff'
bellen
bellen
```

```
wau
wau
wuff wuff wuff
wau
wau
```

Ah, viel besser. (Info am Rande: Wir haben es in der Mitte mit einem englischen Hund zu tun, der sagt nämlich "wuff".)

Wir haben also eine Methode *bellen* definiert. (Methodennamen beginnen mit einem kleinbuchstaben, genau wie Variablennamen. Es gibt ein paar Ausnahmen wie + und ==.) Aber müssen Methoden nicht immer zu Objekten gehören? Ja, ganeu, das stimmt.

Und in diesem fall hier, wie auch bei puts und gets gehört die Methode zu dem Objekt, das das ganze Programm darstellt. Im nächsten Kapitel werden wir kennenlernen, wie man Methoden mit anderen Objekten assoziiert. Aber erst kümmern wir uns um ...

Methoden-Parameter

Sicher hast du schon gemerkt, dass manche Methoden (wie gets, to_s, reverse...) auf Objekten einfach aufgerufen werden. A ndere Methoden (wie +, -, puts...) benötigen *Parameter*, damit das Objekt weiß, was es genau tun soll. Du würdest je auch nicht wissen, was zu tun ist, wenn du nur 5 + bekommst, oder? Damit wird der 5 gesagt, dass addiert werden soll, doch es wird nicht gesagt, *was* addiert wird.

Um unserer bellen-Methode einen Parameter zu übergeben (der angibt, wie oft gebellt werden soll), schreiben wir dies:

```
def bellen num
  puts 'wau '*num
end


bellen 5
bellen 3
puts 'wuff wuff wuff'
bellen 6
bellen 4
bellen  # hier sollte ein Fehler gemeldet werden,
        # da der Parameter fehlt
```

```
wau wau wau wau wau
wau wau wau
wuff wuff wuff
wau wau wau wau wau wau
```

```
wau wau wau wau

bellen2.rb:10:in `bellen': wrong number of arguments
(0 for 1) (ArgumentError)

from bellen2.rb:10
```

num ist in der Methode bellen eine Variable, die auf den Wert zeigt, der als Parameter übergeben wird. Ich sag's noch einmal, weil es recht verwirrend ist: num ist in der Methode bellen eine Variable, die auf den Wert zeigt, der als Parameter übergeben wird. Wenn ich also bellen 3 aufrufe, dann ist der Parameter 3 und die Variable num zeigt drauf.

Wie du siehst, ist der Parameter jetzt *erforderlich*. Womit soll schließlich 'wau' multipliziert werden, wenn du ihm diese Information nicht sagst? deine armer Computer weiß es einfach nicht.

Wenn man Objekte in Ruby vergleicht mit Nomen im Deutschen, so sind die Methoden wie die Verben und man kann die Parameter als Adverben ansehen (z.B. hat uns in bellen der Parameter ja auch gesagt *wie* es zu tun sei) oder manchmal als direkte Objekte im Satz (beispielsweise in puts, wo der Parameter angibt, *was* ge-puts-t werden soll.

Lokale Variablen

Im folgenden Programm sind zwei Variablen:

```
def verdoppeln zahl

  ergebnis=zahl*2

  puts zahl.to_s+' verdoppeln ergibt '+ergebnis.to_s

end


verdoppeln 44
```

```
44 verdoppeln ergibt 88
```

Die Variablen heißen zahl und ergebnis. Beide leben innerhalb der Methode verdoppeln. Diese Variablen (und alle anderen, die du bisher gesehen hast) heißen *lokale Variablen*. Dies bedeutet, dass sie innerhalb der Methode leben und diese nicht verlassen können. Wenn du das versuchst, wirst du einen Fehler erhalten:

```
def verdoppeln zahl

  ergebnis=zahl*2

  puts zahl.to_s+' verdoppeln ergibt '+ergebnis.to_s

end



verdoppeln 44

puts ergebnis.to_s
```

```
44 verdoppeln ergibt 88

doubleThis2.rb:7: undefined local variable or method
`ergebnis' for main:Object (NameError)
```

Undefinierte lokale Variable... in der Tat *haben* wir zwar die lokale Variable ergebnis definiert, aber nicht lokal dort, wo wir versuchen, sie zu verwenden; sie ist lokal in der Methode.

Das mag unpraktisch erscheinen, ist aber der Erfahrung nach sehr praktisch! Einerseits bedeutet es, dass man außerhalb einer Methode keinen Zugriff auf die lokalen Variablen der Methode hat, heißt es auch, dass die Methoden keinen Zugriff auf *deine* lokalen Variablen haben, und sie somit nicht zerstören können:

```
def kaputt var

  var=nil

  puts 'haha, ich habe deine Variable kaputt gemacht'

end



var='meine Variable ist vor dir sicher!!!'
```

```
kaputt var

puts var
```

```
haha, ich habe deine Variable kaputt gemacht

meine Variable ist vor dir sicher!!!
```

In diesem Programm gibt es nun zwei Variablen, die var heißen: eine innerhalb der Methode kaputt und eine außerhalb. Beim Aufruf kaputt var haben wir den String von einer Variable in die andere übergeben, dass sie beide auf denselben Inhalt zeigten. Danach hat die Methode kaputt seine *lokale* Variable var auf nil gesetzt, was aber nichts zu tun hatte mit unserer Variablen var außerhalb der Methode.

Rückgabewerte

Du hast sicherlich schon festgestellt, dass manche Methode etwas zurückgeben, wenn man sie aufruft. Zum Beispiel gibt gets einen String zurück (nämlich den, den du eingegeben hast) und die +-Methode in 5+3(was ja eigentlich 5.+(3) ist) gibt 8 zurück. Arithmetische Methoden für Zahlen geben Zahlen, arithmetische Methoden für Strings geben Strings zurück.

Es ist wichtig, sich den Unterschied zu verdeutlichen zwischen einer Methode, die einen Wert zurückgibt, wenn man sie aufruft und einer Methode, die Information auf dem Bildschirm ausgibt. Halt dir vor Augen, dass 5+3 zwar 8 zurückgibt, aber nicht am Bildschirm ausgibt!

Was gibt also puts zurück? Das hat uns bisher nicht interessiert, aber schauen wir das mal genauer an:

```
wert = puts 'Rückgabe von puts:'

puts wert
```

Rückgabe von puts:

```
nil
```

Also, erstens einmal gibt puts nil zurück. Obwohl wir es nicht getestet haben, gibt auch das zweite puts nil zurück; puts gibt immer nil zurück. Jede Methode gibt etwas zurück, auch wenn es nur nil ist.

Mach hier mal eine kleine Pause, in der du herausfindest, was die Methode bellen zurück gibt.

Überrascht? So funktioniert es: zurückgegeben wird der Wert der letzten Zeile der Methode. Bei der Methode bellen gibt es also den Wert von puts 'wau ' zurück, was nil ist, weil puts immer nil liefert. Wenn wir wollen, dass diese Methode den String 'Gelbes U-Boot' zurückgibt, so müssen wir *dies* als letzte Zeile schreiben.

```
def bellen num

  puts 'wau '*num

  'gelbes U-Boot'

end



x = bellen 5

puts x
```

```
wau wau wau wau wau

gelbes U-Boot
```

Versuchen wir das alles man mit unserer psychologischen Umfrage, aber diesmal mit einer Methode, die die Fragerei für uns übernimmt. Die Methode braucht also die Fragestellung als Parameter und sie soll true zurückgeben, wenn mit 'ja' geantwortet wurde, und false, wenn mit 'nein geantwortet wurde. (Auch wenn wir diese Antworten meist ignorieren werden ist es kein schlechter Ansatz, wenn die Methode die Antwort mal zurückgibt. Außerdem können wir die Methode auf diese Weise auch für die Bettnässer-Frage verwenden.) Ich erlaube mir ein paar Abkürzungen bei der Begrüßung und der Auswertung, aber du weißt ja schon, worum es geht, und so ist es leichter zu lesen:

```
def fragen fragestellung

  richtig = false
```

```
  while (not richtig)
    puts fragestellung
    antwort = gets.chomp.downcase
    if (antwort=='ja' or antwort=='nein')
      richtig=true
      if (antwort=='ja')
        antwort=true
      else
        antwort=false
      end
    else
      puts 'Bitte antworte mit "ja" oder "nein"'
    end
  end
  antwort # der Rückgabewert!!!
end


puts 'Hallo und danke, dass du ...'
puts


fragen 'Magst du Spaghetti Bolognese?'  # ignorieren!
fragen 'Magst du Maccaroni mit Käse?'


bettnaesser = fragen 'Nässt du dein Bett?'  # Antwort
merken!
```

```
fragen 'Magst du Penne arrabiata?'

fragen 'Magst du Calzone?'

fragen 'Magst du Pizza Hawaii?'

fragen 'Magst du Vitello Tonnato?'

fragen 'Magst du Tira-mi-su?'


puts

puts 'Auswertung'

puts 'Danke, ...'

puts

puts bettnaesser
```

```
Hallo und danke, dass du ...


Magst du Spaghetti Bolognese?

ja

Magst du Maccaroni mit Käse?

nein

Nässt du dein Bett?

nein

Magst du Penne arrabiata?

ja

Magst du Calzone?

nein

Magst du Pizza Hawaii?
```

```
ja

Magst du Vitello Tonnato?

ja

Magst du Tira-mi-su?

ja


Auswertung

Danke, ...


false
```

Gar nicht schlecht, hm? Wir haben ein zusätzliche Fragen hinzugefügt (und das ist jetzt *wirklich einfach*) und doch ist unser Programm kürzer! Das ist ein großer fortschritt - und der Traum jedes faulen Programmierers.

Noch ein großes Beispiel

Ich denke mal, ein weiteres Beispiel für Methoden ist an dieser Stelle hilfreich. Wir nennen die Methode *deutscheZahl*. Sie nimmt eine Zahl (z.B. 22) und gibt das deutsche Wort dafür zurück (in diesem Fall *zweiundzwanzig*. Erstmal wollen wir nur Zahlen zwischen 0 und 100 betrachten.

(*Hinweis:* hier werden zwei neue Tricks verwendet: mit dem Schlüsselwort return kann man eine Methode frühzeitig verlassen und elsif bietet eine weitere Möglichkeit der Verzweigung. Ich denke, im Kontext wird die Bedeutung klar.)

```
def deutscheZahl zahl

  #  Nur Zahlen 0-100.

  if zahl < 0

    return 'Bitte gib eine Zahl 0 oder größer ein.'

  end

  if zahl > 100
```

```
    return 'Bitte gib eine Zahl 100 oder kleiner ein.'
  end


  zahlString = ''  #  das wird unser Ergebnis werden


   # "links" ist der Anteil von 'zahl' den wir noch
verarbeiten müssen
   # "schreiben" ist der Anteil, den wir jetzt schon
rausschreiben können
  links  = zahl
   schreiben = links/100                # Wie viele
Hunderter müssen noch geschrieben werden
   links  = links - schreiben*100  # Diese Hunderter
abziehen.


  if schreiben > 0
    return 'einhundert'
  end


   schreiben = links%10   # wie viele Einer müssen wir
noch schreiben?


  if links/10 %10 !=1      # Sonderfall: Teenager!!!
    links  -=schreiben     # diese Einer abziehen!


    if schreiben > 0
      if    schreiben == 1
        if links>0
```

```
        zahlString = zahlString + 'ein'
      else
        zahlString = zahlString + 'eins'
      end
    elsif schreiben == 2
      zahlString = zahlString + 'zwei'
    elsif schreiben == 3
      zahlString = zahlString + 'drei'
    elsif schreiben == 4
      zahlString = zahlString + 'vier'
    elsif schreiben == 5
      zahlString = zahlString + 'fünf'
    elsif schreiben == 6
      zahlString = zahlString + 'sechs'
    elsif schreiben == 7
      zahlString = zahlString + 'sieben'
    elsif schreiben == 8
      zahlString = zahlString + 'acht'
    elsif schreiben == 9
      zahlString = zahlString + 'neun'
    end

    if links>0
      zahlString +='und';
    end
```

```
    end
  end


   schreiben = links/10            #   wie viele Zehner
müssen geschrieben werden?
   links   = links - schreiben*10   #   Diese Zehner
abziehen.


  if schreiben > 0
    if schreiben == 1  #  Uh-oh...
      #  Sonderfall für die 'Teenager' 11-19
      if    links == 0
        zahlString = zahlString + 'zehn'
      elsif links == 1
        zahlString = zahlString + 'elf'
      elsif links == 2
        zahlString = zahlString + 'zwölf'
      elsif links == 3
        zahlString = zahlString + 'dreizehn'
      elsif links == 4
        zahlString = zahlString + 'vierzehn'
      elsif links == 5
        zahlString = zahlString + 'fünfzehn'
      elsif links == 6
        zahlString = zahlString + 'sechzehn'
      elsif links == 7
```

```
        zahlString = zahlString + 'siebzehn'
      elsif links == 8
        zahlString = zahlString + 'achtzehn'
      elsif links == 9
        zahlString = zahlString + 'neunzehn'
      end
       # In diesem Fall haben wir auch schon die Einer
beachtet
      # und wir müssen nichts mehr schreiben
      links = 0
    elsif schreiben == 2
      zahlString = zahlString + 'zwanzig'
    elsif schreiben == 3
      zahlString = zahlString + 'dreißig'
    elsif schreiben == 4
      zahlString = zahlString + 'vierzig'
    elsif schreiben == 5
      zahlString = zahlString + 'fünfzig'
    elsif schreiben == 6
      zahlString = zahlString + 'sechzig'
    elsif schreiben == 7
      zahlString = zahlString + 'siebzig'
    elsif schreiben == 8
      zahlString = zahlString + 'achzig'
    elsif schreiben == 9
      zahlString = zahlString + 'neunzig'
```

```
    end
  end

  if zahlString == ''
    # zahlString kann nur leer sein,
    # wenn der Parameter zahl gleich 0 ist.
    return 'null'
  end

  #  Nun haben wir den zahlString erstellt
  # und müssen ihn nur noch zurück geben.
  zahlString
end

puts deutscheZahl(  0)
puts deutscheZahl(  1)
puts deutscheZahl(  9)
puts deutscheZahl( 10)
puts deutscheZahl( 11)
puts deutscheZahl( 17)
puts deutscheZahl( 32)
puts deutscheZahl( 51)
puts deutscheZahl( 88)
puts deutscheZahl( 99)
puts deutscheZahl(100)
```

```
puts deutscheZahl(555)
```

```
null
eins
neun
zehn
elf
siebzehn
zweiunddreißig
einundfünfzig
achtundachzig
neunundneunzig
einhundert
Bitte gib eine Zahl 100 oder kleiner ein.
```

Da gibt's natürlich ein paar Stellen, die ich in diesem Programm nicht mag. Erst einmal gibt's zu viel Wiederholung. Außerdem werden Zahlen größer als Hundert nicht behandelt. Und drittens, gibt es zu viele Spezialfälle, zu viele returns. Wir verwenden mal ein paar Arrays und räumen etwas auf:

```
def deutscheZahl zahl
  # negative Zahlen: 'minus' und die positive Zahl
  if zahl < 0
    return 'minus '+ (deutscheZahl (-1)*zahl)
    # Dies nennt man "Rekursion".
       # Die Methode ruft sich selber mit anderen
Parametern wieder auf.
```

```
      # So kann man verschiedene Fälle eleganter von
einander trennen.
  end
  if zahl == 0
    return 'null'
  end
  if zahl == 1
    return 'eins'
  end

  zahlString = ''  #  das wird unser Ergebnis werden

  # merken wir uns die deutschen Worte in Arrays,
  # dann brauchen wir keine lange Fallunterscheidung
  einer = ['ein', 'zwei', 'drei', 'vier', 'fünf',
    'sechs', 'sieben', 'acht', 'neun']

   zehner = ['zehn', 'zwanzig', 'dreißig', 'vierzig',
'fünfzig',
    'sechzig', 'siebzig', 'achzig', 'neunzig']

  # besonderie Namen zwischen 11 und 19
   teenager = ['elf', 'zwölf', 'dreizehn', 'vierzehn',
'fünfzehn',
                  'sechzehn',  'siebzehn',  'achtzehn',
'neunzehn']
```

```
# angenommen die Zahl ist 1000 oder kleiner...
if zahl <1000
  h = zahl/100   #hunderter
  z = zahl/10%10 #zehner
  e = zahl%10    #einer

  if h>0
    zahlString = zahlString + einer[h-1]+'hundert'
  end

  if z==1 && e>0 #teenager
    zahlString = zahlString + teenager[e-1]
  else
    if e>0
      zahlString = zahlString + einer[e-1]
      if z>1
        zahlString = zahlString +'und'
      end
    end
    if z>0
      zahlString = zahlString +zehner[z-1]
    end
  end
  return zahlString
end
```

```
  # nun haben wir nur Zahlen, die größer gleich 1000
sind...
  # wir betrachten immer Blöcke von drei Ziffern (also
eine Zahl kleiner 1000)
  # und fügen ihnen eine Stellen-Wertigkeit an
   wert = ['', 'tausend', 'millionen', 'milliarden',
'billionen']  # das muss reichen.

  w=0
  while zahl>0
    z = zahl%1000
    if z>0
          zahlString = (deutscheZahl z) + wert[w] +
zahlString
      # wieder Rekursion!
    end
    w = w+1
    zahl = zahl/1000
  end
  return zahlString
end

#Ausprobieren:
puts deutscheZahl(-17)
puts deutscheZahl( 51)
puts deutscheZahl(34555)
```

```
puts deutscheZahl(3455500000)

puts deutscheZahl(-439587000555)

puts deutscheZahl(-10000000034)
```

```
minus siebzehn

einundfünfzig

vierunddreißigtausendfünfhundertfünfundfünfzig

dreimilliardenvierhundertfünfundfünfzigmillionenfünfhu
nderttausend

minus
vierhundertneununddreißigmilliardenfünfhundertsiebenun
dachzigmillionenfünfhundertfünfundfünfzig

minus zehnmilliardenvierunddreißig
```

Ahhh... sehr viel schöner. Das Programm ist deutlich kompakter, weshalb ich so viele Kommentare eingebaut habe. Es funktioniert auch für größere und negative Zahlen... wenn es auch Schönheitsfehler gibt: ein paar Leerzeichen oder Bindestriche würden die Lesbarkeit erhöhen und probier doch mal eine Million aus! Aber das kannst du ja jetzt selber machen...

Selber ausprobieren

- Erweitere die Methode deutscheZahl so, dass sie auch Tausender verarbeiten kann und z.B. 'eintausend' zurückgibt, statt 'zehnhundert'.

- Nun sollst du die Methode deutscheZahl weiter verfeinern, dass sie auch Millionen, Milliarden und Billionen zurückgeben kann.

Herzlichen Glückwunsch! Jetzt bist du ein echter Programmierer! Du hast alles gelernt, was du brauchst, um große Programme von Anfang an zu schreiben. Wenn du selber Ideen hast für Programme, die du gerne schreiben würdest, solltest du jetzt damit anfangen!

Es ist natürlich ein langsamer Prozess, wenn man alles von Anfang an neu schreibt. Und warum sollte man Code schreiben, den andere schon geschrieben haben? Willst du, dass dein Programm E-Mails versenden kann? Möchtest du Dateien auf deinem Computer speichern und laden? Wie sieht es aus mit Webseiten für ein Tutorial, in dem der Code automatisch getestet wird? ;) Ruby bietet viele verschiedene Arten von Objekten, die uns unterstützen, bessere Anwendungen schneller zu schreiben. Schauen wir uns diese Klassen an...

Klassen

Bisher haben wir verschiedene Arten, bzw. *Klassen*, von Objekten kennengelernt: Strings, Integers, Floats, Arrays und ein paar Sonderfälle (true, false und nil), über die wir später reden. In Ruby beginnen diese Klassen immer mit einem Großbuchstaben: String, Integer, Float, Array... und so weiter. Wollen wir eine neues Objekt einer gewissen Klasse erzeugen, verwenden wir das Schlüsselwort new:

```
a = Array.new + [12345]  # Array addition

b = String.new + 'hello'      # String addition

c = Time.new

puts 'a= '+a.to_s

puts 'b= '+b.to_s

puts 'c= '+c.to_s
```

```
a= 12345

b= hello

c= 2011-06-05 18:21:21 +0100
```

Da man Array-Objekte mit [...] und String-Objekte mit '...' erzeugen kann, verwendet man in diesem Zusammenhang selten new. (Allerdings erzeugt in obigem Beispiel Array.new ein leeres Array und String.new einen leeren String.) Auch Integer bilden einen Sonderfall: doch kann man einen Integer nicht mit Integer.new erzeugen, sondern muss die entsprechende Zahl schreiben.

Die Klasse Time

Was steckt nun hinter dieser Klasse Time? Time-Objekte stellen Zeitpunkte dar. Man kann Zahlen hinzufügen oder abziehen, um ein neues Time-Objekt zu erhalten. Wenn man 1.5 zu einem Time-Objekt hinzuzählt, wird ein neues Time-Objekt erzeugt, das den Moment anderthalb Sekunden später darstellt:

```
time = Time.new      # Zeitpunkt, zu dem ich dieses
Programm geschrieben habe
```

```
time2 = time + 60  # eine Minute später


puts time

puts time2
```

```
Tue Jun 07 12:34:22 +0200 2011

Tue Jun 07 12:35:22 +0200 2011
```

Man kann auch ein Time-Objekt für einen gegebenen Zeitpunkt erstellen:

```
puts Time.mktime(2000, 1, 1)          #  1.Januar 2000

puts  Time.mktime(1976,  8,  3,  10,  11)    #    mein
Geburtstag
```

```
Sat Jan 01 00:00:00 +0100 2000

Tue Aug 03 10:11:00 +0100 1976
```

Bemerkung: Die Zeitangaben gelten bezüglich der am Computer eingestellten Zeitzone. Die Klammern dienen dazu, die Parameter zu gruppieren. Je mehr Parameter man angibt, desto genauer wird die Zeit angegeben.

Man kann Zeiten mit Vergleichsmethoden vergleichen (ein früherer Zeitpunkt ist kleiner als ein späterer) und wenn man eine Zeit von einer anderen abzieht, erhält man die Anzahl der Sekunden zwischen den Zeitpunkten. Spiel einfach mal damit herum!

Selber ausprobieren

- Eine Milliarde Sekunden... ermittle die genaue Sekundenzahl bei deiner Geburt (wenn du das weißt). Berechne, wann du eine Milliarde Sekunden alt wird (oder wurdest). Markier es dir in deinem Kalender :-)

- Alles Gute zum Geburtstag! Frage eine Person nach dem Jahr, dem Monat und dem Tag ihrer Geburt, berechne, wie alt sie ist, und gib einen Glückwunsch für jedes Jahr aus!

Die Hash-Klasse

Eine weitere nützliche Klasse ist die Klasse Hash. Hashes sind Arrays sehr ähnlich: sie haben viele Felder, die auf andere Objekte zeigen können. Im Array liegen diese Felder hinter einander und sind nummeriert (bei 0 anfangend). Im Hash sind diese Felder in keiner Reihenfolge (sie sind nur irgendwie da) und man kann jedes Objekt verwenden, um ein Feld anzusprechen. Ein typische Anwendungsfall liegt vor, wenn man eine Menge von Objekten verwalten will, aber keine Reihenfolge gegeben ist, zum Beispiel die Farben, die ich für verschiedene Teile in einem längeren Text verwenden könnte:

```
farbArray = []   #  neues Array, wie Array.new
farbHash  = {}   #  neues Has, wie Hash.new


farbArray[0]          = 'rot'
farbArray[1]          = 'grün'
farbArray[2]          = 'blau'
farbHash['string']  = 'rot'
farbHash['zahlen']  = 'grün'
farbHash['schlüssel'] = 'blau'


farbArray.each do |farbe|
  puts farbe
end
farbHash.each do |code, farbe|
  puts code + ':  ' + farbe
end
```

```
rot

grün

blau

schlüssel:  blau

zahlen:  grün

string:  rot
```

Wenn ich im Beispiel ein Array verwende, muss ich mir merken, dass ich im Feld 0 die Farbe für Strings gespeichert habe, im Feld 1 die Farbe für Zahlen und so weiter. Einfacher geht's mit Hashes: Das Feld 'string' enthält die Farbe für Strings, das Feld 'zahlen' enthält die Farbe für Zahlen und das Feld 'schlüssel' enthält die Farbe für Ruby-Schlüsselwörter. Man muss sich nichts merken. Vielleicht hast du im Beispiel festgestellt, dass die Objekte im Hash nicht in derselben Reihenfolge ausgegeben wurden, in der wir sie hineingeschrieben haben. Arrays behalten die Reihenfolge bei, Hashes nicht.

Obwohl es meist geschickt ist, Strings zu verwenden, um die Felder in einem Hash zu bezeichnen, kann man jedes Objekt von jeder Klasse verwenden, sogar Arrays oder andere Hashes (der Sinn ist dann vielleicht fraglich...):

```
komisch = Hash.new

komisch[12] = 'Affen'

komisch[[]] = 'totale Leere'

komisch[Time.new] = 'ich hab keine Zeit'
```

Hashes und Arrays haben jeweils ihre Vorteile. Du musst selber entscheiden, was am besten zu deiner Situation passt.

Klassen erweitern

Am Ende des letzten Kapitels haben wir ein Beispiel geschrieben, in dem wir das deutsche Wort für eine Zahl zusammengebaut haben. Es war keine Methode, die zur Klasse Integer gehört, sondern nur irgendeine Programm-Methode. Wäre es nicht nett, wenn man etwa 22.to_deu schreiben könnte, statt deutscheZahl 22? Das geht so:

```
class Integer
```

```
  def to_deu
    if self == 5
      deutsch = 'fünf'
    else
      deutsch = 'achtundfünfzig'
    end
    deutsch
  end
end


#  machen wir den Test:
puts 5.to_deu
puts 58.to_deu
```

```
fünf
achtundfünfzig
```

Ich hab's ausprobiert und es funktioniert ;-)

Wir haben eine Integer Methode definiert, indem wir in die Integer-Klasse hineingesprungen sind, die Methode definiert haben und wieder herausgesprungen sind. Jetzt haben alle Integer-Objekte diese Methode. Wenn du die Art und Weise, wie to_s funktioniert, nicht magst, kannst du die Methode ebenso neu schreiben... was ich hier aber nicht raten würde! Am besten lässt man alte Methoden in Ruhe und fügt neue hinzu, wenn man etwas neues machen will.

Verwirrt? Schauen wir uns das letzte Programm noch einmal in Ruhe an. Wenn wir bisher eine Methode geschrieben haben, war sie immer im Standard- Programm-Kontext abgelegt. In diesem Programm hingegen verlassen wir diesen Kontext und gehen in die Klasse Integer, in der wir die Methode definieren (was sie zu einer

Integer-Methode macht) und alle Integers können sie nutzen. Innerhalb der Methode nutzen wir das Wort self um auf das aktuelle Objekt zu verweisen.

Eigene Klassen schreiben

Wir haben schon eine Reihe verschiedener Klassen und Objekte gesehen. Dennoch ist es nicht schwer, sich eine Klasse zu überlegen, die von Ruby nicht angeboten wird. Glücklicherweise ist das Schreiben einer neuen Klasse genau so einfach wie das Erweitern einer bestehenden. Angenommen, wir würden gerne mit Würfeln in Ruby arbeiten. Eine Klasse Wuerfel könnten wir zum Beispiel so schreiben:

```
class Wuerfel

  def wuerfeln

    1 + rand(6)

  end

end


#  Erzeugen wir mal zwei Würfel...

wuerfel = [Wuerfel.new, Wuerfel.new]


#  ...und werfen wir sie:

wuerfel.each do |w|

  puts w.wuerfeln

end
```

```
5

2
```

(Wenn du den Abschnitt über Zufallszahlen ausgelassen hast: rand(6) liefert uns eine zufällige Zahl von 0 bis 5.)

Und da haben wir es auch schon: eine Klasse für unsere eigenen Objekte.

Wir können beliebige Methoden für unsere Objekte definieren... aber etwas fehlt noch: diese KLassen sind ein bisschen wie unsere Programme bevor wir Variablen kennengelernt haben. Schauen wir uns zum Beispiel die Würfel an: wir können sie werfen und jedes Mal erhalten wir eine neue Zahl, aber wir können später nicht mehr auf die geworfene Zahl zugreifen. Dazu bräuchten wir eine Variable, die auf diese Zahl zeigt. Verglichen mit der Realität sollte jeder Würfel eine Zahl haben und das Würfeln sollte diese Zahl verändern.

Wenn wir nun versuchen, die gewürfelte Zahl in einer (lokalen) Variablen der Methode wuerfeln zu speichern, wird der Inhalt der Variablen weg sein, sobald die Methode wuerfeln beendet ist. Diese Zahl müssen wir in einer anderen Art von Variablen speichern:

Instanz-Variablen

Wenn wir normalerweise über einen String sprechen, sagen wir einfach *String*. Wir könnten auch *String-Objekt* sagen. Manche Programmierer würden es *eine Instanz der Klasse String* nennen, was aber recht lang ist. Unter einer *Instanz* einer Klasse versteht man ein Objekt dieser Klasse.

Eine *Instanz-Variable* ist eine Variable eines Objektes. Die lokalen Variablen von Methoden existieren nur bis die Methode beendet ist. Instanzvariablen eines Objektes hingegen bestehen so lange, wie das Objekt existiert. Um Instanzvariablen von lokalen Variablen unterscheiden zu können, wird ihnen ein @ vorangestellt:

```
class Wuerfel
  def wuerfeln
    @zahl = 1 + rand(6)
  end
  def zeigen
    @zahl
  end
end


wuerfel = Wuerfel.new
```

```
wuerfel.wuerfeln
puts wuerfel.zeigen
puts wuerfel.zeigen
wuerfel.wuerfeln
puts wuerfel.zeigen
puts wuerfel.zeigen
```

```
5
5
1
1
```

Sehr schön! Nun würfelt die Methode wuerfeln den Würfel und die Methode zeigen zeigt uns, welche Zahl gewürfelt worden ist. Allerdings: was passiert, wenn wir zeigen aufrufen, bevor gewürfelt worden ist (bevor @zahl überhaupt gesetzt worden ist)?

```
class Wuerfel
  def wuerfeln
    @zahl = 1 + rand(6)
  end
  def zeigen
    @zahl
  end
end

puts Wuerfel.new.zeigen
```

... wenigstens keine Fehlermeldung! Aber dennoch macht es keinen Sinn, dass ein Würfel keine gewürfelte Zahl anzeigt. Es wäre schöner, wenn jedem Würfel-Objekt eine gewürfelte Zahl zugewiesen wird, wenn es erzeugt wird. Dafür gibt es die Methode initialize:

```
class Wuerfel

  def initialize

    wuerfeln

  end

  def wuerfeln

    @zahl = 1 + rand(6)

  end

  def zeigen

    @zahl

  end

end


puts Wuerfel.new.zeigen
```

```
3
```

Wenn ein Objekt erzeugt wird, wird diese Methode initialize aufgerufen, wenn es eine hat.

Unser Würfel ist quasi perfekt. Das einzige, was vielleicht fehlen könnte, wäre die Möglichkeit festzulegen, welche Zahl der Würfel gerade anzeigt... warum schreibst du nicht eine Methode, die genau das macht? Komm wieder, wenn du fertig bist und dein Programm erfolgreich getestet hast. Stelle auch sicher, dass niemand den Würfel auf 7 stellen kann!

Ziemlich coole Sachen, die wir hier gerade gemacht haben... echt tricky. Deshalb machen wir jetzt noch ein Beispiel. Angenommen, wir wollen ein virtuelles Tier bauen, etwa einen Babydrachen. Wie die meisten Babys kann er essen, schlafen und pinkeln. Dies bedeutet, dass wir ihn füttern müssen, schlafen legen und spazieren schieben. Intern merkt sich unser Drache, ob er gerade hungrig oder müde ist oder aufs Töpfchen muss. Aber das sehen wir von außen nicht, wie bei einem echten Baby kann man nicht fragen "Bist du hungrig?". Ein paar lustige Sachen kommen noch hinzu und natürlich bekommt unser Drachen einen Namen, wenn er geboren wird. (Was der new Methode übergeben wird, wird an initialize weitergegeben.) Ok, los geht's:

```
class Drache

  def initialize name

    @name=name

    @schlafen=false

    @magen=10     # =nicht hungrig, 0=hungrig!

      @blase=0          #  =muss  nicht  pinkeln,  10=muss
pinkeln!

    puts @name + ' wird geboren.'

  end


  def fuettern

    puts @name + ' wird gefuettert.'

    @magen=10

    zeitvergeht

  end


  def spazieren

    puts @name + ' wird spazieren gefahren.'

    @blase=0

    zeitvergeht
```

```
  end

  def schlafenlegen
    puts @name + ' wird schlafen gelegt.'
    @schlafen=true
    3.times do
      if @schlafen
        zeitvergeht
      end
      if @schlafen
            puts @name + ' schnarcht, es raucht und
qualmt.'
      end
    end
    if @schlafen
      @schlafen=false
      puts @name + ' wacht langsam auf.'
    end
  end

  def hochwerfen
    puts 'Du wirfst ' + @name + 'hoch in die Luft.'
         puts @name + 'lacht und versengt Deine
Augenbrauen.'
    zeitvergeht
  end
```

```
  def schaukeln
    puts 'Du schaukelst ' + @name + ' sanft.'
    @schlafen = true
    puts @name + ' schlaeft ein.'
    zeitvergeht
    if @schlafen
      @schlafen = false
          puts '... wacht aber wieder auf, wenn Du
aufhoerst.'
    end
  end


  private
    # private bedeutet, dass die Methoden, die nun
definiert werden,
    # nur innerhalb des Objektes sichtbar sind und
aufgerufen werden können.


  def hungrig?
    # Methodennamen dürfen mit ? enden.
     # Das macht man aber nur, wenn die Methode true
oder false zurück gibt.
    @magen<=2
  end


  def blasevoll?
```

```
    @blase>=8
  end


  def zeitvergeht
    if @magen >0
      # Nahrung bewegt sich vom Magen in die Blase
      @magen = @magen-1
      @blase = @blase+1
    else
      # Drache hungert!
      if @schlafen
        @schlafen=false
        puts @name + ' wacht ploetzlich auf!'
      end
        puts @name + ' hat einen Riesenhunger... und
frisst DICH!!!'
      exit    # beendet das Programm
    end


    if @blase>=10
      @blase=0
      puts 'oha - ' + @name + ' macht auf den Boden.'
    end


    if hungrig?
      if @schlafen
```

```
        @schlafen = false
        puts @name + ' wacht auf.'
      end
      puts @name + ' hat Magenknurren.'
    end

    if blasevoll?
      if @schlafen
        @schlafen = false
        puts @name + ' erwacht ploetzlich.'
      end
      @name + ' geht aufs Toepfchen.'
    end
  end
end

kuscheltier = Drache.new 'Norbert'
kuscheltier.fuettern
kuscheltier.hochwerfen
kuscheltier.spazieren
kuscheltier.schlafenlegen
kuscheltier.schaukeln
kuscheltier.schlafenlegen
kuscheltier.schlafenlegen
kuscheltier.schlafenlegen
```

```
kuscheltier.schlafenlegen

kuscheltier.schlafenlegen
```

```
Norbert wird geboren.

Norbert wird gefuettert.

Du wirfst Norberthoch in die Luft.

Norbertlacht und versengt Deine Augenbrauen.

Norbert wird spazieren gefahren.

Norbert wird schlafen gelegt.

Norbert schnarcht, es raucht und qualmt.

Norbert schnarcht, es raucht und qualmt.

Norbert schnarcht, es raucht und qualmt.

Norbert wacht langsam auf.

Du schaukelst Norbert sanft.

Norbert schlaeft ein.

... wacht aber wieder auf, wenn Du aufhoerst.

Norbert wird schlafen gelegt.

Norbert wacht auf.

Norbert hat Magenknurren.

Norbert wird schlafen gelegt.

Norbert wacht auf.

Norbert hat Magenknurren.

Norbert wird schlafen gelegt.

Norbert wacht auf.

Norbert hat Magenknurren.
```

```
Norbert wird schlafen gelegt.

Norbert wacht ploetzlich auf!

Norbert hat einen Riesenhunger... und frisst DICH!!!
```

Wow! Natürlich wäre es noch schöner, wenn es ein interaktives Programm wäre, aber das kannst du ja später selber machen. Ich wollte dir nur die wichtigen Teile zeigen, die man für eine Drachen-Klasse braucht.

Wir haben in diesem Beispiel ein paar neue Dinge gesehen: Das erste ist einfach: exit beendet das Programm. Als zweites haben wir das Wort private kennengelernt, das mitten in der Drachen-Klassen-Definition steckt. Man könnte es auch weglassen, doch ich wollte dir klar machen, was der Unterschied ist zwischen den Methoden, die von außen aufgerufen werden können (also Dingen, die mit dem Drachen gemacht werden können) und Methoden, die nur innerhalb des Drachens abspielen. Man kann sich diese auch als *unter der Motorhaube* vorstellen: wenn du nicht Automechaniker bist, musst du dich nur um Gaspedal, Bremse und Lenkrad kümmern. Ein Programmierer bezeichnet dies als die *öffentliche Schnittstelle* deines Autos. Doch wie genau der Airbag herausbekommt, wann er auslösen muss, liegt im Inneren des Autos, der typische Autofahrer weiß das nicht.

Um ein konkreteres Beispiel heranzuziehen, denken wir mal über ein Auto in einem Videospiel nach: Erst einmal sollte man entscheiden, wie die *öffentliche Schnittstelle* aussehen sollte, in anderen Worten, was sollen die Spieler mit dem Auto machen können. Also, sie sollen das Gas- und das Bremspedal treten können, und sie müssen irgendwie sagen können, wie kräftig sie das Pedal treten. (Denn es ist ein Unterschied, ob man es nur leicht berührt oder richtig durchtritt.) Außerdem müssen sie lenken können und auch da ist es wieder interessant, wie weit sie das Lenkrad einschlagen wollen. Und so weiter und so fort... was genau man dem Benutzer ermöglichen möchte, hängt von der Art des Spiels ab.

Aber innerhalb des Autos muss man noch mehr Daten und Methoden verwalten: die aktuelle Geschwindigkeit, die Richtung, die Position, und vieles mehr. Diese Attribute werden verändert durch die öffentlichen Methoden, aber der Benutzer kann sie nicht direkt setzen. Dies geschieht innerhalb des Autos.

Selber ausprobieren

- Schreibe eine Klasse *OrangenBaum*. Sie soll eine Methode hoehe haben, die die Höhe zurückgibt, und eine Methode einjahrvergeht, welche den Baum um ein Jahr altern läßt. Jedes Jahr wird der Baum höher und nach einer bestimmten Anzahl von Jahren stirbt der Baum. In den ersten Jahren trägt er keine Früchte, doch nach einer Weile schon und ältere Bäume trägen mehr Früchte als junge

Bäume... wie es für dich Sinn macht. Und natürlich musst du auch die Orangen zaehlen und eine Orange pfluecken können. Stell sicher, dass alle Orangen, die du nicht erntest, vor dem nächsten Jahr herunterfallen.

- Schreib ein interaktives Programm für den *Babydrachen*. Dabei sollst du Kommandos eintippen, wie fuettern oder spazieren, und die entsprechenden Methoden des Drachens werden aufgerufen. Weil du nur Strings eingibst, muss du dafür irgendwie die richtigen Methoden zuweisen.

Und das war's auch schon... halt, Moment, ich habe dir noch nichts erzählt über all die Klassen in Ruby, die Emails verschicken, Dateien speichern oder laden, Fenstern oder Buttons anzeigen oder gar 3D-Welten oder so...! Also, es gibt soooo viele Klasse, dass ich dir unmöglich alles zeigen kann. Die meisten kenne ich selber nicht! Ich *kann* dir aber noch ein paar Tips geben, wo du weitere Informationen finden kannst, damit du genau über die KLassen nachlesen kannst, die du zum Programmieren brauchst. Es gibt nur noch ein Thema, das ich dir zeigen möchte... ein Thema, das die meisten Programmiersprachen nicht haben, ohne das ich aber nicht mehr leben könnte: Blocks und Procs.

Blocks und Procs

Jetzt kommen wir zu einer der coolsten Möglichkeiten in Ruby. Wenige andere Sprachen kennen dieses Feature, die meisten bezeichnen es anders (wie etwa *closures*), aber die meisten verbreiteten kennen es nicht, und das ist eine Schande!

Also, was ist das für ein cooles neues Ding? Es handelt sich um die Möglichkeit, einen Block Programm-Code (also Code zwischen einem do und einem end), ihn in ein Objekt einzupacken und in einer Variable zu speichern oder ihn einer Methode zu übergeben, ihn an beliebigen Stellen auszuführen, wo auch immer du willst (auch mehrfach, wenn du willst). Damit ist der Block selber so eine Art Methode, außer dass er nicht an ein Object gebunden ist (sondern selbst ein Objekt *ist*) und man kann ihn behandeln wie jedes andere Objekt auch.

Es ist an der Zeit für ein Beispiel:

```
Trinkspruch = Proc.new do

  puts 'Prost!'

end



Trinkspruch.call

Trinkspruch.call

Trinkspruch.call
```

```
Prost!

Prost!

Prost!
```

Hier habe ich einen Proc (was wohl eine Abkürzung für Procedure/Prozedur sein soll, aber, weitaus wichtiger, sich auf Block reimt) der den Programm-Block enthielt, den ich daraufhin dreimal aufgerufen habe. Wie du siehst ist es eine Methode recht ähnlich.

Es gleicht sogar noch mehr einer Methode, denn Blocks können auch Parameter annehmen:

```
magstDu = Proc.new do |leckerei|

  puts 'ich *liebe* '+leckerei+'!!!'

end


magstDu.call 'Schokolade'

magstDu.call 'Prosecco'
```

```
ich *liebe* Schokolade!!!

ich *liebe* Prosecco!!!
```

Wir sehen also, was Blocks und Procs sind und wie man sie benutzt, aber was soll das alles? Warum verwenden wir nicht einfach Methoden? Nun, es gibt eben ein paar Dinge, die man mit Methoden nicht machen kann. Insbesondere kannst du einer Methode keine andere Methode übergeben (aber man kann Procs an Methoden übergeben) und Methoden können keine anderen Methoden zurückgeben (aber sie können Procs zurückgeben). Das ist ganz einfach und logisch, denn Procs sind Objekte, Methoden nicht.

Übrigens: schaut das irgendwie bekannt aus? Ja, du hast Blocks schon kennengelernt... als du über Iteratoren gelernt hast. Aber darüber werden wir ein bisschen später sprechen.)

Methoden können Procs übernehmen

Wenn wir einen Proc an eine Methode übergeben, können wir steuern, wie, ob oder wie of der Proc aufgerufen werden soll. Angenommen, wir wollen etwas spezielles tun, bevor und nachdem der eigentliche Programm-Code ausgeführt wird:

```
def wichtig eineProc

   puts 'Alles ruhig halten! Ich muss etwas wichtiges
machen...'

  eineProc.call

   puts 'OK, hat sich erledigt, ich bin fertig. Ihr
könnt weitermachen.'
```

```
end


begruessen = Proc.new do
  puts 'hallo'
end


verabschieden = Proc.new do
  puts 'servus'
end


wichtig begruessen
wichtig verabschieden
```

```
Alles ruhig halten! Ich muss etwas wichtiges machen...
hallo
OK, hat sich erledigt, ich bin fertig. Ihr könnt
weitermachen.
Alles ruhig halten! Ich muss etwas wichtiges machen...
servus
OK, hat sich erledigt, ich bin fertig. Ihr könnt
weitermachen.
```

Das mag jetzt nicht besonders großartig erscheinen, ist es aber tatsächlich ;-) In der Programmierung sind strikte Regel üblich, wann was zu tun sei. Wenn du zum Beispiel in eine Datei speichern willst, so musst du sie öffnen, die Information hineinschreiben und die Datei wieder schließen. Wenn du vergisst, sie zu schließen, können schlimme Dinge passieren. Aber jedes mal, wenn du etwas mit einer Datei machen will, muss du diese 3 Schritte abarbeiten: öffnen, das, was du *eigentlich* machen willst, und wieder

schließen. Das ist nervig und kann leicht vergessen werden. In Ruby wird das Speichern (oder Laden) einer Datei analog zu dem obigen Code abgearbeitet und du musst dir überhaupt keine weiteren Gedanken machen als über das, was du eigentlich speichern willst (oder laden). (Im folgenden Abschnitt werde ich dir zeigen, wo du Information über das Laden und Speichern von Dateien findest.)

Du kannst auch Methoden schreiben, die selbst entscheiden ob und wie oft eine Proc ausgeführt werden soll. Hier ist ein Beispiel für eine Methode, die in 50% der Fälle die Proc ausführt. Die andere Methode führt die Proc 2-mal aus.

```
def vielleicht eineProc
  if rand(2) ==  0
    eineProc.call
  end
end

def zweimal eineProc
  eineProc.call
  eineProc.call
end

rechts = Proc.new do
  puts 'rechts '
end

links = Proc.new do
  puts 'links '
end

vielleicht rechts
```

```
vielleicht links
zweimal rechts
zweimal links
```

```
rechts
links
rechts
rechts
links
links
```

Hier stelle ich ein paar verbreitete Verwendungsmöglichkeiten von Procs vor, die wir mit Methoden alleine einfach nicht hätten. An diesem Beispiel sieht man recht gut, wie sehr eine Methode von der Proc abhängen kann, die ihr übergeben wird. Unsere Methode übernimmt ein Objekt und ein eProc und ruft die Proc auf dem Objekt auf. Wenn die Proc false zurückgibt, beenden wir die Methode; ansonsten rufen wir die Proc mit dem zurückgegebenen Objekt noch einmal auf. Das machen wir so lange, bis die Proc false zurückgibt (was sie irgendwann tun sollte, da ansonsten das Programm abstürzt). Unsere Methode gibt den letzten nicht-false-Wert zurück, den es von der Proc erhalten hatte.

```
def bisFalsch einObjekt, eineProc
  eingabe =einObjekt
  ausgabe =einObjekt
  while ausgabe
    eingabe =ausgabe
    ausgabe =eineProc.call eingabe
  end
  eingabe
```

```
end


quadrateArray = Proc.new do |array|
  letzte = array.last
  if letzte <=0
    false
  else
    array.pop
    array.push letzte*letzte
    array.push letzte-1
  end
end


immerFalsch = Proc.new do |ignorieren|
  false
end


puts bisFalsch([13], quadrateArray).inspect
puts bisFalsch('ich sitze hier mitten in der Nacht und
programmiere: es macht Spaß!', immerFalsch)
```

```
[169, 144, 121, 100, 81, 64, 49, 36, 25, 16, 9, 4, 1,
0]
ich sitze hier mitten in der Nacht und programmiere:
es macht Spaß!
```

Ok, ich gebe zu, das war ein ziemlich komisches Beispiel. Aber es zeigt, wie unterschiedlich unsere Methode sich verhalten kann mit verschiedenen Procs.

Die verwendete inspect-Methode ist der Methode to_s recht ähnlich, außer dass der zurückgegebene String noch versucht den Ruby-Code darzustellen, den man zum Bauen des Objektes verwendet hat. Hier zeigt es uns das ganze Array, das beim ersten Aufruf von bisFalsch erzeugt worden ist. Vielleicht ist dir aufgefallen, dass wir nie das Quadrat von 0 gebildet haben, aber da das Quadrat von 0 wieder 0 ergibt, mussten wir auch nicht. Und weil immerFalsch immer false ergab, hat bisFalsch die Methode nur einmal aufgerufen und nur das Objekt zurückgegeben, was wir übergeben hatten.

Methoden, die Procs zurückgeben

Eine weitere coole Sache, die du mit Procs machen kannst, ist, sie in einer Methode herzustellen und anschließend zurückzugeben. Das ermöglicht uns verschiedene Varianten verrückter Programmierung (Viele Sachen mit beeindruckenden Namen wie *lazy evaluation*, *unendliche Datenstruktuen* und *Currying*), aber in der Praxis verwende ich dies fast nie, noch kenne ich jemanden, der das macht. Ich denke, es ist nicht das Übliche, was man als Ruby-Programmierer tagtäglich einsetzt, oder aber Ruby bietet viele Möglichkeiten für andere Lösungen. Ich bin mir da nicht sicher. Auf jeden Fall werde ich dies nur kurz ansprechen.

Die Methode zusammen in diesem Beispiel nimmt zwei Procs und gibt eine neue zurück, die erst die erste Proc aufruf und das Ergebnis an die zweite Proc weitergibt.

```
def zusammen proc1, proc2

  Proc.new do |x|

    proc2.call(proc1.call(x))

  end

end


weiter = Proc.new do |x|

  x+1

end


quadrat = Proc.new do |x|
```

```
  x*x
end


weiter_dann_quadrat = zusammen weiter, quadrat
quadrat_dann_weiter = zusammen quadrat, weiter


puts weiter_dann_quadrat.call(5)
puts quadrat_dann_weiter.call(5)
```

```
36
26
```

Man beachte, dass der Aufruf von proc1 in Klammern stehen muss, um zuerst ausgeführt zu werden.

Blocks (nicht Procs!) an Methoden übergeben

OK, das war nun von akademischem Interesse und auch etwas umständlich zu verwenden. Das größte Problem besteht darin, dass man drei Schritte abarbeiten muss, (Definieren der Methode, Erzeugen der Proc und Aufruf der Methode mit der Proc) obwohl man das Gefühl hat, es sollten nur zwei sein (Definieren der Methode und Übergeben des Blocks in die Methode, ohne überhaupt eine Proc zu verwenden), weil man in der Regel den Block (die Proc) sowieso nicht mehr verwendet, nachdem man ihn der Methode übergeben hat. Kaum zu glauben, die Macher von Ruby hatten diese Idee auch! Tatsächlich hast du das auch schon gemacht, jedesmal wenn wir Iteratoren verwendet haben.

Schauen wir uns ein Beispiel an, dann reden wir darüber.

```
class Array
  def eachEven(&einBlock)
    gerade=true
```

```
    self.each do |obj|
      if gerade
        einBlock.call obj
      end
      gerade = (not gerade)
    end
  end
end


['Apfel',          'Pferdeapfel',          'Pflaumen',
'Dattel'].eachEven do |obst|
  puts 'Lecker! Ich liebe '+obst+'-Kuchen! Du nicht?'
end


# Zur Erinnerung: mit eachEven bekommen wir die
geraden Elemente
# des Arrays, die hier zufällig die ungeraden Zahlen
sind...
[1,2,3,4,5].eachEven do |ungerade|
  puts ungerade.to_s + ' ist keine gerade Zahl!'
end
```

```
Lecker! Ich liebe Apfel-Kuchen! Du nicht?
Lecker! Ich liebe Pflaumen-Kuchen! Du nicht?
1 ist keine gerade Zahl!
```

```
3 ist keine gerade Zahl!

5 ist keine gerade Zahl!
```

Um also einen Block an eachEven zu übergeben, mussten wir den Block lediglich an die Methode anhängen. Auf diese Art kann man einen Block an jeder Methode übergeben, obwohl die meisten Methoden den Block einfach ignorieren werden. Um zu bewirken, dass deine eigene Methode den Block *nicht* ignoriert, nimm ihn, verwandle ihn in eine Proc und hänge den Namen der Proc an die Parameterliste der Methode an, mit einem vorangestellten `&`-Zeichen. Dies ist ein bisschen trickreich, aber nur ein bisschen, und du musst es auch nur einmal bei der Methodendeklaration machen. Danach kannst du die Methode immer wieder mit einem Block verwenden, genauso wie die vordefinierten Methoden, die Blocks übernehmen, wie each oder times. (Du erinnerst dich an 5.times do ..., oder?)

Wenn es dich verwirrt, dann erinnere dich daran, wie eachEven funktioniert: ruf den Block auf mit jedem zweiten Element aus dem Array. Einmal richtig geschrieben, musst du nicht mehr wissen, wie es funktioniert. Du musst auch nicht mehr darüber nachdenken, welcher Block wann aufgerufen wird. Und das ist genau *der* Grund, warum wir Methoden dieser Art schreiben: wir müssen nie wieder darüber nachdenken, wie sie arbeiten... wir verwenden sie einfach.

Ich weiß noch genau, wie ich einmal eine Möglichkeit brauchte, um zu messen, wie lange einzelne Bereiche eines Programms bei der Ausführung brauchen. (Dies wird auch als *Profiling* des Codes bezeichnet.) Also habe ich eine Methode geschrieben, die einen Block übernimmt, die Zeit vor Abarbeitung des Blocks speichert, den Block abarbeitet, die Zeit wieder nimmt und die Differenz berechnet. Ich finde den Code im Moment nicht, aber er sah in etwa so aus:

```
def profile beschreibung, &block

  start = Time.now

  block.call

  ende  = Time.now

  dauer = ende - start

  puts beschreibung+': '+dauer.to_s+' Sekunden'

end
```

```
profile '25000 Verdoppelungen' do

  nummer =1

  25000.times do

    nummer = nummer+nummer

  end

   puts nummer.to_s.length.to_s+ 'Ziffern' # Anzahl der
Ziffern in dieser großen Zahl

end



profile 'Zählen bis zu 1 Million' do

  nummer =0

  1000000.times do

    nummer = nummer+1

  end

end
```

```
7526Ziffern

25000 Verdoppelungen: 0.358207 Sekunden

Zählen bis zu 1 Million: 2.107091 Sekunden
```

Wie einfach und elegant! Mit dieser kleinen Methode kann ich nun ganz schnell jeden Programmabschnitt mal eben messen. Ich packe den Code nur in einen Block und sende ihn an die Methode profile. Geht's einfacher? In den meisten Programmiersprachen müsste ich an jeder Stelle haufenweise Code zur Zeitmessung einfügen (was wir hier alles in profile haben). In Ruby kann man es an einer Stelle schreiben und (besonders wichtig) aus dem anderen Code raushalten!

Selber ausprobieren

- *Großvaters Uhr*: Schreib eine Methode, die einen Block übernimmt und diesen für jede Stunde, die am heutigen Tage bereits verstrichen ist, einmal aufruft. Also, wenn man dem Programm einen Block übergibt, der "DingDong" ausgibt, so schlägt es die aktuelle Stunde, wie die Uhr bei meinem Großvater... oder zumindest so ähnlich. Teste dein Programm mit mehreren verschiedenen Blöcken (z.B. den DingDong-Block, den ich eben beschrieben habe).
  Tipp: mit Time.now.hour erhältst du die aktuelle Stunde, allerdings als eine Zahl zwischen 0 und 23. Du musst selber rechnen, dass du die normale Stundenzahl (1-12) hast.

- *Programm Logbuch*: Schreibe eine Methode log, die eine Beschreibung eines Blocks und -natürlich- einen Block übernimmt. Ähnlich wie im Beispiel wichtig soll sie einen String ausgeben, mit dem sie mitteilt, dass sie den Block nun startet, einen anderen String am Ende, sowie die Dauer und das Ergebnis des Blocks. Teste deine Methode mit einem Block, innerhalb dem du wiederum log aufruft und einen anderen Block übergibst. (Dies wird als *Verschachtelung* bezeichnet.) Lass dich von der Ausgabe überraschen.

- *Verbessertes Logbuch*: Die Ausgabe vom letzten Logger war relativ schwer zu lesen und es wurde immer schlimmer, je tiefer man den Aufruf des Loggers schachtelte. Es wäre soviel einfacher zu lesen, wenn es den inneren Block einrücken würde. Um dies zu erreichen, muss man mitzählen, wie tief man verschachtelt ist und entsprechend viele Leerzeichen voranstellen wann immer der Logger etwas schreiben will. Dafür verwendet man ein sogenannte *globale Variable*. Das ist eine Variable, die überall im Code sichtbar ist. Um eine globale Variable zu definieren stelle dem Variablennamen ein $ voran, wie etwa $global, $tiefe und $ueberall. Die Ausgabe soll dann in etwa so aussehen:

```
START "äußerer Block"
  START "kleiner Block"
    START "kleinerer Block"
    ENDE  "kleinerer Block" Dauer:0.001379 Ergebnis: 40320
  ENDE  "kleiner Block" Dauer: 0.001493 Ergebnis: gute Nacht
ENDE  "äußerer Block" Dauer: 0.001634 Ergebnis: false
```

Das war nun alles, was du in diesem Tutorial lernen wirst. Glückwunsch!!! Du hast *unheimlich viel* gelernt! Vielleicht erinnerst du dich nicht an alles oder du hast ein paar Abschnitte schnell überlesen... das ist völlig in Ordnung! Beim Programmieren geht es nicht darum, was du weißt, sondern darum, was du herausfinden kannst.

Solange du weißt, wo du nachlesen kannst, was du vergessen hast, bist du im grünen Bereich. I hoffe, du denkst nicht, dass ich all dies hier geschrieben habe, ohne mal hier und da nachzusehen. Natürlich habe ich nachgeschlagen. Und ich bekam viel Unterstützung bei dem Code, der dieses Tutorial (in der englischen Fassung!) enthält. Aber wo genau schau ich nach, wenn ich Hilfe brauche? Im letzten Kapitel bekommst du ein paar Tips...

Was noch?

Also, wie geht's weiter? Wen kannst du fragen, wenn du Antworten suchst? Was tust du, wenn du ein Programm schreiben willst, um eine Website zu öffnen, eine Email zu schreiben oder die Größe eines Bildes zu ändern? Nun, es gibt viele viele Orte, an denen Ruby hilft. Dummerweise ist das nicht sehr hilfreich, oder?

Ich selbst bevorzuge die folgenden drei Quellen: wenn die Frage klein ist und ich glaube, dass ich es selber herausfinden kann, verwende ich irb.

IRB: Interaktives Ruby

Wenn du Ruby installiert hast, so ich auch irb installiert. Um es zu nutzen, öffnest du die Kommandozeile und tippst irb. Wenn man im irb ist, kann man jeden Ruby-Ausdruck tippen und die Umgebung antwortet mit dem entsprechenden Wert. Tippe 1+2, und es sagt 3. (Bemerkung: man muss das puts nicht schreiben.) Es ist wie ein riesiger Ruby-Rechner. Wenn du fertig bist, tippst du exit.

irb bietet noch viel mehr und alles weitere kann man im so genannten Eispickel-Buch lesen:

Eispickel-Buch: 'Programmieren mit Ruby'

Dieses Buch ist die absolute Referenz für Ruby: 'Programmieren mit Ruby' von David Thomas und Andrew Hunt (die sich als die 'Pragmatischen Programmierer' einen Namen gemacht haben). Natürlich ist es empfehlenswert die aktuellste Version zu lesen, allerdings kann man eine nur etwas veralterte (aber größtenteils weiterhin richtige) Variante kostenlos im Internet finden. (Wenn du die Windows-Version von Ruby installiert hast, ist das Buch bereits dabei... auf Englisch.)

Hier findet man quasi alles über Ruby, von Informationen für Anfänger, wie auch Fortgeschrittene. Es ist leicht zu lesen; es ist kompakt; es ist perfekt. Ich wünschte, jeder Programmiersprache hat ein Buch dieser Qualität. Am Ende des Buches findest du einen langen Abschnitt, der alle Methoden in allen Klassen im Detail erklärt und Beispiele gibt. Ich liebe dieses Buch!

Man findet es häufig im Internet (z.B. auf der Seite der pragmatischen Programmierer), aber ich empfehle ruby-doc.org. Diese Version hat seitlich ein nettes Inhaltsverzeichnis sowie einen Index. (ruby-doc.org hat weitere großartige Dokumentationen, wie über die CORE API und die Standard Bücherei... es dokumentiert einfach alles, was mit Ruby zu tun hat. Schau's dir mal an!)

Und warum heißt es "Eispickel"? Wegen des Bildes eines Eispickels auf der Vorderseite. Dummer Name, aber so wird es nun einmal genannt.

Ruby-Talk: eine Ruby-Mailing-Liste

Selbst mit irb und dem Eispickel-Buch bleiben manchmal Fragen offen. Oder du möchtest wissen, ob jemand schon mal genau das geschrieben hat, was du gerade brauchst, und e dir bereit stellt. Für diese Fälle gibt's den Ruby-Talk, die Ruby-Mailing-Liste: lauter nette und hilfsbereite Menschen! Hier findest du Ruby-Talk.

Vorsicht: In der Mailing-Liste kommen täglich *viele* Mails. Ich lasse meine erst mal in ein anderes Mail-Verzeichnis schicken, damit sie mir nicht im Weg sind. Wenn du dich mit den Mails nicht belasten willst, musst du nicht! Die Ruba-Mailing-Liste ist in die Newsgroup comp.lang.ruby gespiegelt, und umgekehrt. Wie auch immer, du siehst dieselben Nachrichten in einem anderen Format.

Tim Toady

Es gibt ein Konzept in Ruby, vor dem ich dich bisher beschützt habe, das du aber sicher bald kennenlernen wirst: **TMTOWTDI** (Aussprache: Tim Toady): There is More Than One Wß To Do It. (Es gibt mehr als eine Art, es zu tun.)

Manche schwärmen von **TMTOWTDI**, während andere es nicht mögen. Mir persönlich ist es ziemlich egal, aber ich halte es für einen *schrecklichen* Ansatz, jemandem damit das Programmieren beizubringen. (Als ob das Erlernen eines möglichen Weges nicht schon reizvoll und zugleich verwirrend genug wäre!)

Nun, da du aus diesem Tutorial hinausgehst, wirst du vielen unterschiedlichen Code sehen. Ich kenne vielleicht 5 verschiedene Möglichkeiten, einen String zu bauen (neben der, dass der Text mit einfachen Anführungszeichen umschlossen wird) und jede funktioniert ein bisschen anders. Ich habe dir nur die leichteste der sechs Varianten gezeigt.

Und als wir über Verzweigungen gesprochen haben, habe ich dir if gezeigt, aber nicht unless. Das musst du in irb selber ausprobieren.

Eine weitere kleine Verkürzung, die man mit if, unless oder while verwenden kann, ist die Einzeiler-Version:

```
# Die Worte sollen nur irgendeinen Text darstellen...

puts 'Lorem ipsum dolor sit amet' if 5==2**2+1**1

puts 'consectetur adipisici elit' unless 'Chris'.length ==5
```

Lorem ipsum dolor sit amet

... und schließlich gibt es noch eine weitere Möglichkeit, eine Methode zu schreiben, die Blocks verarbeitet (nicht Procs). Wir haben bereits die Konstruktion gesehen, in der wir den Block genommen und mit dem &block-Trick in der Parameterliste der Funktionsdefinition in eine Proc verwandelt haben. Um den Block aufzurufen, verwendet man block.call. Nun, das geht auch kürzer (obwohl ich persönlich es verwirrender finde). Statt diesem hier:

```
def zweiMal(&block)
  block.call
  block.call
end
doItTwice do
  puts 'Was für ein schöner Tag heute!'
end
```

```
Was für ein schöner Tag heute!
Was für ein schöner Tag heute!
```

... probiere mal das hier:

```
def zweiMal
  yield
  yield
end
zweiMal do
  puts 'Was für ein schöner Tag heute!'
end
```

Was für ein schöner Tag heute!

Was für ein schöner Tag heute!

Ich weiß nicht recht... was ist deine Meinung? Vielleicht liegt es an mir, aber... yield (dt.: halt)? Wenn es irgendwas wie ruf_den_versteckten_block_auf wäre, das wäre für mich sehr *viel* klarer. Einige Leute sagen, dass yield (halt) für sie Sinn macht. Und genau darum geht es bei **TMTOWTDI**: sie machen es auf ihre Art, ich auf meine.

Das Ende

Mach das Beste draus... :-)

Und wenn Du dieses Tutorial hilfreich fandest (oder verwirrend) oder du einen Fehler findest, so lass es mich wissen!