Skriptsprachen
* Ruby *

Container, Blöcke
&
Iteratoren

Naumann
Sommersemester 2016

4.1 Container

- Um größere Mengen von Daten zu verwalten eignen sich in Ruby besonders die zwei vordefinierten Klassen **Array** und **Hash**, die wir bereits in den vergangenen Sitzungen kennengelernt haben.

- Beide Klassen verfügen über ein komplexes Interface oder anders ausgedrückt: Für jede der beiden Klassen gibt es zahlreiche Methoden um ihre Instanzen zu manipulieren.

- Besonders durch die Kombination mit (Anweisungs)Blöcken lassen sich so mit minimalem Aufwand leistungsfähige Ruby-Programme schreiben.

4.1 Container

Arrays

Ein Array besteht aus einer Sammlung von Objektverweisen. Jeder Verweis belegt eine bestimmte Position im Array und wird durch eine Zahl (0) identifiziert.

```
a = [ 3.14159, "kuchen", 99 ]
a.class                    => Array
a.length                   => 3
a[0]                       => 3.14159
a[1]                       => "kuchen"
a[2]                       => 99
a[3]                       => nil
b = Array.new
b.class                    => Array
b.length                   => 0
b[0] = "zweiter"
b[1] = "array"
b                          => ["zweiter", "array"]
```

4.1 Container

Arrays

Arrays werden mit Hilfe des [ ]-Operators, der wie die meisten Operatoren in Ruby als Methode realisiert ist, indiziert. In Ruby ist es möglich, negative Zahlen als Array-Indices zu verwenden. In diesem Fall wird vom Ende des Arrays aus gezählt: Mit ‘-1‘ wird das letzte, mit ‘-2‘ das vorletzte Element des Arrays ausgewählt.

```
a = [ 1, 3, 5, 7, 9 ]
a[-1]          => 9
a[-2]          => 7
a[-99]         => nil
```

Durch Angabe einer zweiten Zahl ([Index, Anzahl]) kann man einen Abschnitt des Arrays auswählen:

```
a[1, 3]        => [3, 5, 7]
a[3, 1]        => [7]
a[-3, 2]       => [5, 7]
```

4.1 Container

Arrays

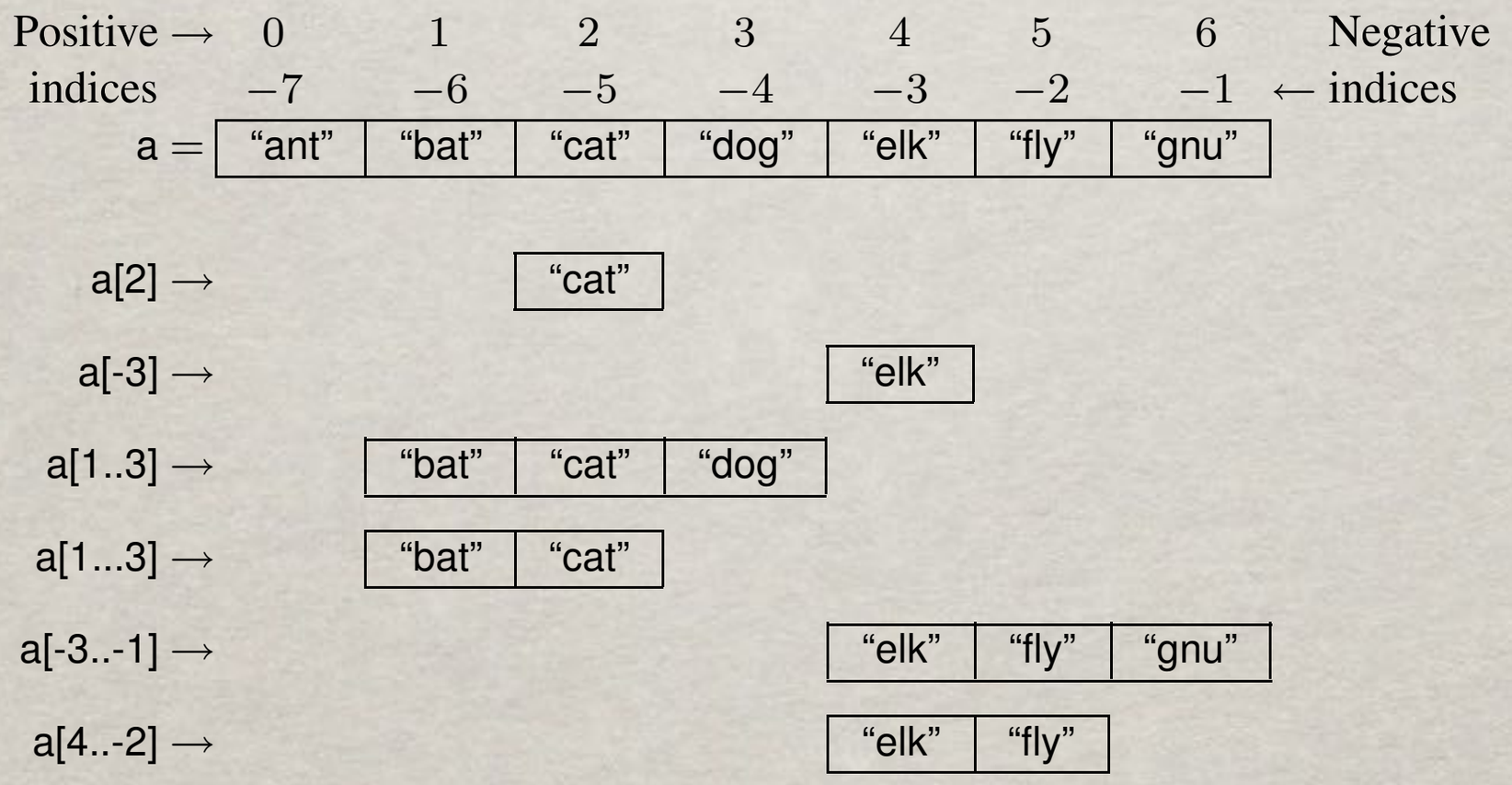
Einen ähnlichen Effekt kann man erzielen, indem man den [ ]-Operator mit einer Bereichsangabe verwendet:

```
a = [1, 3, 5, 7, 9]
a[1..3]                 => [3, 5, 7]
a[1...3]                => [3, 5]
a[3..3]                 => [7]
a[-3..-1]               => [5, 7, 9]
```

In beiden Fällen (zwei numerische Argumente bzw. eine Bereichsangabe) erhält man als Wert einen neuen Array, der die Elemente des spezifizierten Abschnitts des Ausgangsarray enthält.

4.1 Container

Figure 4.1. How Arrays Are Indexed



4.1 Container

Arrays

Dem [ ]-Operator korrespondiert der [ ]=-Operator, der verwendet werden kann, um einzelnen Feldern eines Arrays Werte zuzuweisen:

```
a = [ 1, 3, 5, 7, 9 ]        => [1, 3, 5, 7, 9]
a[1] = 'bat'                 => [1, "bat", 5, 7, 9]
a[-3] = 'cat'                => [1, "bat", "cat", 7, 9]
a[3] = [ 9, 8 ]              => [1, "bat", "cat", [9, 8], 9]
a[6] = 99                    => [1, "bat", "cat", [9, 8], 9, nil, 99]
```

Wird der Zuweisungsoperator mit zwei Argumenten verwendet, dann werden alle Felder des Arrays in dem bezeichneten Bereich mit Objekt besetzt, das durch das zweite Argument des Zuweisungsoperators bezeichnet wird:

```
a = [ 1, 3, 5, 7, 9 ]        => [1, 3, 5, 7, 9]
a[2, 2] = 'cat'              => [1, 3, "cat", 9]
a[2, 0] = 'dog'              => [1, 3, "dog", "cat", 9]
a[1, 1] = [ 9, 8, 7 ]        => [1, 9, 8, 7, "dog", "cat", 9]
a[0..3] = []                 => ["dog", "cat", 9]
a[5..6] = 99, 98             => ["dog", "cat", 9, nil, nil, 99, 98]
a[4] = 1, 2                  => ["dog", "cat", 9, nil, [1, 2], 99, 98]
```

4.1 Container

Arrays

In der Klasse **Array** sind zahlreiche nützliche Methoden definiert, die es z.B. ermöglichen, Arrays wie *Stacks, Mengen, Queues*, etc. zu behandeln:

a) Stack: push / pop

```
stack = []
stack.push "rot"
stack.push "grün"
stack.push "blau"
p stack                 => ["rot", "grün", "blau"]
puts stack.pop          => blau
puts stack.pop          => grün
puts stack.pop          => rot
p stack                 => []
```

4.1 Container

Arrays

b) Queue (FIFO): **shift** / **unshift**

```
queue = []
queue.push "rot"
queue.push "grün"
p queue                          => [rot, grün]
puts queue.shift                 => rot
queue.unshift “blau“
p queue                          => [blau, grün]
puts queue.shift                 => blau
```

c) **first** / **last**

```
array = [ 1, 2, 3, 4, 5, 6, 7 ]
p array.first(4)                 => [1, 2, 3, 4]
p array.last(4)                  => [4, 5, 6, 7]
```

4.1 Container

Berechnung von Wortfrequenzen mit *Arrays* und *Hashes*

Für einen Text soll berechnet werden, wie oft ein Wort bzw. eine Wortform in ihm vorkommt. Dieses Problem lässt sich in zwei Teilprobleme zerlegen:

1. Angenommen der Text liegt als ein (großer) String vor, dann muss er in einzelne Elemente zerlegt werden, die dann in einem *Array* gespeichert werden können.
2. Anschließend ist für jede Wortform die Frequenz zu berechnen. Hier bietet sich die Verwendung eines *Hashes* an.

Wie wir einen String in einzelne Token zerlegen können, haben wir bereits gesehen:

```
def words_from_string(string)
    string.downcase.scan(/[\w']+/)
end

p words_from_string("But I didn't inhale, he said (emphatically)")
["but", "i", "didn't", "inhale", "he", "said", "emphatically"]
```

4.1 Container

Berechnung von Wortfrequenzen mit *Arrays* und *Hashes*

Die Wortfrequenzen lassen sich leicht mit Hilfe eines *Hashes* speichern:

```
{ ..., "the" => 1, ... }
```

Angenommen, wir verwenden eine Variable counts um den Hash zu verwalten und die Variable next_word enthält das Wort "the", dann kann der Zähler durch die Anweisung

```
counts[next_word] += 1
```

aktualisiert werden. Beim Eintragen neuer Wortformen ist zu beachten, dass Ruby normalerweise für unbekannte Schlüssel den Wert nil liefert. Dieses Standardverhalten lässt sich durch Angabe eines geeigneten Arguments bei der Generierung des Hashes ändern:

```
Hash.new(Wert)
```

4.1 Container

Berechnung von Wortfrequenzen mit *Arrays* und *Hashes*

words_from_string.rb

```
def words_from_string(string)
    string.downcase.scan(/[\w']+/)
end

def count_frequency(word_list)
    counts = Hash.new(0)
    for word in word_list
        counts[word] += 1
    end
    counts
end

Dir.chdir("code/wordfreq")
raw_text  = File.read("para.txt")
word_list = words_from_string(raw_text)
counts    = count_frequency(word_list)
sorted    = counts.sort_by {|word, count| count}
top_five  = sorted.last(5)
```

count_frequency.rb

```
for i in 0...5
  word = top_five[i][0]
  count = top_five[i][1]
  puts " {word}:   {count}"
end
```

that: 2
sounds: 2
like: 3
the: 3
a: 6

4.1 Container

```
require_relative ,words_from_string.rb'
require 'minitest/autorun'

class TestWordsFromString < MiniTest::Test
  def test_empty_string
    assert_equal([], words_from_string(""))
    assert_equal([], words_from_string("     "))
  end
  def test_single_word
    assert_equal(["cat"], words_from_string("cat"))
    assert_equal(["cat"], words_from_string("  cat   "))
  end
  def test_many_words
    assert_equal(["the", "cat", "sat", "on", "the", "mat"],
        words_from_string("the cat sat on the mat"))
  end
  def test_ignores_punctuation
    assert_equal(["the", "cat's", "mat"],
        words_from_string("<the!> cat's, -mat-"))
  end
end
```

```
Loaded suite /tmp/prog
Started
....
Finished in 0.000578 seconds.
4 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

4.1 Container

```
require_relative  'count_frequency.rb'
require 'minitest/autorun'

class TestCountFrequency < MiniTest::Test
  def test_empty_list
    assert_equal({}, count_frequency([]))
  end
  def test_single_word
    assert_equal({"cat" => 1}, count_frequency(["cat"]))
  end
  def test_two_different_words
    assert_equal({"cat" => 1, "sat" => 1},
                 count_frequency(["cat", "sat"]))
  end
  def test_two_words_with_adjacent_repeat
    assert_equal({"cat" => 2, "sat" => 1},
                 count_frequency(["cat", "cat", "sat"]))
  end
  def test_two_words_with_non_adjacent_repeat
    assert_equal({"cat" => 2, "sat" => 1},
                 count_frequency(["cat", "sat", "cat"]))
  end
end
```

```
Loaded suite /tmp/prog
Started
.....
Finished in 0.000534 seconds.
5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

4.2 Blöcke und Iteratoren

```
for i in 0...5
    word = top_five[i][0]
    count = top_five[i][1]
    puts " {word}:  {count}"
end
```

klassische Schleife

```
top_five.each do |word, count|
    puts " {word}:  {count}"
end
```

Iterator (each)

```
puts top_five.map {|word, count| " {word}:  {count}"}
```

Iterator (map/collect)

4.2 Blöcke und Iteratoren

Ein Block besteht aus einer Folge von Anweisungen, die durch geschweifte Klammern oder do ... end begrenzt wird. Ein Block kann als eine anonyme (also namenlose) Methode verstanden werden. Wie Methoden können auch Blöcke Parameter akzeptieren. Sie können in einem Rubyprogramm nur direkt auf einem Methodenaufruf folgen:

Methode Args Block*

```
sum = 0
[1, 2, 3, 4].each do |value|
    square = value * value
    sum += square
end
puts sum
30
```

Zu beachten ist in diesem Fall, dass die Variable *sum* außerhalb des Blocks deklariert wird, innerhalb des Blocks aktualisiert und nach verlassen des Blocks ihr aktueller Wert zurückgegeben wird. Es gilt also: Ein Block kann auf alle Variablen einer Umgebung, in die er eingebettet ist, zugreifen. Auf im Block definierte Variablen kann außerhalb des Blocks dagegen nicht zugegriffen werden.

4.2 Blöcke und Iteratoren

```
square = “Ein Quadrat hat vier gleichlange Seiten.“

  ... jede Menge Kode

sum = 0
[1, 2, 3, 4].each do |value|
    square = value * value
    sum += square
end
puts sum              => 30
puts square           =>  ???
```

4.2 Blöcke und Iteratoren

Grundregeln

1. Der Geltungsbereich der Parameter eines Blocks ist immer nur der Block selbst

```
value = "some shape"
[ 1, 2 ].each {|value| puts value }
puts value
1
2
some shape
```

2. Zusätzliche lokale Variablen für einen Block können mit den Parametern nach einem Semikolon deklariert werden.

```
square = "some shape"
sum = 0
[1, 2, 3, 4].each do |value; square|
  square = value * value      this is a different variable
  sum   += square
end
puts sum                  => 30
puts square               => some shape
```

4.2 Blöcke und Iteratoren

Implementierung von Iteratoren

Iteratoren können in Ruby verstanden werden als Methoden, die einen Anweisungsblock mehrfach bzw. für alle Objekte einer Gruppe von Objekten ausführen.

Der Anweisungsblock muss mit einem Methodenaufruf assoziiert sein und wird zu einem gegebenen Zeitpunkt ausgeführt: Aus der Methode heraus kann er durch eine yield-Anweisung aufgerufen und ausgeführt werden.
Nach Ausführung der Anweisungsblocks werden die auf die yield-Anweisung der Methode folgenden Anweisungen ausgeführt.

```
def three_times
    yield
    yield
    yield
  end
three_times { puts "Hello" }
Hello
Hello
Hello
```

4.2 Blöcke und Iteratoren

Die Leistungsfähigkeit dieses Mechanismus basiert vor allem darauf, dass es möglich ist, Parameter an Blöcke zu übergeben und die von ihnen gelieferten Werte weiter zu verarbeiten:

```
def fib_up_to(max)
    i1, i2 = 1, 1   parallele Zuweisung (i1 = 1 and i2 = 1)
    while i1 <= max
        yield i1
        i1, i2 = i2, i1+i2
    end
end
fib_up_to(1000) {|f| print f, " " }

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

In diesem Beispiel wird durch die yield-Anweisung ein Parameter an den Block übergeben. Grundsätzlich können an einen Block beliebig viele Parameter übergeben werden.

4.2 Blöcke und Iteratoren

Der Wert der letzten Anweisung eines Block ist der Wert, den der Block an die ihn aufrufende Methode zurückgibt. Die Methode find der Klasse **Array** nutzt diese Eigenschaft, um in einem Array ein gesuchtes Objekt zu finden. Sie könnte wie folgt implementiert sein:

```
class Array
    def find
        for i in 0...size
            value = self[i]
            return value if yield(value)
        end
        return nil
    end
end
[1, 3, 5, 7, 9].find {|v| v*v > 30 }   => 7
```

Sobald ein Objekt die durch den Block spezifizierte Eigenschaft erfüllt, wird es als Wert der Methode zurückgegeben.

4.2 Blöcke und Iteratoren

Neben find gibt es in Ruby viele weitere Iteratoren. Zu den wichtigsten gehören each und collect. each ist konzeptuell sehr einfach: Es ruft mit yield den Anweisungsblock nacheinander mit jedem Element der Objektsammlung auf.

```
[ 1, 3, 5, 7, 9 ].each {|i| puts i }
1
3
5
7
9
```

collect oder map (ein Alias von collect) arbeitet wie each, sammelt aber die durch den Block erzeugten Werte in einem Array.

```
["H", "A", "L"].collect {|x| x.succ }   => ["I", "B", "M"]
```

4.2 Blöcke und Iteratoren

Auch in Verbindung mit IO-Objekten, etwa um Informationen aus einer Datei zu lesen, lassen sich Iteratoren verwenden:

```
f = File.open("testfile")
f.each do |line|
    puts "Die Zeile enthält:  {line}"
end
f.close
Die Zeile enthält: <Inhalt der ersten Zeile>
Die Zeile enthält: <Inhalt der zweiten Zeile>
...
```

Sollte es wichtig sein zu wissen, wie oft man den mit einer Methode assoziierten Block aufgerufen hat, kann man statt each each_with_index verwenden:

```
f = File.open("testfile")
f.each_with_index do |line, index|
    puts "Zeile  {index}:  {line}"
end
f.close
Zeile 0: <Inhalt der ersten Zeile>
Zeile 1: <Inhalt der zweiten Zeile>
...
```

4.2 Blöcke und Iteratoren

Ein weiterer nützlicher Iterator ist die inject Methode, die es ermöglicht, die von dem Block gelieferten Werte zu akkumulieren.

```
[1,3,5,7].inject(0) {|sum, element| sum+element}   => 16
[1,3,5,7].inject(1) {|product, element| product*element}   => 105
```

Beim ersten Aufruf des Blocks wird die Variable sum mit dem Wert des Parameters (0) und element mit dem ersten Objekt des Arrays (1) initialisiert. Bei jedem weiteren Aufruf wird auf sum den Wert des letzten Aufrufs des Blocks gesetzt. inject liefert als Wert den Wert des letzten Aufrufs des Blocks.

Wird inject ohne Argumente aufgerufe, wird die Resultatsvariable mit dem ersten Element der Kollektion initialisiert und anschließend wird über die übrigen Elemente iteriert:

```
[1,3,5,7].inject {|sum, element| sum+element}   => 16
[1,3,5,7].inject {|product, element| product*element}   => 105
```

Man kann auch die anzuwendende Methode als Parameter an inject übergeben. In diesem Fall ist die Methode durch ein geeignetes Symbol zu bezeichnen:

```
[1,3,5,7].inject(:+)   => 16
[1,3,5,7].inject(:*)   => 105
```

4.3 Emumeratoren

In Ruby sind die Iteratoren als Methoden realisiert, d.h. sie sind Bestandteil der Klassen (wie **Array**, **Hash**, etc.), die in Ruby verwendet werden, um Sammlungen von Objekten zu verwalten.

Allerdings gibt es Situationen, in denen Rubys *interne* Iteratoren nicht die beste Lösung darstellen:

1. Manchmal ist es notwendig, Iteratoren als eigenständige Objekte behandeln zu können.
2. Es ist schwierig, parallel über mehrere Objektsammlungen zu iterieren.

Allerdings verfügt Ruby 1.9 über die vordefinierte **Emunerator** Klasse, die es erlaubt, die für die Behandlung dieser Situationen erforderlichen *externen* Iteratoren zu generieren.

4.3 Emumeratoren

Eine Möglichkeit, ein Enumerator-Objekt zu generieren, besteht darin, die to_enum Methode für einen Array oder Hash aufzurufen:

```
a = [ 1, 3, "cat" ]
h = { dog: "canine", fox: "lupine" }

  Generierung von Enumeratoren
enum_a = a.to_enum
enum_h = h.to_enum

  Verwendung der Enumeratoren
enum_a.next   => 1
enum_h.next   => [:dog, "canine"]
enum_a.next   => 3
enum_h.next   => [:fox, "lupine"]
```

Viele Methoden, die interne Iteratoren realisieren, liefern als Wert ein Enumerator Objekt, wenn sie ohne Block aufgerufen werden:

```
a = [ 1, 3, "cat" ]

enum_a = a.each   erzeugt ein Enumerator-Objekt

enum_a.next   => 1
enum_a.next   => 3
```

4.3 Emumeratoren

Rubys loop Methode macht nichts anderes, als den assoziierten Block immer wieder aufzurufen. In Verbindung mit Enumeratoren erweist sich loop als sehr nützlich, da die Schleife terminiert, sobald einer der Enumeratoren keinen Wert mehr liefert:

```
short_enum = [1, 2, 3].to_enum
long_enum = ('a'..'z').to_enum

loop do
    puts " {short_enum.next} -  {long_enum.next}"
end

1 - a
2 - b
3 - c
```

4.3 Emumeratoren

Enumeratoren verwandeln normalen ausführbaren Kode in ein Objekt und ermöglichen es so, Aufgaben mit Enumeratoren zu lösen, die durch einfache Schleifen (loop) nicht ohne weiteres gelöst werden können.

Mit each_with_index erhält man die aufeinander folgenden Objekte einer Sammlung zusammen mit ihrem Index:

```
result = []
[ 'a', 'b', 'c' ].each_with_index {|item, index| result << [item, index] }
result   => [["a", 0], ["b", 1], ["c", 2]]
```

Was aber, wenn eine Klasse nicht über eine entsprechende Methode verfügt? So gibt es z.B. für Strings keine Methode each_char_with_index. Enumeratoren lösen dieses Problem:

```
result = []
"cat".each_char.each_with_index {|item, index| result << [item, index] }
result   => [["c", 0], ["a", 1], ["t", 2]]
```

4.3 Emumeratoren

Enumeratoren können als *Generatoren* und *Filter* verwendet werden. Wird bei der expliziten Generierung eines Enumerators ein Block übergeben, dann werden die Anweisungen des Blocks ausgeführt, sobald der Enumerator einen neuen Wert liefern soll. Die Ausführung des Kodes wird nach der yield-Anweisung unterbrochen und dort beim nächsten Aufruf des Enumerators fortgesetzt.

```
triangular_numbers = Enumerator.new do |yielder|
  number = 0
  count = 1
  loop do
    number += count
    count += 1
    yielder.yield number
  end
end
3.times { puts triangular_numbers.next }
1
3
6
```

4.3 Emumeratoren

Bei der Verwendung von Enumeratoren, die nicht-endliche Folgen von Objekten generieren, muss man vorsichtig sein; denn bestimmte Methoden wie count und select versuchen die komplette Folgen von Objekten zu lesen, bevor sie einen Wert zurückgeben. In solchen Fällen muss man eigene Versionen dieser Methoden definieren:

```
triangular_numbers = Enumerator.new do |yielder|
  ...   s.o.
end

def infinite_select(enum, &block)
  Enumerator.new do |yielder|
    enum.each do |value|
      yielder.yield(value) if block.call(value)
    end
  end
end

p infinite_select(triangular_numbers) {|val| val % 10 == 0}.first(5)
[10, 120, 190, 210, 300]
```

4.4 Mehr Blöcke

Transaktionen

Mit Blöcken ist es einfach ein Programm zu schreiben, das bestimmte Transaktions-regeln beachten soll. Etwa wird häufig eine Datei geöffnet, der Inhalt verarbeitet und abschließend muss sichergestellt werden, dass sie wieder ordnungsgemäß geschlossen wird. Naive Implementierung:

```
class File
    def self.open_and_process(*args)
      f = File.open(*args)
      yield f
      f.close()
    end
  end

  File.open_and_process("testfile", "r") do |file|
    while line = file.gets
      puts line
    end
  end
```

Zeile 1
Zeile 2
...

4.4 Mehr Blöcke

Abhängig davon, ob sie mit einem Block aufgerufen wird oder nicht, reagiert die vordefinierte Methode File.open ganz unterschiedlich:
Ist ein Block vorhanden, wird er mit einem Dateiobjekt als Parameter ausgeführt und anschließend die Datei geschlossen. Ohne Block liefert sie das Dateiobjekt als Wert.

Man kann sich diese Methode wie folgt definiert vorstellen:

```
class File
  def self.my_open(*args)
    resultat = datei = File.new(*args)
      Wenn es einen Block gibt, dann übergebe die
      Datei und schließe sie nach Beendigung des
      Blocks
    if block_given?
      resultat = yield datei
      datei.close
    end
    return resultat
  end
end
```

4.4 Mehr Blöcke

Blöcke verhalten sich zunächst wie anonyme Methoden. Man kann sie aber auch in Objekte konvertieren, in Variablen speichern, weiterreichen und zu einem beliebigen Zeitpunkten ausführen lassen.

Parameter in Methoden, deren Bezeichner ein ‚&' vorangestellt wird, erwarten einen Block und konvertieren ihn in ein Objekt der Klasse **Proc**.

```
class ProcExample
  def pass_in_block(&action)
    @stored_proc = action
  end
  def use_proc(parameter)
    @stored_proc.call(parameter)
  end
end


eg = ProcExample.new
eg.pass_in_block { |param| puts "Der Parameter ist  {param}" }
eg.use_proc(99)

Der Parameter ist 99
```

*In der ersten Instanzenmethode wird ein **Proc**-Objekt erzeugt und in einer Instanzenvariable gespeichert. In der zweiten Methode wird das **Proc**-Objekt aufgerufen.*

4.4 Mehr Blöcke

Statt das erzeugte **Proc**-Objekt in einer Instanzenvariablen zu speichern, kann man es auch direkt zurückgeben und aufrufen:

```
def create_block_object(&block)
  block
end

bo = create_block_object { |param| puts "I wurde aufgerufen
mit  {param}" }

bo.call 99
bo.call "cat"
```

Ich wurde aufgerufen mit 99
Ich wurde aufgerufen mit cat

Variante:

```
bo = lambda { |param| puts "I wurde aufgerufen mit  {param}" }
bo.call 99
bo.call "cat"
```

Ich wurde aufgerufen mit 99
Ich wurde aufgerufen mit cat

4.4 Mehr Blöcke

Ein Block kann, wir wir wissen, auf die Variablen zugreifen, die in der Umgebung definiert werden, in die der Block eingebettet ist:

```
def n_times(thing)
   lambda {|n| thing * n }
 end

 p1 = n_times(23)
 p1.call(3)                    => 69
 p1.call(4)                    => 92
 p2 = n_times("Hello ")
 p2.call(3)                    => “Hello Hello Hello“
```

*Die Methode erzeugt **Proc**-Objekt, dass auch den Parameter* `thing` *zugreift, der außerhalb des **Proc**-Objekts definiert wurde.*

4.4 Mehr Blöcke

```
def power_proc_generator
  value = 1
  lambda { value += value }
end

power_proc = power_proc_generator

puts power_proc.call
puts power_proc.call
2
4
8
```

Auch hier greift das Proc-Objekt auf eine Variable zu, die in der es einbettenden Umgebung definiert wurde. Das Ergebnis ist eine Generator für 2er-Potenzen.

4.4 Mehr Blöcke

In Ruby 1.9 gibt eine weitere Möglichkeit, um **Proc**-Objekt zu erzeugen. Statt:
lambda {|parameter| ...}
kann man auch
->parameter {...}
schreiben, wobei die Paramter auch geklammert werden können.

```
proc1 = -> arg { puts "In proc1 with  {arg}" }
proc2 = -> arg1, arg2 { puts "In proc2 with  {arg1} and  {arg2}" }
proc3 = ->(arg1, arg2) { puts "In proc3 with  {arg1} and  {arg2}" }

proc1.call "ant"
proc2.call "bee", "cat"
proc3.call "dog", "elk"

In proc1 with ant
In proc2 with bee and cat
In proc3 with dog and elk
```

4.4 Mehr Blöcke

Die ‚->'-Notation ist deutlich kompakter als die ‚lambda'-Notation und bietet sich besonders dann an, wenn mehrere **Proc**-Objekte als Argumente einer Methode verwendet werden.

```
def my_if(condition, then_clause, else_clause)
  if condition
    then_clause.call
  else
    else_clause.call
  end
end

5.times do |wert|
  my_if wert < 3,
        -> { puts " {wert} ist klein" },
        -> { puts " {wert} ist gross" }
end
```

```
0 ist klein
1 ist klein
2 ist klein
3 ist gross
4 ist gross
```

4.4 Mehr Blöcke

Ein guter Grund dafür, Blöcke als Argumente an Methoden zu übergeben besteht darin, das sie sooft man will und wann man will ausgeführt werden können.

```
def my_while(cond, &body)
  while cond.call
    body.call
  end
end

a = 0
my_while -> { a < 3 } do
   puts a
   a += 1
end
```