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
```

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|
2   print "#{i}, "
3 end
```

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

```
1 7.times do
2   puts "I know something"
3 end
```

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

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

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

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

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

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

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

```ruby
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:

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

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

```ruby
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]
```