Überblick

- Prinzipien, Konzepte, Ziele
- Methoden
- Blöcke & Closures
- Klassen, Attribute & Objekte
- Build-In Klassen
- Module
- Duck Typing
- Expressions

Prinzipien, Konzepte & Ziele

Produktivität durch Handlungsfreiheit

Dynamisches Typsystem

Dynamische Klassen-, Methoden-, Modul-, Objekt- (re-)definition

Offenheit

Erweiterbarkeit / Redefinierbarkeit von Build-In-Klassen, Methoden, ...

Operatoren sind Methoden => Operator-Redefinition

```
a * b + c  <=> ( a.*( b ) ).+(c)
```

Lesbarkeit / Nachvollziehbarkeit

'Pure' OO ('alles ist Objekt')

```
-23.abs  vs. Math.abs( new Integer(23) )
```

Funktionale Programmierung

```
3.times { print 'odelay' }
```

Prinzipien, Konzepte & Ziele

Einfachheit

'Convention over Configuration (Declaration)'

Erweitertes Expression-Konzept (Kontrollstrukturen sind Expressions)

Keine tiefen Vererbungshierarchien durch Mixins

Best Practices

Pattern in der Sprache angelegt (Visitor, Iterator, Enum., Observer, ...)

Viele Idiome

'Patterns applied' mittels Mixins

=> Ruby als Programmier**er**-Sprache

Methoden

```
def simple_palindrom( word )
    word + word.reverse
end

self.puts( simple_palindrom( 'test' ) )       -> 'testtset'
```

'Convention over Configuration'

- Letztes ausgewertetes Statement ist automatisch Return-Value

- Methodenaufruf mit oder ohne Klammerung von Argumenten

- Zeilenende begrenzt automatisch EIN Statement (kein Semicolon)

=> 'No fluff - Just stuff'

Klassen / Vererbung

```
class ExtendedString < String
    def simple_palindrom()
        self + self.reverse
    end
end

myExtString = ExtendedString .new( 'test' )
puts myExtString.simple_palindrom        -> 'testtset'
```

Einfach-Vererbung

'Implementierungsvererbung', da dynamisches Typsystem

'Mehrfach-Vererbung' durch Mixins

Dynamische Klassenerweiterung

```
class String
   def simple_palindrom
      self + reverse
   end
end

puts 'test'.simple_palindrom        -> 'testtset'
```

Klassen sind niemals geschlossen

Klassen-Objekte sind globale Konstanten

Dynamische Erweiterung von Build-In-Klassen

Literale sind Objekte

Variablen

```
$LieblingsTitel = 'Pragmatic Ruby'
class Buch < Object
   @@instances
   def initialize( titel )
      @titel = titel
      @@instances++
   end
end

myFirstRubyBook = Buch.new($LieblingsTitel)
```

Convention over Declaration:

Variablendefinition dort wo sie zum ersten mal genutzt werden

'Scope by prefix' / 'Scope by location'

Attribute

```
class Buch
   (public) def autor
      @autor
   end
   def autor=(autor)
      @autor = 'Pragmatic ' + autor
   end
   attr_writer : :isbn, :preis, :bewertung
end

myFirstRubyBook.autor = 'Dave Thomas'
puts myFirstRubyBook.autor -> 'Pragmatic Dave Thomas'
```

Setter ~ Überschreiben des Attribut-Zuweisungs-Operators

Attribute-Definition – Shortcuts

Default-Access-Modifier: public (zusätzlich: protected, private)

Dynamische Klassendefinition

```
class Buch
   include Tracing if $DEBUG
  if $Owner == 'myself'
    def verkaufen
      verkaufspreis = @preis + 20
    end
  else
  def verkaufen
    verkaufspreis = @preis+ 10
  end
end
```

Klassendefinition = ausführbarer Code (zur Laufzeit erzeugt)

Einmalentscheidungen werden auch nur einmal getroffen

Erweiterbarkeit der Ausdrucksfähigkeit der Klassendef. (Direktiven)

Beispiel Direktiven / Ausdruckserweiterung

```
def once(*ids) # :nodoc:
  for id in ids
      module_eval <<-"end;"
        alias_method :__#{id.to_i}__, :#{id.to_s}
        private :__#{id.to_i}__
        def #{id.to_s}(*args, &block)
          (@__#{id.to_i}__ ||= [__#{id.to_i}__(*args, &block)])[0]
        end
      end;
  end
end

class ExampleDate
  def as_day_number
    @day_number
  end
  def as_string
    # complex calculation
  end
  def as_YMD
    # another calculation
  end
  once :as_string, :as_YMD
end
```

Operator-Redefinition

```
class Fixnum
   alias old_plus +

   def +(other)
      old_plus(other).succ
   end
end
```

```
1 + 2          -> 4
a = 3
a += 4         -> 8
a + a + a      -> 26
```

Operatoren sind Methoden

Aliasing von bestehenden Methoden

Operator-Definition

```
class Buch

   def text=(text)
     @text = text
   end

   def[ ](fromChar, toChar)
     @text[fromChar .. toChar]
   end
end

myFirstRubyBook.text = 'Dies ist der Text des Buches'

myFirstRubyBook[9,16]    -> 'der Text'
```

Container

Build-In-Klassen: Array und Hash

- Indexed Collections (Key (index) – Value – Paare)
    - basiert wesentlich auf Definition des Index-Operators

- 'Value-Objekte' beliebigen 'Typs' lesezeichen = ['H',3,'A',7,'L']

- Array: Number-Index lesezeichen[2] -> 'A'
    positive und negative Indizes lesezeichen[-2] -> 7
    Range-Indexes lesezeichen[1,3] -> [3,'A',7]
    lesezeichen[1..3] -> [3,'A',7]

- Hash: Index (Key) beliebigen 'Typs'

titel = {'dave'=>'Pragmatic Ruby', 23 => 'Illuminati' }

titel[19+4] -> 'Illuminati'

Container

```
class Regal
   def initialize
     @stapel = Array.new
   end

   def hinzustellen( gegenstand )
     @stapel.push( gegenstand )
   end

  def[ ](index)
      @stapel[index]
   end
end

myRegal = Regal.new
myRegal.hinzustellen(Buch.new('Pragmatic Ruby'))
            .hinzustellen(Zeitschrift.new('Diamonds and Perls'))
            .hinzustellen(Magazin.new('Phytons and other Snakes'))
```

Duck Typing

Typ eines Objekts wird nicht durch seine Herkunftsklasse bestimmt

Typ eines Objekts bestimmt sich durch dessen Schnittstelle

'If it walks like a Duck and if it talks like a Duck, then the interpreter is happy to treat it like a Duck'

```
class Buch
   def append_to_file( file )
      file << @titel << " " << @autor << " " << @isbn
   end
end

class TestAddBook < Test::Unit::TestCase
   def test_add
      book = Buch.new( AspectJ in Action', 'Laddad', 1445118 )
      fileDummy = [ ]
      book.append_to_file( fileDummy )
      assert_equal(['AspectJ in Action','Laddad', 1445118], fileDummy)
   end
end
```

'Singleton'-Objects

Jedes Objekt ist (potentiell) einzigartig

Dynamische Objekt-Erweiterung / -Redefinition

```
meinLieblingsBuch = Buch.new( 'Pragmatic Ruby' )

class << meinLieblingsBuch
   def titel
      @titel + '(Lieblingsbuch)'
   end
end

def meinLieblingsBuch.lieblingsPassage
   @text[ 1035, 1950 ]
end
```

Expressions

Alles was potentiell einen Wert liefern kann

Statements sind Expressions (Statement-Chaining möglich !)

```
a = b = c = d = 0                -> 0
[3,1,7,0].sort.reverse           -> [7,3,1,0]
```

Kontrollstrukturen sind Expressions

```
beurteilung = if book.bewertung < 2 then
                'nicht lesenswert'
              elsif book.bewertung >= 2 && book.bewertung < 5
                'lohnt einen Blick' ...
              else
                'Hervorragende Lektüre'
              end
```

Boolesche Expressions (alle Werte != nil bzw. != false sind true)

&&, || liefern den ersten Wert, welcher den Ausdruck entscheidet

```
nil && true              -> nil
99 and false             -> false
23 and 'Illuminati'      -> 'Illuminati'

buecher[autor] ||= [ ]      <=>    buecher[autor] = buecher[autor] || [ ]
```

Blöcke & Iteratoren

vgl. Funktionale Programmierung

Block = 'anonyme', geschlossene Menge von Code

Block kann mit einem Namen / Variablen assoziiert (Proc) und wie jedes andere Objekt behandelt werden

```
class Buch
   def each_chapter_title( &block )
      # retrieve each chapter title out of @text
      block.call( current_chapter )
end

inhaltsverzeichnis = [ ]
meinLieblingsBuch.each_chapter_title do |chapter|
   inhaltsverzeichnis << chapter
end
```

Blöcke & Iteratoren

```
Class Regal
   attr_reader : stapel
end

sum_preis = 0
myRegal.stapel.each{ |item| sum_preis += item.preis }

myRegal.stapel.find{ |item| item.titel == 'Diamonds and Perls' }

bewertungen = myRegal.stapel.collect do |item|
   item.bewertung if item.response_to?( :bewertung )
end
bewertungen   -> [5, 4, 7]

4.upto( 6 ) { |i| print 'Zum ' + i + '. ' }
'Zum 4.' 'Zum 5.' 'Zum 6.'
```

Closures / Lambda

Closures sind Blöcke, welche ihren Kontext, in welchem sie definiert wurden beibehalten

Lambda-Operator zur expliziten Konvertierung eines Blocks / Closures in ein Proc-Objekt

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

meinLieblingsBuch.preis = 50

buchVerkauf = n_times( meinLieblingsBuch.preis )
buchVerkauf.call( 5 )
```

-> 250

```
buchTitelBekanntmachung = n_times( meinLieblingsBuch.titel )
buchTitelBekanntmachung.call( 3 )
```

-> 'Pragmatic Ruby Pragmatic Ruby Pragmatic Ruby'

Module

Möglichkeit zur Bildung von Namensräumen

```
module Amazon
  EIN_STERN   = 1 ...

  def beurteilung( bewertung )
    if bewertung < 2 then
      EIN_STERN ...
    else
      FUENF_STERNE
    end
  end
end

module Bol
  def beurteilung( bewertung )
    if bewertung < 2 then
      'nicht lesenswert' ...
    end
  end
end


beurteilung_A = Amazon.beurteilung( book.bewertung )
beurteilung_B = Bol.beurteilung( book.bewertung )
```

Mixins

Inkludieren (referenzieren) von Modulen in Klassen -> 'einmischen'

Modul verhält sich wie Superklasse (~Mehrfachvererbung)

Interaktion mit Klassen-Elementen (Instanz-Methoden, -Variablen)

```
class Buch
  include Comparable      # define <=> and get <,<=,==,>= and > for free

  def <=>(other)
       self.isbn <=> other.isbn
    end
end

book1 <=> book2     -> -1
book1 <= book2      -> true
book1 == book1      -> true


class Buch
    if $PORTAL == 'Amazon'
        include Amazon
    elsif $PORTAL == 'Bol'
        include Bol
    end
end
```

Mixins

Modul-Instanz-Variablen 'werden zu' Klassen-Instanz-Variablen im Rahmen der dynamischen Erzeugung des Klassen-Objekts (zur Laufzeit)

```
module Observable

  def observers
      @observer_list ||= [ ]
    end

  def add_observer( obj )
      observers << obj
    end

  def notify_observers( *messages )
      observers.each{ |observer| observer.notify( messages ) }
    end
end


class Regal
   include Observable

  def hinzufuegen( gegenstand )
      ...
      notify_observers( self, :hinzufuegen, gegenstand )
    end
end
```

... und noch vieles mehr

- Exception-Handling (RuntimeExceptions, rescue, ensure, retry)
- Reflection
- Threads und Prozesse
- Security (Safe Levels, Tainted Objects)
- Build-Ins / Bibliotheken / Pakete
    - Reguläre Ausdrücke
    - I/O
    - Ruby TK
    - Net / Distributed Ruby / Web
    - XML / SOAP / WebServices
- Frameworks
    - RUnit
    - Rails
- Ruby Gems (Package-Management)
- Interactive Ruby Shell (irb)
- Documentation (rdoc)
- ...

Quellen

- Dave Thomas
  Programming Ruby – The Pragmatic Programmers' Guide
  Pragmatic Bookshelf
  http://www.rubycentral.com/book/
  http://home.vrweb.de/~juergen.katins/ruby/buch/ (Übersetzung)

- Ruby Central
  http://www.rubycentral.com/

- Ruby Homepage
  http://www.ruby-lang.org/

- Why's Guide to Ruby
  http://www.poignantguide.net/ruby/index.html