

---

# **Uni Programming Languages Notes**

Felicitas Pojtinger (fp036)

2022-10-24

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Contributing . . . . .	3
1.2	License . . . . .	3
<b>2</b>	<b>Overview</b>	<b>4</b>
2.1	General Design . . . . .	4
2.2	Implementation Details . . . . .	4
2.3	Users . . . . .	4
2.4	Timeline . . . . .	5
<b>3</b>	<b>Syntax</b>	<b>5</b>
3.1	Logic . . . . .	5
3.2	Loops . . . . .	7
3.3	Arrays . . . . .	9
3.4	Hashes . . . . .	11
3.5	Ranges . . . . .	12
3.6	Functions . . . . .	13
3.7	Classes . . . . .	15
3.8	Files, Modules and Mixins . . . . .	19

# 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](https://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

Uni Programming Languages Notes (c) 2021 Felicitas Pojtinger and contributors

SPDX-License-Identifier: AGPL-3.0

## 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, Lisp
- 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:

```
1 >> 2 < 3
2 => true
```

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

Comparisons are type checked:

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

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

```
1 >> String === "abc"
2 => true
```

If, else, etc work as expected:

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

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

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

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

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

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

```
1 case a
2   when 1
3     spell = "one"
4   when 2
5     spell = "two"
6   when 3
7     spell = "three"
8   when 4
9     spell = "four"
10  when 5
11    spell = "five"
12  else
13    spell = nil
14 end
```

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

```
1 a = "Zigor"
2 case a
3   when String
4     puts "Its a string"
5   when Fixnum
6     puts "Its a number"
7 end
```

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

```
1 case string
2   when /Ruby/
3     puts "string contains Ruby"
4   else
5     puts "string does not contain Ruby"
6 end
```

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

```
1 case num
2   when -> (n) { n % 2 == 0 }
3     puts "#{num} is even"
4   else
5     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|
```

```
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:

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

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

```
1  loop do  
2    puts "I Love Ruby"  
3  end
```

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

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

```
1  10.times do |num|  
2    next if num == 6  
3    puts num  
4  end
```

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



### 3.3 Arrays

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

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

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

```
1 my_array.each do |element|
2   puts element
3 end
```

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

```
1 >> countries << "India"
2 => ["India"]
3 >> countries
4 => ["India"]
5 >> countries.size
6 => 1
7 >> countries.count
8 => 1
```

And access elements with `[0]`:

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

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

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

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

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

And `delete` to delete elements:

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

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

```
1 >> array = [1, 5, [7, 9, 11, ["Treasure"], "Sigma"]]
2 => [1, 5, [7, 9, 11, ["Treasure"], "Sigma"]]
3 >> array.dig(2, 3, 0)
4 => "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:

```
1 >> volleyball = ["Ashok", "Chavan", "Karthik", "Jesus", "Budha"]
2 => ["Ashok", "Chavan", "Karthik", "Jesus", "Budha"]
3 >> cricket = ["Budha", "Karthik", "Ragu", "Ram"]
4 => ["Budha", "Karthik", "Ragu", "Ram"]
5 >> volleyball & cricket
6 => ["Karthik", "Budha"]
```

Or `+` to merge them:

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

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

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

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

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

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

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

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

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

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

### 3.4 Hashes

Hashes can be used to store mapped information:

```
1 mark = {}
2 mark['English'] = 50
3 mark['Math'] = 70
4 mark['Science'] = 75
```

And we can define a default value:

```
1 mark = {}
2 mark.default = 0
3 mark['English'] = 50
4 mark['Math'] = 70
5 mark['Science'] = 75
```

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

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

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

```
1 total = 0
2 mark.each { |key,value|
3   total += value
4 }
5 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:

```
1 mark = {}
2 mark[:English] = 50
3 mark[:Math] = 70
4 mark[:Science] = 75
```

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

```
1 c = "able was i ere i saw elba"
2 d = "able was i ere i saw elba"
3 >> c.object_id
4 => 21472860
5 >> .object_id
6 => 1441620
```

```
1 e = :some_symbol
2 f = :some_symbol
3 >> e.object_id
4 => 1097628
```

```
5 >> f.object_id
6 => 1097628
```

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

```
1 >> hash = {a: 1, b: 2, c: 3}
2 => {:a=>1, :b=>2, :c=>3}
3 >> hash.transform_values{ |value| value * value }
4 => {: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 >> (1..5).each {|a| print "#{a}, " }
2 => 1, 2, 3, 4, 5, => 1..5
```

We can also use them on strings:

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

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

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

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

```
1 print "Enter any letter: "
2 letter = gets.chomp
3
4 puts "You have entered a lower case letter" if ('a'..'z') === letter
5 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 >> (1..5).to_a
2 => [1, 2, 3, 4, 5]
3 >> (1...5).to_a
4 => [1, 2, 3, 4]
```

It is also possible to define endless ranges:

```
1 print "Enter your age: "
2 age = gets.to_i
3
4 case age
5 when 0..18
6   puts "You are a kid"
7 when (19..)
8   puts "You are grownup"
9 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:

```
1 def print_line
2   puts '_' * 20
3 end
4
5 print_line
```

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

```
1 def print_line length = 20
2   puts '_' * length
3 end
4
5 print_line
6 print_line 40
```

Arguments are always passed by reference:

```
1 def array_changer array
2   array << 6
3 end
4
5 some_array = [1, 2, 3, 4, 5]
6 p some_array
7 array_changer some_array
```

```
8 p some_array
9
10 => [1, 2, 3, 4, 5]
11 => [1, 2, 3, 4, 5, 6]
```

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

```
1 def addition x, y
2   x + y
3 end
4
5 addition 3, 5
6
7 => 8
```

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

```
1 def say_hello name: "Martin", age: 33
2   puts "Hello #{name} your age is #{age}"
3 end
4
5 say_hello name: "Joseph", age: 7
```

Arguments can also be variadic:

```
1 def some_function a, *others
2   puts a
3   others.each do |x|
4     puts x
5   end
6 end
7
8 some_function 1,2,3,4,5
```

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

```
1 def print_something string
2   puts string
3 end
4
5 def decorate(...)
6   puts "#" * 50
7   print_something(...)
8   puts "#" * 50
9 end
10
11 decorate "Hello World!"
```

We can also define a function in more concise way:

```
1 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:

```
1 class Square
2 end
```

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

```
1 class Square
2   attr_accessor :side_length
3 end
```

They can be read and written with `.`:

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

Methods can be defined with `def`:

```
1 class Square
2   attr_accessor :side_length
3
4   def area
5     @side_length * @side_length
6   end
7
8   def perimeter
9     4 * @side_length
10  end
11 end
```

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

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

```
1 class Square
2   attr_accessor :side_length
3
4   def initialize side_length = 0
5     @side_length = side_length
6   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 omitting 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
```



```
11   @@robot_count
12   end
13 end
```

Similarly so, we can define class constants like so:

```
1  class Something
2    Const = 25
3
4    def Const
5      Const
6    end
7  end
8
9  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:

```
1  class Rectangle
2    attr_accessor :length, :width
3  end
4
5  class Square < Rectangle
6    def initialize length
7      @width = @length = length
8    end
9
10   def side_length
11     @width
12   end
13 end
```

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

```
1  class Square < Rectangle
2    def set_dimension side_length
3      super side_length, side_length
4    end
5  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`:

```
1  >> "a".methods
2  =>
3  [:unicode_normalized?,
```

```
4  :encode!,
5  :unicode_normalize,
6  :ascii_only?,
7  :unicode_normalize!,
8  :to_r,
9  :encode,
10 :to_c,
11 :include?,
12 :%,
13 :*,
14 :+,
15 :unpack,
16 # ...
17 ]
```

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

Another feature fairly unique to Ruby is method aliasing:

```
1  class Something
2    def make_noise
3      puts "AAAAAAAAAAAAAAAAHHHHHHHHHHHHHHHHHH"
4    end
5
6    alias :shout :make_noise
7  end
8
9  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:

```
1  person = Class.new do
2    def say_hi
3      'Hi'
4    end
5  end.new
```

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

```
1  class Robot
2    attr_accessor :name
3  end
4
5  robot = Robot.new
6  robot.name = "Zigor"
```

```
7 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:

```
1 # break_square.rb
2
3 class Square
4   attr_accessor :side_length
5
6   def perimeter
7     @side_length * 4
8   end
9 end
```

```
1 # break_main.rb
2
3 require "./break_square.rb"
4
5 s = Square.new
6 s.side_length = 5
7 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:

```
1 module Star
2   def line
3     puts '*' * 20
4   end
5 end
6
7 module Dollar
8   def line
9     puts '$' * 20
10  end
11 end
```

If we `include` `Star` and call `line`, we will print a line of stars, 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:

```
1 >> Dollar::line
2 => $$$$$$$$$$$$$$$$$$$$$$
```

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:

```
1 class Sphere
2   include Constants
3   attr_accessor :radius
4
5   def volume
6     (4.0/3) * Pi * radius ** 3
7   end
8 end
```