



Ruby

„Ruby is designed to make
programmers happy.“

- Yukihiro Matsumoto -

Programmiersprachenseminar SS 2011

geleitet von Dr. Jutta Göers

vorgetragen von:

Manuel Schwarz



- Entstehungsgeschichte
- Sprachmerkmale
- Syntax
- Konzepte
- Ruby vs. Java
- Fazit
- Quellen



Geschichte

- Entwickler: Yukihiro „Matz“ Matsumoto
- erschienen 1995
- ~2000 Bekanntheit außerhalb Japans

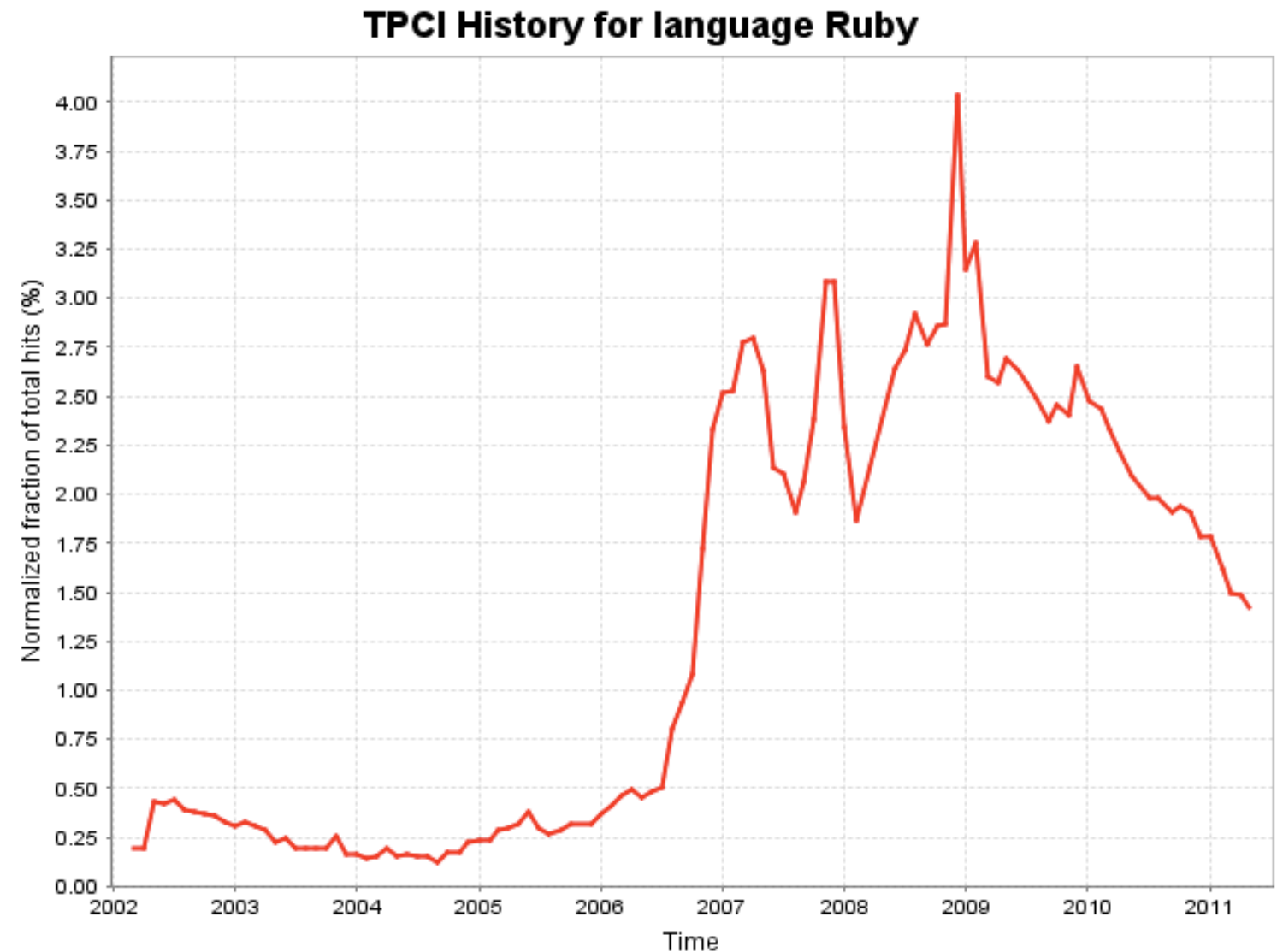
Einflüsse

- Lisp, Smalltalk and Perl
- Einklang vom funktionalen und imperativen Programmieren



TIOBE-Index

- 10. Position
(Stand: Mai 2011)
- Anteil ca. 1.4 %
- 2006 Programming
Language of the
year (18. Position)



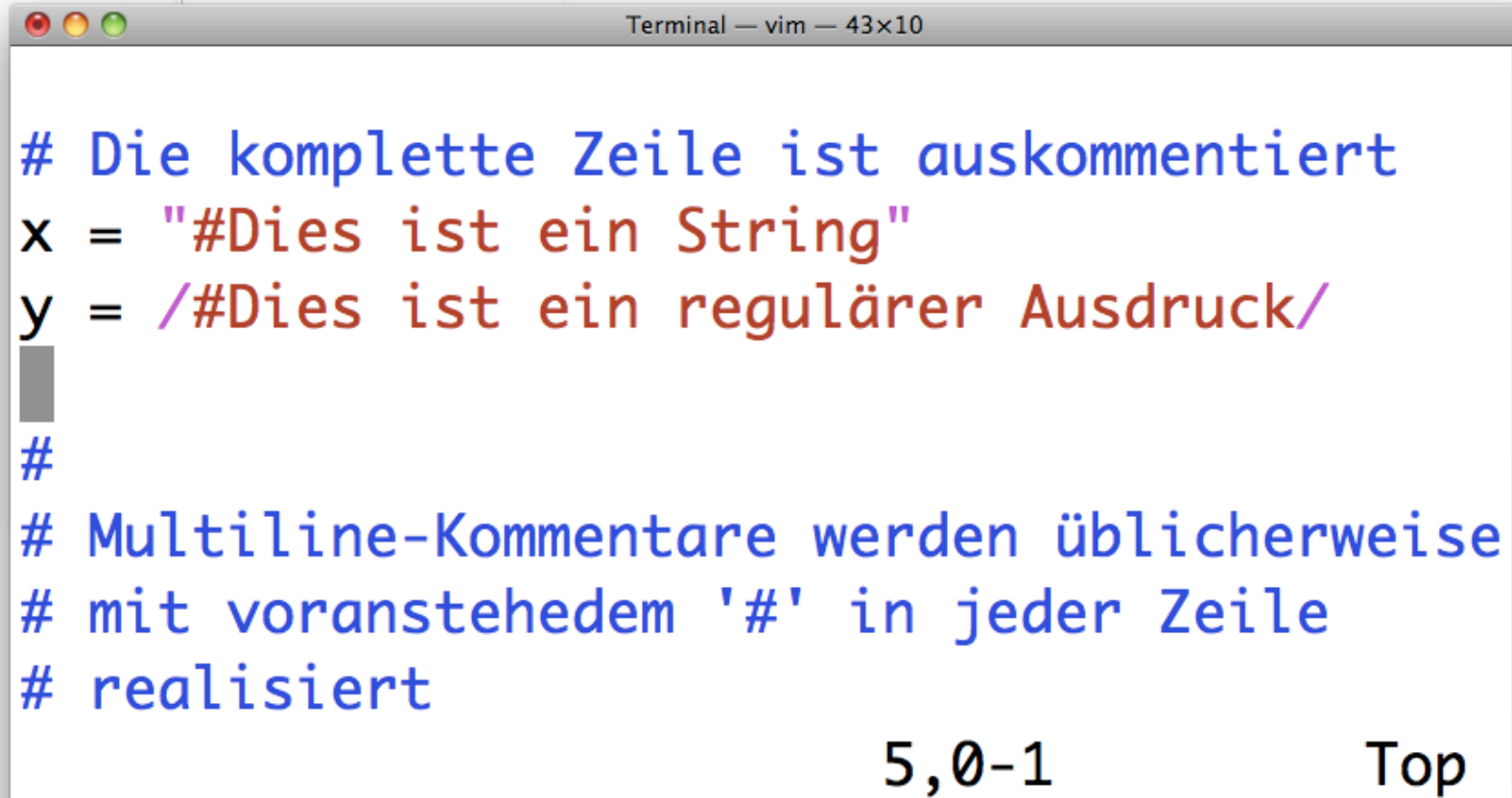


Besonderheiten

- Objektorientierung (alles ist ein Objekt)
- Garbage-Collection
- dynamische und starke Typisierung
- Operator-Überladung
- Block-Syntax
- Duck-Typing
- Metaprogramming



Kommentare



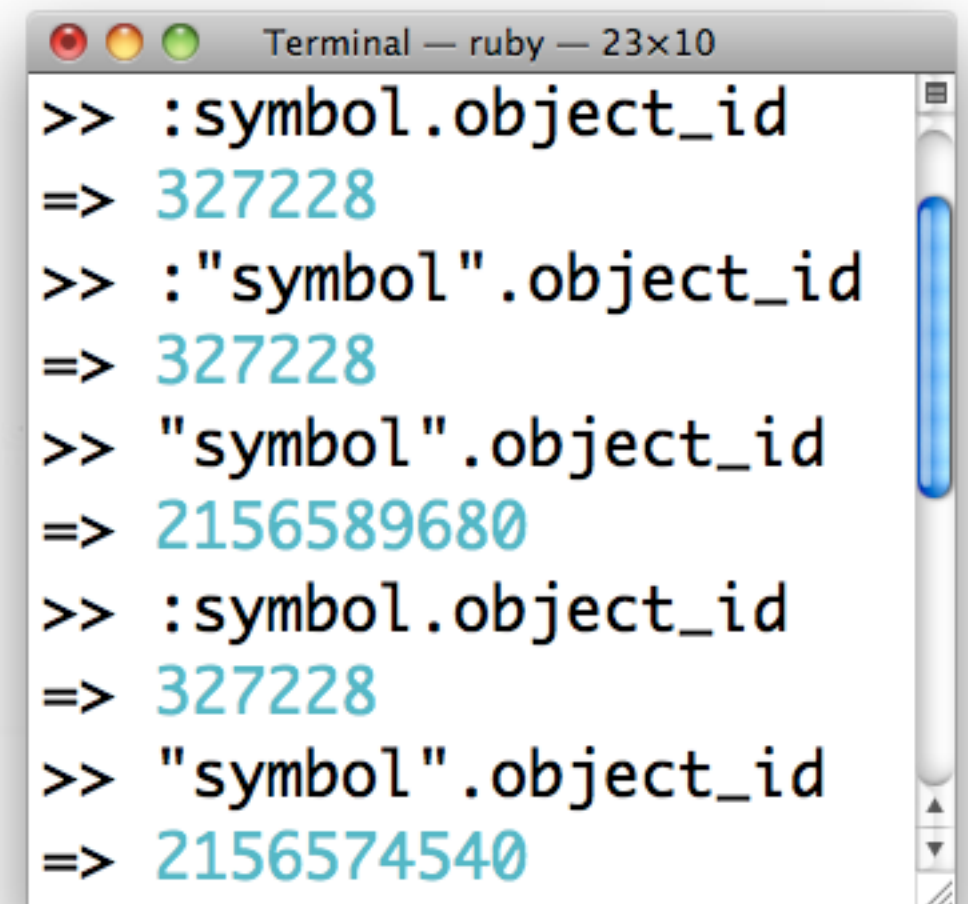
```
Terminal — vim — 43x10
# Die komplette Zeile ist auskommentiert
x = "#Dies ist ein String"
y = /#Dies ist ein regulärer Ausdruck/
#
# Multiline-Kommentare werden üblicherweise
# mit voranstehendem '#' in jeder Zeile
# realisiert
5,0-1 Top
```

- beginnen mit einem `#`
- bis zum Ende der Zeile
- keine Inline-Kommentare à la `/* ... */`



Basisdatentypen

- 42 ist ein Integer (Fixnum)
42.even? liefert TRUE
- 3.14 ist vom Typ Float
- „Hallo“ ist ein String mit Inhalt Hallo
- :symbol ist ein „konstanter String“

A screenshot of a terminal window titled "Terminal — ruby — 23x10". It shows the following commands and their outputs:

```
>> :symbol.object_id
=> 327228
>> "symbol".object_id
=> 327228
>> "symbol".object_id
=> 2156589680
>> :symbol.object_id
=> 327228
>> "symbol".object_id
=> 2156574540
```



Collections

- `[]` ist ein leeres Array \Leftrightarrow `Array.new`
- `a = [3, 7]` ist ein gefülltes Array
- `a[0]` gibt den ersten Eintrag zurück
- `{ }` ist ein leerer Hash
- `h = { :key=>„value“, :key2=>„value2“ }`
- `h[:key]` liefert den Wert für den Schlüssel `:key` zurück



Operatoren

```
Terminal — vim — 57x10
1 + 2 # => 3: Addition
2 * 3 # => 6: Multiplikation
7 % 4 # => 3: Modulo
1 + 2 == 3 # => true: == testet Gleichheit
"a" == "a" # => true
"a".equal? "a" # => false: equal? testet Identität
2 ** 10 # => 1024: Potenzieren
max = x > y ? x : y # Conditional Operator
```

9,0-1 All

- kein ++ bzw. -- aber: +=, -=, usw.
- kein Überladen von Zuweisungsop. „=“



Methoden & Variablen

```
def method
  lokaleVariable1 = 42  #dynamisch typisiert
  lokaleVariable2 = "Hallo Welt!"
  KONSTANTE = 3.14

  @instanzVariable = "in der Instanz sichtbar"
  @@klassenVariable = "vgl. static in Java"
  $globaleVariable = "überall sichtbar"
end
```

6,0-1

Top

Impliziter Rückgabewert der Methode
method ist der String „überall sichtbar“



Klammern

- Weglassen von Klammern bei Eindeutigkeit
- `[3, 7].include? 3 <=> [3, 7].include? (3)`
- `methode :key => „value“`
 `<=>`
 `methode ({:key => „value“})`



Zuweisungen & Range

● Parallelzuweisungen

```
Terminal — vim — 49x12
x, y = 1, 2      # <=> x = 1; y = 2
a, b = b, a      # Werte vertauschen
x,y,z = [1,2,3]  # autom. Zuweisung v. Arrayelem.
█
1..10            # inklusive 10
1...10           # exklusive 10
'a'..'z'         # inklusive z
'a'...'z'        # exklusive z
(1..10) === 5     # => true
(1..10) === 10    # => true
(1...10) === 10   # => false
```

4,0-1

All



Case-Anweisung

- `===` prüft auf case-Gleichheit

```
Terminal - vim - 49x11
a = 1..10
b = 1..4
c = 5
case c
  when a # umformuliert: wenn c in a enthalten...
    # Code wird ausgeführt
  when b # wenn c in b enthalten ist...
    # Code wird nicht ausgeführt
end
4,0-1 All
```



Blöcke & Iteratoren

- Übergabe von Codeblöcken an Methoden

```
Terminal — vim — 20x8
10.times do
  print "Hallo! "
end

(1..10).each do |i|
  puts i*i
end

4,0-1 Top
```

```
Terminal — vim — 29x8
[1,2,3,4,5,6,7,8].each do |x|
  if x.even?
    print x
  else
    print "... "
  end
end

14,1 50%
```



Reguläre Ausdrücke

- `/regexp/ <=> Regexp.new („regexp“)`
- Muster, das auf einen String angewendet werden kann
- `/[Rr]uby/ # „Ruby“, oder „ruby“ => true`
- `/\d{5}/ # 5 aufeinanderfolg. Ziffern`
- **DEMO**



Klassen

- Sammlung von Methoden, die auf dem Zustand eines Objektes operieren können
- Zustand durch Instanzvar. repräsentiert

```
Terminal — vim — 70x15
class Person
  attr_accessor :name, :age, :footsize    #fügt getter und setter ein

  # Konstruktor
  def initialize(name, age, footsize)
    @name, @age, @footsize = name, age, footsize
  end

  # to_String-Methode geerbt von Object
  def to_s
    "Name: #{@name}\nAlter: #{@age}\nSchuhgroesse: #{@footsize}"
  end
end

8,1      All
```




Module

- können Konstanten, Klassen, Methoden, usw. beinhalten
- können nicht instanziiert werden
- man kann nicht von ihnen erben
- `Class < Module`
- Module mit Modulmethoden bilden Namespaces
- Module mit Instanzmethoden sind Mixins



Module als Namespace

```
Terminal — vim — 50x11
1 module Mein_Modul
2   def self.modulmethode
3     puts "Eine Modulmethode!"
4   end
5
6   def Mein_Modul.zweite_modulmethode
7     puts "Noch eine!"
8   end
9 end
10 Mein_Modul.modulmethode #=> Eine Modulmethode!
                             5,1           All
```



Module als Mixins

```
Terminal — vim — 20x6
module Mein_Modul
  def meine_methode
    puts "Hello!"
  end
end
4,1 All
```

- „include“ zum Einbinden
- anderen Klassen Methoden zur Verfügung stellen

```
Terminal — vim — 46x5
class MeineKlasse
  include Mein_Modul #<== Einbinden des Moduls
end
MeineKlasse.new.meine_methode #=> Hello!
2,1 All
```



Vererbung

- keine Mehrfachvererbung
- Simulation mit Mixins

```
Terminal — vim — 47x13
class Student < Person          # erbt von Person
  attr_accessor :matrikel

  def initialize(name, age, footsize, matrikel)
    super(name, age, footsize)
    @matrikel = matrikel
  end

  def to_s                      #überschreibt to_s
    "#{super.to_s}\nMatrikel: #{@matrikel}"
  end
end

8,1                               All
```



Duck-Typing

- Gegenkonzept zu typisierten Sprachen
- Behandlung eines Objekts nach seiner Funktionalität (implementierte Methoden)
- Klassenzugehörigkeit egal
- nicht der Typ, sondern die Funktionalität ist entscheidend
- **BEISPIEL**



Metaprogramming

- Writing Code, that writes code.
- Code, der Sprachkonstrukte (Klassen, Module, Instanzvar.) zur Laufzeit ändert
- Code zur Laufzeit schreiben und ausführen ohne Programmneustart



Gemeinsamkeiten

- Garbage-Collector
- Objekte sind stark getypt / typisiert
- Methoden können `private`, `public` oder `protected` sein
- Dokumentationswerkzeug (RDoc anstatt JavaDoc)



Unterschiede (1)

- Interpretieren statt Kompilieren
- alles ist ein Objekt
- Blöcke (`do, end` anstatt `{ }`)
- Klammern bei Methodenaufruf optional
- dynamic typing
- alle Instanzvariablen sind privat



Unterschiede (2)

- `require` **anstatt** `import`
- kein Casting
- `foo = Foo.new("hi")` anstatt `Foo`
`foo = new Foo("hi")`
- Mixins statt Interfaces
- `nil` **statt** `null`
- `==` und `equal?()` „vertauscht“



- aktive Community
- viele Tutorials, Beispiele
- intuitiv (POLS)
- leichter Umstieg von Java / C
- gute Literatur (siehe Quellen)
- große API, gute Dokumentation
- Plattformunabhängig, kostenlos

Start with Ruby



- <http://www.ruby-lang.org>
 - Try Ruby (im Browser)
 - Download
 - Ruby in 20 minutes (Einführung)
 - viele weitere Links
- <http://ruby-doc.org>
 - Dokumentation



Literatur

- Flanagan, David / Matsumoto, Yukuhiro: The Ruby Programming Language. 1. Auflage. O'Reilly Media Inc., 2008
- Thomas, Dave / Fowler, Chad / Hunt, Andy: Programming Ruby 1.9 - The Pragmatic Programmers' Guide. 3. Auflage. Pragmatic Programmers, 2009
- Groh, Jens: Ruby. Erschienen in: Henning, Peter A. / Vogelsang, Holger: Taschenbuch Programmiersprachen. 2. Auflage. Carl Hanser Verlag, 2007, S. 486-507

Internetquellen

- <http://www.ruby-lang.org>
- <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

Vielen Dank!



Vielen Dank für Ihre Aufmerksamkeit

Fragen?