Object-oriented script language, Ruby

RED STAR

TJABO KLOPPENBURG

Ruby is an object-oriented script language, which has won itself a place among the established languages Perl, Python and maybe PHP, too. Ruby, a development from Japan, also has a certain entertainment value, which makes getting started that much easier.

From Japan, the Land of the Rising Sun, comes a new star in the sky of script languages. But does the world need a new script language? After all, there are already established ones such as Perl, Python, Tcl and many more. Ruby is a new attempt to produce a well-thought-out and object-oriented script language, which is easy to use.

Since it is very recent, there are not yet as many libraries available as there are for Perl or Python, and yet it is already usable for many purposes. This article introduces the concepts by means of a few examples and shows how compact and yet neatly structured Ruby can be.

Jack of all languages?

Ruby was developed between 1993 and 1995 by Yukihiro (Matz) Matsumoto in Japan; Perl 4 and Python already existed. While Python is a hybrid language with functions for procedural programming and with objects for OO-programming, Ruby is purely object-oriented: There are no functions, just methods — and everything is an object. There is a substitute, though, for functions: A method defined outside a class turns into a private method of the main class *Object*, as the result of which it becomes available globally.

The syntax and the design philosophy are heavily based on Perl, so there are statement modifiers (*if, unless, while* and others), integral regular expressions, special variables such as *$_* and the difference between *"..."* and *'...'* strings. Ruby thus helps itself to various programming languages and combines them into a new one.

Unlike Perl, *$,@* and *%* do not refer to different types of data, but to the scope of a variable: Normal variables can manage perfectly well without characters such as $, % or @, *$var* refers to a global variable of any type and *@var* an instance variable of any type in an object. There are no semicolons at the end of the characters in Ruby programs.

Object-oriented

As with all OO-languages, there are also practical standard classes available in Ruby, which are easy to use. Of course you can also define classes yourself, including inheritance and private,

protected and public methods. But the object orientation of Ruby goes a bit further, so to cut a long story short, everything that can be modified is an instance of a class (or else a reference thereto).

This also applies even to completely ordinary figures, so for example -42 is an instance of the integer class and thus also has access to the methods defined in the integer class. As the result of this object orientation, which extends as far as the basic elements, one obtains information about a number or a string, not through a function such as *sizeof(scalar)* or *int2str(int)*, but direct from the object, in fact through one of its methods.

For example, in order to obtain the string representation of 42, you can call up the method *42.to_s*. This is, of course, a banal example, but the (overwritable) method *.to_s* is already defined in the class *Object*, the mother of all classes, and thus available in absolutely every class — including its own.

As well as the usual class and instance methods, Ruby offers the option of using so-called iterators. These are methods which iterate via the individual elements of an object container, thus via the elements of an array, a hash, the lines in a text file or again, its own container class.

Even the integer class, itself not actually a container, has the use of helpful iterators such as *.upto(num)* or *.downto(num)*, which iterate starting from the object in steps of one to *num*. In this case the iterator is given a block of functions, which are called up in sequence for each element.

To show how this works, let's take a look at a small sample of code, where the *.upto* iterator of integers is put to use to create an output for the numbers from one to ten:

```
#!/usr/bin/ruby
Max = 10    # constant
1.upto(Max) do
 |i|      # iterated element
 print  "#{i} ^2 = ", i*i, "\n"
end
```

In the first line we define a constant *Max*, whose name begins with a capital letter, *.upto* is the iterator already mentioned, which in this case iterates from *1* to *Max*. *do* and *end* include the code block which is transferred to the iterator. Alternatively the block can also be bracketed with *{* and *}*. *|i|* assigns the element of the current iteration to the variable *i*, the *print* line finally outputs *i* and *i*i*.

Bear in mind that normal variables have to start with a lower-case letter and constants with a capital letter. If there is an objection on the tip of your tongue, that *for* is much more universal, I admit it, you're right. But since in nine out of ten cases an increment of one is being counted up or down, the *upto* method (and *downto*, *step* and others) is a genuine advantage.

As we are about to see from a somewhat more complex example, iterators also in many other cases allow for highly efficient programming. But first take a look at the line *Max = 10*. We need to visualise once more the idea that in this case the *10* is not simply a ten, but an instance of the integer class.

The assignment operator creates a new instance and then *Max* assigns a reference to this. Let's look at a somewhat more realistic problem: You want to rename all the files in the current directory ending in *.MP3*, so that they end in *.mp3* . Since the *mv* command from Linux is little use in this everyday problem (unless you speak fluent Bash), we shall build our own script in Ruby.

The task: One has a directory, wants to read out the content and, depending on the state of the entry, rename the file. While in Perl one has to tussle with functions such as *opendir* and *readdir*, the problem can be solved in Ruby exactly as we have formulated it: One takes a *Dir* object, allows the entries to be assigned and renames files as appropriate.

So now look at the following code:

```
Dir.open(".").entries.each do
 |entry|
 new = entry.gsub(/\.MP3$/, ".mp3");
 File.rename(entry,new) if new != entry
end
```

This is of course really easy to understand: *Dir.open* provides a new instance of the class *Dir*, which we do not assign to a variable, but re-use immediately. The *entries* method of the *Dir* instance provides an array with all directory entries in the form of string objects, and lastly *each* is the standard iterator over all elements of a container (here: arrays). All elements of the array land one after another in the variable *entry*. The string method *gsub* replaces *.MP3* with *.mp3* and the result lands in the variable *new*. The last line finally uses the class method *rename* of the class *file*, to undertake any renaming which may be necessary.

Class methods, unlike instance methods, can be called up without an existing instance of a class. The attached *if expr* is a construct familiar from Perl,

Listing 1: The Tribble class

```
class Tribble  # Start with a capital letter!
 def initialize  # constructor
  @conditions = Array.new  # possible conditions
  @condition  = nil       # default-condition
 end

 def setValues( arr )    # enter permitted values
  if (arr.type.to_s == "Array")
    arr.uniq!           # kill duplicated elements
   @condition = arr  if arr.size == 3
  end
 end
 protected :setValues

 def get       # method, read out condition
  @condition
 end

 def set( new ) # method, set condition

  @condition = new if@conditions include?(new)
 end

 def ==(other)  # comparison operator
   return nil if get.nil? or other.get.nil?
   @condition  == other.get
 end
end
```

which sometimes makes your hair stand on end. It would of course have been possible to use a normal three-line construct *if*, *Commands*, *end*, but the notation selected is often clearer especially with single-line *if* blocks.

Even if the example may have appeared rather odd at first, realisation is very much in line with how one thinks — and one becomes accustomed correspondingly rapidly to this type of programming. Another option worth mentioning here is that of passing small programs on the fly with the -e `...`parameter to the interpreter. But in that case the individual commands must be separated by semi-colons.

Your own classes

Now that we have seen how standard classes and iterators are used in two simple examples, we will now look at how you can define your own classes and later iterators in Ruby.

In order to also to show more specialised properties of classes in Ruby, we shall look at the implementation of a class whose instances can include three stable conditions: the Tribble class. Why none of the common languages has come up with this sort of data type is remarkable in itself. Such classes are not actually in any way absurd, as one can for example use a class and the conditions *should*, *must* and *either/or* really well to express the wishes of a customer with respect to an article — in order to then to seek out the most suitable thing for precisely this customer.

Or how would it be for example with a class that helps us by using the conditions *yes*, *no* and *ye-no* in making difficult decisions? We will next define a general class *Tribble*, from which we will then, for test purposes, derive the class *YNTribble*, which will help us to make decisions. Look at Listing 1 for a first simple class *Tribble*.

The name of the class must begin with a capital letter. A method is defined using *def*, *initialize* is the reserved name for the constructor. This is where we make the instance variable *@conditions* (options) and *@condition* (actual condition), which can be accessed by all methods within an instance.

The method *setValues* serves to define the possible conditions of the Tribble. This method is protected, so that it can only be used in *Tribble* and derived classes. *.get* serves to read out the current condition, *.set(value)* sets the current condition. In *.get* there is no *return*, as it can be left out, since the value of the last expression is automatically the return value of the method.

The third method finally defines the == operator for the Tribble class: If the condition of one of the Tribbles is *nil*, then the result is also *nil*, otherwise the conditions of the two Tribbles involved are compared and returned as the results *true* or *false*.

The *.nil?* method is defined in the class *Object* and only comes back with *true* if the object whose *nil?* method is being called is *nil*. In other words: A variable is never undefined, but at least an instance of the *nil* class.

To have one applicable class to play with, we shall derive from *Tribble* the new class *YNTribble*, which makes life easier for us with analytical functions. The new class is to have the functionality of the Tribble class and be able to include the conditions *yes*, *no* and *ye-no*.

In Ruby a class can in each case only inherit one other class. To make additional methods available in a class, modules can be integrated which are defined outside the class. In our new class (Listing 2) we define a new constructor, which calls up the constructor of the parent class and defines the possible conditions of instances of our class. It is possible here to specify a start condition as an option, the default is *yes*.

Also, we define a new operator +, which links two instances of *YNTribble* and depending on the conditions, comes up with a new *YNTribble* instance. The logic can be accommodated in three lines, by using statement modifiers: If two Tribbles to be compared, the result is *yes*, *yes* + *no* turns into *ye-no* and the rest becomes *no*.

This means there is now a useable Tribble class available for first tests. To do this, we instance the *YNTribble* twice and compare the contents of the objects:

```
t1 = YNTribble.new("yes")
t2 = YNTribble.new("no")
print "comparison", t1 == t2, "\n"

print "yes + no = "
t3 = t1 + t2  # yes + no = ?
puts t3.get
```

The output is:

```
Comparison: false
yes + no = ye-no
```

Obviously, since *yes* + *no* now produces a crystal-clear *ye-no*, what else. The instruction *puts* outputs the transferred string followed by \n.

Home-made iterators

Where, then, are the promised iterators? Well, we shall build ourselves a container for *YNTribbles*, *HTArray*, with an iterator *each*, which can iterate via all elements (Tribbles), without bothering the user with tedious details about the internal storage of the Tribbles. Also, both methods integrate *add(object)* and *get(num)* to roll in or read out Tribbles.

Listing 2: YNTribble — The defined Ye-no

```
class YNTribble < Tribble  # inherits all methods
 def initialize(condition="yes") # optional default value
  super()       # calls up the method of the parent class.
  setValues( ["yes", "no", "ye-no" ] ) # Array on the fly...
 @condition = condition  if@conditions.include?(condition)
 end

 def +(other)  # a self-defined operator: t2 = t1 + t2
  return YNTribble.new("yes")   if (@condition == "yes") and (other.get == "yes")
  return YNTribble.new("ye-no") if (@condition == "yes")  or (other.get == "yes")
  return YNTribble.new("no")   end
end
```

Listing 3 shows a simple implementation which would not only roll in Tribbles (arrays can store any objects) and which in this case could actually be replaced by a normal array. But using one's own class with iterators does have the advantage that later we could change the in-class realisation of the storage without any problem, while the interface remained the same externally.

Listing 3: An array made of Tribbles

```
class HTArray
 def initialize
  @arr = Array.new    # Data in array
 end

 def add( tribble )   # attach tribble
  @arr.push( tribble )
 end

 def get ( num )      # read out tribble
  @arr[num]
 end

 def each             # Iterator over all tribbles
  @arr.each {
    |tribble|
    yield tribble     # call up block with tribble
  }
 end
end
```

The definition of the iterator method is relatively simple: One writes a method, which takes all the stored objects by the hand in turn and passes them on to the *yield* command. When an iterator is used, then for each object the transferred code block stands in for the *yield*, where the parameters of the *yield* commands land in the variable specified in |...|.

Listing 3 shows one possible implementation of *HTArray* with an *.each* iterator. You can try out the HTArray with the following snippet of code. If you have saved the listings for *YNTribble* and *HTArray* in individual files, you must integrate the class definitions using *require*:

```
require `yntribble.rb'
require `htarray.rb'

ha = HTArray.new
ha.add( YNTribble.new("yes")   )
ha.add( YNTribble.new("ye-no") )
ha.add( YNTribble.new("no") )
print "Compare `yes' with all values:\n"
ha.each {
 |t|            # yield tribble, tribble -> t
 print ((t +
YNTribble.new("yes")).get,"\n")
}
```

Previously, we left off the brackets from *print*, but as soon as the expressions after *print* become complicated, you should include the brackets, so that the interpreter knows which part of the text after *print* is a parameter for the call. In case of doubt, ambiguous points in the source code can be displayed with the aid of *ruby -w*.

Controlled exceptions: Exceptions

When interpreting Ruby programs, it is possible for type conversions to fail (if, unexpectedly, a variable turns out to be *nil*) or a file to be read out is not available. In these exceptional situations the interpreter raises up so-called exceptions, which are also familiar from Java and C++.

If one does not catch an exception oneself, the program automatically shuts down. But this is not necessarily in one's own best interests, as this can result, for example, in painstakingly calculated figures or tedious user inputs being lost. Therefore, exceptions can be caught in the program, so as to react appropriately. The following small example reacts, when storing a result, to the problem that the file is not writeable from time to time. This effect sometimes occurs if one starts a program from the home directory of another user, who has hard-wired the name of the destination file.

Exceptions are caught using *begin /rescue/end*:

```
resfile = "/home/root/.result"
begin
 file = File.open(resfile, "w")
 # save data
rescue
 puts "error!"
 print "Specify a writeable "
 print "file for the results: "
 resfile = STDIN.gets
 retry  # agaaaain!
ensure
 file.close  # close file
end
```

If *resfile* is not writeable, *File.open(...,"w")* raises a corresponding exception. *rescue* catches it — and we try to get round the problem: The user is challenged to enter a new file name, which will be read in via *STDIN.gets*. The effect of *retry* is that the critical block will run through one more time. In this example this could go round in circles forever, until the user finally specifies the name of a writeable file. The code after *ensure* will in any case be executed. If one uses *ensure* and *rescue*, *rescue* must appear first.

Catching exceptions is one thing — but of course, one can also raise exceptions oneself. This is done simply via the command *raise*, to which a descriptive string is given. This can then be read out via *$!* in the exception treatment:

```
begin
 raise "unexpected error"
rescue
 print "Exception!: ", $!, "\n"
 print "(",$!.type,")\n"
end
```

Exceptions are instances of exception classes, which are all derived from *Exception*. These in turn are children of the parent of all objects (*Object*) and thus can use the method *.type*, with which the

name of the class of an instance can be read out. So we can find out the type of exception in the exception treatment:

```
begin
a = 1 / 0
rescue
 print $!.type, " : ", $!, "\n"
end
```

In the first example it still looked as if *$!* was simply a string. But it is obviously a class instance and *print* has somehow called up an appropriate method from *$!*, in order to display the string. We should now no longer be surprised that this is the standard method *.to_s*.

Graphical Allsorts

In case the properties described thus far have given the impression that Ruby is purely a console language — this is not so. Ruby has integral support for Tk, with which an expert programmer can at least quickly knit himself some simple GUIs.

Unfortunately there is no proper graphical snap-on GUI tool yet, although there is already an interface to Glade, the Gtk GUI builder. And, of course, a direct interface to Gtk+, which can be found at *www.ruby-lang.org/gtk/en* complete with documentation.

For programming Tk you should in any case take a look in the obligatory reading for Ruby, for example the downloadable online version of the just-published book "Programming Ruby" .

All about Ruby online

The fairly-expensive printed book is available in English. It is very up-to-date, which sadly cannot necessarily always be said about the official reference documentation. Ruby is after all a Japanese development and any current documentation usually comes out first in Japanese, and then with a bit of luck is translated into English.

Even the XML sources of the book are available. The XML code is hideous, though, as confirmed by one of the authors, who extracted the XML code from TeX code. Nevertheless it is possible to patch together very passable brief references from the XML files or adapt the layout of the book to one's own preferences. Unfortunately the appropriate XSLT code for the conversion does not come with the book — this is something you will have to think out for yourself.

For example, you could use the class *xmlparser* or one of the other XML expansions found in the Ruby Application Archive or at Ruby Mine. When doing so it is often advisable not to follow the download link immediately, but to look for a more recent version on the homepage of the respective expansion, since every author updates his homepage first.

Info

Which language is better?: http://www.perl.com/pub/2000/12/advocacy.html
Ruby homepage: http://www.ruby-lang.org/en/
Thomas, Hunt, Matsumoto: programming Ruby; Addison-Wesley; ISBN 0201710897
Online version of the book: http://www.pragmaticprogrammer.com/ruby/
ruby-gtk: http://www.ruby-lang.org/gtk/en/
Ruby Application Archive: http://www.ruby-lang.org/en/raa.html
Ruby Mine: http://www.ale.cx/mine/raa.html
Ruby-FAQ: http://www.ruby-lang.org/
Open Directory Project:
http://www.dmoz.org/Computers/programing/Languages/Ruby

For the rest — it's written in the book

This brings us to the end of the article. One or two things have not yet been addressed — for example threads - since this article is not intended to be a substitute for a book. Information about threads and about the testing of Ruby can of course be found in the online book mentioned and maybe also in diverse other online documents, which any good search engine should be able to find on the Net.

An up-to-date reference can be obtained via the tool *ri*, which can be had from the Application Archive. The call *ri File* for example spits out all methods of the file class. The console single-liner *ruby -e `puts file .methods.join("\n")* does just that too, but without any further explanations. The English Ruby mailing list is also good for additional information (info on the Ruby homepage). Anyone who reads this will gain an overview on the latest developments and can even take part in further development, as "Matz" Matsumoto makes enquiries in the ML before making any changes.

Finally, flat-rate and other hard-core surfers can also set a course for the channel *#ruby-lang* in the IRC via the server *irc.openprojects.net*, in which there are always a few Ruby enthusiasts, including the authors of the online book.

Anyone who wants to go further with Ruby should in any case read up on the difference between code blocks of iterators and the code within *while, until* and *if* — and which of the variables can be changed locally and which cannot.

Have fun. ■

The author

Tjabo Kloppenburg is, strictly speaking, studying electrical and electronic engineering at the Uni of Siegen, although he has tended to specialise in IT. Is there anything nicer than a bit of DIY with IT scripts? He thinks not.