

Proseminar Programmiersprachen

Ruby

Markus Pölloth
Betreuer: Florian Haftmann

24. Januar 2007

TU München, Wintersemester 2006/07
Kursleiter: Prof. Tobias Nipkow, Dr. Stefan Berghofer

Inhaltsverzeichnis

1	Einführung	3
1.1	Entstehung von Ruby	3
1.2	Die Ziele von Ruby	3
1.3	Was für eine Sprache ist Ruby?	3
2	Überblick über Ruby	4
2.1	Grundlagen	4
2.2	Klassen, Objekte und Variablen	4
2.3	Blöcke und Iteratoren	5
2.4	Methoden	6
2.5	Ausdrücke	7
2.6	Exceptions	9
2.7	Module und Mixins	11
2.8	Threads und Prozesse	11
2.8.1	Threads	11
2.8.2	Prozesse	13
3	Fazit	14
3.1	kurzer Vergleich mit anderen Sprachen	14
3.1.1	Ruby und Java	14
3.1.2	Ruby und C	14
3.1.3	Ruby und Python	14
3.2	Nachteile	14
3.3	Vorteile	15

1 Einführung

1.1 Entstehung von Ruby

Der Japaner Yukihiro „Matz“ Matsumoto begann 1993 mit der Entwicklung von Ruby. Seine Motivation für die Entwicklung war, dass er schon als Student von einer idealen Programmiersprache träumte. Unter einer idealen Sprache versteht er, dass sie einfach zu erlernen und zu verwenden ist. Außerdem soll sie ein breites Anwendungsgebiet abdecken. Seine ideale Sprache soll mächtiger als Perl und objektorientierter als Python sein. Entscheidend im Entstehungsprozess war, dass er viel von Skriptsprachen hält und Fan der objektorientierten Programmierung ist.

Im Jahr 1995 veröffentlichte er eine erste Version von Ruby, die Version 1.0 im Dezember 1996. Inzwischen ist die aktuelle Version 1.8.5, die seit August 2006 verfügbar ist.

Ruby wird unter der GNU General Public License (GPL) verbreitet und unter einer eigenen Lizenz, die weniger einschränkt als die GPL (Änderungen sind zulässig, sofern sie öffentlich zugänglich gemacht werden).

Anfangs war Ruby ausschließlich in Japan verbreitet, da die Dokumentation (sofern überhaupt vorhanden) nur auf japanisch existierte. Erst 2000 wurde das Buch *Programming Ruby [1]* veröffentlicht, das auch heute noch in der 2. Auflage das Standard-Werk ist. Es ist auch auf der offiziellen Ruby Homepage als Ebook verfügbar. Inzwischen ist Ruby in der ganzen Welt verbreitet, unter anderem auch durch die Entwicklung des Ruby on Rails Frameworks. In Japan hat Ruby aber immer noch den höchsten Verbreitungsgrad, wo es inzwischen auch Python überholt haben soll.

1.2 Die Ziele von Ruby

Ruby wurde unter dem Ziel entwickelt, dem Programmierer die Arbeit einfach zu machen. Deshalb wurde es so konzipiert, dass es die Vorteile von Smalltalk (vollständig objektorientiert), Perl und Python (Flexibilität) und einigen anderen Sprachen bietet. Es wurde auch viel Wert darauf gelegt, dass die Sprache eine einfache Syntax hat. Dies wird dadurch erreicht, dass keine allzu ungewohnten Features in die Sprache integriert sind und dass die Ähnlichkeit von Syntax und Bezeichnern zu anderen Sprachen, von denen Ruby abstammt, relativ hoch ist.

Ein zentraler Begriff ist auch das „*principle of least surprise*“, damit das Verhalten möglichst dem Erwarten des Entwicklers entspricht. Erreichen kann man dies unter anderem indem man die Sprache frei von Inkonsistenzen und Fehlern hält.

Matsumoto sagte auf dem *Lightweight Languages Workshop 2002* [3], dass die „*brain power*“, die der Programmierer braucht möglichst gering sein soll. Erreicht wird das in Ruby, indem man versucht dem Programmierer Stress (in Form großer Probleme bei der Entwicklung) zu ersparen. Der Hintergedanke davon ist, dass die Entwicklung von Programmen schneller voranschreitet und der Quellcode qualitativ besser wird. Ein weiteres Ziel, das Matsumoto mit Ruby erreichen wollte, ist dass programmieren Spaß machen soll.

1.3 Was für eine Sprache ist Ruby?

Ruby ist vollständig objektorientiert, da unter anderem auch die elementaren Datentypen (im Gegensatz zu Java) und reguläre Ausdrücke ebenfalls Objekte sind. Zusätzlich weist Ruby auch einige Merkmale typischer Skriptsprachen auf, unter anderem die Möglichkeit einfach auf die Programme des zugrundeliegenden Betriebssystems zugreifen zu können.

Allerdings wird Ruby nur sehr selten als Skriptsprache eingesetzt, denn es lassen sich auch einfache Programme wie GUI-Anwendungen und Mehrschicht-Server-Prozesse schreiben. Auch Programme zur Serververwaltung oder zum Datenbankmanagement sind nicht schwer zu realisieren. Ruby wird sogar von Entwicklern benutzt um künstliche Intelligenz zu programmieren und zur Untersuchung der natürlichen Evolution. Eine weitere große Anwendung ist *Ruby on Rails*, das ein klassisches *Model-View-Controller-Framework* ist. Damit erreicht man eine Trennung der Programmlogik von der GUI-Steuerung und der reinen Darstellung. Es wird vor allem im Web-Bereich verwendet und stellt eine Alternative zu PHP dar.

2 Überblick über Ruby

2.1 Grundlagen

Rubys Syntax ist darauf ausgelegt, möglichst kurz und prägnant zu sein. Sonderzeichen werden im Gegensatz zu Java kaum verwendet. Ruby interpretiert Zeilennumbrüche als Zeilenende, ein Semikolon o.ä. ist nicht erforderlich. Bei einstelligen Funktionen können die normalen Klammern weggelassen werden. Blöcke und Rümpfe werden durch `{ }` gekennzeichnet, können aber auch durch *begin* und *end* ersetzt werden. In Ruby gibt es in der Regel mehrere Möglichkeiten etwas ausdrücken.

Um die Verwendungweise eines Namens leichter zu ermitteln, verwendet Ruby folgende Konvention: Lokale Variablen, Methodennamen und Parameter sollten immer mit einem Kleinbuchstaben beginnen. Globalen Variablen stellt man ein `$` voran, Instanzvariablen werden mit einem `@` gekennzeichnet. Klassenvariablen beginnen mit `@@`. Klassennamen, Modulnamen und Konstanten beginnen mit einem Großbuchstaben.

2.2 Klassen, Objekte und Variablen

Klassen werden in Ruby wie folgt definiert:

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

Konstruktor einer Klasse ist immer die initialize Methode. Ruby unterstützt Vererbung, die Klasse `Student` erbt die Methoden von `Person`:

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

Mit `hans = Student.new(„Hans“, „Informatik“)` erzeugt man ein neues Objekt der Klasse `Student`. Allerdings sind darauf noch keine weiteren Methoden definiert. Man kann aber an beliebiger Stelle im Quelltext eine Klasse erweitern.

```
1 class Student
2   def name
3     puts @name
4   end
5 end
```

erweitert die Klasse `Student` um die Methode `name`, allen bestehenden Objekten von `Student`, steht nun auch die Methode `Name` zur Verfügung.

```
1 hans.name
2     # => Hans
```

Mehrfachvererbung wird von Ruby nicht unterstützt. Eine Rubyklasse kann aber die Funktionalität beliebig vieler Mixins enthalten. Ein Mixin (2.7) ist so etwas wie eine partielle Klassendefinition.

In Ruby ist alles ein Objekt. Eine Zahl z. B. ist ein Objekt der Klasse `Fixnum`, die die wichtigsten Methoden zum Umgang mit Zahlen zur Verfügung stellt.

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

Eine Variable ist in Ruby allerdings kein direktes Objekt. Jede Variable ist eine Referenzen auf ein Objekt, das sich im großen Pool (meist der Heap) befindet. Damit hat man die Möglichkeit mehrere Variablen auf das selbe Objekt zu referenzieren. Variablen in Ruby sind *dynamisch typisiert*, der Typ der Variable muss nicht vorher deklariert werden, sondern der Wert einer Variablen bestimmt den Typ. Ruby unterstützt das *Duck Typing*, es ist egal was für ein Objekt übergeben wird, solange das Objekt die benötigten Methoden unterstützt. Die Grundtypen in Ruby sind Zahlen, Zeichenketten, Arrays, Hashes, Wertebereiche, Symbole und reguläre Ausdrücke. Wertebereiche (Range-Objekte) werden mit dem `..` Operator erzeugt. `„aa“..„ac“` erzeugt die Folge `„aa“`, `„ab“`, `„ac“`. Braucht man eine Variable oder ein Objekt nicht mehr, muss man sich nicht um die Freigabe des Speichers kümmern, das tut der *Garbage Collector*. Dadurch werden auch viele Fehler vermieden.

2.3 Blöcke und Iteratoren

Ein Block in Ruby hat nicht wie in C++ oder Java nur den Sinn Anweisungen zu gruppieren. Ein Block kann in Ruby nur an einen Methodenaufruf angrenzen. Er muss in der selben Zeile beginnen, in der das letzte Argument der Methode steht. Der Block wird nicht ausgeführt, wenn man auf ihn trifft, sondern Ruby merkt sich den Kontext in dem der Block steht und beginnt mit der Verarbeitung der Methode. Innerhalb der Methode kann mit `yield` der Code im Block aufgerufen werden, als wäre der Block ebenfalls eine Methode. Sobald der Block endet, wird die Methode fortgesetzt. Einem Block können auch Parameter übergeben werden. Das nächste Beispiel zeigt eine Anwendung von Blöcken zur Berechnung von Elementen der Fibonacci-Folge bis zu einem übergebenen Maximalwert.

```
1 def fibonacci(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 fibonacci(1000) { |f| print f, " " }
10     # => 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

```

11 sum = 0
12 fibonacci(1000) { |f| sum += f }
13     # => 2583

```

Der Parameter der `yield` Anweisung wird an den Block übergeben und der Code ausgeführt. Ein Vorteil der Verwendung von Blöcken ist, dass die selbe Methode aufrufen kann, aber die Rückgabewerte der Methode je nach Aufrufkontext unterschiedlich behandeln kann. Eine weitere Anwendung der Fibonacci Zahlen wäre hier, sie in ein Array zu speichern: `fibonacci(1000) { |f| array << f }` Blöcke können außerdem auch Werte an die aufrufende Methode zurückgeben.

Ein Iterator in Ruby ist eine Methode, die sobald sie einen neuen Wert generiert hat `yield` aufruft und damit einen Codeblock. Es muss keine Helferklasse generiert werden, die den Zustand des Iterators enthält, wie z.B. in Java oder C++. Einige Iteratoren gibt es für viele Ruby-Sammlungen (Arrays, Range-Objekte, IO-Objekte), z.B. `each`, der über jedes Element bzw. jede Zeile, ... hinweggeht.

```

1 (1..4).each { |x| puts x }
2     # => 1
3     # => 2
4     # => 3
5     # => 4

```

Mit Blöcken kann man ein Codestück definieren, das unter irgendeiner Art von Transaktionssteuerung ausgeführt werden muss. Beispielsweise möchte man eine Datei öffnen, mit ihrem Inhalt etwas machen und danach sicherstellen, dass die Datei wieder geschlossen wird. Mit Blöcken lässt sich das einfach realisieren. Man schreibt eine Funktion, die die Datei öffnet, sie dann mit Aufruf von `yield` an einen Block übergibt und nachdem der Block beendet ist, die Datei wieder schließt.

Blöcke können auch Abgeschlossene Einheiten sein. Beginnt der letzte Parameter einer Methodendefinition mit einem `&`, dann sucht Ruby bei jedem Aufruf der Methode nach dem zugehörigen Block. Dieser Block wird dann in ein Objekt der Klasse `Proc` umgewandelt und dem Parameter zugewiesen. Mit diesem `Proc`-Objekt ist dann der gesamte Kontext des Blocks verbunden, der Wert von `self` sowie die Methoden und Variablen im Gültigkeitsbereich. Der Block kann die den Kontext in dem er definiert wurde also noch nutzen, wenn die Umgebung ansonsten verschwunden ist.

2.4 Methoden

Da in Ruby Klassen an beliebiger Stelle erweiterbar, bzw. modifizierbar 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

```

Dies ermöglicht es auch Standardmethoden von selbsterstellten Objekten neu zu definieren, damit die Anwendung der Methoden sinnvoller ist. Z. B. ließe sich eine Klasse für komplexe Zahlen aus der Klasse `Fixnum` ableiten, bei der dann aber noch die gängigen arithmetischen Operationen angepasst werden müssten.

In Ruby kann in den meisten Fällen auf den **Return**-Ausdruck verzichtet werden, da eine Methode oder ein Ausdruck immer den Wert des zuletzt ausgewerteten Ausdrucks zurückgibt.

2.5 Ausdrücke

Einer der auffälligsten Unterschiede zwischen Ruby und anderen Programmiersprachen ist, dass in Ruby fast alles ein Ausdruck ist, wenn es einen sinnvollen Wert zurückliefert. Als Folge davon ergibt sich die Möglichkeit, Anweisungen aneinander zu ketten.

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

Die meisten Operatoren in Ruby sind als Folge der vollständigen Objektorientierung eigentlich Methodenaufrufe. Der Ausdruck `a*b+c` bedeutet eigentlich, dass das von `a` referenzierte Objekt die Methode `*` mit dem Parameter `b` ausführen soll. Mit dem zurückgegebenen Objekt wird nun die Methode `+` mit Parameter `c` aufgerufen. Folgende Schreibweise ist gleichbedeutend: `(a.*(b)).+(c)`

Ruby unterstützt wie viele Skriptsprachen auch eine Befehlsweiterung, die Ausdrücke, die in `'` eingeschlossen sind, direkt ans Betriebssystem weiterleitet. Der Wert des Ausdrucks ist die Standardausgabe des Befehls.

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

Eine interessante Eigenschaft von Ruby ist die parallele Zuweisung. Sie ermöglicht es z. B. die Werte zweier Variablen ohne eine Hilfsvariable zu vertauschen.

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

Anweisungen in Ruby werden effizient parallel ausgeführt. Zuerst werden die Werte auf der rechten Seite in der Reihenfolge, in der sie dastehen, ausgewertet und erst danach werden die Werte der linken Seite zugewiesen.

Hat eine Zuweisungsoperation rechts mehr Werte als links, so werden die `r`Werte als Array zusammengefasst. Gibt es mehr `l`Werte als `r`Werte, werden die überschüssigen `l`Werte auf `nil` gesetzt. Mit dem Zuweisungsoperator von Ruby können Arrays auch auf- und zugeklappt werden. Steht vor dem letzten `l`Wert ein Sternchen `*`, werden alle restlichen `r`Werte als Array gesammelt. Ist der letzte `r`Wert ein Array mit vorangestelltem Sternchen `*`, wird das Array expandiert und die Werte den `l`Werten zugewiesen, sind mehr `r`Werte verfügbar als `l`Werte, werden die überzähligen `r`Werte ignoriert. Gibt es mehrere `l`Werte, aber nur einen `r`Wert, der ein Array ist, wird das Array automatisch expandiert.

```
1 a = [1, 2, 3, 4]
2 head, *tail = a
3 # => head = 1, tail = [2, 3, 4]
4 first, second = 99, *a
5 # => first = 99, second = 1
```

Boolesche Ausdrücke nehmen in Ruby den Wert `true` an, sofern sie nicht `false` oder `nil` (nicht definiert) sind. Im Unterschied zu C++ wird die Zahl Null und Zeichenketten der Länge null nicht als `false`-Wert interpretiert. Ruby unterstützt alle

gängigen Booleschen Standardoperatoren und bietet zusätzlich noch den Operator `defined?`. `defined?` gibt `nil` zurück, falls der übergebene Ausdruck nicht definiert ist, ansonsten gibt es eine Beschreibung des Arguments zurück.

```
1 defined? abc
2     # => nil
3 defined? printf
4     # => "method"
```

Außerdem kann auch ein regulärer Ausdruck als Boolescher Operator verwendet werden.

Ruby kennt natürlich auf `if`-Ausdrücke, die sich nicht wesentlich von anderen gängigen Programmiersprachen unterscheiden.

```
1 if a < 0 then "kleiner Null"
2 elsif a == 0 then "gleich Null"
3 else "größer Null"
4 end
```

Ruby verfügt zusätzlich auch über eine negierte Form des `if`-Ausdrucks: `unless`. Die Anweisungen werden nur dann ausgeführt, wenn der Boolesche Ausdruck falsch annimmt. Auch die aus C bekannte Kurzform des bedingten Ausdrucks gibt es in Ruby: `a < 0 ? „kleiner Null“ : „größer oder gleich Null“`. Eine nützliche Eigenschaft, die Ruby mit Perl gemeinsam hat sind die `if`- und `unless`-Modifikatoren, sie werden an das Ende einer normalen Anweisung angehängt und sind ausschlaggebend dafür, ob die Anweisung ausgeführt wird.

```
1 a = a.abs if a < 0
```

Da `if` selbst wieder ein Ausdruck ist, kann auch an einen `if`-Ausdruck ein `if`- oder `unless`-Modifikator angehängt werden:

```
1 if defined? a
2     b = a.abs
3 end unless defined? b
```

Würde man das ganze als Block formulieren, könnte man erneut einen Modifikator anhängen. Allerdings ist die Verwendung nicht sonderlich sinnvoll, da es zu äußerst unübersichtlichem Code führt. Alternativ könnte man beide Bedingungen im Booleschen Ausdruck des `if`-Ausdrucks prüfen.

Auch `Case`-Ausdrücke werden von Ruby unterstützt. Die Syntax ist im wesentlichen identisch mit gängigen Programmiersprachen. Eine kleine Besonderheit ist es, dass der Vergleichsoperator modifiziert werden kann indem die entsprechende Methode in der Klassendefinition des Vergleichsobjekts überschrieben wird. Reguläre Ausdrücke sind standardmäßig zulässig.

Ruby unterstützt wie C, Java und viele andere Programmiersprachen `while`- und `until`-Schleifen. Zu diesen beiden Schleifenarten gibt es wie bei `if`-Ausdrücken ebenfalls Modifikatoren.

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

Allerdings gibt es in Ruby keine `for`-Schleife der Art wie es sie in C oder Java gibt. Stattdessen verwendet Ruby Iteratormethoden, die in den verschiedenen integrierten Klassen definiert sind. Sie bieten die gleiche Funktionalität, sind jedoch weniger fehleranfällig:

```
1 3.times do
2     print Schleife!
3 end
```

Rückgabewert des Ausdrucks ist „Schleife! Schleife! Schleife!“. Ein häufiger Fehler der dadurch vermieden ist, dass eine Schleife einen Durchlauf zu viel oder zu wenig macht, das ist hier offensichtlich. Die Iteratoren `downto`, `upto` und `step` durchlaufen bestimmte Wertebereiche von Zahlen als Schleife. Die Methode `each`, die z. B. über Array iteriert, kann auch für Schleifen verwendet werden. Eleganter geht das aber mit dem `for ... in` Konstrukt:

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

Ruby wandelt dieses Konstrukt automatisch in die Variante mit `each` um. Der Unterschied besteht aber im Gültigkeitsbereich der Variablen. Rubys `while`-, `until`- und `for`-Schleifen führen keine neuen Gültigkeitsbereiche ein. Bei den Blöcken, die von Iteratoren (`each`, `loop`) verwendet werden, sind die erstellten Variablen nicht von außerhalb erreichbar.

Die einfachste Schleife in Ruby kommt durch den Iterator `loop` zustande. Der Block wird in der Schleife solange aufgerufen, bis sie abgebrochen wird.

Schleifen lassen sich in Ruby relativ gut steuern. Neben der `break` Anweisung, die die Schleife sofort abbricht, gibt es noch `redo`, das den Block nochmal wiederholt, allerdings ohne die Bedingung auszuwerten und das nächste Element zu holen, und `next`, das sofort zum Ende der Schleife springt und die nächste Iteration beginnt. Die Anweisung `retry` startet die Schleife von Anfang an neu. Beispiel für eine Schleife mit `loop`, die die Eingabezeilen in eine Datei schreibt und abbricht, sobald „EOF“ eingegeben wird.

```
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
```

2.6 Exceptions

Traditionell werden zur Behandlung von Fehlern Ergebniscodes eingesetzt. Eine Methode gibt einen speziellen Wert zurück, durch den sie mitteilt, dass sie gescheitert ist. Dieser Wert wird durch alle Schichten der aufrufenden Routinen gereicht, bis eine Methode dafür Verantwortung übernimmt. Ein Nachteil dieses Verfahrens ist, dass die Verwaltung dieser Fehlercodes mühsam ist. Treten im Verlauf des „durchreichens“ mehrere Fehler auf, ist es schwierig die Fehlercodes so zu übergeben, dass die Methode, die sie abfängt, jeden der Fehlercodes der Methode, wo er entstand, zuzuordnen.

Dieses Problem ist in Ruby, wie auch in einigen anderen Sprachen, durch Ausnahmen (Exceptions) gelöst. Die Informationen werden in ein Objekt gespeichert und an den aufrufenden Stack zurückgereicht, bis das Laufzeitsystem Code findet, der deklariert wie dieser Ausnahmetyp zu behandeln ist. Ruby hat eine feste Hierarchie von Exceptions, die auszugsweise aufgelistet ist:

```

Exception
  fatal
  Interrupt
  NoMemoryError
  ScriptError
    NotImplementedError
    SyntaxError
  StandardError
    ArgumentError
    IOError
    TypeError
    ZeroDivisionError
    SystemExit

```

Eigene Exceptionklassen können als Unterklasse von StandardError definiert werden. Mit jeder Exception ist ein Nachrichtenstring und ein Stack-Backtrace verbunden. Bei selbst definierten Exceptions kann man auch zusätzliche Informationen hinzufügen.

Bei Ausnahmenbehandlung werden wieder Blöcke eingesetzt. Fasst man kritischen Code in einem Block zusammen, kann man auch mit dem Schlüsselwort rescue und der Angabe des Fehlertyps lokal definieren, wie bei einer Ausnahme verfahren wird. Man kann auch mehrere rescue Anweisungen definieren. Fügt man das raise Statement am Ende hinzu, wird der Fehler an die nächsthöhere Ebene weitergeben, falls er lokal nicht behandelt wurde. Oft will man aber wenn ein Fehler aufgetreten ist, dass bestimmte Anweisungen des Blocks trotzdem noch ausgeführt werden. Dazu dient das ensure Statement, das nach der Fehlerbehandlung definiert wird und gewährleistet, dass der Code ausgeführt wird. Es spielt dabei keine Rolle, ob der Block normal endet oder nicht. Im folgenden Beispiel wird eine Datei verarbeitet, wobei ein Fehler auftritt. Man möchte zum einen, dass der Fehler gleich lokal behandelt wird und dass die Datei am Ende auf jeden Fall wieder geschlossen wird.

```

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? end
7
8 # Bei der Ausführung ergibt sich:
9   # => Fehler bei Dateiverarbeitung: not opened for reading
10 f # Zur Kontrolle, ob die Datei geschlossen wurde:
11   # => #<File:eineDatei (closed)>

```

Es gibt aber auch Fälle, in denen möchte man bei einem Fehler nicht abbrechen, sondern einen erneuten Versuch zu starten, bei dem die Ursache möglicherweise behoben werden kann. Angewendet wird dies z.B. bei der Minero Aokis Bibliothek von Ruby, die Funktionen zur Kommunikation mit SMTP-Server bereitstellt.

```

1 @smtp = true
2 begin
3   if @smtp then
4     @command.ehlo(helodom)
5   else

```

```

6         @command. hello (helodom)
7     end
8     rescue ProtocolError
9         if @esmpt then
10            @esmtp = false
11            retry
12        else
13            raise
14        end
15 end

```

Liegt das Problem nur daran, dass der Server den Befehl ehlo nicht unterstützt, wird ein erneuter Versuch mit hello gestartet, scheitert auch der, wird die Ausnahme an den Aufrufer der Methode weitergereicht.

Ruby bietet auch die Möglichkeit, einen Fehler sofort an einer bestimmten Stelle im Block zu behandeln und danach an der selben Stelle im Block weiterzumachen. Dazu wird mit `catch (:name)` ein Block definiert, der den Code zur Fehlerbehandlung beinhaltet. In einem anderen Block kann man dann mit `throw :name` eine Ausnahme auslösen. Dadurch wird der gesamte Aufruf-Stack aufgerollt und nach einem `catch`-Block mit selben Namen gesucht und ausgeführt. Ist der `catch`-Block beendet, wird die Ausführung im ursprünglichen Block fortgesetzt. Der `catch`-Block muss nicht innerhalb des statischen Gültigkeitsbereichs des Blocks mit der `throw`-Anweisung zu sein.

2.7 Module und Mixins

Module stellen eine Möglichkeit dar, Methoden, Klassen und Konstanten zu gruppieren. Sie bieten eigene Namensräume und tragen dazu bei, Namenskonflikte zu vermeiden und sie implementieren die Mixin-Fähigkeit.

In einem Modul können beliebig viele Klassen, Methoden und Konstanten definiert sein. Eine Moduldatei beginnt mit `modul Name` und endet mit `end`. Eingebunden wird es mit der Anweisung `require „name“`. Modulmethoden werden wie Klassenmethoden aufgerufen, indem der Methode der Modulname vorangestellt wird: `Modulname.methode`, Konstanten werden mit dem Modulnamen und `::` referenziert.

Module lassen sich auch als Mixins benutzen und ersetzen die Mehrfachvererbung. Ein eingebundenes Modul kann in einer Klassendefinition mit `include Modulname` „hineingemischt“ werden. Der Klasse stehen dann alle Instanzmethoden des Moduls zur Verfügung. Im Prinzip verhalten sich die Mixins wie Oberklassen.

Das `include` von Ruby unterscheidet sich aber wesentlich von dem `#include` in C. Das Modul muss schon vorher per `require` eingebunden sein. Außerdem werden nicht die Methoden des Mixins in die Klasse geschrieben, sondern es wird aus der Klasse heraus auf das eingefügt Modul verwiesen. Binden mehrere Klassen dasselbe Modul ein, wirkt sich eine Änderung (Erweiterung) des Moduls auf alle diese Klassen aus.

2.8 Threads und Prozesse

Ruby bietet zwei Möglichkeiten Parallelität zu realisieren: Threads und Prozesse.

2.8.1 Threads

Threads sind vollständig im Ruby Interpreter implementiert. Der Vorteil davon ist, dass das Programm portierbar ist und nicht vom Betriebssystem abhängt (Linux unterstützt z. B. derzeit keine Threads). Großer Nachteil der Lösung ist, dass

Threads verhungern oder Deadlocks hervorrufen können. So kann es passieren, dass der ganze Prozess angehalten wird. Ebenfalls ungünstig ist es, dass wenn ein Thread einen länger dauernden Aufruf an das Betriebssystem macht, alle Threads solange angehalten werden, bis der Interpreter wieder die Steuerung bekommt. Da der Ruby Interpreter nur ein Prozess ist, haben auch Mehrprozessorsysteme keine Vorteile durch den Einsatz von Threads. Trotz dessen kann man mit Threads in Ruby einfach und effizient Parallelität erreichen.

Einen Thread startet man mit `Thread.new` und übergibt einen Block, der dann von diesem Thread ausgeführt wird. Ruby stellt auch einige Methoden zur Arbeit mit Threads zur Verfügung. Mit `Thread.join` erreicht man, dass das Programm erst endet, sobald der entsprechende Thread beendet ist. `Thread.current` gewährt Zugriff auf den Thread und `Thread.list` gibt alle existenten Threads aus. Außerdem kann man jedem Thread eine Priorität zuweisen.

Einem Thread stehen alle Variablen zur Verfügung, die sich bei der Erstellung im Gültigkeitsbereich befinden. Außerdem kann jeder Thread über eigene Variablen verfügen, die nur innerhalb des jeweiligen Threads benutzt werden. Trotzdem ist es möglich auf Variablen anderer Threads zuzugreifen. Der Zugriff erfolgt ähnlich wie bei Hashes: `Thread.current[„a“]` greift auf die Variable „a“ innerhalb des aktuellen Threads zu.

Es ist auch möglich Threads in der Ausführung zu manipulieren. Die Threadmethoden `start`, `stop` und `pass` starten den Thread, halten ihn an oder löschen die aktuellen Einträge des Threads im Ruby-Scheduler.

Im Modul `Thread` gibt es eine Klasse `Mutex`, die einen Semaphor zur Verfügung stellt, mit dessen Hilfe sich immer nur ein Thread in einem kritischen Bereich befinden kann. Folgendes Beispiel zeigt die Anwendung von Threads in Verbindung mit einem `Mutex`.

```

1 def synchron()
2   require "thread"
3   mutex = Mutex.new
4
5   count1 = count2 = 0
6   difference = 0
7   counter = Thread.new do
8     loop do
9       mutex.synchronize do
10        count1 += 1
11        count2 += 1
12      end
13    end
14  end
15  spy = Thread.new do
16    loop do
17      mutex.synchronize do
18        difference += (count1 - count2).abs
19      end
20    end
21  end
22
23  sleep 1
24  mutex.lock
25  puts count1
26  puts count2
27  puts difference

```

```

28 end
29
30 synchron
31     # => 23983
32     # => 23983
33     # => 0

```

Ohne den Semaphor würde der `counter` immer wieder von `spy` unterbrochen. Es wäre aber nicht gewährleistet, dass der `counter` gerade beide `count`-Werte erhöht hat, also gäbe es einen Unterschied zwischen `count1` und `count2`.

Manchmal genügt es aber nicht nur Daten durch einen Mutex zu schützen. Z. B. befindet sich ein Thread im kritischen Bereich, muss aber auf eine weitere Ressource warten. Der Thread wird sich schlafen legen, aber es könnte passieren, dass der andere Thread, der über die Ressource verfügt, in den kritischen Bereich muss um die Ressource wieder freizugeben. Dafür gibt es in Ruby neben dem Mutex noch `ConditionVariables`. Eine `ConditionVariable` ist ein Semaphor für eine bestimmte Ressource innerhalb eines Mutex. Threads warten jetzt „auf“ der `ConditionVariable` und geben den Mutex wieder frei. Sobald die bisher blockierte Ressource frei ist, bekommt der wartende Thread ein Signal und kann fortsetzen.

2.8.2 Prozesse

Manchmal möchte man eine Aufgabe in mehrere größere Prozesse aufteilen oder einen separaten Prozess ausführen, der nicht in Ruby geschrieben wurde. Dazu stellt Ruby mehrere Möglichkeiten zur Verfügung.

Zum einen gibt es die Möglichkeit einen Teilprozess so zu erzeugen, dass die Standardeingabe und -ausgabe mit einem IO-Objekt verbunden wird. Dadurch hat man eine einfache Möglichkeit mit dem Teilprozess Nachrichten auszutauschen. Schreibt man in das IO-Objekt, kann der Prozess es auf der Standardeingabe lesen, die Standardausgabe schreibt in das IO-Objekt.

```

1 pipe = IO.popen("-", "w+")
2 pipe.puts e = 2
3 pipe.puts f = 3
4 pipe.puts f - e
5 puts pipe.gets
6     # Nach 6 mal aufrufen: (Die Rückmeldungen des Kind-Interpreters
7     # sind noch im IO-Objekt gespeichert)
8     # => 1

```

Muss man nicht weiter mit dem Prozess interagieren, bietet es sich an, ihn zu starten, im Hintergrund ablaufen zulassen und mit dem eigentlichen Prozess fortzufahren. Ein Beispiel dafür ist, einen Teilprozess zu starten, der eine Datei sortiert und währenddessen ein paar andere Operationen durchzuführen und dann auf Beendigung des Kindprozesses zu warten:

```

1 exec("sort unsortierteDatei > sortierteDatei") if fork == nil
2     # Die Sortierung läuft nun
3     # Hier kann im Hauptprozess weiter gearbeitet werden
4
5 # Warte auf das Ende der Sortierung
6 Process.wait

```

Da ein Prozess der aufgerufen wird ein IO-Objekt ist, kann man ihn auch einem Block zur weiteren Verarbeitung übergeben.

```

1 IO.popen("date") { |f| puts "Das Datum ist #{f.gets}" }
2     # => Das Datum ist Do 18 Jan 22:21:40 CET 2007

```

3 Fazit

3.1 kurzer Vergleich mit anderen Sprachen

3.1.1 Ruby und Java

Ruby ist im Vergleich zu Java eine Nischensprache. Als Skriptsprache für Java-Programmierer spielt es eine untergeordnete Bedeutung, denn Jython (eine Python-Implementierung in Java) stellt eine ähnliche Funktionalität zur Verfügung und ist besser in Java integrierbar.

Java ist streng typisiert, größtenteils sogar statisch, in Java 5 wurde der Compiler mit Generics und Autoboxing erweitert um mehr Flexibilität zu bieten. Ruby setzt im Gegensatz dazu vollständig auf dynamische Polymorphie. Der Typ einer Variable steht erst zur Laufzeit fest.

Java hat eine standardisierte Laufzeitumgebung, die Virtuelle Maschine (VM), die von Sun entwickelt wird. Vorteil davon ist, dass der Code zu 100% portabel ist. Ruby hat einen C-Interpreter, der im Vergleich zur VM relativ langsam ist. Er kann aber durch Auslagern von Code in C-Routinen etwas optimiert werden.

Java unterstützt wie Java nur die Einfachvererbung. In Ruby kann Mehrfachvererbung aber über Mixins emuliert werden. In Java werden dafür Konzepte wie das Aspektorientierte Programmieren eingesetzt.

3.1.2 Ruby und C

C ist im Gegensatz zu Ruby eine kompilierte Sprache. Vorteil davon ist die schnellere Ausführungsgeschwindigkeit, Nachteil aber, z. B. dass Duck Typing nicht möglich ist.

C ist eine rein prozedurale Sprache, Ruby dagegen vollständig objektorientiert.

C ermöglicht hardwarenahe Programmierung, Ruby bietet dieses nur über C-Module.

C ist maschinennaher, man denkt oft aus der Sicht der Maschine, wenn man Programme schreibt. Bei Ruby steht das zu lösende Problem mehr im Mittelpunkt, maschinenspezifische Dinge werden verborgen.

3.1.3 Ruby und Python

Gemeinsamkeiten der beiden Sprachen sind, dass sie beide einen interaktiven Interpreter haben, mit dem sich Code und Codestücken sofort testen lassen. Objekte sind bei beiden dynamisch typisiert. Außerdem ist Konzept der Exceptions bei Ruby und Python gleich.

Es gibt auch einige wesentliche Unterschiede. Ruby kennt im Gegensatz zu Python Konstanten. Ruby hat nur einen Listentyp (Array), der dafür aber veränderbar ist, auch Strings sind in Ruby modifizierbar. In Ruby kann man auf Objektattribute nicht direkt zugreifen, man muss dafür entweder Methoden schreiben oder die Rubymethode `attr_reader` für die Attribute freischalten. Python unterstützt Mehrfachvererbung. Ruby nicht, bietet dafür die Möglichkeit Mixins in Klassen „hineinzumischen“ und somit eine Alternative zur Mehrfachvererbung. Eine weiteres interessantes Konzept, das Ruby von Python unterscheidet ist, dass man in Ruby Klassen jederzeit erweitern und modifizieren kann.

3.2 Nachteile

Ein großer Nachteil von Ruby ist im Gegensatz zu kompilierten Sprachen, die längere Ausführungsgeschwindigkeit und die Abhängigkeit von der Laufzeitumgebung bzw. dem Interpreter, da kein nativer Code erzeugt werden kann.

Überraschend für eine Sprache, die aus Japan kommt ist, dass Ruby kein Unicode und UTF-8 nur zum Teil unterstützt. Abgesehen von ASCII gibt es nur eine japanische Kodierung (Kanji) mit der Quelltext geschrieben werden kann.

So fanszinierend die Sprache auch teilweise ist, gibt es doch ein paar Inkonsistenzen oder zu komplexe Sachverhalte:

- Das Paradigma der Objektorientierung wird von dem zentralen Modul Math verletzt. Von dem Ausdruck `2.Math.sqrt` würde man erwarten, dass er den Wert von $\sqrt{2}$ ausgibt. Leider bekommt man nur die Fehlermeldung, „undefined method“. Der richtige Aufruf erfolgt über `Math.sqrt(2)`.
- Die Sichtbarkeit und Gültigkeit ist in Ruby nicht ganz einfach zu verstehen. Eigentlich sollten Variablen, die in einem Block definiert wurden, nur dort dort gültig sein. Existiert aber bereits in einem äußeren Block eine Variable mit gleichem Namen, wird diese verwendet. In einem `do |x| ... end` Block könnte die äußere Variable `x` überschrieben werden.
- Auch das Subtyping ist etwas merkwürdig, da man eine öffentliche Methode mit einer privaten überschreiben kann, so dass die Subklasse nicht mehr das gleiche öffentliche Interface besitzt.

Ein weiterer großer Nachteil ist, dass manche Fehler nur schwer zu finden sind. Der Interpreter führt beim Laden des Skripts nur wenige Überprüfungen durch. Tritt im Verlauf des Programms ein Fehler auf, kann es passieren, dass sich der Interpreter erst ein paar hundert Zeilen später beklagt. Das wird allerdings durch den mitgelieferten Ruby-Debugger kompensiert.

3.3 Vorteile

Ruby bietet auch einige Vorteile. Es ist eine moderne, aktiv weiterentwickelte Sprache mit einem sehr hohen Abstraktionsgrad. Ruby ist leicht zu erlernen, was e auch an der Ähnlichkeit zu anderen Sprachen liegt.

Der Code ist aufgrund sinnvoller Konventionen und dem Verzicht auf viele Sonderzeichen meist gut lesbar.

Ruby ist sehr gut dokumentiert. Zu fast allen Bibliotheken gibt es eine Dokumentation. Das Ruby Standardwerk [2] ist auch in einer deutschen Übersetzung verfügbar. Ein deutsches Wiki zu Ruby [6] ist im Aufbau.

Ein wesentlicher Vorteil von Ruby ist, dass es unabhängig von der Plattform ist. Der Interpreter ist für Linux, Windows, MacOS X, Unix und DOS erhältlich. Da er in C geschrieben ist, ist er auch leicht auf andere Systeme portierbar.

Der Sprachumfang ist durch die mitgelieferten Bibliotheken sehr groß. Z. B. sind standardmäßig Reguläre Ausdrücke unterstützt, als auch gängige Kommunikationsprotokolle (http, ftp, imap, ...), desweiterern gibt es Unterstützung für Datenbanken, Open SSL und einige Erweiterungen für grafische Benutzeroberflächen.

Mit Ruby lässt es sich einfach programmieren und die Sprache hinterlässt einen sehr flexiblen und mächtigen Eindruck.

Literatur

- [1] David Thomas, Andrew Hunt:
Programming Ruby, The Pragmatic Programmer's Guide.
Addison-Wesley, 978-0974514055, 2004
<http://www.rubycentral.com/book/>

- [2] David Thomas, Andrew Hunt:
Programmieren mit Ruby
Deutsche Übersetzung von: Programming Ruby, The Pragmatic Programmer's Guide.
Addison-Wesley, 3-8273-1965-X
Online-Version: <http://home.vrweb.de/~juergen.katins/ruby/buch/>

- [3] Vortragsfolien von Yukihiro Matsumoto
<http://www.rubyist.net/~matz/slides/112/>

- [4] Offizielle Ruby Homepage:
<http://www.ruby-lang.org>

- [5] Offizielle Ruby Dokumentation:
<http://www.ruby-doc.org>

- [6] Deutsches Wiki zu Ruby:
<http://www.rubywiki.de>