Programming Ruby

Marek Hulán, Marek Jelen, Ivan Nečas

Version 0.2, September 25, 2017

Table of Contents

Ruby basics

Get started with Ruby

Chapter 1. Instalation

1.1. Linux

Do not use prepackaged Ruby on Linux distributions.

On Linux use `rbenv` package manager with the `ruby-build` extension to install Ruby.

1.2. Windows

For windows there is an installer Ruby Installer simplifies the installation to the "windows standard approach".

1.3. macOS

The simplest way to install Ruby on macOS is to use `homebrew` package manager

```
$ brew install ruby
```

in case there is a need to have multiple installations `rbenv` is a good tool to use. Another popular tool is called `rvm` but nowadays they do too much magic and we recommend using `rbenv` instead.

Chapter 2. Running Ruby

Usually you put `ruby` and relates command on PATH so you do not need to always specify the whole path to the executable. With tools like `rbenv` this is done for you.

```
$ ruby -v
```

runs Ruby and prints it’s version. Simplest way to verify that your instalation works as expected.

```
$ ruby script.rb
```

interprets the script.rb. However instead of running the script explicitly, you can add the shebang line to your script

```
#!/usr/bin/env ruby
```

and make it executable (`chmox +x script.rb`) and run it simply as `./script.rb`.

2.1. Interactive Ruby

Ruby comes with an interactive interpreter called `irb` it allows you to enter commands and see results instantly.

```
$ irb
```

Chapter 3. About Ruby

Ruby a programming language with following aspects

3.1. Interpreted

The Ruby code is interpreted when the script is load, there is no compile phase.

3.2. Universal

Ruby can be used for writing scripts, one-liners, web applications and even for mobile and desktop applications.

3.3. Fully Object oriented

In Ruby everything is an object and you send messages to those objects.

3.4. Everything is an expression

Everything command in Ruby has a return value.

3.5. Atomic memory management

3.6. MRI

`MRI` is the Ruby reference implementation, with several other implementations like `JRuby`, `Rubinius`, `IronRuby` or `MagLev`. As well, there is ISO standard implemented in `mruby`.

3.7. Strongly typed

There are some implicit conversions (e.g. every object has `to_s` method to provide string representation) but in most cases in case you try to operate on different types Ruby will complain, unless such operation is explicitly provided.

```
5 + "a" # => TypeError: String can't be coerced into Fixnum
```

3.8. Dynamically typed

The type of a variable is defined by the value assigned to that variable. There is no explicit type information in the code.

```
a = "string"
```

3.9. What is written in Ruby?

- programming languages: Ruby (Rubinius), compilers (LessJS)
- web applications: Github, Gitlab, Redmine, BaseCamp
- devops tools: Puppet/Chef/Vagrant
- cloud platforms: OpenShift (v2)
- cloud management: Foreman, ManageIQ
- VIM/Emacs scripts
- static pages generators - Jekyll
- programming tools

Chapter 4. Conventions

- class names are CamelCase
- file names reflect class names in snake_case format
- method names are snake_case
- constants are UPPER_CASE
- indent by 2 spaces
- methods returning boolean values end with `?`
- methods mutating state end with `!`

Chapter 5. Basic syntax

Basic and most explicit syntax looks like

```
puts("hello world!");
```

However the parenthesis are optional in case the only one interpretation of the expression

```
puts "hello world!";
```

and semicolons are optional as well

```
puts "hello world!"
```

String use either quotation marks `"` or apostrophes `'`. Apostrophes does not provide string substitution as quotations marks do.

```
a = 'Hello'       # => "Hello"
b = "#{a} world" # => "Hello world"
b = '#{a} world' # => "\#{a} world"
```

5.1. Operator priority

5.2. Special characters in method names

Ruby allows usage of special characted in method names. The standard is to use

- the question mark `?` for methods that return boolean value
- the exclamation mark `!` for methods that mutate the object

```
1.even?                 # => false
"ruby".upcase.reverse   # => 'YBUR'
"ruby".size.even?       # => true
```

5.3. Comments

In the code above the single line comment is used. It starts with hash `#` and follows to the end of the line. In case of commenting multiple lines, it is customary to comment every line with single line comment.

```
# you should
# do this
# to comment
# multiple lines
```

Chapter 6. Data types

As mentioned above in Ruby everything is an object, including arrays or numbers. However there are special syntax shorthands to create instances of special classes.

6.1. Strings

As mentioned above String are create by quoting the charachters

```
a = "string"
b = 'string'
```

6.2. Numbers

Ruby has two basic number classes `Fixnum` and `Float`.

```
a = 1    # => 1
a.class # => Fixnum

b = 1.1 # => 1.1
b.class # => Float
```

6.3. Empty value

Special value that represents "nothing" is `nil`.

```
a = nil # => nil
```

In a boolean expression, `nil` is considered false, i.e. it’s only of two possible values that are not considered true.

6.4. Booleans

As usual there is either `true` or `false`.

```
a = true    # => true
b = false   # => false
a && b      # => false
a || b      # => true
```

6.5. Arrays

Arrays is an ordered sequence of values. There are no restrictions on what types can be in a single array.

```
["a", 1, true] # => ["a", 1, true]
```

6.6. Hashes

Hash is a structure that maps key to a value.

```
{"a" => true, "b": false} # => {"a"=>true, :b=>false}
```

There are two approaches how to write the mapping, either `rocket` style

```
key => value
```

or `json` style

```
key: value
```

You can use both syntaxes, however with the `json` style the value is converted to `symbol`, so in case you need to use `String` or some other type, or get the name of the key from a variable, you need to use the `rocket` style. Several well-known coding guidelines recommend (and enforce) using `rockets` everywhere.

6.7. Symbols

Symbol is a keyword. It always maps to the same object instance

```
a = "a" # => "a"
b = "a" # => "a"

a.object_id # => 70224766839340
b.object_id # => 70224750415480

a = :a # => :a
b = :a # => :a

a.object_id # => 722268
b.object_id # => 722268
```

Chapter 7. Objects and methods

Methods are called by using the `.`. Operators are actually methods.

```
3 + 3  # => 6
3.+(3) # => 6

[1,2][0]    #=> 1
[1,2].[](0) # => 1
```

Chapter 8. Variables

Ruby has global variables prefixed by `$`.

```
$stdout
```

Classes and object can use class variables, though there are not used very much. Prefer `@instance_variables` in class-level methods, as they have more predicatable behavior.

```
@@class_variables = 1
```

Objects have instance variables.

```
@instance_variable = 1
```

Local variables have no prefix.

```
local_variable = 1
```

And finally constants are all upper case.

```
CONSTANT = 1
```

Chapter 9. Conditions

Everything is considered `true` except `false` and `nil`.

```
a = nil
b = ""

if a
  "we do not get in here"
elsif b
  "we got here"
else
  "we did not get here"
end
```

Ruby has negative variant to `if` called `unless`. Essentially `unless bool_expr` is equivalent to `if !(bool_expr)`. It is used the same way as normal `if`.

```
a = nil
b = ""

unless a
  "we do get in here"
elsif b
  "we did not get here"
else
  "we did not get here"
end
```

Ruby has inline method of using conditionals called modifier statements.

```
puts "Hello" if true
puts "Hello" unless false
```

Ternary operator is available as well.

```
experssion ? 'was evaluated true' : 'was evaluated false'
```

Another way to do conditions is to use `case` statement.

```
case input
  when 'q', 'e'
    quit
  when 'f'
    format
  else
    help
end
```

`Case` statement can as well check on variable class.

```
case var
  when String
    "it's string"
  when Class
    "it's class"
  when Number
    "it's number"
end
```

Another way to use `case` statement is to use it as `if` and `elsif`.

```
case
  when a == "a"
    "a equals a"
  when b == "b"
    "b equals b"
end
```

Chapter 10. Logical operators

There are basic logical expressions

- and `&&`
- or `||`
- not `!`

as well `&&` can be replaced with `and`, `||` can be replaced with `or` and `!` can be replaced with `not`.

There are basic comparison operators

- equal `==`
- not equal `!=`
- lesser then `<`
- greater then `>`
- lesser then or equal `⇐`
- greater then or equal `>=`
- regular expression match `=~`

Chapter 11. Regular expressions

Regular expressions are enclosed with `/`. The simplest way is to use the regexp operator.

```
string = 'localhost:2000'
string =~ /.*:.+/     # 0
string =~ /(.)*:(.)+/ # sets $1 a $2
```

as well there is a `match` method on string.

```
data = string.match(/^(.):(\d+)$/)
data[1] # => localhost
data[2] # => 2000
```

Chapter 12. Loops

`while` repeats as long as the condition is true.

```
while a < b
  a += 1
end
```

To go through the body of the loop at least once

```
begin
  a += 1
end while a < b
```

There is as well inline way to write the loop

```
a += 1 while a < b
```

And finally the negative counterpart `until`

```
until a > b
  a += 1
end
```

Chapter 13. Methods

Methods in Ruby always return some value. If it is not explicitly returned using the `return` keyword, the return value is the value of the last expression in the method. Return as usual returns from method and ends the execution of the method.

```ruby
# Simple method with two arguments
def mth(a, b)
end

# Method with default value for 2nd argument
def mth(a, b=1)
end

# Method accepting any number of arguments, available as Array args
def mth(*args)
end

# Method requiring at least two arguments
def mth(a, b, *args)
end
```

Chapter 14. Reusing code from other files

The `require` method loads code from another file. Ruby keeps track of required files and skips loading files that would be loaded 2nd time. Files are looked up using Ruby's load path, which is represented using an array in `$LOAD_PATH` and `$:`. The `load` method does not keep track of loaded files.

In case the required file ends with [rb, so, o, dll, bundle, jar] extension, the extension may be omitted. There two commands are equivalent

```
require "somefile"
require "somefile.rb"
```

To keep track of required files, Ruby keep list of all files that were required in the `$"` variable.

Chapter 15. Blocks

Blocks have many uses-cases. One of the use cases is the replacement for `for` cycles another use case is `anonnymous functions`. Block are not executed when defined, but have to be called through the `call` method (though the calling of `call` method is most of the times hidden from the develop as in the examples below).

Arrays have method called `each` that accepts block and calls the block for every single element in the array.

```
arr = [1,2,3,4]
arr.each do |el|
  puts el
end
```

will print all four values to the standard output. Blocks can be written in one more way

```
arr = [1,2,3,4]
arr.each { |el| puts el }
```

this variant is usually used for single-line blocks.

Block see their own scope plus can access scope in which were defined.

```
sum = 0
arr = [1,2,3,4]
arr.each { |el| sum += el }
```

Any method can accept a block and call it

```
def mth
  return nil unless block_given?
  yield
end
```

This method will return nil if no block was given or will call the block without any argument and the return value of the block will be return from the method.

Method may also accept blocks as a named argument which is prefixed by `&`.

```
def mth(num, &block)
  block.call(num)
end
```

this method will call block saved in the variable `block` and will pass one argument which is the first argument passed to the method itself.

Chapter 16. Objects

In Ruby everything is an object. Object is an instance of some class. Even every class is an instance of class that inherits from Class. Object can have methods

```
class Hello
  def say
    "Hello, world!"
  end
end

puts Hello.new.say
```

and instance variables

```
class Hello
  def initialize(msg=nil)
    @msg = msg
  end

  def say
    @msg
  end
end

puts Hello.new("Hello, world!").say
```

To make your instance variables accessible from outside, you define them as attributes. Attributes can be either read-only, write-only or both.

```
class Hello
  attr_reader :one         # allows reading by using the .one method
  attr_writer :two         # allows writing by using the .two = "xy" method
  attr_accessor :three     # allows both, reading and writing
end
```

Chapter 17. Inheritance

Ruby allows object inheritance. All methods including constructor are inherited. Methods can be overridden by children. `super` is then used to call the original method.

```ruby
class A
  def a
    "hello"
  end
end

puts A.new.a # => hello

class B < A
end

puts B.new.a # => hello

class C < A
  def a
    super + " world"
  end
end

puts C.new.a # => hello world
```

Chapter 18. Class methods and attributes

As known from other language, except in Ruby class variables are not used because of some pitfalls in their inheritance.

```
class A
  def self.a
    "hello"
  end
end

puts A.a # => hello
```

Chapter 19. Modules

Modules are a way to organize your classes in a similar fashion to namespaces. Classes can be included into modules or into other classes.

```ruby
class A
  class B
  end
end

module Some
  class Thing
  end
end
```

Module are however used as well as mixins. When module is included into class all methods defined for that module are available in the class as instance methods.

```ruby
module Helper
  def something
  end
end

class A
  include Helper
end

A.new.something
```

and when used with `extend` the mehtods are included as class methods

```ruby
module Helper
  def something
  end
end

class A
  extend Helper
end

A.something
```

Ruby has only single inheritance, mixins allow to get around this and provide a way to get some kind of multiple inheritance.

Chapter 20. Method access

By default methods are `public`, explicitly methods can be made `protected` or `private`.

```
class A
  def public_method
  end

  protected

  def protected_method
  end

  private

  def private_method
  end
end
```

Chapter 21. Duck typing

Ruby encourages to react based on behaviour rather then on identity.

```ruby
class Hunter
  def shoot(animal)
    bang! if animal.class == Duck
  end
end
```

in this case the code checks if it’s a duck and shoots it, however

```ruby
class Hunter
  def shoot(animal)
    animal.respond_to?(:quack) && bang!
  end
end
```

in this case we care if the animal quacks and the it’s shot.

Chapter 22. Exceptions

Exceptions represnt a special state in the execution in a program. When an exception is raised, it will bubling thorugh the stack until is caught.

Exceptions are raised using the `raise` keyword

```
raise "This is not expected"
```

On the other hand when an exception needs to be caught, code block is extended with `rescue` statement that is called when an exception is caught and optionally `ensure` that is called after both exceptional and non-exceptional state. Unless an exception class is specified explicitly after the `rescue` keyword, the `StandardError` class and it’s ancestors are rescued.

```
begin
  raise "This is not expected"
rescue => e
  puts e.message
ensure
  puts "always"
end
```

❗ Don’t inherit directly from `Exception` class but use `StandardError` instead. The direct descendants of `Exception` are usually exceptions one doesn’t want to rescue from, such as `SystemExit` or `NoMemoryError`.

Advanced Ruby

Meta-programming, DSLs, etc.

Chapter 23. Return values

In Ruby everything is an expression - it executes and returns some value, the result of it's execution. Let's take a look at a simple example

```
5 + 2         # => 7
"hello"       # => "hello"
"a" if true  # => "a"
"a" if false # => nil

class A; end # => ???
```

What is the result of the last expression? It could be obvious - "we defined class A". But not really. The fact we have defined a class is only an effect of the expression but not the result, result is a value returned from the expression itself.

```
class A; end # => nil
```

The result is simply `nil`. It's similar to the expression `"a" if false` above. The condition is evaluated as false and there is no `else` so there is nothing to return, so the result is `nil`. In this case a class was defined, but there was nothing to be return, as the body of the class was empty, so the result is `nil`. Let's modify the example to return a value

```
class A; 1; end            # => 1
class A; 1; "hello"; end   # => "hello"
class A; self; end         # => A
```

Before it was said that the return value of a method is the result of it's last expression. And it's obvious that in this example it's very similar, and it can be generalized for all structures, even though sometimes it may not be obvious on the first sight.

```
the return value is the the result of the last expression
```

Chapter 24. Context

In Ruby everything is executed in some context. This context is know as `current object` and is always represented by self.

```
self.class # => Object

class B
  self
end
# => Class

class A
  def call
    self
  end
end

A.new.call # => #<A:some number>
```

Chapter 25. Class

25.1. Open classes

Unlike most languages, Ruby classes are open for modifications. Developers can modify behavior of classes defined by frameworks or Ruby itself. This technique is called `Monkey patching`.

```
class Clazz
  def call
    "A"
  end
end

class Clazz
  def call
    "B"
  end
end

Clazz.new.call # => "B"
```

25.2. What is a class?

Let’s start with a definition

```
classes are instances of class Class
```

and as mentioned many times before, everything in Ruby is an object … even a class.

```
class A
  def self.call
    "A"
  end
end

B = Class.new

def B.call
  "called"
end

A.call      # => "called"
B.call      # => "called"

A.object_id # => [some number]
B.object_id # => [some number]

A.class     # => Class
B.class     # => Class
```

Class A was defined using the `class` keyword and then a class method was defined. Class B was created by creating new instance of the `Class` class and the object was assigned to constant B. As both of those classes are objects, it's possible to check it's class and the ID of the object.

25.3. Inheritance

In Ruby classes can inherit from each other, though Ruby has only single-class inheritance - it's not possible to inherit from multiple classes, only from one.

```
class A
 def call
   "called"
 end
end

class B < A
end

C = Class.new(B)

B.new.call # => "called"
C.new.call # => "called"
```

25.4. Mixins

When some class needs to inherit from multiple classes, it's not possible, but Ruby provides a workaround through mixins. It is possible to include many Modules into a class the methods defined in those modules will become part of the lookup path as if they were defined in the class.

```
module Methods
  def call
    "called"
  end
end

class A
  include Methods
end

A.new.call # => "called"
```

25.5. Class introspections

Ruby allows many introspections on classes and many other objects.

There is method `name` defined on a class that returns the name of the current class.

```
Array.name # => "Array"

[].class.name # => "Array"
```

It’s possible to list methods of an object

```
class A
  def call
  end
end

A.new.methods # => array of methods
```

Chapter 26. Methods

As everything else in Ruby even methods are instances of class Method.

26.1. Extracting methods

Sometimes it is useful to pass around only a method instead of the whole object. Ruby allows extraction of a method for later usege.

```
class A
  def call(arg1)
    self
  end
end

meth = A.new.method(:call) # => #<Method: A#call>
```

In the example method `call` from class `A` was "extracted". The method is still bound to the instance of class A and the method will be evaluated in the context of the object (`self` will be the instance). The method can be executed by calling the `call` method with appropriate arguments.

```
meth.call("some string") # => #<A:some_number>
```

26.2. Checking method existence

Because Ruby is a very dynamic language, it’s not possible to know in advance what kind of arguments will be received. In most cases the developer should not care what class the argument is, but whether the argument responds to a method.

```
Do not care what the object is, only care whether it behaves as expected.
```

This technique is called Duck typing.

```
class A
  def call
  end
end

a = A.new

a.respond_to?(:call) # => true
a.respond_to?(:wtf)  # => false
```

26.3. Dynamic method calling

Let’s define a class with a method, create an instance and call the method.

```ruby
class A
  def call
  end
end

A.new.call
```

The method is called, but the develeoper had to know the name of the method beforehand … in the time the code is written. What if the method name is not known and there has to be some method called. Do not be surprised, this is very common use-case in Ruby.

```ruby
class A
  def call(arg1)
  end
end

a = A.new
a.call("some string")
a.send(:call, "some string")
```

Well, not so identical. When you use the send method on an object, you effectively bypass the access modifiers.

Ruby has three access levels `public` is default, `protected` and `private`.

```
class A

  def public_method
  end

  protected

  def protected_method
  end

  private

  def private_method
  end

end

a = A.new

a.public_method     # => nil
a.protected_method  # => NoMethodError: protected method `protected_method' called ...
a.private_method    # => NoMethodError: private method `private_method' called ...

a.send(:public_method)      # => nil
a.send(:protected_method)   # => nil
a.send(:private_method)     # => nil
```

26.4. Defining methods programmatically

The way to define methods using the `def` keyword shown before is not the only one. It’s also possible to define method in a more dynamic way. It makes sense. We can inspect methods of an object, we can extract methods of an object and also call methods of an object in a dynamic way. To dynamically define a method use the `define_method` method of a class, however

```
Class.define_method is private
```

To get around this obstacle, it’s possible to use the `send` method and bypass the access modifier.

```
class A
end

a = A.new

logic = Proc.new do
  "data"
end

A.send(:define_method, :some_method_name, logic)

a.some_method_name # => "data"
```

26.5. Missing methods

Every object can define special `method_missing` method that is called whenever there is a call to undefined method on that object.

```
class A
  def method_missing(name, *args, &block)
    puts "method #{name} called with args #{args.inspect}"
  end
end

A.new.something("a") # => method something called with args ["a"]
```

Chapter 27. Objects

Objects complement classes in a way that

```
objects define state and classes define behavior
```

Behavior id defined as a class, then an object is created for that class to hold the state. Every object has to be of some class.

27.1. Creating new object

To create an object of a class there is the `new` method on respective class.

```
class Dog
end

dog = Dog.new
```

27.2. Defining methods

In the example above many methods were defined in simple or more fancy styles. But let's get back to the core and try to define a method

```
class A
  def call
  end
end
```

here we use `def` keyword to define method `call`. Where will def define the method? The answer is simple and complex

```
def defines methods into the nearest class
```

So in the previous example the nearest class is A. That is obvious from next example where the current context is returned and inspected

```
var = class A; self; end

var.class  # => Class
var.name   # => "A"
```

OK, so the the current context is a Class and thus is't obvious that the nearest class is this class. Now let’s try to define a class method

```ruby
class A
  def self.call
    "string"
  end
end
```

Where will Ruby define the method now?? It is a bit more complicated. To understand this, we have to explain something else first.

27.3. Eigenclass

To understand how Ruby works, we have to understand what `eigenclasses` are. Let’s start with simple definition

```
every object in Ruby has it's own eigenclass => an instance of Class
```

eigen means "it’s own" in German

Why is this important? Because, however the `eigenclasses` are basically invisible to developers, they take an important part in method lookups.

When Ruby is trying to look up a method, it follows a basic chain (will be described a bit later). Important is, that before the class the object is linked to, there is the one more class - object’s eigenclass. Every single object in Ruby has it’s own `eigenclass` and because Classes are object as well, `eigenclasses` has their own `eigenclasses` as well.

```
The closest class to an object is not it's class but it's eigenclass.
```

Back to the example we were talking about

```ruby
class A
  def self.call
    "string"
  end
end
```

to see it more clearly we can rewrite this example identically as

```
class A
end

def A.call
  "string"
end
```

these two expressions are identical. To understand why it is important to understand this

```
class A
end

scope = class A
  self
end

A == scope # => true
```

but back to the original question … where is the method going to be defined? In the context of the instance of the class A. The important part is the **instance of**. What is the closest class to an instance (object)? As stated above it’s its eigenclass. Now you might have guessed that from implementation point of view

```
there are no class methods in Ruby
```

What would be called a class method is only an instance method defined on the eigenclass associated with object representing the class.

So eigenclass is some stealth object that we can not see? Not really. Ruby has ways to access eigenclasses

```
eigenclass = class << some_object
  self
end

eigenclass = some_object.singleton_class
```

now that we can access eigenclasses, let’s see how we could define "class methods" (instance methods in the eigenclass).

```
class A
  def self.call
    "called"
  end
end

class B
  class << self
    def call
      "called"
    end
  end
end

D = Class.new
class << D
  def call
    "called"
  end
end
```

all those examples are identical.

27.4. Method lookups

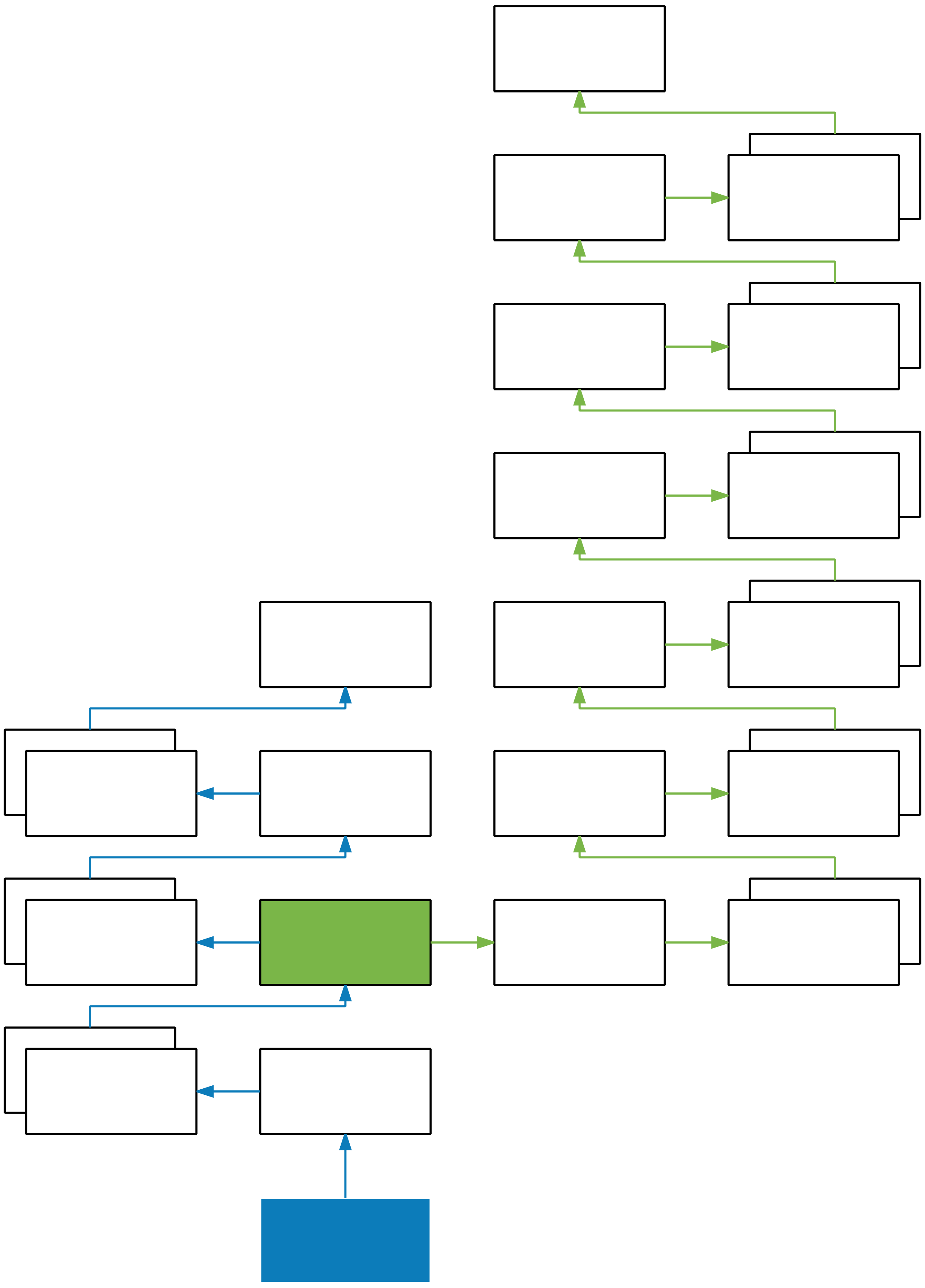
Now that you know where and how are methods defined, lets see how methods are looked up. Let's see how the class hierarchy looks for class

```
SomeClass -> Class -> Module -> Object -> BasicObject
```

and for objects

```
object -> SomeClass -> Object -> BasicObject
```

though in real it is a bit more complex as seen in this picture



```
Eigenclasses are not visible as classes of objects.
```

```
o1 = Object.new

def o1.meth
  "string"
end

o1.meth  # => "string"
o1.class # => Object

o2 = Object.new

o2.meth  # => undefined method `meth`
o2.class # => Object
```

This example shows that having two instances of same objects. Both can behave differently. Because in the case of o1 the method is stored in the eigenclass, that is not accessible by o2.

```
Eigenclasses are used when a specific behavior of an object is expected.
```

Testing

Writing automated tests for the code is essential. Ruby community emphasizes it and most of projects are well covered. A TDD is also popular among rubyist.

Chapter 28. Testing frameworks

Today de facto standard is to use Minitest testing framework. You can see RSpec being still used too but Minitest already offers the capabilities and more. Both can be easily used for TDD. For BDD there’s popular ecosystem called Cucumber which started as Ruby gem but quickly evolved into polyglot tool.

Chapter 29. Minitest

A syntax you can see in tests can be in two forms. Either something we call testunit (aka junit) and spec that was taken from Rspec. The internal implementaion is the same for both and it's mostly matter of taste. However tests are regular ruby scripts that test other scripts. Test are usually to be found at `test/` directory, file name should reflect test class defined inside, e.g. morse_coder_test.rb.

Example output of

29.1. Testunit syntax

```ruby
require 'minitest/autorun'
require 'morse_coder.rb'

class MorseCoderTest < Minitest::Test
  def setup
    @coder = MorseCoder.new(...)
  end

  def test_encode_for_single_letters
    assert_equal ".-", @coder.encode "a"
    assert_equal "-...", @coder.encode "b"
  end
end
```

Test class should inherit from Minitest::Test so test helpers (assertions) are available. Testing methods must start with "test_". Other methods are regular methods that can be used from testing methods, e.g. to build some fixtures.

There are few method names with special meaning. In the example you can see method with name `setup`. This method gets automatically executed before every testing method. Similarly there's `teardown` method that get's executed after each testing method. It's usually used for cleaning up mess the testing method created.

29.2. Assertions

One testing method can contain more than one assertion. First assertion failure stops the method run and mark the test as failure (F). If method raises an exception the result of test is marked as error (E). If all assertions defined in method passes, test succeeds (.). If you plan to implement the test later you can skip the test by calling `skip("Sorry, I'm lazy")`.

The simplest assertion is to test boolean value like this

```ruby
assert @something
```

This will succeed if @something is considered true, fail otherwise. The negative form is refute, e.g.

following would pass.

```
refute false
```

You could obvisouly add tests like `assert @something == 'that I expect'` but it would generate very generic messages on failures. You can specify custome message by passing extra argument like this

```
assert @something == 'that I expect', '@something does not match expected string'
```

but it’s always better to use assert helper that matches the use-case best. Following example demonstrates how to check equality of two values, the failure message would automatically include information about what’s it @something and what was expected it to be.

```
assert_equal @something, 'that I expect'
```

Useful assert helpers are listed in example below. All of them can be found in Minitest documentation.

```
assert arg            # arg is true
refute arg            # arg is false

assert_equal          expected, actual
assert_includes       collection, object
assert_kind_of        class, object
assert_nil            object
assert_match          expected, actual
assert_raises         exception_class &block
```

29.3. Spec syntax

Subjectively better structured, less repeating, more readable and TDD supporting syntax can be used. See the following example.

```
require 'minitest/autorun'
require 'morse_coder.rb'

describe MorseCoder do
  let(:coder) { MorseCoder.new(...) }

  describe 'single letters encoding' do
    let(:a) { coder.a }
    let(:b) { coder.b }

    specify { a.must_equal '.-' }
    specify { b.must_equal '-...' }
  end
end
```

Describe block wraps logical block. Each such block can have it's own `before` (aka setup). With `let` we define method that can be called later within any nested block. Let is lazy. `specify` accepts a lock that uses assertion helpers in form of `must_$assert` or `wont_$assert`. There are many other extensions to this language so it reads more naturally.

Note that since the implementation is the same, you can combine both at the same time.

29.4. Output of test run

```
Run options: --seed 25127

# Running tests:

..S.....F........

Finished tests in 101.524752s, 6.4319 tests/s, 9.0618 assertions/s
63 tests, 92 assertions, 1 failures, 0 errors, S skips

  1) Failure:
TestConnector#test_connection [./connector.rb:5]:
  Expected: nil
  Actual: "that I expect"
```

The seed is random number representing the order of test. Note that your tests should be order indpenendant.

29.5. Running multiple test files

It's common to have more than just one test file in project. To run all tests at once we can use Rake. Usually tests are put in `test` directory in the project tree structure. In such setup we can easily define test task in Rakefile. Rake provides built-in class for this, we just need to configure it. Just put following into your Rakefile.

```
require 'rake/testtask'

Rake::TestTask.new do |t|
  t.libs << 'test'
  t.test_files = Dir.glob('test/**/*_test.rb')
  t.verbose true
end
```

we can run `rake test` which will load a run all ruby scripts with `_test` suffix in the test directory including all of its subdirectories. If you prefer test to be the default rake task, add following to the Rakefile

```
task :default => [ :test ]
```

now you can run all tests just by running `rake`.

Another common practise is to have one file that is loaded at start, usually named test_helper.rb. This file contains everything that is needed for all tests, like requiring additional testing libraries. You can also put `require minitest/autorun` there. Just note that you need to `require 'test_helper'` as first line of every test file.

29.6. Test coverage

To get a good overview of what needs test coverage it's useful to setup code coverage check. A simplecov gem can generate html report. Just put following on top of you test_helper.rb

```
require 'simplecov'
SimpleCov.start
```

you can also define a minimum coverage in percents

```
SimpleCov.minimum_coverage 95
```

Now when you run your test suite, new directory called `coverage` will be created. See `coverage/index.html` for details how well your code is covered with tests.

29.7. Stubbing

Sometimes we don't want to call all method chain when we test just single method behavior. This applies especially in unit testing where we test just small piece of code. Since Ruby is dynamic language, it's easy to cut off some methods. This is called stubbing (leaving stubs).

Let's look at following example

```
class TemperatureMeter
  def measure(output)
    temp = rand(21) + 20
    output.puts temp
    temp
  end
end
```

The test covering this should call method measure and verify it returns reasonable temperature. We don't want our test to print anything to STDOUT. We can stub out puts method easily like this

```
def test_measure
  meter = TemperatureMeter.new
  STDOUT.stub(:puts, nil) do
    result = meter.measure(STDOUT)
    assert_kind_of Fixnum, result
    assert_includes 20..40, result
  end
end
```

With this stubbing, `puts` method is replaced by new empty method that returns the second argument, in this case `nil`. The stub is applied only within the stub block.

29.8. Mocking

Mocking is related to stubbing. Imagine we wanted to check that measure method really called puts on output object. The method is written in a way that it accepts custom output object, which makes testing easy. We can simply pass any object that implements method `puts`, e.g. file handler, socket or our own testing object. Or we can use mocks. Mock is a blank object on which we can define expectations.

For example we can create a mock instance and specify that its method puts should be called exactly once during the test.

```
def test_measure_print_the_value
  meter = TemperatureMeter.new
  mock = Minitest::Mock.new
  mock.expect(:puts, nil, [20..40])
  result = meter.measure(mock)
  mock.verify
end
```

First `expect` argument is the name of method to be called, second is the return value and third is the array containing arguments which the puts should be called.

You could also stub the `rand` method to return let's say `0` and then setup expectation that mock's `puts` method will receive 20 as a parameter to print. But the range also works so the mock accepts any value between 20 and 40.

You have to call verify on mock so it runs assertions on how many times the expected method was called. To expect another call of puts, just define new expectation with .expect.

29.9. Stubbing network calls

If your app communicates with external services over HTTP you most likely need to fake the communication in your test suite. Reasons include performance, spamming of remote services, avoiding credentials leaks, error state testing. Constructing the whole net/http response object can be complicated. Luckily there are tools that can help you greatly.

First is webmock gem. It provides helpers to stub low-level methods easily. To use it, install the gem and just add following to your tests.

```
require 'webmock/minitest'
stub_request(:get, 'www.example.com')

Net::HTTP.get('www.example.com', '/') # this will succeed
```

You can also specify more conditions to match the request as well as return value

```
stub_request(:post, 'www.example.com').with(:body => 'ping').to_return(:body =>
'pong')
```

Custome headers can be added too. Webmock works with higl level libraries such as popular Restclient gem.

Another useful tool is vcr. The name was chosen because of analogy with videocassette recorder. It can record a real network communication and replay it later. This can be nicely used in tests. You only record the communication once during writing tests and replay it while running tests in future or on CI server. You can have multiple communications recorded and just swap cassetes for each test. Example follows

```
require 'vcr'

VCR.configure do |config|
  config.cassette_library_dir = "fixtures/vcr_cassettes"   # storage for cassetes
  config.hook_into :webmock                                # webmock integration
end

class VCRTest < Minitest::Test
  def test_example_dot_com
    VCR.use_cassette("success_info") do
      response = Net::HTTP.get_response(URI('http://www.example.com/'))
      assert_match /Example Domain/, response.body
    end
  end
end
```

29.10. Testing web applications

If you work on web app you can also easily test the interaction like users will interact through web browser. This is useful when you write integration tests. A de facto standard is capybara gem that provides drivers for various browser backends. The simplest to setup driver is RackTest, so you can start with it as long as your app uses rack.

If you need advanced stuff like testing pages with asynchronous requests through AJAX you can use Selenium driver which runs firefox in headless mode. If you want to run such tests on CI server without X11 server, there’s Poltergeist driver using PhantomJS.

An example of simple test, supposing my_app.rb contains rack based app (e.g. using Sinatra).

```
require 'minitest/autorun'
require 'capybara/dsl'
require './my_app.rb'

Capybara.app = MyApp
Capybara.default_driver = :rack_test

class MyAppTest < Minitest::Test
  include Capybara::DSL

  def test_index
    visit '/'
    click_link 'login'
    fill_in('Login', with: 'Marek')
    fill_in('Password', with: 'secret')
    click_button('Submit')

    assert page.has_selector('div p.success')
    assert page.has_content?('Welcome Marek')
  end

  def teardown
    Capybara.reset_sessions!
    Capybara.use_default_driver
  end
end
```

29.11. Cucumber

We can use Cucumber framework for BDD aproach. It allows us to write the behavior specification in natural language first and then convert it to test step by step. Imagine you'd describe a feature like this

```
Feature: logout of logged in user

  Scenario: User can log out from app
    Given I'm logged in as user ares
      And I'm on host list page
    When I click logout link
    Then I should see logout notification
```

It's a valid cucumber test (aka feature) which only needs implementing those steps, usign capybara for example.

```
Given(/^I'm logged in as user (.*)$/) do |user|
  visit '/'
  fill_in "login", with: user
  fill_in "password", with: 'testpassword'
  click_button 'login'
end

Given(/^I'm on (.*) (.*) page$/) do |resource, action|
  visit "/#{resource}/#{action}"
end

When(/^I click (.*) link$/) do |identifier|
  click_link identifier
end

Then(/^I should see logout notification$/) do
  assert page.has_content 'div p.logout_notification'
end
```

One advantage that it brigns is, that your tests are live documentation too.

Standard library & libraries

Get the know all the awesome stuff included in Ruby and in ecosystem

Chapter 30. Files

Ruby on Rails

Develop web applications with ease.

Chapter 31. Introduction