

RUBY *for* Testing Automation

By Eduardo Gomes Heinen

Versions

Introduction

In this ebook we will talk about Ruby to be used in testing automation. You will learn some important concepts about Ruby and how to start applying in your projects.



Eduardo Heinen

How to get it better

To improve the speed of your learning, I recommend you redo all examples. That way you will understand better how it works. I created a repository to train people on Github. You can also try to develop the exercise there. The link to repository is [here](#).

You will see a pattern of blocks in the ebook, following what which block means:

Terminal or Prompt Command (cmd)

irb - Interactive Ruby Console

Just enter **irb** on your Terminal on Linux or Mac
Open the interactive ruby exe on Windows

The signal => means return or result of some execution

Command sintaxes

Explanation of something

Tips

Share this book

If you enjoy this book, please share it in your social media and invite me on LinkedIn. You may also send me an email, I will read it as soon as possible, promise.

LinkedIn: <https://linkedin.com/in/eduardogheinen>

Github: <https://github.com/eheinen/>

Email: eduardogheinen@gmail.com

Eduardo Heinen

Summary

OOP Fundamentals

[Object-Oriented Programming](#)

Ruby Fundamentals

[Installing Ruby](#)

[Configuring Ruby Project](#)

[Basic Commands](#)

[Comments](#)

[Variables I](#)

[Operators](#)

[Array](#)

[Hash](#)

Loops

[While](#)

[For](#)

[Each](#)

[Each with index](#)

[Times](#)

[Until](#)

Conditions

[IF](#)

[UNLESS](#)

[Conditions Inline](#)

[Methods](#)

[Variables II](#)

[Class](#)

[Handle Exceptions](#)

[References](#)

OOP Fundamentals

Object-Oriented Programming

Before to start developing, let's talk about OOP concepts a little bit.

- Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects.

Objects

Any entity that has state and behaviour is known as an object. Object means a real word entity. It can be physical and logical.

Example: TV, radio, mouse, keyboard, etc.

Class

Collection of objects is called class. It is a **logical** entity.

A class is simply a representation of a type of object. It is the blueprint, or plan, or template, that describes the details of an object.

A class is the blueprint from which the individual objects are created.

Class is composed of three things: A **name**, **attributes**, and **operations**.

Example: Fruit, vehicle, colour, animal, etc.

OOP Fundamentals

Object-Oriented Programming

Attributes

Attributes may be defined as characteristics of a class.

For example:

Class: **Animal**

Which characteristics an animal has?

- Type (Kind of the animal)
- Eye colour
- Hair colour
- Size
- Weight
- Behaviour
- Etc.

OOP Fundamentals

Object-Oriented Programming

Operations | Methods | Functions

For now, pretend those 3 names are the same thing for us. (There are a little difference between them).

An operator is composed by some instructions passed to a block to do something.

Normally, it describes an action of something. For example:

In the same **Animal** class, let think about which actions an animal may has.

- Eat
- Run
- Poop :)
- Search food
- Etc.

In each action above, we have to describe which **instructions** are necessary be executed to achieve its purpose. For example:

Operation: Poop :)

- The animal needs to search food;
- The animal needs to kill/catch the food;
- The animal needs to eat the food;
- The animal needs to wait the food be digested;
- The animal search a place to poop;
- The animal poop.

OOP Fundamentals

Object-Oriented Programming

Operations | Methods | Functions

In the last example we defined some steps to be followed until an animal is able to poop. That was a simple physical representation. We could create an logical representation of our animal using a class with attributes and operations to define the actions.

Eduardo Heinen

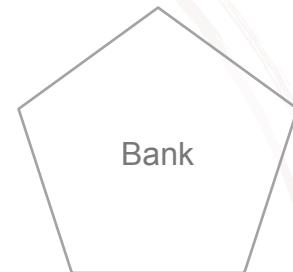
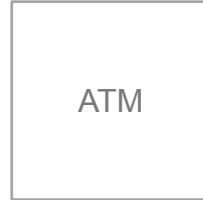
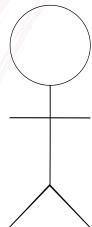
OOP Fundamentals

Object-Oriented Programming

Exercise:

Now, you will think about which class, attributes and operations the following images could have:

Bank
Customer



I challenge you think about restrictions also.

For example:

- A bank costumer must be authenticated to do anything in the ATM.
- ???

Ruby Fundamentals

Ruby is a language programming designed and developed in the mid of 1990s by Yukihiro Matsumoto in Japan. And the language is dynamic, reflective, and object-oriented.

Installing Ruby

To install Ruby on Windows you need to download a version of Ruby and install through Ruby Installer. The link where you will find the ruby versions: <http://rubyinstaller.org/downloads/>

After you installed Ruby, you need to download and install Ruby DevKit. It is found in the same link above.

Once you have installed Ruby and Ruby DevKit, you must to include your Ruby path to a environment variable.

I recommend you use this [tutorial](#).

To install Ruby on Ubuntu or Mac OS I strongly recommend use RVM to manage the Ruby versions in your computer. You need just install using this link: <https://rvm.io/rvm/install>

After you installed RVM just open your terminal and type it:

```
rvm install 2.2.3
```

You are able to change the Ruby version.

Once you have Ruby installed in your machine, navigate to your project directory and execute the following command:

```
gem install bundler
```

Ruby Fundamentals

Configuring Ruby Project

We installed Bundler because we will use it to install our gems through a easier way.

If you don't want to install gems with bundler, just include the following command in your terminal:

```
gem install 'name of your gem'
```

I don't recommend to install gems without bundler, because if you are sharing your project with other people in your team and a certain gem must has an exactly version to work with your project, then someone may install a different version of yours.

Furthermore, we just install all gems once using the command:

```
bundle install
```

Bundler will search for the file Gemfile in the same directory you are at the moment you typed it. For instance, If you are inside 'workspace' directory and execute "bundle install", then you must have a Gemfile inside 'workspace' directory.

Bundler is similar to Maven on Java and Gemfile is similar to pom.xml

Ruby Fundamentals

Basic Commands

In Ruby we have a lot of resources to use in our day by day and know them is really important to develop faster and better.

You can access the Ruby website to get more information about:

<https://www.ruby-lang.org/en/documentation/quickstart/> and

<http://tryruby.org/>

To start to play with Ruby, just open your terminal and execute:

```
irb
```

The command above will open the terminal to interact with Ruby. Now you can start to play! Let's try!

```
n1 = "Hello World!"  
n1  
=> Hello World!
```

In the terminal, attribute the text "Hello World!" to a variable and then just type the variable name and enter.

Congratulations for your first lines of code!

Ruby Fundamentals

Comments

Comments are used to document something and it won't be executed. I recommend you comment all your methods because that is a good practice. Sometimes we just develop a lot of methods and when we have to maintain, or even use it, we don't remember what it does.

If you want to comment something in Ruby, you just need to include the "#" symbol before the wordier phrase.

```
# That is a simple comment inline.
```

There is another comment used to comment a block. Let's see:

```
=begin
```

Everything you include here will be commented.

This comment is really useful when you need to comment more detailed.

```
=end
```

Ruby Fundamentals

Variables I

Variables are labels used to store a value. In other languages is mandatory to define which kind of variable is and when you pass a value to this variable, must be the same type.

In Ruby we don't define the kind of our variables, Ruby identifies it implicitly.

Variables have scopes (life time), that we will see in the second part of variables.

```
a = 10  
=> 10  
  
b = 10.5  
=> 10.5  
  
c = "Hello World!"  
=> "Hello World!"
```

Ruby Fundamentals

Operators - Arithmetic

Operators are important because depending on which project you are developing you will use it. Follow some examples you are able to try it also:

```
a = 10  
b = 2  
  
addition      = a + b  
subtraction   = a - b  
multiplication = a * b  
division       = a / b  
modulus        = a % b  
exponation     = a ** b
```

```
# Result:  
  
addition      = 12  
subtraction   = 8  
multiplication = 20  
division       = 5  
modulus        = 0  
exponation     = 100
```

Ruby Fundamentals

Operators - Comparison

Some comparison operators are mandatory to know by heart! For sure you will use it in your project.

a = 10	
b = 2	
equality	= a == b
difference	= a != b
left_greater	= a > b
left_greater_or_equal	= a >= b
left_lesser	= a < b
left_lesser_or_equal	= a <= b
combined	= a <=> b
type_and_value_equality	= a.eql? b
same_object	= a.equal? b

Result:

equality	= false
difference	= true
left_greater	= true
left_greater_or_equal	= true
left_lesser	= false
left_lesser_or_equal	= false
combined	= 1
type_and_value_equality	= false
same_object	= false

Ruby Fundamentals

Operators - Comparison <=>

Notice in the three last comparison operators are different a little bit. Let me explain them:

The operator "<=>" is a combined operator used to get a value depends on the comparison. If the first operand is lesser than second operand, then it will return the value **-1**. If the first operand is equal the second operand, then it will return the value **0**. And if the first operand is greater than the second operand, then it will return the value **1**.

```
a = 9 <=> 10  
b = 10 <=> 10  
c = 10 <=> 9
```

```
-1  
0  
1
```

Ruby Fundamentals

Operators - Comparison **==**

The operator "**==**" is used to compare an equality inside a "when" statement. For instance, if you have a range of values to iterate and you want to include a certain number to test the equality with one number in iteration, then you may include the operator "**==**" to works for you.

```
a = (0...100) == 150  
  
min_value = 12  
max_value = 31  
b =(min_value..max_value) == 30
```

```
a = false  
b = true
```

In the variable "**a**" it was false because it didn't find the number 150 inside the loop between 0 and 100. And in the variable "**b**" it was true because the number 30 is between the min and max value that we have passed.

Ruby Fundamentals

Operators - Comparison `eql?` and `equal?`

The operator "`eql?`" is used to compare if the type and value of our operands are the same.

```
a = 10.eql? 10  
b = 10.eql? 11  
c = 10.eql? 10.0
```

```
a = true  
b = false  
c = false
```

Observer the variable "`a`" is true because it has the same value and same type (Fixnum). But in the variables "`b`" and "`c`" are false because in "`b`" the value is different and "`c`" the value is relatively the same except for the variable type (Float).

```
a = 10.class  
b = 10.0.class
```

```
a = Fixnum  
b = Float
```

If you want to know which type is your variable/
object just include `.class`

Ruby Fundamentals

Array

When you create a variable of an Array in instance, you are able to store more than one value in this variable, differently of normal variables that stores just one value per time.

Arrays are used in practically all language programmings and it's super useful.

In Ruby you may add any type of value inside an array and you can create an Array object using `Array.new` or just `[]`. Let's see.

```
array = []
array = Array.new
```

Adding items/values

If you want to add a value to an array, just use "`<<`" signal:

```
array << 1
=> [1]
array << 1.5
=> [1, 1.5]
array << "Hello"
=> [1, 1.5, "Hello"]
array << true
=> [1, 1.5, "Hello", true]
array << Array.new
=> [1, 1.5, "Hello", true, []]
```

<https://ruby-doc.org/core-2.2.0/Array.html>

Ruby Fundamentals

Array

Accessing items/values

If you want to access a value in an array, just use "[index]" where index is the position (Starting by **0**) or even using the method "**at**":

```
array = [1, 2, 3, 4, 5]

array[0] # Accessing the first item
=> 1

array[1] # Accessing the second item
=> 2

array.at(0) # Accessing the first item
=> 1

array.at(1) # Accessing the second item
=> 2
```

You can also use ".first" or ".last" to get the first and last item consecutively. Example:

```
array = [1, 2, 3, 4, 5]

array.first # Accessing the first item
=> 1

array.last # Accessing the last item
=> 5
```

Ruby Fundamentals

Array

Deleting items/values

If you want to delete a value in an array, just use "**delete**" passing the value you want to remove or even use "**delete_at**" passing the index position:

```
array = [1, 2, 3, 4, 5]

# Deleting the item with value equals to 1
array.delete(1)
=> [2, 3, 4, 5]

# Deleting the item with index equals to 1
array.delete_at(1)
=> [1, 3, 4, 5]
```

Ruby Fundamentals

Array

Min and Max

Min and max are methods used to get the minimum and maximum value consecutively.

```
array = [1, 2, 3, 4, 5, 5, 6, 7, 3]
array.min
=> 1

array.max
=> 7
```

Reverse

Reverse is a method used to reverse the order of your array.

```
array = [1, 2, 3, 4, 5, 5, 6, 7, 3]
array.reverse
=> [3, 7, 6, 5, 5, 4, 3, 2, 1]
```

Shuffle

Shuffle is a method used to reorganise randomly your array.

```
array = [1, 2, 3, 4, 5, 5, 6, 7, 3]
array.shuffle
=> [5, 7, 2, 6, 5, 1, 3, 4, 3]
```

Ruby Fundamentals

Array

Uniq

If you want reject duplicate values in your array, then you should use uniq method.

```
array = [1, 1, 2, 2, 3]
array.uniq
=> [1, 2, 3]
```

Length

Length is a method used to get how many items there are in your array.

```
array = [1, 2, 3, 4, 5, 5, 6, 7, 3]
array.length
=> 9
```

Select

Select is powerful method used to filter values according to your specification. In the following example, I select just values lesser than 5:

```
array = [1, 2, 3, 4, 5, 5, 6, 7, 3]
array.select { |value| value < 5 }
=> [1, 2, 3, 4, 3]
```

Ruby Fundamentals

Array

There are a lot of methods inside Array class you can use. Take a look:

array._id_	array.each	array.instance_eval	array.pop	array.size
array._send_	array.each_cons	array.instance_exec	array.private_methods	array.slice!
array.all?	array.each_entry	array.instance_of?	array.product	array.slice_after
array.any?	array.each_index	array.instance_variable_defined?	array.protected_methods	array.slice_before
array.assoc	array.each_slice	array.instance_variable_get	array.public_method	array.slice_when
array.at	array.each_with_index	array.instance_variable_set	array.public_methods	array.sort
array.bsearch	array.each_with_object	array.instance_variables	array.public_send	array.sort!
array.bsearch_index	array.empty?	array.is_a?	array.push	array.sort_by!
array.chunk	array.entries	array.itself	array.rassoc	array.sort_by!
array.chunk_while	array.enum_for	array.join	array.reduce	array.taint
array.class	array.eql?	array.keep_if	array.reject	array.tainted?
array.clear	array.equal?	array.kind_of?	array.reject!	array.take
array.clone	array.extend	array.last	array.remove_instance_variable	array.take_while
array.collect	array.fetch	array.lazy	array.repeated_combination	array.tap
array.collect!	array.fill	array.length	array.repeated_permutation	array.to_a
array.collect_concat	array.find	array.map	array.replace	array.to_ary
array.combination	array.find_all	array.map!	array.respond_to?	array.to_enum
array.compact	array.find_index	array.max	array.reverse	array.to_h
array.compact!	array.first	array.max_by	array.reverse!	array.to_s
array.concat	array.flat_map	array.member?	array.reverse_each	array.transpose
array.count	array.flatten	array.method	array.rindex	array.trust
array.cycle	array.flatten!	array.methods	array.rotate	array.uniq
array.define_singleton_method	array.freeze	array.min	array.rotate!	array.uniq!
array.delete	array.frozen?	array.min_by	array.sample	array.unshift
array.delete_at	array.grep	array.minmax	array.select!	array.unshift!
array.delete_if	array.grep_v	array.minmax_by	array.send	array.untrust
array.detect	array.group_by	array.nil?	array.shift	array.untracked?
array.dig	array.hash	array.none?	array.shuffle	array.values_at
array.display	array.include?	array.object_id	array.shuffle!	array.zip
array.drop	array.index	array.one?	array.singleton_class	
array.drop_while	array.inject	array.pack	array.singleton_method	
array.dup	array.insert	array.partition	array.singleton_methods	
	array.inspect	array.permutation		

If you want to check which methods a Class has, just type the name of the class and ".methods".

Example: `Array.methods`

Ruby Fundamentals

Hash

Hash is a really useful resource in Ruby, because we work with key and value. We can create an instance of Hash just using Hash.new or even {}.

```
hash = {}  
=> {}  
  
hash = Hash.new  
=> {}
```

Adding items:

To add items to a hash is pretty simple, just indicate the key and the value of this key. You are able to use symbols to define a key.

```
hash = {}  
hash['my_key'] = "my value"  
  
hash = { 'a' => 1, 'b' => 2, 'c' => 3 }  
  
hash[:name] = "Eduardo Heinen"
```

A good example of where Hash is used is JSON parses

<https://ruby-doc.org/core-2.2.0/Hash.html>

Ruby Fundamentals

Hash

Accessing items:

To access items in a hash, just use "[key]" where key is the name of your key.

```
hash = { 'a' => 1, 'b' => 2, 'c' => 3 }
hash['a']
=> 1

hash = { :a=> 1, :b=> 2, :c=> 3 }
hash[:b]
=> 2
```

Deleting items:

To delete items in a hash, just use "**delete(key)**" and replace key for your key.

```
hash = { 'a' => 1, 'b' => 2, 'c' => 3 }
hash.delete('a')
=> { 'b' => 2, 'c' => 3 }

hash = { :a=> 1, :b=> 2, :c=> 3 }
hash.delete(:b)
=> { :a=> 1, :c=> 3 }
```

Ruby Fundamentals

Hash

Count:

Count is a method used to count how many items a hash has.

```
hash = { 'a' => 1, 'b' => 2, 'c' => 3 }
hash.count
=> 3
```

Keys:

Keys is a method used to list all keys inside the hash (returned as Array).

```
hash = { 'a' => 1, 'b' => 2, 'c' => 3 }
hash.keys
=> ["a", "b", "c"]
```

Values:

Values is a method used to list all values inside the hash (returned as Array).

```
hash = { 'a' => 1, 'b' => 2, 'c' => 3 }
hash.values
=> [1, 2, 3]
```

Ruby Fundamentals

Hash

Select:

Select is a method used to filter values in a hash. In the following example I filter first by value and after that by key.

```
person1 = { first_name: "Eduardo", last_name: "Heinen", hobbie: "Develop" }

person1.select { |key, value| value == "Eduardo" }
=> { :first_name=>"Eduardo" }

person1.select { |key, value| key == :hobbie }
=> { :hobbie=>"Develop" }
```

Merge:

Merge is a method used to merge values in a hash. If the key is the same but the value is different between those hashes, then the first hash will be replaced by the value of the second hash.

```
numbers1 = { a: 1, b: 10, c: 3 }
numbers2 = { a: 1, b: 15, c: 3 }
numbers1.merge(numbers2)
=> { :a=>1, :b=>15, :c=>3 }
```

Ruby Fundamentals

Hash

There are a lot of methods inside Hash class you can use. Take a look:

hash._id	hash.dig	hash.first	hash.keep_if	hash.protected_methods	hash.sort_by
hash._send_	hash.display	hash.flat_map	hash.key?	hash.public_method	hash.store
hash.all?	hash.drop	hash.flatten	hash.keys	hash.public_methods	hash.taint
hash.any?	hash.drop_while	hash.freeze	hash.kind_of?	hash.public_send	hash.tainted?
hash.assoc	hash.dup	hash.frozen?	hash.lazy	hash.rassoc	hash.take
hash.chunk	hash.each	hash.grep	hash.length	hash.reduce	hash.take_while
hash.chunk_while	hash.each_cons	hash.grep_v	hash.map	hash.rehash	hash.tap
hash.class	hash.each_entry	hash.group_by	hash.max	hash.reject	hash.to_a
hash.clear	hash.each_key	hash.has_key?	hash.max_by	hash.reject!	hash.to_enum
hash.clone	hash.each_pair	hash.has_value?	hash.member?	hash.remove_instance_variable	hash.to_h
hash.collect	hash.each_slice	hash.include?	hash.merge	hash.replace	hash.to_hash
hash.collect_concat	hash.each_value	hash.index	hash.merge!	hash.respond_to?	hash.to_proc
hash.compare_by_identity	hash.each_with_index	hash.inject	hash.method	hash.reverse_each	hash.to_s
hash.compare_by_identity?	hash.each_with_object	hash.inspect	hash.methods	hash.select	hash.trust
hash.count	hash.entries	hash.instance_eval	hash.min	hash.send	hash.untrust
hash.cycle	hash.enum_for	hash.instance_exec	hash.min_by	hash.shift	hash.untrusted?
hash.default	hash.eql?	hash.instance_of?	hash.minmax	hash.singleton_class	hash.update
hash.default=	hash.equal?	hash.instance_variable_defined?	hash.minmax_by	hash.singleton_method	hash.value?
hash.default_proc	hash.extend	hash.instance_variable_get	hash.nil?	hash.singleton_methods	hash.values
hash.default_proc=	hash.fetch	hash.instance_variable_set	hash.none?	hash.size	hash.values_at
hash.define_singleton_method	hash.fetch_values	hash.instance_variables	hash.object_id	hash.slice_after	hash.zip
hash.delete	hash.find	hash.invert	hash.one?	hash.slice_before	
hash.delete_if	hash.find_all	hash.is_a?	hash.partition	hash.slice_when	
hash.detect	hash.find_index	hash.itself	hash.private_methods	hash.sort	

Eduardo Heinen

Ruby Fundamentals

Loops - While

```
while condition [do]
  code
end
```

```
count = 0
while count < 5
  count += 1
  puts count
end
```

Once you executed the above statement then you should see this:

```
1
2
3
4
5
```

Ruby Fundamentals

Loops - For

```
for variable [, variable ...] in expression [do]
  code
end
```

```
for i in 0..5
  puts i
end
```

Once you executed the above statement then you should see this:

```
0
1
2
3
4
5
```

Notice in the code has ".." between 0 and 10. We also could put "...". The difference between both ".." and "..." is:

"0..10" is equal to "0 >= 10" and "0...10" is equal to "0 > 10"

If we had put "..." then we had seen printed 0 until 9 just. The loop would executed while "i" was lesser than "10".

Ruby Fundamentals

Loops - Each

```
(expression).each do |variable[, variable...]|
  code
end
```

```
(0..5).each do |variable|
  puts variable
end
```

Once you executed the above statement then you should see this:

```
0
1
2
3
4
5
```

The loop each is really useful daily, principally when used with arrays like:

```
fruits = ["Banana", "Strawberry", "Lemon", "Orange"]
fruits.each do |fruit|
  puts fruit
end
```

Ruby Fundamentals

Loops - Each and Each with index

In our example of each with fruit array, after I executed in my terminal I got the following result:

```
Banana - 0
Strawberry - 1
Lemon - 2
Orange - 3
```

Case you also need to get the index, then there is another command to execute:

```
(expression).each_with_index do |variable, index|
  code
end
```

```
fruits = ["Banana", "Strawberry", "Lemon", "Orange"]
fruits.each_with_index do |fruit, index|
  puts fruit + " - " + index.to_s
end
```

Remember, the second variable is the index, take a look at the statement executed:

```
Banana - 0
Strawberry - 1
Lemon - 2
Orange - 3
```

Ruby Fundamentals

Loops - Times

```
number.times do |variable|
  code
end
```

```
5.times do |i|
  puts i
end
```

Once you executed the above statement then you should see this:

```
0
1
2
3
4
5
```

There is another way to execute times like:

```
10.times { puts "hello!" }
```

The statement above will print 10 times the word "hello!"

Ruby Fundamentals

Loops - Until

```
until conditional [do]
  code
end
```

```
count = 0
until count > 5
  count += 1
  puts count
end
```

Once you executed the above statement then you should see this:

```
1
2
3
4
5
6
```

Ruby Fundamentals

Condition IF

```
if conditional [then]
  code...
[elsif conditional [then]
  code...]...
[else
  code...]
end
```

```
fruit = "strawberry"
if fruit == "strawberry"
  puts "Strawberry!"
elsif fruit == "banana"
  puts "Banana!"
else
  puts "Other fruit!"
end
```

Once you executed the above statement then you should see this:

```
1
2
3
4
5
6
```

Ruby Fundamentals

Condition UNLESS

```
unless conditional [then]
  code
[else
  code ]
end
```

```
price = 950
unless price >= 1000
  puts "Hey, the price is lesser than $1000"
else
  puts "Hey, the price is bigger than $1000"
end
```

Once you executed the above statement then you should see this:

```
Hey, the price is lesser than $1000
```

Ruby Fundamentals

Conditions Inline

You may also work with conditions inline. When we are developing an IF or UNLESS block we are obligated to include the keyword **end** to finish our block. But when we work inline it is not necessary.

```
code if condition
```

```
code unless condition
```

```
count = 100  
count += 50 if count >= 100  
count
```

```
=> 150
```

```
count = 100  
count -= 50 unless count < 100  
count
```

```
=> 50
```

Ruby Fundamentals

Methods

If you want to create a method, just use the following syntax:

```
def method_name(params)
  code
end
```

or

```
def method_name params
  code
end
```

Notice these methods above, the unique difference is ")("). You are able to define which one you want to use. It doesn't matter to Ruby.

```
def format_fruits fruit
  "I love eating " + fruit
end

format_fruits "strawberry"
```

When executed the method above, it will return a phrase with the fruit's name that you passed. Like:

```
=> I love eating strawberry
```

Ruby Fundamentals

Methods

Notice that all methods in Ruby returns something! Even if you just include a `puts`, it will return `nil`. You may also include `return` to return something. Like:

```
def format_fruits fruit
  return "I love eating " + fruit
end

format_fruits "strawberry"
```

There is no difference with or without `return` in the method, the return will be the same.

```
=> I love eating strawberry
```

Ruby Fundamentals

Methods

If your method has an argument and it doesn't need to be required, you may initialize this argument and when you call your method you choose if you want or not include arguments. Let's see:

```
def format_fruits(fruit = "strawberry")
  return "I love eating " + fruit
end

format_fruits
```

```
=> I love eating strawberry
```

Observe I just called the method `format_fruits` without pass any arguments. When the method was executed I could see the value "strawberry" I included as default value.

Ruby Fundamentals

Methods

You also may define an argument able to receive various value, like an array. Let's see:

```
def format_fruits(*fruits)
  fruits.each do |fruit|
    puts fruit
  end
end

format_fruits("strawberry", "banana", "orange")
```

```
strawberry
banana
orange
```

Actually, Ruby transforms the argument fruits in an Array type.

Ruby Fundamentals

Variables II

In Ruby there are 4 kinds of variable scope in Ruby:

Local Variables: Local variables are defined inside methods and can't be accessed outside of the method. Local variables begin with a "lowercase letter" or "_".
Example: count = 0 or _count = 0

Instance Variables: Instance variables are available across methods for any particular instance or object. Instance variables are preceded by the at sign "@" followed by the variable name.

Example: @array = Array.new

Class Variables: Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign "@@" and are followed by the variable name.

Example: @@color

Global Variables: Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign "\$".

Example: \$status

Ruby Fundamentals

Class

```
class ClassName  
  ...  
end
```

When you create a class, it's important to respect patterns. The name of the class uses Pascal case, first letter of each word is uppercase.

To instantiate a class (create objects of a class) you need just write the class name proceeded of ".new".

```
class Fruit  
end  
  
fruit = Fruit.new
```

If you execute the code above you will see something like this:

```
=> #<Fruit:0x007f9d3d0d5278>
```

Now you are able to interact with the methods inside the class.

Ruby Fundamentals

Class

```
class Fruit
  def format_fruits(fruit)
    fruit.capitalize
  end
end

fruit = Fruit.new
fruit.format_fruits("sTrAWBerry")
```

Observe I create an object of Fruit class and called the method format_fruits passing a value not formatted. After I executed it you are able to see the following statement:

```
=> Strawberry
```

If you would like to know which class an object belongs, just type ".class" after the variable. Like:

Ruby Fundamentals

Class

```
fruit.class  
=> Fruit < Object
```

Sometimes we need to interact with a class specifically, and to do it work, we need to be sure which class our variable is. To be sure about it we just can use: "instance_of?"

```
fruit.instance_of? Fruit  
=> true
```

```
fruit.instance_of? Object  
=> false
```

Ruby Fundamentals

Class

Notice our variable fruit is not an instance of Object, but it is an Object because the Object class is the superclass (All classes are children of Object). Now if you try ".is_a? Object" or ".kind_of? Object", the result will be "true". Let's see:

```
fruit.is_a? Object  
=> true
```

```
fruit.kind_of? Object  
=> true
```

Ruby Fundamentals

Class

In all variable you are able to check if it is nil or empty, like:

```
fruit = nil  
fruit.nil?  
  
=> true
```

```
name = ""  
name.empty?  
  
=> true
```

Eduardo Heinen

Ruby Fundamentals

Class

In Ruby classes we are able to create a constructor to initialize things before to interact with something else inside the class, like:

```
class Fruit
  def initialize(fruit, flavor)
    @fruit = fruit
    @flavor = flavor
  end

  def save
    DataBase.new.save(@fruit, @flavor)
    puts "Fruit successfully saved"
  end
end
```

```
Fruit.new("Banana", "Sweet").save
```

```
Fruit successfully saved
```

Notice I didn't need to create a variable to store my object class to interact with elements inside the class. I just create the object and called the method "save" inline.

Ruby Fundamentals

Handle Exceptions

Treatment of exception is really important when you are developing, because sometimes the user or something else makes something unexpected like a division by zero! So, when an exception occurs the program stops! You need to foresee situations that can explode an exception in your system.

```
begin
  code
rescue Exception One
  code
rescue Exception Two
  code
else
  code
ensure
  code
end
```

We have to define a block if we want to get the exception followed by **rescue** keyword. If you know the kind of exception that could happen you may inform after rescue, like:

Ruby Fundamentals

Handle Exceptions

```
def divide(number1, number2)
    number1 / number2
end

divide 10, 0
```

```
# Exception thrown:
ZeroDivisionError: divided by 0
```

```
def divide(number1, number2)
begin
    number1 / number2
rescue ZeroDivisionError
    puts "Ops! You cannot divide by zero"
end
end

divide 10, 0
```

```
Ops! You cannot divide by zero
```

Ruby Fundamentals

Handle Exceptions

If you don't know which kind of exception could be thrown in your method, just put rescue and treat it in a generic format. Like:

```
def divide(number1, number2)
  begin
    number1 / number2
  rescue
    puts "Ops! Something went wrong!"
  end
end

divide 10, 0
```

```
Ops! Something went wrong!
```

If you need something be always executed after do something, you may use **ensure**. It will execute your code inside your method and if something goes wrong then ensure will do something. The ensure is widely used to close connections with database. Even the method doesn't throw any exception you need to close the database connection, or else, it will be open.

Ruby Fundamentals

Handle Exceptions

If you don't know which kind of exception could be thrown in your method, just put rescue and treat it in a generic format. Like:

```
def save
  begin
    # Code to open database connection
    # Code to save and commit
  rescue
    puts "Ops! Something went wrong!"
  ensure
    # Code to close database connection
    puts "Database was closed!"
  end
end
```

The ensure above will **always** be executed! It doesn't matter if everything goes right or not. The database connection will be closed.

Ruby Fundamentals

Handle Exceptions

You may also want to throw an exception, and you can do it using raise. Especially if you are working with testing automation.

```
def divide(number1, number2)
  raise "You cannot divide by zero" if number2.zero?
  number1 / number2
end

divide 10, 0
```

```
=> RuntimeError: You cannot divide by zero
```

If you are executing a test and an exception is thrown, then your test will fail. That is a kind of the Asserts' principle.

References

<http://www.javatpoint.com/java-oops-concepts>

<https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep>

<https://ruby-doc.org/core-2.2.0/>

Eduardo Heinen