Comparing and introducing Ruby

Michael Neumann

5. February 2000

If questions occur or you discover errors or if you have improvement suggestions you can write me an email: `<neumann@s-direktnet.de>`.

Informations about Ruby and the Ruby interpreter as well as many libraries are availabe from the official Ruby-Homepage: http://www.ruby-lang.org.

I'll compare Ruby with Perl and Python, because I think they are the most frequently used and best known ones. Ruby has so much advantages against Perl and Python, that I'll try to mention here as much as possible.

At first I'll shortly explain what Ruby is:
Ruby is a modern, interpreted and object-orientated programming language. It has many similarities with Smalltalk ("everything is an object", simple inheritance, metaclass-model, code-blocks, garbage-collector, typeless variables, etc...), but takes much of the well formed syntax of Eiffel (or who don't know that great language, it's a little bit like Modula or Ada). Additionally many useful elements from Perl were added (e. g. regular expressions, text-processing, text-substitution, iterators, variables like $_ $/ ...). Therefore, Ruby is a very good alternative to Perl and Python. The difference between Perl and Ruby is the much easier and better to understand syntax and the easy-to-use *"real"* object-orientation. Following term which I have found on a Ruby-page expresses the power of Ruby:

Ruby > (Smalltalk + Perl) / 2

Some time ago I asked the author of Ruby, Yukihiro Matsumoto (aka matz), about the history of Ruby and why he developed a new language. Here is his original answer:

> *"Well, Ruby was born in Feb. 23 1993. At that day, I was talking with my colleague about the possibility of object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had smell of toy language (it still has). The object-oriented scripting language seemed very promising.*
>
> *I knew Python then. But I didn't like it, because I didn't think it was a true object-oriented language. OO features are appeared to be add-on to the language. I, as a language mania and OO fan for 15 years, really really wanted a genuine object-oriented, easy-to-use object-oriented scripting language. I looked for, but couldn't find one.*
>
> *So, I decided to make it. It took several months to make the interpreter run. I put it the features I love to have in my language, such as iterators, exception handling, garbage collection.*
>
> *Then, I reorganized the features in Perl into class library, and implemented them. I posted Ruby 0.95 to the Japanese domestic newsgroups in Dec. 1995.*
>
> *Since then, mail lists are established, web pages are formed. Highly active discussion was held in the mail lists. The oldest list ruby-list has 14789 messages until now.*
>
> *Ruby 1.0 was released in Dec. 1996, 1.1 in Aug. 1997, 1.2 (stable version) and 1.3 (development version) were released in Dec. 1998.*
>
> *Next stable version 1.4 will be shipped this months (June 1999), hopefully."*

As you can see, Ruby was developed, having Perl, Python, Smalltalk and Eiffel (as well as some other languages) in mind. So *matz* took the best from the above called languages to make a new, better, object-orientated scripting-language. Unlike Perl and Python Ruby was designed totally object-orientated right from the beginning. So there's no clumsy syntax for declaring a class like in Perl. That is why many people, myself included, say that Perl isn't really object-oriented. I agree with Stroustrup, the developer of C++, who once said that a special programming-style (e. g. OOP) is only sufficient supported if the language makes it easy to use this one. And I do not think that Perl supports sufficient enough the use of the object-oriented paradigm.

A big advantage of Ruby is, that it is very easy to learn, and so could perhaps become a language to introduce people into programming or object-orientation (maybe at school instead of the often used language Pascal). It took me only one day to get into Ruby and after some weeks I was nearly an expert! For learning Python it took me a little bit longer, but for learning Perl you normally need months and to become an expert even years. The syntax of Ruby is IMHO so easy that even non-Rubyists can read and understand most of the sourcecode, if it is written in clean Ruby and not Perl-like.

A good tutorial is very important when starting to learn a new language. Therefore big thanks to *matz* and the translators of *Ruby User's Guide*[1] , which is a short but almost everything covering introduction into Ruby. Also the *Ruby Language Reference Manual*[2] is very good. Looking at the tutorials of Python and Perl I have noticed following: either they are very long or they do not cover all aspects (especially Perl).

As already mentioned above, the syntax of Ruby is very easy, i. e. it is very clean, readable but also short. The code nearly documents itself, like in Eiffel and unlike Perl, which is the opposite. Clean, understandable and short code increases productivity, because it increases the speed of coding, reduces the need of documenting, is less error-prone and therefore it is easier to maintain. In Perl, very much time is wasted in finding errors or to document code. Even some errors are first detected after some time. In Ruby you have the possibility to make your code much more readable by inserting additionally keywords or by using words instead of operators. Now a small example of doing one and the same task in different styles in Ruby:

```
# short form
(1..10).each { |i|
   print "#{i}\n" if i % 2 == 0 && i > 5
}

# the same more readable
(1..10).each do |i|
   if i % 2 == 0 and i > 5
      print i, "\n"
   end
end

# the same more readable, with syntax-sugar for (1..10).each
```

[1] http://www.math.sci.hokudai.ac.jp/%7Egotoken/ruby/ruby-uguide

[2] http://hydrogen.ruby-lang.org/en/man-1.4

```
for i in 1..10 do
   if i % 2 == 0 and i > 5 then
      print i, "\n"
   end
end
```

In Perl you would write:

```
for (1..10) {
   if($_ % 2 == 0 && $_ > 5) {
      print "$_\n";
   }
}
```

And in Python:

```
for i in range(1,11):
   if(i % 2 == 0 and i > 5): print i,"\n"
```

The `..` operator in Ruby creates an `Range`-object. You can take every object which is comparable with the `<=>` operator and which have the `succ`-method. So you can also iterate over a string-range (e. g. `"a".."ab"`). In Python you need the function `range` to create an array over which you can then iterate. In Ruby you can choose between the keywords `and`, `or`, `not` and the operators `&&`, ||, `!`. The keyword `then` is optional and you can choose between {...} and `do` ... `end`.

Semicolons are only necessary if you want to write more than one statement into one line. This is like Python but unlike Perl, where you have to end every statement with a semicolon, which has only disadvantages, e. g. it result in more errors, because you often forget them, and make code less readable.

Another disadvantage of Perl is, that it differentiates between scalar (e. g. string, integer, reference) and non-scalar (e. g. array, hash) variables which makes programming much more difficult. A problem appears, when you want to create an array which itself contains arrays, because arrays or hashs in Perl can only contain scalar values. The solution is to put the references of the arrays into the array. But that's not easy and you can make many faults. Not so in Ruby and Python, where variables only contain references to objects.

An advantage of Ruby is, that all constructs have a value. For example an `if`-construct returns a value, so you can use `if` also on the right side of an expression. The following example shows this:

```
txt = "Hello World"

a = "size " +
    if txt.size > 3 then
       "greater"
    else
       "less"
    end +
    " than 3"

print a       # prints "size greater than 3"
```

Functions in Perl do not automatically introduce a new scope, so if you use a variable, which was already declared outside the function, it will be overwritten. You need 'my' or 'local' to declare local variables. Ruby let you easily create constants (begins with a capital letter), local variables (begins with a small letter), global variables (begins with a $) and instance variables (begins with a @).

But the most important aspect, why I am using Ruby instead of Python or Perl are the object-orientated features of Ruby, and that Ruby was designed object-oriented right from beginning, unlike Python and Perl where object-orientation was added on later. You can recognize this in e. g. in Python very good, because the first parameter (often named `self`) of every method of a class is the object on which the method is called:

```
class A:
   def method_a (self):
      # do what you want
```

The syntax of declaring a class in Ruby couldn't be easier:

```
class X
   ...
end
```

Now you see how a class in Perl is declared:

```
sub new {
   my ($class,@args) = @_;
   bless({@args}, $class);
}
```

You see, Ruby is much more intuitive. Now an example of declaring a `Point`-class in Ruby:

```
class Point

   # initialize is called implicit when Point.new is called
   def initialize (x, y)
      @x = x   # @x is an instance variable
      @y = y   # @y is an instance variable
   end

   # returns x (because instance variables are only
   #            visible inside the class)
   def x
      @x # the same as return @x
   end

   # the same like 'def x'
   def y; @y end

   # the setter-function
```

```
   def x= (x)
      @x = x
   end

   def y= (y)
      @y = y
   end

end
```

To make the code shorter and better to read, the same class using the `attr_accessor`-function of module `Module`, which dynamically creates a getter- and setter-method for each parameter is shown:

```
class Point
   attr_accessor :x, :y

   def initialize (x, y)
      @x = x
      @y = y
   end
end
```

There are also some other useful methods, like `attr_reader`, `attr_writer` as well as `attr`.

Ruby's instance variables are only accessible from inside the class, you cannot change this. This is *advanced* object-orientation. You'll see this in Eiffel as well as in Java but not in Python, where you can access the variables from outside the class. For example JavaBeans use method-names like *get_varname* and *set_varname* to access instance variables (where varname is the name of the variable). These are also called attributes. But Ruby has the ability to access the instance-variables through methods as if they were directly assigned. Here an example of using the `Point`-class:

```
a = Point.new (1,6)  # create object "a" of class Point

a.x = 5              # calls the method x= with the paramter 5
print a.x,"\n"       # prints 5
print a.y,"\n"       # prints 6
```

There's no direct access to instance variables. But sometimes you want to use a class the same way as in Python, where you can assign instance variables from outside without declaration. A short example in Python:

```
# empty class
class A:
   pass

x = A()              # create object "x" of class A

x.a = 3              # create new instance variable
print x.a            # prints 3
```

The same behavior can be reached in Ruby by using the class `OpenStruct` defined in file *ostruct.rb*. Now the same example like above in Ruby:

```
require 'ostruct'

x = OpenStruct.new   # create object "x" of class OpenStruct

x.a = 3
print x.a            # prints 3
```

There is almost no difference between the two example. But in Python you have to declare an own class! This behavior is very easy to impemented in Ruby, because Ruby calls the method `method_missing` for every unknow method (in this case this would be the method `a=`. So you can then dynamically create the method `a` which returns the value 3.

Now we'll extend the `Point`-class of an equality-operator. You do not need to insert the "=="-method into the above written `Point`-class, you can also extend the existing `Point`-class in adding a whole `Point`-class (not recommended in this case). The whole `Point`-class could now look like:

```
class Point
   attr_accessor :x, :y

   def initialize (x, y)
      @x = x
      @y = y
   end
end

class Point
   def == (aPoint)
      aPoint.x == x and aPoint.y == y
   end
end
```

or better like this:

```
class Point
   attr_accessor :x, :y

   def initialize (x, y)
      @x = x
      @y = y
   end

   def == (aPoint)
      aPoint.x == x and aPoint.y == y
   end

end
```

As you can see, Ruby is very easy and clean. But there are more features. In Ruby there are some conventions which you should not break. Method names which ends with:

- `=` should be used as *setter* of instance-variables
- `?` should return a boolean (e.g. `has_key?` of class `Hash`)
- `!` signalize that data inside the object is changed and not the values which is returned (e.g. `downcase!` which directly changes the objects value and `downcase` which returns the downcased value)

Now we'll extend the given `Array`-class which comes with Ruby for a missing method `count(val)`, which counts the occurence of *val*. We do not need to change any given sourcecode or inherit a given class:

```
class Array
   def count (val)
      count = 0

      # each iterates over every item and executes the block
      # between 'do' and 'end'.
      # the actual element of the iteration is stored into 'i'
      each do |i|
         if i == val then count += 1 end
      end
      count     # returns 'count'
   end
end

# now every declared Array has the method count(val)

print [1, 5, 3, 5, 5].count (5)       # prints 3
```

Sometimes it could happen, that you do not want to construct e.g. a `Point`-object via `Point.new` but via `Point.new_cartesian`. Ruby has not only classes but also meta-classes, like Smalltalk, i.e. the class-definition is available during runtime and could also be changed. In Ruby *every* class is an object constructed from the class `Class`. There are *instance methods* and *class methods*. *Instance methods* do not exist without objects. *Class methods* do exist without objects, they exist as far as a class is created. *Class methods* are called on classes (e.g. `Point.new`), *instance methods* on objects or instances (e.g. `"hallo".length`). Now we will create a *class methods* `new_cartesion` for the class `Point`:

```
class Point
   def new_cartesian (x, y)
      aPoint = new(x,y)
      # here you can do what you want
      return aPoint
   end
```

```
   # makes new_cartesian a class method
   module_function :new_cartesian
end

# now you can instatiate a Point-object with new_cartesian:
a = Point.new_cartesian(1, 43)
```

The same can also be done this way:

```
class Point
   def Point.new_cartesian (x, y)
      aPoint = new(x,y)
      # here you can do what you want
      return aPoint
   end
end
```

Now about iterators, they can be declared very easy:

```
# iterates n-times over the given block
def times (n)
   while n > 0 do
      yield n
      n -= 1
   end
end

times(5) {|i| print i," " }
```

Prints `5 4 3 2 1` onto the screen. Here's a more advanced example of using iterators:

```
include FileTest

FILE, DIRECTORY, DIRECTORY_UP = 0..2
PATH_SEP = "/"

#
# depth:  -1 = recurse all
# yield:  path, name, type
#
def scan_dir(path, depth=-1)
   # remove PATH_SEP at the end if present
   if path[-1].chr == PATH_SEP then path = path.chop end

   Dir.foreach (path) do |i|
      next if i =~ /^\.\.?$/
      if directory? (path+PATH_SEP+i) then
         yield path, i, DIRECTORY
         scan_dir (path+PATH_SEP+i, depth-1) do |a,b,c|
            yield a,b,c
         end unless depth==0
```

```
         yield path, i, DIRECTORY_UP
      elsif file? (path+PATH_SEP+i)
         yield path, i, FILE
      end
   end
end

# prints all files in directory /home and subdirectories
scan_dir ("/home",-1) { |path, name, type|
   print path, PATH_SEP, name, "\n" if type==FILE
}
```

The iterator `scan_dir` iterates over all files and subdirectories and calls the given block with path, filename and type as parameter.

Using blocks with methods, you can program very flexible, because you can extend the method from outside. Here an example of measuring the time to execute of a piece of code:

```
def Time.measure
   start = Time.times.utime
   yield
   Time.times.utime - start
end

# measures the time used by the loop between { and }
print Time.measure {
   for i in 1..100 do a = 10 end
}
```

Or counting the number of lines in a file:

```
def get_num_lines (file)
   # iterates over every line of "file"
   IO.foreach(file){}

   # $. returns the number of lines read since
   # last explicit call of "close"
   $.
end
print get_num_lines ("/home/michael/htdocs/index.html")
```

`IO.foreach(path)` is a short form for:

```
port = open(path)
begin
   port.each_line {
      ...
   }
ensure
   port.close
end
```

So in any case the file will be closed. You need not explicitly open the file or close it. Another important use of blocks is for synchronizing threads:

```
require "thread"

m = Mutex.new

a = Thread.start {
   while true do
      sleep 1
      m.synchronize do print "a\n" end
   end
}

b = Thread.start {
   while true do
      sleep 2
      m.synchronize do print "b\n" end
   end
}

sleep 10

a.exit        # kill thread
b.exit        # kill thread
```

The `Mutex`-object makes sure that only one thread can call simultaneous it's method `synchronize`. In Java e. g. to reach the same effect, the new keyword *synchronize* was introduced (in comparison to C++), but in Ruby this is done with a block-construct, so you're much more flexible, because you can extend or change the semantics.

Socket are also very easy to program. Following code will connect to a Whois-server and get information about a domain:

```
require 'socket'

def raw_whois (send_string, host)
   s = TCPsocket.open(host, 43)
   begin
      s.write(send_string+"\n")
      return s.readlines.to_s
   ensure
      s.close
   end
end

print raw_whois("page-store.de", "whois.ripe.net")
```

There is also a `TCPserver`-class in Ruby, which makes it much easier to build a server. Following a multi-threaded echo-server is shown:

```
require 'socket'
```

```
require 'thread'

server = TCPserver.open(5050)        # build server on port 5050

while true do                        # loop endless (until Ctrl-C)
   new_sock = server.accept          # wait for new connection
   print new_sock, " accepted\n"

   Thread.start do                   # new thread for connection
      sock = new_sock
      sock.each_line do |ln|         # read line from socket
         sock.print ln               # put line back to socket
      end
      sock.close                     # close connection
      print sock, " closed\n"
   end
end
```

The exception-model of Ruby is very close to the one of Eiffel, where you have pre- and post-conditions. In Ruby you have only post-conditions (ensure)! Now a presentation of the possibilities the exception-model of Ruby gives you:

```
begin
   # do anything...

   # raise an exception of type RuntimeError
   raise "Error occured"

   # raise an exception of user-defined class
   raise MyError.new(1,"Error-Text")

rescue
   # is called when an exception occures, you can
   # access the exception-object through $!, the error-message
   # is accessible through $!.message and the file and
   # line-number where the exception occured is stored in $@

   # solve problem...and retry the whole block
   retry

   # or re-raise exception
   raise

   # or raise new exception
   raise "error..."

ensure
   # is always called before the block
   # surrounded by "begin" and "end" is left
end
```

Regular Expressions are used like in Perl. To extract the top-level domain from a domain-name you can define following method:

```
def extract_tld (domain)
   domain =~ /\.([^\.]+)$/
   $1
end

print extract_tld ("www.coding-zone.de")     # prints "de"
```

Most of the features of Perl's regular expressions are also available in Ruby.

Database-access is also available in Ruby, but not all databases are yet supported. Currently only MySQL, Msql, PostgreSQL, Interbase and Oracle are available. A generalized database-access standard like ODBC, JDBC or DBD/DBI (Perl) would be very nice. Following code-example shows, how to print out a whole database-table with MySQL:

```
require 'mysql'

m = Mysql.new(host, user, passwd, db)    # creates new Mysql-object

res = m.query("select * from table")     # query the database

# gets all fieldnames of the query
fields = res.fetch_fields.filter {|f| f.name}

puts fields.join("\t")                   # prints out all fieldnames

# each row is printed
res.each do |row|                        # row is an array of the columns
  puts row.join("\t")
end
```

In Ruby, you can very easiliy implement dynamic argument-type checking. Following code implements this behavior:

```
class Object
   def must( *args )
      args.each do |c|
         if c === self then return self end
      end

      raise TypeError,
            "wrong arg type \"#{type}\" for required #{args.join('/')}"
   end
end


# this method requires an Integer as argument
def print_integer( i )
   i.must Integer
```

```
   print i
end

# you can also allow more than one type
# "name2s" returns an String, but takes a String or an Integer
def name2s( arg )
   arg.must String, Integer
   case arg
   when String  then arg
   when Integer then arg.id2name
   end
end

# :Hello is an Integer representing the string "Hello"
print name2s(:Hello)      # prints "Hello"
print name2s(" World")    # prints " World"
print name2s([1,2,3])     # raises TypeError
```

Writing C/C++ extension for Ruby is very easy. You can do all in C what is possible in Ruby. It is also possible to use the SWIG interface-generator for Ruby, but using the ruby-functions directly is very easy. Following a C-program which declares a module and one method:

```
// filename: str_func.c
#include "ruby.h"
#include <stdlib.h>   // for malloc

//
// Function will add "add" to each character of string "str"
// and return the result. Function do not change "str"!
// "obj" is assigned the class/module-instance on which method is called
//
extern "C" VALUE add_string( VALUE obj, VALUE str, VALUE add )
{
   int len, addval, i;
   char *p, *sptr;
   VALUE retval;

   // checks parameter-types
   // if type is wrong it raises an exception
   Check_Type(str, T_STRING);
   Check_Type(add, T_FIXNUM);

   // length of string "str"
   len = RSTRING(str)->len;

   // convert FixNum to C-integer
   addval = FIX2INT(add);

   // alloc temorarily memory for new string
```

```
   p = (char*) malloc(len+1);

   // raise an exception if not enough memory available
   if( !p )
      rb_raise(rb_eRuntimeError, "couldn't alloc enough memory for string");

   // get pointer to string-data
   sptr = RSTRING(str)->ptr;

   // iterate over each character, and add "addval" to it
   for(i=0; i<len; ++i) p[i] = sptr[i] + addval;
   p[i] = 0;

   // create Ruby-string which contains the C-string "p"
   retval = rb_str_new2(p);

   // free memory
   free(p);

   return retval;
}


//
// function is called, when module is loaded (e.g. through require 'str_func')
//
extern "C" EXTERN void Init_str_func(void)
{
   // declare new module
   VALUE module = rb_define_module(String_Functions");

   // declare module-function
   rb_define_module_function(
      module,                                        // module
      "add_string",                                  // name of method in Ruby
      (unsigned long (__cdecl *)(void)) add_string,  // pointer to C-function
      2                                              // number of arguments
   );
}
```

When compiled this as dynamic load library or static into the Ruby-interpreter, you can use the module in Ruby:

```
require 'str_func'

# should print "JgnnqYqtnf"
print String_Functions.add_string("HelloWorld", 2)
```

Inheritance is very easy:

```
class A
```

```
   def method_a
      print "A::method_a"
   end
end

# class B inherits from A
class B < A
   def method_b
      print "B::method_b"
   end

   # extend "method_a"
   def method_a
      super           # calls A::method_a
      print "B::method_a"
   end
end

a = A.new
b = B.new

a.method_a             # prints "A::method_a"
b.method_a             # prints "B::method_a"
b.method_b             # prints "A::method_a" and "B::method_a"

# now we'll dynamically change a method of object "b"
# this will not change the class B!
class << b
   def method_a
      print "B::method_a"
   end
end

b.method_a             # prints "B::method_a"
```

Multiple inheritance is supported through mix-in, where a class includes one or more modules. For example `Array` is a class which inherits from `Object` (like every class) and includes the module `Enumerable`. This works a bit like the *interfaces* in Java, but with the difference that an *interface* in Java cannot contain code. In Ruby every module can be included (with `include` modulename) into a class. This is like the programming-language *Sather* (very close to Eiffel) do it.

For those who came from Python to Ruby and want to use their programs written in Python inside Ruby-scripts, *Ruby/Python*[3] is the solution. Here is a simple example of using *Ruby/Python*:

```
require 'python'
require 'python/ftplib'
```

[3] http://www.goto.info.waseda.ac.jp/˜fukusima/ruby/python/doc/index.html

```
# create new object of Python-class "FTP"
ftp = Py::Ftplib::FTP.new('ftp.netlab.co.jp')
ftp.login
ftp.cwd('pub/lang/ruby')
ftp.dir
ftp.quit
```

This example shows how to use the Python-module `Ftplib` inside Ruby. You can do almost everything which is possible in Python with *Ruby/Python* in Ruby.