

Proseminar Programmiersprachen

Ruby

Markus Pölloth

Betreuer: Florian Haftmann

TU München

24. Januar 2007

Entstehung von Ruby

- ▶ Yukihiro „Matz“ Matsumoto beginnt 1993 mit der Entwicklung von Ruby
- ▶ 1995 ist die erste Version von Ruby fertig
- ▶ Im Dezember 1996 wurde die Version 1.0 veröffentlicht
- ▶ Seit August 2006 ist die Version 1.8.5 erhältlich
- ▶ Seit 2000 gibt es die Dokumentation von Ruby auch auf Englisch, seit 2002 auch auf Deutsch.

Motivation der Entwicklung

Matz entwickelte Ruby, weil er schon als Student von einer „idealen Sprache“ träumte. Unter einer idealen Sprache versteht er:

- ▶ Einfach erlernbar und anwendbar
- ▶ Abdeckung eines breiten Anwendungsfeldes
- ▶ Mächtiger als Perl
- ▶ Objektorientierter als Python

Entscheidend im Entwicklungsprozess war, dass Matz viel von Skriptsprachen hält und Fan der vollständigen Objektorientierung ist.

Das Ziel von Ruby

Ein wichtiges Ziel der Entwicklung war, dass dem Programmierer die Arbeit leicht gemacht wird. Dies soll erreicht werden durch:

- ▶ Integration der Vorteile von Smalltalk, Perl und Python
- ▶ Eine einfache Syntax
- ▶ Einhalten des „*principle of least surprise*“
- ▶ Geringer Verbrauch von „*brain power*“

Ein weiteres wichtiges Ziel war, dass programmieren Spaß machen soll.

Ruby ist eine vollständig objektorientierte Sprache.

- ▶ Variablen und sogar Zahlen sind Objekte

Ruby kann aber zugleich als Skriptsprache eingeordnet werden.

- ▶ Z. B. einfacher Zugriff auf Programme des Betriebssystems

Anwendung von Ruby

- ▶ Eher selten als Skriptsprache eingesetzt
- ▶ Systemverwaltung
- ▶ Mehrschicht-Prozessor-Systeme
- ▶ KI-Programmierung
- ▶ Ruby on Rails Framework
- ▶ Linux Distribution Rubyx

Ruby wird sehr vielseitig eingesetzt.

- ▶ In Ruby werden kaum Sonderzeichen verwendet
- ▶ Das Zeilenende muss nicht mit ';' o. ä. markiert werden
- ▶ Es gibt meist mehrere Schreibweisen für identische Ausdrücke
- ▶ Ruby hat eine klare Namenskonvention
 - ▶ Lokale Variablen, Methodennamen und Parameter beginnen mit Kleinbuchstaben
 - ▶ Konstanten, Klassennamen und Modulnamen beginnen mit einem Großbuchstaben
 - ▶ Globalen Variablen wird ein '\$' vorangestellt, Instanzvariablen ein '@' und Klassenvariablen ein '@@'

Klassendefinition in Ruby:

```
1 class Person
2   def initialize(name) # Konstruktormethode
3     @name = name
4   end
5 end
```

Erstellen von Subklassen ist auch nicht aufwändig:

```
1 class Student < Person
2   def initialize(name, fach)
3     super(name)
4     @fach = fach
5   end
6 end
```

Klassen, Objekte und Variablen - Erweiterung von Klassen

Ein neues Objekt einer Klasse erzeugt man mit dem Operator „new“:

```
1 hans = Student.new("Hans", "Informatik")
```

Dieses Objekt ist noch ziemlich sinnlos, da es keine Methoden besitzt. In Ruby kann man aber eine Klassendefinition an einer beliebigen Stelle im Code erweitern:

```
1 class Student
2   def fach
3     puts @fach
4   end
5 end
6
7 hans.fach
8   # => Informatik
```

Erweitert man eine Klasse, stehen die Methoden auch bereits erzeugten Objekten zur Verfügung.

Klassen, Objekte und Variablen

In Ruby ist alles ein Objekt. Eine Zahl ist ein Objekt der Klasse `Fixnum`, deren Methoden auf jede Zahl anwendbar sind:

```
1 -3.class
2   # => Fixnum
3
4 Fixnum.instance_methods
5   # => ["to_s", "abs", ">", "nil?", "instance_of?", ...]
6
7 -3.abs
8   # => 3
```

Klassen, Objekte und Variablen

Ruby typisiert Variablen dynamisch zur Laufzeit.
Eine Variable kann ihren Typ ändern, da sie nur eine Referenz auf ein Objekt darstellt. (Duck Typing)

```
1 a = 3
2 a.class
3     # => Fixnum
4 a = [1, 2]
5 a.class
6     # => Array
```

Standardtypen

- ▶ Fixnum (−262 bis 261)
- ▶ Bignum
- ▶ Float
- ▶ String
- ▶ Array
- ▶ Hash
- ▶ Regulärer Ausdruck
- ▶ Wertebereich (1..4 oder "ac"..."ag")

- ▶ Ein Block hat in Ruby nicht nur den Sinn Code zu gruppieren.
- ▶ Grenzt ein Block an einen Methodenaufruf, kann innerhalb der Methode mit *yield* der Block aufgerufen werden und ihm optional ein Parameter übergeben werden.
- ▶ Endet der Block, wird die Ausführung der Methode fortgesetzt.
- ▶ Ein Block ist eine Art Coroutine zu einer Funktion.

Blöcke und Iteratoren

Ein Beispiel zur Verwendung von Blöcken ist die Berechnung der Fibonaccifolge und Ausgabe:

```
1 def fibanocci(max)
2   i1, i2 = 1, 1      #parallele Zuweisung
3   while i1 <= max
4     yield i1
5     i1, i2 = i2, i1+i2
6   end
7 end
8
9 fibanocci(500) { |f| print f, " " }
10    # => 1 1 2 3 5 8 13 21 34 55 89 144 233 377
11
12 array = []
13 fibanocci(100) { |f| array << f }
14    # => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Blöcke und Iteratoren

- ▶ Ein Block kann auch durch einen Iterator aufgerufen werden.
- ▶ Iteratoren sind für bestimmte Klassen definierte Methoden, die nacheinander die Werte oder Unterobjekte des Objekts zurückliefern.
- ▶ Iteratoren gibt es z. B. für Arrays, Wertebereiche, IO-Objekte, ...
- ▶ Beispiel zur Verwendung von Iteratoren:

```
1 sum = 0
2 (1..4).each { |x| sum += x }
3 sum
4      # => 10
```

Methoden

Da Klassen an beliebiger Stelle im Code erweiterbar bzw. auch überschreibbar sind, kann man auch Standardmethoden überschreiben:

```
1 class Fixnum
2   alias oldplus +
3   def +(wert)
4     oldplus(wert).succ
5   end
6 end
7
8 1 + 2
9     # => 4
```

Rechnet man z. B. mit Komplexen Zahlen, könnte man die Klasse dafür von Fixnum ableiten und die entsprechenden Methoden überschreiben.

Da in Ruby fast alles ein Ausdruck ist, ergibt sich die Möglichkeit Ausdrücke aneinander zu ketten:

```
1 a = b = c = 0
2
3 [3, 1, 7, 0].sort.reverse.first.succ.downto(0) do |x|
4     print x.to_s + " "
5     end
6     # => 8 7 6 5 4 3 2 1 0
```

Wie in vielen anderen Skriptsprachen gibt es eine Befehlsweiterung. Ausdrücke, die in „`‘`“ eingeschlossen werden, gibt Ruby direkt an das Betriebssystem weiter:

```
1 'date '  
2      # => "Mi 10 Jan 20:45:23 CET 2007"
```

Ausdrücke - parallele Zuweisung

Eine interessante Eigenschaft ist auch die parallele Zuweisung:

```
1 a, b = 1, 2
2     # => a = 1, b = 2
3
4 a, b = b, a
5     # => a = 2, b = 1
```

Man benötigt keine Hilfsvariable um den Inhalt zweier Variablen zu tauschen.

Parallele Zuweisung mit „zu wenig“ bzw. „zu vielen“ Werten:

```
1 a, b, c = 1, 2
2     # => a = 1, b = 2, c = nil
3 d, e = 3, 4, 5
4     # => d = 3, e = 4
```

Ruby erlaubt es Funktionen zu schreiben, denen beliebig viele Parameter übergeben werden können:

```
1 def begruesser(gruss, *namen)
2   namen.each { |name|
3     puts gruss + " " + name
4   }
5 end
6
7 begruesser("Hallo", "Hans", "Gretel")
8     # => Hallo Hans
9     # => Hallo Gretel
```

Der `*` vor dem Parameternamen weist an, dass alle folgenden Parameter in das Array *namen* zusammengefasst werden sollen.

Boolesche Ausdrücke

- ▶ Ruby kennt:
 - ▶ *nil* (nicht definiert)
 - ▶ *false*
 - ▶ *true*
- ▶ Die Zahl Null und eine leere Zeichenkette werden nicht als *false* interpretiert
- ▶ Reguläre Ausdrücke sind als Boolesche Ausdrücke zulässig
- ▶ Alle gängigen Booleschen Operatoren werden unterstützt, zusätzlich noch *defined?*

```
1 2 > 3
2      # => false
3
4 defined? abc
5      # => nil
6 defined? printf
7      # => "method"
```

if- und unless-Ausdrücke

Der *if*-Ausdruck unterscheidet sich nicht wesentlich von dem in C und Java.

```
1 a = 3  # (a war vorher ein Array)
2 if a < 0 then "kleiner Null"
3 elseif a == 0 then "gleich Null"
4 else "größer Null"
5 end
6      # => größer Null
```

Zusätzlich gibt es noch einen *unless*-Ausdruck, eine Negierung von *if*. Nimmt der Boolesche Ausdruck *false* an, wird die Anweisung ausgeführt.

Es gibt auch das aus C und Java bekannte *case*-Konstrukt, das sich syntaktisch nur unwesentlich unterscheidet.

Ausdrücke - Schleifen

Wie C und Java unterstützt Ruby *while*- und *until*-Schleifen.

```
1 a *= 2 while a < 100
```

Die klassische *for*-Schleife wird allerdings nicht unterstützt, sie kann durch Iteratoren leicht ersetzt werden:

```
1 3.times do
2   print "Schleife! "
3 end
4      # => Schleife! Schleife! Schleife!
```

Hier ist offensichtlicher, wie viele Durchläufe die Schleife macht. Weitere, für Schleifen wichtige Iteratoren sind *downto*, *upto* und *step*, sie durchlaufen Wertebereiche.

Ein elegantes Konstrukt ist *for...in*:

```
1 c = [1, 2, 3, 4]
2 for x in c
3   puts x
4 end
```

Ausdrücke - Schleifen

Der einfachste Iterator ist *loop*, der eine Endlosschleife erstellt.

Zur Steuerung von Schleifen gibt es die Anweisungen:

- ▶ *break* (Abbruch der Schleife)
- ▶ *redo* (Wiederholen des aktuellen Durchlaufs)
- ▶ *next* (Sofortiger Beginn des nächsten Durchlaufs)
- ▶ *retry* (Startet die Schleife neu)

```
1 def schreibeDatei(file)
2   datei = File.open(file, "w+")
3   print "Bitte den Text eingeben. Ende mit EOF\n"
4   loop {
5     eingabe = gets
6     break if eingabe.chomp =~ /EOF/
7     datei.puts eingabe
8   }
9   datei.close
10 end
```

Exceptions

- ▶ Keine Verwendung von klassischen Fehlercodes
- ▶ Stattdessen Einsatz von Exceptions wie in Java
- ▶ Eine Exception ist ein Objekt, das Informationen zu dem Fehler speichert
- ▶ Exception-Objekte werden an den aufrufenden Stack zurückgereicht bis Code gefunden wird, der für die Behandlung zuständig ist
- ▶ Eigene Exceptionklassen können auch erstellt werden

Exceptions

Ruby hat eine feste Hierarchie von Exception-Klassen:

Exception

 fatal

 Interrupt

 NoMemoryError

 ScriptError

 NotImplementedError

 SyntaxError

 StandardError

 ArgumentError

 IOError

 TypeError

 ZeroDivisionError

Exceptions

In Blöcken kann mit dem Schlüsselwort *rescue* definiert werden, wie bei bestimmten Exceptions verfahren werden soll:

```
1 f = File.open("eineDatei", "w")
2 begin
3   f.each { |line| puts line }
4   rescue StandardError
5     print "Fehler bei Dateiverarbeitung: " + \#!
6   ensure f.close unless f.nil?
7 end
8
9 # Bei der Ausführung ergibt sich:
10   # => Fehler bei Dateiverarbeitung:
11   #   not opened for reading
12 f # Zur Kontrolle, ob die Datei geschlossen wurde:
13   # => #<File:eineDatei (closed)>
```

ensure stellt sicher, dass die Anweisungen, auch wenn Fehler auftreten, ausgeführt werden.

Exceptions

Manchmal ist es sinnvoll, einen erneuten Versuch zu starten:

```
1 @esmtplib = true
2 begin
3   if @esmtplib then
4     @command.ehlo(helodom)
5   else
6     @command.helo(helodom)
7   end
8   rescue ProtocolError
9     if @esmtplib then
10      @esmtplib = false
11      retry
12    else
13      raise
14    end
15 end
```

Dieses Beispiel ist aus der Minero Aokis Bibliothek, die Unterstützung für das SMTP-Protokoll bietet.

Sofortige Fehlerbehandlung mit *catch (:name)* und *throw :name*

- ▶ Klassen und Konstanten können in Module zusammengefasst werden *module Name ... end*
- ▶ Import von Modulen mit *require 'name'*
- ▶ Module bieten eigene Namensräume *ModuleName. ...*
- ▶ Zugriff auf Modulklassen bzw. Konstanten:
 - ▶ *ModuleName.methode*
 - ▶ *ModuleName::Konstante*

- ▶ Module lassen sich auch als Mixins benutzen
- ▶ Import in einer Klassendefinition mit *include ModulName*
- ▶ Allen Objekten stehen die Methoden des Moduls zur Verfügung, im Prinzip sind alle Klassen des Moduls Oberklassen dieser Klasse (Ersatz für Mehrfachvererbung)
- ▶ *include* nicht mit dem *#include* von C vergleichbar, da die Methoden nur referenziert werden (nachträgliche Änderungen der Module wirken sich auf die Klassen aus)

- ▶ Threads
 - ▶ Laufen nur innerhalb des Ruby-Interpreters ab
 - ▶ Vorteil: Nicht vom Betriebssystem abhängig
 - ▶ Nachteil: Kein nativer Schutz vor Deadlocks oder Verhungern
 - ▶ Sehr einfache Möglichkeit Code parallel auszuführen
- ▶ Prozesse
 - ▶ Werden vom Betriebssystem verwaltet
 - ▶ Ermöglicht es Prozesse aufzurufen, die nicht in Ruby geschrieben wurden
 - ▶ Vorteil: Bei Absturz ist nur ein Teilprozess betroffen
 - ▶ Nachteil: Verschiedene Kommandos für *fork*, etc.

Ruby im Vergleich zu Java

- ▶ Ruby ist Nischensprache. Java Entwickler benutzen eher Python
- ▶ Java ist streng typisiert, Ruby bietet dynamische Polymorphie
- ▶ Java hat eine standardisierte Laufzeitumgebung, der Code ist zu 100% portabel
- ▶ Java ist vorkompiliert und dadurch schneller
- ▶ In Java gibt es keine Möglichkeit Mehrfachvererbung umzusetzen

Ruby im Vergleich zu C

- ▶ C ist kompiliert und dadurch wesentlich schneller
- ▶ C ist streng typisiert
- ▶ C ist rein prozedural, Ruby streng objektorientiert
- ▶ C ermöglicht hardwarenahe Programmierung, Ruby bietet das nur durch C-Module
- ▶ C ist maschinennaher, man denkt meist aus der Sicht der Maschine, in Ruby steht das zu lösende Problem im Mittelpunkt

Ruby im Vergleich zu Python

- ▶ Gemeinsamkeiten:
 - ▶ Beide haben einen interaktiven Interpreter
 - ▶ Objekte sind dynamisch typisiert
 - ▶ Konzept der Exceptions ist gleich
- ▶ Unterschiede:
 - ▶ In Ruby gibt es Konstanten und Strings sind veränderbar
 - ▶ Ruby hat nur einen Listentyp (Array), der veränderbar ist
 - ▶ In Ruby kann man auf Objektattribute nicht direkt zugreifen
 - ▶ Ruby verwendet Mixins, anstatt Mehrfachvererbung
 - ▶ Klassen sind jederzeit erweiterbar und modifizierbar

Nachteile von Ruby

- ▶ Da Ruby interpretiert ist, hat es eine langsame Ausführungsgeschwindigkeit
- ▶ Ruby unterstützt kein Unicode und UTF-8 nur zum Teil
- ▶ Manche Fehler sind nur schwer zu finden, da der Interpreter beim Laden des Skripts nur wenig Überprüfungen durchführt
- ▶ Es gibt auch ein paar Inkonsistenzen und zu komplexe Sachverhalte:
 - ▶ Das Modul `Math` verletzt das Paradigma der Objektorientierung. Man würde erwarten, dass `2.Math.sqrt` $\sqrt{2}$ berechnet. Die Methode muss allerdings mit `Math.sqrt(2)` aufgerufen werden.
 - ▶ Die Sichtbarkeit und Gültigkeit von Variablen ist nicht ganz einfach zu verstehen: Eigentlich sollten Variablen, die in einem Block definiert wurden nur dort gültig sein. Das ist aber nicht umgesetzt.
 - ▶ Merkwürdigerweise kann man eine öffentliche Methode mit einer privaten überschreiben, sodass die Klasse danach ein anderes öffentliches Interface besitzt.

Vorteile von Ruby

- ▶ Ruby wird aktiv weiterentwickelt.
- ▶ Ruby hat einen sehr hohen Abstraktionsgrad.
- ▶ Durch die Ähnlichkeit zu anderen Sprachen ist Ruby leicht erlernbar.
- ▶ Der Code ist durch Verzicht auf viele Sonderzeichen und aufgrund sinnvoller Konventionen gut lesbar.
- ▶ Ruby ist sehr gut dokumentiert. Das Standard-Werk zu Ruby gibt es auch als deutsche Übersetzung, die die meisten Bibliotheken beschreibt.
- ▶ Ruby ist weitgehend Plattformunabhängig. Der Interpreter ist für Unix, Linux, Windows, MacOS X und DOS erhältlich.
- ▶ Ruby bietet einen sehr großen Sprachumfang. Es gibt Bibliotheken für gängige Kommunikationsprotokolle (http, ftp, imap, smtp, ...), Datenbanken, Open SSL und gängige Grafiksysteme (für GUIs).
- ▶ Mit Ruby lässt es sich sehr einfach programmieren und die Sprache hinterlässt einen sehr flexiblen Eindruck.

- ▶ David Thomas, Andrew Hunt:
Programming Ruby, The Pragmatic Programmer's Guide
Addison-Wesley
Ebook: <http://www.rubycentral.com/book/>
- ▶ David Thomas, Andrew Hunt:
Programmieren mit Ruby
(deutsche Übersetzung von: Programming Ruby)
Addison-Wesley
Ebook: <http://www.homevrweb.de/~juergen.katins/ruby/buch/>
- ▶ **Offizielle Ruby Homepage:** <http://www.ruby-lang.org>
- ▶ **Offizielle Ruby Dokumentation:** <http://www.ruby-doc.org>
- ▶ **Deutsches Wiki zu Ruby:** <http://www.rubywiki.de>

Fragen?

Gibt es noch Fragen???