



Ruby Hands On

Mario Manno

Welcome



Content

- about ruby
- base knowledge
 - handling data
- oop knowledge
- fancy knowledge

about ruby



- history
 - made in japan
 - 10 Jahre alt
- features
 - Skriptsprache
 - Dynamische Typen
 - Objektorientiert

base knowledge - using it

- `ruby script.rb`
- `irb`

base knowledge - variable definition

```
num = 3
```

```
str = 'drei'
```

```
arr = [ 1 , 2 , 3 ]
```

```
hash = {a => '1' , b=> '2' }
```

```
hash = Hash.new
```

```
range = (1..3)
```

```
regexp = /[1-3]/
```

```
x,y = y,x
```

constants are capitalized: SomeConstant, TRUE, ARGV, ENV

handling data - string control

```
# Ausgabe
puts "yello"
s = "old data"
print "#{s}"
printf "%s", s

# Modifikation
s.chomp!
s.gsub!( "old", "new" )

zahl = s.length
s = "12"
zahl = s.to_i
```

base knowledge - Kontrollstrukturen

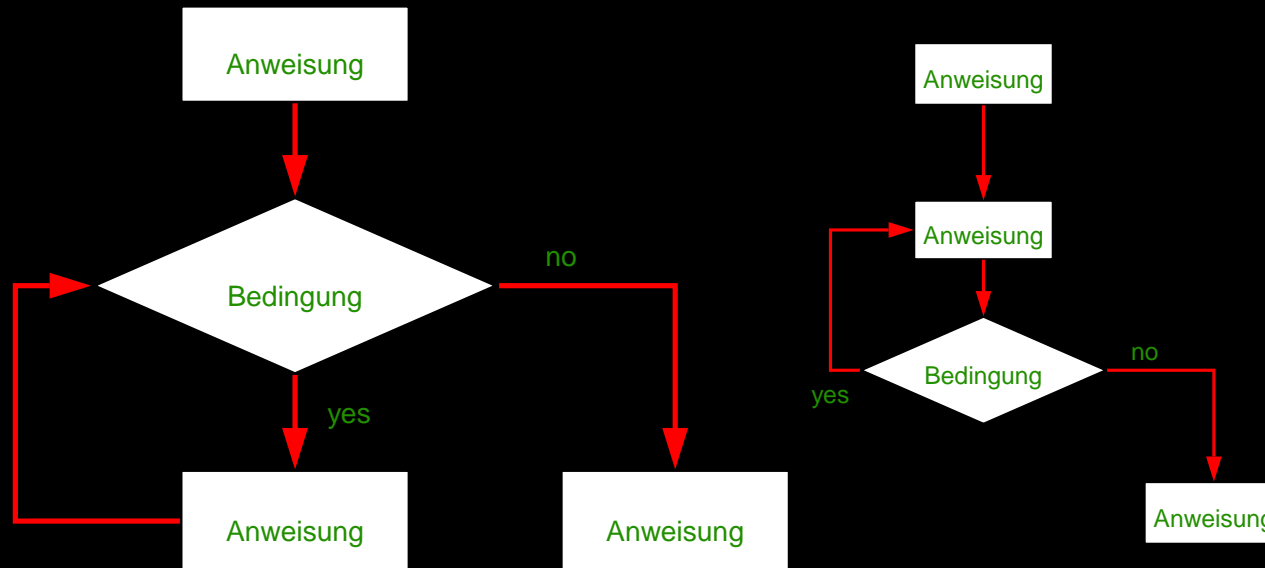
Programme bestehen aus Sequenzen.

- Sequenzen, also aufeinanderfolgenden Code Zeilen, lassen sich in Blöcke zusammenfassen.
- Zwei Arten existieren um Blöcke zu notieren:

```
do  
  ...  
end
```

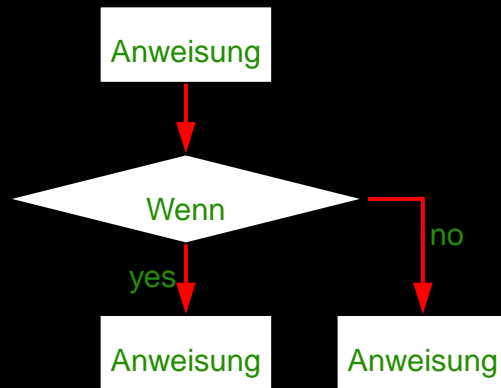
```
{ ... }
```


base knowledge - Kontrollstrukturen



Schleifen sind einer der Grundbestandteile jeder Sprache. Sie wiederholen Blöcke von Anweisungen in Abhängigkeit von einer Bedingung.

base knowledge - Kontrollstrukturen



Die Auswahl, auch Fallunterscheidung genannt, führt Code in Abhängigkeit von einer logischen Bedingung aus.

base knowledge - loop

Endlosschleife

```
loop do  
  print "hi"  
end
```

base knowledge - while / until

```
while variable.nil?  
  print "hi"  
end
```

base knowledge - times

```
10.times { print "text" }
```

base knowledge - for

```
for i in (1..10)
  print i
end
```

base knowledge - next / break

- next - beendet durchlauf eines blocks, springt zur bedingung
- break - beendet einen block

```
loop do
  next unless STDIN.readline.strip == 'jo'
  print "jo was entered n"
end
```

base knowledge - if / unless

```
if a
  print a
elsif b == 'jo'
  print 'jo'
end
```

```
auch
print "wahr" unless False
```


base knowledge - case

```
case s
when /Ich/
  return "mm"
when /Du/
  return "xy"
end
```

base knowledge - block arguments

Blöcke können Argumente übergeben bekommen.

`{ |x,y| p x*y }`

base knowledge - iterators

- Iteratoren bearbeiten Elemente einer Liste
- each ist der Standard Iterator
- select, inject, collect sind von each abgeleitet
- Objekte können ihre eigenen Iteratoren definieren

base knowledge - iterators

```
(0..10).each { |i| print i }  
# 012345678910
```

```
s = "hello world"  
a = s.split(//)
```

```
a.each do |c| print c end  
"hello world".split(//).each { |c| p c }
```

```
a.select do |c| c =~ /[a-m]/ end  
# => ["h", "e", "l", "l", "l", "d"]
```

```
Dir.open(".").sort.each{ |f| print f }  
Dir.foreach(".") { |f| puts f }
```

handling data - array control

```
arr=[2,4,8,16]
arr2 = Array.new # []
arr.push "element"

s = arr.join(',')
s = arr.to_s

arr + arr2
[arr[1],arr[-1]]

arr.reverse
arr.include? "element"
arr.each { |e| p e }
arr.each_index { |i| print arr[i] }
```

handling data - hash control

Hashes sind Name-Wert Paare. Auch Dictionaries oder Assoziative Arrays genannt.

```
# Zuweisung
h = Hash.new # {}
h = { 'a'=>1, 'b'=>2 }
a = h.keys

# Iterieren
h.each { |n,v| print "#{n} = #{v}" }
h.each_key { |n| print "#{n} = #{h[n]}" }

# Tests
h.empty?
h.key?( "test" )
```

handling data - Hash of Arrays

```
# construct
h = Hash.new
h[ 'mm' ] = [ 'ford', 'nixon' ]

# add
h[ 'mm' ].push( "bush" )
h[ 'c4' ] = Array.new
h[ 'c4' ].push( "w8n" )
# { "mm"=>[ "ford", "nixon", "bush" ], "c4"=>[ "w8n" ] }

# output
p h[ 'mm' ][1]
# "nixon"
```

handling data - file control

```
file = File.open('/etc/shadow')
file.chmod(0777)

# und alles was io kann:
file.readlines.each { |l| print l }

# Ausgabe an abuse@koeln.ccc.de senden
```


handling data - file control

```
File.open( '/etc/passwd' ).each{ |f| puts f }.close
```

```
File.open( '/etc/passwd' ).select{ |l|  
  l =~ /root/  
}.collect{ |m|  
  " t"+m+" n"  
}.each{ |n|  
  puts n  
}
```

handling data - regular expressions

● match

```
s = '3llo.rb'  
res = /^(d+).*\.rb/\.match(s)  
  
p res[1]  
# "3"
```

● substitution:

```
s = "chaos computer club cologne"  
neu = s.gsub(/^(chaos computer.*)$/) {  
  "club mate" + $1 }  
  
p neu  
# "club mate club cologne"
```

handling data - regular expressions

- perl style

```
line = "open chaos"  
if line =~ /(.* ) chaos/  
  p $1  
  # open  
  p $&  
  # open chaos  
end
```

handling data - other builtins

Time
Dir
Thread
Number
Range

base knowledge - function interface

- a,b - Normale Objekte
- *c - Liste von Objekten, z.B. ein Array oder Hash
- &d - Code Block (proc objekt)

Deklaration:

```
def throw (a,b,*c)  
end
```

Aufruf:

```
throw(a,b,c)  
throw(a,b,c,d)
```

base knowledge - function return values

```
def test
  "test"
end

# optionale parameter
def test2(a,b='')
  return a+" "+b
end

print test
# "test"
print test2("hallo")
# "hallo "
print test2("hallo","welt")
# "hallo welt"
```

base knowledge - exceptions

```
begin
  dangerous_code
rescue Exception1, Exception2 => ex
  p ex
rescue Exception3
  puts $!
else
  puts "fine"
ensure
  puts "always"
end
```

base knowledge - exceptions

```
def dangerous_function
  error=1
  if error
    raise "the message"
  end
end

begin
  dangerous_function
rescue
  puts $!
end
```


base knowledge - exceptions real example

```
begin
  f = File.new('test')

  rescue Errno::ENOENT
    puts "file open error: #{${!}}"

  rescue Exception => ex
    puts "other error: #{ex.to_str}"

  ensure
    f.close if f

end
```

oop knowledge

Die Instanz einer Klasse wird Objekt genannt.

- Klasse -> Objekt (Instanz)
- Eine Klasse besteht aus Methoden (Operationen) und Attributen.
- Die Schnittstelle einer Operation und ihr Rückgabewert werden Signatur genannt.
- Alle Signaturen eines Objekts bestimmen sein Interface.
- Man spricht auch vom Typ eines Objekts, damit ist ein spezifischer Teil seines Interfaces gemeint.
- Kapselung, Objekte verstecken ihre Daten vor der Aussenwelt.

handling objects - get information

```
puts FALSE.class
puts FALSE.methods.sort
puts $stdin.methods.select { |m| m =~ /read/ }

a = Array.new
if a.respond_to?('clear')
  a.send('clear')
end
```

oop knowledge - variable scope

Der Gültigkeitsbereich einer Variable wird in Ruby wie folgt notiert:

- instance **@a**
- global **\$a**
i.e. **STDERR \$stderr**
- class **@@a**

oop knowledge - class constructor

- Erschafft die Objektinstanz während der Laufzeit.
- Initialisiert die Objektinstanz.
- Die Attribute des Objekts können mit Werten aus der Schnittstelle des Konstruktors vorbelegt werden.

```
def initialize  
def initialize(wert)
```

oop knowledge - class/func definition

```
class Elefant
  def initialize
    @farbe = 'grau'
  end

  def troete
  end
end
```

oop knowledge - class attrib reader/writer

Es ist nicht notwendig get_ und set_ Methoden zu schreiben.

- Lesender Zugriff auf die Attribute a und c

```
attr_reader :a, :c
```

- Schreibender Zugriff auf das Attribut a

```
attr_writer :a
```

oop knowledge - class creation (new)

Erschaffung eines Objektes instanz der Klasse Something.
Dem Konstruktor werden in diesem Beispiel keine Parameter übergeben.

```
instanz = Something.new
```


oop knowledge - object debugging

Nützliche Ausgaben zu einem Objekt.

- `p instanz`
- `p instanz.methods`
- `p instanz.display`
- `p instanz.inspect`

oop knowledge - full class example, definition

```
class Addition
  def initialize(a,b)
    @a = a
    @b = b
  end
  attr_reader :a
  attr_writer :b

  def ergebnis
    @a+@b
  end
end
```

continued on next page ...

oop knowledge - full class example, usage

```
add = Addition.new(2, 3)
p add.ergebnis
p add.a
add.b = 5
p add.ergebnis
```

oop knowledge - class inheritance

- Vererbung (Inheritance), Objekte können Attribute und Methoden von anderen Objekten erben.
- Polymorphie, dynamische Bindung, es spielt keine Rolle welche Klasse die betreffende Methode implementiert, solange die Signatur stimmt.
- Spezialisierung ist der häufigste Anwendungsfall für Vererbung.
- Mixins und Komposition sind Alternativen zur Vererbung.

oop knowledge - class inheritance

```
class Subtraktion < Addition
  def ergebnis
    @a - @b
  end
end
```

oop knowledge - Mixins

- Module sind gebräuchliche Mixins für Klassen.
- Sie sind nichtinstanzierbare Klassen.
- Über Module lassen sich auch Namespaces realisieren,
- In manchen Sprachen werden Mixins über multiple Vererbung realisiert.

Definition: `module Tools`
Verwendung: `include Tools`

```
module Tools
  def ausgabe(s)
    printf( "%10.3f" ,s)
  end
end
```

oop knowledge - Mixins

```
class Subtraktion
  include Tools
  def ergebnis_ausgeben
    ausgabe(ergebnis)
  end
end

sub = Subtraktion.new(5, 2)
sub.ergebnis_ausgeben
# 3.000
```

oop knowledge - Assoziation

- Die Assoziation wird durch keinen besonderen Mechanismus ausgedrückt.
- Assoziation liegt vor wenn zwei, oder mehr Objekte voneinander wissen.
- Enthält ein Objekt eine Referenz auf ein Anderes, ist es für das Andere verantwortlich, spricht man auch von Aggregation.

oop knowledge - Komposition

Die stärkste Form der Aggregation ist die Komposition. Statt unübersichtliche Vererbungs Hierarchien zwischen den Klassen aufzubauen, besitzt ein Objekt andere Objekte ganz.

- Die Objekte müssen nicht vom gleichem Typ sein.
- Die Objekte müssen komplett im Hauptobjekt liegen.
- Die Objekte sind an die Laufzeit des Hauptobjektes gebunden.
- Vererbungs ist “white-box“ Reuse.
- Komposition ist “black-box“ Reuse.

oop knowledge - Komposition

```
class Rechner
  def rechne (op='+' , a=0 , b=0)
    case op
    when /+/
      rechenwerk = Addition.new(a,b)
    when /-/
      rechenwerk = Subtraktion.new(a,b)
    end
    rechenwerk.ergebnis # Polymorphie
  end
end
```

```
rechner = Rechner.new
p rechner.rechne('+' , 2 , 3)
p rechner.rechne('-' , 5 , 3)
```

design knowledge - delegation

- Mittels Delegation wird Komposition ähnlich mächtig wie Vererbung.
- Delegation unterscheidet in ein empfangendes und ein delegiertes Objekt.
- Das empfangende Objekt ruft eine Methode des delegierten Objektes auf und übergibt sich selbst.
- Das delegierte Objekt arbeitet dann auf dem empfangenden Objekt.

oop knowledge - Delegation

```
class Multiplikator
  def berechne(obj,a,b) obj.erg = a*b end
end
class Rechner
  def initialize
    @op = Multiplikator.new
  end
  attr_writer :erg
  def multipliziere(a,b)
    @op.berechne(self,a,b)
    p @erg
  end
end
r = Rechner.new
r.multipliziere(2,4)
```

design patterns - observer/singleton/visitor

- Singleton - Es kann nur eine Instanz dieser Klasse geben
- Visitor - externe Operationen auf Listen von Objekten
- Observer - Objekte schicken sich Updates

design patterns - singleton

- Es kann nur eine Instanz einer Klasse geben.

```
require 'singleton'  
  
class Highlander  
  include Singleton  
  def slay_stupid_code_examples  
  end  
end
```

design patterns - visitor

- Arbeitet auf Listen nicht gleichartiger Objekte.
- Der Visitor enthält Methoden fuer jede Klasse in der Liste.
- Die Objekte rufen ihre entsprechende Methode im Visitor auf und uebergeben sich selbst.
- Mit dem Visitor lassen sich Objekte erweitern ohne sie zu ändern.
- Verschiedene Konkrete Visitor Klassen sind möglich.

design patterns - visitor, Erweiterung

```
class Addition
  def accept(v)
    v.visitAddition(self)
  end
end

class Subtraktion
  def accept(v)
    v.visitSubtraktion(self)
  end
end
```

continued on next page ...

design patterns - visitor, Klassen

```
class HalbierVisitor
  def visitAddition(o)
    p o.erg/2
  end
  def visitSubtraktion(o) p o.erg/2 end
end

class QuadrierVisitor
  def visitAddition(o)
    p o.ergebnis**2
  end
  def visitSubtraktion(o) p o.ergebnis**2 end
end
```

continued on next page ...

design patterns - visitor, usage

```
allerechner = []
allerechner.push( Addition.new(2,4) )
allerechner.push( Subtraktion.new(5,3) )

visitor = HalbierVisitor.new
allerechner.each { |r| r.accept(visitor) }

visitor = QuadrierVisitor.new
allerechner.each { |r| r.accept(visitor) }
```

design patterns - observer

- Ein Objekt schickt Updates an unbekannte Objekte.
- Empfänger registrieren sich beim Producer Objekt.
- Producer informiert alle Empfänger bei Veränderung seines States.
- Producer / Consumer Problem (Publish-Subscribe)

design patterns - observer example, Generator

```
require "observer"  
class Generator  
  include Observable  
  def loop  
    loop do  
      changed  
      notify_observers("value received")  
    end  
  end  
end  
end
```

continued on next page ...

design patterns - observer example, Consumer

```
class Consumer
  def initialize(id,g)
    @id = id
    g.add_observer(self)
  end
  def update(val)
    print "#{@id} #{val}  n"
  end
end
```

continued on next page ...

design patterns - observer example, Usage

```
generator = Generator.new  
c1 = Consumer.new(1,generator)  
c2 = Consumer.new(2,generator)  
generator.loop
```

fancy knowledge - yield

- Iteratoren werden ueber yield implementiert.
- yield gibt einen Wert an den aufrufenden Block.
- Das Programm läuft anschliessend nach dem yield weiter.

```
def random_list  
  loop do yield rand end  
end
```

```
random_list { |i| p i }
```

fancy knowledge - blocks

```
def work(&block)
  (1..10).each { |i|
    block.call('ruby', 'rails', i)
  }
end

work { |a,b,i| puts "#{i}: #{a} on #{b}" }
```


fancy knowledge - inherit from Enumerable

```
class RandomList
  include Enumerable
  def each
    loop do
      r = rand
      yield r
      break if r > 0.8
    end
  end
end

list = RandomList.new
a = list.select { |i| i > 0.2 }
p a
```

fancy knowledge - drb

- Verteilte Ruby Programme
- Ein Server, Viele Clients
- Es wird ein komplettes Ruby Objekt exportiert

fancy knowledge - drb client

```
#!/usr/bin/env ruby
require "drb"
require "readline"
DRb.start_service()
obj = DRbObject.new(nil, 'druby://localhost:9001')
print obj.help
loop do
  begin
    line = readline("#")
    eval line
  rescue NameError, EOFError
    puts "error!"
  end
end
```

fancy knowledge - drb server

```
require "drb"
class TheServer
  def initialize
    @arr= []
    @s="test"
  end
  attr_reader :arr, :s
  attr_writer :arr, :s
  def add (elem)
    @arr.push elem
  end
end
```

continued on next page ...

fancy knowledge - drb server, cont.

```
def help
  return <<EOF
  You can use:
    p obj.methods
    p obj.display
    p obj.inspect
    obj.arr.push "data"
    obj.s = "asd"
EOF
end
end
# start the server
aServerObject = TheServer.new
DRb.start_service('druby://:9001', aServerObject)
DRb.thread.join # Don't exit just yet!
```

fancy knowledge - ruby doc

- Erzeugt html aus .rb Dateien
Aufruf: `rdoc test.rb`
- Ruby Info
Aufruf: `ri Hash`
- Inline Dokumentation
`=begin`
`=end`
Aufruf: `rd2 -r rd/rd2man-lib.rb -o rcguard rcguard.rb`

fancy knowledge - raa

- Enthält viele Ruby Module
<http://raa.ruby-lang.org/>
- Einfache Installation über GEM
<http://raa.ruby-lang.org/project/rubygems/>

fancy knowledge - using c code with swig

Swig ist ein Interface Konverter von C zu vielen Skriptsprachen.
Auch zu Ruby:

```
# wrapper generieren
swig -ruby -module ArpNG arpNG.h
# objekte kompilieren
gcc -c arpNG.c arpNG_wrap.c
      -I/usr/lib/ruby/1.8/i386-linux/
      -D_GNU_SOURCE
# shared object file linken
ld -G arpNG.o arpNG_wrap.o -lnet -o ArpNG.so
# testen
ruby -e "require 'ArpNG'; p ArpNG.methods;"
```


Ressources

- aptitude install rubybook
- “ruby in a nutshell“, O’Reilly
- <http://poignantguide.net/ruby/>
- “Design Patterns“, Erich Gamma
- “Lehrbuch der Objektmodellierung“, Heide Balzert
- “UML kurz&gut“, O’Reilly
- Extending Ruby
http://www.onlamp.com/pub/a/onlamp/2004/11/18/extending_ruby.html
- <http://www.approximity.com/rubybuch2/>

Thank You

