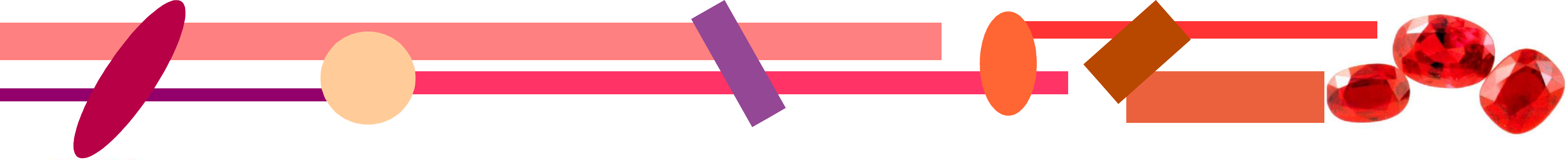
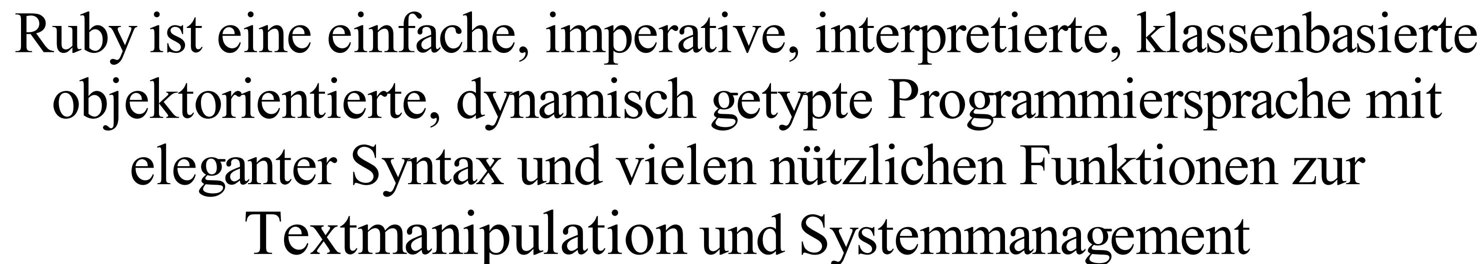


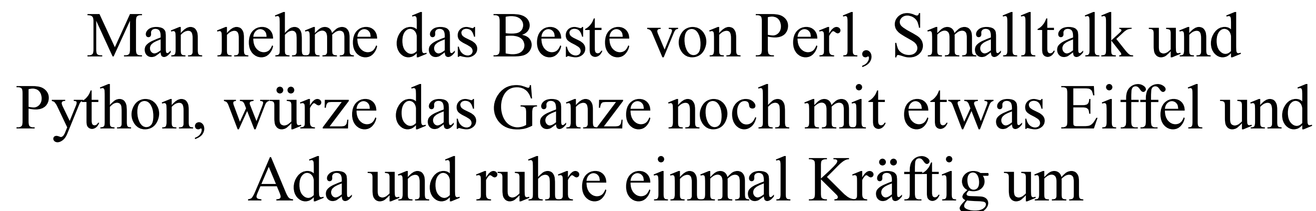
Ruby

Eine syntaktisch relativ saubere, einfache und zugleich unglaublich elegante und mächtige Sprache von Yukihiro Matsumoto

Lange nachdem „mats“ die Sprache Ruby genannt hatte, stellte er fest, dass Der Rubin der Nachfolger der Perle ist (in der japanischen Astrologie ö.Ä)



oder



- vollständig objektorientiert (alles ist ein Objekt), dynamisch getypt
- OO ist Klassenbasiert (Einfachvererbung, Mix-Ins von Modulen als Ersatz für Mehrfachvererbung, Polymorphie)
- hat Module
- Eingebaute Klassen Regex, Range, Hash, Array, String, ... und Literalschreibweise für selbige.
- hat eine Operatorsyntax
- unterstützt Blöcke
- unterstützt Exceptionhandling
- unterstützt beliebig grosse Zahlen
- echter mark and sweep Garbage Collector
- portabel
- Spracheigene Threadunterstützung
- leicht erweiterbar in C
- Continuations und Bindings

Entwicklungsumgebung RDE

RDE ist eine IDE, keine spezielle aber RDE ist durchaus recht nützlich. Sie bietet neben einem Debugger einige Merkmale, die die Programmierung vereinfachen. Sie leidet aber unter der dynamischen Typisierung Rubys. Dadurch kann sie nur selten Methoden vorschlagen

Kontrollstrukturen

Ruby bietet die üblichen Kontrollstrukturen einer imperativen Programmiersprache. Daneben gibt es noch den aus Perl bekannten if- und unless-Modifikator.

```
# Verzweigung mehrere Bedingungen
if( preis > 200 )
    puts "ade"
elsif( geld > preis )
    puts "kaufen"
else
    handeln();
end
```

```
# Mehrfachauswahl
case tagname
    when "br"
        br;
    when "h1", "h2", "h3"
        head;
    when 2
        # keine Anw.
    else
        fehler("k.A.");
end
```

```
# For-Schleife
for i in 1..393
    # Negatives if
    unless i%2==1 then
        # Alle weiteren Anw. übergehen
        next
    end
    # Anw. zum Verlassen der Schlefe
    # mit if-Modifikator
    break if (i % 5==0)
    # Anw. um Schleife erneut zu
    # beginnen mit unless-Modifikator
    redo unless frage(i) !=
"angebot"
end
```

```
# Endlosschleife
z = "AAA";
loop do
    # Operiere auf z, AAA=>AAB...
    puts z.succ!
    # Abbruch bei ABA
    break if z=="ABA";
end
```

```
# Kopfgesteuertes Until
a = c= 5;
until a==c
    puts "dd: #{a+=1}";
end

# Fussgesteuertes While
a = c = 5;
# Blockkonsturkt mit while-Modifikator
begin
    puts "lauf";
end while (a != c)

# puts mit until-Modifikator
i=7;
puts i+=1 until i>22;
```

```
# While-Schleife
a=[];
while a.size<20
    a.push(a.size);
end
# a ausgeben
p a.inspect
```

Grundsätzliches

- Kommentare arbeiten Zeilenweise. Alles hinter # wird ignoriert.
- Ruby arbeitet primär Zeilenorientiert. das lässt sich durchbrechen mittels \, *Operator*, *Komma* als letztes Zeichen einer Zeile.
- (Fast) alles ist eine Anweisung.
- Jede Anweisung evaluiert zu etwas. Also fast alles ist ein Ausdruck.
- Automatische Konvertierungen werden soweit mögl. vorgenommen.
- Semikola sind optional. Bei mehr als einer Anw. Pro Zeile erf.

Ruby hat eine ganze Reihe in den Interpreter eingebaute Klassen. Diese sind wieitestgehend optimiert.

Liste der Klassen

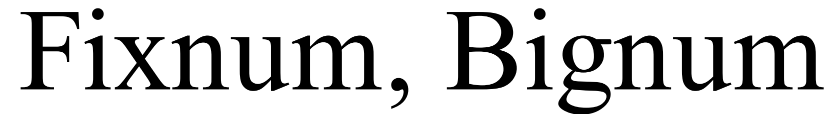
- Array
- Bignum
- Binding
- Class
- Continuation
- Dir
- Exception
- FalseClass
- File::Stat
- File
- Fixnum
- Float
- Hash
- Integer
- IO
- MatchData
- Method
- Module
- NilClass
- Numeric
- Object
- Proc
- Range
- Regexp
- String
- Struct
- Struct::Tms
- Symbol
- ThreadGroup
- Thread
- Time
- TrueClass

Mehrfachzuweisung

Ruby bietet neben einer einfachen Zuweisung noch die Mehrfachzuweisung. Sie funktioniert atomar, so dass man mit einer Anweisung den Ihnalt zweier Variablen tauschen kann. Man kann sich die Mehrfachzuweisung wie mit Arrays realisiert vorstellen.

```
# die Parallelzuweisung
a,b,c = 1, 2.0, "drei"
puts (a,b,c);     # -> 1  2.0   drei
# Tausch von Variableninhalten
a,b = b,a;
puts (a,b);       # -> 2.0   1
# Verschieden viele Variablen
e,f,g = b,c
puts (e,f,g)      # -> 1  drei  nil
# kombiniert mit Mehrfachzweisung
b,e,a = (c = d = 5)-2, "txt", c-3;
puts (a,b,c,d,e) # -> 2  3  5  5  txt
# Verschachtelte Zuweisungen
a,(b,*c),d = 11,[2,3,4,5],66
p [a,b,c,d].inspect #-> [11, 2, [3, 4, 5], 66]
```

```
# unpassende Variablenzahl
a,*b = 1,2,3,4
puts [a,b].inspect #-> [1 [2, 3, 4]]
c,d = b,3,a,5      # a und 5 wird verschluckt
puts [c,d].inspect #-> [[2, 3, 4], 3]
c,*d = 0,b,5
puts [c,d].inspect #-> [0, [[2, 3, 4], 5]]
(d,e,f),a = b,5    # b wird expandiert
p [d,e,f,a].inspect#-> [2, 3, 4, 5]
a,c,d,*e = 77,*b
puts [a,c,d,e].inspect #-> [77, 2, 3, [4]]
# weitere Varianten
a,(b,c) = 5,[3,4]  #-> 5, 3, 4
a,b,c = 5,[3,4]  #-> 5, [3, 4], nil
```



In Ruby können Zahlen beliebig groß werden. Die Intention dabei ist, dass sich der Anwender keinerlei Gedanken über Bitbreiten etc. machen soll.
Zahlen, die die Maschine verarbeiten kann, sind dabei von Typ Fixnum. Größere werden automatisch zu Bignum

```
puts 99.class               #->Fixnum
puts 3874889983.class       #-> Bignum

n = 5555;
puts n.class;               #->Fixnum
n *= n*n;
puts n.class;               #->Bignum
n -= n;
puts c=n.class;             #->Fixnum
puts c.ancestors.inspect
#->[Fixnum,Integer,Precision,Numeric,Comparable,Object,Kernel]
```

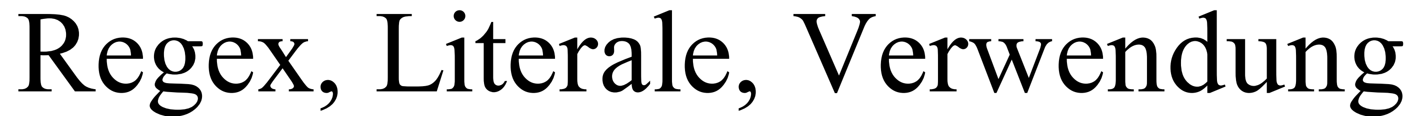
Daneben gibt es noch eine ganze Reihe nützlicher Mehtoden in den Basisklassen.

```
# Bitschiebereien
a = 0xF
b = 0011
puts a       #->15
puts b       #->9
c= a&b
puts c       #->9
puts c|32    #->41
puts (1^1 , 0^1, 1^0, 0^0 )
#->  0  1  1   0
```

Strings, Literale, Verwendung

- Strings sind in Ruby veränderbare Zeichenketten aus 8-Bit-Zeichen mit dynamischer Länge.
- Strings sind das Mittel der Wahl um Binärdaten zu schreiben.
- Gleiche Strings in unterschiedlichen Objekten haben unterschiedliche IDs.
- Universelle Begrenzer / HEREDOC – interpretiert oder literal
- Variablenexpansion mittels #{}

```
inh = "Inhalt"
p inh.class       #->String
p inh.id          #->20675016
zweit = "ist #{inh.reverse}" #->"ist tlahnI"
p 'Inhalt'.id == inh.id  #-> false
p ohne = 'ohne #{inh}'    #->"ohne #{inh}"
heredoc = <<INTERPREtierteHD
blablaba
{/$%2$Dgo #{inh} \\usw.
INTERPREtierteHD
p heredoc #->"blablaba\n{/$%2$Dgo Inhalt \usw.\n"
# Universelle Begrenzer
dd= %q/String,wie,mit,einfachen,Anführungszeichen/
%Q!String wie mit doppelten Anführungszeichen!
%Q{Sek: #{24*60*60}} #-> "Sek: 86400"
p dd.split(",").class  #-> Array
```



Regex, Literale, Verwendung

Der Sprachumfrang von Ruby enthält Literale für Reguläre Ausdrücke. Wie auch in JavaScript und Perl werden sie in Schrägstriche (/) eingeschlossen. Neben der einfachen, perl-kompatiblen, Verwendung mit globalen Variablen $1, $2, .. existiert noch die objektorientierte Ruby-Variante mit Match-Objekten.
Reguläre Ausdrücke sind ein Mächtiges Werkzeug, das unbedingt zum Standardreportoire eines Informatikers gehört.

```
# Perl-artig
if( "bl(3949h)ub" =~ /\((\d*).?\)/ )
    puts "Extrahiert: "+$1; #->3949
end
puts $`+"-" + $& + "-"+$'; #->bl-(3949h)-ub
```

```
# OO-Ansatz
re=/(.)(.)(.)/
puts re.source;     #-> (.)(.)(.)
md = re.match("abc");
p md
puts md[0]+"  "+md[2]; #->abc b
# Zeichenketten indizieren
str="Tri too trallala";
re= Regexp.new("too");
str[re]="tra";
puts str      #-> Tri tra trallala
```

Ranges, Literale, Verwendung

Wie Python kennt auch Ruby sog. Ranges. Dabei handelt es sich um eine abstrakte Darstellung einer Auflistung. Will man sich auf die Zahlen 4-99 beziehen, dann kann man entweder alle 95 auflisten oder eben diesen Bereich angeben (im Vertrauen, dass der Andere zählen kann). Ruby hat auch für Ranges eine literale Darstellung: Aufzänlbare (Mix-In Comparable) Objekte durch .. (zwei Punkte) verbunden ergeben eine Range mit Ende inklusive. Objekte verbunden durch ... (drei Punkte) ergeben eine Range ohne das angegebene Ende.

```
bereich = 4..8;
p bereich              #-> 4..8
p bereich.to_a         #-> [4, 5, 6, 7, 8]
anders = 'a'..'g'
p arr=anders.to_a;  #-> ["a", "b", "c", "d", "e", "f", "g"]
p arr[2..4];           #-> ["c", "d", "e"]
p arr[-3..-1];         #-> ["e", "f", "g"]
p (8..-3).to_a
# bisher inklusive, nun exklusive ende
range = -5...1;
p range
p (('d'...'g').to_a)     #-> ["d", "e", "f"]
puts range.first            #-> -5
puts range.last             #-> 1
puts range.exclude_end?  #-> true
# Dieses Spiel funktioniert mit jeder Klasse mit Comparable
```

Arrays, Literale, Verwendung

Wie jede anständige Sprache, so hat auch Ruby Arrays. Sie lassen sich auf mehrere Weisen als Literale schreiben, aber auch 'normal' indiziert verwenden.

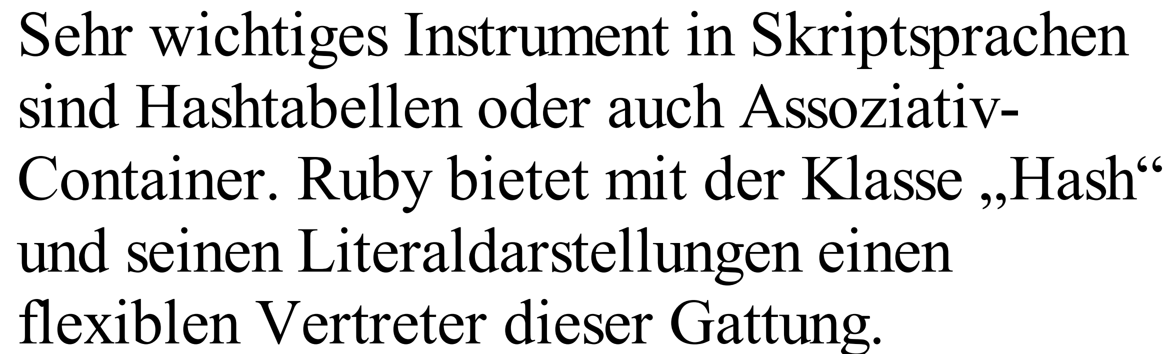
Ruby-Arrays sind ziemlich flexibel: Neben dynamischem Wachstum erlauben sie auch eine Verwendung als Stapel, Warteschlange oder Menge+Operationen.

Ein Array speichert immer Objektreferenzen.

Indizes beginnen bei 0. Negative Indizes zählen von hinten her. Mittels Ranges Indizierte Arrays liefern Subarrays.

```
A = [];
a.push(„blub“);
b=
a*b   schnittmenge
c = %w( a g v s);
c.size  #-> 4
c[8]=Array.new
```

Hashes, Literale, Verwendung



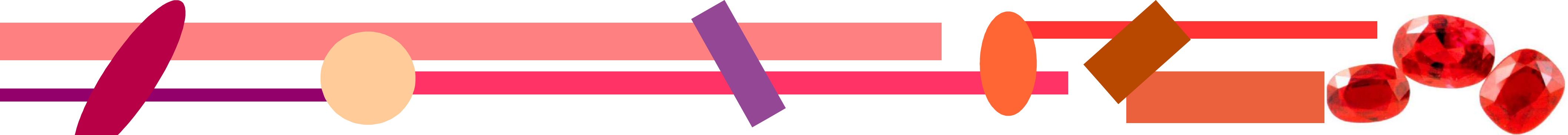
Sehr wichtiges Instrument in Skriptsprachen sind Hashtabellen oder auch Assoziativ-Container. Ruby bietet mit der Klasse „Hash“ und seinen Literaldarstellungen einen flexiblen Vertreter dieser Gattung.

H = {a=>4, 'ffs'=>“ghx“ };
H.size

Ein von Smalltalk stammendes und in Ruby wichtiges Sprachmittel sind die sog. Codeblöcke.

```
X= {|g| „Proc-objekt #{g}“}

[5, 4, 3, 6].each {|n| n*n }
#-> 25 26 9 36
```

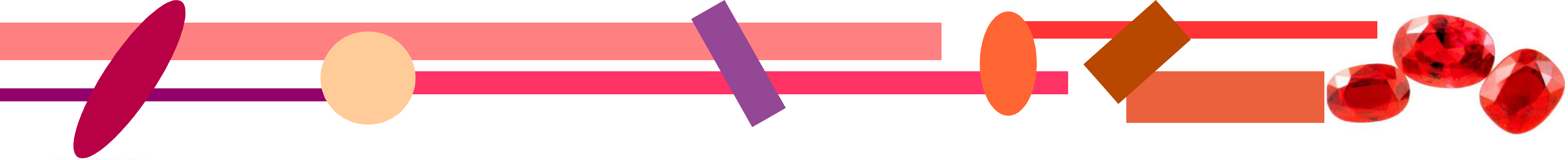


Module

Klassen

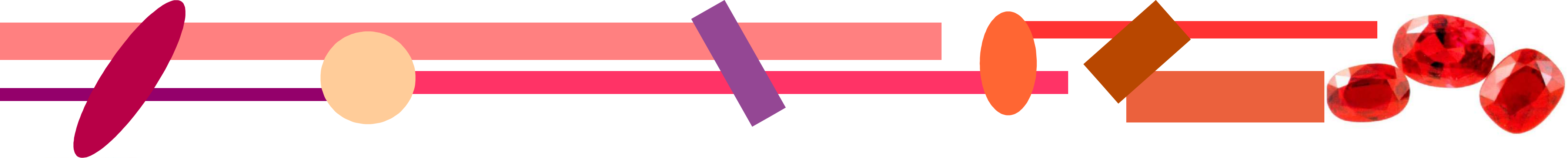
Ruby ist eine objektorientierte Sprache. Ein Weg, neueartige Objekte zu definieren sind Klassen. Da bei Ruby wie Smalltalk alles ein Objekt ist, ist auch eine Klasse ein Objekt. Nämlich Exemplar der Klasse Class. Alle Klassen befinden sich in einer Erbhierarchie, an deren Spitze die Klasse Object steht. Sie bringt allen Objekten die grundliegenden Methoden wie (to_s, class, id, members, ...)

Erben, Mix-In,

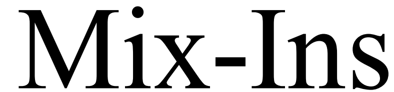


Vererbung

Wie jede Klassenbasierte OO-Sprache beherrscht auch Ruby Vererbung. Es handelt sich dabei um Einfachvererbung. Da es bei Ruby jedoch nur um das Vorhandensein einer Methode ankommt, ist eine Mehfachvererbung recht leicht per sogenanter Mix-Ins möglich.



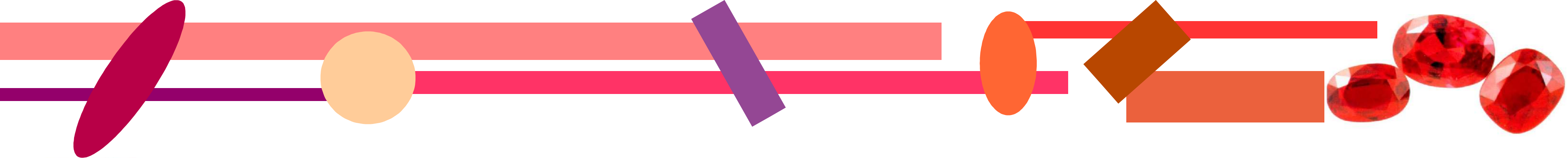
Dynamische Klasenredefinition



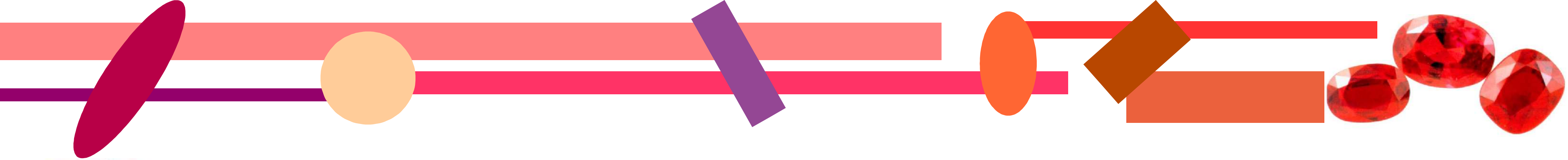
Mix-Ins

Ruby unterstützt keine Interfaces und muss es auch nicht. Mit Mix-Ins gibt es ein viel flexibleres Konzept für eine dynamische Sprache. Das geht zwar auf Kosten der Geschwindigkeit, doch die ist so oder so nicht vergleichbar mit einem Compiler.
Mix-Ins sind Module. Die bestehen bekanntlich aus Funktionen und Variablen. Mit „include“ kann man nun ein Modul in eine Klasse oder ein anderes Modul hineinmixen.

```
Class Indizierbar
    def <=>(a,b) end
    include Comparable
end
```



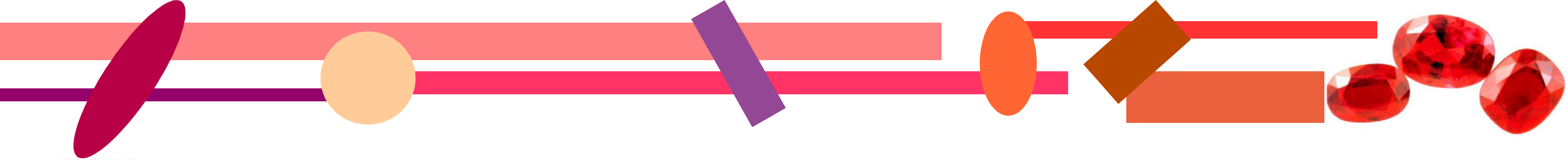
Singletonmethoden



Reflexion

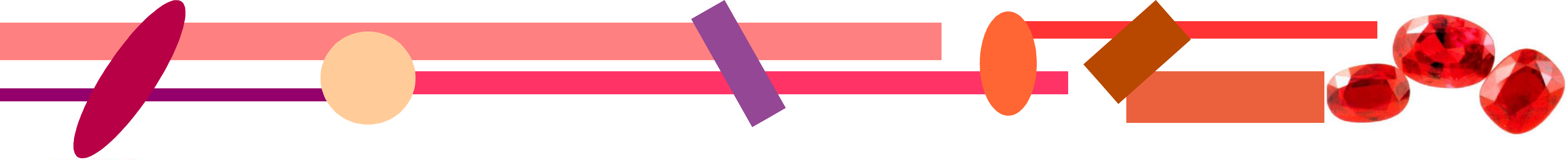
Ruby besitzt eine Vollständige Reflexion. Man kann den gesamten Namens- und Objektraum von Ruby zur Laufzeit durchsuchen und auch modifizieren.

p String.members
-> gibt alle Methoden von String aus



Continuations

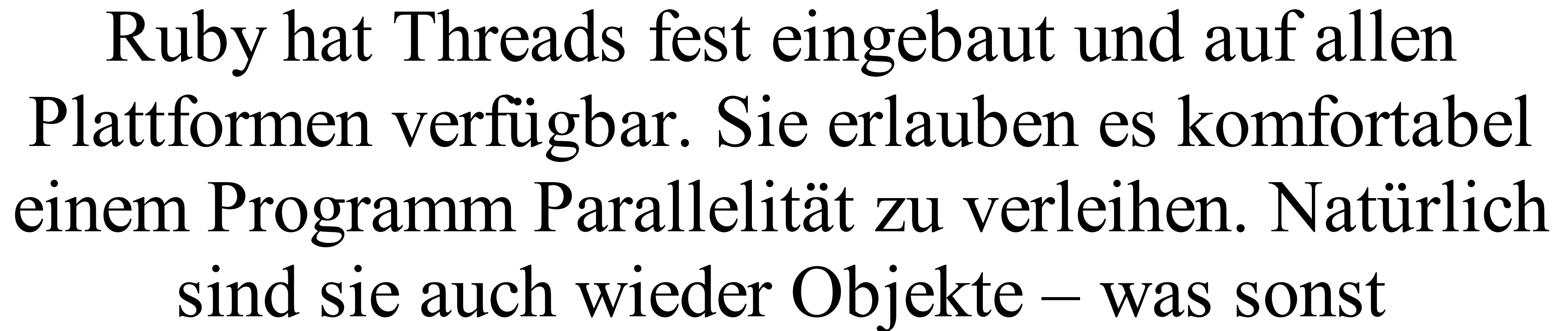
Mit Continuations kann man den Augenblicklichen Programmfluss + Zustand einfrieren und zu gegebener Zeit fortsetzen



Bindings

Damit kann man den augenblicklichen Kontext „Abfotografieren“ und irgendwann eine Funktion in diesem gespeicherten Kontext ausführen.

Intrinsisches Multithreading


Ruby hat Threads fest eingebaut und auf allen
Plattformen verfügbar. Sie erlauben es komfortabel
einem Programm Parallelität zu verleihen. Natürlich
sind sie auch wieder Objekte – was sonst

Ein heutzutage wichtiges Sprachmittel für bessere Programme sind Exceptions. Ruby unterstützt auch Exceptions als intrinsisches Sprachmittel. Man kann auf diese Weise den Programmfluss schreiben als gäbe es keine Fehler. Passiert doch etwas, wird das Programm an einer definierten Stelle fortgesetzt und kann aufräumen.

Schnittstellen

Ruby hat inzwischen 10 Jahre auf dem Buckel und das sieht man auch an den Schnittstellen. Es gibt Schnittstellen zu allen wichtigen Dingen.
z.B. DBI, GTK, QT, TK, FOX, COM, libsgml, ...

Erweiterungen in C gibt's auch und die sind nichtmal schwer zu programmieren. Na und SWIG gibt es auch.