

Programmieren

Einführung

mit

Ruby

Teil 1: Grundlagen



Verbesserungen, Ergänzungen, Vervielfältigung erwünscht!

9. September 2007

Franz Burgmann

f.burgmann@gmail.com

Inhaltsverzeichnis

Vorwort.....	4
Programmieren.....	4
Programmiersprachen.....	5
Installation von Ruby und erste Schritte.....	5
Installation in Linux.....	7
Installation in Microsoft Windows.....	7
Die Kommandozeile (Windows).....	7
Programm starten.....	8
Ein- und Ausgabe von Daten.....	8
Steuerzeichen entfernen.....	10
Rechnen.....	12
Datentypen.....	12
Datentypen umwandeln.....	14
Fehlermeldungen.....	15
Kommentare.....	16
Bedingungen und Verzweigungen.....	17
Wiederholungen (Schleifen).....	19
times-Schleife.....	19
while-Schleife.....	19
until-Schleife.....	20
while – und until-Schleifen zumindest einmal durchlaufen.....	20
for-Schleife.....	21
loop-Schleife.....	22
Schleifen manipulieren.....	23
Arrays.....	24
Methoden.....	26
Objekt-Orientierte-Programmierung.....	28
Instanz-Methoden (OOP).....	30
Konstruktor (OOP).....	32
Direkt les- und schreibbare Membervariablen (OOP).....	33
Vererbung (OOP).....	34
Überschreiben von Methoden (OOP).....	37
Das Schlüsselwort „super“ (OOP).....	38
Aufteilen eines Programmes auf mehrere Dateien.....	39
Iteratoren.....	42
Blöcke.....	43
Umgang mit Dateien.....	43
Lesen und Schreiben von Textdateien.....	43
Speichern von Objekten mit Hilfe von Textdateien.....	45
Speichern von Objekten mittels Serialisierung.....	48
Dokumentation.....	50
Interactive Ruby (irb).....	51
Geschafft!.....	52

Nützliche Links.....	52
Anhang: Einrichten des Editors „ConTEXT“.....	53
Ruby-Sprachdatei nachrüsten.....	53
Deutsche Sprache einstellen.....	53
Ruby-Programm starten.....	55
Stichwortverzeichnis.....	56

Vorwort

Ziel dieses Skriptums ist es, dir beim Einstieg in die Computerprogrammierung zu helfen. Aus diesem Grund lege ich keinen Wert auf Vollständigkeit, sondern verfolge das Ziel, einfach und verständlich zu sein.

Ruby eignet sich hervorragend für Einsteiger und ist darüber hinaus eine umfassende Sprache. Das heißt, wenn du Ruby gelernt hast, wirst du nicht nur die Prinzipien der Programmierung verstehen und somit sehr schnell auch andere Sprachen lernen können, sondern du kannst mit Ruby auch produktiv arbeiten.

Wie auf der ersten Seite angegeben, kannst du dieses Skriptum sehr gerne und ohne Einschränkung weitergeben.

Programmieren

Programmieren bedeutet im Wesentlichen, dem Computer zu „sagen“, was er machen soll.

Nun gibt es leider im Sprachgebrauch einen eklatanten Unterschied: wir sprechen eine Umgangssprache, z.B. deutsch, der Computer aber versteht ausschließlich Maschinencode, eine binäre Sprache.

Ein Beispiel:

deutsch: Hallo, könntest du mir bitte sagen, wie spät es ist?
Maschinencode (z.B.): 0010 1000 1100 0000 1111 ...

Keine Angst, du musst dir nicht derartige Zahlenfolgen notieren (oder gar auswendig lernen!), um dem Computer Anweisungen geben zu können. Es geht zum Glück sehr viel einfacher, und das verdanken wir den Programmiersprachen.

Programmiersprachen

Programmiersprachen sind, einfach ausgedrückt, eine Art Zwischensprache. Sie sind zwischen der dem Menschen ohne weiteres verständlichen Umgangssprache und der Computersprache, dem Binärcode, angesiedelt.

Es gibt Programmiersprachen, die näher am Verständnis des Computers angesiedelt sind und solche, die für den Menschen leichter verständlich und damit auch leichter erlernbar sind.

Die frühen Programmiersprachen sind vor allem zur ersten Kategorie zu zählen. Die Entwickler haben jedoch dazu gelernt und deshalb ist es heute für Programmierer, wenn sie auf moderne Programmiersprachen zurückgreifen, viel einfacher, dem Computer zu sagen, was er für sie machen soll. Und es ist damit auch sehr viel einfacher als früher, die edle Programmierkunst zu erlernen – dies kommt jetzt dir zugute!

Zu den modernen Sprachen zählt Ruby. Die Sprache kommt aus Japan, sie ist dort sehr populär. Inzwischen findet sie auch in der restlichen Welt immer mehr Beachtung.

Beim Design der Sprache stand für den Entwickler Yukihiro Matsumoto der Programmierer mit seinen Bedürfnissen im Vordergrund, deshalb ist die Sprache sehr intuitiv, es werden dem Programmierer keine unnötigen Steine in den Weg gelegt.

Installation von Ruby und erste Schritte

Zum Schreiben deiner Programme kannst du einen beliebigen Editor verwenden, du kannst sogar MS Word oder OpenOffice „missbrauchen“, wenn du die Datei im Textmodus speicherst.

Es empfiehlt sich jedoch, einen Editor zu verwenden, der Syntax-Highlighting beherrscht, der also Schlüsselwörter farbig hervorheben kann. Dies erleichtert es dem Programmierer (also dir!) ungemein, die Übersicht zu haben und zu bewahren.

Nimm also einen Editor deiner Wahl und gib folgende Zeile ein:

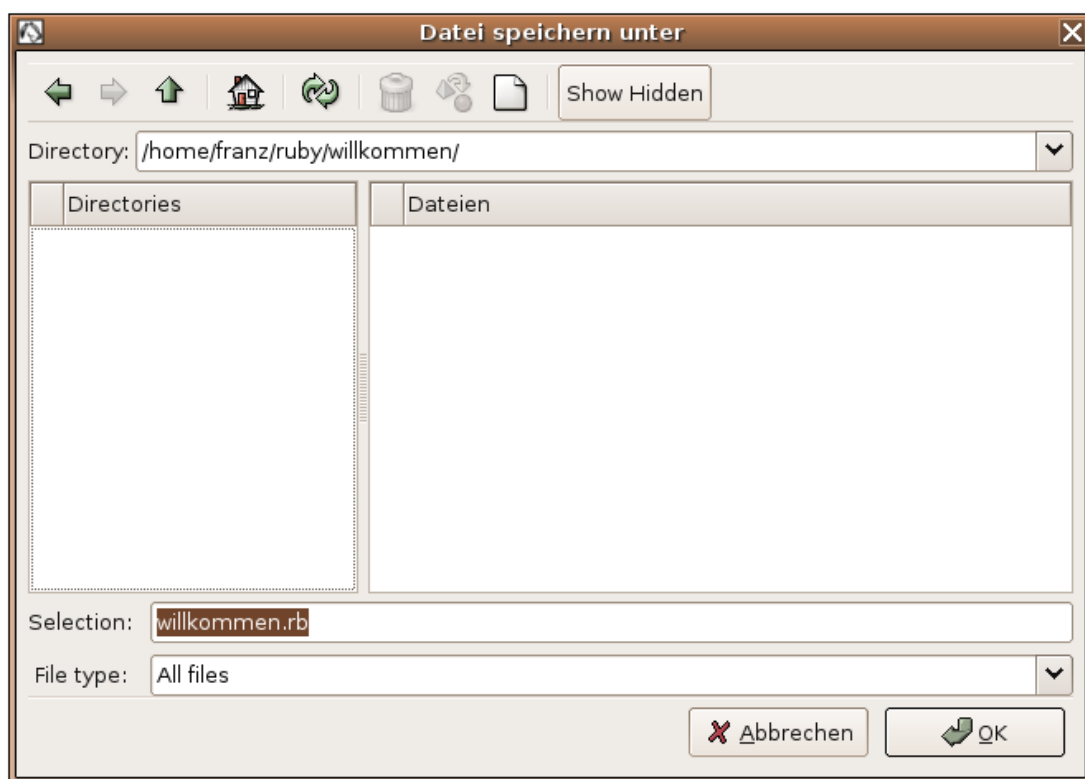
```
(1) puts "Hallo!"
```

Die Eins in Klammern am Anfang („1“) stellt lediglich die Zeilennummer dar (ist bei umfangreichen Programmen hilfreich). Schreibe sie also bitte in deinem Programm nicht.

Was diese Zeile genau zu bedeuten hat, sehen wir gleich anschließend.

Speichere nun die Datei. Für Ruby-Programme ist standardmäßig die Dateiendung „rb“ vorgesehen, weiters solltest du dir angewöhnen, Programmen einen aussagekräftigen Namen zu geben. Dieser könnte, wollen wir eben gesagtes berücksichtigen, etwa „willkommen.rb“ lauten.

Du solltest auch der Empfehlung folgen, jedes Programm in einem eigenen Ordner abzulegen:



Das Programm ist nun direkt lauffähig, es funktioniert ohne Änderung auf einer Vielzahl von Plattformen, etwa Linux, Windows und Mac OS.

Damit das Programm ausgeführt werden kann, braucht es allerdings noch den so genannten Interpreter, welcher den von dir geschriebenen Quellcode in die Computersprache (Binärcode) übersetzt und das Programm ausführt.

Installation in Linux

Die gängigen Distributionen bringen den Ruby-Interpreter bereits mit, eventuell muss er jedoch von den Installationsmedien oder übers Internet installiert werden. In Ubuntu geschieht dies mittels APT und beispielsweise dem Frontend Synaptic, das Paket nennt sich „ruby“.

Installation in Microsoft Windows

Für Windows ist ein praktisches Installations-Paket verfügbar, das zusätzlich zum Ruby-Interpreter ein Ruby-Handbuch, den schlichten aber sehr nützlichen Editoren SciTE und zusätzliche Erweiterungen beinhaltet. Du findest das Paket unter folgender URL: <http://rubyinstaller.rubyforge.org/>. Von mir getestet wurde die Version 1.8.6-25.

Das Einrichten des Editors „ConTEXT“, der auch das komfortable Starten von Ruby-Programmen aus dem Editor heraus beherrscht, wird im Anhang erklärt.

Die Kommandozeile (Windows)

Die folgenden Schritte sind für Windows-Benutzer, für Linux und Mac OS müssen sie entsprechend angepasst werden.

Begib dich nun nach erfolgter Installation zum Starten des Interpreters auf die Kommandozeile (Eingabeaufforderung).

In Windows XP findest du diese unter „Start-Programme-Zubehör“, alternativ kannst du in „Start-Ausführen“ „cmd“ eingeben. Wechsle ins Wurzelverzeichnis des Laufwerks, auf dem sich dein Programm befindet (im Beispiel ist es Laufwerk „c“):

```
cd c:\
```

Gib nun den Pfad zu deinem Programm ein, z.B.

```
cd Dokumente und Einstellungen\Franz\ruby\willkommen
```

oder schrittweise

```
cd Dokumente und Einstellungen
cd Franz
cd ruby
cd willkommen
```

Wenn du ein Unterverzeichnis verlassen willst, kannst du dies mit folgendem Befehl tun:

```
cd ..
```

Groß- und Kleinschreibung spielen in Windows im Gegensatz zu Linux keine Rolle.

Mit dem Befehl „dir“ kannst du dir den Inhalt eines Verzeichnisses anzeigen lassen. Auch hilft das Betriebssystem beim Tippen, indem Verzeichnis- und Programmnamen durch Drücken der Tabulator-Taste vervollständigt werden.

Programm starten

Führe dann das Programm aus:

```
ruby willkommen.rb
```

Und? Überrascht? Nein!? Na ja, das Ergebnis war vorauszusehen. Nichtsdestotrotz: es handelt sich um dein erstes Ruby-Programm! Und das ist doch ein Grund zur Freude, oder!?

Ein- und Ausgabe von Daten

```
name.rb:
(1) puts "Deinen Namen, bitte: "
(2) name = gets
(3) puts name + "ist ein schöner Name!"
```

Das Programm liefert (in meinem Fall) folgende Ausgabe:

```
Deinen Namen, bitte:
Franz
Franz
ist ein schöner Name!
```

Wie du bereits in unserem allerersten Programm (willkommen.rb) gesehen hast, bewirkt der Befehl „puts“, dass alles, was nach diesem Befehl in Anführungszeichen geschrieben wird, in der Konsole ausgegeben wird.

Die Zeichen zwischen den Anführungszeichen nennt man Zeichenkette oder, kürzer, String.

In Zeile 2 siehst du den Befehl „gets“. Dieser macht genau das entgegengesetzte von „puts“, er liest eine Eingabe von der Konsole in Form eines Strings ein.

Der Teil vor dem „gets“, das „name =“, hat zur Folge, dass die Eingabe, die mit „gets“ eingelesen wird, in der Variablen „name“ gespeichert wird. „name“ wird deshalb als Variable bezeichnet, weil sie ihren Inhalt ändern kann.

Du kannst dir eine Variable wie einen Schrank vorstellen, in den du etwas „hineinwerfen“ und es jederzeit wieder herausholen kannst.

Im nächsten Beispiel wird der Inhalt der Variable „name“ geändert:

```
name2.rb
(1)puts "Deinen Namen, bitte:"
(2)name = gets
(3)name = "Otto"
(4)puts name + "ist ein schöner Name!"
```

Ausgabe:

```
Deinen Namen, bitte:
Franz
Ottoist ein schöner Name!
```

In Zeile 2 liest das Programm den Namen des Benutzers von der Konsole ein, dieser wird wie gehabt in der Variablen „name“ gespeichert. In Zeile 3 jedoch wird direkt anschließend der Inhalt der Variable auf „Otto“ geändert.

Wenn du, bildlich gesprochen, in den Kasten „name“ etwas Neues hinein gibst (Zeile 3), fliegt das vorher dagewesene (Zeile 2) automatisch raus, weswegen die Ausgabe in Zeile 4 unabhängig von der Benutzereingabe (Zeile 2) ist.

Steuerzeichen entfernen

Noch zwei Dinge fallen auf: erstens fehlt bei der Ausgabe ein Leerschritt nach dem Namen - diesen Fehler kannst du beheben, indem du in Zeile 4 eine kleine Änderung durchführst:

```
puts name + " ist ein schöner Name!"
```

Zum Zweiten steht jetzt die Ausgabe in einer Zeile, vorhin waren es noch zwei.

Beim Einlesen des Namens wird nämlich nicht nur der Name eingelesen, sondern auch die am Ende gedrückte Return-Taste, sodass der Inhalt der Variablen `name` nicht „Franz“, sondern „Franz\n“ entspricht. (das Steuerzeichen „\n“ steht für neue Zeile).

Wenn also der Inhalt von `name` aus `name.rb` in Zeile 3 ausgegeben wird, erfolgt automatisch ein Zeilenumbruch, weil das entsprechende Steuerzeichen mit dem Namen eingelesen und in der Variable `name` gespeichert wurde.

Um in `name2.rb` denselben Effekt zu erzielen, müsstest du Zeile 3 durch folgende ersetzen:

```
name = "Otto\n"
```

Wir haben jedoch entgegengesetztes vor: wir wollen einen Namen einlesen und diesen mit einem weiteren String gemeinsam in derselben Zeile ausgeben.

Da hilft uns die Methode (was das genau ist, werden wir etwas später sehen) `chomp`. Diese entfernt alle Steuerzeichen (wie eben den Zeilenumbruch) am Ende eines Strings.

`name3.rb`

```
(1) puts "Deinen Namen, bitte:"  
(2) name = gets  
(3) name_bereinigt = name.chomp  
(4) puts name_bereinigt + " ist ein schöner Name!"
```

In Zeile 3 verwenden wir eine neue Variable (`name_bereinigt`) und weisen dieser den Wert von `name` zu, ohne „\n“, welches mit `chomp` abgeschnitten wird.

Du kannst die Methode `chomp` auch direkt auf den eingelesenen String anwenden:

`name4.rb`

```
(1) puts "Deinen Namen, bitte:"  
(2) name = gets.chomp  
(3) puts name + " ist ein schöner Name!"
```

Ausgabe:

```
Deinen Namen, bitte:  
Franz  
Franz ist ein schöner Name!
```

Wenn du auch die Eingabe in einer Zeile erledigen möchtest, kannst du die Methode `print` verwenden, die bis auf den fehlenden Zeilenvorschub äquivalent zu `puts` funktioniert.

`name5.rb`

```
(1) print "Deinen Namen, bitte: "  
(2) name = gets.chomp  
(3) puts name + " ist ein schöner Name!"
```

Ausgabe:

```
Deinen Namen, bitte: Franz  
Franz ist ein schöner Name!
```

Ganz nebenbei hast du bei diesen Beispielen noch etwas gelernt: zwei oder mehrere Strings werden einfach „addiert“, also durch „+“ miteinander verknüpft.

Arbeitsauftrag: Lies drei Namen ein und gib sie anschließend in der Konsole aus! Verknüpfe deine Ausgaben mit einer Aussage!

Rechnen

Wenn die Addition schon mit Strings funktioniert, dann erst recht mit Zahlen:

```
rechnen1.rb  
(1) puts 2 + 3
```

Was, denkst du, wird da wohl ausgegeben? Klar doch!

5

Ebenso gehen die anderen Grundrechnungsarten vonstatten, nur so mancher Quotient könnte dich überraschen:

```
rechnen2.rb  
(1) puts 5 / 3
```

Ausgabe:

1

Da hast du (hoffentlich!) was dagegen. Das Ergebnis sollte (auf zwei Nachkommastellen gerundet) 1.67 lauten.

Datentypen

Der Fehler bei der Division im letzten Beispiel hat mit den Datentypen zu tun.

Ebenso wie eine Eingabe von der Konsole mit „gets“ als String interpretiert wird, werden 5 und 3 hier als ganze Zahlen betrachtet, wir bezeichnen solche ganzen Zahlen als Integer.

Werden zwei Integer dividiert, ist auch der Quotient wieder eine ganze Zahl, die für ein exaktes Ergebnis notwendigen Nachkommastellen werden einfach ignoriert (abgeschnitten).

Du kannst Abhilfe schaffen, indem du einen der beiden Operanden als so genannte Fließkommazahl (Float) deklarierst.

Was charakterisiert nun eine Fließkommazahl: nun, die Nachkommastellen!

```
rechnen3.rb  
(1) puts 5 / 3.0
```

Ausgabe:

```
1.666666666666667
```

Du kannst Zeile 1 auch so schreiben:

```
puts 5.0 / 3
```

oder so:

```
puts 5.0 / 3.0
```

Die Umwandlung kann auch mittels einer Methode erfolgen:

```
puts 5 / 3.to_f
```

Wichtig ist schlicht, dass zumindest einer der beiden, Dividend oder Divisor, vom Typ Float ist, dann ist automatisch das Ergebnis vom Typ Float.

Ein weiterer wichtiger Datentyp, auf den du sicher später noch häufig zurückgreifen wirst, ist Boolean. Boolean ist ein „Wahrheitswert“, eine Variable dieses Typs kann nur zwei Werte annehmen: true oder false. Die Verwendung von Boolean siehst du ein klein wenig später.

Zusammenfassend noch mal die bisher behandelten Datentypen:

Datentyp	Beispiele
String	„Susi“, „sa4_xz“
Integer	1032, -12000000
Float	45.02, 0.0001
Boolean	true, false

Du kannst die Ziffern einer Zahl der Übersichtlichkeit wegen auch (beliebig) gruppieren: 1_000_000_056 (dasselbe wie 1000000056).

Datentypen umwandeln

Angenommen, du schreibst ein Programm, um zwei Zahlen zu addieren, weiters soll der Benutzer des Programms die Zahlen selbst vorgeben können, dann könnte dies so aussehen:

```
rechnen4.rb
(1) puts "Hallo! Ich addiere zwei Zahlen, die du vorgibst!"
(2) print "Zahl 1: "
(3) zahl1 = gets.chomp
(4) print "Zahl 2: "
(5) zahl2 = gets.chomp
(6) ergebnis = zahl1 + zahl2
(7) puts "Ergebnis: " + ergebnis
```

Wählt der Benutzer als erste Zahl 2 und als zweite 4, ist als Ausgabe (kurzes Fingerrechnen) eine 6 zu erwarten:

```
Ergebnis: 24
```

?! Der Computer rechnet hier nicht falsch, wie zunächst anzunehmen wäre, der Fehler liegt vielmehr in der Tatsache begründet, dass hier schlicht zwei Strings „addiert“ werden. „gets“ liefert ausschließlich Strings zurück, also ist auch die Variable ergebnis (Zeile 6) als Summe zweier Strings automatisch vom Typ String.

Damit die Variable „zahl1“ als Integer gilt, muss die Eingabe umgewandelt werden:

```
rechnen5.rb
(1) puts "Hallo! Ich addiere zwei Zahlen, die du vorgibst!"
(2) print "Zahl 1: "
(3) eingabe1 = gets.chomp
(4) zahl1 = eingabe1.to_i
(5) print "Zahl 2: "
(6) eingabe2 = gets.chomp
(7) zahl2 = eingabe2.to_i
(8) ergebnis = zahl1 + zahl2
(9) puts "Ergebnis: " + ergebnis
```

Die angekündigte Umwandlung erfolgt in den Zeilen 4 und 7. Die Methode „to_i“ bedeutet „to Integer“.

Leider ist die Ausgabe (noch) nicht die erhoffte:

```
rechnen5.rb:9:in `+': cannot convert Fixnum into String (TypeError)
from rechnen5.rb:9
```

Womit wir beim nächsten Thema angekommen sind, den Fehlermeldungen des Interpreters.

Fehlermeldungen

Was beim Ausführen des Programms `rechnen5.rb` auftritt ist keine „gewollte“ Ausgabe, sondern eine Fehlermeldung des Interpreters.

Vor allem bei großen und komplexen Programmen ist es an der Regel, dass ein Programm nach dem ersten Hinschreiben noch so genannte syntaktische Fehler enthält. Damit sind Ungereimtheiten gemeint, z.B. ein falsch geschriebener Befehl oder, wie im Beispiel eben, wenn du versuchst, einen Integer wie einen String zu verwenden. Dazu gleich mehr.

Fehlermeldungen weisen nicht nur auf einen Fehler im Allgemeinen hin, sondern haben vielmehr den Zweck, den Programmierer auf den Ort und die Art des Fehlers hinzuweisen. Allerdings gehört ein klein wenig Übung dazu, aus diesen Fehlermeldungen schlau zu werden, lass dich deshalb am Anfang keinesfalls von dir zunächst unverständlichem Kauderwelsch entmutigen!

Zu Beginn unserer Fehlermeldung hier steht der Dateiname, darauf folgt die Nummer der Zeile, in der der Fehler auftritt. Spätestens jetzt lohnt es sich für dich, einen Editor dein Eigen zu nennen, der die Zeilennummern auch anzeigt.

Darauf folgt die eigentliche Fehlermeldung:

```
cannot convert Fixnum into String (TypeError)
```

Also ein „Fixum“ kann nicht in einen „String“ konvertiert werden. Nun, das Fixum ist hier ein Integer, unser „ergebnis“. Der Interpreter macht uns darauf aufmerksam, dass der Wert der Integervariable „ergebnis“ nicht in einen für die Ausgabe notwendigen String umgewandelt werden kann.

Genauer: nicht automatisch. Denn es geht sehr wohl. Nur muss der Programmierer das selbst erledigen.

Um einen Integer in einen String umzuwandeln, steht in Ruby die Methode „`to_s`“ zur Verfügung.

Der umgekehrte Weg, einen String in einen Integer umzuwandeln, wird übrigens folgerichtig mit der Methode „`to_i`“ beschritten, wie wir bereits gesehen haben.

Zeile 9 muss also so lauten:

```
puts "Ergebnis: " + ergebnis.to_s
```

Das korrigierte Programm liefert dann auch das erwartete Ergebnis:

```
Hallo! Ich addiere zwei Zahlen, die du vorgibst!  
Zahl 1: 2  
Zahl 2: 4  
Ergebnis: 6
```

Arbeitsauftrag: die interne Zeitrechnung des Computers beginnt mit dem 1. Januar 1970. Das folgende Programm gibt die Sekunden seit diesem Tag, sogar bis auf die hundert tausendstel Sekunde genau, aus.

```
(1) t = Time.now  
(2) puts t #aktuelle Zeit ausgeben  
(3) zeit_sec = t.to_f #Zeit in Sekunden seit 1. Januar 1970  
(4) puts "Sekunden seit 1. Januar 1970: " + zeit_sec.to_s
```

Ausgabe:

```
Wed Sep 08 22:29:08 CEST 2004  
Sekunden seit 1. Januar 1970: 1094675348.27541
```

Ergänze das Programm so, dass die verstrichene Zeit in Jahren, Monaten, Tagen, Stunden, Minuten und Sekunden ausgegeben wird, z.B. könnte die Ausgabe so lauten: Seit dem 1. Januar 1970 sind 35 Jahre, 9 Monate, 2 Wochen, 3 Tage, 16 Stunden, 6 Minuten und 3 Sekunden vergangen.

Kommentare

Im letzten Arbeitsauftrag siehst du in den Zeilen 2 und 3 einen „Zaun“ (#). Dieses Zeichen leitet einen Kommentar ein.

Kommentare werden im Quellcode vorwiegend aus dem Grund eingefügt, die Verständlichkeit des Codes zu verbessern. Der Interpreter ignoriert eine Zeile ab dem Kommentarzeichen bis zum Zeilenende.

Kommentare, die sich über mehrere Zeilen erstrecken, kannst du auch mit „=begin“ einleiten und „=end“ abschließen, allerdings müssen beide Ausdrücke am Zeilenanfang stehen.

Bedingungen und Verzweigungen

Wenn jemand eine Bedingung stellt, macht er weiteres Vorgehen von einem bestimmten Umstand abhängig. Beispielsweise bleibt Frau Nagl nur dann in der Firma, wenn sie einen Arbeitsplatz am Fenster bekommt.

In unserem Beispiel wird der Benutzer nur dann begrüßt, wenn er den richtigen Namen hat.

```
gruss.rb
(1) print "Deinen Namen, bitte: "
(2) name = gets.chomp
(3) if name == "Franz"
(4)   puts "Hallo Franz! Franz ist aber ein schöner Name!"
(5) end
(6) puts „das wars...“
```

In Zeile 3 siehst du die Bedingung

```
if name == "Franz"
```

Die zwei hintereinander geschalteten „=-Zeichen stellen den Vergleichsoperator dar, ein „=“-Zeichen ist, wie bereits gesehen, ein Zuweisungsoperator (z.B. name = „Franz“)

Ist die Bedingung wahr, wird „true“ zurückgeliefert und der Block bis „end“ wird ausgeführt. Deshalb wird der Codeblock auch der Übersichtlichkeit wegen eingerückt.

Im anderen Fall ist die Bedingung falsch, d.h. es wird „false“ zurückgeliefert und die Abarbeitung des Codes wird erst nach „end“ fortgesetzt. Die Ausgabe „das wars...“ erfolgt also in jedem Fall.

Du kannst den Code auch verzweigen. Im nächsten Beispiel wollen wir erreichen, dass alle anderen auch noch einen Gruß „weg kriegen“:

```
gruss2.rb
(1) print "Deinen Namen, bitte: "
(2) name = gets.chomp
(3) if name == "Franz"
(4)   puts "Hallo Franz! Franz ist aber ein schöner Name!"
(5) else
(6)   puts "Hallo " + name + "!"
(7) end
(8) puts "das wars..."
```

Jetzt wird Franz mit einem Extragruß bedacht, aber auch alle anderen gehen nicht mehr leer aus.

Du kannst auch beliebig viele Unterscheidungen(=Verzweigungen) vorsehen:

gruss3.rb

```
(1) print "Deinen Namen, bitte: "  
(2) name = gets.chomp  
(3) if name == "Franz"  
(4)   puts "Hallo Franz! Wie geht's?"  
(5) elsif name == "Evi"  
(6)   puts "Hallo Evi! Schön, dich zu treffen!"  
(7) elsif name == "Hugo"  
(8)   puts "Hallo Hugo! Lange nicht gesehen!"  
(9) else  
(10)  puts "Hallo " + name + "!"  
(11)end  
(12)puts "das wars..."
```

Die Struktur schaut im Überblick so aus:

```
if Bedingung  
  Anweisungen  
elsif Bedingung  
  Anweisungen  
elsif Bedingung  
  Anweisungen  
...  
else  
  Anweisungen  
end
```

elsif- und else-Zweige können je nach Bedarf, wie in gruss.rb , auch wegfallen.

Arbeitsauftrag: schreibe ein Programm, welches eine Zahl einliest und in der Konsole die Meldung ausgibt, ob die Zahl durch 2, 3, beide Zahlen oder keine der beiden teilbar ist!
Hinweise: der Operator „%“ (modulo) gibt den Rest bei ganzzahliger Division zurück. Zum Beispiel ist $10 \% 3 = 1$, weil bei ganzzahliger Division von 10 und 3 ($10/3$) eins Rest bleibt.

„&&“ und „and“ stellen das logische „und“, „||“ und „or“ das logische „oder“ dar.

Wiederholungen (Schleifen)

Du wirst als Programmierer mit Sicherheit immer wieder in die Situation geraten, den Computer zu beauftragen, denselben Arbeitsschritt öfters durchzuführen.

schleife0.rb

```
(1) puts "Franz"  
(2) puts "Franz"  
(3) puts "Franz"
```

Dieses Programm tut nichts weiter, als drei mal denselben Namen zu schreiben. Wenn schon nicht stilvoll, so ist es dennoch für den Programmierer keine nennenswerte Arbeit, eine derart kurze Zeile zwei mal zu wiederholen.

Wenn diese Zeile aber nicht drei mal, sondern dreißig mal, dreihundert mal, dreitausend mal zu schreiben ist?

Hier kommen Schleifen ins Spiel, es gibt deren mehrere, eine sehr einfache Form ist folgende:

times-Schleife

schleife1.rb

```
(1) 300.times do  
(2)   puts "Franz"  
(3) end
```

Arbeitsauftrag: schreibe ein Programm, das den Benutzer fragt, wie oft ein einzugebender Name geschrieben werden soll und das mittels einer times-Schleife dann auch macht!

while-Schleife

Das nächste Programm verwendet eine while-Schleife zum Auflisten der Zahlen von 100 bis einschließlich 200:

```
schleife2.rb
(1) zahl = 100
(2) while zahl <= 200
(3)     puts zahl.to_s
(4)     zahl = zahl + 1
(5) end
```

Dasselbe noch mal mit einer until-Schleife:

until-Schleife

```
schleife3.rb
(1) zahl = 100
(2) until zahl > 200
(3)     puts zahl.to_s
(4)     zahl = zahl + 1
(5) end
```

Bei den beiden vorherigen Schleifen erfolgt die Prüfung, ob die Schleife durchlaufen wird, zu Beginn, die Schleife wird also unter Umständen nie durchlaufen.

while – und until-Schleifen zumindest einmal durchlaufen

Die Schleife in folgendem Beispiel wird nie durchlaufen, wenn der Benutzer eine Zahl größer als 100 eingibt:

```
schleife4.rb
(1) print "Gib bitte eine Zahl <= 100 ein: "
(2) eingabe = gets.chomp
(3) zahl = eingabe.to_i
(4) while zahl <= 100
(5)     puts zahl.to_s
(6)     zahl = zahl + 1
(7) end
```

In manchen Fällen möchte der Programmierer jedoch, dass die Schleife auf jeden Fall zumindest einmal durchlaufen wird. In diesem Fall verschiebt man die Prüfung nach hinten:

```

schleife5.rb
(1) print "Gib bitte eine Zahl <= 100 ein: "
(2) eingabe = gets.chomp
(3) zahl = eingabe.to_i
(4) begin
(5)     puts zahl.to_s
(6)     zahl = zahl + 1
(7) end while zahl <= 100

```

Die Schleife läuft von Zeile 4 bis Zeile 7, die Prüfung erfolgt am Ende, sodass in jedem Fall zumindest eine Zahl ausgegeben wird.

for-Schleife

Eine weitere sehr wichtige Form der Schleife ist die for-Schleife:

```

schleife6.rb
(1) for i in 100..200
(2)     puts i.to_s
(3) end

```

Die Variable „i“ durchläuft hier alle Zahlen von 100 bis einschließlich 200. Die zwei Punkte in 100..200 stehen für den Bereich, fügen du einen dritten Punkt ein, wird die Obergrenze nicht eingeschlossen, die letzte ausgegebene Zahl wäre somit 199.

In vielen Fällen ist es schlicht egal (oder besser: deinem persönlichen Geschmack überlassen), für welche Art von Schleife du dich entscheidest, so kann beispielsweise jede for-Schleife durch eine while- oder until-Schleife ersetzt werden.

Arbeitsaufträge:

- 1) Es sollen alle Zahlen von 1 bis 100 aufsummiert werden. Schreibe dazu sechs Programme mit jeweils einer der sechs bisher behandelten Schleifenformen (times-, while-, until-, while- und until- mit Überprüfung am Ende, for-Schleife)!
- 2) Es sollen alle durch 3 teilbaren Zahlen zwischen 1 und 1000 ausgegeben werden. Schreibe dazu sechs Programme mit jeweils einer der sechs unterschiedlichen Schleifenformen!
- 3) Schreibe ein Programm, das die Teiler einer beliebigen Zahl ausgibt!

- 4) Es sollen zwei Zahlen (a und b) eingelesen und die Potenz (a hoch b) ausgegeben werden. Schreibe dazu sechs Programme mit jeweils einer der sechs unterschiedlichen Schleifenformen!
- 5) Schreibe ein Programm, welches von der Konsole eine Zahl einliest und ausgibt, ob es sich dabei um eine Primzahl handelt.
- 6) Schreibe ein Programm, welches alle Primzahlen kleiner gleich der eingegebenen Zahl in der Konsole ausgibt.
- 7) Schreibe ein Programm, welches das x-malige Werfen mit einem Würfel simuliert. Gib aus, wie oft welche Augenzahl vorgekommen ist. Hinweis: die Methode „rand(n)“ gibt eine Zufallszahl größer gleich 0 und kleiner n zurück.

loop-Schleife

Eine loop-Schleife wird, wenn du nicht eingreifst, endlos ausgeführt:

```
loop.rb
(1) i = 0
(2) loop do
(3)   puts i.to_s
(4)   i += 1
(5) end
```

Das Programm lässt sich mit der Tatenkombination Strg+C abbrechen. Dasselbe Ergebnis lässt sich natürlich auch mit einer while- (oder until-) Schleife erreichen:

```
(1) i = 0
(2) while true do
(3)   puts i.to_s
(4)   i += 1
(5) end
```

while (Zeile 2) verlangt nach einer Bedingung, welche ausgewertet wird und „true“ oder „false“ zurück liefert. Wenn, wie hier, „true“ steht, ist das dasselbe wie eine Bedingung, die immer richtig ist, z.B.:

```
while 10 == 10 do
  ...
```

Schleifen manipulieren

Du kannst das standardmäßige Ab- bzw. Durchlaufen der Schleifen beeinflussen, um beispielsweise eine loop-Schleife zu beenden, aber auch um ein unnötiges Weiterlaufen einer Schleife zu verhindern, wenn etwa das Ziel bereits erreicht worden ist.

Zum angesprochenen Verändern des Ablaufs von Schleifen dienen die Schlüsselwörter `break`, `redo`, `next` und `retry`.

„`break`“ verlässt die Schleife, auch wenn deren „natürliches“ Ende noch nicht gekommen ist:

`zwei_teiler.rb` gibt die beiden kleinsten Teiler einer Zahl zurück:

```
(1) print "Zahl: "  
(2) zahl = gets.chomp.to_i  
(3)  
(4) for i in 1..zahl do  
(5)   if zahl % i == 0  
(6)     puts i.to_s  
(7)     if i > 1  
(8)       break  
(9)     end  
(10)  end  
(11)end
```

Beim ersten Schleifendurchlauf hat `i` den Wert 1 und ist damit immer ein Teiler der eingegebenen (natürlichen) Zahl. Da das Programm zwei Teiler der Zahl ausgeben soll, wird durch die Bedingung in Zeile 7 gewährleistet, dass „`break`“ in Zeile 8 erst beim zweiten Teiler ausgeführt wird.

Die Methode „`redo`“ durchläuft die Schleife nochmal mit demselben Index, „`next`“ macht mit dem nächsten Index weiter und „`retry`“ beginnt den Schleifendurchlauf nochmal von ganz vorne.

Arrays

Stellst du dir eine Variable wie einen Schrank mit einem Fach vor, so ist ein Array der Größe n wie ein Schrank mit n Fächern, die von 0 bis $n - 1$ durchnummeriert sind.

Arrays werden primär dann verwendet, wenn mehrere Variablen gleichen Datentyps im selben Zusammenhang anfallen.

berge.rb

```
(1) berge = [ "Kronplatz", "Plose", "Haunold", "K2" ]
(2) puts "1. Berg im Array: " + berge[ 0 ];
(3) puts "4. Berg im Array: " + berge[ 3 ];
```

Ausgabe:

```
1. Berg im Array: Kronplatz
4. Berg im Array: K2
```

In Zeile 1 wird das Array initialisiert, es beinhaltet vier Strings. Auf die einzelnen Werte des Arrays wird über Indizes zugegriffen, das erste Element hat den Index 0, das zweite den Index 1, das n -te den Index $n-1$. Die vier Elemente des Arrays `berge` tragen also die Indizes 0, 1, 2 und 3.

Hinweis: da Ruby mit dem Ziel entwickelt wurde, dem Programmierer das Schreiben von Code so weit wie möglich zu erleichtern, gibt es auch beim Initialisieren von String-Arrays eine kürzere Schreibweise:

```
berge = [ "Kronplatz", "Plose", "Haunold", "K2" ]
```

lässt sich auch einfacher so schreiben:

```
berge = %w( Kronplatz Plose Haunold K2 )
```

Arrays können natürlich auch während des Programmablaufes „bestückt“ werden:

berge2.rb

```
(1) berge = []
(2) for i in 0..3
(3)   print "Gib bitte einen Berg ein: "
(4)   berge[ i ] = gets.chomp
(5) end
(6)
(7) for i in 0..berge.length
(8)   puts berge[ i ]
(9) end
```

In Zeile 1 wird mit dem Befehl „berge = []“ ein (noch) leeres Array erzeugt. In der darauf folgenden for-Schleife werden dann vier Elemente in das Array eingefügt. Das Array passt seine Größe während der Laufzeit dynamisch an.

Die nächste Schleife (Zeilen 7 bis 9) gibt die eingegebenen Berge in der Konsole aus. Die Obergrenze der for-Schleife lautet hier `berge.length`, diese Methode gibt die Anzahl der Elemente, die im Array enthalten sind, zurück.

Ausgabe:

```
Gib bitte einen Berg ein: Mount Everest
Gib bitte einen Berg ein: Kilimandscharo
Gib bitte einen Berg ein: Ortler
Gib bitte einen Berg ein: Zugspitze
Mount Everest
Kilimandscharo
Ortler
Zugspitze
nil
```

Als abschließende Ausgabe siehst du hier „nil“ (not in list), was soviel bedeutet wie „nicht vorhanden“.

Der Grund für diese Ausgabe ist, dass die zweite Schleife im Programm fünf mal durchlaufen wird, da `berge.length` den Wert 4 zurückgibt. „i“ nimmt also die Werte 0, 1, 2, 3 und 4 an (5 Werte!), im Array „berge“ sind jedoch nur die Indizes 0 bis 3 verfügbar.

Um dies zu korrigieren, kannst du Zeile 7 so abändern:

```
for i in 0..berge.length - 1
```

oder

```
for i in 0...berge.length
```

Um ein Objekt am Ende des Arrays einzufügen, kannst du die (etwas untypische) Methode „<<“ verwenden:

```
zahlen_einlesen.rb
(1) zahlen = []
(2) for i in 0..10
(3)     zahlen << i
(4) end
```

Arbeitsauftrag: schreibe ein Programm, das so lange Berge einliest, bis der Benutzer „a“ für abbrechen eingibt, nachher sollen die Berge aufgelistet werden.

Methoden

Ich habe den Begriff „Methode“ bereits im Zusammenhang mit „to_s“ und „to_i“ verwendet. Jetzt wirst du kennen lernen, was es damit auf sich hat und wirst weiter lernen, eigene Methoden zu schreiben.

Es folgen zwei Programme, beide addieren zwei Zahlen, das erste wie gehabt, das zweite mit Hilfe einer Methode:

addieren1.rb

```
(1) print "Zahl 1: "  
(2) eingabe1 = gets.chomp  
(3) zahl1 = eingabe1.to_i  
(4) print "Zahl 2: "  
(5) eingabe2 = gets.chomp  
(6) zahl2 = eingabe2.to_i  
(7) ergebnis = zahl1 + zahl2  
(8) puts "Ergebnis: " + ergebnis.to_s
```

addieren2.rb

```
(1) def addieren( eing1, eing2 )  
(2)   z1 = eing1.to_i  
(3)   z2 = eing2.to_i  
(4)   erg = z1 + z2  
(5)   return erg.to_s  
(6) end  
(7)  
(8) print "Zahl 1: "  
(9) eingabe1 = gets.chomp  
(10) print "Zahl 2: "  
(11) eingabe2 = gets.chomp  
(12) ergebnis = addieren( eingabe1, eingabe2 )  
(13) puts "Ergebnis: " + ergebnis
```

Die Methode (Zeilen 1 bis 6) nennt sich addieren. In den angehängten runden Klammern steht die Argumentenliste, das sind die Werte, die der Methode übergeben werden.

Eine Methode wird durch das Schlüsselwort „def“ eingeleitet und mit „end“ abgeschlossen.

Die Methode „addieren“ wandelt die zwei ihr übergebenen Strings in Integer um und addiert diese. Das Ergebnis wird in einen String umgewandelt und das Schlüsselwort „return“ sorgt dafür, dass der darauf folgende Wert zurückgegeben wird (Zeile 5). Im Beispiel wird das Ergebnis in der Variable ergebnis (Zeile 12) gespeichert.

Du fragst dich, wo da der Vorteil des zweiten Programms gegenüber dem ersten liegt. Gute Frage, immerhin ist das zweite Programm um vier Zeilen (entspricht 50 Prozent!) länger.

Ein nicht zu verachtender Vorteil liegt in der Übersichtlichkeit: durch die Verwendung der Methode gelingt eine klarere Strukturierung.

Ein weiterer Punkt auf der Habenseite ist die Wiederverwendbarkeit: jedes Mal, wenn du weitere zwei Zahlen addieren möchtest, genügt ein einfacher Methodenaufruf.

Drittens: wenn du später den Additionsalgorithmus verändern willst, weil du beispielsweise die Methode effizienter gestalten möchtest oder weil sich gar ein Fehler darin befindet, reicht es aus, den Code an einer Stelle zu verändern.

Als Faustregel kannst du dir merken, dass im Zweifelsfall die Verwendung einer Methode immer vorzuziehen ist.

Packe auch nicht zu viele Abläufe in eine und dieselbe Methode. Schreibe lieber mehrere, spezialisierte Methoden.

Arbeitsauftrag: schreibe ein Programm, das zwei Zahlen a und b einliest und die Potenz (a hoch b) ausgibt. Verwende für das Potenzieren eine Methode. Übergib dazu der Methode zwei Integer-Werte und gib einen eben solchen zurück.

Objekt-Orientierte-Programmierung

Die eben angesprochenen Vorteile der Methoden sind auch Schlagworte im Zusammenhang mit Objekt-Orientierter-Programmierung (OOP).

OOP soll weiters den Grad der Abstraktion vergrößern, was im Wesentlichen bedeutet, dass du als Programmierer das reale Problem annähernd so umsetzen kannst, wie es sich dir darstellt und du nicht die Problemstellung für die Programmiersprache völlig neu modellieren musst.

Du hast bisher ganz nebenbei in jedem Programm mit Objekten gearbeitet. Wenn du etwa einen String anlegst, erzeugst du ein Objekt. Ein Objekt der Klasse String.

Hier haben wir bereits eine wichtige Grundlage herausgearbeitet: den Zusammenhang zwischen Klasse und Objekt. Eine Klasse ist, bildlich gesprochen, der Bauplan für z.B. das Haus, der die Eigenschaften der Häuser, die nach diesem Plan erstellt werden, beschreibt.

Ein Beispiel: wir wollen im Auftrag einer Stadt Häuser katalogisieren. Dazu sind die Eigenschaften Quadratmeter, die Anzahl der Räume und die Anzahl der dort lebenden Personen zu erfassen.

```
haus.rb
(1) class Haus
(2)   @quadratmeter
(3)   @anzahl_raeume
(4)   @anzahl_personen
(5) end
```

Damit haben wir die Klasse und somit den Plan für die zu registrierenden (und damit realen) Häuser erstellt und können diese nun anlegen. Jedes Haus ist also ein Objekt der Klasse Haus, ebenso spricht man vom Objekt als Instanz der Klasse.

Die Eigenschaften einer Klasse (hier `quadratmeter`, `anzahl_raeume` und `anzahl_personen`) nennt man Membervariablen, Instanzvariablen, Attribute, Eigenschaften oder Felder einer Klasse.

Die Instanzvariablen haben in Ruby einen vorangestellten „Klammeraffen“ (@).

Ein Objekt dieser Klasse erzeugst du mit dem Befehl `new`:

```
Haus.new
```

Sinnvollerweise schaffst du noch eine Referenz auf das Objekt, sprich eine Variable, die das neue Haus als Inhalt hat:

```
mein_haus = Haus.new
```

Nun kannst du die Eigenschaften deines neuen Hauses definieren. Auf Membervariablen und Methoden einer Klasse greifst du über die Punktnotation zu:

```
mein_haus.quadratmeter = 130
```

Allerdings erzeugt dieser Zugriff einen Fehler:

```
undefined method `quadratmeter=' for #<Haus:0x4026bfac>  
(NoMethodError)
```

Am Beginn der Fehlermeldung steht hier: `undefined method quadratmeter`.

Nun, Methode ist „quadratmeter“ auch keine, sondern eine Instanzvariable. Die Fehlermeldung zeigt trotzdem in die richtige Richtung: Instanzvariablen können standardmäßig nicht direkt verändert werden, der Zugriff darauf erfolgt über Methoden.

Es sind dies die so genannten „setter-“ und „getter-“ Methoden, wobei „set“ für das Schreiben und `get` für das Auslesen der Eigenschaft steht (siehe nächstes Beispiel).

Instanz-Methoden (OOP)

Mittels Methoden gelingt lesender und schreibender Zugriff auf die Membervariablen der Haus-Objekte:

```
haus2.rb
(1) class Haus
(2)   @quadratmeter
(3)   @anzahl_raeume
(4)   @anzahl_personen
(5)
(6)   #Methoden zum Setzen der Eigenschaften
(7)   def set_quadratmeter( qm )
(8)     @quadratmeter = qm
(9)   end
(10)
(11)  def set_anzahl_raeume( ar )
(12)    @anzahl_raeume = ar
(13)  end
(14)
(15)  def set_anzahl_personen( ap )
(16)    @anzahl_personen = ap
(17)  end
(18)
(19)  #Methoden zum Auslesen der Eigenschaften
(20)  def get_quadratmeter
(21)    return @quadratmeter
(22)  end
(23)
(24)  def get_anzahl_raeume
(25)    return @anzahl_raeume
(26)  end
(27)
(28)  def get_anzahl_personen
(29)    return @anzahl_personen
(30)  end
(31)end
(32)
(33)mein_haus = Haus.new
(34)mein_haus.set_quadratmeter( 70 )
(35)mein_haus.set_anzahl_raeume( 4 )
(36)mein_haus.set_anzahl_personen( 3 )
(37)
(38)puts "Quadratmeter: " + mein_haus.get_quadratmeter.to_s
(39)puts "Anzahl Raeume: " + mein_haus.get_anzahl_raeume.to_s
(40)puts "Anzahl Personen: " + mein_haus.get_anzahl_personen.to_s
```

Arbeitsauftrag: schreibe ein Programm, welches beliebig viele Häuser einliest; der Benutzer soll die Eigenschaften über die Konsole eingeben können. Verwende zum Referenzieren der Häuser ein Array. Am Ende sollen die Häuser mit ihren Eigenschaften aufgelistet werden.

Hinweis: zum Auflisten der Häuser kannst du der Klasse Haus die Fähigkeit, sich zu beschreiben, hinzufügen (also eine Methode, welche die Eigenschaften des Hauses als String zurück liefert):

```
def beschreibung
  return "Quadratmeter: " + @quadratmeter.to_s + "\n" + \
    "Anzahl Raeume: " + @anzahl_raeume.to_s + "\n" + \
    "Anzahl Personen: " + @anzahl_personen.to_s
end
```

Die Backslashes (\) am Ende der Zeilen zeigen lediglich an, dass der Interpreter alles als eine Zeile betrachten soll, du kannst natürlich den Quellcode ebenso gut in einer Zeile schreiben.

Konstruktor (OOP)

Der Konstruktor ist Teil einer jeden Klasse. Er ist eine Methode, die bei jedem Erzeugen eines Objektes (mit dem Schlüsselwort „new“) automatisch aufgerufen wird.

In Ruby wird der Konstruktor in Form der Methode „initialize“ implementiert. Wird in eine Klasse kein Konstruktor implementiert (gibt es also keine Methode „initialize“), so gilt der Standard-Konstruktor, der eine Instanz der Klasse mit nicht initialisierten Eigenschaften (=Instanzvariablen) erzeugt.

Im Beispiel haus3.rb sorgt der Konstruktor dafür, dass die Instanzvariablen des neu erstellten Objektes sofort initialisiert werden:

```
haus3.rb
(1) class Haus
(2)
(3)     #Konstruktor
(4)     def initialize( qm, ar, ap )
(5)         @quadratmeter = qm
(6)         @anzahl_raeume = ar
(7)         @anzahl_personen = ap
(8)     end
(9)
(10)    #Methode zur Beschreibung
(11)    def beschreibung
(12)        return "Quadratmeter: " + @quadratmeter.to_s + "\n" + \
(13)            "Anzahl Raeume: " + @anzahl_raeume.to_s + "\n" + \
(14)            "Anzahl Personen: " + @anzahl_personen.to_s
(15)    end
(16)
(17)end
(18)
(19)mein_haus = Haus.new( 60, 4, 5 )
(20)puts mein_haus.beschreibung
```

Die „getter-“ und „setter-“ Methoden können und sollen natürlich je nach Bedarf trotzdem vorkommen. Wolltest du etwa hier nachträglich beispielsweise die Quadratmeter ändern, wäre die Methode `set_quadratmeter` (siehe `haus2.rb`) nötig.

Direkt les- und schreibbare Membervariablen (OOP)

Die Methoden zum Lesen und Schreiben der Attribute eines Objektes sind sinnvoll, Ruby sieht jedoch auch die Möglichkeit vor, diese direkt anzusprechen. Zu diesem Zweck müssen die les- und schreibbaren Felder eigens angeführt werden.

haus4.rb

```
(1) class Haus
(2)
(3)   #Membervariablen
(4)   @quadratmeter
(5)   @anzahl_raeume
(6)   @anzahl_personen
(7)
(8)   #direkt auslesbare Attribute
(9)   attr_reader :quadratmeter, :anzahl_raeume
(10)
(11)  #direkt schreibbare Attribute
(12)  attr_writer :quadratmeter
(13)
(14)  #Konstruktor
(15)  def initialize( qm, ar, ap )
(16)    @quadratmeter = qm
(17)    @anzahl_raeume = ar
(18)    @anzahl_personen = ap
(19)  end
(20)
(21)  #Methode zur Beschreibung
(22)  def beschreibung
(23)    return "Quadratmeter: " + @quadratmeter.to_s + "\n" + \
(24)          "Anzahl Raeume: " + @anzahl_raeume.to_s + "\n" + \
(25)          "Anzahl Personen: " + @anzahl_personen.to_s
(26)  end
(27)
(28)end
(29)
(30)mein_haus = Haus.new( 60, 4, 5 )
(31)puts "Quadratmeter vorher: " + mein_haus.quadratmeter.to_s
(32)mein_haus.quadratmeter = 61
(33)puts mein_haus.beschreibung
```

In den Zeilen 9 und 12 werden die Zugriffsrechte auf die Attribute von Außen ermöglicht. Dies erst ermöglicht es, diese in den Zeilen 31 und 32 auszulesen bzw. zu ändern. Würdest du etwa in Zeile 32 nicht die Quadratmeter, sondern die Anzahl der Zimmer ändern wollen, würde der Interpreter einen Fehler melden.

Arbeitsauftrag: Schreibe ein Programm, welches Telefonnummern verwaltet. Erstelle für jede Person ein Objekt, die Eigenschaften Name und Telefonnummer sollen im Konstruktor gesetzt werden.

Siehe folgende Möglichkeiten vor:

- neue Daten eingeben
- alle Daten auflisten
- Telefonnummer einer Person suchen
- zu einer Telefonnummer die dazugehörige Person suchen
- Daten ändern
- Daten löschen

Vererbung (OOP)

Der Name sagt es bereits aus: ebenso wie Lebewesen ihre Merkmale und Fähigkeiten an die Nachkommen vererben, kann in der OOP eine Klasse ihre Merkmale (Membervariablen) und Fähigkeiten (Methoden) einer anderen Klasse vererben.

Die „Kindklasse“ bekommt durch die Vererbung präzise alle Felder und Methoden der Elternklasse, weder mehr noch weniger. Anstatt des Begriffs Elternklasse wird häufig auch der Begriff Superklasse verwendet.

Der wirkliche Sinn der Vererbung jedoch ist die Möglichkeit zur Spezialisierung. Zu diesem Zweck werden in der Kindklasse weitere Merkmale und/oder Methoden implementiert.

Ein Beispiel

Im folgenden Beispiel erweitern wir das Projekt zum Katalogisieren von Häusern.

Wir wollen jetzt auch Kirchen erfassen, dort ist für uns neben den Eigenschaften Quadratmeter, Anzahl der Räume und der Anzahl der Personen noch die Höhe des Kirchturms von Belang. Auch Methoden zum Schreiben und Auslesen der Membervariablen und zur Beschreibung sollen beide bekommen.

Eine Kirche hat somit alle Eigenschaften eines Wohnhauses und als weitere noch die Höhe des Kirchturms. In unserem Beispiel ist eine Kirche also ein Haus mit einem weiteren Merkmal – ein „spezialisiertes Haus“, deshalb leiten wir Kirche auch von „Haus“ ab:

kirche1.rb

```
(1) class Haus
(2)
(3)   #Membervariablen
(4)   @quadratmeter
(5)   @anzahl_raeume
(6)   @anzahl_personen
(7)
(8)   def initialize( qm, ar, ap )
(9)     @quadratmeter = qm
(10)    @anzahl_raeume = ar
(11)    @anzahl_personen = ap
(12)  end
(13)
(14)  #setter-Methoden
(15)  def set_quadratmeter( qm )
(16)    @quadratmeter = qm
(17)  end
(18)
(19)  def set_anzahl_raeume( ar )
(20)    @anzahl_raeume = ar
(21)  end
(22)
(23)  def set_anzahl_personen( az )
(24)    @anzahl_personen = az
(25)  end
(26)
(27)  #getter-Methoden
(28)  def get_quadratmeter
(29)    return @quadratmeter
(30)  end
(31)
(32)  def get_anzahl_raeume
(33)    return @anzahl_raeume
(34)  end
(35)
(36)  def get_anzahl_personen
(37)    return @anzahl_personen
(38)  end
(39)
(40)  def beschreibung
(41)    return "Quadratmeter: " + @quadratmeter.to_s + "\n" + \
(42)           "Anzahl Raeume: " + @anzahl_raeume.to_s + "\n" + \
(43)           "Anzahl Personen: " + @anzahl_personen.to_s
(44)  end
(45)end
(46)
(47)class Kirche < Haus
(48)
(49)  #Membervariablen
(50)  @hoehe_turm
(51)
(52)  #setter-Methoden
(53)  def set_hoehe_turm( ht )
(54)    @hoehe_turm = ht
(55)  end
(56)
```

```

(57)  #getter-Methoden
(58)  def get_hoehe_turm
(59)      return @hoehe_turm
(60)  end
(61)end
(62)
(63)eine_kirche = Kirche.new( 600, 10, 750 )
(64)eine_kirche.set_hoehe_turm( 56 )
(65)puts eine_kirche.beschreibung

```

Interessant ist Zeile 47:

```
class Kirche < Haus
```

Durch „< Haus“ wird die Vererbung vollzogen: die Klasse Kirche erbt alle Eigenschaften und Membervariablen der Klasse Haus.

Zusätzlich wird in der Klasse Kirche noch die Eigenschaft „hoehe_turm“ mit den entsprechenden getter- und setter-Methoden implementiert.

Da die Klasse Kirche über keinen eigenen Konstruktor verfügt, wird auf jenen der Elternklasse, also Haus, zurückgegriffen. Dadurch allerdings wird die Eigenschaft hoehe_turm nicht berücksichtigt. Deshalb muss im Beispiel diese Membervariable außerhalb des Konstruktors (Zeile 64) initialisiert werden.

Auch bei der Beschreibung wird die Höhe des Turms nicht berücksichtigt, da wiederum die Methode „beschreibung“ der Superklasse (Haus) aufgerufen wird:

```

Quadratmeter: 600
Anzahl Raeume: 10
Anzahl Personen: 750

```

Diese Rangordnung wird übrigens immer eingehalten: rufst du eine Instanz-Methode auf, wird diese Methode in der Klasse selbst gesucht. Wird sie dort nicht gefunden, wird versucht, die Methode gleichen Namens der Superklasse aufzurufen. Ist sie auch dort nicht zu finden, wird sie in der Superklasse der Superklasse gesucht und so fort. Wird die Methode in der gesamten Vererbungshierarchie nicht gefunden, setzt es eine Fehlermeldung.

Überschreiben von Methoden (OOP)

Um in unserem Beispiel der Klasse Kirche zu einer eigenen, spezialisierten Version des Konstruktors und der Methode „beschreibung“ zu verhelfen, ersetzen wir jene aus der Klasse Haus geerbte durch eigene.

Du ersetzt eine Methode aus der Superklasse „Haus“ einfach dadurch, dass du in der Kindklasse „Kirche“ die Methode mit demselben Namen nochmal deklarierst. Aufgrund der Rangordnung werden nun der Konstruktor und die Methode „beschreibung“ der Klasse Kirche und nicht mehr jene von Haus aufgerufen.

kirche2.rb

```
(1) class Haus
(2)     ...
(3) end
(4)
(5) class Kirche < Haus
(6)
(7)     #Membervariablen
(8)     @quadratmeter
(9)     @anzahl_raeume
(10)    @anzahl_personen
(11)    @hoehe_turm
(12)
(13)    #Konstruktor
(14)    def initialize( qm, ar, ap, ht )
(15)        @quadratmeter = qm
(16)        @anzahl_raeume = ar
(17)        @anzahl_personen = ap
(18)        @hoehe_turm = ht
(19)    end
(20)
(21)    #setter-Methoden
(22)    def set_hoehe_turm( ht )
(23)        @hoehe_turm = ht
(24)    end
(25)
(26)    #getter-Methoden
(27)    def get_hoehe_turm
(28)        return @hoehe_turm
(29)    end
(30)
(31)    #weitere Methoden
(32)    def beschreibung
(33)        return "Quadratmeter: " + @quadratmeter.to_s + "\n" + \
(34)            "Anzahl Raeume: " + @anzahl_raeume.to_s + "\n" + \
(35)            "Anzahl Personen: " + @anzahl_personen.to_s + "\n" + \
(36)            "Hoehe Kirchturm: " + @hoehe_turm.to_s
(37)    end
(38)
(39)end
```

```

(40)
(41) ein_haus = Haus.new( 120, 6, 4 )
(42) puts "Haus: \n" + ein_haus.beschreibung + "\n\n"
(43)
(44) eine_kirche = Kirche.new( 600, 10, 750, 56 )
(45) puts "Kirche: \n" + eine_kirche.beschreibung

```

Ausgabe:

```

Haus:
Quadratmeter: 120
Anzahl Raeume: 6
Anzahl Personen: 4

```

```

Kirche:
Quadratmeter: 600
Anzahl Raeume: 10
Anzahl Personen: 750
Hoehe Kirchturm: 56

```

Das Schlüsselwort „super“ (OOP)

Durch die Verwendung des Schlüsselwortes super kannst du Methoden und den Konstruktor der Superklasse aufrufen:

```

kirche3.rb
(1) class Haus
(2)   ...
(3) end
(4)
(5) class Kirche < Haus
(6)
(7)   #Membervariablen
(8)   @hoehe_turm
(9)
(10)  #Konstruktor
(11)  def initialize( qm, ar, ap, ht )
(12)    super( qm, ar, ap )
(13)    @hoehe_turm = ht
(14)  end
(15)
(16)  #setter-Methoden
(17)  def set_hoehe_turm( ht )
(18)    @hoehe_turm = ht
(19)  end
(20)
(21)  #getter Methoden
(22)  def get_hoehe_turm
(23)    return @hoehe_turm
(24)  end
(25)

```

```

(26)  #weitere Methoden
(27)  def beschreibung
(28)      super + "\nHoehe Kirchturm: " + @hoehe_turm.to_s
(29)  end
(30)
(31)end
(32)
(33)ein_haus = Haus.new( 120, 6, 4 )
(34)puts "Haus: \n" + ein_haus.beschreibung + "\n\n"
(35)
(36)eine_kirche = Kirche.new( 600, 10, 750, 56 )
(37)puts "Kirche: \n" + eine_kirche.beschreibung

```

Aufteilen eines Programmes auf mehrere Dateien

Das Programm „kirche3.rb“ sollte in dieser Form nicht vorkommen. Verwende vielmehr für jede Klasse eine eigene Datei:

```

haus3.rb
(1) class Haus
(2)
(3)     #Membervariablen
(4)     @quadratmeter
(5)     @anzahl_raeume
(6)     @anzahl_personen
(7)
(8)     def initialize( qm, ar, ap )
(9)         @quadratmeter = qm
(10)        @anzahl_raeume = ar
(11)        @anzahl_personen = ap
(12)    end
(13)
(14)    #setter-Methoden
(15)    def set_quadratmeter( qm )
(16)        @quadratmeter = qm
(17)    end
(18)
(19)    def set_anzahl_raeume( ar )
(20)        @anzahl_raeume = ar
(21)    end
(22)
(23)    def set_anzahl_personen( az )
(24)        @anzahl_personen = az
(25)    end
(26)
(27)    #getter-Methoden
(28)    def get_quadratmeter
(29)        return @quadratmeter
(30)    end
(31)

```

```

(32)  def get_anzahl_raeume
(33)    return @anzahl_raeume
(34)  end
(35)
(36)  def get_anzahl_personen
(37)    return @anzahl_personen
(38)  end
(39)
(40)  def beschreibung
(41)    return "Quadratmeter: " + @quadratmeter.to_s + "\n" + \
(42)          "Anzahl Raeume: " + @anzahl_raeume.to_s + "\n" + \
(43)          "Anzahl Personen: " + @anzahl_personen.to_s
(44)  end
(45)end

```

kirche.rb

```

(1)  require 'haus'
(2)
(3)  class Kirche < Haus
(4)
(5)    #Membervariablen
(6)    @hoehe_turm
(7)
(8)    #Konstruktor
(9)    def initialize( qm, ar, ap, ht )
(10)     super( qm, ar, ap )
(11)     @hoehe_turm = ht
(12)  end
(13)
(14)  #setter-Methoden
(15)  def set_hoehe_turm( ht )
(16)    @hoehe_turm = ht
(17)  end
(18)
(19)  #getter-Methoden
(20)  def get_hoehe_turm
(21)    return @hoehe_turm
(22)  end
(23)
(24)  #weitere Methoden
(25)  def beschreibung
(26)    super + "\nHoehe Kirchturm: " + @hoehe_turm.to_s
(27)  end
(28)
(29)end

```

hauptprogramm.rb

```
(1) require 'kirche'  
(2)  
(3) ein_haus = Haus.new( 120, 6, 4 )  
(4) puts "Haus: \n" + ein_haus.beschreibung + "\n\n"  
(5)  
(6) eine_kirche = Kirche.new( 600, 10, 750, 56 )  
(7) puts "Kirche: \n" + eine_kirche.beschreibung
```

Speichere alle drei Dateien im selben Verzeichnis.

Mit dem Schlüsselwort „require“ (Zeile 1 in kirche.rb und hauptprogramm.rb) bindest du im Programm benötigte Klassen ein, die Endung der Datei (.rb) kann dabei weggelassen werden.

Das Programm wird durch die Aufteilung übersichtlicher, weiter kannst du auf diese Weise eine der Klassen direkt in andere Programme übernehmen.

Es geht nun weiter mit Iteratoren.

Iteratoren

Iteratoren sind spezielle Methoden, die auf Einträge zugreifen lassen.

Viele Objekte besitzen Iteratoren, im Beispiel sehen wir ein Array:

```
tiere.rb
(1) tiere = [ "Katze", "Hund", "Pferd", "Kuh", "Huhn" ]
(2) tiere.each do
(3)     | tier |
(4)     puts "-> " + tier
(5) end
```

Ausgabe:

```
-> Katze
-> Hund
-> Pferd
-> Kuh
-> Huhn
```

Der Iterator „each“ sorgt dafür, dass die Schleife so oft durchlaufen wird wie Elemente im Array vorhanden sind. Weiters wird der jeweilige Wert auf die Variable zwischen den „Pipelines“ (|) gespeichert, hier also „tier“.

Weitere Beispiele für Iteratoren:

```
(1) 3.times do
(2)   | i |
(3)   puts i.to_s
(4) end
```

Ausgabe:

```
0
1
2
```

Es gibt noch zwei ähnliche Schleifentypen mit Iteratoren:

```
(1) 0.upto 2 do
(2)   | i |
(3)   puts i.to_s
(4) end
```

```
(1) 2.downto 0 do
(2)   | i |
(3)   puts i.to_s
(4) end
```

Blöcke

Du kannst Blöcke anstatt mit do-end auch durch geschwungenen Klammern definieren:

```
(1) tiere = [ "Katze", "Hund", "Pferd", "Kuh", "Huhn" ]
(2) tiere.each { | i | puts i.to_s }
```

Es hat sich eingebürgert, für Blöcke, die sich über mehrere Zeilen erstrecken, do-end zu verwenden, für einzeilige Blöcke geschwungene Klammern.

Umgang mit Dateien

Du wirst als Programmierer immer wieder mit Dateien zu tun haben. Mal will der Inhalt einer Textdatei eingelesen, vielleicht sogar bearbeitet und der geänderte Text wieder in derselben oder einer neuen Datei gespeichert werden.

Ein anderes Mal willst du beispielsweise dein Adressbuch sichern, damit die bereits getätigten Einträge nach dem nächsten Programmstart wieder vorhanden sind.

Unterscheide dabei grundsätzlich zwischen zwei Möglichkeiten: dem Umgang mit Textdateien, also Dateien, denen sich mit einem beliebigen Texteditor (sinnvolle) Daten entlocken lassen und jenem mit binären Dateien, in denen Ruby z.B. ganze Objekte speichert, um sie bei Bedarf wieder herzustellen. Zunächst zu ersteren:

Lesen und Schreiben von Textdateien

```
datei1.rb
(1) datei = File.new( "datei.txt", "r" )
(2) i = 0
(3) zeilen = []
(4) zeile = datei.gets
(5) while zeile != nil
(6)     zeilen[ i ] = zeile
(7)     i += 1
(8)     zeile = datei.gets
```

```

(9) end
(10)datei.close
(11)
(12)for i in 0...zeilen.length
(13)  puts zeilen[ i ]
(14)end

```

Ausgabe (entspricht dem Inhalt der Datei „datei.txt“):

```

zeile1
zeile2

zeile4

```

In Zeile 1 wird eine Datei geöffnet, das „r“ im Konstruktor veranlasst Ruby, die Datei im Lesemodus (readonly) zu öffnen.

Weitere Möglichkeiten sind „a“ (**a**ppend) zum Öffnen und Hinzufügen von Text zu einer bestehenden Datei (ist sie nicht vorhanden, wird sie neu angelegt), „w“ zum Öffnen der Datei zum Schreiben (**w**rite), eine eventuell vorhandene Datei wird in diesem Modus überschrieben.

Dasselbe Beispiel wie oben, jetzt mit Iterator:

```

datei2.rb
(1) datei = File.new( "test.txt", "r" )
(2) i = 0
(3) zeilen = []
(4) datei.each_line do
(5)   | zeile |
(6)   zeilen[ i ] = zeile
(7)   i += 1
(8) end
(9) datei.close
(10)
(11)zeilen.length.times do
(12)  | x |
(13)  puts zeilen[ x ]
(14)end

```

Schreiben einer Textdatei:

```

datei3.rb
(1) datei = File.new( "test2.txt", "w" )
(2) zahlen = [ 100, 21, 52, 10 ]
(3) for i in 0...zahlen.length
(4)   datei.puts zahlen[ i ].to_s
(5) end
(6) datei.close

```

Dasselbe nochmal mit Iterator:

```
datei4.rb
(1) datei = File.new( "test3.txt", "w" )
(2) zahlen = [ 100, 21, 52, 10 ]
(3) zahlen.length.times do
(4)   | x |
(5)   datei.puts zahlen[ x ].to_s
(6) end
(7) datei.close
```

Der wohl kaum überraschende Inhalt der Datei:

```
100
21
52
10
```

Erinnere dich immer daran, geöffnete Dateien wieder zu schließen, ansonsten droht Datenverlust.

Speichern von Objekten mit Hilfe von Textdateien

Es folgt ein Beispielprogramm, um Namen und Alter von Personen zu erfassen. Dazu wird eine Klasse „Person“ mit den entsprechenden Membervariablen erstellt. Eine Schleife ermöglicht es, mehrere Benutzer einzugeben, ohne das Programm dafür jedes Mal neu starten zu müssen.

```
person.rb
(1) class Person
(2)
(3)   #Membervariablen
(4)   @name
(5)   @alter
(6)
(7)   def initialize( n, a )
(8)     @name = n
(9)     @alter = a
(10)  end
(11)
(12)  #getter-Methoden
(13)  def get_name
(14)    return @name
(15)  end
(16)
(17)  def get_alter
(18)    return @alter
(19)  end
(20)
```

```

(21)  #weitere Methoden
(22)  def beschreibung
(23)      return @name + " ist " + @alter.to_s + " Jahr(e) alt."
(24)  end
(25)
(26)end

```

personenverwaltung.rb

```

(1) require 'person'
(2)
(3) personen = []
(4)
(5) begin
(6)     puts "(1)...Person hinzufuegen"
(7)     puts "(2)...Personen auflisten"
(8)     puts "(3)...Programm beenden\n"
(9)     print "->: "
(10)    auswahl = gets.chomp.to_i
(11)
(12)    if auswahl == 1
(13)        #Daten einlesen
(14)        print "Name: "
(15)        name = gets
(16)
(17)        print "Alter von " + name + ": "
(18)        alter = gets.chomp.to_i
(19)
(20)        #Objekt erzeugen und initialisieren
(21)        personen << Person.new( name, alter )
(22)    elsif auswahl == 2
(23)        personen.length.times do
(24)            | i |
(25)                puts personen[ i ].beschreibung
(26)        end
(27)    end
(28)
(29)end while auswahl != 3

```

Wie alle Objekte derselben Klasse unterscheiden sich die verschiedenen Objekte der Klasse Person lediglich in ihren Ausprägungen (Attributen), also im Namen und im Alter. Wenn du diese Werte in eine Textdatei schreibst, so kannst du die angelegten Objekte auch nach einem Neustart des Programms von Hand wieder herstellen.

personenverwaltung2.rb

```

(1) require 'person'
(2)
(3) personen = []
(4)

```

```

(5) #Daten von Datei einlesen...
(6) if File.exists?( "personen.txt" )
(7)   zeilen = []
(8)   datei = File.new( "personen.txt", "r" )
(9)   datei.each_line do
(10)    | zeile |
(11)    zeilen[ zeilen.length ] = zeile.chomp
(12)  end
(13)  datei.close
(14)
(15)  #...und Objekte wiederherstellen
(16)  if zeilen != nil
(17)    i = 0
(18)    begin
(19)      personen << Person.new( zeilen[ i ], zeilen[ i + 1 ] )
(20)      i += 2
(21)    end while i < zeilen.length
(22)  end
(23)end
(24)
(25)begin
(26)  puts "(1)...Person hinzufuegen"
(27)  puts "(2)...Personen auflisten"
(28)  puts "(3)...Programm beenden\n"
(29)  print "->: "
(30)  auswahl = gets.chomp.to_i
(31)
(32)  if auswahl == 1
(33)    #Daten einlesen
(34)    print "Name: "
(35)    name = gets.chomp
(36)
(37)    print "Alter von " + name + ": "
(38)    alter = gets.chomp.to_i
(39)
(40)    #Objekt erzeugen und initialisieren
(41)    personen << Person.new( name, alter )
(42)  elsif auswahl == 2 #auflisten
(43)    personen.length.times do
(44)      | i |
(45)      puts personen[ i ].beschreibung
(46)    end
(47)    print "weiter..."
(48)    gets
(49)  elsif auswahl == 3 #speichern
(50)    datei = File.new( "personen.txt", "w" )
(51)    personen.length.times do
(52)      | i |
(53)      datei.puts personen[ i ].get_name
(54)      datei.puts personen[ i ].get_alter.to_s
(55)    end
(56)    datei.close
(57)  end
(58)
(59)end while auswahl != 3

```

Dem logischen Ablauf folgend beschäftigen wir uns zunächst mit dem Speichern, dies geschieht ohne weiteres Zutun des Anwenders beim Beenden des Programms (Zeilen 50 bis 56).

Die Werte der Membervariablen aller Objekte der Klasse Person (zwei Werte je Objekt) werden zeilenweise in die Datei „personen.txt“ geschrieben. Die Zeilen 1 und 2 gehören also zum ersten Objekt, die Zeilen 3 und 4 zum nächsten usw.

Beim Starten des Programms wird der Inhalt der Datei „personen.txt“ zeilenweise in ein Array kopiert (Zeilen 6 bis 13) und anschließend die Objekte „rekonstruiert“ (Zeilen 16 bis 22).

Speichern von Objekten mittels Serialisierung

Bei der eben angesprochenen Art der Speicherung von Objekten in Dateien hat der Programmierer einige Arbeit zu erledigen.

Es ist aber auch möglich (und zwar sehr viel einfacher), das Speichern der Objekte Ruby zu überlassen. Das Format dieser Datei ist proprietär, nur Ruby selbst kann damit umgehen.

Den Vorgang, Objekte zu speichern und wiederherzustellen, nennt man „Serialisieren“ bzw. „De-Serialisieren“ oder auch „Marshalling“ bzw. „De-Marshalling“.

```
personenverwaltung3.rb
(1) require 'person'
(2)
(3) personen = []
(4)
(5) #von Datei einlesen
(6) if File.exists?( "personen.dat" )
(7)   datei = File.new( "personen.dat", "r" )
(8)   personen = Marshal.load( datei )
(9)   datei.close
(10)end
(11)
(12)begin
(13)  puts "(1)...Person hinzufuegen"
(14)  puts "(2)...Personen auflisten"
(15)  puts "(3)...Programm beenden\n"
(16)  print "->: "
(17)  auswahl = gets.chomp.to_i
(18)
(19)  if auswahl == 1
```

```

(20)      #Daten einlesen
(21)      print "Name: "
(22)      name = gets.chomp
(23)
(24)      print "Alter von " + name + ": "
(25)      alter = gets.chomp.to_i
(26)
(27)      #Objekt erzeugen und initialisieren
(28)      personen << Person.new( name, alter )
(29)  elsif auswahl == 2 #auflisten
(30)      personen.length.times do
(31)          | i |
(32)          puts personen[ i ].beschreibung
(33)      end
(34)      print "weiter..."
(35)      gets
(36)  elsif auswahl == 3 #speichern
(37)      datei = File.new( "personen.dat", "w" )
(38)      Marshal.dump( personen, datei )
(39)      datei.close
(40)  end
(41)
(42)end while auswahl != 3

```

Die Methode Marshal.dump (Zeile 38) serialisiert das Array (und damit alle enthaltenen Objekte) und speichert es in der Datei.

In Zeile 8 wird das Array aus der Datei wieder hergestellt.

Arbeitsauftrag: erweitere das Adressen-Verwaltungsprogramm von weiter oben um das automatische Speichern und Laden der Objekte mittels Serialisierung.

Dokumentation

Um die Möglichkeiten der mit Ruby mitgelieferten Klassen wie Integer, String oder Array auszuschöpfen, musst du auch wissen, welche diese Möglichkeiten sind, also welche Eigenschaften (Membervariablen) und Fähigkeiten (Methoden) die Klassen bereitstellen.

Dazu ist die Dokumentation da.

Die Klassendokumentation kannst du online unter der URL <http://www.ruby-doc.org/core/> einsehen. Du kannst sie auch offline verfügbar machen.

Mit Ruby wird auch der Konsolen-Dokumentationsbrowser „ri“ installiert (in Ubuntu-Linux mit „apt-get install ri“ nachholen).

Der Befehl „ri Array“ etwa liefert folgendes Ergebnis:

```
----- Class: Array
Sequences: Array#to_yaml
-----

Includes:
-----
Enumerable(all?, any?, collect, detect, each_with_index, entries,
find, find_all, grep, include?, inject, inject, map, max, member?,
min, partition, reject, select, sort, sort_by, sort_by, to_a,
to_set, to_set, zip)

Class methods:
-----
[], new

Instance methods:
-----
&, *, +, -, <<, <=>, ==, [], []=, abbrev, abbrev, assoc, at, clear,
collect, collect!, compact, compact!, concat, delete, delete_at,
delete_if, each, each_index, empty?, eql?, fetch, fill, first,
flatten, flatten!, frozen?, hash, include?, index, indexes,
indices, initialize_copy, insert, inspect, is_complex_yaml?, join,
last, length, map, map!, nitems, pack, pop, pretty_print,
pretty_print_cycle, push, quote, rassoc, reject, reject!, replace,
reverse, reverse!, reverse_each, rindex, select, shift, size,
slice, slice!, sort, sort!, to_a, to_ary, to_s, to_yaml,
to_yaml_type, transpose, uniq, uniq!, unshift, values_at, zip, |
```

Wenn du beispielsweise genaueres über die Methoden „delete_at“ der Klasse Array wissen möchtest, machst du dies mit dem Befehl

```
ri Array.delete_at oder ri Array#delete_at
```

Du erhältst folgendes Ergebnis:

```
----- Array#delete_at
array.delete_at(index)  -> obj or nil
-----
Deletes the element at the specified index, returning that element,
or +nil+ if the index is out of range. See also +Array#slice!+.

a = %w( ant bat cat dog )
a.delete_at(2)    #=> "cat"
a                 #=> ["ant", "bat", "dog"]
a.delete_at(99)  #=> nil
```

Du kannst das Ausgabeformat von ri auch anpassen, mehr dazu liefert „ri --help“. Der Befehl „ri Array --format ansi“ liefert beispielsweise eine farblich differenzierte Ausgabe.

Interactive Ruby (irb)

Interactive Ruby ist eine Ruby-Kommandozeile, in der du Ruby-Code eingeben und direkt ausführen kannst, es eignet sich deshalb hervorragend zum Testen von Code.

Du startest Interactive Ruby in der Konsole durch Eingabe von „irb“. Du wirst von folgendem Prompt begrüßt:

```
irb(main):001:0>
```

Nun kannst du auch schon loslegen:

```
irb(main):001:0> puts "Hallo!"
Hallo!
=> nil
irb(main):001:0>
```

Du kannst Interactive Ruby mit dem Befehl „quit“ verlassen.

Geschafft!

Sehr gut! Mit den bisher erlernten Techniken hast du das Rüstzeug für das Schreiben von Programmen nahezu beliebiger Vielfalt!

Vergiss dabei nie, dass nichts wichtiger ist beim Programmieren (und nicht nur dort!), als es mit Freude zu tun.

Es folgt noch ein weiteres Skriptum, das dir die Programmierung grafischer Oberflächen mit Ruby/GTK, einer Anbindung an die Grafikbibliothek GTK+, näher bringen wird.

Nützliche Links

Aktuelle Versionen dieses Skriptums und von jenem zur Programmierung grafischer Oberflächen in Ruby/GTK im PDF-Format: <http://franz.hob-brunneck.info/downloads/skripten/ruby/>

Offizielle Seite des Ruby-Projektes: <http://www.ruby-lang.org/>

Deutsche Übersetzung des „Pickaxe“-Buches: <http://home.vr-web.de/juergen.katins/ruby/buch/>

Wiki zu Ruby (mit vielen weiteren Links): <http://www.rubywiki.de/>

Anhang: Einrichten des Editors „ConTEXT“ (für Microsoft Windows)

ConTEXT (<http://www.context.cx/>) ist einer von vielen Editoren mit Hilfen wie Syntax-Highlighting und weiteren Hilfestellungen beim Schreiben deines Ruby-Codes.

ConTEXT hebt sich jedoch durch die Eigenschaft von vielen anderen ab, Ruby-Programme aus dem Editor heraus starten zu können (FreeRIDE beispielsweise kann dies zwar auch, hat dabei jedoch Schwierigkeiten bei der Anzeige von Fehlermeldungen).

Ruby-Sprachdatei nachrüsten

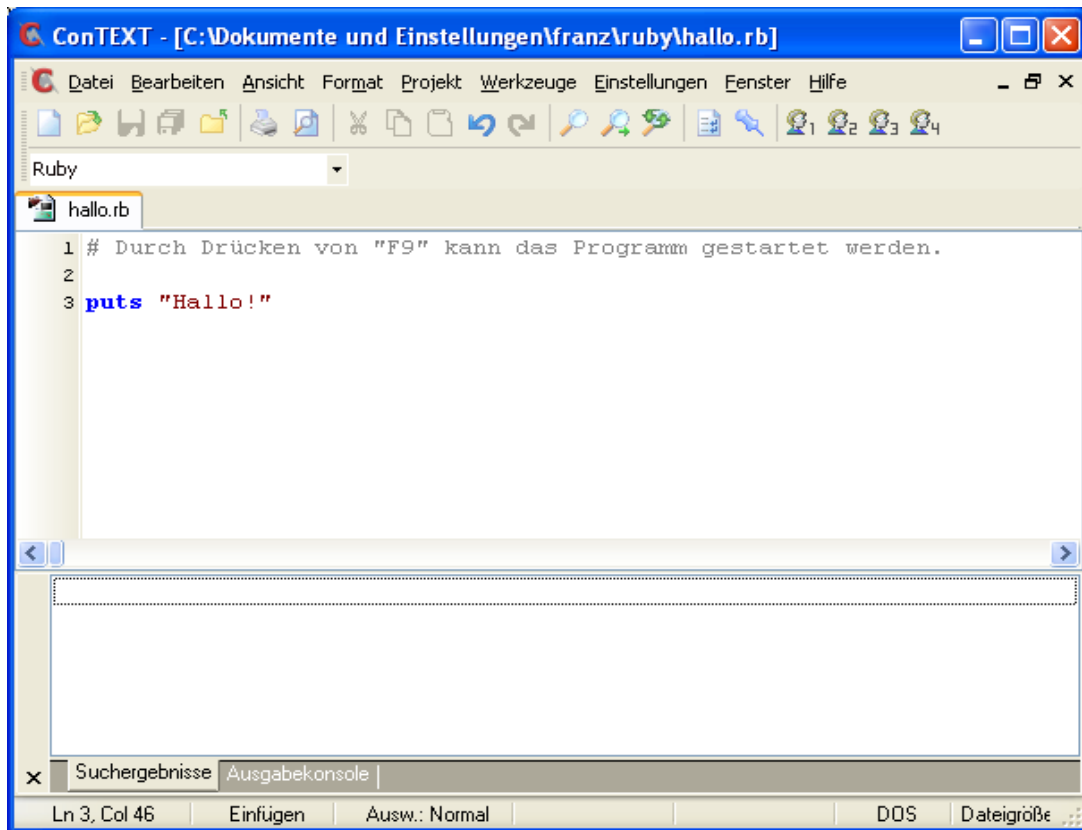
ConTEXT wird mittels einer Installationsroutine installiert, die Erkennung der Schlüsselwörter für Ruby mit dem damit verbundenen Syntax-Highlighting kann in Form der Datei „Ruby.chl“ nachgerüstet werden. Dazu kann die Datei von der Projektseite von ConTEXT heruntergeladen werden und wird im Ordner

C:\Programme\ConTEXT\Highlighters\

abgelegt (Pfad bitte gegebenenfalls anpassen). ConTEXT wird wie üblich über einen Eintrag im Startmenü gestartet.

Deutsche Sprache einstellen

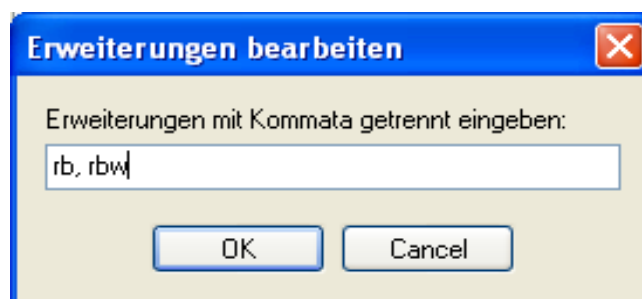
Diejenigen von euch, die deutschsprachige Menüs und Dialoge gegenüber ihren englischsprachigen Pendanten bevorzugen, können unter Options - Environment Options – General - Language die entsprechende Anpassung vornehmen.



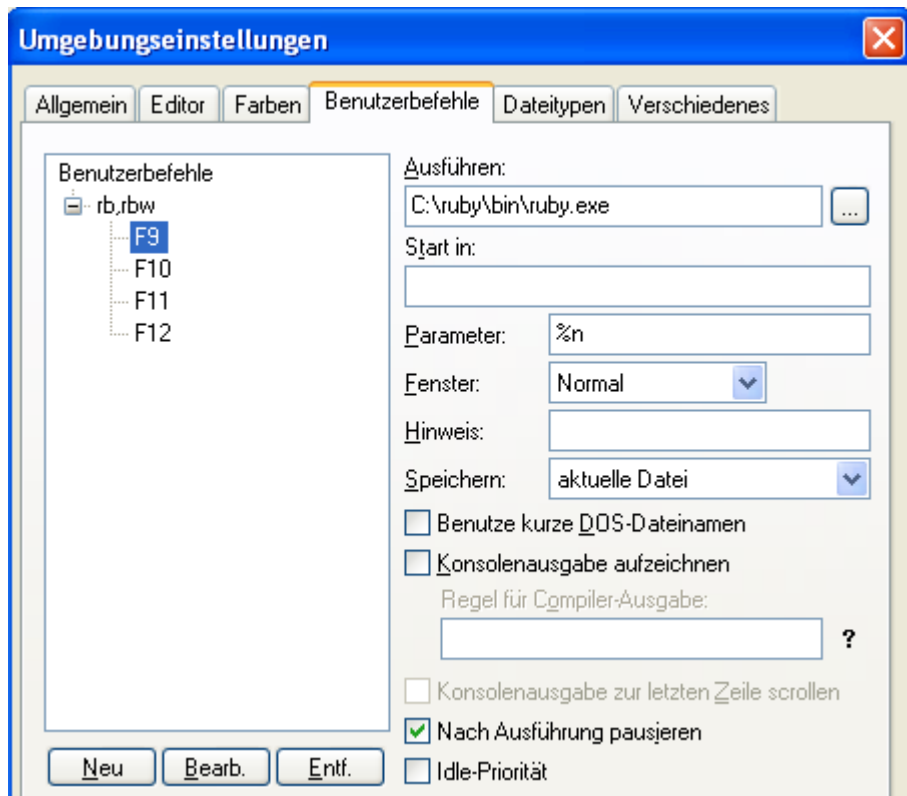
Damit Ruby-Programme jetzt, wie im Kommentar des Programms bereits angekündigt, durch Drücken von „F9“ gestartet werden kann, muss noch eine Einstellung erfolgen:

Unter dem Menüpunkt „Einstellungen-Umgebungseinstellungen“ können neben viele weiteren Optionen das gerade angesprochene Ausführen des Programms eingestellt werden, und zwar im Reiter „Benutzerbefehle“.

Nach Anwählen der Schaltfläche „Neu“ werden Ruby-Skripts, für die die Endungen „rb“ und „rbw“ vorgesehen sind, mit einer Aktion, dem Starten des Skripts, verknüpft.

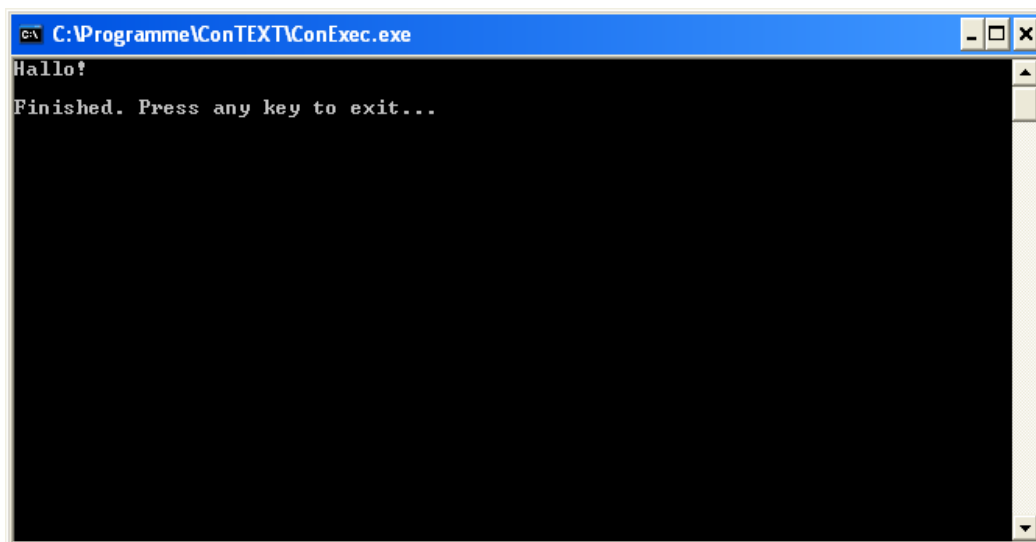


Du kannst die Einstellungen vom folgenden Screenshot übernehmen und gegebenenfalls deinen Wünschen anpassen.



Ruby-Programm starten

Anschließend kannst du das Programm durch Drücken der F9-Taste ausführen:



Weitere empfehlenswerte Optionen befinden sich im Reiter „Editor“, so ist etwa das Abstellen der „Smarten Tabulatoren“ sinnvoll, ebenso das Anschalten der Zeilennummern.

Stichwortverzeichnis

A	
Abstraktion.....	28
Array.....	24f., 30, 42, 48ff.
Attribut.....	28, 33, 46
Ausprägung.....	46
B	
Bauplan.....	28
Bedingung.....	17f., 22f.
Binärcode.....	5f.
Boolean.....	13
break.....	23
C	
Code.....	4ff., 16f., 24, 27, 31, 51, 53
Computersprache.....	5f.
ConTEXT.....	7, 53
D	
Datei.....	5f., 15, 39, 41, 43ff., 53
Daten.....	8, 12ff., 24, 34, 43, 45ff., 49
Dokumentation.....	50
E	
Editor.....	5, 7, 15, 43, 53, 55
Eigenschaft.....	28ff., 36, 50, 53
Eingabeaufforderung.....	7
Elternklasse.....	34, 36
Endung.....	6, 13, 27, 38, 41, 54
F	
Fehlermeldung.....	14f., 29, 36, 53
Feld.....	28, 33f.
Fixum.....	15
Float.....	12f.
FreeRIDE.....	53
H	
Highlighting.....	5, 53
I	
Installation.....	5, 7, 53
Instanz.....	28ff., 32, 36
Integer.....	12ff., 27, 50
Interpreter.....	6f., 14ff., 31, 33
irb.....	51
Iterator.....	41f., 44f.

K	
	Kindklasse.....34, 37
	Klasse.....28f., 31f., 34, 36ff., 41, 45f., 48, 50f.
	Kommandozeile.....7, 51
	Kommentar.....16, 54
	Konsole.....9, 11f., 18, 22, 25, 30, 50f.
	Konstruktor.....32f., 36ff., 40, 44
L	
	Linux.....6ff., 50
M	
	Mac OS.....6f.
	Marshalling.....48
	Maschinencode.....4
	Membervariable.....28ff., 33ff., 45, 48, 50
	Merkmale.....34
	Methode.....10f., 13ff., 22f., 25ff., 42, 45f., 49ff.
	modulo.....18
N	
	next.....23
	nil.....25, 43, 47, 51
O	
	Objekt.....25, 28ff., 32f., 42f., 45ff.
P	
	Pipeline.....42
	Primzahl.....22
	Programmieren.....4, 52
	Programmiersprache.....4f., 28
Q	
	Quellcode.....6, 16, 31
R	
	rand.....12, 22
	redo.....23
	Referenz.....29f.
	retry.....23
S	
	Schleife.....19ff., 25, 42, 45
	Schlüsselwort.....5, 23, 26f., 32, 38, 41, 53
	SciTE.....7
	Serialisieren.....48
	Speichern.....45, 47ff.
	Spezialisierung.....34
	Steuerzeichen.....10
	String.....9ff., 24, 27f., 31, 50
	Superklasse.....34, 36ff.
	Syntax.....5, 53

T	
	Textdatei.....43ff.
U	
	Ubuntu.....7, 50
	Umgangssprache.....4f.
V	
	variable.....9f., 13ff., 21, 24, 27ff., 32ff., 42, 45, 48, 50
	Verzeichnis.....7f., 41
	Verzweigung.....17f.
W	
	Windows.....6ff., 53
Z	
	Zufallszahl.....22
	Zwischensprache.....5