Uni Programming Languages Notes

Felicitas Pojtinger (fp036)

2022-10-24

Contents

1 Introduction

1.1 Contributing

These study materials are heavily based on professor Ihler's "Aktuelle Programmiersprachen" lecture at HdM Stuttgart.

Found an error or have a suggestion? Please open an issue on GitHub (github.com/pojntfx/uni-programminglanguages-notes):

Figure 1: QR code to source repository

If you like the study materials, a GitHub star is always appreciated :)

1.2 License

Figure 2: AGPL-3.0 license badge

2 Overview

2.1 General Design

- “A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.”
- Inspired by Perl, Smalltalk, Eiffel, Ada, Lips
- Multi-paradigm from the beginning: Functional, imperative and object-oriented
- Radical object orientation: Everything is an object, there are no primitive types like in Java (`5.times { print "We *love* Ruby -- it's outrageous!"}`)
- Very flexible, i.e. operators can be redefined
- Built-in blocks (closures) from the start, excellent mapreduce capabilities
- Prefers mixins over inheritance
- Syntax uses limited punctuation with some notable exceptions (instance variables with `@`, globals with `$` etc.)

2.2 Implementation Details

- Exception handling similar to Java & Python, but no checked exceptions
- Garbage collection without reference counts
- Simple C/C++ extension interface
- OS independent threading & Fibers, even if OS is single-threaded (like MS-DOS)
- Cross-platform: Linux, macOS, Windows, FreeBSD etc.
- Many implementation (MRI/CRuby, JRuby for Ruby in the JVM, TruffleRuby on GraalVM, mruby for embedded uses, Artichoke for WebAssembly and Rust)

2.3 Users

- Twitter
- Mastodon
- GitHub
- Airbnb
- Shopify
- Twitch
- Stripe
- Etsy
- Soundcloud

- Basecamp
- Kickstarter

2.4 Timeline

- First concepts and prototypes ~1993
- First release ~1995, became most popular language in Japan by 2000
- Subsequent evolution and growth outside Japan
- Ruby 3.0 released ~2020, introducing a type system for static analysis, fibers (similar to Goroutines, asyncio etc.), and completing optimizations making it ~3x faster than Ruby 2.0 (from 2013)

3 Syntax

3.1 Logic

Typical logical operators:

```
>> 2 < 3
=> true
```

```
>> 1 == 2
=> false
```

Comparisons are type checked:

```
>> 1 == "1"
=> false
```

Trip equals can be used to check if if an instance belongs to a class:

```
>> String === "abc"
=> true
```

If, else, etc work as expected:

```
if name == "Zigor"
  puts "#{name} is intelligent"
end
```

However Ruby also allows interesting variations of this, such as putting the comparions behind the block to execute:

```ruby
puts "#{name} is genius" if name == "Zigor"
```

We can also use `unless`, which is a more natural way to check for negated expressions:

```ruby
p "You are a minor" unless age >= 18
```

`switch` statements are known as **`case`** statements, but don't `fallthrough` by default like in Java:

```ruby
case a
  when 1
    spell = "one"
  when 2
    spell = "two"
  when 3
    spell = "three"
  when 4
    spell = "four"
  when 5
    spell = "five"
  else
    spell = nil
end
```

Since everything is an object, we can also use **`case`** statements to check if instances are of a class:

```ruby
a = "Zigor"
case a
when String
  puts "Its a string"
when Fixnum
  puts "Its a number"
end
```

As mentioned before, Ruby is a very flexible language. The case statement for example also allows to us to check regular expressions:

```ruby
case string
when /Ruby/
  puts "string contains Ruby"
else
  puts "string does not contain Ruby"
end
```

We can even use Lambdas in case statements, making long **`if ... else`** blocks unnecessary:

```ruby
case num
when -> (n) { n % 2 == 0 }
  puts "#{num} is even"
else
  puts "#{num} is odd"
```

```
6 end
```

And the object orientation becomes very clear; we can even define our own matcher classes:

```
1  class Zigor
2    def self.===(string)
3      string.downcase == "zigor"
4    end
5  end
6
7  name = "Zigor"
8
9  case name
10 when Zigor
11   puts "Nice to meet you Zigor!!!"
12 else
13   puts "Who are you?"
14 end
```

We can also assign values from a case statement:

```
1 grade = case mark
2         when 80..100  : 'A'
3         when 60..79   : 'B'
4         when 40..59   : 'C'
5         when 0..39    : 'D'
6         else "Unable to determine grade. Try again."
7 end
```

3.2 Loops

Ruby does not have traditional for loops, but multiple, more general constructs that allow for the usecases.

Ruby has the `for` loop that we are all used to, but also more specialized constructs that allow for more expressive usecases:

```
1 for i in 0..10
2     p i
3 end
```

For example `upto` and `downto` methods:

```
1 10.downto 1 do |num|
2   p num
3 end
```

```
1 17.upto 23 do |i|
```

```
  print "#{i}, "
end
```

Or the `times` method, which is much more readable:

```
7.times do
  puts "I know something"
end
```

`while`, `until` and the infinite `loop` loops still exist however:

```
i=1
while i<=10 do
  print "#{i}, "
  i+=1
end
```

```
i=1
until i>10 do
  print "#{i}, "
  i+=1
end
```

```
loop do
  puts "I Love Ruby"
end
```

We can also use **`break`**, `next` and `redo` within a loop's block:

```
1.upto 10 do |i|
  break if i == 6
  print "#{i}, "
end
```

```
10.times do |num|
  next if num == 6
  puts num
end
```

```
5.times do |num|
  puts "num = #{num}"
  puts "Do you want to redo? (y/n): "
  option = gets.chop
  redo if option == 'y'
end
```

3.3 Arrays

Arrays in Ruby can contain multiple types and work as expected; there is no array vs collection divide:

```
my_array = ["Something", 123, Time.now]
```

Instead of loops you can use the `each` method to iterate:

```
my_array.each do |element|
  puts element
end
```

We can use << to add things to an array:

```
>> countries << "India"
=> ["India"]
>> countries
=> ["India"]
>> countries.size
=> 1
>> countries.count
=> 1
```

And access elements with `[0]`:

```
>> countries[0]
=> "India"
```

Thanks to the `..` syntax we can also access multiple elements at once in a very simple way:

```
>> countries[4..9]
=> ["China", "Niger", "Uganda", "Ireland"]
```

And use the `includes?` method (note the `?!`) to check if elements are present:

```
>> countries.include? "Somalia"
=> true
```

And `delete` to delete elements:

```
>> countries.delete "USA"
=> "USA"
```

If we have a nested array, using `dig` fill allow us to find deeply nested elements in a simple way:

```
>> array = [1, 5, [7, 9, 11, ["Treasure"], "Sigma"]]
=> [1, 5, [7, 9, 11, ["Treasure"], "Sigma"]]
>> array.dig(2, 3, 0)
=> "Treasure"
```

Another very useful set of features are set operations, allowing us to modify arrays in a simple way, for example we can use the & operator to find elements that are in two arrays:

```
>> volleyball = ["Ashok", "Chavan", "Karthik", "Jesus", "Budha"]
=> ["Ashok", "Chavan", "Karthik", "Jesus", "Budha"]
>> cricket = ["Budha", "Karthik", "Ragu", "Ram"]
=> ["Budha", "Karthik", "Ragu", "Ram"]
>> volleyball & cricket
=> ["Karthik", "Budha"]
```

Or + to merge them:

```
>> volleyball + cricket
=> ["Ashok", "Chavan", "Karthik", "Jesus", "Budha", "Budha", "Karthik",
    "Ragu", "Ram"]
```

Or use | to merge both, but de-duplicating at the same time:

```
>> volleyball | cricket
=> ["Ashok", "Chavan", "Karthik", "Jesus", "Budha", "Ragu", "Ram"]
```

Finally, we can also use – to remove multiple elements at once:

```
>> volleyball - cricket
=> ["Ashok", "Chavan", "Jesus"]
```

For those who are familiar with MapReduce, Ruby provides all of it in the language. For example `map`:

```
>> array = [1, 2, 3]
=> [1, 2, 3]
>> array.map{ |element| element * element }
=> [1, 4, 9]
```

Note that this doesn't modify the array; we can use `map!` for that, which works for lots of Ruby methods:

```
>> array.collect!{ |element| element * element }
=> [1, 4, 9]
>> array
=> [1, 4, 9]
```

The `filter` method for example can be used in the same way (named `keep_if`, with the opposite `delete_if` also existing), and works like how you already know if from JS:

```
>> array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> array.keep_if{ |element| element % 2 == 0}
=> [2, 4, 6, 8, 10]
```

3.4 Hashes

Hashes can be used to store mapped information:

```
mark = {}
mark['English'] = 50
mark['Math'] = 70
mark['Science'] = 75
```

And we can define a default value:

```
mark = {}
mark.default = 0
mark['English'] = 50
mark['Math'] = 70
mark['Science'] = 75
```

The hash literal `{}` also allows us to create hashes with pre-filled information:

```
marks = { 'English' => 50, 'Math' => 70, 'Science' => 75 }
```

To loop over hashes, we can use the `each` method again:

```
total = 0
mark.each { |key,value|
  total += value
}
puts "Total marks = "+total.to_s
```

A very interesting feature to use in combination with hashes are symbols; they are much more efficient than strings as they are global and thus use less memory:

```
mark = {}
mark[:English] = 50
mark[:Math] = 70
mark[:Science] = 75
```

We can check this by getting their `object_id` (a kind of pointer):

```
c = "able was i ere i saw elba"
d = "able was i ere i saw elba"
>> c.object_id
=> 21472860
>> .object_id
=> 1441620
```

```
e = :some_symbol
f = :some_symbol
>> e.object_id
=> 1097628
```

```
>> f.object_id
=> 1097628
```

Just like accessing hash values is similar for arrays and hashes, we can use the same MapReduce functions on hashes:

```
>> hash = {a: 1, b: 2, c: 3}
=> {:a=>1, :b=>2, :c=>3}
>> hash.transform_values{ |value| value * value }
=> {:a=>1, :b=>4, :c=>9}
```

3.5 Ranges

Ranges are a cool concept in Ruby that we've used before. We can use them with the `..` notation:

```
>> (1..5).each {|a| print "#{a}, " }
=> 1, 2, 3, 4, 5, => 1..5
```

We can also use them on strings:

```
>> ("bad".."bag").each {|a| print "#{a}, " }
=> bad, bae, baf, bag, => "bad".."bag"
```

They can be very useful in **case** statements, where you can replace lots of `or` operators with them:

```
grade = case mark
  when 80..100
    'A'
  when 60..79
    'B'
  when 40..59
    'C'
  when 0..39
    'D'
  else
    "Unable to determine grade. Try again."
end
```

In addition to using them in **case** statements as described before, they can also serve as conditions:

```
print "Enter any letter: "
letter = gets.chop

puts "You have entered a lower case letter" if  ('a'..'z') === letter
puts "You have entered a upper case letter" if  ('A'..'Z') === letter
```

We can also use triple dots, which will remove the last value:

```
>> (1..5).to_a
=> [1, 2, 3, 4, 5]
>> (1...5).to_a
=> [1, 2, 3, 4]
```

It is also possible to define endless ranges:

```
print "Enter your age: "
age = gets.to_i

case age
when 0..18
  puts "You are a kid"
when (19..)
  puts "You are grownup"
end
```

3.6 Functions

As mentioned before, Ruby draws a lot of inspiration from functional programming languages, and functions are a primary building block in the language as a result.

We can define functions with `def` and call them without parentheses:

```
def print_line
  puts '_' * 20
end

print_line
```

It is also possible to define default arguments unlike in Java:

```
def print_line length = 20
  puts '_'*length
end

print_line
print_line 40
```

Arguments are always passed by reference:

```
def array_changer array
  array << 6
end

some_array = [1, 2, 3, 4, 5]
p some_array
array_changer some_array
```

```
p some_array

=> [1, 2, 3, 4, 5]
=> [1, 2, 3, 4, 5, 6]
```

There is no need for a **return** statements as returns are implicit (but optional for control flow support):

```
def addition x, y
  x + y
end

addition 3, 5

=> 8
```

We can also define named arguments, with or without defaults:

```
def say_hello name: "Martin", age: 33
  puts "Hello #{name} your age is #{age}"
end

say_hello name: "Joseph", age: 7
```

Arguments can also be variadic:

```
def some_function a, *others
  puts a
  others.each do |x|
    puts x
  end
end

some_function 1,2,3,4,5
```

A very neat function is to use argument forwarding to call a function with all used parameters:

```
def print_something string
  puts string
end

def decorate(...)
  puts "#" * 50
  print_something(...)
  puts "#" * 50
end

decorate "Hello World!"
```

We can also define a function in more consise way:

```ruby
def double(num) = num * 2
```

3.7 Classes

Besides the functional influence, Ruby is also a radically object-oriented language. As a result, it makes working with objects and classes very easy:

```ruby
class Square
end
```

Through the `attr_reader`, `attr_writer` and `attr_accessor` notation we can add instance variables to a class:

```ruby
class Square
  attr_accessor :side_length
end
```

They can be read and written with `.`:

```ruby
s1 = Square.new # creates a new square
s1.side_length = 5 # sets its side length
puts "Side length of s1 = #{s1.side_length}" # prints the side length
```

Methods can be defined with `def`:

```ruby
class Square
  attr_accessor :side_length

  def area
    @side_length * @side_length
  end

  def perimeter
    4 * @side_length
  end
end
```

Note the use of `@` to access instance variables.

Like many object-oriented languages, Ruby supports constructors (called initializers):

```ruby
class Square
  attr_accessor :side_length

  def initialize side_length = 0
    @side_length = side_length
  end
```

```
7
8     def area
9       @side_length * @side_length
10    end
11
12    def perimeter
13      4 * @side_length
14    end
15  end
```

Variables defined by `attr_accessor` as public; we can make them private by ommiting their definition:

```
1  class Human
2    def set_name name
3      @name = name
4    end
5
6    def get_name
7      @name
8    end
9  end
```

In a similar way, we can use **`private`** and **`protected`** to change the visibility of methods:

```
1  class Human
2    attr_accessor :name, :age
3
4    def tell_about_you
5      puts "Hello I am #{@name}. I am #{@age} years old"
6    end
7
8    private def tell_a_secret
9      puts "I am not a human, I am a computer program. He! Hee!!"
10   end
11 end
```

In addition to instance variables, we can also create class variables which work similar to static variables in Java using the `@@` notation:

```
1  class Robot
2    def initialize
3      if defined?(@@robot_count)
4        @@robot_count += 1
5      else
6        @@robot_count = 1
7      end
8    end
9
10   def self.robots_created
```

```
    @@robot_count
  end
end
```

Similarly so, we can define class constants like so:

```
class Something
  Const = 25

  def Const
    Const
  end
end

puts Something::Const
```

While inheritance is not the primary means of reusing code in Ruby, there is support for it in the language using the < notation:

```
class Rectangle
  attr_accessor :length, :width
end

class Square < Rectangle
  def initialize length
    @width = @length = length
  end

  def side_length
    @width
  end
end
```

We can overwrite methods; interestingly it is possible to change a child's signature and use the **super** method in the child:

```
class Square < Rectangle
  def set_dimension side_length
    super side_length, side_length
  end
end
```

I won't go into more details on these aspects as they are mostly similar to Java; the same goes for Threads, Exception and more. One thing uniquely powerful in Ruby is reflection; for example, you can get the methods of a class as an array using `.methods`:

```
>> "a".methods
=>
[:unicode_normalized?,
```

```
  :encode!,
  :unicode_normalize,
  :ascii_only?,
  :unicode_normalize!,
  :to_r,
  :encode,
  :to_c,
  :include?,
  :%,
  :*,
  :+,
  :unpack,
  # ...
]
```

We can also get private methods using `.private_methods`, instance variables using `.instance_variables` etc.

Another feature fairly unique to Ruby is method aliasing:

```
class Something
  def make_noise
    puts "AAAAAAAAAAAAAAHHHHHHHHHHHHHH"
  end

  alias :shout :make_noise
end

Something.new.shout
```

This makes it very easy to define multiple method names for things that are frequently interchanged, such as `.delete` and `.remove`, or `.filter` and `.keep_if`.

Due to Ruby's dynamic nature, we can also define classes dynamically and anonymously:

```
person = Class.new do
  def say_hi
    'Hi'
  end
end.new
```

To deal with the complexities of such a dynamic language, Ruby has support for a safe navigation operator similar to Typescript:

```
class Robot
  attr_accessor :name
end

robot = Robot.new
robot.name = "Zigor"
```

```ruby
puts "The robots name is #{robot.name}" if robot&.name
```

3.8 Files, Modules and Mixins

We can use the `require` function to import things from files; this is very similar to how early NodeJS works:

```ruby
# break_square.rb

class Square
  attr_accessor :side_length

  def perimeter
    @side_length * 4
  end
end
```

```ruby
# break_main.rb

require "./break_square.rb"

s = Square.new
s.side_length = 5
puts "The squares perimeter is #{s.perimeter}"
```

However this quickly leads to problems with code organization, for example when two functions with a different purpose are named the same way. Ruby solves this issue with modules:

```ruby
module Star
  def line
    puts '*' * 20
  end
end

module Dollar
  def line
    puts '$' * 20
  end
end
```

If we `include Star` and call `line`, we will print a line of starts, and if we do so with `Dollar`, calling `line` again will print dollar signs. Without including `line`, the method will be undefined.

We can also call methods and access other objects in a module using the `::` operator:

```ruby
>> Dollar::line
=> $$$$$$$$$$$$$$$$$$$$
```

The `include` keyword can be used to form Mixins, which will expose reusable code only to a specific class, i.e. make the `Pi` constant only accessible from a single class:

```ruby
class Sphere
  include Constants
  attr_accessor :radius

  def volume
    (4.0/3) * Pi * radius ** 3
  end
end
```

3.9 Metaprogramming

Ruby is a very flexible langauge, and as such it allows metaprogramming. For example, directly call a method using the `send` function by passing in the `speak` symbol:

```ruby
class Person
  attr_accessor :name

  def speak
    "Hello I am #{@name}"
  end
end


p = Person.new
p.name = "Karthik"
puts p.send(:speak)
```

This allows for very powerful, but dangerous things, such as calling arbitrary functions by passing in the method name as a string:

```ruby
class Student
  attr_accessor :name, :math, :science, :other
end

s = Student.new
s.name = "Zigor"
s.math = 100
s.science = 100
s.other = 0
```

If we want to give a user access to any of the properties using `send`, we can get their input using `gets.chop`:

```ruby
print "Enter the subject who's mark you want to know: "
subject = gets.chop
```

```
puts "The mark in #{subject} is #{s.send(subject)}"
```

We can also catch a developer calling methods that don't exist at runtime and handle that usecase explicitly by implementing a `method_missing` method:

```
class Something
  def initialize
    @name = "Jake"
  end

  def method_missing method, *args, &block
    puts "Method: #{method} with args: #{args} does not exist"
    block.call @name
  end
end

s = Something.new
s.call_method "boo", 5 do |x|
    puts x
end
```

As you can see, we're now able to call a method that doesn't exist, and provide the implementation ourselves:

```
=> Method: call_method with args: ["boo", 5] does not exist
=> Jake
```

Instead of passing in an implementation in the form of a block ourselves, we can also do other things, such as matching the incoming method name against a regular expression and then manually calling the method:

```
class Person
  attr_accessor :name, :age

  def initialize name, age
    @name, @age = name, age
  end

  def method_missing method_name
    method_name.to_s.match(/get_(\w+)/)
    send($1)
  end
end

person = Person.new "Zigor", "67893"
puts "#{person.get_name} is #{person.get_age} years old"

=> Zigor is 67893 years old
```

It is also possible to use `define_method` to dynamically define a method at runtime:

```
class Person
  def initialize name, age
    @name, @age = name, age
  end
end

Person.define_method(:get_name) do
  @name
end

person = Person.new "Zigor", "67893"

>> person.get_name
=> "Zigor"
```

We can also define class methods etc. using `define_singleton_method` or `class_eval` and `instance_eval` etc. to add arbitrary things such ass `attr_accessor`s to classes or even instances.

4 Practical Examples

4.1 dRuby

4.2 Sinatra

5 Questions