Uni Programming Languages Notes

Felicitas Pojtinger (fp036)

2022-10-24

Introduction

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. :)

License

Figure 2: AGPL-3.0 license badge

Overview

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.)

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)

Users

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

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)

Syntax

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:

Loops

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

```
for i in 0..10
    p i
end
```

For example upto and downto methods:

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

```
17.upto 23 do |i|
  print "#{i}, "
end
```


Or the times method, which is much more readable:

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
```

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:

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
```

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
```

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:

```
class Square
end
```

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

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

They can be read and written with .:

```
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 si
```

Files, Modules and Mixins

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

```
# break_square.rb

class Square
  attr_accessor :side_length

  def perimeter
    @side_length * 4
  end
end

# break_main.rb

require "./break_square.rb"
```

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:

```
class Person
  attr_accessor :name

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


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

Usecases for Ruby

Usecases for Ruby

Recommended:

- Scripting
- Web Development, especially old Web 2.0-style
- MVPs in startups (see Twitter etc.)
- Applications that require excellent extensibility (see Discourse etc.)
- Applications working with highly dynamic data models
- Systems administration on UNIX (i.e. Metasploit, Chef, Puppet, Homebrew)
- “Glue code” between cloud systems (i.e. Fluentd)
- Business Intelligence apps/CRUD systems (esp. with Ruby on Rails)

Not Recommended:

- Latency-dependend/real-time applications (garbage collection)
- High throughput systems (i.e. high-RPS web services)
- Memory- or CPU-constrained systems

Practical Examples

dRuby

While not recommended in modern applications (see professor Kriha's "Distributed Systems" course), dRuby is an excellent example of an idiomatic Ruby way of creating servers and clients, specifically distributed objects. We can define a server like so:

```
require 'drb/drb'

URI = 'druby://localhost:8787'

class PersonServer
  attr_accessor :name

  def initialize(name)
    @name = name
  end
```

Sinatra

Aside from Ruby on Rails, Sinatra is a very neat web framework. You can define a web server in just three lines of code:

```
require 'sinatra'

get '/' do
  'Hello, world!'
end
```

Handling POST requests and parsing data is also very simple:

```
before do
  next unless request.post?

  request.body.rewind
  @request_payload = JSON.parse request.body.read
end
```

Questions