4

4 Ruby: Das Wichtigste

4.1 Eine kurze Einführung

Sie kennen sich im Umfeld von Java bestens aus und haben sich in den vorangegangenen Kapiteln bestimmt mal gefragt, was an Ruby so besonders ist und was Ruby on Rails so produktiv macht. Es sind die dynamischen Eigenschaften sowie – vor allem – Möglichkeiten der Metaprogrammierung. Damit Sie einen weitreichenden Eindruck von Rubys Spracheigenschaften bekommen, habe ich dieses Kapitel in unser Buch aufgenommen.

Sie erhalten eine kurze Einführung in die wichtigsten Spracheigenschaften von Ruby. Ein Anspruch auf Vollständigkeit besteht allerdings nicht, dafür gibt es reichlich andere und sehr gute Literatur.

Wir verweisen auf das Referenzwerk „Programming Ruby“[1] von Dave Thomas, aufgrund seiner Coverabbildung auch Pickaxe genannt. Eine Online-Version der ersten Auflage bietet mehr Details als diese kurze Einführung und ist daher als Ergänzung wärmstens zu empfehlen. Sie finden diese – zugegeben veraltete, aber immer noch brauchbare – Version unter der folgenden URL.

```
http://www.ruby-doc.org/docs/ProgrammingRuby/
```

Die Spracheigenschaften von Ruby werden mit vielen Beispielen illustriert, die Sie während Ihrer Lektüre mit der Interactive Ruby Shell (IRB) ausgiebig testen können. Dabei ist es durch „Herumspielen“ auch besser möglich, akute Fragen direkt am Code zu klären. Sie starten hierzu die Ruby-Shell mit dem folgenden Aufruf in der Kommandozeile.

```
$ irb
```

Oder für die JRuby-Variante.

```
$ jirb
```

[1] Siehe auch http://www.pragprog.com

Daraufhin ist die Shell startklar und kann bei Bedarf wieder mit der Eingabe des Befehls `exit` verlassen werden.

Wenn Ihnen das Erscheinungsbild der Ruby Shell zu trist vorkommt und Ihnen etwa Code-Completion wichtig ist, können Sie ein Gem für die IRB installieren, das zudem ein Syntax-Highlighting bietet, sofern Ihr Terminalprogramm damit umgehen kann. Beachten Sie hierbei bitte, dass sich das folgende beschriebene Gem nur mit Ruby und nicht mit JRuby nutzen lässt. Das Gem heißt Wirble und wird mit folgendem Aufruf installiert.

```
$ sudo gem install wirble

complete
Successfully installed wirble-0.1.2
1 gem installed
Installing ri documentation for wirble-0.1.2...
Installing RDoc documentation for wirble-0.1.2...
```

Jetzt können Sie in der Ruby Shell wesentlich effektiver arbeiten, denn die farbliche Hervorhebung der Ergebnisse trägt zu einer verbesserten Übersichtlichkeit bei. Mit der Tab-Taste können Sie sich Variablennamen und Methoden komplettieren lassen, was bei der Einarbeitung in die Standardbibliothek von Ruby sehr hilfreich sein kann.

4.1.1 Konzepte von Ruby

Ruby ist eine moderne Programmiersprache, die von Yukihiro Matsumoto, kurz Matz, Mitte der 1990er-Jahre entwickelt wurde und lange Zeit nicht über die Grenzen Japans hinaus bekannt wurde. Mit dem Buch *Programming Ruby* von Dave Thomas änderte sich dies in den letzten Jahren deutlich, und Ruby hat heute eine stetig steigende Anzahl von Anhängern auf der ganzen Welt.

Ruby ist eine interpretierte objektorientierte Programmiersprache, die dank Ruby on Rails extrem an Bekanntheit gewonnen hat. So wird den meisten Einsteigern in Ruby on Rails der erste Kontakt nicht über die Programmiersprache Ruby, sondern über das Framework Rails gelungen sein, was wahrscheinlich ein Grund dafür ist, dass auf Podiumsdiskussionen immer wieder Sprachen mit Frameworks gleichgesetzt werden und daraus die skurrilsten Argumentationen entstehen.

Ruby verfügt zur Speicherverwaltung über einen Garbage Collector, den Java bekanntlich auch mitliefert, um den Entwickler von lästigen Speicherallozierungen zu befreien.

Die Sprache ist stark getypt ohne jedoch auf eine statische Typisierung zurückgreifen zu müssen. Damit ist die Notwendigkeit eines Compilers für Ruby nicht gegeben. Die Konzepte der Objektorientiertung stammen überwiegend aus dem Umfeld von Smalltalk, was dazu führt, dass Ruby keine primitiven Datentypen kennt, sondern ausschließlich Objekte. Das Motto von Smalltalk: „Everything is an object“ gilt für Ruby ebenso. Methoden werden nicht auf Objekten aufgerufen, sondern Nachrichten an Objekte gesendet. Diese Sichtweise ist ebenfalls aus Smalltalk bekannt und findet sich beispielweise auch in Objective-C wieder, der Sprache, die Apple als Haus- und Hofsprache für MacOS X verwendet.

Wenn Nachrichten an Objekte gesendet werden, sieht das in Ruby ungefähr so aus.

```
irb(main):001:0> 2.to_s
=> "2"

irb(main):002:0> 2.send(:to_s)
=> "2"
```

Im obigen Beispiel wird das gleiche Ergebnis wie im unteren erzeugt. Im ersten Fall wird die Methode `to_s` auf der Zahl `2` (Klasse: `Fixnum`) aufgerufen. Im zweiten Beispiel wird die Methode `send` genutzt, um die Nachricht `to_s` an die `2` zu senden. Mit der Methode `send` ist es also möglich, dynamisch Methodenaufrufe abzusetzen. Beispiele hierfür zeige ich Ihnen im weiteren Verlauf dieses Kapitels.

Generell ist Ruby eher für den Menschen als für die Computer gestaltet worden, was eine viel bessere Lesbarkeit als beispielsweise bei Java zur Folge hat.

Ruby arbeitet zeilenorientiert, was bedeutet, dass jede neue Anweisung nach einem Zeilenumbruch beginnt. Sollten Sie einmal mehrere kurze Anweisungen in einer Zeile implementieren wollen, so müssen die Elemente mit einem Semikolon voneinander getrennt werden. Das folgende Beispiel verdeutlicht dies.

```
(1..10).each do |n|; puts n; end
```

Die obige Zeile könnte alternativ so aussehen:

```
(1..10).each do |n|
  puts n
end
```

Wie in anderen Programmiersprachen können in Ruby sehr lange Anweisungen mit einem Backslash auf mehrere Zeilen umbrochen werden. Zeilenumbrüche nach Punkten, Operatoren oder Kommata wertet Ruby als eine Zeile, womit sich eine bessere Lesbarkeit erreichen lässt.

Eine Ausnahme sind Strings. Diese können beliebige Zeilenumbrüche enthalten und werden trotzdem wie eine einzige Zeile interpretiert.

4.1.2 Einige Programmierrichtlinien

Damit Sie von den Ruby-Veteranen nicht als Umsteiger erkannt werden, sollten Sie einige Konventionen kennen, die typisch für Ruby sind und in Java eher selten Verwendung finden.

Für Klassennamen gilt dieselbe Schreibweise wie in Java. Sie beginnen mit einem Großbuchstaben und werden bei zusammengesetzten Wörtern *Camelcased* geschrieben, also mit einem Großbuchstaben für jedes Substantiv im Klassennamen. Hierzu einige Beispiele.

```
KlassenName

FeedsController

ActiveRecord
```

Methoden- und Variablennamen schreibt man klein. Es gibt zwei Schreibweisen für die Namen von zusammengesetzten Begriffen.

- Variante I:

```
methoden_name

variablen_name
```

- Variante II:

```
methodenName

variablenName
```

Die Variante mit dem Unterstrich ist allerdings wesentlich öfter anzutreffen und meiner Meinung nach etwas besser lesbar.

Es gibt Methoden, die `true` oder `false` zurückgeben und dies durch ein abschließendes Fragezeichen im Methodennamen deutlich machen.

```
ob_ruby_wohl_toll_ist?
```

Methoden, die eine Instanz direkt manipulieren und somit auch als „gefährlich“ (engl. dangerous) bezeichnet werden, machen dies durch ein Ausrufezeichen deutlich. Somit manipuliert das nächste Beispiel direkt das Objekt, auf dem diese Methode aufgerufen wird, beziehungsweise diese Nachricht gesendet bekommt.

```
ruby_ist_unkaputtbar!
```

Sie kennen aus der Objektorientierung die Instanz- sowie die Klassenvariablen. In Ruby erhalten Instanzvariablen vor ihrem Namen einen Klammeraffen.

```
@instanz_variable
```

Klassenvariablen bekommen gleich zwei dieser Klammeraffen vorangestellt.

```
@@klassen_variable
```

Außerdem gibt es die globalen Variablen, die mit einem Dollar-Zeichen vor dem Namen beginnen.

```
$globale_variable
```

4.1.2.1 Konstanten

Konstanten werden, wie auch in Java üblich, durchgehend in Großbuchstaben geschrieben.

```
DIES_IST_EINE_KONSTANTE
```

Diese Schreibweise ist allerdings ebenfalls häufig anzutreffen.

```
DiesIstAuchEineKonstante
```

Es ist nicht möglich, Konstantennamen mit einem Kleinbuchstaben zu beginnen. Die Probe aufs Exempel macht klar, warum.

Zunächst definiert man eine Variable namens `tier` mit dem String `Hund`.

```
>> tier = "Hund"
=> "Hund"
```

Danach wird ein neuer Wert `Katze` zugewiesen.

```
>> tier = "Katze"
=> "Katze"
```

Nun wird eine Konstante mit dem Namen `Tier` und dem Wert `Hund` definiert.

```
>> Tier = "Hund"
=> "Hund"
```

Jetzt wird, wie im obigen Beispiel, versucht, der Konstante `Tier` einen neuen Wert zuzuweisen, was bekanntlich nur den Variablen vorbehalten ist.

```
>> Tier = "Katze"
(irb):30: warning: already initialized constant Tier
=> "Katze"
```

Konstanten, die innerhalb einer Klasse oder eines Moduls definiert wurden, sind innerhalb dieser Klasse oder dieses Moduls direkt verfügbar. Von außerhalb kann mit dem Operator `::` auf sie zugegriffen werden. Auch hierzu ein kleines Beispiel.

```
MyApp::Person.GENDER
```

Alle Konstanten, die außerhalb eines Moduls oder einer Klasse definiert wurden, sind ohne diesen Operator direkt verfügbar.

Hinweis:
Konstanten können nicht innerhalb von Methoden definiert werden.

Mit diesem Grundschatz an Konventionen kommen Sie schon mal ein gutes Stück weiter. Der nächste Abschnitt beschäftigt sich direkt mit den Spracheigenschaften von Ruby.

4.1.3 Einfache Rechenbeispiele

Zunächst sollten Sie ein wenig mit den vier Grundrechenarten experimentieren, um ein Gefühl für die Shell und den Umgang mit Zahlen in Ruby zu bekommen.

4.1.4 Zahlen

Zwei ganz zahlige Werte lassen sich wie in jeder anderen Sprache mit einem Plus-Operator addieren. Die Summe stimmt.

```
>> 10 + 2
=> 12
```

Alternativ könnten Sie auch Folgendes schreiben.

```
10.send(:+, 2)
```

Dabei wird die Methode `send` auf dem Objekt `10` aufgerufen. Als Parameter werden die Methode `:+` sowie das Argument `2` übergeben. Die Differenz stimmt natürlich ebenfalls.

```
>> 5 - 3
=> 2
```

Auch in Ruby gilt Punktrechnung vor Strichrechnung.

```
>> 10 * 2 + 1
=> 21
```

Klammern haben Vorrang.

```
>> 10 * (2 + 1)
=> 30
```

Zwei ganzzahlige Werte sorgen ebenso für ein ganzzahliges Ergebnis.

```
>> 15 / 4
=> 3
```

Wenn ein Float-Wert durch einen Fixnum-Wert geteilt wird, ist das Ergebnis ebenfalls ein Float-Wert.

```
>> 15.0 / 4
=> 3.75
```

Potenzieren wird mit dem Doppelstern durchgeführt.

```
>> 3 ** 2
=> 9
```

Der Modulo-Operator ist selbstverständlich auch in Ruby vorhanden.

```
>> 3 % 2
=> 1
```

Auch bei Dezimalzahlen kann der Modulo-Operator angewendet werden.

```
>> 15.5 % 3
=> 0.5
```

4.1.5 Strings

Strings oder auch Zeichenketten können in Ruby so

```
>> "Hallo".class
=> String
```

oder so

```
>> 'hallo'.class
=> String
```

definiert werden.

Dabei ist Arithmetik mit Zeichenketten ebenso möglich, wie das folgende Beispiel zeigt.

```
>> "bla" * 3
"blablabla"
```

Das obige Beispiel liefert das gleiche Ergebnis wie `"bla" + "bla" + "bla"`. Wenn Sie die Zeit haben, überlegen Sie sich einmal, wie man das in Java implementieren würde.

Das Zusammenfügen von Strings funktioniert aber auch so wie in Java.

```
>> "Hallo " + "Welt!"
=> "Hallo Welt!"
```

Hier werden zwei Strings zusammengefügt.

```
>> "1" + "2"
=> "12"
```

Weil die `2` hier eine Zeichenkette ist, wird sie dreimal aneinandergehängt.

```
>> "2" * 3
=> "222"
```

Einige nützliche Methoden der Klasse `String` zeigt das nächste Beispiel. `capitalize` wandelt das erste Zeichen des Strings in einen Großbuchstaben um.

```
>> "ruby".capitalize
=> "Ruby"
```

Mit `reverse` wird die Reihenfolge der Zeichen eines `String`-Objektes umgedreht.

```
>> "ruby".reverse
=> "ybur"
```

Alle Zeichen eines `String`-Objektes lassen sich mit der Methode `upcase` in Großbuchstaben umwandeln.

```
>> "ruby".upcase
=> "RUBY"
```

Übrigens können in Ruby auch Strings hochgezählt werden.

```
>> "ruby".next
=> "rubz"
```

Eine Invertierung der Groß- und Kleinbuchstaben eines Strings erledigt die Methode `swapcase`.

```
>> "RuBy".swapcase
=> "rUbY"
```

Die Länge eines Strings lässt sich mit der Methode `length` abfragen.

```
>> "Hallo Welt!".length
=> 11
```

Eine weitere gängige `String`-Funktion, die der Klasse String von Ruby on hinzugefügt wurde, ist beispielsweise `blank?`.

```
>> " ".blank? == true

=> true
```

Mit `blank?` kann abgefragt werden, ob ein String leer ist.

Eine weitere Methode, die sehr nützlich sein kann, ist `each_with_index`.

```
>> "Hallo\nWelt".each_with_index {|line, i| puts "#{i}: #{line}"}
0: Hallo
1: Welt

=> "Hallo\nWelt"
```

`each_with_index` ruft den übergebenen Block mit den beiden Parametern `line` und `index` (`i`) auf. Diese Funktionalität stammt aus dem Modul Enumerable, das von `String` eingebunden wird.

Ebenfalls sehr nützlich ist die Methode `scan`.

```
irb(main):009:0> "abc".scan(/./).each {|char| puts char}
a
b
c

=> ["a", "b", "c"]
```

Das obige Beispiel verwendet die Methode `scan`, um mit dem übergebenen regulären Ausdruck nach Zeichen zu suchen, die von `each` und dem übergebenen Block einzeln ausgegeben werden.

Zur Handhabung von Strings mit Leerzeichen bieten sich `split` und `join` an.

```
irb(main):010:0> "split splittet Strings auf".split.join(", ")

=> "split, splittet, Strings, auf"
```

Mit `split` werden Strings aufgesplittet, wobei als Übergabeparameter ein weiterer String die Splitgrenzen angibt. Auch hier können reguläre Ausdrücke verwendet werden. Mit `join` lassen sich einzelne Strings zusammenfügen, wobei der Übergabeparameter den Teil-String definiert, der zwischen den Elementen eingebracht wird; in diesem Beispiel das Komma mit angehängtem Leerzeichen.

4.1.6 Konvertierungen

Selbstverständlich können auch `String`-Objekte, die Zahlenwerte enthalten, in Zahlen vom Typ `Fixnum` und `Float` umgewandelt werden. Hierzu gibt es eine Reihe Methoden, die sowohl für `String`-Objekte als auch für Zahlen verfügbar sind.

Aus einer Zahl wird auf diese Weise ein String.

```
>> 4711.to_s

=> "4711"
```

Ein `Fixnum`-Objekt kann so zu einem `Float`-Wert werden.

```
>> 4711.to_f

=> 4711.0
```

Eine Zeichenkette lässt sich in einen `Fixnum`-Wert überführen.

```
>> "0815".to_i

=> 815
```

Das Zusammenfügen von Zeichenketten mit Objekten, die nicht vom Typ `String` sind – was Sie aus der Java-Welt kennen – ist mit Ruby nicht möglich. Das folgende Beispiel verdeutlicht diesen Umstand.

```
>> puts "Ein Tag hat " + 24 + " Stunden."

TypeError: can't convert Fixnum into String
     from (irb):31:in `+'
     from (irb):31
     from :0
```

Das Beispiel funktioniert erst, wenn die `24` mit der Methode `to_s` in einen String umgewandelt wurde.

```
>> puts "Ein Tag hat " + 24.to_s + " Stunden."

Ein Tag hat 24 Stunden.

=> nil
```

4.1.7 Schleifen und Bedingungen

Wie jede andere Programmiersprache bietet auch Ruby die Verarbeitung von Schleifen an und kann ebenso mit Bedingungen umgehen. Das folgende Beispiel zeigt eine typische Schleife in Ruby:

```
>> 6.times do
?>   puts "Schönen guten Tag"
>> end

Schönen guten Tag
Schönen guten Tag
Schönen guten Tag
Schönen guten Tag
Schönen guten Tag
Schönen guten Tag

=> 6
```

Sie sehen im obigen Beispiel gleich, wie in Ruby Blöcke definiert werden. Hierzu dienen die Schlüsselworte `do` und `end`. Sie können hierfür auch geschweifte Klammern verwenden, allerdings findet man die `do-end` Combo wesentlich öfter vor. Eine Mischform von Klammern mit Schlüsselwörtern ist nicht zulässig.

Hinweis:
Es ist eine Konvention, für einzeilige Blöcke geschweifte Klammern zu verwenden. Mehrzeilige Blöcke werden in `do` und `end` geschrieben.

Ein weiterer Verwendungszweck von Schleifen kann das Hochzählen einer Variablen sein, wie dies der folgende Code zeigt.

```
>> n = 0

=> 0

>> 5.times do
?>     n += 1
>>   n
>>   end

=> 5
```

In diesem Beispiel wird zunächst die Variable `n` mit dem Wert `0` initialisiert und dann in fünf Schleifendurchläufen jeweils um den Wert `1` erhöht und ausgegeben.

Die Ausgabe sieht dann so aus:

```
>> n = 0

=> 0

>> 5.times { puts n+= 1 }
1
2
3
4
5

=> 5
```

Der Operator `++`, wie Sie ihn aus Java kennen, ist in Ruby nicht verfügbar.

4.1.7.1 while-Schleife

Die `while`-Schleife ist Ihnen sicher aus Java bekannt. `while` erhält eine Abbruchbedingung, die den Inhalt der Schleife so lange ausführt, bis die Bedingung falsch (`false`) ist.

Hierzu ein kleines Beispiel.

```
while (text = gets).chomp != "Exit"
  puts "Die Eingabe lautet " + text
end
```

Das Beispiel nimmt eine Eingabezeile entgegen, speichert das Ergebnis in der variablen `test` ab und vergleicht die Eingabe dann mit dem String `Exit`. Weil die Eingabezeile zugleich einen Zeilenumbruch mitliefert, wird dieser mit der Methode `chomp` entfernt.

Wenn das kleine Programm läuft, wird die Eingabe so lange auf der Kommandozeile ausgegeben, bis der Benutzer das Wort `Exit` eingibt.

Versuchen Sie es ruhig einmal.

4.1.7.2 Bedingungen

Ruby kennt verschiedene Bedingungen, die auch in anderen Sprachen verfügbar sind. Lediglich die Syntax unterscheidet sich teilweise von der anderer Sprachen. Starten wir mit einigen einfachen Beispielen.

Listing 4.1 http://www.pastie.org/234620

```
class Bedingungen
  def test
  n = 10

  if n == 10 then
    puts "n ist 10"
  end
end
end

test = Bedingungen.new

test.test
```

Die Klasse `Bedingungen` enthält eine Methode namens `test`, die eine Variable `n` mit dem Wert `10` initialisiert. Danach prüft eine `if`-Bedingung, ob der Wert gleich `10` ist. Zu sehen ist sofort, dass der Vergleichsoperator `==` ebenso wie in Java zu finden ist.

Hinweis:
Das Schlüsselwort `then` ist optional und kann somit weggelassen werden.

Tabelle 1.1 zeigt eine Liste mit den am häufigsten verwendeten Vergleichsoperatoren.

Tabelle 4.1 Die gängigsten Vergleichsoperatoren

Operator	Beschreibung
`==`	Gleich
`!=`	Ungleich
`>`	Größer als
`<`	Kleiner als
`>=`	Größer oder gleich
`<=`	Kleiner oder gleich

In Java sind solche Vergleiche fast immer auf numerische Werte beschränkt, da hier ebenfalls keine Operatoren überladen werden können. In Ruby ist das anders, wie die folgenden Zeilen zeigen. Zunächst wird hier ein `String`-Vergleich durchgeführt.

```
>> "Ja" != "Nein"

=> true
```

Das nächste Beispiel ist ein Vergleich, ob das Zeichen `a` kleiner dem Zeichen `b` ist.

```
>> "a" < "b"

=> true
```

In der ASCII-Tabelle findet man das große `A` vor dem kleinen `a`. Aus diesem Grunde wird die nächste Bedingung als `false` ausgewertet.

```
>> "a" < "A"
=> false
```

Da `a` nicht nach `b` kommt, gibt das folgende Beispiel `false` zurück.

```
>> "a" > "a".next
=> false
```

Der umgekehrte Vergleich führt zu einem `true`.

```
>> "a".next > "a"
=> true
```

In den vorherigen Beispielen ist relativ leicht zu erkennen, dass für den Vergleich von `String`-Objekten der jeweilige Zeichen-Code herangezogen wird.

Auch in Ruby gibt es ein Konstrukt namens `elsif`, das es erlaubt, verschiedene Bedingungen nacheinander zu prüfen. So wird im nächsten Beispiel lediglich eine Anweisung ausgeführt.

Listing 4.2 http://www.pastie.org/234622

```
class Comparer

  def initialize(wert)
    @wert = wert
  end

  def compare(wert)
    if wert > @wert
      puts "Wert ist größer als #{@wert}"
    elsif wert < @wert
      puts "Wert ist kleiner als #{@wert}"
    elsif wert == @wert
     puts "Wert ist gleich #{@wert}"
    end
  end
end
```

Wenn Sie nun in der Interactive Ruby Shell ein wenig mit dieser Klasse spielen, können Sie das Verhalten leicht selbst überprüfen. Zunächst wird die Klasse `Comparer` geladen.

```
>> require 'Comparer'
=> true
```

Danach bekommt die Variable `c` eine Instanz von `Comparer` zugewiesen.

```
>> c = Comparer.new(100)
=> #<Comparer:0x1849ecc @wert=100>
```

Als Nächstes starten die Vergleiche. Im ersten Beispiel wird der Wert `10` mit dem Wert `100` verglichen.

```
>> c.compare(10)
Wert ist kleiner als 100
=> nil
```

Jetzt wird mit dem Wert `100` verglichen.

```
>> c.compare(100)
Wert ist gleich 100

=> nil
```

Und zuletzt mit dem Wert `1000`.

```
>> c.compare(1000)
Wert ist größer als 100

=> nil
```

Hinweis:

Beim `elsif`-Konstrukt sollte beachtet werden, dass nur die erste als wahr ermittelte Bedingung ausgeführt wird. Nachfolgende Bedingungen, die ebenfalls zutreffen, werden nicht mehr ausgeführt.

Ein weiteres Konstrukt für die Überprüfung von Bedingungen ist `unless`. Hier ein Beispiel.

```
customer.save unless customer.nil?
```

Dabei wird die Lesbarkeit im Vergleich zu `if`-Abfragen etwas erhöht.

Die Negierung einer `if`-Bedingung kann ebenfalls mit `unless` erreicht werden, wie das nächste Beispiel zeigt.

```
unless user.not_logged_in?
  order.submit
end
```

Arrays

Dieser Abschnitt befasst sich mit Arrays und deren flexibler Handhabung. Arrays sind Ihnen sicher bekannt, und so verdeutlicht das folgende Beispiel die konkrete Schreibweise in Ruby.

Zuerst wird eine Variable namens `tiere` mit einem Array und den dazugehörenden Werten initialisiert.

```
>> tiere = ["Hund", "Katze", "Maus"]

=> ["Hund", "Katze", "Maus"]
```

Dann können Sie den Typ von `tiere` abfragen.

```
>> tiere.class

=> Array
```

Zu guter Letzt kann noch die Größe des Arrays ausgegeben werden.

```
>> tiere.size

=> 3
```

Der Zugriff auf einzelne Objekte des Arrays erfolgt wie in Java über einen Index, der auch in Ruby bei `0` beginnt:

```
>> tiere[0]
=> "Hund"
>> tiere[1]
=> "Katze"
>> tiere[2]
=> "Maus"
>> tiere[3]
=> nil
```

Im obigen Beispiel sehen Sie, dass für den Index `3` kein Objekt im Array gespeichert ist. Sie haben nun die Möglichkeit, einzelne neue Objekte an beliebiger Stelle im Array einzufügen.

```
>> tiere[6] = "Fisch"
=> "Fisch"
>> tiere[10] = "Frosch"
=> "Frosch"
>> tiere
=> ["Hund", "Katze", "Maus", nil, nil, nil, "Fisch", nil, nil, nil, "Frosch"]
```

Das `Array` erhält also an Stelle `6` und `10` einen neuen Eintrag. Damit sind alle dazwischenliegenden Einträge automatisch `nil`. Natürlich lassen sich beliebige Objekte in ein Array einfügen, womit auch `Array`-Objekte selbst in Arrays eingefügt werden können.

Das folgende Beispiel zeigt ein `Array` von Personen, die jeweils aus einem `Array` mit Vor- und Nachnamen bestehen:

```
>> personen = [["Helge", "Schneider"], ["Johann", "Könich"], ["Atze", "Schröder"]]
=> [["Helge", "Schneider"], ["Johann", "K\303\266nich"], ["Atze",
"Schr\303\266der"]]
```

Mit dem Index `1` erreichen wir die Daten zu Johann Könich.

```
>> personen[1]
=> ["Johann", "K\303\266nich"]
```

Den Vornamen von Herrn Schneider können wir so abfragen.

```
>> personen[0][0]
=> "Helge"
```

Einige weitere Dinge, die mit Arrays möglich sind, zeige ich Ihnen in den folgenden Zeilen.

Mit Hilfe der Methode `sort` kann das Array sortiert werden.

```
>> personen.sort

=> [["Atze", "Schr\303\266der"], ["Helge", "Schneider"], ["Johann",
"K\303\266nich"]]
```

Mit der Methode `puts` lässt sich ein Array auf der Console ausgeben, wobei die Klammern und Anführungszeichen entfernt werden.

```
>> puts personen

Helge
Schneider
Johann
Könich
Atze
Schröder
```

Das interessanteste Feature ist allerdings das Addieren und Subtrahieren von Arrays. Dabei kann der Inhalt von Array `a` vom Inhalt des Arrays `b` subtrahiert werden.

```
>> a = [1, 3, 5, 7]

=> [1, 3, 5, 7]

>> b = [3, 7]

=> [3, 7]

>> a - b

=> [1, 5]
```

Dies funktioniert nicht nur mit Arrays, in denen Zahlen gespeichert sind.

Eine weitere Möglichkeit, schnell aus einer Liste von Strings ein Array zu machen, ist `%w()`. Das folgende Beispiel verdeutlicht die Funktionsweise.

```
>> satz = %w(Dies ist ein Array)

=> ["Dies", "ist", "ein", "Array"]
```

Iterationen

Durch die Inhalte von Arrays können Sie selbstverständlich auch iterieren. Hierzu kommt üblicherweise folgendes Konstrukt zum Einsatz.

Zuerst wird ein `Array` namens `tiere` angelegt.

```
>> tiere = %w(Hund Katze Maus Fisch Kochschinken)

=> ["Hund", "Katze", "Maus", "Fisch", "Kochschinken"]
```

Nun wird die Iteration formuliert.

```
>> tiere.each do |tier|
?>     puts "Mein Tier ist von Natur aus " + tier
>>   end

Mein Tier ist von Natur aus Hund
Mein Tier ist von Natur aus Katze
Mein Tier ist von Natur aus Maus
Mein Tier ist von Natur aus Fisch
Mein Tier ist von Natur aus Kochschinken
=> ["Hund", "Katze", "Maus", "Fisch", "Kochschinken"]
```

Sie wissen nun, dass `each` ein Iterator ist. Das zurückgegebene Objekt wird in der Variablen `tier` abgelegt. Eine weitere Form eines Iterators ist die Methode `times`, die Sie weiter oben schon gesehen haben.

Geben Sie in der Ruby Shell den folgenden Block ein, dann wird schnell klar, warum dies ein Iterator ist.

```
>> 3.times do |n|
?>     puts n
>>   end
0
1
2

=> 3
```

Hash-Objekte

Ein weiteres, oft genutztes Objekt ist das `Hash`-Objekt. Hier können ähnlich wie in einem `Array` Objekte abgelegt werden. Allerdings wird für den Zugriff kein Index in Form einer Zahl verwendet. Vielmehr ist das `Hash`-Objekt mit der `java.util.Hashtable` zu vergleichen, die Key-Value-Paare speichert. Daher kann mit dem Key auf einen Wert zugegriffen werden. Diesen Umstand verdeutlicht das folgende Beispiel.

```
>> person = {
?>     "anrede" => "Herr",
?>     "vorname" => "Helge",
?>     "nachname" => "Schneider",
?>     "Beruf" => "Entertainer"
>>   }

=> {"Beruf"=>"Entertainer", "anrede"=>"Herr", "nachname"=>"Schneider", "vorna-
me"=>"Helge"}
```

Nun kann mit dem gewünschten Key ein zugeordneter Wert abgefragt werden.

```
>> person["nachname"]

=> "Schneider"
```

Auch über `Hash`-Inhalte kann iteriert werden. Hierzu gibt es die Methode `each`, die jeweils die Objekte `key` und `value` liefert.

```
>> person.each do |key, value|
?>     puts key.capitalize + ": " + value
>>   end
Beruf: Entertainer
Anrede: Herr
Nachname: Schneider
Vorname: Helge
```

Ebenso können Sie durch die Liste der Keys iterieren.

```
>> person.each_key do |key|
?>     puts key
>>   end
Beruf
anrede
nachname
vorname
```

Oder Sie iterieren durch die Liste der Werte.

```
>> person.each_value do |value|
?>     puts value
>>   end
Entertainer
Herr
Schneider
Helge
```

Hinweis:
Es ist eine Konvention, die Keys in Kleinbuchstaben zu schreiben.

So weit zu den grundlegenden Sprachkonstrukten von Ruby. Der nächste Abschnitt befasst sich mit den objektorientierten Konzepten von Ruby und geht dabei näher auf die Unterschiede zu Java ein.

4.2 Sprachkonzepte und vertiefende Informationen

4.2.1 Alles ist ein Objekt

Grundsätzlich sind in Ruby auch Zahlen Objekte, daher können Sie auch folgende interessante aber auch sehr lesbare Konstrukte verwenden.

So finden Sie den Klassennamen der Zahl `100` heraus.

```
>> 100.class
=> Fixnum
```

Dezimalzahlen sind Instanzen der Klasse `Float`.

```
>> 100.0.class
=> Float
```

Wollen Sie wissen, ob ein Wert `0` ist?

```
>> 100.zero?
=> false
```

Nur eine echte `0` ist auch `zero`.

```
>> 0.zero?
=> true
```

Absolutwerte werden durch die Methode `abs` ausgegeben.

```
>> -1942.abs
=> 1942
```

Wenn Sie wissen wollen, welche Methoden die Klasse `Fixnum` kennt, rufen Sie die Methode `methods` auf.

```
>> 1001.methods
=> [...] #Liste aller Methoden von Fixnum
```

Auch in Ruby haben Zahlen eine bestimmte Größe. Größere Zahlen werden dem Typ `Bignum` zugewiesen.

```
>> (100**100).class
=> Bignum
```

4.2.2 `nil` ist auch ein Objekt

Sie kennen `null` aus der Java-Welt, wo `null` bedeutet, dass keine Objektreferenz vorhanden ist. Des Weiteren ist `null` auch kein Objekt. In Ruby ist das Pendant zu `null`, das Objekt `nil`.

Die Klasse `NilClass` ist der Typ für `nil`, was Sie sich leicht ausgeben lassen können.

```
>> nil.class
=> NilClass
```

Sie können `nil` einer Variablen zuweisen.

```
>> x = nil
=> nil
```

Danach lässt sich auch hier der Klassenname abfragen.

```
>> x.class
=> NilClass
```

Wie bei allen Objekten können Sie auch bei `nil`-Objekten eine Liste der verfügbaren Methoden abfragen.

```
>> x.methods
=> [] # Liste aller Methoden von NilClass
```

Der Vorteil bei `nil` als Objekt ist die Tatsache, dass es keine `NullPointerExceptions` gibt. Genauer gesagt, laufen Sie nicht Gefahr, sich mit `NullPointer`-Fehlern beschäftigen zu müssen. Das befreit Sie aber nicht von der Aufgabe, in vielen Situationen zu prüfen, ob Ihre Variablen und Objekte gültige Werte enthalten.

4.2.3 Methoden sind Nachrichten

In der Sprache Smalltalk werden Objekten Nachrichten gesendet, anstatt Methoden auf Objekten aufgerufen. In Java werden Methoden oder auch Funktionen von Objekten aufgerufen. Ruby hat das Senden von Nachrichten von Smalltalk übernommen. Die folgenden Code-Zeilen sind Beispiele für Nachrichten an Objekte.

Zunächst wird eine Variable `s` mit dem `String` `"Dies ist ein Text"` definiert.

```
>> s = "Dies ist ein Text"
```

```
=> "Dies ist ein Text"
```

Nun wird die Nachricht `index` mit dem Parameter `"ist"` an das Objekt `s` gesendet.

```
>> s.index("ist")
=> 5
```

Danach wird die Nachricht `length` an das Objekt `s` gesendet, um die Anzahl der Zeichen abzufragen.

```
>> s.length
=> 17
```

Eine Nachricht wird innerhalb einer Klasse verwendet und an sich selbst gesendet.

```
>> eine_nachricht_an_self
```

Jedes Objekt kennt die Methode oder besser Nachricht `send`, die als Übergabeparameter den Namen einer weiteren Methode bekommen kann.

```
>> "Hallo".send("reverse")
=> "ollaH"
```

4.2.4 Klassen und Subklassen

In Ruby werden Klassen wie im folgenden Beispiel erstellt.

```
class Person
end
```

Einen Konstruktor wie in Java gibt es, indem die Methode `initialize` implementiert wird. Hierbei können auch beliebige Argumente übergeben werden. Dazu ein Beispiel.

Listing 4.3 http://www.pastie.org/234624

```
class Person
  def initialize(vorname, nachname)
    @vorname = vorname
    @nachname = nachname
  end
end
```

Mit der Definition der Methode `initialize` können nun Instanzen der Klasse `Person` erzeugt werden.

```
helge = Person.new(„Helge“, „Schneider“)
```

In Java wie in Ruby gibt es einen Garbage Collector, der nicht mehr referenzierte Objekte aus dem Speicher entfernt. Damit ist in Ruby ebenfalls kein explizites Aufräumen durch den Programmierer nötig. Einen Destruktor müssen Sie ebenfalls nicht programmieren, weil der Garbage Collector mit zyklischen Referenzen umgehen kann.

4.2.5 Attribute

Variablen haben Sie bereits kennengelernt. Attribute gibt es ebenfalls, und Sie können sich genau wie in Java die Arbeit machen, entsprechende Getter und Setter zu schreiben. Das sieht dann in etwa so aus.

Listing 4.4 http://www.pastie.org/234625

```
class Person
  def name
    @name
  end

  def name=(name)
    @name = name
  end
end

p = Person.new
p.name = "Helge"
puts p.name
```

Ruby ist allerdings nicht darauf angewiesen, dass der Programmierer explizit Getter- und Setter-Methoden in seine Klassen einbaut. Wenn Sie lediglich die Inhalte von Instanzvariablen abfragen oder setzen wollen, können Sie auch Folgendes schreiben.

Listing 4.5 http://www.pastie.org/234626

```
class Person
  attr_writer :name
  attr_reader :name
end

p = Person.new
p.name = "Helge"
puts p.name
```

Die Methoden `attr_writer` und `attr_reader` können eine Liste von Symbolen verarbeiten. Diese Symbole sind in Ruby eindeutig und beginnen mit einem Doppelpunkt. In diesem Beispiel ist das Symbol `:name`. Mit `attr_writer` werden implizit Methoden zum Setzen von Variablenwerten bereitgestellt. `attr_reader` stellt die Getter zur Verfügung.

Wenn Sie für Ihre Attribute immer ein Paar von Gettern und Settern benötigen, können Sie auch folgende Methode verwenden.

Listing 4.6 http://www.pastie.org/234627

```
class Person
  attr_accessor :name
end

p = Person.new
p.name = "Helge"
puts p.name
```

Mit `attr_accessor` werden Ihnen Getter und Setter implizit zur Verfügung gestellt.

4.2.6 Struct als Klasse

Eine Datenstruktur lässt sich als Klasse verwenden. Hierzu müssen lediglich einige Konventionen beachtet werden. Zunächst definieren wir ein `Struct`-Objekt:

```
AddressStruct = Struct.new("Address", :street, :city, :zipcode)
```

Dann kann eine Instanz erstellt werden, wobei es zwei Möglichkeiten gibt. Hier Möglichkeit Nummer eins.

```
address = AddressStruct.new
```

Nummer zwei:

```
address = Struct::Address.new
```

Nun können Sie auf die Attribute zugreifen und diese mit Werten füllen.

Listing 4.7 http://www.pastie.org/234630

```
AddressStruct = Struct.new("Address", :street, :city, :zipcode)

address = AddressStruct.new

address.street = "Ameisenweg 1"
address.city   = "Itzenplitz"
address.zipcode = 47112

puts "Bob Ross wohnt in " + address.street
puts "in " + address.zipcode.to_s + " " + address.city
```

Sie sehen also gleich, wie praktisch `Structs` sein können.

4.2.7 Die Methode dup

Jedes Objekt lässt sich mit Hilfe der Methode `dup` duplizieren. Dies entspricht der Methode `clone()` in `java.lang.Object` und fertigt eine einfache Kopie eines Objektes an. Das folgende Beispiel verdeutlicht den Einsatz von `dup`.

Listing 4.8 http://www.pastie.org/234631

```
class Clone
  attr_accessor :anrede, :vorname, :nachname

  def initialize(anrede, vorname, nachname)
    @anrede = anrede
    @vorname = vorname
    @nachname = nachname
  end
end

c = Clone.new("Herr", "Helge", "Schneider")

puts "Sehr geehrter " + c.anrede + " " + c.vorname + " " + c.nachname

d = c.dup

puts "Sehr geehrter " + d.anrede + " " + d.vorname + " " + d.nachname
```

Im obigen Beispiel wird erst einmal eine Klasse namens `Clone` angelegt. Danach werden die drei Attribute `anrede`, `vorname` und `nachname` mit Accessor-Methoden versehen. Der Konstruktor erlaubt das Erstellen von Instanzen bei Übergabe der drei angegebenen Attribute.

Ein Praxisbeispiel für die Methode `dup` findet sich in der Klasse `ActiveRecord::Base`.

Listing 4.9 http://www.pastie.org/234633

```
# File active_record/base.rb
def attributes=(new_attributes, guard_protected_attributes = true)
  return if new_attributes.nil?
  attributes = new_attributes.dup
  attributes.stringify_keys!

  multi_parameter_attributes = []
  attributes = remove_attributes_protected_from_mass_assignment(attributes) if
guard_protected_attributes

  attributes.each do |k, v|
    k.include?("(") ? multi_parameter_attributes << [ k, v ] : send(k + "=", v)
  end

  assign_multiparameter_attributes(multi_parameter_attributes)
end
```

4.2.8 Reguläre Ausdrücke

Ruby bietet viele flexible Möglichkeiten, um mit regulären Ausdrücken zu arbeiten. Die folgenden Beispiele zeigen die gängigsten Vorgehensweisen, wobei auf eine Einführung in reguläre Ausdrücke verzichtet wird.

Vergleiche von Strings mit Mustern lassen sich beispielsweise folgendermaßen umsetzen.

```
puts "passt" if "Ruby" =~ /(ruby|smalltalk)/i
```

Wichtiges Kennzeichen für die Verwendung von regulären Ausdrücken zum Vergleich ist die Tilde nach dem Gleichheitszeichen sowie das Muster innerhalb der beiden Slash-Zeichen.

```
"Ruby\nist\ncool" =~ /Ruby\s+ist\s+(\w+)/

puts $1
```

Mit diesem Ausdruck wird das Wort `cool` aus dem String extrahiert und in der Variablen `$1` abgelegt.

Das folgende Beispiel zeigt, wie mit der Methode `match` der Klasse `String` eine Reihe von Variablen definiert und initialisiert werden können.

Listing 4.10 http://www.pastie.org/234634

```
text = "Ich finde Ruby toll"
m, wer, verb, was, wie = *text.match(/^(\w+)\s+(\w+)\s+(\w+)\s+(\w+)/)
puts "Kompletter match: #{m}"
puts "Variable wer: #{wer}"
puts "Variable verb: #{verb}"
puts "Variable was: #{was}"
puts "Variable wie #{wie}"
```

Das obige Beispiel ist absolut selbsterklärend, weshalb ich an dieser Stelle nicht weiter beschreibe, was dort geschieht.

4.2.9 Typen herausfinden

Jedes Objekt besitzt einen Klassentyp, der ebenfalls bei Bedarf abgefragt werden kann. Hierzu stehen verschiedene Möglichkeiten zur Verfügung.

Jedes Objekt kennt die Methode `type`, mit der der Klassentyp zurückgegeben wird. Allerdings ist die Methode `type` als Deprecated markiert, was genau wie in der Java-Welt bedeutet, dass man vom Einsatz absehen und die empfohlene Alternative `class` verwenden sollte. Das folgende Beispiel zeigt, wie eine entsprechende Warnung erzeugt werden kann.

```
>> 2.type
(irb):1: warning: Object#type is deprecated; use Object#class
=> Fixnum
>> 2.class
=> Fixnum
```

Hier wird der Typ nach der Konvertierung in einen `String` untersucht.

```
>> 2.to_s.class
=> String
```

4.2.10 Subklassen

Auch in Ruby können Klassen von anderen Klassen erben. Angenommen, Sie haben eine Klasse `Person` in der Datei `person.rb` implementiert, die so aussieht.

Listing 4.11 http://www.pastie.org/234635

```
class Person
  # Konstruktor
  def initialize(name)
    # Instanzvariable
    @name = name
  end

  def say_hello # Instanzmethode
    puts "#{@name} sagt hallo."
  end
end
```

Dann könnten Sie eine Klasse `Programmer` auf die folgende Art ableiten.

Listing 4.12 http://www.pastie.org/234636

```
require 'person'
# Programmer ist eine Person und erweitert
# sie mit mit zusätzlichen Merkmalen
class Programmer < Person
  def initialize(name, favorite_ide)
    super(name)
    @favorite_ide = favorite_ide
  end
  # say_hello wird hier überschrieben
```

```
  def say_hello
    super
    puts "Bevorugte IDE ist #{@favorite_ide}"
  end
end

helge = Programmer.new("Helge", "Textmate")
helge.say_hello
```

Die Ausgabe des Programms sieht dann so aus.

```
Helge sagt hallo.
Bevorugte IDE ist Textmate
```

4.2.11 Mehr Wissenswertes zu Methoden

In Bezug auf Methoden wäre allgemein noch das eine oder andere zu sagen. Beispielsweise sind runde Klammern bei Methodenaufrufen optional, wobei die meisten Programmierer diese optionalen Klammern nicht verwenden.

Methoden haben immer einen Empfänger, sprich: ein Objekt, das implizit `self` sein kann.

Des Weiteren werden Methoden nur mittels ihres Namens identifiziert, weshalb ein Überladen durch unterschiedliche Methodensignaturen, wie Sie es aus Java kennen, nicht möglich ist.

Um den Scope von Methoden einzuschränken, können Methoden `public`, `protected` oder `private` sein.

Übergabeparameter können in Ruby Standardwerte bekommen, wie das folgende Beispiel zeigt.

```
def my_method(a, b = {})
```

Im obigen Beispiel bekommt der Parameter `b` ein leeres `Hash`-Objekt als Standardwert zugewiesen.

Der zuletzt in einer Methode ausgewertete Ausdruck ist zugleich der Rückgabewert der Methode.

Und last but not least sind die geschweiften Klammern bei `Hash`-Objekten optional, wenn das `Hash`-Objekt als letzter Parameter eines Methodenaufrufs übergeben wird.

4.2.11.1 Klassenmethoden definieren

Es gibt mehrere Arten, Klassenmethoden zu definieren. Diese drei Arten werden im folgenden Listing gezeigt.

Listing 4.13 http://www.pastie.org/234637

```
class Person
  def self.class_method
    puts "Klassenmethode class_method aufgerufen"
  end

  class << self
    def class_method2
      puts "Klassenmethode class_method2 aufgerufen"
```

```
    end
  end
end

class User < Person

end

class << User
  def class_method3
    puts "Klassenmethode class_method3 aufgerufen"
  end
end
```

Die Methode `class_method` wird durch das Präfix `self` als Klassenmethode deklariert, während die Methode `class_method2` innerhalb eines Blocks definiert wird. In der Subklasse `User` werden zunächst keine Klassenmethoden definiert. Dies geschieht in einem Block für die Klasse `User`. Das Ergebnis kann mittels der drei nachfolgenden Methodenaufrufe überprüft werden.

```
User.class_method
User.class_method2
User.class_method3
```

Die Ausgabe der obigen Zeilen sollte nun folgendermaßen aussehen.

```
Klassenmethode class_method aufgerufen
Klassenmethode class_method2 aufgerufen
Klassenmethode class_method3 aufgerufen
```

4.2.11.2 Singleton-Methoden

Es ist möglich, Klassenmethoden auf Instanzen zu definieren, die dann als Singleton-Methoden fungieren. Das folgende Beispiel zeigt diese Möglichkeit.

Listing 4.14 http://www.pastie.org/234638

```
class Person
  # Konstruktor
  def initialize(name)
    # Instanzvariable
    @name = name
  end

  def say_hello # Instanzmethode
    puts "#{@name} sagt hallo."
  end
end
# Jedes Objekt hat zwei Klassen: Der Typ der Instanz
# und eine Singleton-Klasse. Methoden der Singleton-
# Klasse werden Singleton-Methoden genannt und können
# nur auf einem bestimmten Objekt aufgerufen werden.
michael = Person.new("Michael")
def michael.michael_say_hello
  puts "Michael says hello"
end
michael.michael_say_hello
# Klassenmethoden sind Singleton-Methoden des
# Klassenobjektes und können folgendermaßen definiert
# werden
def Person.count
  @@count
end
```

4.2.12 Zuweisungen

Es gibt noch einige erwähnenswerte Zuweisungen, die Sie im Folgenden sehen. Hier werden die Werte der beiden Variablen vertauscht.

Zunächst wird der Variablen `a` der Wert `1` zugewiesen.

```
irb(main):003:0> a = 1
=> 1
```

Als Nächstes erhält die Variable `b` den Wert `2`.

```
irb(main):004:0> b = 2
=> 2
```

Danach werden die Werte vertauscht.

```
irb(main):005:0> b, a = a, b
=> [1, 2]
```

Und schließlich werden die beiden Variablen zur Kontrolle angezeigt.

```
irb(main):006:0> a
=> 2
irb(main):007:0> b
=> 1
```

Die folgenden Zeilen zeigen eine Abkürzung, um verschiedenen Variablen den gleichen Wert zuzuweisen.

```
a = 1; b = 1
a = b = 1
```

Die folgende Schreibweise ist auch in Java möglich und addiert einen Wert mit einem vorhandenen Wert.

```
a += 1 # a = a + 1
```

Im nachfolgenden Beispiel wird ein `Array` verwendet, um den Variablen `a` und `b` einen Wert zuzuweisen.

```
a, b = [1, 2]
```

Die nächste Zeile weist der Variablen `a` den Wert aus `b` oder `c` zu. Wenn beispielsweise `b` `nil` ist und `c` einen gültigen Wert enthält, wird `c` zugewiesen.

```
a = b || c
```

In der letzten Zeile wird eine Art caching realisiert, indem `a` nur den Wert `b` zugewiesen bekommt, wenn `a` noch nicht definiert wurde.

```
a ||= b
```

4.2.12.1 Logische Zuweisung

Das nächste Beispiel zeigt, wie eine komplexe `if`-Abfrage vereinfacht geschrieben werden kann.

Listing 4.15 http://www.pastie.org/234639

```
# Etwas sehr viel zu lesen
user_id = nil
if comments
  if comments.first
    if comments.first.user
      user_id = comments.first.user.id
    end
  end
end

# Prägnanter:
user_id = comments &&
          comments.first &&
          comments.first.user &&
          comments.first.user.id
```

4.2.13 Module – MixIns

Ruby kennt keine Mehrfachvererbung, was in Java ja ebenfalls kein Feature ist. Anstelle von Mehrfachvererbung können in Ruby Module implementiert werden, die eine Klasse dann einbinden kann.

Dabei beschreiben Module Merkmale und Eigenschaften von Objekten oder Entitäten und tragen als Namen Adjektive wie beispielweise `Comparable`, `Enumerable` etc. Klassen können beliebig viele Module einbinden, allerdings lassen sich von Modulen keine Instanzen erstellen. Das nächste Listing zeigt ein Beispiel für ein Modul.

Listing 4.16 http://www.pastie.org/234640

```
# Mixins - Anstelle von Mehrfachvererbung
module FullName
  def full_name
    "#{@first_name} #{@last_name}"
  end
end

class Person
  include FullName

  def initialize(firstname, lastname)
    @first_name = firstname
    @last_name = lastname
  end
end

puts Person.new("Michael", "Johann").full_name
```

Das Modul definiert eine Methode `full_name`, die die beiden Instanzvariablen `@first_name` und `@last_name` in einen String zusammenfasst und als Rückgabewert zurückliefert. Durch die `include`-Anweisung innerhalb der Klasse `Person` wird die Funktionalität des Moduls `FullName` der Klasse `Person` zur Verfügung gestellt.

Mit dieser Vorgehensweise werden Mixins erstellt, die eine Mehrfachvererbung nachahmen können. Im nächsten Abschnitt wird ein Modul zur Bereitstellung eines Namensraumes (Namespace) genutzt.

4.2.14 Module – Namespace

Namensräume sind eine gute Möglichkeit, Klassen, die von der Namensgebung schnell zu Konflikten führen können, besser voneinander zu unterscheiden. Ähnlich wie in Java die Packagenamen zu solch einer Unterscheidung führen, nutzt Ruby Module, in denen Klassen definiert werden. Das folgende Beispiel zeigt, wie dies funktioniert.

Listing 4.17 http://www.pastie.org/234641

```
# Namespaces
module  MyApp
  class Person
    attr_accessor :name

    def initialize(name)
      self.name = name
    end
  end
end

MyApp::Person.new("Steve Jobs")
```

Damit kann die Klasse Person außerhalb des Moduls MyApp nur angesprochen werden, wenn der komplette qualifizierte Name angegeben wird.

4.2.15 Introspektion

Sie kennen aus dem Java-Umfeld die Reflection API, die eine Introspektion Ihrer Objekte und Klassen erlaubt. Da Ruby ebenfalls über Reflection-Mechanismen verfügt, ist es hier ebenfalls möglich, die Eigenschaften und Fähigkeiten von Ruby-Klassen zur Laufzeit zu untersuchen. Für eine Übersicht der wichtigsten Methoden sollten Sie sich einmal die nachfolgenden Beispiele ansehen und in der Interactive Ruby Shell ausprobieren.

In der ersten Zeile wird eine Instanz der Klasse `Fixnum` angelegt.

```
irb(main):002:0> michael = Person.new("Michael")

=> #<Person:0x11339e4 @name="Michael">
```

Dann wird mit `inspect` der Inhalt der Klasse auf der Konsole ausgegeben.

```
irb(main):003:0> michael.inspect

=> "#<Person:0x11339e4 @name="Michael">"
```

Nun kann der Name der Klasse für das Objekt `michael` ausgegeben werden.

```
irb(main):004:0> michael.class

=> Person
```

Jetzt wird der Name der Superklasse ermittelt.

```
irb(main):005:0> michael.class.superclass

=> Object
```

Wenn keine weitere Superklasse vorhanden ist, wird `nil` zurückgegeben, wie das nächste Beispiel zeigt.

```
irb(main):006:0> michael.class.superclass.superclass

=> nil
```

Auf einem Objekt kann man sich mit einem Aufruf der Methode `instance_variables` die Instanzvariablen ausgeben lassen.

```
irb(main):007:0> michael.instance_variables

=> ["@name"]
```

Oder die Liste der Instanzmethoden der Klasse `Person`.

```
irb(main):008:0> michael.class.instance_methods

=> ["inspect", "handling_jruby_bug", "v", "clone", "pretty_inspect", "pub-
lic_methods", "instance_variable_defined?", "pretty_print", "display", "equal?",
"freeze", "methods", "history_to_vi", "respond_to?",
"pretty_print_instance_variables", "dup", "instance_variables", "__id__", "eql?",
"ri", "id", "singleton_methods", "hvi", "send", "taint", "method", "h", "frozen?",
"h!", "instance_variable_get", "po", "__send__", "instance_of?", "to_a", "his-
tory_write", "gem", "type", "protected_methods", "instance_eval", "==", "===",
"verbose", "quiet", "instance_variable_set", "kind_of?", "extend", "to_s", "ob-
ject_id", "require", "class", "hash", "history", "private_methods", "=~",
"tainted?", "pretty_print_cycle", "untaint", "nil?", "history_do",
"pretty_print_inspect", "q", "is_a?", "poc"]
```

Die Liste der Module die eine Klasse eingebunden hat kann mit einem Aufruf der Methode `ancestors` ausgegeben werden.

```
irb(main):010:0> Person.ancestors

=> [Person, Object, Wirble::Shortcuts, PP::ObjectMixin, Kernel]
```

4.2.16 Nützliches zur Klasse String

Weitere nützliche Features der `String`-Klasse sind beispielsweise die Methoden zur Manipulation von Strings.

Das folgende Beispiel zeigt die Methoden `upcase` und `capitalize`.

```
irb(main):001:0> "ruby".upcase + " " + "rails".capitalize

=> "RUBY Rails"
```

Inhalte können mit dem Hash-Zeichen abgefragt und in andere Strings eingebettet werden. Das nächste Beispiel fragt den aktuellen Zeitstempel ab und fügt das Ergebnis in einen `String` ein.

```
irb(main):005:0> puts "time is: #{Time.now}\n second line"

time is: Thu Jun 12 20:37:48 +0200 2008
```

```
 second line
=> nil
```

Wäre der obige `String` nicht in Anführungszeichen gefasst, würde der Ausdruck `Time.now` nicht ausgewertet werden, und die Ausgabe würde so aussehen.

```
irb(main):007:0> puts 'time is: #{Time.now}\n second line'
time is: #{Time.now}\n second line
=> nil
```

Strings können mit `<<` aneinandergereiht werden. Allerdings ist diese Variante von der Ruby-internen Verarbeitung her um einiges langsamer als die vorherige.

```
irb(main):009:0> "I" << " love" << " Ruby"
=> "I love Ruby"
```

Das Einbetten in Anführungszeichen (Quotes) kann auch aus Gründen der Übersichtlichkeit folgendermaßen realisiert werden.

```
irb(main):010:0> %Q{"C" oder "Java"}
=> ""C" oder "Java""
```

Einfache Hochkommata werden mit folgender Anweisung erstellt. Der Unterschied ist hier das kleine q anstelle des entsprechenden Großbuchstabens.

```
irb(main):011:0> %q{'single 'quoted''}
=> "'single 'quoted'"
```

Ein letztes interessantes Feature von Ruby ist die Definition von Labels als Markierung für das Ende einer `String`-Definition.

Mit der folgenden Konstruktion kann ein beliebig umbrochener Text als ein `String`-Objekt definiert werden.

Listing 4.18 http://www.pastie.org/234642

```
puts <<-END
A here
document at #{Time.now}
END
```

Dabei wird der String mit `<<-` eingeleitet und mit dem Namen `END`, der ein beliebiger Name sein kann, abgeschlossen.

4.2.17 Weitere Features von Array

Die Klasse `Array` wartet mit weiteren nützlichen Funktionen auf. Wie bereits dargestellt wurde, zeigt die Verwendung von eckigen Klammern das Array einer Variablen an.

```
irb(main):001:0> a = ["Ruby", 99, 3.14]
=> ["Ruby", 99, 3.14]
```

Der Zugriff auf die Elemente eines Arrays wird über den Index, beginnend mit dem Wert `0`, getätigt. Ein Beispiel:

```
irb(main):002:0> a[1] == 99
=> true
```

Ebenso wie die Klasse `String` lässt sich auch ein Array mit `<<` auffüllen.

```
irb(main):003:0> a << "Rails"
=> ["Ruby", 99, 3.14, "Rails"]
```

Ein Array kann auch mit folgender Vorgehensweise erstellt werden.

```
irb(main):004:0> ['C', 'Java', 'Ruby'] == %w{C Java Ruby}
=> true
```

Die Methode `map!` der Klasse `Array` kann durch die einzelnen Elemente iterieren und einen Block ausführen. Dabei wird der im Block manipulierte Eintrag wieder an die ursprüngliche Stelle im Array zurückgeschrieben.

```
irb(main):009:0> a.map! {|x| x**2}
=> [1, 4, 9]
irb(main):010:0> a
=> [1, 4, 9]
```

Mit der Methode `select` lässt sich ein Block übergeben, der eine Bedingung überprüft und bei `true` den entsprechenden Wert zurückgibt.

```
irb(main):023:0> [1, 2, 3].select { |x| x % 2 == 0 }
=> [2]
```

Die Methode `reject` gibt alle Elemente zurück, für die die Bedingung nicht `true` ist.

```
irb(main):024:0> [1, 2, 3].reject { |x| x < 3 }
=> [3]
```

Mit `flatten` kann ein `Array` mit einem enthaltenen `Array` ausgeflacht werden, so dass folgendes `Array` zurückgegeben wird.

```
irb(main):028:0> [1, [2, 3]].flatten! # => [1, 2, 3]
=> [1, 2, 3]
```

Die Methode `include?` prüft, ob das übergebene Objekt im `Array` enthalten ist.

```
irb(main):032:0> language = "C++"
=> "C++"
irb(main):033:0> raise "Invalid language" if !%w{C Java Ruby}.include?(language)
RuntimeError: Invalid language
     from (irb):33
```

4.2.18 Symbole

Symbole sind Objekte von Klassen, die nur eine Instanz haben und mit einem Doppelpunkt vor dem Namen defniert werden. Zuweisungen an Symbole sind nicht möglich.

```
irb(main):001:0> :action.class == Symbol
=> true
```

Im obigen Beispiel wird das Symbol `:action` definiert und dessen Klasse abgefragt und mit `Symbol` verglichen.

Eine Konvertierung nach `String` ist ebenfalls möglich:

```
irb(main):002:0> :action.to_s == "action"
=> true
```

Des Weiteren kann jeder zusammenhängende `String` in ein `Symbol` umgewandelt werden.

```
irb(main):003:0> :action == "action".to_sym
=> true
```

Zum Beweis, dass es nur eine Instanz eines Symbols gibt, soll folgender Vergleich durchgeführt werden.

```
irb(main):003:0> :action == "action".to_sym
=> true
```

Der Gegenbeweis bei Verwendung zweier `String`-Objekte bestätigt die Behauptung.

```
irb(main):005:0> 'action'.equal?('action')
=> false
```

Symbole finden allgemein als Schlüssel in `Hash`-Objekten Verwendung, wie folgendes Beispiel verdeutlicht.

```
link_to "Home", :controller => "home"
```

4.2.19 Mehr zu Methoden

Die folgenden Unterabschnitte offenbaren weitere generelle Informationen und Konventionen zu Methoden.

4.2.19.1 Anzahl Übergabeparameter

Wenn eine Methode eine beliebige Anzahl an Übergabeparametern entgegennehmen soll, wird dies durch das Zeichen `*` definiert.

```
def my_method(*args)
  ...
end
```

4.2.19.2 Array in Parameter umwandeln

Wenn einer Methode ein `Array` übergeben wird, kann die Methode dieses `Array` mit folgender Schreibweise in einzelne Parameter umbrechen.

```
def my_method([a, b]*)
...
end
```

4.2.19.3 Duck-Typing

In Beiträgen zu Ruby wird der Begriff Duck-Typing relativ häufig verwendet. Er bezeichnet eine Art Typisierung, die sich durch die Sicht auf ein Objekt ergibt. Dabei wird davon ausgegangen, dass es sich bei einem Objekt um eine Ente handelt, wenn sie quaken kann und wie eine Ente aussieht. Technisch gesehen wird ein Objekt gefragt, ob es auf eine bestimmte Methode antworten kann. Hierzu gibt es die Methode `respond_to?`, die im folgenden Beispiel vorgestellt wird.

```
object.respond_to?(:method_name)
```

Hierzu ein konkretes Beispiel. Das folgende `String`-Objekt wird gefragt, ob es die Methode `to_i` kennt.

```
irb(main):007:0> "Test".respond_to?(:to_i)

=> true
```

4.2.20 Die Klasse Range

Ein nützliches Feature ist die Klasse `Range`, die oft in Schleifen Verwendung findet. Damit die Nützlichkeit verständlich wird, soll folgendes Beispiel herangezogen werden.

```
irb(main):017:0> (1..5).each {|x| puts x**2}
1
4
9
16
25

=> 1..5
```

Ein weiteres Beispiel erstellt aus einer `Range`-Instanz ein `Array`.

```
irb(main):018:0> (1..5).to_a

=> [1, 2, 3, 4, 5]
```

Wird ein `Range`-Objekt mit drei Punkten definiert, dann wird das letzte Element ausgeschlossen.

```
irb(main):019:0> (1...5).each { |x| puts x**2}
1
4
9
16

=> 1....5
```

4.2.21 Das Schlüsselwort case

Mit `case` können, wie auch in Java, Bereiche abgefragt und danach, je nach Bedingung, Anweisungen ausgeführt werden. Das folgende Listing verdeutlicht den Einsatz.

Listing 4.19 http://www.pastie.org/234643

```
case x
  when 0
  when 1, 2..5
    puts "Second branch"
  when 6..10
    puts "Third branch"
  when *[11, 12]
    puts "Fourth branch"
  when String: puts "Fifth branch"
  when /\d+\.\d+/
    puts "Sixth branch"
  when x.downcase == "michael"
    puts "Seventh branch"
  else
    puts "Eight branch"
  end
```

Sie sehen im obigen Listing, dass mit `case` ein Objekt `x` evaluiert wird. Die entsprechenden Verzweigungen werden mit `when` eingeleitet und vergleichen praktisch den angegebenen Wertebereich mit dem Inhalt der Variablen `x`. Da `x` einen nicht näher spezifizierten Typ besitzt, kann natürlich auch ein `String`-Objekt in `x` enthalten sein. Mit `else` legt man fest, welche Anweisung in allen anderen Fällen ausgeführt wird.

4.2.22 Metaprogrammierung: Blöcke und Closures

Ruby bietet viele elegante Möglichkeiten der Metaprogrammierung, die in vielen Situationen für weniger und vor allem besser lesbaren Code sorgen können. Dabei werden drei unterschiedliche Konzepte angeboten, die wiederum diverse Varianten anbieten. Bevor ich auf die Eigenheiten eingehe, erkläre ich die Begriffe.

4.2.22.1 Blöcke

Blöcke sind Code, der innerhalb einer Markierung defniert wird. Für einzeilige Blöcke kommen geschweifte Klammern zum Einsatz, während mehrzeilige Blöcke zwischen die Schlüsselwörter `do` und `end` gefasst werden. Hier ein Beispiel für einen einzeiligen Block.

```
puts [1,2,3].map { |x| x**2 }
```

Für jedes Element im `Array` wird die Methode `map` aufgerufen und der Wert der Berechnung im Block ausgegeben. Das Ergebnis ist wieder ein `Array` mit den neuen Werten, die mittels `puts` auf der Konsole ausgegeben werden.

Das folgende Beispiel zeigt den Umgang mit einem mehrzeiligen Block.

Listing 4.20 http://www.pastie.org/234644

```
[1,2,3].each do |x|
  puts "Wert für x = #{x}"
  puts "#{x} zum Quadrat ist #{x**2}"
end
```

Die Ausgabe sieht folgendermaßen aus.

```
Wert für x = 1
1 zum Quadrat ist 1
Wert für x = 2
2 zum Quadrat ist 4
Wert für x = 3
3 zum Quadrat ist 9
```

Blöcke können auf Variablen des ihn definierenden Kontexts zugreifen. Das folgende Beispiel zeigt dies.

Listing 4.21 http://www.pastie.org/234645

```
def dreimal
  yield
  yield
  yield
end

x = 3
puts "Wert von x vorher: #{x}"
dreimal { x += 1 }
puts "Wert von x nachher: #{x}"
```

Damit wird die Methode `dreimal` mit der Übergabe eines einzeiligen Blocks aufgerufen und die Variable `x` bei jeder Blockausführung (`yield`) mit einem neuen Wert versehen. Die Werte von `x` werden dabei vor und nach dem Aufruf der Methode `dreimal` ausgegeben.

Die Ausgabe des obigen Beispiels sehen Sie hier.

```
Wert von x vorher: 3
Wert von x nachher: 6
```

Ein Block kann Variablen referenzieren, die im gleichen Kontext wie der Block definiert wurden. Der Kontext, in dem der Block aufgerufen wird, ist nicht für den Block sichtbar, was das folgende Beispiel zeigt.

Listing 4.22 http://www.pastie.org/234646

```
def dreimal
    x = 20
    yield
    yield
    yield
    puts "lokales x nach der dreifachen Ausführung des Blocks: #{x}"
end

x = 3
dreimal { x += 1 }
puts "x nach dem Aufruf: #{x}"
```

In der Methode `dreimal` wird eine lokale Variable `x` mit dem Wert `20` definiert. Der übergebene Block wird dreimal ausgeführt und danach der Wert von `x` ausgegeben. Dieser ist

immer noch `20`, denn der Block kennt nur sein eigenes `x`. Das eigene `x` hat nach dem Durchlauf den Wert `6`.

Ein Block kann immer nur auf Variablen des äußeren Kontexts zugreifen, wenn diese Variablen auch dort definiert wurden. Schauen Sie sich einmal das folgende Beispiel an.

Listing 4.23 http://www.pastie.org/234647

```
def dreimal
    x = 20
    yield
    yield
    yield
    puts "lokales x nach der dreifachen Ausführung des Blocks: #{x}"
end

dreimal do
    n = 5
    puts "Ist n dort definiert, wo es zum ersten Mal einen Wert bekommt?"
    puts "Ja." if defined? n
end

puts "Ist n im äußeren Kontext nach Verlassen des Blocks definiert?"
puts "Nein" unless defined? n
```

Die Methode `dreimal` erhält nun einen mehrzeiligen Block als Übergabeparameter. Die Ausgabe sieht wie folgt aus.

```
Ist n dort definiert, wo es zum ersten Mal einen Wert bekommt?
Ja.
Ist n dort definiert, wo es zum ersten Mal einen Wert bekommt?
Ja.
Ist n dort definiert, wo es zum ersten Mal einen Wert bekommt?
Ja.
lokales x nach der dreifachen Ausführung des Blocks: 20
Ist n im äußeren Kontext nach Verlassen des Blocks definiert?
Nein
```

Damit ist klar, dass der Block `n` nur aus dem eigenen Kontext kommt.

4.2.22.2 Closures

Ein Closure ist ein Code-Block mit den folgenden Eigenschaften:

- Der Block kann als Wert übergeben werden.
- Der Block kann bei Bedarf an der Stelle ausgeführt werden, wo der Block verwendet wird.
- Der Block kann auf Variablen des Kontexts zugreifen, in dem er erzeugt wurde.

Closures kommen aus der Welt der funktionalen Programmiersprachen, finden aber immer mehr Verbreitung in dynamischen Sprachen wie Ruby und Groovy. Auch Java kennt ein solches Konstrukt in Form von Inner-Classes, doch ist die Mächtigkeit in Ruby um einiges größer.

Bleiben wir bei der Methode `dreimal` aus dem vorhergehenden Abschnitt. Blöcke sind im Grunde ähnlich wie Closures. Sie sind in sich abgeschlossen und haben nur Zugriff auf Variablen des eigenen Kontexts. Allerdings kann ein Block nur von `yield` ausgeführt und

nicht weitergegeben werden. Ruby kennt allerdings ein Konstrukt, mit dem sich ein Block wie eine lokale Variable weiterreichen lässt. Das folgende Beispiel verdeutlicht die Verwendung.

Listing 4.24 http://www.pastie.org/234648

```
def dreimal
  yield
  yield
  yield
end

def sechsmal(&block)
  dreimal(&block)
  dreimal(&block)
end

x = 3
sechsmal { x += 1 }
puts "Wert von x: #{x}"
```

Damit wird die Methode dreimal zweimal hintereinander aufgerufen, und zwar mit dem als Parameter übergebenen Block. Das Ergebnis für `x` ist danach `9`.

Mit diesem Konstrukt haben wir fast schon ein vollwertiges Closure, wenn es nicht das Problem gäbe, wonach der Block keiner Variablen zugewiesen werden kann, um den Block an späterer Stelle ausführen zu können. Das folgende Beispiel lässt sich nicht umsetzen.

```
def save_block(&block)
  saved = &block
end
```

Zur Laufzeit gibt es deshalb folgende Fehlermeldung.

```
syntax error, unexpected tAMPER
  saved = &block
```

Abhilfe schafft die Nutzung einer Instanzvariablen und das Weglassen des `&`-Zeichens.

```
def save_block(&block)
  @saved = block
end
```

Damit kann nun der Block mit Hilfe der Instanzvariablen aufgerufen werden, wie im folgenden Beispiel zu sehen ist.

```
save_block { puts "Hallo" }
@saved.call
@saved.call
```

Damit wird als Erstes die Methode `save_block` mit dem Block aufgerufen. Dabei passiert lediglich die Zuweisung des Blocks an die Instanzvariable `@saved`. In den beiden darauffolgenden Aufrufen wird der Block mittels der Methode `call` ausgeführt und jeweils der String `Hallo` ausgegeben.

Zu beachten ist an dieser Stelle, dass pro Methodenaufruf nur ein Block-Parameter übergeben werden kann. Zudem muss dies der letzte Parameter in der Liste der Übergabewerte sein. Das folgende Beispiel zeigt, dass es nicht möglich ist, solche Konstrukte auszuführen.

Listing 4.25 http://www.pastie.org/234649

```
def save_block(&block, &nextone)
  @saved = block
  @next = nextone
end

save_block { puts "Hallo" } { puts "Next"}
@saved.call
@saved.call
```

Stattdessen wird folgende Fehlermeldung ausgegeben.

```
syntax error, unexpected ',', expecting ')'
def save_block(&block, &nextone)
```

Das Zuweisen des Blockarguments durch Weglassen des `&`-Zeichens ist übrigens ein Synonym für folgende Schreiweise.

```
@saved = Proc.new(&block)
```

Somit ist es möglich, einen Block direkt mit `Proc.new` zu definieren.

```
@proc = Proc.new { puts "Ich bin ein Closure" }
@proc.call
```

Nun existieren schon zwei Wege, um echte Closures zu konstruieren. Doch Ruby hat noch mehr Konstrukte auf Lager, die ich Ihnen im Folgenden vorstelle.

proc

Ein weiteres Konstrukt ist `proc`, das es ebenfalls wie `Proc.new` erlaubt ein Closure zu definieren. Zur Verdeutlichung soll folgendes Beispiel beitragen.

```
@proc_proc = proc { puts "Ich bin ein Closure mit proc erstellt" }
```

Die Ausführung mittels der Methode `call` auf dem Objekt `@proc_proc` führt das Closure aus.

```
>> @proc_proc.call

Ich bin ein Closure mit proc erstellt
=> nil
>>
```

lambda

Mit `lambda` wird ebenfalls ein Closure definiert, wie im nächsten Beispiel zu sehen ist.

```
@proc_lambda = lambda { puts "Ich bin ein Closure mit lambda erstellt" }
```

Auch hier kann mit der Methode `call` auf dem Objekt `@proc_lambda` die Ausführung des Closures erfolgen.

```
>> @proc_lambda.call

Ich bin ein Closure mit lambda erstellt
=> nil
>>
```

method

Als letztes Konstrukt steht uns `method` zur Verfügung, bei dem eine Methode als Closure definiert und mittels `call` ausgeführt werden kann. Das folgende Beispiel zeigt dies.

```
def diese_methode
  puts "Ich bin ein Closure mit method erstellt."
end

@proc_method = method :diese_methode
@proc_method.call
```

Bei oberflächlicher Betrachtung der letzten vier Konstrukte drängt sich schnell der Verdacht auf, dass es sich hierbei um die gleiche Funktionsweise handelt, nur unter anderem Namen. Das ist nicht richtig, wenn man sich das Verhalten bei `return`-Anweisungen näher betrachtet. Hierzu zeige ich Ihnen zuerst ein Beispiel mit ganz normalem Return-Verhalten, ohne die explizite Verwendung von `return`.

Listing 4.26 http://www.pastie.org/234650

```
def doit(closure)
  puts "Rufe sogleich closure auf"
  ret = closure.call
  puts "Closure gab #{ret} zurück"
  "Rückgabewert der Methode doit"
end

def diese_methode
  puts "Wert aus method"
end

puts "doit gab zurück: " + doit(Proc.new {"Wert aus Proc.new"})
puts "doit gab zurück: " + doit(proc {"Wert aus proc"})
puts "doit gab zurück: " + doit(lambda {"Wert aus lambda"})
puts "doit gab zurück: " + doit(method(:diese_methode))
```

Die drei verwendeten Konstrukte `Proc.new`, `proc` und `lambda` verhalten sich in diesem Beispiel vollkommen gleich und geben Folgendes aus.

```
Rufe sogleich closure auf
Closure gab Wert aus Proc.new zurück
doit gab zurück: Rückgabewert der Methode doit
Rufe sogleich closure auf
Closure gab Wert aus proc zurück
doit gab zurück: Rückgabewert der Methode doit
Rufe sogleich closure auf
Closure gab Wert aus lambda zurück
doit gab zurück: Rückgabewert der Methode doit
Rufe sogleich closure auf
Wert aus method
Closure gab  zurück
doit gab zurück: Rückgabewert der Methode doit
```

Das eigentliche Problem mit diesen Konstrukten ist ein expliziter Aufruf von `return`. Das folgende Beispiel verdeutlicht Konstrukt eins, bei dem ein `Proc.new` verwendet wird.

Listing 4.27 http://www.pastie.org/234651

```
def doit(closure)
  puts "Rufe sogleich closure auf"
  ret = closure.call
  puts "Closure gab #{ret} zurück"
  "Rückgabewert der Methode doit"
```

```
end

begin
  doit(Proc.new { return "Wert aus Proc.new"})
rescue Exception => e
  puts "Fehler mit #{e.class}: #{e}"
end
```

Folgende Ausgabe wird erzeugt.

```
Rufe sogleich closure auf
Fehler mit LocalJumpError: unexpected return
```

Das Problem ist hier, dass `return` nur innerhalb einer echten Funktion oder Methode verwendet werden kann, was sich von einem `Proc`-Objekt nicht behaupten lässt. Alternativ kann man nun eine Methode verwenden, um das Verhalten zu überprüfen.

Listing 4.28 http://www.pastie.org/234652

```
def doit(closure)
  puts "Rufe sogleich closure auf"
  ret = closure.call
  puts "Closure gab #{ret} zurück"
  "Rückgabewert der Methode doit"
end

def dothis
    result = doit(Proc.new { return "Wert aus Proc.new" })
    puts "doit gab zurück: " + result
    "Rückgabewert aus dothis"
end

puts "dothis gab #{dothis} zurück"
```

Die Ausgabe hiervon sieht so aus.

```
Rufe sogleich closure auf
dothis gab Wert aus Proc.new zurück
```

Überraschend ist hier, dass die `return`-Anweisung wie eine Exception funktioniert und die weitere Ausführung abgebrochen wird. Sowohl in der Methode `doit` als auch in `dothis` wird zum Aufrufer zurückgekehrt. Damit ist ein Closure aus `Proc.new` nicht wirklich in sich geschlossen, weil es abhängig vom aufrufenden Kontext ist, den `return` benötigt.

Das steht im Gegensatz zu `lambda`, das wie erwartet mit `return`-Anweisungen umgehen kann, wie das folgende Beispiel verdeutlicht.

Listing 4.29 http://www.pastie.org/234655

```
def doit(closure)
  puts "Rufe sogleich closure auf"
  ret = closure.call
  puts "Closure gab #{ret} zurück"
  "Rückgabewert der Methode doit"
end

def dothis
    result = doit(lambda { return "Wert aus lambda" })
    puts "doit gab zurück: " + result
    "Rückgabewert aus dothis"
end

puts "dothis gab #{dothis} zurück"
```

Die nachfolgende Ausgabe entspricht nun den Erwartungen.

```
Rufe sogleich closure auf
Closure gab Wert aus lambda zurück
doit gab zurück: Rückgabewert der Methode doit
dothis gab Rückgabewert aus dothis zurück
```

Damit ist klar, dass ein `return` innerhalb eines Closures mit `lambda` lediglich den Block verlässt und in Ruby das einzig *wahre* Konstrukt für die Definition von Closures darstellt.

4.2.22.3 Die Methode arity

Ein weiteres Unterscheidungsmerkmal für Closures in Ruby ist die Methode `arity`, die neben `call` für Closures zur Verfügung steht und die Anzahl der erwarteten Parameter zurückliefert. Entscheidend ist hier das Verhalten bei fehlenden Parametern. Hierzu ein kleines Beispiel.

Listing 4.30 http://www.pastie.org/234656

```
puts "Lambda mit einem Argument"
puts lambda { |x| }.arity
puts "Lambda mit drei Argumenten"
puts lambda { |a, b, c| }.arity
```

Die Ausgabe dieser beiden Aufrufe sieht erwartungsgemäß so aus.

```
Lambda mit einem Argument
1
Lambda mit drei Argumenten
3
```

Was passiert, wenn hier die falsche Anzahl Parameter übergeben wird? Das folgende Beispiel demonstriert, was dann geschieht.

Listing 4.31 http://www.pastie.org/234658

```
def aufruf_zu_viele_parameter(closure)
    begin
        puts "Erwartete Parameter: #{closure.arity}"
        closure.call(1,2,3,4,5,6)
        puts "Zu viele Parameter"
    rescue Exception => e
        puts "Fehler: #{e.class}: #{e}"
    end
end

def zwei_parameter(x,y)
end

puts; puts "Proc.new:"; aufruf_zu_viele_parameter(Proc.new {|x,y|})
puts; puts "proc:"    ; aufruf_zu_viele_parameter(proc {|x,y|})
puts; puts "lambda:"  ; aufruf_zu_viele_parameter(lambda {|x,y|})
puts; puts "Method:"  ;aufruf_zu_viele_parameter(method(:zwei_parameter))
```

Damit werden folgende Ausgaben erzeugt.

```
Proc.new:
Erwartete Parameter: 2
Zu viele Parameter

proc:
```

```
Erwartete Parameter: 2
Fehler: ArgumentError: wrong number of arguments (6 for 2)

lambda:
Erwartete Parameter: 2
Fehler: ArgumentError: wrong number of arguments (6 for 2)

Method:
Erwartete Parameter: 2
Fehler: ArgumentError: wrong number of arguments (6 for 2)
```

So weit ist alles in Ordnung. Das Laufzeitsystem moniert, dass sechs Parameter übergeben und lediglich zwei Parameter erwartet wurden.

Das folgende Beispiel demonstriert, was passiert, wenn weniger Parameter als erwartet übergeben werden.

Listing 4.32 http://www.pastie.org/234659

```
def aufruf_zu_wenig_parameter(closure)
    begin
        puts "Erwartete Parameter: #{closure.arity}"
        closure.call()
        puts "Zu wenig Parameter"
    rescue Exception => e
        puts "Fehler: #{e.class}: #{e}"
    end
end

puts; puts "Proc.new:"; aufruf_zu_wenig_parameter(Proc.new {|x,y|})
puts; puts "proc:"    ; aufruf_zu_wenig_parameter(proc {|x,y|})
puts; puts "lambda:"  ; aufruf_zu_wenig_parameter(lambda {|x,y|})
puts; puts "Method:"  ;aufruf_zu_wenig_parameter(method(:zwei_parameter))
```

Die folgende Ausgabe ist ebenfalls erwartungsgemäß und verdeutlicht, dass zu wenige Parameter übergeben wurden.

```
Proc.new:
Erwartete Parameter: 2
Zu wenig Parameter

proc:
Erwartete Parameter: 2
Fehler: ArgumentError: wrong number of arguments (0 for 2)

lambda:
Erwartete Parameter: 2
Fehler: ArgumentError: wrong number of arguments (0 for 2)

Method:
Erwartete Parameter: 2
Fehler: ArgumentError: wrong number of arguments (0 for 2)
```

Eine weitere Überraschung folgt, wenn lediglich ein Parameter erwartet wird.

Listing 4.33 http://www.pastie.org/234660

```
def aufruf_zu_viele_parameter(closure)
    begin
        puts "Erwartete Parameter: #{closure.arity}"
        closure.call(1,2,3,4,5,6)
        puts "Zu viele Parameter"
    rescue Exception => e
        puts "Fehler: #{e.class}: #{e}"
    end
```

```
end

def zwei_parameter(x,y)
end

def aufruf_zu_wenig_parameter(closure)
    begin
        puts "Erwartete Parameter: #{closure.arity}"
        closure.call()
        puts "Zu wenig Parameter"
    rescue Exception => e
        puts "Fehler: #{e.class}: #{e}"
    end
end

def ein_parameter(x)
end

puts; puts "Proc.new:"; aufruf_zu_viele_parameter(Proc.new {|x|})
puts; puts "proc:"    ; aufruf_zu_viele_parameter(proc {|x|})
puts; puts "lambda:"  ; aufruf_zu_viele_parameter(lambda {|x|})
puts; puts "Method:"  ; aufruf_zu_viele_parameter(method(:ein_parameter))
puts; puts "Proc.new:"; aufruf_zu_wenig_parameter(Proc.new {|x|})
puts; puts "proc:"    ; aufruf_zu_wenig_parameter(proc {|x|})
puts; puts "lambda:"  ; aufruf_zu_wenig_parameter(lambda {|x|})
puts; puts "Method:"  ; aufruf_zu_wenig_parameter(method(:ein_parameter))
```

Das Ergebnis der Ausgabe wird von Warnungen begleitet, wie folgende Ausgabe verdeutlicht.

```
Proc.new:
Erwartete Parameter: 1
Zu viele Parameter

proc:
Erwartete Parameter: 1
Zu viele Parameter

lambda:
Erwartete Parameter: 1
Zu viele Parameter

Method:
Erwartete Parameter: 1
Fehler: ArgumentError: wrong number of arguments (6 for 1)

Proc.new:
Erwartete Parameter: 1
Zu wenig Parameter

proc:
Erwartete Parameter: 1
Zu wenig Parameter

lambda:
Erwartete Parameter: 1
Zu wenig Parameter

Method:
Erwartete Parameter: 1
Fehler: ArgumentError: wrong number of arguments (0 for 1)
-:27: warning: multiple values for a block parameter (6 for 1)
     from -:4
-:28: warning: multiple values for a block parameter (6 for 1)
     from -:4
-:29: warning: multiple values for a block parameter (6 for 1)
     from -:4
-:31: warning: multiple values for a block parameter (0 for 1)
     from -:17
```

```
-:32: warning: multiple values for a block parameter (0 for 1)
    from -:17
-:33: warning: multiple values for a block parameter (0 for 1)
    from -:17
```

Weiterhin fällt auf, dass bei den meisten Aufrufen keine Fehlermeldung erscheint, wenn ein Closure mit nur einem Parameter definiert wurde. Ebenso ist das Verhalten unterschiedlich, wenn gar keine Parameter erwartet werden.

Hinweis:
Es fällt auf, dass lediglich `lambda` und `method` Konstrukte sind, die sicher und souverän für Closures verwendet werden können. Das `lambda`-Konstrukt ist nur zuverlässig, wenn mehr als ein Parameter Verwendung finden.

4.2.22.4 module_eval, class_eval, instance_eval und eval

Mit den Methoden `module_eval` und `class_eval`, wobei `class_eval` ein Alias-Name für `module_eval` ist, können einer Klasse oder einem Modul Methoden dynamisch und zur Laufzeit hinzugefügt werden.

Im nächsten Abschnitt lernen Sie `class_eval` näher kennen, weshalb an dieser Stelle lediglich das Vorhandensein dieser Methoden erläutert wird. `class_eval` kann einen Block übergeben bekommen, der dann auf die Klasse gemappt wird und somit neue Methoden für alle Instanzen der jeweiligen Klasse bereitstellt.

Hier ein Beispiel, in dem dynamische `hash_for`-Methoden bereitgestellt werden.

Listing 4.34 http://www.pastie.org/pastes/213801

```
module TestDataFactory
  def data_factory(*args)
    types = args.extract_options!
    args.map { |type| [type, type.to_s.classify.constantize] }.each { |type, klaz|
types[type] = klaz }

    types.each do |kind, klaz|
      class_eval <<-"end_eval", __FILE__, __LINE__
        def self.hash_for_#{kind}(*args)
          opts = args.extract_options!
          hash = @@hash_for_#{kind}.is_a?(Proc) ? @@hash_for_#{kind}.call :
@@hash_for_#{kind}
          hash.merge(opts)
        end

        def self.hash_for_#{kind}=(h)
          @@hash_for_#{kind} = h
        end
      end_eval
      delegate :"hash_for_#{kind}", :to => :klass

      define_method(:"create_#{kind}") do |*args|
        opts = args.extract_options!
        klaz.create send(:"hash_for_#{kind}").merge(opts)
      end
    end
  end
end
```

Wenn lediglich eine Instanz und ihr Kontext erweitert werden sollen, kommt die Methode `instance_eval` zum Einsatz. Mit `eval` können beliebige Ruby-Anweisungen ausgeführt und ausgewertet werden. Auch hier soll ein kleines Beispiel die Anwendung verdeutlichen.

```
var = nil
eval 'var = 1'
puts var # gibt 1 wieder
```

4.2.22.5 Und die Moral ...

Viele der vorgestellten Konstrukte lassen sich für Closures nur bedingt einsetzen. Dieser Umstand lässt vielleicht Zweifel an der *Schönheit* von Ruby aufkommen. Das folgende Beispiel zeigt allerdings, welche *coolen* Möglichkeiten Ruby mit Closures eröffnet.

Um Ihnen ein Praxisbeispiel zur Metaprogrammierung zu zeigen, stelle ich Ihnen ein Beispiel vor, das ein wenig an die Java-Welt angelehnt ist. Die Beispielklasse MetaProgramming implementiert die Methoden `add_change_listener` und `remove_change_listener`. Damit können Sie beliebige Klassen in ein `Array` einfügen.

Listing 4.35 http://www.pastie.org/234661

```
class MetaProgramming
  def initialize
    @listeners = []
  end

  def add_change_listener(listener)
    @listeners.push(listener) unless @listeners.include?(listener)
  end

  def remove_change_listener(listener)
    @listeners.delete(listener)
  end
end
```

So weit, so gut, mit der Liste der Listener passiert aber noch nichts.

Es soll die Möglichkeit geben, dass Subklassen der Klasse `MetaProgramming` über die Klassenmethode `property` eine Reihe von Eigenschaften übergeben kann, die dann mit einem Change-Listener überwacht werden können. Hierzu muss die Subklasse von `MetaProgramming` über entsprechende Getter- und Setter-Methoden für die jeweilige Property verfügen, die nun dynamisch zur Laufzeit bereitgestellt werden sollen. Damit dies funktioniert, muss die Klassenmethode `property` implementiert werden.

Listing 4.36 http://www.pastie.org/234662

```
def self.property(*properties)
  properties.each do |property|
    class_eval(%Q[
      def #{property}
        @#{property}
      end

      def #{property}=(value)
        old_value = @#{property}
        return if (value == old_value)
          @listeners.each do |listener|
```

```
            listener.property_changed(:#{property}, old_value, value)
          end
          @#{property} = value
        end
    ])
  end
end
```

Die in Fettschrift hervorgehobenen Zeilen sind Code, der mit der Methode `class_eval` ausgeführt wird. Demnach werden eine Getter- und eine Setter-Methode generiert. Die Setter-Methode ruft auf der Liste der Listener die Methode `property_changed` auf, um die Änderung einer Eigenschaft zu propagieren. Demnach muss eine Klasse, die sich als Change-Listener bei der Subklasse von `MetaProgramming` registriert, die Methode `property_changed` implementieren. Dies passiert in der nächsten Beispielklasse `Person`, die von `MetaProgramming` erbt.

```
class Person < MetaProgramming
  property :name, :age

  def property_changed(property, old_value, value)
    print "Property: ", property, " has changed from ",
     old_value, " to ", value, "\n"
  end
end
```

Hier sehen Sie, dass über die Klassenmethode `property` zwei Eigenschaften, nämlich `:name` und `:age`, angegeben werden. Intern werden hierzu die zuvor genannten Getter- und Setter-Methoden generiert. Die Methode `property_changed` gibt lediglich eine Meldung aus, wenn eine entsprechende Setter-Methode aufgerufen wurde. Ein Test könnte beispielsweise so aussehen.

Listing 4.37 http://www.pastie.org/234663

```
person = Person.new
person.add_change_listener(person)

person.name = "Horst Kevin"
person.age = 18
```

Die Ausgabe des Tests sieht dann folgendermaßen aus.

```
Property: name has changed from nil to Horst Kevin
Property: age has changed from nil to 18
```

Mit diesen Ausgaben ist das Kapitel zur Metaprogrammierung in Ruby abeschlossen. Weitere Informationen finden Sie beispielsweise im Quellcode zu Ruby on Rails.

4.2.23 Exceptions

Die Behandlung von Ausnahmefehlern gibt es nicht nur in Java, sondern natürlich auch in Ruby. Ähnlich wie `try-catch-finally`-Blöcke in Java gibt es in Ruby die `begin-rescue-ensure`-Blöcke. Hier zunächst ein Beipiel.

Listing 4.38 http://www.pastie.org/234664

```
begin
  raise(ArgumentError, "No file_name provided") if !file_name
  content = load_blog_data(file_name)
  raise "Content is nil" if !content
  rescue BlogDataNotFound
    STDERR.puts "File #{file_name} not found"
  rescue BlogDataConnectError
    @connect_tries ||= 1
    @connect_tries += 1
    retry if @connect_tries < 3
    STDERR.puts "Invalid blog data in #{file_name}"
  rescue Exception => exc
    STDERR.puts "Error loading #{file_name}: #{exc.message}"
    raise
end
```

Code, der eine oder mehrere Exceptions werfen kann, wird in Klammern von `begin` und `rescue` gesetzt. Im obigen Beispiel beginnt ein derartiger Block mit einer `If`-Abfrage. Ergibt die Abfrage `true`, so wird mit `raise` eine Exception vom Type `ArgumentError` geworfen. Bei Rückgabe von `false` wird die Methode `load_blog_data` ausgeführt, die ihrerseits wieder Exceptions werfen kann. Danach wird eine Exception geworfen, wenn `content` gleich `nil` ist.

Die `Rescue`-Blöcke fangen die jeweiligen Exceptions ab und gehen der Fehlerbehandlung nach. Mit einem `retry` kann versucht werden, den eigentlichen Block noch einmal auszuführen.

Ruby kennt eine Liste von Exceptions, die sich auch aktuell generieren lassen können. Hier zunächst die Liste der Exceptions meiner aktuellen Laufzeitumgebung.

```
Exception
  #<Class:0x8b844>::WindowNotFound
  NoMemoryError
  ScriptError
    LoadError
    NotImplementedError
    SyntaxError
  SignalException
    Interrupt
  StandardError
    ArgumentError
    IOError
      EOFError
    IndexError
    LocalJumpError
    NameError
      NoMethodError
    OSX::PropertyListError
    RangeError
      FloatDomainError
    RegexpError
    RuntimeError
    SecurityError
    SystemCallError
    SystemStackError
    ThreadError
    TypeError
    ZeroDivisionError
  SystemExit
  fatal
```

Das folgende Ruby-Skript von Nick Sieger habe ich im Internet gefunden[2] und will es Ihnen nicht vorenthalten. Es prüft, welche Klassen im `ObjectSpace` von `Exception` abgeleitet sind, und bereitet das Ergebnis – mit entsprechender Einrückung – übersichtlich auf.

Listing 4.39 http://www.pastie.org/234665

```
exceptions = []
tree = {}
ObjectSpace.each_object(Class) do |cls|
  next unless cls.ancestors.include? Exception
  next if exceptions.include? cls
  next if cls.superclass == SystemCallError # avoid dumping Errno's
  exceptions << cls
  cls.ancestors.delete_if {|e| [Object, Kernel].include? e }.reverse.inject(tree)
{|memo,cls| memo[cls] ||= {}}
end

indent = 0
tree_printer = Proc.new do |t|
  t.keys.sort { |c1,c2| c1.name <=> c2.name }.each do |k|
    space = (' ' * indent); space ||= ''
    puts space + k.to_s
    indent += 2; tree_printer.call t[k]; indent -= 2
  end
end
tree_printer.call tree
```

Alternativ kann mit folgendem Einzeiler eine Liste der Exceptions ausgegeben werden.

```
$ ruby -e 'ObjectSpace.each_object(Class) { |cls| puts cls if
cls.ancestors.include? Exception and cls.superclass != SystemCallError}'
```

Die Ausgabe dieser kleinen Variante sieht dann so aus.

```
SystemStackError
LocalJumpError
EOFError
IOError
RegexpError
FloatDomainError
ZeroDivisionError
ThreadError
SystemCallError
NoMemoryError
SecurityError
RuntimeError
NotImplementedError
LoadError
SyntaxError
ScriptError
NoMethodError
NameError
RangeError
IndexError
ArgumentError
TypeError
StandardError
Interrupt
SignalException
fatal
SystemExit
Exception
```

[2] http://blog.nicksieger.com/articles/2006/09/06/rubys-exception-hierarchy

4.2.24 Alias-Methoden

Mit Hilfe von Alias-Methoden ist es möglich, eine alternative Namensgebung von Methoden zu erreichen, wenn eine Klasse um eine Methode erweitert wird, die bereits existiert. Das folgende Beispiel zeigt, was gemeint ist.

Listing 4.40 http://www.pastie.org/234666

```
class Michael
  def say_hello
    puts "Hallo"
  end
end

class Michael
  alias_method :say_hello_orig, :say_hello
  def say_hello
    puts "Vor say hello"
    say_hello_orig
    puts "Nach say hello"
  end
end

m = Michael.new
m.say_hello
```

Die Klasse Michael defniert die Methode say_hello. In einer Erweiterung der Klasse Michael wird wieder eine Methode say_hello implementiert. Darin wird eigentlich die Methode say_hello aufgerufen, was allerdings zu einem Deadlock führen würde. Daher definiert die Klasse Michael einen Alias namens `say_hello_orig` für die Methode `say_hello`. Die Ausführung verdeutlicht den Verlauf der Abarbeitung.

```
Vor say hello
Hallo
Nach say hello
```

4.2.25 Offene Kernel-Klassen

Dass die Klassen in Ruby offen für Erweiterungen sind, zeigt das Framework Ruby on Rails auf beeindruckende Weise. Um die Erweiterbarkeit zu demonstrieren, soll das folgende Beispiel herangezogen werden.

Listing 4.41 http://www.pastie.org/234667

```
class Integer
  def hallo
    puts "Hallo"
  end
end

10.hallo
```

Im obigen Beispiel wird die Klasse Integer um die Methode hallo erweitert. Über die Sinnhaftigkeit dieser Erweiterung machen wir uns zunächst keine Gedanken. Fakt ist, dass eine Instanz von `Integer`, also beispielsweise die Zahl `10`, mit der Methode `hallo` den Text „Hallo“ ausgeben kann.

4.3 Zusammenfassung

In diesem Kapitel konnten Sie genügend Grundlagenwissen zur Sprache Ruby sammeln, um die nächsten Kapitel zu verstehen.

Im Einzelnen lernten Sie die folgenden Aspekte von Ruby kennen.

- Konventionen und Programmierrichtlinien
- Sprachkonzepte und vertiefende Informationen
- Methoden und Blöcke
- Metaprogrammierung